The DAMSON Runtime Notes (Version 1.1)

Last Edited: Paul Richmond (09/02/15)

Table of Contents

Introduction

DAMSONlib is a runtime environment for the SpiNNaker hardware architecture which is designed to provide a multi-threaded environment for executing DAMSON code.

Requirements and Usage

DAMSONlib has a number of header files including damsonrt.h which has a number of common definitions and is shared by the loader. It is important that the same damsonrt.h header file is used by both the loader and DAMSONlib otherwise unpredictable behaviour or failure will result.

DAMSONlib is not used directly by DAMSON users. It is instead required by the dmake makefile produced by the DAMSON code generator. The dmake file will link damsonlib.o with any DAMSON source modules to produce a single aplx file for use within the loader.

Overview of Source Modules

The DAMSONlib runtime consists of the following source modules;

- cdamsonrt.c - A C module containing any damsonlib C functions. Any procedures/functions used externally are prefixed with 'c_' e.g. c_reschedule();
- damsonrt.gas - A GNU ARM assembly module containing any assembly damsonlib procedures or functions.
- sark.c - A (mostly) unmodified version of the SARK C module for compatibility with SCAMP.
- sark_init.gas - A unmodified version of the sark assembly module for compatibility with SCAMP.

- spinn_sync.c - A C source module containing code for providing node level synchronisation. This has been re-factored from the SpiNNaker C API.

DAMSONlib is provided with a single makefile gnu.make which will build the damsonlib.o object in the 'lib' directory. Usage of the makefile is as follows;

make -f gnu..make

The damsonlib.o object can be removed with any temporary files using the following command;

make -f gnu.make clean

Initialisation and Entry

Before execution is started on any given core the DAMSON loader will have initialised all aspects of memory excluding the SpiNNaker DTCM and ITCM (see Figure 1). The code and C runtime data parts of the packaged DAMSON module and runtime system will have been loaded as an APLX binary (by the loader) to DTCM at address 0x403800. As part of the loaded binary an APLX command table will have been embedded at the start of the aplx module. Various values within this table will have been filled at linker time by dmake and the special GNU linker script file (example.lnk). The APLX table has the following format;

```
aplx_table:     .word   APLX_RCOPY
                .word   RO_TO
                .word   RO_FROM
                .word   RO_LENGTH

                .word   APLX_RCOPY
                .word   RW_TO
                .word   RW_FROM
                .word   RW_LENGTH

                .word   APLX_FILL
                .word   ZI_TO
                .word   ZI_LENGTH
                .word   0

                .word   APLX_EXEC
                .word   RO_TO
                .word   0
                .word   0
```

Each 4 word block contains an instruction and three parameters. The first of these two (APLX_RCOPY) instructions notify SCAMP that a given LENGTH of data FROM a specified address should be copied TO a specified address. In the first case, the instruction will copy Read Only data (i.e. program instructions) from the DAMSON object, RO_DATA, .text and .rodata parts of damsonlib to RO_TO which is the beginning of ITCM (i.e. 0x0). The second instruction will copy and Read Write parts of damsonlib (i.e. the .data part) to the 4kb of SpiNNaker runtime reserved space at RW_TO which is the start of DTCM (i.e. 0x400000). The third instruction fills any Zero Initialised data to 0 and the final instructions causes program execution to branch to RO_TO.

