PLUTO 5 DEVELOPMENT KIT SOFTWARE USER MANUAL

Document No. 80-16040 Issue 6r1 HEBER LTD

Current Issue: - Issue 6r1 - 30th January 2007

Previous Issue: - Issue 6 - 12th January 2005

HEBER LTD
Belvedere Mill
Chalford
Stroud
Gloucestershire
GL6 8NT
England

Tel: +44 (0) 1453 886000
Fax: +44 (0) 1453 885013
Email: support@heber.co.uk
http://www.heber.co.uk

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 OVERVIEW

This document is intended to provide all the information required to enable the reader to understand and use the software provided with the Pluto 5 development kit. The software is supplied in compiled library format and packaged in archive files, which can be read by the GNU linker. The development kit software is a subset of the Pluto 5 'FastTrack' software package, figure 1 shows the overall structure of the 'FastTrack' software and which modules are included in the development kit.

1.1 Software Structure

The software is structured in four discrete layers. Each of the three lower layers has a corresponding archive file containing the various object modules shown in figure1.

1.1.1 Hardware Abstraction Layer

The hardware abstraction layer interfaces directly to the Pluto 5 hardware. It allows the user access to all the functions of the hardware without having to directly address the hardware. Structuring the software in this manner allows all the higher levels of software to be platform independent.

1.1.2 Peripheral Driver Layer

The peripheral driver layer consists of drivers for many commonly used peripherals. The drivers are written following a standard template. This allows the drivers to be manipulated in a standard way by the device manager so that peripherals can be automatically installed and shutdown and restored in the event of an error condition.

1.1.3 Interface Layer

The Interface layer consists of two different types of module. It provides an easy to use interface for the game programmer to manipulate the hardware via the drivers.

1.1.3.1 Virtual Device Drivers

Virtual device drivers follow the same template as the drivers in the peripheral layer but they differ from these in one of two ways. They are either not associated with any physical hardware device or they are associated with one or more physical devices, which they address through their associated device drivers.

1.1.3.2 Service Modules

Service modules do not follow the device driver template. They provide services to the application layer, which may or may not be associated with the manipulation of the hardware.

1.1.4 Application Layer

The application layer consists of game modules written by the user. Example game software will be supplied as part of the 'FastTrack' software package.

1.2 Conventions

Throughout the documentation and the software itself certain conventions are adhered to. It is strongly recommended that this is continued throughout the user code.

1.2.1 Type Definitions

Type definitions will be in upper case
E.g. typedef char BYTE;

1.2.2 #Defines

#Defines will be in upper case so that #defines can be distinguished from variables or constants.
E.g. #define TRUE 1

1.2.3 Global Functions

Global function names will be one or more words concatenated; each word will begin with a capital letter for readability.
E.g. HumpBackedCamel();

1.2.4 Static Functions

Static function names will be one or more words concatenated, all letters will be in lower case.
E.g. staticfunction();

1.2.5 Global Variables

Global variable names will be one or more words concatenated; each word will begin with a capital letter for readability.
E.g. GlobalByte=0;

1.2.6 Static Variables

Static variable names will be one or more words concatenated; all letters will be in lower case.
E.g. staticbyte=0;

1.2.7 Local Variables

Local variable names will be one or more words concatenated; all letters will be in lower case.
E.g. localbyte=0;

1.2.8 Structure Elements

Structure elements will be one or more words concatenated, all letters will be lower case.
E.g. ArbitraryStructure.firstelement=0;

1.3 Fundamental Types and #Defines

The types and defines used by each module will be discussed in the documentation for that module, For portability all type declarations will be based on certain fundamental types shown below and not on the standard 'C' integer types which can vary in size from one target processor to another.

1.3.1 BYTE

The BYTE type is an eight bit signed integer in the range –128 to +127.

1.3.2 UBYTE

The UBYTE type is an eight bit unsigned integer in the range 0 to +255.

1.3.3 WORD

The WORD type is a sixteen bit signed integer in the range –32767to +32768

1.3.4 UWORD

The UWORD type is a sixteen bit unsigned integer in the range 0 to +65535

1.3.5 LONG

The LONG type is a thirty-two bit signed integer in the range –2147483647 to +2147483648.

1.3.6 ULONG

The ULONG type is a thirty-two bit unsigned integer in the range 0 to +4294967295.

1.3.7 TRUE

#define TRUE 1

1.3.8 FALSE

#define FALSE 0

1.4 Memory Sections

The memory available to the user software is divided into four sections as follows.

1.4.1 Identity Ram

This section of memory is cleared on start-up only when a change of program identity is detected. It is intended to hold variables whose values must be preserved through a software revision update. It is the responsibility of the game programmer to ensure that each game has a unique identity. The 32bit constant integer 'MachineId' holds the value, which identifies the software. Variables may be placed in this section only on a file by file basis by modifying the linker script. The demonstration software contains a module called 'idram' any uninitialised variables defined in this file are located in the identity ram section of memory. Check the linker script file to see how this is done.

1.4.2 System Ram

This section of memory is cleared on start-up when a change of identity is detected or when a change of EPROM checksum is detected. This section is intended for variables whose values are not initialised on start-up and retain their values through a power down. Variables may be placed in this section only on a file by file basis by modifying the linker script. The demonstration software contains a module called 'sysram' any uninitialised variables defined in this file are located in the system ram section of memory. Check the linker script file to see how this is done.

1.4.3 Initialised Variables

This section holds the standard 'C' declared initialised variables. These variables are automatically set to their assigned values each time the software starts up.

1.4.4 Uninitialised Variables

This section holds the standard 'C' declared uninitialised variables. These variables are automatically set to zero each time the software starts up. The stack and the heap are included in this section.

1.5 Interrupt Management

The system has a periodic interrupt, which occurs every 1ms. The system also has a 10ms periodic interrupt, and a 250ms periodic interrupt generated by timer interrupts.

The 250ms interrupt is used to lock out mainline execution when the system is shut down in the event of an error. Any functions installed in the 250ms interrupt will therefore not be called during system shutdown.

2 HARDWARE ABSTRACTION LAYER

2.1 Entry

The entry module is responsible for the low-level initialisation of the hardware and the memory management functions as described above. In addition the module provides one public function.

2.1.1 InstallIntFnc

BYTE InstallIntFnc(void(*fnc)(),void(**fnc_lst)());

2.1.1.1 Description

Four interrupt scheduler lists are available with pluto5

They are declared and initialised in *config.c* in the *project* directory

```
void  (*Int1msLst[ ] )    ( ) = { 0, 0, 0, 0, 0, 0, 0, 0, 0, EOF_FUNCTIONS};
void (*Int10msLst[ ] )   ( ) = { 0, 0, 0, 0, 0, 0, 0, 0, 0, EOF_FUNCTIONS};
void (*Int250msLst[ ] ) ( ) = { 0, 0, 0, 0, 0, 0, 0, 0, 0, EOF_FUNCTIONS};
void (*IntDuartLst[ ] )   ( ) = { 0, 0, 0, 0, 0, 0, EOF_FUNCTIONS};
```

InstallIntFnc() adds a pointer to an interrupt handler function *fnc,* to the end of the specified interrupt scheduler list *fnc_lst*.

The zeroed entries in each of the above lists are *place holders* for *pointers to interrupt service routines*
The number of entries in a list must be greater than or equal to the number of interrupts *plus 2*
The last entry must be *EOF_FUNCTIONS*

The *order of execution* for handler functions is the same as the *order of installation*

2.1.1.2 Parameters

2.1.1.2.1 Parameter 1

Address of a function to be installed into the given interrupt.

2.1.1.2.2 Parameter 2

Allowed values:

Int1msLst	1ms periodic interrupt list.
Int10msLst	10ms periodic interrupt list.
Int250msLst	250ms periodic interrupt list.
IntDuartLst	Shared interrupt for: DUART0, DUART1, Duart2 and Calypso16 Video.

2.1.1.3 Return Value

Returns TRUE on success. If the interrupt scheduler list is full FALSE is returned.

It is recommended that the **return value is always tested.**

2.1.1.4 Example

Let's consider that "**testinp*()***" needs to be executed every 10ms

```
static void testinp()
{
        do something….
}
```

Thus "**testinp*()***" will have to be installed into the list ran every 10 ms. To do so InstallIntFnc needs to be called during initialisation as below

```
if (InstallIntFnc(testinp, Int10msLst)==FALSE)    /* testinp() will now be run every 10 ms */
        error();                                  /* error: list is full */
```

2.1.2 Managing correctly a specified interrupt scheduler list

In general, the device drivers to be installed are declared in *devices.c*
For example:

```
const   INSTALLDEVICE   DeviceList[]=
        {
        {&LampDevice            ,0,0,0},
        {&InputDevice           ,0,0,0},
        {&OutputDevice          ,0,0,0},
        {&SoundDevice           ,0,0,0},
        {&I2CPICDevice          ,0,0,0},
        {&PICDevice             ,0,0,0},
        {&E2RomDevice           ,0,0,0},
        {&RTCDevice             ,0,0,0},
        {&SerialDevice          ,0,0,0},
        {&DUART0Device          ,0,0,0},
        {&DUART1Device          ,0,0,0},
        {&DUART2Device          ,0,0,0},
        {&UARTDevice            ,0,0,0},
        {&CremsonDevice         ,0,0,0},
        {&TimerDevice           ,0,0,0},
        {&StepperDevice         ,0,0,0},
        {0,0,0}
        };
```

Each driver installed *may* add interrupt handlers to any of the four *interrupt scheduler lists.*

The following is a list of all the interrupt handlers added by various device drivers:

DUART0Device			*IntDuartLst*
DUART1Device			*IntDuartLst*
DUART2Device			*IntDuartLst*
I2CPICDevice	*Int1msLst*		
InputDevice	*Int1msLst*		
LampDevice	*Int1msLst*	*Int10msLst*	
MXIDevice	*Int1msLst*		
OutputDevice	*Int1msLst*		
UARTDevice	*Int1msLst*		
CremsonDevice	*Int1msLst*		*IntDuartLst*
EloDevice		*Int10msLst*	
MicroTouchDevice		*Int10msLst*	
StepperDevice	*Int1msLst*		
TimerDevice	*Int1msLst*	*Int10msLst*	

Each *interrupt scheduler list* must be large enough to accommodate:

- All the interrupt handlers installed by the device drivers specified
- Any additional interrupt handlers installed by the user application

To satisfy this:

List Size >= Maximum Number of Interrupt Handlers + 2

Example:
Casino Board with all three DUART drivers and Cremson Video driver installed.
Total of four interrupts added to IntDuartLst.

