PAWS a Risc-V RV32IMCB CPU – Programming Guide

About

PAWS is a project to occupy my time in retirement, with the aim of teaching myself about FPGA programming. It is based upon the idea of the 8-bit computers and consoles from the 1980s, but using a modern CPU.

It is a development from the work I did on the J1 CPU, which is a 16-bit CPU designed to run Forth natively, to which I added a display via HDMI, plus various input/out facilities such as a UART, button input, LED output and basic audio.

The J1 CPU has been swapped out for a Risc-V RV32 processor, as this can easily be programmed via C using GCC, in addition to being relatively straightforward to being implemented on an FPGA.

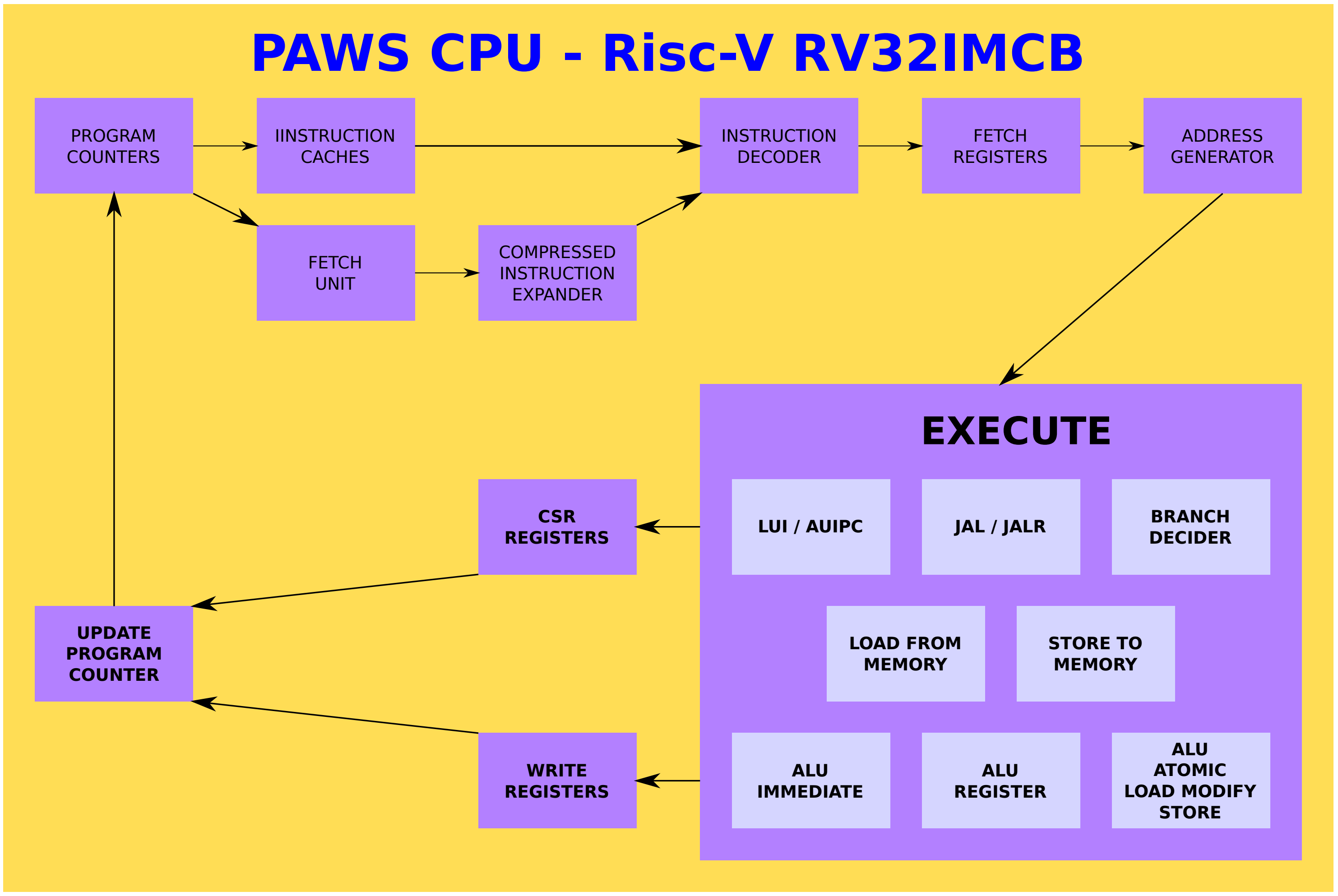
A support library, libPAWS, for easy access to the hardware is provided. This documentation details libPAWS and describes the hardware.

Silice

PAWS is coded in Silice, a hardware description language developed by @sylefeb. Details can be found here GitHub (https://github.com/sylefeb/Silice).

My coding style may not result in the *best* design, the aim was to create a design that could be easily understood.

PAWS CPU and SOC



The PAWS CPU is a Risc-V RV32IMCB that implements only the features needed to run GCC compiled code.