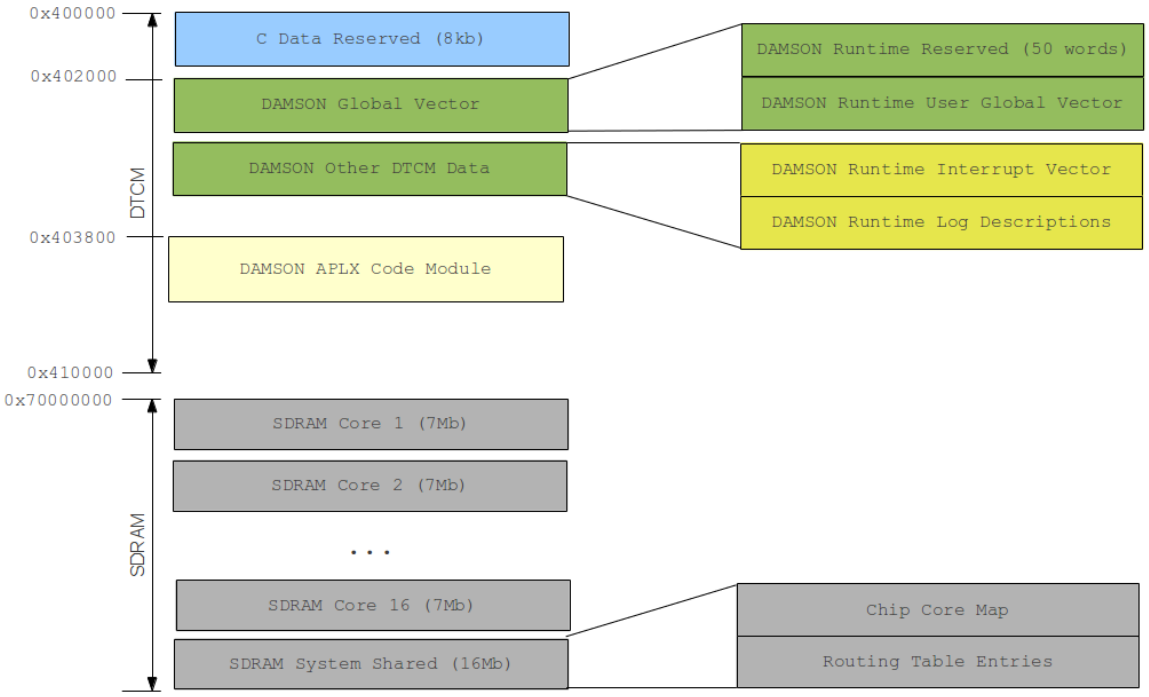

Figure 1: SpiNNaker memory layout after loading stage.

Processing of the APLX commands is triggered by the DAMSON loader when cores are started. The initial entry point at the start of ITCM is the SARK environment. This then loads the damsonlib library entry function c_main from the cdamsonrt.c module.

After entry to c_main, the initial APLX program at 0x403800 will have been copied to the appropriate place and the memory is able to be recycled as stack space. The damson runtime has both an initial single threaded mode which uses 4kb of reserved C runtime stack located at the end of DTCM (starting at address 0x40F000, see Figure 2). During this single threaded mode the multithreaded environment is configured and a number of other initialisation tasks also take place. The first of these is to detect if the current CPU core should be identified as a 'root' core. Root cores are identified as having a virtual CPU number of 1 (see mapper/loader documentation for details) and are required to perform initialisation of the routing tables and perform additional functionality during node level synchronisation. The following initialisation steps are then undertaken by all cores;

1. sync_init - Initialisation of the synchronisation primitive. The core_map at the start of shared SDRAM is used by each core.
2. init_interrupt_vector - Pointers to the interrupt vector hash structure, its size and a pointer to the clock interrupt vector item are initialised by getting the interrupt vector location and size from the DAMSON reserved globals 6 and 5 respectively.
3. init_dma_controller - Initialises the DMA controller.
4. init_stack - The DAMSON multi-threaded stack spaces are initialised according the the details in the section: Multi-Threading.
5. init_damsonlib - The assembly environment is initialised by copy the runtime address of system functions into the appropriate DAMSON reserved global vector locations. The FIQ SP register is also set to the interrupt stack space address ready to handle initial

synchronisation packets.

6. init_interrupts - Interrupts are initialised by selecting appropriate interrupt handlers for for the Vector Interrupt Controller (VIC). See section: Interrupts for more details.
7. init_logging - Logging is initialised by setting log interval counts of any runtime log items to the log interval (to be reduced by the tickrate on each clock event). A number of log buffers are also initialised by setting the buffer status to free (i.e. not occupied by data waiting to be written via DMA).

