SARK - SpiNNaker Application Runtime Kernel

SpiNNaker Group, School of Computer Science, University of Manchester
Steve Temple - 8 Mar 2016 - Version 2.0.0

Introduction

SARK is the lowest level of software which runs on a SpiNNaker core (CPU). It is linked together with the application code and performs three main functions.

- It initialises the ARM core, setting up stacks and setting various control registers in the core and some of its peripherals. It then calls the main procedure of the application to start the application running.
- It provides a library of low-level functions for the application which perform operations such as memory management, interrupt control, etc.
- It provides a mechanism for the monitor processor to communicate with the application core, mostly using SDP packets. This allows the application's memory to be read and written from a host, provides a basis for simple I/O functions and allows the application to be controlled from the host.

Application Loading and Start-up

A SpiNNaker application is a program which runs on a SpiNNaker core and has sole use of that core. The same application may be loaded into many cores. It will have been built with a cross-compiler, linked with various libraries, including SARK, and converted into a format known as an APLX file ready to be loaded onto a SpiNNaker core. Typically, the APLX file is loaded into an area of shared memory on the SpiNNaker chip by the monitor processor which then causes the relevant application cores to load and start the application. The loading of the application results in the executable part of the APLX file being copied to the bottom of ITCM on the core and the data part of the APLX file being used to initialise the application's data which is placed in DTCM. Finally, the application core starts executing the application by branching to address zero where the application's executable code has been placed. The code at this address is the SARK start-up code.

Execution Environment

As the ARM968 cores used in SpiNNaker have a Harvard architecture, two blocks of local memory (ITCM and DTCM) are provided in SpiNNaker to take advantage of this and SARK loads application programs into these memories. Executable code is placed at the bottom of ITCM and the data variables used by this code are placed in DTCM. The cores also have access to other memories which are shared by all cores. The executable code in ITCM occupies a contiguous block at the bottom of the memory and there is usually an unused portion of memory above this unless the application fully occupies ITCM. There is a reserved area of 256 bytes at the top of ITCM which is used for application loading and other system functions. Figure 1 shows the layout of ITCM and DTCM.

The application's program variables are stored in DTCM. As most applications are currently written in C, the allocation of DTCM is geared towards the standard C data memory usage.

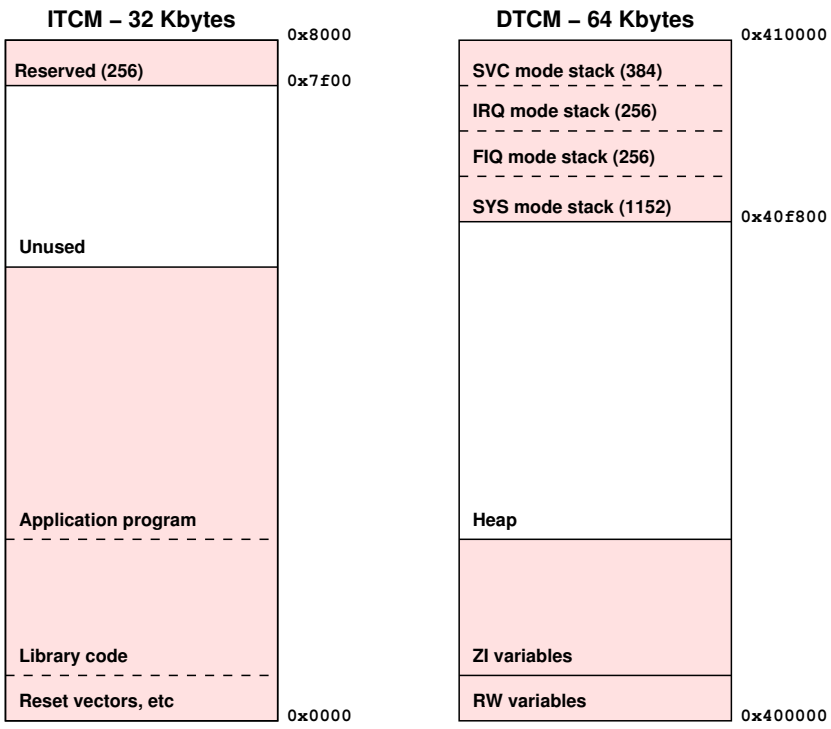

Figure 1: Layout of ITCM and DTCM memories

Two types of C static variables are stored at the bottom of DTCM. At the very bottom are those static variables which are explicitly initialised in the C source. Their values are copied out of the APLX file just before the application starts. Above these initialised variables, which are known as the RW variables, are all the remaining static variables which are typically uninitialised scalars, structs and arrays. These are known as the ZI variables and they are all set to zero before the application starts.