- No interrupts.
- Machine mode only.

The PAWS CPU has two modes:

- Single thread – all available cycles.
- Dual thread
  - Execute an instruction on each thread alternatively.
  - Second thread can be stopped/started as required.

Instruction and SDRAM Caches

There are 3 types of cache in PAWS. Each CPU thread, referred to as “MAIN” and “SMT”, has a 32 instruction cache, of the most recently fetched instructions. These are specifically designed to assist in the execution of small loops.

The SDRAM has separate instruction (8k) and data (4k) directly mapped caches. Memory accesses to memory are marked by the CPU as being to fetch instructions or data so that the appropriate cache is used.

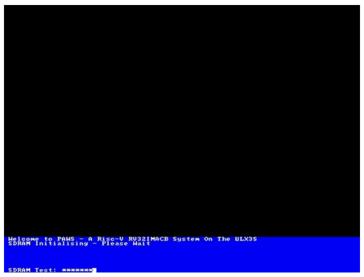
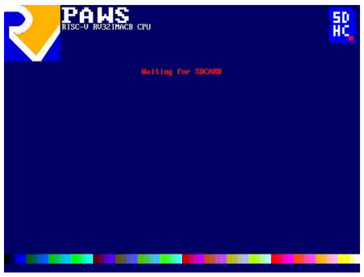
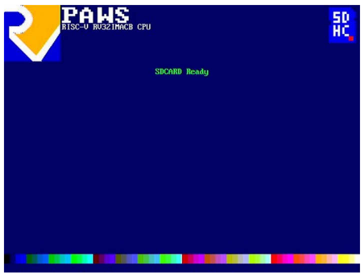
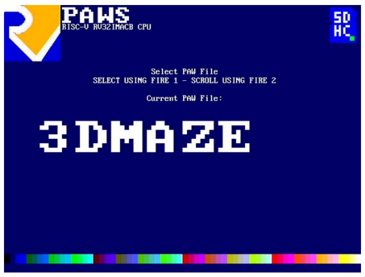
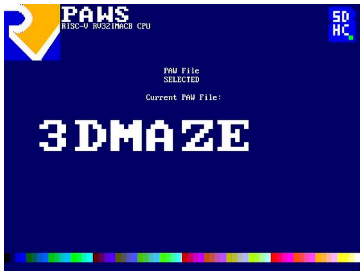
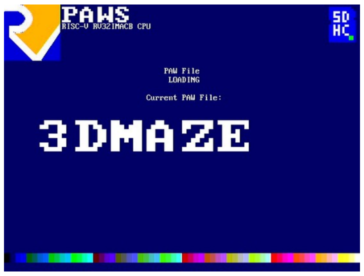
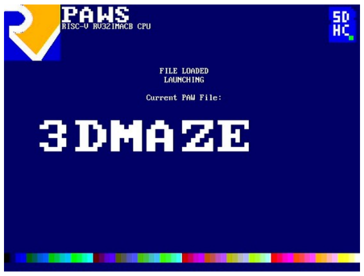
Memory access are organised as 16 bit, the bus width of the SDRAM chip on the ULX3S. 16-bit compressed instructions are preferred as due to latency during instruction fetching these will be fetched, decoded and executed quicker than 32-bit instructions.

The bit manipulation extension was implemented to increase code density, as when output by GCC they will replace multiple RV32I instructions, again leading to quicker code execution due to reduced memory access. Presently maintained to draft 0.94 from January 2021.

Instruction Set Listings

Risc-V	Implemented	Not Implemented	Notes
BASE	ADD[I] SUB SLT[I][U] AND[I] OR[I] XOR[I] SLL[I] SRL[I] SRA[I] AUIPC/LUI		
BASE unconditional jumps	JAL[R}		
BASE conditional brnaches	BEQ/BNE BLT[U] BGE[U]		
BASE load and store	LB[U] LH[U] LW SB SH SW		
BASE		FENCE FENCE.I	
BASE CSR	RDCYCLE[H] RDTIME[H] RDINSTRET[H]		READ ONLY
BASE SYSTEM		ECALL EBREAK	
M EXTENSION	DIV[U] REM[U] MUL MULH[[S]U]		
A EXTENSION	AMOADD AMOSWAP AMOAND AMOOR AMOXOR AMOMAX[U] AMOMIN[U]		AQ / RL flags are ignored. The AMO instructions do operate as a complete READ-MODIFY_WRITE operation, as intended. The other thread will not execute an instruction until this has completed.
B EXTENSION Zba Zbb Zbc Zbe Zbf Zbp Zbs	ANDN ORN XNOR BEXT BDEP BFP CLMUL[[H]R] CLZ CTZ PCNT GREV[I] GORC[I] MIN[U] MAX[U] ORC.B PACK[H][U] REV8 ROL ROR[I] SBCLR[I] SBINV[I] SBSET[I] SBEXT[I] SEXT.B SEXT.H SH[[1][2][3]]ADD SHFL[I] UNSHFL[I] XPERM.B XPERN.H XPERM	CRC32 CRC32C CMOV CMIX FSL FSR[I]	