(*config.c*)

```
        void  (*IntDuartLst[ ] )  ( ) = { 0, 0, 0, 0, 0, EOF_FUNCTIONS};      /*  4 interrupts therefore List size=  6 entries*/
```

2.2 Device Manager

The device manager module offers a number of functions for manipulating device drivers in a standard way. This module also deals with system shutdown in the event of errors. Most of the functions in this module are for system use and will be of no interest to the game programmer.

2.2.1 InstallDevices

BYTE InstallDevices(const INSTALLDEVICE* dlist)

2.2.1.1 Description

This function installs the device drivers in the given list. Each entry in this list contains four elements.

2.2.1.1.1 Device

Address of the device driver structure to be installed

2.2.1.1.2 Device Select Function

Pointer to the device select function. This function returns TRUE if the device is to be installed. If this field is zero the device is installed.

2.2.1.1.3 Configuration Select Function

Pointer to the configuration select function. The device driver structure contains a pointer to an array of configuration structures for the device. This function is responsible for calculating the index into this array and thereby selecting the configuration data to be used. If this field is zero configuration index zero is used.

2.2.1.1.4 Optional Parameter

This is an optional parameter, which is passed into the configuration select function. It will typically be some parameter that the configuration select function uses to determine which index to return. (E.g. a list of DIL switches)

2.2.1.2 Parameters

2.2.1.2.1 Parameter 1 – dlist

System device list.

2.2.1.3 Return Value

Returns TRUE on success.
Return FALSE if one or more drivers have reported errors.

2.2.1.4 Example

InstallDevices(DeviceLst);

2.2.2 SetDeviceErrorMask

void SetDeviceErrorMask(ERRBIT bit, const DEVICE* dev)

2.2.2.1 Description

Set the given bits in the device error mask. This will cause the device to ignore the errors corresponding to these bits. The meaning of the bits varies from device to device. Refer to the device header file for the error bits for any particular device.

2.2.2.2 Parameters

2.2.2.2.1 Parameter 1 – bit

Bits corresponding to errors to be masked.

2.2.2.2.2 Parameter 2 - dev

Address of the device driver structure.

2.2.2.3 Return Value

None.

2.2.2.4 Example

Instruct the coin acceptor driver to ignore pulses which are below the valid minimum.
SetDeviceErrorMask(&CoinAccDevice, ERR_UNDERPULSE);

2.2.3 ClearDeviceErrorMask

void ClearDeviceErrorMask(ERRBIT bit, const DEVICE* dev)

2.2.3.1 Description

Clear the given bits in the device error mask. This will cause the device to detect the errors corresponding to these bits. The meaning of the bits varies from device to device. Refer to the device header file for the error bits for any particular device.

2.2.3.2 Parameters

2.2.3.2.1 Parameter 1 – bit

Bits corresponding to errors to be detected.

2.2.3.2.2 Parameter 2 - dev

Address of the device driver structure.

2.2.3.3 Return Value

None.

2.2.3.4 Example

Instruct the coin acceptor driver to detect pulses which are below the valid minimum and initiate an error condition.
ClearDeviceErrorMask(&CoinAccDevice, ERR_UNDERPULSE);

2.2.4 ShutdownSystem

void ShutdownSystem();

2.2.4.1 Description

Shutdown all devices and suspend mainline execution and 250ms interrupt. Other interrupts continue to run. The installed error handler will be called. If the error handler returns, and all devices are clear of errors, the system will be restored and mainline execution will recommence.

2.2.4.2 Parameters

None.

2.2.4.3 Return Value

None.

2.2.4.4 Example

ShutdownSystem();

2.2.5 SystemShutdown

void SystemShutdown();

2.2.5.1 Description

Test the overall system status

2.2.5.2 Parameters

None.

2.2.5.3 Return Value

Return TRUE if the system is in shutdown.
Return FALSE if the system is running normally.

2.2.5.4 Example

```
while(SystemShutdown ());        //wait for error condition to clear
```

2.2.6 InstallErrorHandler

```
void    InstallErrorhandler(void (*err)(const DEVICE* dev));
```

2.2.6.1 Description

The given function is installed as the system error handler.

2.2.6.2 Parameters

2.2.6.2.1 Parameter 1 – err

Pointer to the function to be installed as system error handler.

2.2.6.3 Example

```
InstallErrorHandler(ErrorHandler);          // set error handler
```

2.2.7 BuildTypeLists

```
void    BuildTypeLists ();
```

2.2.7.1 Description

This function scans the 'DeviceTypeBuildList' and using this information builds the device type lists that are undefined at compile time. A device type list is a list which contains a pointer to all the installed devices of a certain type. For example the 'AcceptorLst' contains all the acceptor devices installed on the system. In the case of devices which are detected on start-up or installed optionally dependant on switch settings these lists must be built at run time.

2.2.7.2 Parameters

None.

2.2.7.3 Return Value

None.

2.2.7.4 Example

```
BuildDeviceTypeLists();
```

2.3 Input

The Input module is a device driver controlling system inputs. There are thirty-two system inputs. In their inactive state they are pulled up to +5v and read by the software as '1' in this state. When the inputs are pulled down to ground they are read as '0'.

2.3.1 Test

```
BYTE    Test(const INPUT ip);
```

2.3.1.1 Description

The interrupt function reads the inputs every N* milliseconds and a change of state is registered only after three consecutive samples read the same.
If the multiplexed inputs driver (MXIDevice) is loaded in place of the standard inputs driver (InputsDevice) then this function can test inputs wired to one of the input strobes instead of zero volts. See examples.

* N is the Debounce sample time defined in the INPUTCFG structure at application level in config.c

2.3.1.2 Parameters

2.3.1.2.1 Parameter 1 - ip

Input to be tested. This will be one of the macros defining the Heber input numbers (IP0 – IP31).
Passing IPTRUE causes the function to return TRUE.
Passing IPFALSE causes the function to return FALSE.
Passing ACT1 | IP20 causes the input to be treated as active high. That is the function returns TRUE when the input is inactive (read as 1)

2.3.1.3 Return Value

Returns TRUE if the given input is active (read as 0). Otherwise returns FALSE.

2.3.1.4 Example

```
if(Test(IP20)) return(TRUE);
if(Test(ACT1 | IP20)) return(TRUE);
if(Test(MXI0 | IP20)) return(TRUE);                    // test the input IP20 with reference to input strobe 0
```

Which outputs are used as input strobes is defined in config.c in MxiStrobeLst[].

2.3.2 TestDir

BYTE TestDir(const INPUT ip);

2.3.2.1 Description

This function tests the I/O space directly and calculates its return value based on the instantaneous value read.

2.3.2.2 Parameters

2.3.2.2.1 Parameter 1 - ip

Input to be tested.
Passing IPTRUE causes the function to return TRUE.
Passing IPFALSE causes the function to return FALSE.
Passing ACT1 | IP20 causes the input to be treated as active high. That is the function returns TRUE when the input is inactive (read as 1)

2.3.2.3 Return Value

Returns TRUE if the given input is active (read as 0). Otherwise returns FALSE.

2.3.2.4 Example

```
if(TestDir(IP20)) return(TRUE);
if(TestDir (ACT1 | IP20)) return(TRUE);
```

2.3.3 CurrentSense

BYTE CurrentSense();

2.3.3.1 Description

Test the current sense circuitry.

2.3.3.2 Parameters

None.

2.3.3.3 Return Value

Returns TRUE if the current sense circuit is active. (I.e. current is flowing). This is used to test for the presence of electro-mechanical counters.

2.3.3.4 Example

```
if(CurrentSense()) return(TRUE);
```

2.4 Outputs

The output module is a device driver controlling system outputs. There are 64 open drain outputs, which are high impedance when inactive and pull down to ground when active. The status of all 64 outputs is maintained in Ram and written out to the I/O space every millisecond to give good noise immunity.

2.4.1 OutputOn

void OutputOn(OUTPUT op);

2.4.1.1 Description

Set the status of the given output to on (or active) in the status RAM. It is up to 1 millisecond before the output status is written out to the I/O space and the output pulls down to ground.

2.4.1.2 Parameters

2.4.1.2.1 Parameter 1 - op

Output to be asserted.