Following this initialisation stage a new processes is created for the DAMSON user main function (renamed _mainprog by the DAMSON compiler). The DAMSON environment then synchronises across all active cores, enables the timer and reschedules causing the multi-threaded environment to start executing the users DAMSON main function (i.e. the only active DAMSON process). Once the multi-threaded environment is entered the single threaded stack is discarded (see Figure 2).

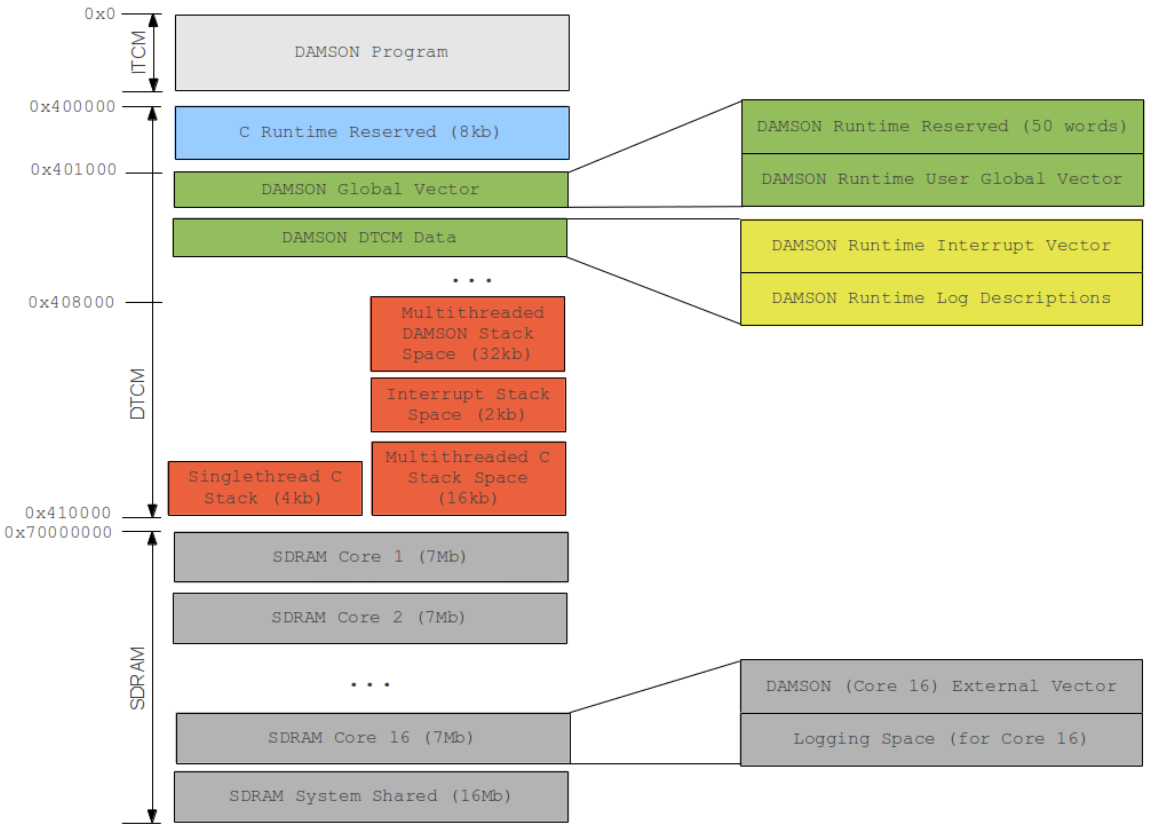

Figure 2: he SpiNNaker memory map at runtime. Showing both single threaded and multi-threaded stack layouts.