Using PAWS

 Welcome to PAWS - A Risc-V RV32IMACB System On The ULX3S SDRAM Initialising - Please Wait SDRAM Test: ******** BIOS SDRAM Initialisation	 Waiting for SDCARD BIOS SDCARD Initialisation	 SDCARD Ready BIOS Reading SDCARD	 Select PAW File SELECT USING FIRE 1 - SCROLL USING FIRE 2 Current PAW File: 3DMAZE BIOS PAW File Selection
 PAW File SELECTED Current PAW File: 3DMAZE BIOS PAW File Selected	 PAW File LOADING Current PAW File: 3DMAZE BIOS PAW File Loading	 FILE LOADED LAUNCHING Current PAW File: 3DMAZE BIOS PAW File Loaded	

Upon startup PAWS runs the BIOS, initialises the display, tests and clears the SDRAM, clears the input/output buffers, and reads the ROOT DIRECTORY from PARTITION 0 of a FAT16 formatted SDCARD.

PAW (compiled programs) files are display for selection. Scroll through the available files using “LEFT” and “RIGHT”, then select a file for loading and executing using “FIRE 1”.

Upon selection, the BIOS will load the selected PAW into SDRAM, reset the display, and launch the selected PAW program.

PAWS a Risc-V RV32IMCB CPU – Programming Guide

Compiling Programs For PAWS

The default language for PAWS is C, specifically GCC.

To create a program for PAWS, create a C file in the SOFTWARE/c directory. It is advised to use the SOFTWARE/template.c as a starting point.

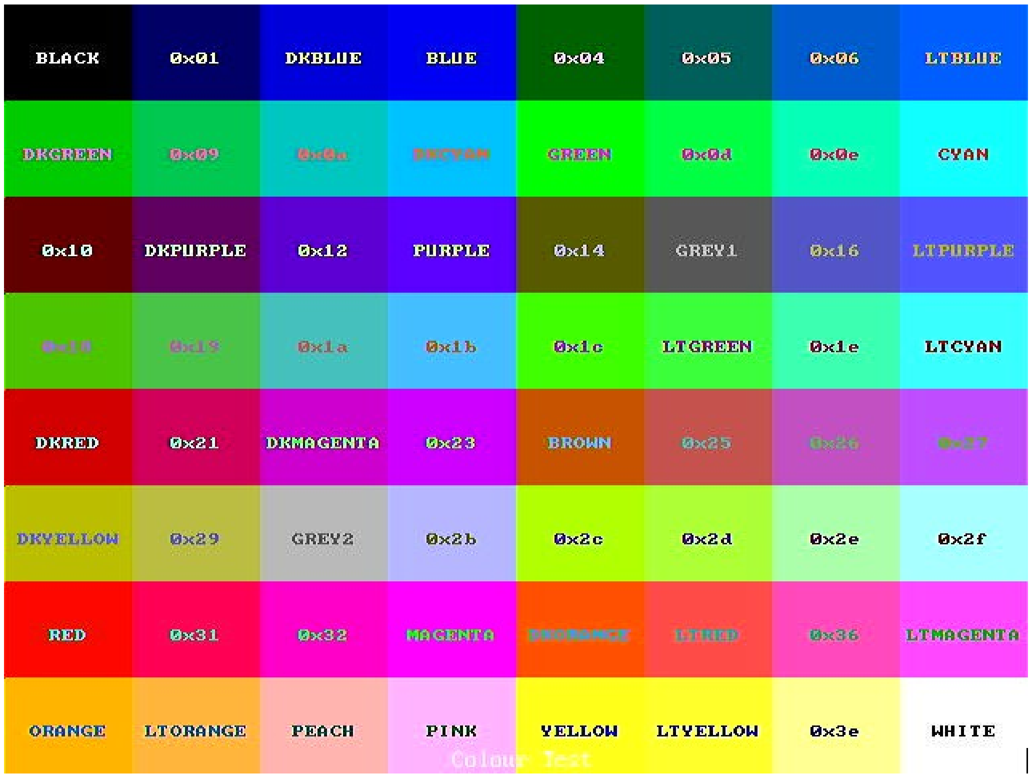
Contents of template.c	Explanation
`#include "PAWSlibrary.h"`	Use libPAWS for definitions and helper functions.
`void main( void ) {`	MAIN program loop
`    INITIALISEMEMORY();`	Setup the memory map
`    while(1) {`	Main loop.
`    }`	
`}`	

Compile your code using the helper shell script. For example, to compile the included asteroids style arcade game, `./compile.sh c/asteroids.c PAWS/ASTROIDS.PAW` . This will compile the program to PAWS/ASTROIDS.PAW, which can be copied to the SDCARD for loading via the BIOS.