At the top of DTCM are four stacks for four modes of the ARM968 core. The stack pointers for these modes are initialised by SARK at start-up. The SVC mode stack is currently only used when the application starts but is also available for any system calls that may be implemented using the SVC instruction. There are two stacks for the two interrupt modes (IRQ and FIQ) which are used by SpiNNaker applications. The fourth stack is for System mode and this is the mode in which SpiNNaker applications run. The stacks hold the automatic C variables (ie those which are declared inside functions) as well as other information such as return addresses and saved registers.

The area between the top of the ZI variables and the bottom of the stacks is initialised by SARK for use as a heap which can be accessed with malloc and free functions which are provided by SARK. The size of the heap will depend on the size of the RW and ZI data sections. The location of the various data areas (RW, ZI, heap) and the sizes and position of the stacks can be changed in specific circumstances but the defaults shown here should be adequate for most applications.

Access to ITCM and DTCM from the ARM core is fast, usually only requiring a single CPU clock cycle (typically 5ns). Data and instructions should be kept in these memories where possible as access to shared memories (System RAM and SDRAM) is typically 20 to 30 times slower.

SARK Start-up

The SARK start-up code is placed in ITCM at address zero and is entered there to start the application. The overall aim of the start-up code is to establish the environment before the main procedure of the application, `c_main`, is called. This involves a significant number of steps, not all of which will be described here.

The start-up code contains the ARM exception vectors and these are all initialised to point to suitable handler routines. The start-up begins at the reset vector and then proceeds to do the following steps

- Replaces the reset vector with an error handler to trap branches through zero.
- Sets the ARM968 CP15 (coprocessor) control register to enable the write buffer, trap misaligned data accesses, enable Thumb code, and use low vectors.
- Calls a function `sark_config` to allow the application to adjust stack positions and size and set some other configuration values before the main SARK initialisation takes place. This function is declared **weak** so if the application does not define it, an empty function is called instead.
- Fills the stack areas with a predefined data value so that it's possible to see how much stack has been used.
- Calls a function `sark_init` to set up the stacks for the various modes and set up heap and SDP message buffers. An area of shared system RAM which allows the application to communicate with the monitor processor is also set up and the VIC interrupt controller is initialised. `sark_init` is also weak and so can be superseded by the application but this is unlikely to be necessary in most cases. On return from `sark_init` the core is placed into System mode with (both) interrupts enabled which is the mode in which applications execute.
- Calls a function `sark_wait` to optionally wait for a signal from the host before proceeding. The wait is done in sleep mode to minimise power consumption.
- Calls a (weak) function `sark_pre_main` to allow the application to set up anything that needs to be initialised before the application starts to execute. This is typically used to transparently set up system infrastructure such as the Spin1 API.
- Calls the function `c_main` which must be defined by the application and causes the application to start execution. If `c_main` returns, the following steps also take place.
- Calls a (weak) function `sark_post_main` which may be used to tidy up after the application has finished. Again, this is typically used to clean up system infrastructure such as that provided by Spin1 API.
- Calls the function `sark_sleep` which causes the core to go into a low power mode. In this mode, interrupts are still serviced so it will still be possible for the monitor processor to talk to the core.

SARK Source Files

SARK is written mostly in C with a single assembly language source file which contains the start-up code and a number of hand-coded library routines. Two C header files are required and the full list is as follows

```
spinnaker.h      # SpiNNaker hardware definitions
sark.h           # SARK definitions and documentation

sark_alib.s      # Assembly language bits and pieces

sark_alloc.c     # Memory and router allocation
sark_base.c      # SDP message passing and some basic functions
sark_event.c     # Event handling
sark_hw.c        # Routines to interface to SpiNNaker hardware
sark_io.c        # A simple IO library providing "printf"
sark_isr.c       # Interrupt service routines
sark_pkt.c       # Packet transmission routines
sark_timer.c     # Routines providing timed events
```

To build an application using SARK it is necessary to include the header file `sark.h` in your C source. This will also automatically include `spinnaker.h`.

`sark.h` is commented in Doxygen style to allow automatic generation of documentation and this file should be used as the primary reference for documentation of SARK's types, data structures and functions.