2.4.1.3 Return Value

None.

2.4.1.4 Example

OutputOn(OP24);

2.4.2 OutputOff

void OutputOff(OUTPUT op);

2.4.2.1 Description

Set the status of the given output to off (or inactive) in the status RAM. It is up to 1 millisecond before the output status is written out to the I/O space and the output goes into a high impedance state.

2.4.2.2 Parameters

2.4.2.2.1 Parameter 1 - op

Output to be negated.

2.4.2.3 Return Value

None.

2.4.2.4 Example

OutputOff(OP24);

2.5 Multiplex

The multiplex module is a device driver controlling the multiplexed lamps and seven segment displays. The Pluto 5 board has drives for 256 lamps in a 16x16 array and 32 7segment LED digits also in a 16x16 array with two digits on each strobe. Both arrays support variable brightness on a strobe by strobe basis. The variable brightness is achieved by dividing the duration the strobe is active into two periods, (period0 and period1) lamps may be set to be on in either both or no periods. The percentage of the total strobe time taken up by period0 may be varied and period 1 will automatically take the percentage not allocated to period0.

2.5.1 Lamp

void Lamp(const LAMP lp, const STATUS st);

2.5.1.1 Description

The given lamp is set to the given status.

2.5.1.2 Parameters

2.5.1.2.1 Parameter 1 - lp

The lamp to be operated on. There are macros defined to assist in defining the lamps. Either column and row numbers or plug and pin numbers for source and sinks can be used.

2.5.1.2.2 Parameter 2 - st

The status the lamp is to be set to. Possible values and their meanings are shown below.
ON sets the lamp on
OFF sets the lamp off
F1 sets the lamp to flash
F2 sets the lamp to flash in antiphase to F1
FF1 sets the lamp to fast flash (Fast flash is 4 times the rate of normal flash)
FF2 sets the lamp to fast flash in antiphase to FF2

(ON & PERIOD0) Set the lamp on for period0 only.
(ON & PERIOD1) Set the lamp on for period1 only.
N.B PERIOD0 and PERIOD1 may be used as above in combination with any defined status shown above.

2.5.1.3 Return Value

None.

2.5.1.4 Example

The example shows how to define a lamp on column0 and row0.

2.5.1.4.1 Lamp Definition

```
#define LABEL1          LAMP(LC0, LR0)
```
LABEL1 can now be used to refer to the lamp on column0 and row0

2.5.1.4.2 Lamp Processing

```
Lamp(LABEL1,F1);                    // Flashes lamp defined above
Lamp(LABEL1,FF1);                   // Fast flashes lamp defined above
Lamp(LABEL1,FF2 & PERIOD0);         // Fast flashes lamp defined above at reduced brightness as defined by period0
```

2.5.2 Lamp0

```
void    Lamp0(const LAMP lp, const STATUS st);
```

2.5.2.1 Description

Set the lamp to the given status operating only on period0. The period1 status of the lamp remains unchanged

2.5.2.2 Parameters

As function Lamp().

2.5.2.3 Return Value

None.

2.5.2.4 Example

Assume lamp to be OFF to begin with.

```
Lamp0(LABEL1,ON);   // Lamp is now on reduced brightness as defined by period0
Lamp(LABEL1,ON);    // Lamp is now on full brightness
Lamp0(LABEL1,OFF);  // Lamp is now on reduced brightness as defined by period1
```

2.5.3 Lamp1

```
void Lamp1(const LAMP lp, const STATUS st);
```

2.5.3.1 Description

Set the lamp to the given status operating only on period1. The period0 status of the lamp remains unchanged

2.5.3.2 Parameters

As function Lamp().

2.5.3.3 Return Value

None.

2.5.3.4 Example

Assume lamp to be OFF to begin with.

```
Lamp1(LABEL1,ON);    // Lamp is now on reduced brightness as defined by period1
Lamp(LABEL1,ON);     // Lamp is now on full brightness
Lamp1(LABEL1,OFF);   // Lamp is now on reduced brightness as defined by period0
```

2.5.4 SetFlashTime

void SetFlashTime(BYTE time)

2.5.4.1 Description

Set the global lamp/LED flash time to the given value.

2.5.4.2 Parameters

2.5.4.2.1 Parameter 1 – time

The time in 10ms units. This time represents the duration of one phase of the fast flash.

2.5.4.3 Return Value

None.

2.5.4.4 Example

```
SetFlashTime(5);
```

2.5.5 SetDefaultFlashTime

void SetDefaultFlashTime()

2.5.5.1 Description

Restore the global lamp/LED flash time to its default value.

2.5.5.2 Parameters

None.

2.5.5.3 Return Value

None.

2.5.5.4 Example

```
SetDefaultFlashTime();  // restore the default flash time
```

2.5.6 SetBrightness

void SetBrightness(WORD strobe, BYTE brightness)

2.5.6.1 Description

Set the lamp/LED brightness on the given strobe to the given value. The value should be in the range 0-255 although there are actually only 16 discrete levels. A value of 0 allocates 1/16 of the total strobe duration to period0 and 15/16 to period1.

2.5.6.2 Parameters

2.5.6.2.1 Parameter 1 – strobe

Strobe number whose brightness value is to be modified. The strobe number is in the range 0 to 15 (LC0 to LC15).

2.5.6.3 Return Value

None.

2.5.6.4 Example

SetBrightness(0,100); // Set brightness for strobe 0

2.5.7 SetDefaultBrightness

void SetDefaultBrightness()

2.5.7.1 Description

Set the lamp/LED brightness on all strobes to the default value.

2.5.7.2 Parameters

None.

2.5.7.3 Return Value

None.

2.5.7.4 Example

SetDefaultBrightness(); // restore the default brightness

2.5.8 LEDBits

Void LEDBits(DIGIT dig, UBYTE mask, STATUS st);

2.5.8.1 Description

The 'bits' are mapped to the segment of the digit such that segments corresponding to ones are set to status 'st' and segments corresponding to zeros are set OFF.

2.5.8.2 Parameters

2.5.8.2.1 Parameter 1 – dig

Digit to be operated on. There are macros defined to assist in defining the digits.

2.5.8.2.2 Parameter 2 – mask

Byte value to be mapped onto the seven-segment digit. Which segment of the actual digit corresponds to which bit in the mask is dependant on the wiring of the digits but it is suggested that the digits be wired according to the following table.

2.5.8.2.3 Parameter 3 – st

Status to set segments to. For information and possible values see function Lamp().

2.5.8.3 Return Value

None.

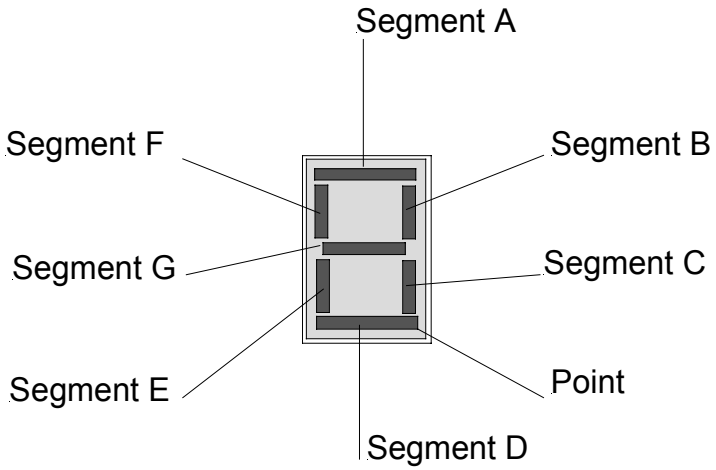
2.5.8.4 Example

SetBrightness(0,LampsCfg.brightness); // restore the default brightness

Table 1 - Mapping of bits to segments

Bit in Mask	LED0 - LED15	LED16 – LED31	Wire to Segment
Bit 0	SEG0 (P5-A9)	SEG8 (P5-A13)	Segment A
Bit 1	SEG1 (P5-B9)	SEG9 (P5-B13)	Segment B
Bit 2	SEG2 (P5-A10)	SEG10 (P5-A14)	Segment C
Bit 3	SEG3 (P5-B10)	SEG11 (P5-B14)	Segment D
Bit 4	SEG4 (P5-A11)	SEG12 (P5-A15)	Segment E
Bit 5	SEG5 (P5-B11)	SEG13 (P5-B15)	Segment F
Bit 6	SEG6 (P5-A12)	SEG14 (P5-A16)	Segment G
Bit 7	SEG7 (P5-B12)	SEG15 (P5-B16)	Point

Table 2 – Segment naming convention



2.6 Sampled Sound

The sampled sound module is a device driver controlling the sampled sound. There are two completely independent sound channels, which are identical, and each transfers ADPCM compressed sound data to the OKI sound chip using DMA. The sound ADPCM data is generated from .WAV files using the supplied 'Sndcnv32' utility which converts .WAV files into assembly source files suitable for the GNU assembler. For detailed information on generating sound source files see the documentation supplied with 'Sound Solutions'.

2.6.1 Playback

void Playback(const BYTE ch, const SAMPLE*const* tune, BYTE vol)

2.6.1.1 Description

Play the given tune 't' on the specified channel. Any sound currently playing on that channel is cut short.

2.6.1.2 Parameters

2.6.1.2.1 Parameter 1 – ch

Sound channel to play the tune on.
Passing 0 specifies channel 1
Passing 1 specifies channel 2

2.6.1.2.2 Parameter 2 – tune

Tune to be played on channel 1. A tune is defined as a list of samples.