Alternatively, an experimental script, which strips unused functions from the finished file is provided, use `./wpcompile.sh` instead of `./compile.sh`.

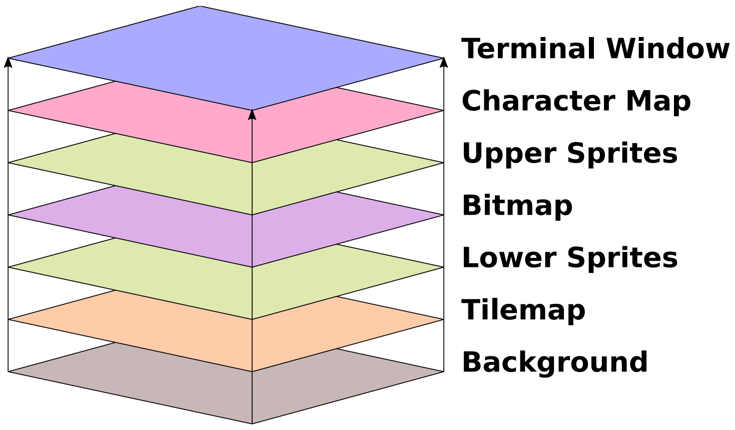
Colours

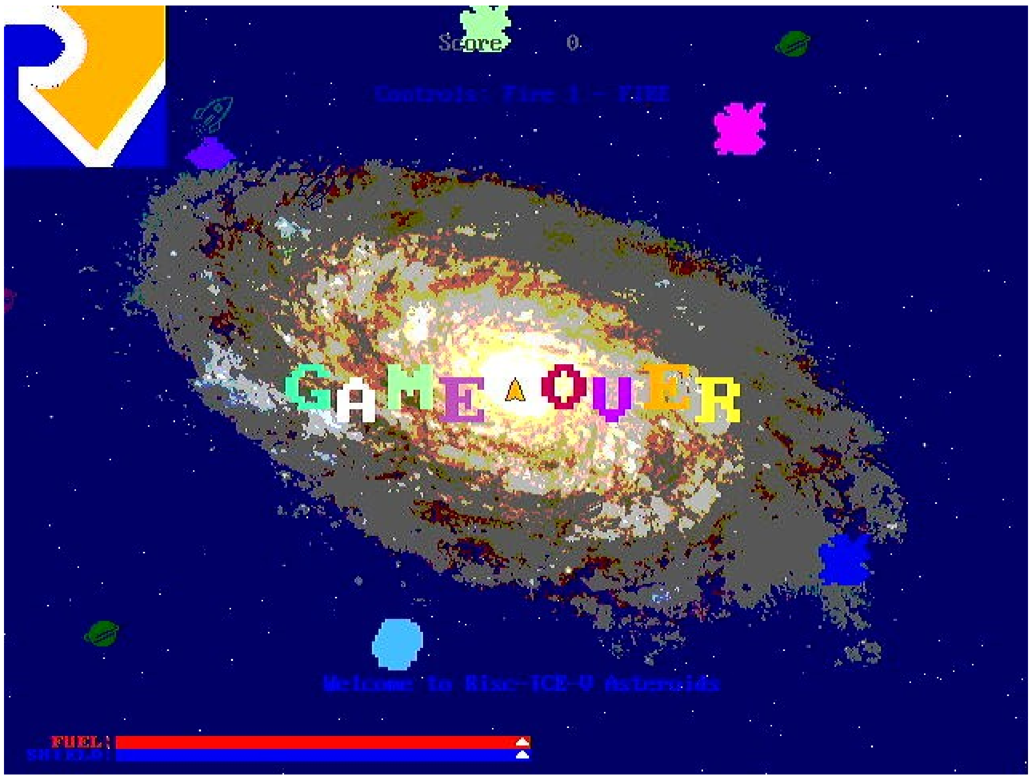
PAWS uses a 6-bit colour attribute, given as RRGGBB. This gives 64 colours, specified in hexadecimal as per the table below. Names defined in libPAWS are shown.



BLACK	0x01	DKBLUE	BLUE	0x04	0x05	0x06	LTBLUE
DKGREEN	0x09	0x0a	DKCYAN	GREEN	0x0d	0x0e	CYAN
0x10	DKPURPLE	0x12	PURPLE	0x14	GREY1	0x16	LTPURPLE
0x18	0x19	0x1a	0x1b	0x1c	LTGREEN	0x1e	LTCYAN
DKRED	0x21	DKMAGENTA	0x23	BROWN	0x25	0x26	0x27
DKYELLOW	0x29	GREY2	0x2b	0x2c	0x2d	0x2e	0x2f
RED	0x31	0x32	MAGENTA	DKORANGE	LTRED	0x36	LTMAGENTA
ORANGE	LTORANGE	PEACH	PINK	YELLOW	LTYELLOW	0x3e	WHITE

Display Structure

 Terminal Window Character Map Upper Sprites Bitmap Lower Sprites Tilemap Background Default Display Layer Ordering	The display, 640 x 480 pixels in 64 colours, is output via HDMI. It is formed of layers, starting with the character map window at the top, down to the Background. All other layers, have transparency so that if no pixel is showing in that layer, the layer below shows.