SARK Library Functions

SARK provides a set of low-level functions which may be used by application programs or by other system-level software. They are documented in `sark.h` and the following list is only intended to give a general indication of what is provided. The functions fall into the following categories.

- Control of the ARM968 CPU - disabling and enabling interrupts, writing and reading the CP15 control register and the CPSR, putting the CPU into sleep mode, shutting down the CPU.

```
uint cpu_fiq_disable (void);
uint cpu_fiq_enable (void);
uint cpu_int_disable (void);
uint cpu_int_enable (void);
uint cpu_irq_disable (void);
uint cpu_irq_enable (void);

void cpu_int_restore (uint cpsr);

uint cpu_get_cpsr (void);
void cpu_set_cpsr (uint cpsr);

uint cpu_get_cp15_cr (void);
void cpu_set_cp15_cr (uint value);

void cpu_wfi (void);
void cpu_sleep (void) __attribute__((noreturn));
void cpu_shutdown (void) __attribute__((noreturn));
```

- Memory manipulation - simple routines to copy and fill memory which are generally smaller (but less efficient) than the corresponding ARM library routines.

```
void sark_mem_cpy (void *dest, const void *src, uint n);
void sark_str_cpy (char *dest, const char *src);
uint sark_str_len (char *string);
```

```
void sark_msg_copy (sdp_msg_t *to, sdp_msg_t *from);
void sark_word_cpy (void *dest, const void *src, uint n);
void sark_word_set (void *dest, uint data, uint n);
```

- Random number generation - a simple pseudo-random number generator for 32-bit numbers with a seeding function.

```
void sark_srand (uint seed);
uint sark_rand (void);
```

- SDP messaging - allocation of buffers in DTCM and shared memory for SDP messages and a routine to send an SDP message via the monitor processor.

```
sdp_msg_t *sark_msg_get (void);
void sark_msg_free (sdp_msg_t *msg);
sdp_msg_t *sark_shmsg_get (void);
void sark_shmsg_free (sdp_msg_t *msg);
uint sark_msg_send (sdp_msg_t *msg, uint timeout);
```

- Error handling routines to allow an application to signal that it has encountered an error.

```
void rt_error (uint code, ...);
void sw_error (sw_err_mode mode);
```

- Simple text output - a primitive `printf` which sends text output via SDP, to an SDRAM buffer or to a string (like `sprintf`).

```
void io_printf (char *stream, char *format, ...);
```

- Locks and semaphores - routines to acquire and free one of the 32 hardware locks provided by the chip. Routines to lower and raise arbitrary 8-bit semaphore variables using one of the hardware locks to control access. Routines to lower and raise application-specific semaphores and to count the number of cores running an application.

```
uint sark_lock_get (spin_lock lock);
void sark_lock_free (uint cpsr, spin_lock lock);

void sark_sema_raise (uchar *sema);
uint sark_sema_lower (uchar *sema);

uint sark_app_raise (void);
uint sark_app_lower (void);
uint sark_app_sema (void);

uint sark_app_lead (void);
uint sark_app_cores (void);
```

- Memory management - malloc and free routines for memory in DTCM, System RAM and SDRAM. Routines to allocate and free blocks of router MC table entries.

```
void *sark_alloc (uint count, uint size);
void sark_free (void *ptr);

void *sark_xalloc (heap_t *heap, uint size, uint id, uint lock);
void sark_xfree (heap_t *heap, void *ptr, uint lock);
uint sark_xfree_id (heap_t *heap, uint id, uint lock);

void *sark_tag_ptr (uint tag, uint app_id);
uint sark_heap_max (heap_t *heap, uint lock);
heap_t *sark_heap_init (uint *base, uint *top);
```

```
uint rtr_alloc (uint size);
uint rtr_alloc_id (uint size, uint app_id);
void rtr_free (uint entry, uint clear);
uint rtr_free_id (uint id, uint clear);
```

- Environment queries - routines to get core ID, chip ID, application ID and name.

```
uint sark_core_id (void);
uint sark_chip_id (void);
uint sark_app_id (void);
char *sark_app_name (void);
```