2.6.1.2.3 Parameter 3 – vol

Any non-zero value supplied overrides the current default volume for this channel. Volume is in the range 0 to 255 where 255 is maximum volume and 0 specifies default volume is to be used.

2.6.1.3 Return Value

None.

2.6.1.4 Example

2.6.1.4.1 Defining a sample

Assuming the file SampleData1 has been generated using the 'Sndcnv32' utility a sample can be defined as follows.

```
extern const SAMDATA SampleData1;
const SAMPLE Sample1={16000,&SampleData1};
```

This specifies that the sample data 'SampleData1' generated by the 'Sndcnv32' utility should be played back at 16 kHz.

```
extern const SAMDATA SampleData2;
const SAMPLE Sample2={8000,&SampleData2};
```

This specifies that the sample data 'SampleData2' generated by the 'Sndcnv32' utility should be played back at 8 kHz.

2.6.1.4.2 Defining a tune

Assuming the above samples are defined.

```
const SAMPLE*const Tune1[]={&Sample1,&Sample2,EOF_SAMPLES};
```

This tune will play 'Sample1' immediately followed by 'Sample2'. The tune terminates after 'Sample2' finishes.

```
const SAMPLE*const Tune1[]={&Sample1,&Sample2,RST_SAMPLES};
```

This tune will play ‘Sample1’ immediately followed by ‘Sample2’. When ‘Sample2’ finishes the tune will restart.

```
const SAMPLE*const     Tune1[]={&Sample1,&Sample2,REWIND1};
```

This tune will play ‘Sample1’ immediately followed by ‘Sample2’. When ‘Sample2’ finishes ‘Sample2’ will be restarted.

2.6.1.4.3 Playing a tune

```
Playback(1,Tune1,0);          // Play Tune1 on channel 1 at the channel1 default volume .
Playback(2,Tune1,100);        // Play Tune1 on channel 2 at volume 100.
```

2.6.2 Quiet

```
void     Quiet(const BYTE chl)
```

2.6.2.1 Description

Silence the given sound channel.

2.6.2.2 Parameters

2.6.2.2.1 Parameter 1 – ch

Sound channel to silence.
Passing 0 specifies channel 1
Passing 1 specifies channel 2

2.6.2.3 Return Value

None.

2.6.2.4 Example

```
Quiet(0);          // Silences channel 1
Quiet(1);          // Silences channel 2
```

2.6.3 SetVolume

```
void     SetVolume(const BYTE ch, BYTE vol)
```

2.6.3.1 Description

Set the default volume for the given sound channel.

2.6.3.2 Parameters

2.6.3.2.1 Parameter 1 – ch

Sound channel to set volume for.
Passing 0 specifies channel 1
Passing 1 specifies channel 2

2.6.3.2.2 Parameter 2 – vol

Value supplied becomes the default volume for this channel. Volume is in the range 0 to 255 where 0 is minimum volume and 255 is maximum volume.

2.6.3.3 Return Value

None.

2.6.3.4 Example

```
SetVolume(0,0);          // Sets channel 1 default volume to 0
SetVolume(1,100);        // Sets channel 2 default volume to 100
```

2.6.4 SoundBusy

void SoundBusy(const BYTE ch)

2.6.4.1 Description

Test the status of the given sound channel.

2.6.4.2 Parameters

2.6.4.3 Parameter 1 – ch

Sound channel to set volume for.
Passing 0 specifies channel 1
Passing 1 specifies channel 2

2.6.4.4 Return Value

Return TRUE if the channel is currently playing a tune. Otherwise return FALSE.

2.6.4.5 Example

```
if(SoundBusy(0)) return;          // Returns if channel 1 is currently playing
```

2.7 Serial Communications

The serial communications module contains wrapper functions which give access to the services of the various serial communications device drivers.

2.7.1 ConfigureSerial

Void ConfigureSerial(const DEVICE* dev,BYTE channel,BYTE txbaud,BYTE rxbaud,BYTE data,BYTE stop,BYTE parity,BYTE txhs ,BYTE rxhs);

2.7.1.1 Description

Configure transmitter and receiver of the specified port with the given settings.

2.7.1.2 Parameters

2.7.1.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used
Available devices are:

SerialDevice (Standard Pluto5 DUART)
DUART0Device (Pluto 5 Casino only)
DUART1Device (Pluto 5 Casino only)
DUART2Device (Pluto 5 Casino only)
UARTDevice (Calypso 16 UART)

2.7.1.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.1.2.3 Parameter 3 – txbaud

Baud rate for the transmitter. The following are valid values for this parameter.

BAUD50
BAUD75
BAUD110
BAUD134
BAUD150
BAUD200
BAUD300
BAUD600
BAUD1050

BAUD1200
BAUD1800
BAUD2000
BAUD2400
BAUD4800
BAUD7200
BAUD9600
BAUD19200
BAUD38400
BAUD76800 (Pluto 5 DUART “SerialDevice” only)

2.7.1.2.4 Parameter 3 – rxbaud

Baud rate for the receiver. Permitted values are as for the transmitter.

2.7.1.2.5 Parameter 4 – data

The number of data bits. The following are permitted values.
BITS6
BITS7
BITS8

2.7.1.2.6 Parameter 5 – stop

The number of stop bits. The following are permitted values.
STOP1
STOP2

2.7.1.2.7 Parameter 6 – parity

Parity setting for the port. The following are permitted values.
PARITYEVEN
PARITYODD
PARITYNONE

2.7.1.2.8 Parameter 7 – txhs

Handshake type for the transmitter. The following are permitted values.
TXRTS
TXCTS
TXNONE

2.7.1.2.9 Parameter 8 – rxhs

Handshake type for the receiver. The following are permitted values.
RXRTS
RXCTS
RXNONE

2.7.1.3 Return Value

Returns TRUE if the port was configured successfully.

2.7.1.4 Restrictions

The two channels within a given serial device can be set to different baud rates.

If looking at the datasheet for the DUART SCC68692 p16. The two channels of the duart or serial device are using the same tables to be configured.

As table 1 is read first, it has priority in setting up the baud rate. For this reason, it should be noted that **certain combinations of baud rates are not allowed**: due to a feature of the DUART IC devices used (see list on next page)

50	75
200	150
1050	1800
7200	2000
9600	19200

It is **not** possible to set any baud rate selected from the first column in combination with any baud rate selected from the second column

For example, the following combination cannot be used within a single serial device:
channel 1 = 7200, channel 0 = 150 (Not allowed)

2.7.1.5 Example

The following function call will configure channel 0 of the SerialDevice (68340 serial module)
ConfigureSerial(&SerialDevice,0,BAUD9600,BAUD9600,BITS8,STOP1,PARITYNONE,TXNONE,RXNONE);

2.7.2 TxChar

BYTE TxChar(const DEVICE* dev, BYTE channel, BYTE ch);

2.7.2.1 Description

Transmit the character from the given port.

2.7.2.2 Parameters

2.7.2.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.2.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.2.2.3 Parameter 3 – ch

Character to be transmitted.

2.7.2.3 Return Value

Return TRUE if the character has been placed in the transmit buffer.
Return FALSE if the buffer is full.

2.7.2.4 Example

```
while(!TxChar(&SerialDevice,0,'A')); // wait until there is room in the tx buffer then put 'A' into buffer
```

2.7.3 TxString

void TxString(const DEVICE* dev, BYTE channel, const BYTE* s);

2.7.3.1 Description

Transmit the character string from the given port. If necessary the function will wait for space in the transmit buffer before recommencing.

2.7.3.2 Parameters

2.7.3.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.3.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.3.2.3 Parameter 3 – s

Address of the character string to be transmitted.

2.7.3.3 Return Value

None.

2.7.3.4 Example

TxString(&SerialDevice,0,"transmit this string");

2.7.4 RxChar

WORD RxChar(const DEVICE* dev, BYTE channel);

2.7.4.1 Description

Get a character from the receive buffer of the given port.

2.7.4.2 Parameters

2.7.4.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.4.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.4.3 Return Value

Return the character received or –1 if the buffer is empty.

2.7.4.4 Example

```
WORD ch;
while((ch=RxChar(&SerialDevice,0))<0);          // wait for character to arrive
```

2.7.5 TxEmpty

BYTE TxEmpty(const DEVICE* dev, BYTE channel);

2.7.5.1 Description

Test the status of the transmit buffer for the given port.

2.7.5.2 Parameters

2.7.5.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.5.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.5.3 Return Value

Return TRUE if the buffer is empty.
Return FALSE if there are characters awaiting transmission.

2.7.5.4 Example

```
While(!TxEmpty(&SerialDevice,0));       // wait for characters to go
```

2.7.6 RxEmpty

BYTE RxEmpty(const DEVICE* dev, BYTE channel);

2.7.6.1 Description

Test the status of the receive buffer for the given port.

2.7.6.2 Parameters

2.7.6.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.6.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.6.3 Return Value

Return TRUE if the buffer is empty.
Return FALSE if there are characters waiting.

2.7.6.4 Example

```
while(!RxEmpty(&SerialDevice,0));        // wait for character to arrive
```

2.7.7 ClearTx

```
void    ClearTx(const DEVICE* dev, BYTE channel);
```

2.7.7.1 Description

Clear the transmit buffer for the specified port.