PAWS running Asteroids, showing the background layer (dark blue with falling stars), the bitmap layer (Risc-V logo, galaxy image, “GAME OVER” message and the fuel bars), the tilemap (the small planets and rocket ships), the sprite layers (asteroids, UFO and player ship), and the character map layer (the player score and instructions).

PAW Asteroids runs in screen mode 2, where the bitmap is displayed below the lower sprites and the tilemap.

libPAWS Variables and Functions

`void await_vblank( void )`	Waits for the screen vertical blank to start.
`void screen_mode( unsigned char screenmode )`	Changes the display layer order. 0 = default, as above. 1 = bitmap between lower sprites and the tilemap. 2 = bitmap between the tilemap and the background. 3 = bitmap between the upper sprites and the character map.

Memory Management

The BIOS will initialise the memory, and allocates space at the top of fast BRAM memory for the CPU STACK, and space at the top the SDRAM for SDCARD buffers.

Address Range	Memory Type	Usage
0x00000000 – 0x00004000	Fast BRAM	0x00000000 – 0x00001000 BIOS 0x00004000 – 0x00002000 Main Stack 0x00002000 – 0x00001000 SMT Stack
0x00008000 – 0x0000ffff	I/O Registers	Commuincation with the PAWS hardware. No direct hardware access is required, as libPAWS provides functions for all aspects of the PAWS hardware.
0x1000000 -0x1fffffff 0x1000000 – 0x10800000 0x10800000 -	SDRAM PROGRAM + LOADED DATA MALLOC ALLOCATED MEMORY	Program and data storage. Accessed via instruction and data caches. SDCARD buffers and structures are allocated at the top of this address range.

libPAWS variables and functions

`unsigned char *MEMORYTOP`	Points to the top of unallocated memory.
`void INITIALISEMEMORY( void )`	Sets up the memory map using parameters passed from the BIOS. Allocates the buffers used for SDCARD access and initialises malloc.

`void *malloc( size_t size )`
`void free( void *pointer )`
	Simple versions of C's malloc and free from https://github.com/andrestc/linux-prog/blob/master/ch7/malloc.c

`unsigned char *filemalloc( unsigned int size )`	Allocates space for a file to be read into memory. Preferred over malloc so as to allow for clusters from the SDCARD to be read into.

File Management

libPAWS has the ability to load files from the SDCARD directly into memory.

libPAWS variables and functions

`unsigned short sdcard_findfilenumber( unsigned char *filename, unsigned char *ext )`	Returns the number of the file in the root directory on the SDCARD, or 0xffff if the file is not found.
`unsigned int sdcard_findfilesize( unsigned short filenumber )`	Returns the size of the file, in bytes.
`void sdcard_readfile( unsigned short filenumber, unsigned char * copyAddress )`	Reads the file into memory.