- Hardware management - routines to control the VIC (vectored interrupt controller). Routines to control any LEDs attached to the SpiNNaker chip. A routine to busy-wait for a number of microseconds. Routines to initialise and control the various tables in the router.

```
void sark_vic_init (void);
void sark_vic_set (vic_slot slot, uint interrupt, uint enable,
                   int_handler handler);

void sark_led_init (void);
void sark_led_set (uint leds);

void sark_delay_us (uint n);

void rtr_mc_init (void);
uint rtr_mc_clear (uint start, uint count);
void rtr_mc_set (uint entry, uint key, uint mask, uint route);
uint rtr_mc_get (uint entry, rtr_entry_t *r);
uint rtr_mc_load (rtr_entry_t *e, uint count, uint offset, uint app_id);

void rtr_p2p_init (void);
void rtr_p2p_set (uint entry, uint value);
uint rtr_p2p_get (uint entry);

void rtr_diag_init (const uint *table);
void rtr_init (uint monitor);
```

- Timer management - routines to schedule (and cancel) events at some time in the future to microsecond resolution.

```
void event_register_timer (vic_slot slot);

void timer_schedule (event_t *e, uint time);
uint timer_schedule_proc (event_proc proc, uint arg1, uint arg2, uint time);
void timer_cancel (event_t *e, uint ID);
```

- Packet transmission - routines to transmit packets.

```
void event_register_pkt (uint queue_size, vic_slot slot);

uint pkt_tx_k (uint key);
uint pkt_tx_kd (uint key, uint data);
uint pkt_tx_kc (uint key, uint ctrl);
uint pkt_tx_kdc (uint key, uint data, uint ctrl);
```

- Event management - Registration of event handlers, routines to start and stop event processing, routines to add events to event queues.

```
void event_register_int (event_proc proc, event_type event, vic_slot slot);
void event_register_queue (event_proc proc, event_type event, vic_slot slot,
                        event_priority priority);
void event_register_pause (event_proc proc, uint arg2);

void event_enable (event_type event, uint enable);

uint event_start (uint period, uint events, uint wait);
void event_stop (uint rc);
void event_wait (void);
void event_run (uint restart);
void event_pause (uint pause);

event_t* event_new (event_proc proc, uint arg1, uint arg2);
void event_alloc (uint events);

uint event_queue (event_t *e, event_priority priority);
uint event_queue_proc (event_proc proc, uint arg1, uint arg2,
                    event_priority priority);

uint event_user (uint arg1, uint arg2);
```

SARK Data Structures

An application which has been linked with SARK has access to a number of data structures which contain information relevant to the application. Some of these data structures are in core-local memories (ITCM and DTCM) while some are in shared memories (System RAM and SDRAM). They are all documented in the SARK header files - `spinnaker.h` and (primarily) `sark.h`. A brief description follows and you should refer to the header files for full details of the data structures.

A struct of type `sark_vec_t` is located in ITCM at address 0x20. A pointer variable `sark_vec` provides access to its fields. This data structure is intended to be changed only occasionally. It contains vector addresses for the various ARM exceptions, specifications of the location and sizes of the ARM stacks, sizes of various buffers and pointers into various parts of the DTCM such as the heap base and stack limits. It is built into the application code and is intended to be used during application start-up to configure the system. It can be modified during application start-up (via function `sark_config`) to change some configuration information and as the application runs to change the exception vectors.

A struct of type `sark_data_t` is located in DTCM at a link-time dependent address. It may be accessed using the variable `sark`. This data structure contains various pieces of state which must be accessed as efficiently as possible. It contains several pointers to other data structures associated with the application such as the heap (in DTCM) and core-specific buffers in System RAM and SDRAM.

A struct of type `event_data_t` is also located in DTCM at a link-time dependent address. It may be accessed using the variable `event`. It contains state relating to the SARK event handling routines and will not be present if this facility is not being used.

A struct of type `vcpu_t` is located in System RAM. A pointer to this structure is available in the variable `sark.vcpu`. This structure contains data which must be accessible to other cores in the chip, primarily the monitor processor (and hence the host). Some fields are used for message passing between cores. There is a register dump field where debug information is placed when the core encounters a fatal error and also information about what application is running on the core, what its current state is, when it started execution, etc. Most of this data structure is

reinitialised every time a new application starts on the core but there are 4 words (`user0-user3`) which are not and may be used to pass data between successive applications starting on the core.

Each core also has a private buffer in SDRAM which is unstructured and of size 128KB (the size is configurable at system boot time). There is a pointer to it in the variable `sark.sdram_buf`.

System Variables

In addition to the per-core data structures there are a number of shared data structures such as heaps in the SDRAM and System RAM and an area of system variables in System RAM. These are mostly set up and maintained by the monitor processor. The system variables are of interest to applications because they provide much information about the state of the system and provide pointers into various pieces of shared state. The system variables are described by a struct of type `sv_t` and this is mapped into System RAM and accessible via a pointer variable `sv`. The system variables are documented inline in the definition of `sv_t` in `sark.h`.