2.7.7.2 Parameters

2.7.7.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.7.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.7.3 Return Value

None.

2.7.7.4 Example

```
ClearTx(&SerialDevice,0);        // Abort transmission
```

2.7.8 ClearRx

```
void    ClearRx(const DEVICE* dev, BYTE channel);
```

2.7.8.1 Description

Clear the receive buffer for the specified port.

2.7.8.2 Parameters

2.7.8.2.1 Parameter 1 – dev

Address of the device driver structure for the serial device to be used

2.7.8.2.2 Parameter 2 – channel

Port within the device to be used.

2.7.8.3 Return Value

None.

2.7.8.4 Example

```
ClearRx(&SerialDevice,0);        // Clear buffer ready for data to be received
```

2.7.9 Calypso UART

A simple UART is implemented in the FPGA on the Calypso 16 Video Board.
All the standard serial functions described above are implemented.

There are some restrictions on the configuration parameters:

Void ConfigureSerial(const DEVICE dev,BYTE channel,BYTE txbaud,BYTE rxbaud,BYTE data,BYTE stop,BYTE parity,BYTE txhs ,BYTE rxhs);*

2.7.9.1 Parameters

2.7.9.1.1 Parameter 1 – dev

dev=UARTDevice

2.7.9.1.2 Parameter 2 – channel

channel=0 only

2.7.9.1.3 Parameter 3 – txbaud

The baud rate for the transmitter. The following are valid values for this parameter.
BAUD1200
BAUD2400
BAUD4800
BAUD9600

2.7.9.1.4 Parameter 3 – rxbaud

The baud rate for the receiver. Must be identical to txbaud
rxbaud=txbaud

2.7.9.1.5 Parameter 4 – data

The number of data bits = BITS8 only

2.7.9.1.6 Parameter 5 – stop

The number of stop bits = STOP1 only

2.7.9.1.7 Parameter 6 – parity

The parity setting for the port = PARITYNONE only

2.7.9.1.8 Parameter 7 – txhs

The handshake type for the transmitter is not supported (txhs=0 only)

2.7.9.1.9 Parameter 8 – rxhs

The handshake type for the transmitter is not supported (rxhs=0 only)

2.7.9.2 Return Value

None.

2.8 Vacuum fluorescent display

The VFD module is a service module providing low level functions for addressing vacuum fluorescent displays. The system is capable of driving two VFDs from the TTL connector P12. Which VFD signals are driven from which pins on the connector is defined in two structures, vfd1 and vfd2 found in config.c in the game directory.

2.8.1 TxVFD1

Void TxVFD1(BYTE ch);

2.8.1.1 Description

Transmit the given character to VFD1

2.8.1.2 Parameters

2.8.1.2.1 Parameter 1 – ch

Character to be sent to VFD1.

2.8.1.3 Return Value

None.

2.8.1.4 Example

```
TxVFD1(0x01); // Send the byte value 0x01 to VFD1
```

2.8.2 ResetVFD1

void ResetVFD1();

2.8.2.1 Description

Reset the display. The reset line is asserted for the time defined in the vfd1 structure in config.c.

2.8.2.2 Example

```
ResetVFD1();  // Reset the display
```

2.8.3 TxVFD2

Void TxVFD2(BYTE ch);

2.8.3.1 Description

Transmit the given character to VFD2.

2.8.3.2 Parameters

2.8.3.3 Parameter 1 – ch

Character to be sent to VFD2.

2.8.3.4 Return Value

None.

2.8.3.5 Example

```
TxVFD2(0x01); // Send the byte value 0x01 to VFD2
```

2.8.4 ResetVFD2

void ResetVFD2();

2.8.4.1 Description

Reset the display. The reset line is asserted for the time defined in the vfd2 structure in config.c.

2.8.4.2 Parameters

None.

2.8.4.3 Return Value

None.

2.8.4.4 Example

ResetVFD2(); // Reset the display

2.9 I2C Bus

The I2C module contains wrapper functions, which call the appropriate functions from the particular I2C driver passed into the function. There are 3 different I2C drivers available I2CintDevice, I2Cextdevice, and I2CPICDevice. This allows device drivers to be written for particular I2C devices and the I2C driver to be used to communicate with the device can be specified in the device configuration structure found in config.c.

2.9.1 I2CRead

BYTE I2CRead(const DEVICE *i2c,BYTE devaddr,BYTE intaddr,BYTE *dst,UBYTE bytecount)

2.9.1.1 Description

Reads ‘bytecount’ from the device on the I2C bus to the destination address ‘dst’.

2.9.1.2 Parameters

2.9.1.2.1 Parameter 1 – I2C

Pointer to the device driver which will control the I2C bus transactions required to perform the read of the device. This will be &I2CintDevice, &I2Cextdevice, or &I2CPICDevice.

2.9.1.2.2 Parameter 2 – devaddr

Address of the device on the I2C bus.

2.9.1.2.3 Parameter 3 – intaddr

Address within the device at which the read will start.

2.9.1.2.4 Parameter 4 – dst

Destination address for the data. The data is copied from the device on the I2C bus to this location.

2.9.1.2.5 Parameter 5 – bytecount

Number of bytes to be read form the device.

2.9.1.3 Return Value

Returns TRUE on success.
Returns FALSE if the specified number of bytes has not been read.

2.9.1.4 Example

Read 8 bytes from internal address 0x10 of a device at address 0xa6 on the external I2C bus and place them into the variable ‘bytearray’.

```
If(I2CRead(&I2CExtDevice,0xa6,0x10,bytearray,8))
{
        Process the valid data in bytearray
}
else
{
        Initiate error processing
}
```

2.9.2 I2CWrite

BYTE I2CWrite(const DEVICE *i2c,const BYTE* src,UBYTE bytecount,BYTE devaddr,BYTE intaddr)

2.9.2.1 Description

Write data from the source address ‘src’ to a device on the I2C bus at device address ‘devaddr’. The data will be ‘bytecount’ bytes long and be written to the device internal address ‘intaddr’

2.9.2.2 Parameters

2.9.2.2.1 Parameter 1 – i2c

Pointer to the device driver which will control the I2C bus transactions required to perform the write to the device. This will be &I2CintDevice, &I2Cextdevice, or &I2CPICDevice.

2.9.2.2.2 Parameter 2 – src

Source address of the data. The data is copied from this address to the device on the I2C bus.

2.9.2.2.3 Parameter 3 – bytecount

Number of bytes to be written to the device.

2.9.2.2.4 Parameter 4 – devaddr

Address of the device on the I2C bus.

2.9.2.2.5 Parameter 5 – intaddr

Address within the device at which the write will start.

2.9.2.3 Return Value

Returns TRUE on success.
Returns FALSE if the specified number of bytes have not been written.

2.9.2.4 Example

Write 8 bytes from the variable ‘bytearray’ to the internal address 0x10 of a device at address 0xa6 on the external I2C bus.

```
if(I2CWrite(&I2CExtDevice, bytearray,8, 0xa6,0x10))
{
        Process the valid data in bytearray
}
else
{
        Initiate error processing
}
```

2.10 ATA Driver

This module provides low level functions for control and read-only access to ATA compliant external devices connected to the IDE expansion connector on the Calypso 16 Video Board.

Examples of such devices are:
- IDE format **Hard Drive**
- **Compact Flash Card**, using the ***Heber Compact Flash Adaptor Board***

External IDE devices should be configured as **IDE MASTER** devices. This is the default setting with the Heber Compact Flash Adaptor, and is normally a link option with IDE format Hard Drives.

When the drive is initialised, files on the drive will normally be accessed using functions provided by the **FAT32 Filesystem Driver** (See later section 4.3)

2.10.1 InitialiseHDD

ATASTS InitialiseHDD(const DEVICE *dev)

2.10.1.1 Description

Select and initialise the drive.

2.10.1.2 Parameters

2.10.1.2.1 Parameter1 – dev

Pointer to the device driver which will control the IDE bus transactions required to access the device. This will be **&ATADevice.**

2.10.1.3 Return Value

Returns zero on success. Returns a non-zero value after a timeout if no drive is detected, or if the drive fails to initialise. In case the function does not return a non-zero value it is recommended to check that the IDE cable is plugged correctly, power off the board and power it on again.

2.10.1.4 Example

```
if (InitialiseHDD(&ATADevice) != 0)
        TxString(&SerialDevice,1,"\r\nInitialise Drive: Timeout...");
```

2.10.2 ReadDriveID

ATASTS ReadDriveID(const DEVICE *dev)

2.10.2.1 Description

Read drive and initialise *drive_info* structure with information regarding the drive.
This structure can be accessed with the following functions.

2.10.2.2 Parameters

2.10.2.2.1 Parameter1 – dev

Pointer to the device driver which will control the IDE bus transactions required to access the device. This will be **&ATADevice.**

2.10.2.3 Return Value

Returns zero on success. Returns a non-zero value after a timeout if no drive is detected, or if the drive fails to initialise.