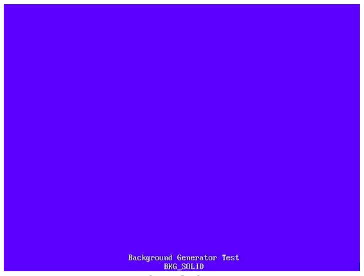
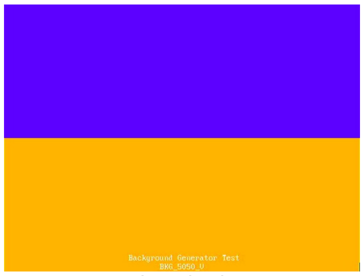
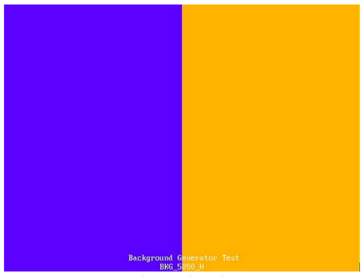
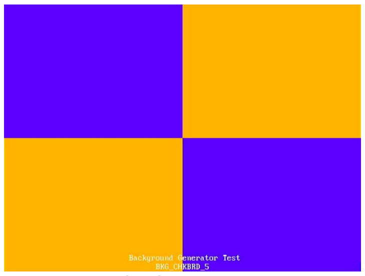
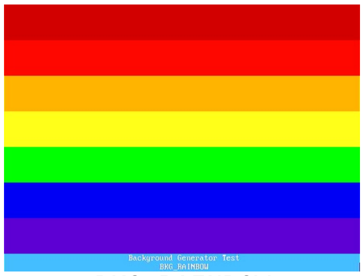
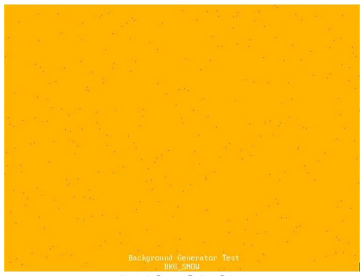
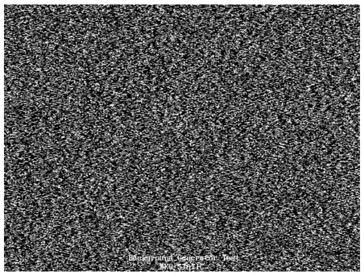
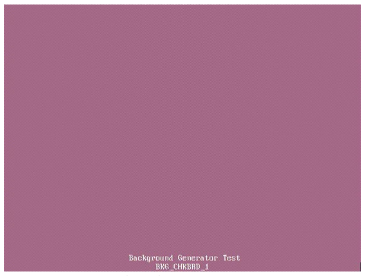
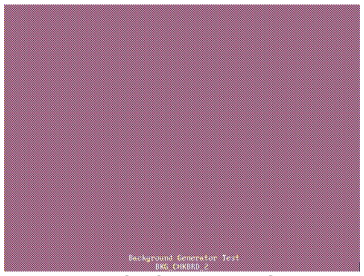
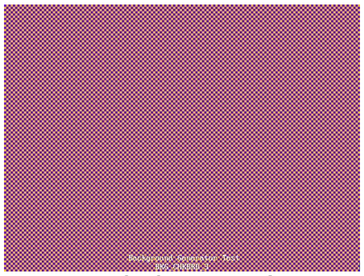
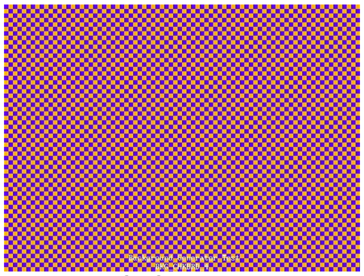
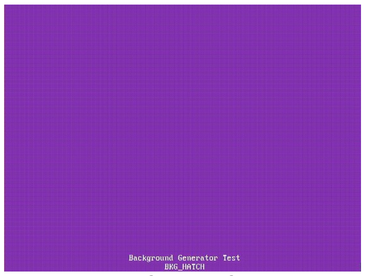
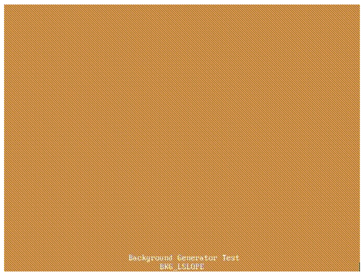
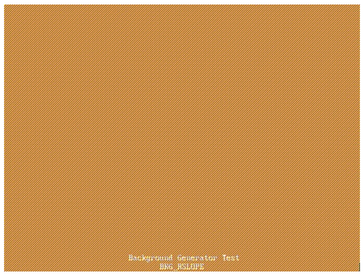
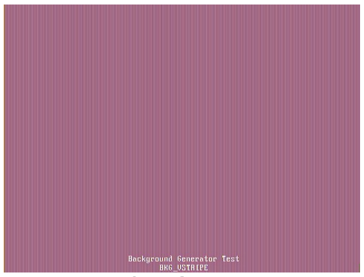
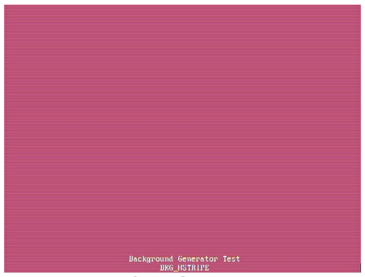
NOTE: memoryspace and filememoryspace are similar to the standard C function malloc. There is no mechanism for freeing memory in libPAWS.

Example code for loading a file (GALAXY.JPG) into memory

```
#include "PAWSlibrary.h"

void main( void ) {
    INITIALISEMEMORY();

    unsigned char *galaxyfilebuffer;
    unsigned short filenumber;

    while(1) {
        outputstring( "Finding File GALAXY.JPG");
        filenumber = sdcard_findfilenumber( "GALAXY", "JPG" );
        if( filenumber == 0xffff ) {
            outputstring( "FILE NOT FOUND" );
        } else {
            galaxyfilebuffer = filemalloc( sdcard_findfilesize( filenumber ) );
            sdcard_readfile( filenumber, galaxybuffer );

        (void)inputcharacter();    }
}
```

Background Generator

The background layer shows when there is nothing to display from the layers above. There are 11 named generators in libPAWS. The result of the individual background generators are shown in the table below.

Result of set_background(PURPLE, ORANGE, *value from table*);			
 BKG_SOLID	 BKG_5050_V	 BKG_5050_H	 BKG_CHKBRD_5
 BKG_RAINBOW	 BKG_SNOW	 BKG_STATIC	 BKG_CHKBRD_1
 BKG_CHKBRD_2	 BKG_CHKBRD_3	 BKG_CHKBRD_4	 BKG_HATCH
 BKG_LSLOPE	 BKG_RSLOPE	 BKG_VSTRIPE	 BKG_HSTRIPE

Single Thread or Dual Thread Mode

On startup PAWS runs in single thread mode. The BIOS will switch back to single thread mode when returning to the BIOS from a program running in dual thread mode.