Multi-Threading

DAMSON runtime multi-threading uses a simple statically declared array (the processes list) of the following C structure to store the state of a process.

typedef struct

{

```
    uint            *saved_FP;
    uint            *saved_SP;
```

```
    uint            *saved_LR;
    uint            *saved_PC;
    uint            *saved_PSR;
    uint            saved_r0;
    uint            saved_r1;
    uint            saved_r2;
    uint            saved_r3;
    uint            saved_r4;
    uint            saved_r5;
    uint            saved_r6;
    uint            saved_r7;
    uint            saved_r8;
    uint            saved_r9;
    uint            saved_r10;
    uint            saved_r12;
    uint            *stack;
    uint            *c_stack;
    uint            handle;

    uint            status;

    DMAWaitItem     dma;

    int             delay_us;

} PCB;
```

The PCB structure stores a pointer to the DAMSON stack and the C environment c_stack. These stacks are kept separate as the DAMSON environment does not use the SP register (see 'An overview of the DAMSON compiler for the ARM processor' for details of the frame format) and each runtime environment uses a different stack direction (DAMSON is ascending where as the C environment is descending). During initialisation of the stack (init_stack) both the DAMSON and C runtime stack stack spaces are partitioned equally and allocated to each of the PCB items in the process list. The PCB structure also stores a handle value (the process slot number), a status, a DMAWaitItem and a delay_us value in microseconds. A DMAWaitItem consists of the transfer details for any queued SDRAM transfer and the delay_us is used to store any delay value in microseconds for the PCB if it is being delayed. The status value may be any one of the following values;

PCB_STATUS_FREE - The PCB is unused and may be selected by create process.
PCB_STATUS_RUNNING - The PCB is active and is either executing or ready to be restored.
PCB_STATUS_DELAYING - The PCB is being delayed by a call to delay.
PCB_STATUS_DMAWAITING - The PCB is waiting for access to the DMA controller to either read or write.
PCB_STATUS_DMATRANFER - The PCB is currently waiting for a DMA transfer to complete.

Create Process

In order to create a new processes a free PCB slot is found by linear search of the process list. The status variable is used to indicate a free slot if its value is PCB_STATUS_FREE. When a new process is created a new DAMSON stack frame is added to the stack to ensure that when a process

completes it calls the end_process function. This is shown below where p_list is the process list and i is the index of a free PCB slot;

```
p_list[i].stack[0] = 0;
p_list[i].stack[1] = 0;
p_list[i].stack[2] = (uint)end_process;
p_list[i].stack[3] = i;
```

The first two items represent the FP and LR of frame, the third item is the PC and the fourth item is an argument i.e. the PCB index. As the end_process function represents the end of the DAMSON environment the FP and LR are not required and are hence 0. The PCB is then configured for the new process by setting the saved registers so that it is ready to be restored; e.g.

```
p_list[i].saved_FP = p_list[i].stack;                  //fp is start of end_process frame
p_list[i].saved_SP = p_list[i].c_stack;                //reset the C stack pointer
p_list[i].saved_LR = (uint*)end_process;               //lr is end_process
p_list[i].saved_PC = pc;                               //pc is pc of new processes
p_list[i].saved_PSR = (uint*)0x1f;                     //no flags set for psr
p_list[i].saved_r0 = arg0;                             //argument on frame
p_list[i].saved_r1 = arg1;                             //argument on frame
p_list[i].saved_r2 = arg2;                             //argument on frame
p_list[i].saved_r3 = arg3;                             //argument on frame
p_list[i].saved_r4 = (uint)(p_list[i].saved_FP + 4);   //r4 is temp frame pointer (offset by size of
end_process frame)
p_list[i].saved_r5 = 0;                                        //r5 is empty
p_list[i].saved_r6 = 0;                                        //r6 is empty
p_list[i].saved_r7 = 0;                                        //r7 is empty
p_list[i].saved_r8 = 0;                                        //r8 is empty
p_list[i].saved_r9 = 0;                                        //r9 is empty
p_list[i].saved_r10 = DAMSONRT_DTCM_START;             //gv
p_list[i].saved_r12 = (uint)&p_list[i];                //r12 is pointer to this pcb
p_list[i].status = PCB_STATUS_RUNNING;
```

End Process

The end_process assembly function is implemented as follows;

end_process:

```
msr     cpsr_c, #0xdf
ldr     r0, [fp, #12]           @ load the process handle from the frame

b       c_delete_process        @ delete process r0=handle
```

The MSR instruction changes the CPU mode to system mode with FIQ and IRQ interrupts disabled. The PCB handle is loaded into register r0 and execution is branched to the C function c_delete_process. This C function frees the PCB slot by setting the saved_FP to 0x0 and then calls reschedule.