2.10.2.4 Example

```
if (ReadDriveID(&ATADevice) != 0)
        TxString(&SerialDevice,1,"\r\nIdentify Drive: Timeout...");
```

2.10.3 GetDriveModel, GetDriveRev, GetDriveSerialno, GetDriveCyls, GetDriveHeads, GetDriveSectors

```
UBYTE       *GetDriveModel(const DEVICE *dev)
UBYTE       *GetDriveRev(const DEVICE *dev)
UBYTE       *GetDriveSerialno(const DEVICE *dev)
UWORD       GetDriveCyls(const DEVICE *dev)
UWORD       GetDriveHeads(const DEVICE *dev)
UWORD       GetDriveSectors(const DEVICE *dev);
```

2.10.3.1 Description

These functions return the following values from the *drive_info* structure:
drive model string
drive revision number string
drive serial number string
number of drive cylinders
number of drive heads
number of drive sectors

2.10.3.2 Parameters

2.10.3.2.1 Parameter1 – dev

Pointer to the device driver which will control the IDE bus transactions required to access the device. This will be **&ATADevice.**

2.10.3.3 Return Values

Return values are as above.

2.10.3.4 Example

```
sprintf(str, "\r\nModel: %s\r\nRev: %s\r\nSerial No: %s\r\nCylinders: %3d, Heads: %3d, Sectors: %3d\r\n",
        GetDriveModel(&ATADevice),
        GetDriveRev(&ATADevice),
        GetDriveSerialno(&ATADevice),
        GetDriveCyls(&ATADevice),
        GetDriveHeads(&ATADevice),
        GetDriveSectors(&ATADevice));
TxString(&SerialDevice,1,str);     //display all above info on serial channel 1
```

2.10.4 DriveSpinup, DriveSpindown

```
ATASTS      DriveSpinup(const DEVICE *dev)
ATASTS      DriveSpindown(const DEVICE *dev)
```

2.10.4.1 Description

These two functions should be used when the device is a hard drive. These functions should initialise the motor to spin.
Spin up drive motor (IDE Hard Drive only)
Spin down drive motor (IDE Hard Drive only)

2.10.4.2 Parameters

2.10.4.2.1 Parameter1 – dev

Pointer to the device driver which will control the IDE bus transactions required to access the device. This will be **&ATADevice.**

2.10.4.3 Return Value

Returns zero on success. Returns a non-zero value after a timeout on failure.

2.10.5 ReadSector

ATASTS ReadSectors(const DEVICE *dev, ULONG lba, UWORD *buffer, UBYTE count)

2.10.5.1 Description

Read *count* sectors of data from drive, starting at linear block address *lba* to system memory at *buffer.*

2.10.5.2 Parameters

2.10.5.2.1 Parameter1 – dev

Pointer to the device. This will be **&ATADevice.**

2.10.5.2.2 Parameter2 – lba

Linear Block Address of the start sector of the block of sectors to be transferred from the drive.

2.10.5.2.3 Parameter3 – buffer

Pointer to memory address to which the data is transferred. This can be in System RAM or Video RAM

2.10.5.2.4 Parameter4 – count

Number of consecutive sectors of data to be transferred. Each sector contains 512 bytes of data.

2.10.5.3 Return Value

Returns zero on success. Returns a non-zero value after a timeout on failure.

2.11 Enhanced Calypso Driver

This is a new improved graphics driver for the Calypso 16 Graphics Board. It is intended as a replacement for the existing Video Driver in the Peripheral Driver Layer.

The Enhanced Calypso Driver has the following improvements over the old video driver:

1. Display list handling: Graphics command lists are executed independently of the CPU
2. Layer-centric commands: Graphics commands include destination display layer as parameter
3. Color depth issues handled automatically.
4. Simplified and cleaner API.
5. New feature: Alpha blending.
6. New feature: UNICODE character support.
7. New feature: Animation sequences.
8. New feature: Enhanced 2D Line drawing (future release)

***Note*:** The enhanced calypso driver functions are non re-entrant. All graphics functions must be called from the main loop, and may not be called from interrupt routines.

2.11.1 General description

2.11.1.1 Display list handling:

The Cremson display controller uses *display lists* A display list is a sequential list of graphics commands and parameters. It is placed in video memory by the CPU and accessed and executed independently by the Cremson device

Performance and structure have been improved by separating display list creation from *execution*
A major benefit of this structure is that all API function calls are non-blocking:

OLD DRIVER: CremsonDriver

```
GraphicsFunction(..)
{
        ...
        createdisplaylist(..)
        rundisplaylist(..)          /* BLOCKING - returns on completion*/
```

```
}
```

ENHANCED DRIVER: CalypsoDriver

```
GraphicsFunction(..)
{
        ...
        addtodisplaylist(..)        /* NON-BLOCKING */
}

VerticalInterrupt()
{
        ...
        rundisplaylist(..)
}
```

Multiple display lists are used. The driver cycles through the display lists, executing one display list every video frame. The display list currently being *written* to lags behind the display list currently being *executed.*

There is a fixed delay of two video frames between graphics function calls and their execution.

2.11.1.2 Layer-centric commands

The existing API requires the game coder to explicitly keep track of memory addresses

The new API eliminates this by making all graphics functions layer centric. All blitting/drawing functions have a parameter defining which display layer is being written to.

A simple example is blitting a sprite image to a display layer:

OLD DRIVER: CremsonDriver

```
        Initialise MIDDLE LEFT layer to memory address A
        ...
        Put sprite to address A
```

ENHANCED DRIVER: CalypsoDriver

```
        Initialise MIDDLE LEFT layer to memory address A
        ...
        Put sprite to MIDDLE LEFT layer
```

This simplifies user code.

2.11.1.3 Colour depth issues

A major benefit of layer-centric commands is that *they remove the need to specify colour depths within user code* (other than at configuration/initialisation)

Colour depth is a defined property of the display layers. If the blitting and drawing commands specify which display layer is being written to, the correct version of the function can be run depending on the known colour depth of that layer.

2.11.1.4 Alpha Blending

API functions have been added to support alpha blending **between the console layer and other layers**. This allows for attractive fading and transparency effects.

2.11.1.5 Unicode character support (future release)

This will allow text display of character sets such as Cyrillic and Greek

This will include:

- support for creation of and loading of fonts covering specified UNICODE character ranges.

- support for displaying UTF16 encoded Unicode characters.

2.11.1.6 Animation Sequence (future release)

An animation sequence is any defined succession of graphics images

Examples are moving sprites, and video clips.

The existing driver requires the developer to explicitly write code to handle timing, double buffering and synchronisation with vertical interrupt.

The enhanced driver will allow all parameters of an animation sequence to be specified in advance. The animation can then be started with a single command: all screen updates will be taken care of automatically by the driver.

This will allow the game developer to simplify his animation code:

Example:

DefineAnimation(EyeCandy, parameters...)
...
RunAnimation(EyeCandy)

2.11.1.7 Enhanced 2D Line drawing(future release)

The existing driver uses software algorithms for 2D line drawing.

The new driver will use the 2D line drawing capabilities of the Cremson device, Support for anti-aliasing, dotted lines and variable line widths will be included.

2.11.2 Installation of the CalypsoDriver

To use this new driver, it should be added to the device install list in *devices.c* in the project directory:

```
const INSTALLDEVICE DeviceList[]=
{
...
/* {&CremsonDevice,0,0,0},  old video driver MUST be commented out as the two drivers cannot work together. */
{&CalypsoDevice,0,0,0},
...
{0,0,0}
};
```

A configuration entry is also required in *config.c* in the project directory:

```
const CALYPSOCFG CalypsoCfg[]=
{
{Calypso800x600,palette},
/* {Calypso640x480,palette}, */
/* {Calypso1024x768,palette}, */
0
};
```

2.11.3 CalypsoInitDisplayMode( )

void CalypsoInitDisplayMode(const WORD *reg_table);

2.11.3.1 Description

Initialise video display resolution

2.11.3.2 Parameters

2.11.3.2.1 Parameter – reg_table
Allowed values:

Calypso1024x768
Calypso800x600
Calypso640x480

2.11.3.3 Return Value
None.

2.11.3.4 Example
CalypsoInitDisplayMode(Calypso800x600);

2.11.4 CalypsoVideoOn( )
void CalypsoVideoOn(void);

2.11.4.1 Description
Turn on video output.

2.11.4.2 Return Value
None.

2.11.4.3 Example
CalypsoVideoOn();

2.11.5 CalypsoVideoOff( );

void CalypsoVideoOff(void);

2.11.5.1 Description
Turn off video output.
Note: Preserves layer enable settings.

2.11.5.2 Parameters
None.

2.11.5.3 Return Value
None.

2.11.5.4 Example
CalypsoVideoOff();

2.11.6 CalypsoGetVsync( )
UBYTE CalypsoGetVsync(void);

2.11.6.1 Description
Returns vertical sync.

2.11.6.2 Parameters
None.

2.11.6.3 Return Value
Returns TRUE if vertical refresh in progress.

2.11.6.4 Example

If (CalypsoGetVsync())
/ do something */*

2.11.7 CalypsoSetVideoMemory( )

ULONG CalypsoSetVideoMemory(LONG video_addr, LONG size_bytes, BYTE color_mode, UWORD color);

2.11.7.1 Description

Block fill video memory.

2.11.7.2 Parameters

2.11.7.2.1 Parameter – video_addr

Start address (offset from start of video memory)

2.11.7.2.2 Parameter – size_bytes

Size in bytes to set –must be multiple of 32 bytes

2.11.7.2.3 Parameter – color_mode

DIRECT_COLOR (16 bit)
INDIRECT_COLOR (8 bit)

2.11.7.2.4 Parameter – color

colour value (8 or 16 bit)

2.11.7.3 Return Value

Next address after block fill.