Shared Memories

Each SpiNNaker chip contains three memories which are accessible by all cores over a shared system bus. One of these is the boot ROM from which the chip bootstraps when it is reset. It is read-only and will not be discussed further here. The other two memories are an on-chip 32KB RAM memory known as the System RAM and a separate 128MB synchronous DRAM chip in the same package known as the SDRAM.

The System RAM (aka SysRAM) and SDRAM are extensively used where data needs to be shared between cores. They are accessible either by normal load and store operations from the cores or via the DMA controller in each core which can be used to copy data to or from the local DTCM and ITCM memories in each core. SARK (and SC&MP) impose some structure on the SysRAM and SDRAM which is described here.

System RAM

The organisation of the SysRAM is shown in figure 2. The names to the left of the diagram indicate the system variable which points to the relevant part of the memory. The addresses to the right are for illustration only and should not be used directly by any user code (ie they may change!).

At the bottom of the SysRAM is a 256 byte Reset Block which is reserved for system use and contains reset vectors and initialisation code to allow application cores to be reset by the monitor processor and then to enter user-specified applications. Applications should not try to alter this area.

Above the Reset Block is an area, the User SysRAM, reserved for user applications. Its size is configurable when the system boots. There is a pointer in the system variables which points to the base of this area. The default configuration sets the size of this area to 4K bytes.

Above the User SysRAM is the SysRAM Heap, a heap from which memory may be allocated and freed by calling the appropriate SARK functions. The size of this area is determined by the amount of User SysRAM which is allocated. It is used to provide shared memory SDP message buffers for communication between the monitor processor and application cores. With the default User SysRAM allocation of 4KB, the heap is around 20KB. The system variable `sv->sysram_heap` points to the heap.

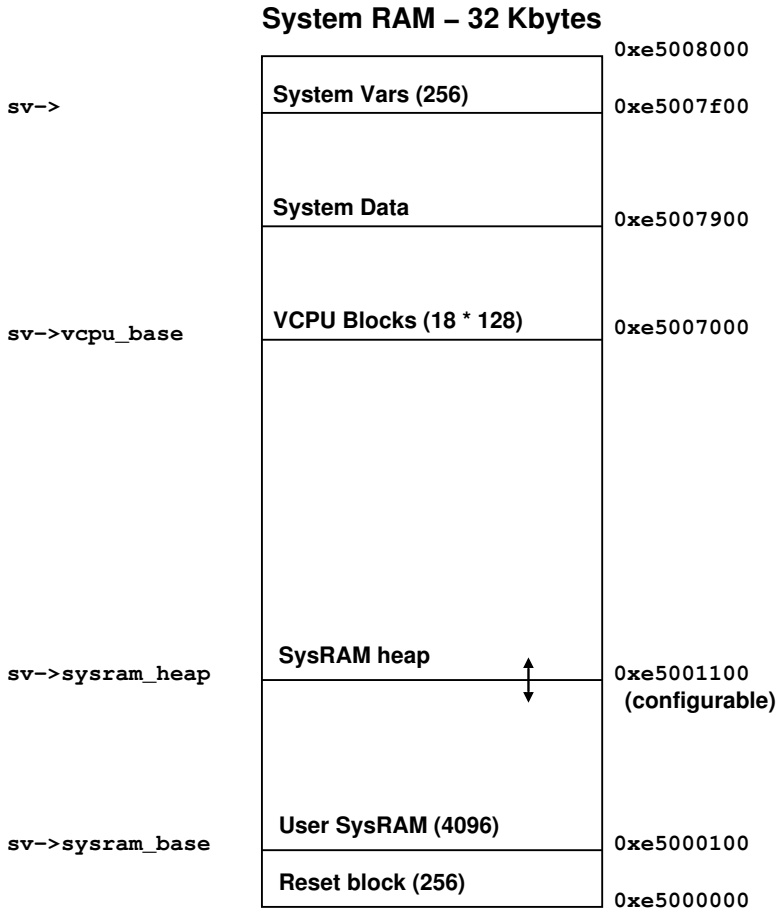

Figure 2: Layout of System RAM memory

Above the SysRAM Heap is an area containing 18 data structures (VCPU blocks) of type `vcpu_t`, one for each core on the chip. These are used for a variety of actions including message passing, status saving, etc, etc. The local state of each core contains a pointer (`sark.vcpu`) to the relevant VCPU block so that it can be readily accessed from application code.