When dual thread mode is activated PAWS will execute one instruction from each thread alternatively. All memory is shared, with no memory protection.

The Risc-V A Extension (Atomic Instructions) are decoded and executed, but ignoring the *aq* and *rl* flags. The whole of the fetch-modify-write cycle will complete before allowing the other thread to execute. FENCE instructions from the Risc-V base are treated as no-ops.

libPAWS variables and functions

`void SMTSTOP( void )void SMTSTART( unsigned int code )`	Stops the SMT thread.
`void SMTSTART( unsigned int code )`	Starts the SMT thread, jumping immediately to the address of the function provided.

Due to the way that all of the I/O operations are memory mapped there are considerations to make when writing dual threaded code. Some suggestions for best practice are listed below:

- Only one thread should access the GPU, tilemap, character map or audio. If both threads attempt to they will be overwriting the control registers leading to an unknown outcome.
- All timers are duplicated. The suggestion is that the main thread uses timer set 0, and the smt thread uses timer set 1. Alternatively for synchronisation, the main thread can set a timer, and BOTH threads can wait for that same timer.
- The sprite layers have duplicated control registers to allow both threads to control the sprites (only the main thread should set the sprite bitmaps as that uses a register from the main set to select the current sprite). See the SMT version of asteroids for an example.

Example code for a simple dual thread program

```
#include "PAWSlibrary.h"

void smtthread( void ) {
    // SETUP STACKPOINTER FOR THE SMT THREAD
    asm volatile ("li sp ,0x2000");
    while(1) {
        gpu_rectangle( rng( 64 ), rng( 640 ), rng( 432 ), rng( 640 ), rng( 432 ) );
        sleep( 500, 1 );
    }
}

void main( void ) {
    INITIALISEMEMORY();

    tpu_outputstringcentre( 27, TRANSPARENT, GREEN, "SMT Test" );
    tpu_outputstringcentre( 28, TRANSPARENT, YELLOW, "I'm Just Sitting Here Doing 
Nothing" );
    tpu_outputstringcentre( 29, TRANSPARENT, BLUE, "The SMT Thread Is Drawing 
Rectangles!" );
```

```
    SMTSTART( (unsigned int )smtthread );

    while(1) {
        tpu_set( 1, 1, TRANSPARENT, WHITE );
        tpu_outputstring( "Main Thread Counting Away: " );
        tpu_outputnumber_short( systemclock() );
        sleep( 1000, 0 );
    }
}
```

NOTE: The first line of code in the smtthread function **must** set the stack pointer to the reserved memory in the fast BRAM.

Bitmap and GPU

PAWS has a GPU to draw to the bitmap. The functions in libPAWS will setup the GPU, wait for the previous GPU command to complete, and then start the GPU with the required function.