2.11.7.4 Example

CalypsoSetVideoMemory(0, 0x1000000, INDIRECT_COLOR, BLUE);
CalypsoSetVideoMemory(0, 0x1000000, DIRECT_COLOR, 0x7FFF);

2.11.8 CalypsoInitLayer( )

LONG CalypsoInitLayer(CALYPSOLAYER *layer, BYTE color_mode, BYTE buffer_mode, WORD transparent_color, WORD width_pixels, WORD height, LONG start_address);

2.11.8.1 Description

Initialise Cremson Display Layer.

2.11.8.2 Parameters

2.11.8.2.1 Parameter – layer

pointer to Display Layer:

&CalypsoLayer[BL_LAYER]	base left layer
&CalypsoLayer[BR_LAYER]	base right layer
&CalypsoLayer[ML_LAYER]	middle left layer
&CalypsoLayer[MR_LAYER]	middle right layer
&CalypsoLayer[W_LAYER]	window layer
&CalypsoLayer[C_LAYER]	console layer

2.11.8.2.2 Parameter – color_mode

Display Layer colour mode:

DIRECT_COLOR	16 bit colour
INDIRECT_COLOR	8 bit colour

2.11.8.2.3 Parameter – buffer_mode

Display Layer buffer mode:

SINGLEBUFFER single buffer mode
DOUBLEBUFFER double buffer mode

This parameter is ignored for console and window layers, which are not double buffered.

2.11.8.2.4 Parameter – transparent_color

Transparent colour (8 or 16 bit)

Display Layer transparent colour. Any pixel in the layer with this colour is *transparent:*
Lower priority layers are visible at this point. The priority order of Display Layers is:

console > middle > base

The *window* layer is not transparent.

2.11.8.2.5 Parameter – width_pixels

Display Layer width in pixels.

MANDATORY:
width_pixels >= display horizontal size
width_pixels must be multiple of 64

2.11.8.2.6 Parameter – height

Display Layer height in lines

2.11.8.2.7 Parameter – start_address

Display Layer start address (offset from start of video memory)

2.11.8.3 Return Value

Returns next free start address.
This allows sequential mapping of video memory to Display Layers.

2.11.8.4 Example

```
/* Initialise BASE  then MIDDLE  then CONSOLE Layers */
base_start    = 0;

middle_start = CalypsoInitLayer(&CalypsoLayer[BL_LAYER],
                        DIRECT_COLOR, DOUBLEBUFFER, 0x0000, H_SIZE, V_SIZE, base_start);

console_start  = CalypsoInitLayer(&CalypsoLayer[ML_LAYER],
                        INDIRECT_COLOR, DOUBLEBUFFER, BLACK, H_SIZE, V_SIZE, middle_start);

offscreen_start  = CalypsoInitLayer(&CalypsoLayer[C_LAYER],
                        INDIRECT_COLOR, SINGLEBUFFER, BLACK, H_SIZE, V_SIZE, console_start);
```

2.11.9 CalypsoEnableLayer( );

void CalypsoEnableLayer(UBYTE select);

2.11.9.1 Description

Enable (make visible) Display Layer

2.11.9.2 Parameters

2.11.9.2.1 Parameter – select

Select layer:

BASE_SELECT
MIDDLE_SELECT
CONSOLE_SELECT
WINDOW_SELECT
ALL_SELECT

2.11.9.3 Return Value

None.

2.11.9.4 Example

```
CalypsoEnableLayer(BASE_SELECT);                 // Make base layer visible
CalypsoEnableLayer(BASE_SELECT | MIDDLE_SELECT); // Make base  and middle layers visible
CalypsoEnableLayer(ALL_SELECT);                  // Make all layers visible
```

2.11.10 CalypsoDisableLayer( )

void CalypsoDisableLayer(UBYTE select);

2.11.10.1 Description

Disable (make invisible) Display Layer

2.11.10.2 Parameters

2.11.10.2.1 Parameter – select

Select layer:
BASE_SELECT
MIDDLE_SELECT
CONSOLE_SELECT
WINDOW_SELECT
ALL_SELECT

2.11.10.3 Return Value

None.

2.11.10.4 Example

```
CalypsoDisableLayer(CONSOLE_SELECT);                 // make console layer invisible
CalypsoDisableLayer(BASE_SELECT | MIDDLE_SELECT);    // make base  and middle layers invisible
```

2.11.11 CalypsoScrollLayerX( )

void CalypsoScrollLayerX(CALYPSOLAYER *layer, ULONG offset);

2.11.11.1 Description

Scroll Display Layer in x-direction with wrap around.

2.11.11.2 Parameters

2.11.11.2.1 Parameter – layer

Display Layer

2.11.11.2.2 Parameter – offset

x-offset in pixels between top left of screen and start address of Display Layer.

2.11.11.3 Return Value

None.

2.11.11.4 Example

/* Continuous scroll at one pixel every 10ms */

```
while(1) {
        CalypsoScrollLayerX(&CalypsoLayer[C_LAYER],  x% H_SIZE);
        Wait(1);
}
```

2.11.12 CalypsoScrollLayerY( )

void CalypsoScrollLayerY(CALYPSOLAYER *layer, ULONG offset);

2.11.12.1 Description

Scroll Display Layer in y-direction with wrap around.

2.11.12.2 Parameters

2.11.12.2.1 Parameter – layer

Display Layer

2.11.12.2.2 Parameter – offset

y-offset in lines between top left of screen and start address of Display Layer

2.11.12.3 Return Value

None.

2.11.12.4 Example

/* Continuous scroll at one line per 10ms */

```
while(1) {
        CalypsoScrollLayerY(&CalypsoLayer[C_LAYER],  y% V_SIZE);
        Wait(1);
}
```

CalypsoClearLayer( )

WORD CalypsoClearLayer(CALYPSOLAYER *layer, BYTE doublebuffer_frame);

2.11.12.5 Description

Clear Display Layer to BLACK (colour 0x0000)

2.11.12.6 Parameters

2.11.12.6.1 Parameter – layer

Display Layer

2.11.12.6.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.12.7 Example

/* Clear both double buffer frames of Base Left Layer */
CalypsoClearLayer(&CalypsoLayer[BL_LAYER], DOUBLEBUFFER_FRAME0);
CalypsoClearLayer(&CalypsoLayer[BL_LAYER], DOUBLEBUFFER_FRAME1);

2.11.13 CalypsoFillLayer( )

WORD CalypsoFillLayer(CALYPSOLAYER *layer, BYTE doublebuffer_frame, UWORD color, UBYTE mode);

2.11.13.1 Description

Fill Display Layer with 8 or 16 bit colour.

2.11.13.2 Parameters

2.11.13.2.1 Parameter – layer

Display Layer

2.11.13.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.13.2.3 Parameter – color

Fill colour.
The colour depth (8-bit or 16-bit) should match the Display Layer colour depth

2.11.13.2.4 Parameter – mode

Fill mode:
BLT_COPY - overwrite current pixels
BLT_AND - logical AND with current pixels
BLT_OR - logical OR with current pixels

2.11.13.3 Return Value

None.

2.11.13.4 Example

/* Fill console layer (8-bit) with RED colour */
CalypsoFillLayer(&CalypsoLayer[C_LAYER], 0, RED, BLT_COPY);

2.11.14 CalypsoSetWindowLayer( )

void CalypsoSetWindowLayer(WORD x_position, WORD y_position, WORD width, WORD height);

2.11.14.1 Description

Set Window Layer parameters.

The Base, Middle and Console display Layers are full screen
The Window Display layer is different: its screen size and position can be controlled.

2.11.14.2 Parameters

2.11.14.2.1 Parameter – x_position

x position in pixels from screen left

2.11.14.2.2 Parameter – y_position

y position in lines from screen top

2.11.14.2.3 Parameter – width

Window width in pixels

2.11.14.2.4 Parameter – height

Window height in lines.

2.11.14.3 Example

/* 320x240 pixel window, top left at screen position x=50, y= 100 */
CalypsoSetWindowLayer(50, 100, 320, 240);

2.11.15 CalypsoSetFrameRate( )

void CalypsoSetFrameRate(BYTE rate);

2.11.15.1 Description

Set maximum frame update rate. When using *double-buffering,* the display output flips between the two double-buffer frames at this rate. This is the *maximum* frame rate: frame update will be delayed if display list command execution has not finished.

2.11.15.2 Parameters

2.11.15.2.1 Parameter – rate

frame rate in Hz:
FRAME_60HZ
FRAME_30HZ
FRAME_20HZ
FRAME_15HZ
FRAME_12HZ
FRAME_10HZ
FRAME_5HZ
FRAME_3HZ
FRAME_2HZ
FRAME_1HZ
FRAME_0HZ - manually advance frames using CalypsoFrameAdvance( )

2.11.15.3 Return Value

None.

2.11.15.4 Example

CalypsoSetFrameRate(FRAME_60HZ);

2.11.16 CalypsoEnableDoubleBuffer( )

void CalypsoEnableDoubleBuffer(void);

2.11.16.1 Description

Enable double buffering.

When double buffering is enabled, the Base Layer and Middle Layers will toggle the display output between their two double buffer frames at the frame rate set by CalypsoSetFrameRate( ).

2.11.16.2 Return Value

None.

2.11.16.3 Example

CalypsoSetFrameRate(FRAME_30HZ);
CalypsoEnableDoubleBuffer( );

2.11.17 CalypsoDisableDoubleBuffer( )

void CalypsoDisableDoubleBuffer(void);

2.11.17.1 Description

Disable double buffering.