Above the VCPU blocks is an area of System Data which is used by SC&MP and which should not be directly accessed by application code.

Finally, at the top of SysRAM is a 256 byte area which contains most of the system variables which are needed by SARK and SC&MP. This area is mapped onto a struct of type `sv_t` and the fields of the struct are accessible via the variable `sv`.

SDRAM

The organisation of the SDRAM is shown in figure 3. As for the SysRAM, the names to the left of the diagram indicate the system variable which points to the relevant part of the memory while the addresses to the right are for illustration only.

At the bottom of the SDRAM is an area containing 18 SDRAM Buffers, one for each core. The size of these buffers is configurable at boot time and defaults to 128KB (each). The local state of each core contains a pointer (`sark.sdram_buf`) to the buffer belonging to that core.

Above the SDRAM Buffers is the User SDRAM, reserved for user applications. Like the SysRAM, the size is configurable at boot time. The default configuration is for 4M bytes and there is a pointer in the system variables to the base of this area.

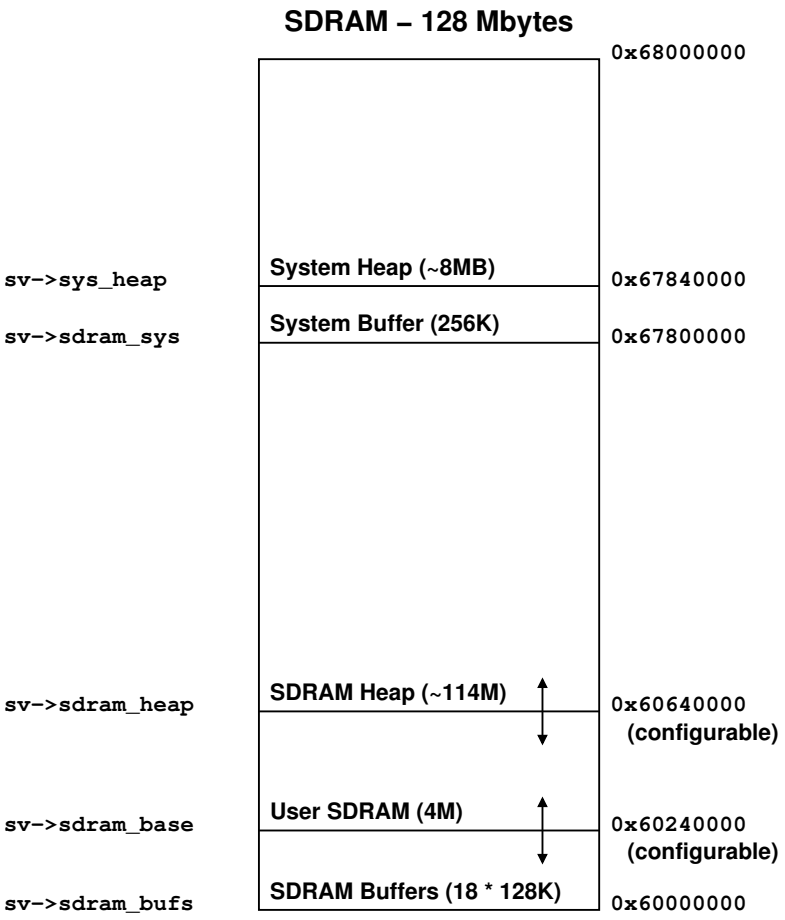

Figure 3: Layout of SDRAM memory

Above the User SDRAM is the SDRAM Heap from which memory may be allocated and freed with the appropriate functions. The size is determined by the amount of memory allocated to the SDRAM Buffers and User SDRAM and the default allocation for these results in a heap of around 114M bytes.

Above the SDRAM Heap is a System Buffer which is used by the monitor processor. At the top of SDRAM is a second heap, the System Heap, which is also used by the monitor processor. Neither of these areas should be accessed by applications.

Change log:

- *1.20 - 07aug13 - ST* - initial release - comments to *steven.temple@manchester.ac.uk*
- *1.21 - 05sep13- ST* - minor update for 1.21
- *1.30 - 07apr14- ST* - update for 1.30
- *1.33 - 19sep14- ST* - update for 1.33 - added app_id parameter to rtr_mc_load.
- *2.0.0 - 08mar16- ST* - update for 2.0.0