Delay

The assembly delay system function operates by saving the registers states into the CURRENT_PCB and then calling a C handler function which updates the PCB status to PCB_STATUS_DELAYING and sets a delay_us value. The delay_us value represents a value in microseconds which is obtained from the fixed point seconds value using the following expression;

int us = fp_seconds*(1000000>>16);

Each clock tick then reduces the delay_us value by the clockrate in microseconds until the value is less than zero.

DMA Transfers

The assembly writesdram and readsdram functions perform the same state saving operations as the delay function before setting a direction flag (read=0, write=1) and calling a common C handler function. The C handler function tests to see if the DMA transfer can be immediately performed, if so the DMA use status global is set to indicate a DAMSON SDRAM transfer is taking place and the PCB handle is also stored in a dma_handle global. This information is required to update the PCB once the transfer has completed (see DMA Transfer Compete Interrupt). If the DMA controller is busy then the DMA request is stored in a DMAWaitItem (which hold the DTCM address, SDRAM address and transfer description) within the PCB and the status is set to waiting. The DMA operation will then be executed by reschedule at the next available opportunity.

Logging

Logging consists of both clock driven logs and snapshots which log calls to the sendpkt system function. In each case Log items are stored sequentially from the first address in SDRAM after the user external vectors. The log functions performs a check to ensure that a log entry will not exceed the fixed block of SDRAM space available for the core.

A log entry has a handle, a size (the a number of log items) and a set of global variable values for each argument in the final formatted log string. The handle is used to determine which formatted string and hence which log file the log entry belongs too. The runtime system does not care about the details of the formatting as file output is
handled by the loader after execution is complete by retrieving all log entries.

Logging can be performed using either DMA or via direct write access to SDRAM. Direct access will be very slow but might be useful for debugging.
Logging via the DMA controller uses a set of log buffers to store a log entry locally. When a log buffer is filled it will start the DMA transfer immediately if possible, setting the DMA use status global to indicate a log write is taking place and the dma_handle to the log buffer index. If it is not possible to start the transfer, either because another core is using the DMA controller or if the runtime is waiting to receive a DMA completion, then the log buffer status will be set to indicate that it is waiting and a DMAWaitItem is set within the buffer to store the transfer details until the DMA controller becomes free. The DMA completion interrupt is then responsible for starting any waiting log writes.

Reschedule and Restore Process

The reschedule function iterates the list of PCBs and selects the running thread to be restored. If a DMA waiting thread is found and the DMA controller is free then the DMA request is started (without restoring the thread). If there has been an advance in the number of clock ticks since the last reschedule then any delayed PCBs are updated to reduce the delay_us value by the clock ticks in microseconds. If the delay has completed then the PCB status is updated to PCB_STATUS_RUNNING.

Once the last running slot is found the process is restarted (or started for the first time) by

calling the restore_process assembly function with the PCB address as the argument. If reschedule is called and there are no active process slots then the processor enters sleep mode. An interrupt will cause the processor to wake and resume normal execution.

By the end of the restore_process assembly function a DAMSON process should be restored with the stacks of each of the interrupt processor modes (IRQ and FIQ) reset with interrupts enabled. This is because any any code executed up to restore_processes will not have naturally unwound the C interrupt stack, it will simply find and restore the first active process. It makes sense to perform this operation here rather than at the start of the interrupt function as at the start of the interrupt function no registers are free to load the SP address. To ensure that the stacks of the interrupt modes are reset the following code is executed during restore_process;

```
msr     cpsr_c, #0xd1
ldr     sp, =I_STACK_END @ reset the sp to end of interrupt stack (as it is a descending stack)
ldr     sp, [sp]
```

The above code changes the processor mode to FIQ and then loads a global variable set by the C initialisation code with the end address (as it is a descending stack) of the interrupt vector stack. The same instructions are repeated to reset the SP of IRQ mode.