libPAWS variables and functions

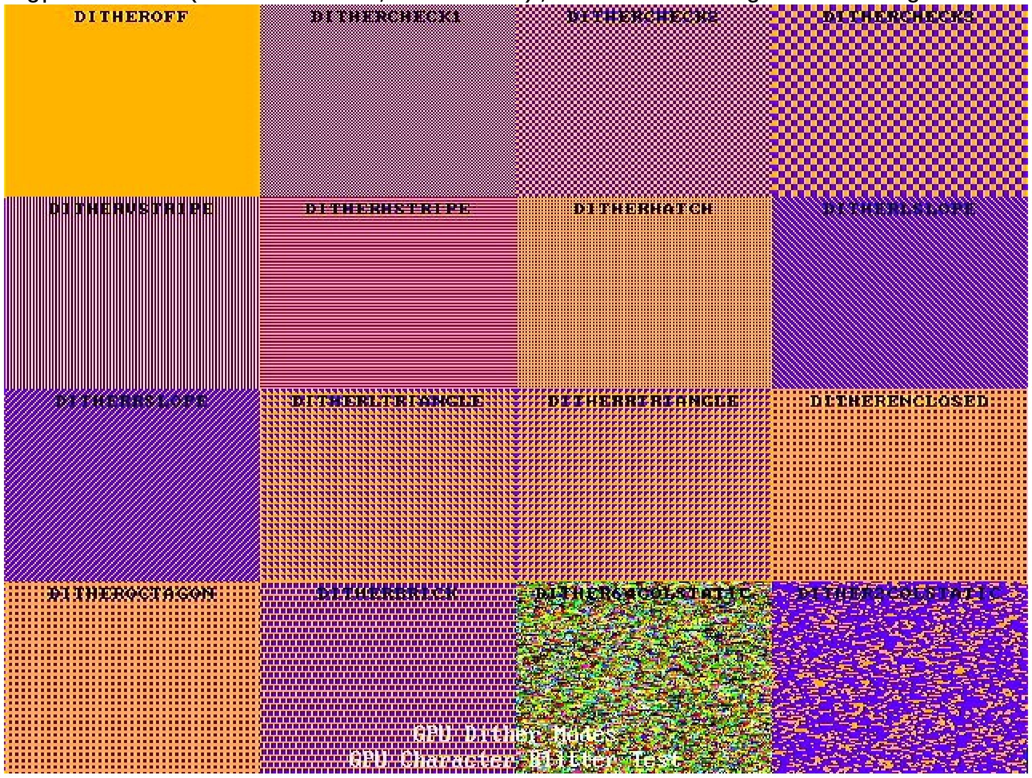
`void gpu_cs( void )`	Clears the bitmap to TRANSPARENT and resets the scroll/wrap.
`void gpu_dither( unsigned char mode, unsigned char colour )`	Sets the dither mode and the alternate colour used for filled rectangles, circles and triangles.
`void gpu_pixel( unsigned char colour, short x, short y )`	Plots a pixel at (x,y) in colour.
`void gpu_line( unsigned char colour, short x1, short y1, short x2, short y2 )`	Draws a line from (x1,y1) to (x2,y2) in colour.
`void gpu_box( unsigned char colour, short x1, short y1, short x2, short y2 )`	Draws an outline rectangle with corners at (x1,y1) and (x2,y2) in colour. NOTE: Drawn by breaking into 4 lines.
`void gpu_rectangle( unsigned char colour, short x1, short y1, short x2, short y2 )`	Draws a filled rectangle with corners at (x1,y1) and (x2,y2) in colour. Uses the dither mode.
`void gpu_circle( unsigned char colour, short x1, short y1, short radius, unsigned char filled )`	Draws a circle at centre (x1,y1) of the given radius in colour, optionally filled. Uses the dither mode when filling.
`void gpu_blit( unsigned char colour, short x1, short y1, short tile, unsigned char blit_size )`	Blit a 16x16 tile (32 user definable tiles) to (x1,y1) in colour. Will size to 16x16, 32x32, 64x64 and 128x128.
`void gpu_character_blit( unsigned char colour, short x1, short y1, unsigned char tile, unsigned char blit_size )`	Blit an 8x8 character (256 user definable characters) to (x1,y1) in colour. Will size to 8x8, 16x16, 32x32, 64x64.
`void gpu_triangle( unsigned char colour, short x1, short y1, short x2, short y2, short x3, short y3 )`	Draws a filled triangle with vertices (x1,y1), (x2,y2) and (x3,y3) in colour. Uses the dither mode. NOTE: Vertices should be presented clockwise from the top.
`void gpu_quadrilateral( unsigned char colour, short x1, short y1, short x2, short y2, short x3, short y3, short x4, short y4 )`	Draws a filled quadrilateral with vertices (x1,y1), (x2,y2), (x3,y3) and (x4,y4) in colour. Uses the dither mode. NOTE: Drawn by breaking into two triangles.
`void gpu_outputstring( unsigned char colour, short x, short y, char *s, unsigned char size )`	Outputs a string by repeatedly using the character blit, starting at (x,y) in colour. Will size the characters and space accordingly.
`void gpu_outputstringcentre( unsigned char colour, short x, short y, char *s, unsigned char size )`	Outputs a string by repeatedly using the character blit, with the top centred at

	(x,y) in colour.Will size the characters and space accordingly.
`void set_blitter_bitmap( unsigned char tile, unsigned short *bitmap )`	Define one of the 32 16x16 blitter tiles.
`void draw_vector_block( unsigned char block, unsigned char colour, short xc, short yc )`	Starts the drawing of one of the 32 user definable vector (line drawn) objects, centred at (x,y) in colour.
`void set_vector_vertex( unsigned char block, unsigned char vertex, unsigned char active, char deltax, char deltay )`	Sets one of the 16 vertices in one of the 32 user definable vector (line drawn) objects.
`void bitmap_scrollwrap( unsigned char action )`	action == 1 LEFT, == 2 UP, == 3 RIGHT, == 4 DOWN, == 5 RESET

Dither Modes

When drawing rectangles, filled circles or filled triangles, the GPU can apply one of 16 “dither” patterns.

Result of `gpu_dither( dithermode, PURPLE ); Plus drawing a rectangle in ORANGE.`


DITHEROFF
DITHERCHECK1
DITHERCHECK2
DITHERCHECK3
DITHERVSTRIPE
DITHERHSTRIPE
DITHERHATCH
DITHERLSLOPE
DITHERRSLOPE
DITHERLTRIANGLE
DITHERRTRIANGLE
DITHERENCLOSED
DITHEROCTAGON
DITHERBRICK
DITHER64COLSTATIC
DITHER2COLSTATIC
GPU Dither Modes
GPU Character Blitter Test