Once the processor mode stack pointers have been reset the processor can be returned to system mode and the registers restored by first loading the PSR value of the PCB into r1 and then the restoring the CPSR flags, status and extension bits (not the control mode). e.g.

```
ldr     r1, [r0, #16]
msr     cpsr_fsx, r1
```

The FP, SP and LR can be restored in a single load multiple instruction;

```
ldmia   r0, {fp, sp, lr}
```

The PC of the PCB is saved into r1 and then pushed onto the C stack. The address of the PCB structure is offset by 5 words to point to the saved_r0 value and then registers r0 through to r12 are restored in a single in single load multiple instruction. Interrupts are then enabled and the PC is popped off the stack to complete the restore process operation.

```
ldr     r1, [r0, #12]
push    {r1}
add     r0, #20
ldmia   r0, {r0-r10, r12}
msr     cpsr_c, #0x1f
```

```
pop     {pc}
```

Interrupts

SpiNNaker interrupts are handled by configuring the Vector Interrupt Controller (VIC). The DAMSON runtime system configures the VIC (init_interrupts) to set packet interrupts to be handled in fast interrupt mode (FIQ) and timer and DMA transfer completion interrupts to be handled in regular interrupt mode (IRQ). Both interrupt modes have banked SP, LR and saved program status registers, the FIQ mode also has banked registers r8 through to r12. A unique assembly interrupt handler routine is provided for both timer, DMA transfer completions and packet interrupts in the damsonrt.gas assembly module.

Timer Interrupts

When a timer interrupt occurs the DAMSON runtime may take one of two paths. If there is no processes currently running then the timer event is handled by creating a new process with a PC retrieved from looking up the clock interrupt vector item. The PC is offset by DAMSON_OBJ_BASE which is set by the linker script and translates the relative PC address in the DAMSON module to the final address within the complete aplx binary. If a process is being executed when the timer interrupt arrives the this must be saved to the current PCB (so that it can be later restored) before the interrupt is handled in the same way as above. In order to test if there is an active process the CURRENT_PCB pointer is loaded and queried as follows (r12 is first saved by pushing onto the interrupt stack);

```
push    {r12}
ldr     r12, =CURRENT_PCB
ldr     r12, [r12]

cmp     r12, #0
```

Registers can then be saved using a series of store instructions which also requires switching processor modes to save the system SP, LR and PSR.

Packet Interrupts

Packet Interrupts differ slightly from timer interrupts as there are two types of MC packet, either a sync packet with a special key value (greater than 0x1ffff << 11) or a DAMSON packet with a node id in the range of 1 1048567. It is therefore required that the packet handling function performs a check to determine the type of MC packet which is received; This is completed by loading the RX key value from the MC packet and comparing it with the value 0x1ffff (or SYNC_PKT);

```
mov     r8, #CC_BASE
ldr     r9, [r8, #CC_RXKEY]
ldr     r8, =SYNC_PKT
cmp     r9, r8
```

Sync packets are handled by saving registers r0-r3 and LR, moving the RX key value to register r0 and then calling the sync_handle_packet function. The saved registers are then restored and the interrupt discarded by returning to what ever it was doing previously through the following instruction which restores the PC and restores the previous operating mode.

```
subs    pc, lr, #4
```

For normal DAMOSN packets handling is performed in the same way as for handling the timer; that is the CURRENT_PCB is queried to see if there is an active process to be saved and then a new processes is created by looking up an interrupt vector item from the hash table using the packets source node number as the hash key.

DMA Transfer Complete Interrupts

A DMA Transfer complete assembly interrupt function is exactly the same as the timer interrupt case with the exception that the c_handle_sdcomplete function is called once the processing state has been saved. A DMA completion may be the result of either a Damson read/write or a log buffer write. Depending on which the stored dma
handle is used to change either the Damson PCB thread state to 'running' or the log buffer state to 'free'. After performing
this operation the DMA use status is set to 'free'. This indicates that the DMA completion has been handled and will allow either a new log buffer to be written or a DAMSON thread waiting for DMA access to start transferring (in that order). The test for any waiting log buffers is performed by the DMA completion handler where as the test for any waiting DAMSON PCBs is performed by reschedule (to avoid iterating over the PCBs twice).