2.11.17.2 Return Value

None.

2.11.17.3 Example

CalypsoDisableDoubleBuffer( );

2.11.18 CalypsoSetDoubleBufferFrame( )

void CalypsoSetDoubleBufferFrame(BYTE frame);

2.11.18.1 Description

When double buffering is enabled, flipping between two display frames is automatic. This function allows manual selection of the currently displayed double buffer frame. (BASE and MIDDLE layers).

2.11.18.2 Parameters

2.11.18.2.1 Parameter – frame

Display Layer double buffer frame:
DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.18.3 Return Value

None.

2.11.18.4 Example

SetDoubleBufferFrame(DOUBLEBUFFER_FRAME0);

2.11.19 CalypsoGetDoubleBufferFrame( )

BYTE CalypsoGetDoubleBufferFrame(void);

2.11.19.1 Description

Get the current non-displayed double buffer frame.

2.11.19.2 Return Value

0 - double buffer frame 0
1 - double buffer frame 1

2.11.20 CalypsoSetFrameNumber( )

void CalypsoSetFrameNumber(WORD n);

2.11.20.1 Description

Initialise Frame Number.

The Frame Number is a 16 bit counter that is incremented on every vertical display frame.
This counter can be used for timing and sequencing display events.

2.11.20.2 Parameters

2.11.20.2.1 Parameter – n

Frame number.

2.11.20.3 Return Value

None.

2.11.20.4 Example

```
/* clear frame number counter */
CalypsoSetFrameNumber(0);
```

2.11.21 CalypsoGetFrameNumber( )

WORD CalypsoGetFrameNumber(void);

2.11.21.1 Description

Read the current Frame Number.

2.11.21.2 Parameters

2.11.21.3 Return Value

Current Frame Number value (16 bit counter).

2.11.21.4 Example

```
CalypsoSetFrameRate(FRAME_30HZ);
CalypsoSetFrameNumber(0);

/* Wait for 30 frames = 1 second*/
while(CalypsoGetFrameNumber( )  < 30)
        ;
/* do something */
```

2.11.22 CalypsoFrameAdvance( )

void CalypsoFrameAdvance( )

2.11.22.1 Description

Advance to next frame on next vertical interrupt.

Used for advancing frames under software control when frame rate set to 0HZ.

Example:

```
/* animation */
while(1) {
        CalypsoFrameAdvance( );

        /* wait for next frame */
        frame_number = CalypsoGetFrameNumber();
        while (frame_number==CalypsoGetFrameNumber())
                ;

        /* clear sprites*/
        /* update sprite xy positions */
        /* draw sprites */
}
```

2.11.23 CalypsoScreenPartitionX( )

void CalypsoScreenPartitionX(WORD partition_xpos);

2.11.23.1 Description

Set screen X position of partition between left layer and right layer.

2.11.23.2 Parameters

2.11.23.2.1 Parameter – partition_xpos

partition_xpos - position in pixels from left of screen to partition.

2.11.23.3 Return Value

None.

2.11.23.4 Example

```
/* Left 25 % of screen: Base Left Layer        Right 75 % of screen: Base Right Layer */
CalypsoEnableLayer(BASE_ENABLE);
CalypsoScreenPartitionX(0.25 * H_SIZE);
```

2.11.24 CalypsoAlphaBlend ( )

void CalypsoAlphaBlend(BYTE enable)

2.11.24.1 Description

Enable /disabling console layer alpha blending.

2.11.24.2 Parameters

2.11.24.2.1 Parameter – enable

0 - Disable
1 - Enable

2.11.24.3 Return Value

None.

2.11.24.4 Example

See below.

2.11.25 CalypsoSetBlendRatio( )

void CalypsoSetBlendRatio(BYTE blend_ratio, BYTE select);

2.11.25.1 Description

Set blend ratio for **Alpha Blending** between Console Layer and Base/Middle/Window Layers.

Alpha blending is useful for *semi-transparency* and *fade* effects.

Alpha Blending works slightly differently for 8-bit colour and 16-bit colour layer.

Console Layer 16-bit Color :

Alpha blending is enabled on a pixel by pixel basis.
pixel bit 15 = 0 - pixel not alpha blended
pixel bit 15 = 1 - pixel alpha blended

Console Layer 8- bit Color :

Alpha blending is enabled depending on console palette table entries
palette table entry bit 15 = 0 - pixels of this colour not alpha blended
palette table entry bit 15 = 1 - pixel of this colour alpha blended

2.11.25.2 Parameters

2.11.25.2.1 Parameter – blend_ratio

blend_ratio - blend ratio in 16 steps:
0 (0/15 Console Layer, 15/15 Other Layers)
to
15 (15/15 Console Layer, 0/15 Other Layers)

2.11.25.2.2 Parameter – select

0 - blend ratio as above
1 - blend ratio reversed from above

2.11.25.3 Example

```
CalypsoAlphaBlend(ENABLE);
for (i=15; i>=0; i--) {
        CalypsoSetBlendRatio( i, 0);      /* slowly fade out console layer */
        Wait(10);
}
```

2.11.26 CalypsoLoadPalette( )

void CalypsoLoadPalette(UBYTE palette_select, const PALETTE_RGB *pal_table);

2.11.26.1 Description

Load palette table.

Palette source files can be generated from Image Alchemy(tm) format palette files using Heber utility *palconv.exe*

This will produce a source file *palette_name.c* containing a converted palette table.
This table is declared as an array of palette entry structures:

const PALETTE_RGB palette_name[256] = { 256 palette entry values… };

Where the palette entry type PALETTE_RGB is:

```
typedef struct palette_rgb
 {
        UBYTE   alpha;          /* 1= enable alpha blending for this colour (console layer only)*/
        UBYTE   red;            /* red value */
        UBYTE   green;          /* green value */
        UBYTE   blue;           /* rblue value */
} PALETTE_RGB;
```

2.11.26.2 Parameters

2.11.26.2.1 Parameter – palette_select

MIDDLE_BASE_LAYER_PALETTE - middle/base layer palette
CONSOLE_LAYER_PALETTE - console layer palette

2.11.26.2.2 Parameter – palette_table

Pointer to palette table.

2.11.26.3 Return Value

None.

2.11.26.4 Example

```
/* load palettes */
CalypsoLoadPalette(MIDDLE_BASE_LAYER_PALETTE, stdpalette);
CalypsoLoadPalette(CONSOLE_LAYER_PALETTE, stdpalette);
```

2.11.27 CalypsoGetPaletteEntry( )

void CalypsoGetPaletteEntry(UBYTE palette_select, UBYTE entry, PALETTE_RGB *rgb);

2.11.27.1 Description

Get palette entry from palette table.

2.11.27.2 Parameters

2.11.27.2.1 Parameter – palette_select

MIDDLE_BASE_LAYER_PALETTE - middle/base layer palette
CONSOLE_LAYER_PALETTE - console layer palette

2.11.27.2.2 Parameter – entry

entry - palette table entry 0-255

2.11.27.2.3 Parameter – rgb

Palette value structure.

2.11.27.3 Return Value

None.

2.11.27.4 Example

```
/* read value of colour 100 */
PALETTE_RGB *rgb;
BYTE alpha, red, green, blue;

CalypsoGetPaletteEntry (CONSOLE_LAYER_PALETTE, 100, rgb);
alpha=rgb.alpha;
red=rgb.red;
green=rgb.green;
blue=rgb.blue;
```

2.11.28 CalypsoSetPaletteEntry( )

void CalypsoSetPaletteEntry(UBYTE palette_select, UBYTE entry, PALETTE_RGB *col);

2.11.28.1 Description

Set palette entry in a palette table.

2.11.28.2 Parameters

2.11.28.2.1 Parameter – palette_select

MIDDLE_BASE_LAYER_PALETTE - middle/base layer palette
CONSOLE_LAYER_PALETTE - console layer palette

2.11.28.2.2 Parameter – entry

entry - palette table entry 0-255

2.11.28.2.3 Parameter – rgb

Palette value structure.

2.11.28.3 Return Value

None.

2.11.28.4 Example

```
/* set value of colour 32 */
PALETTE_RGB *rgb;

rgb.alpha = 1;          /* enable alpha blending for colour 32 */
rgb.red = 0xFF;         /* very red colour */
rgb.green =0x11 ;
rgb.blue = 0x11;

CalypsoSetPaletteEntry (CONSOLE_LAYER_PALETTE, 32, rgb);
```

2.11.29 CalypsoLoadImage( )

LONG CalypsoLoadImage(LONG video_addr, const CALYPSOIMAGE *image);

2.11.29.1 Description

Load image from system to video memory. This function works identically for 8/16 bit images.

Image source files can be generated from standard formats (.bmp, .jpg etc)
using Heber Utilities: (See quick start Guide for more a full description on how to use this utilities)

imagconv8.bat - for 8-bit images (generates optimum palette *palette.c*)
imagconv8std.bat - for 8-bit images (uses standard VGA palette *stdpalette.c*)
imagconv16.bat - for 16-bit images

These will produce source files image_name.c which can be added to the object of the makefile in order to be directly accessible from RAM or EPROM. The source files image_name.c contains image data, in the following structure:

```
const CALYPSOIMAGE image_name =
{
        image parameters…
        palette data.. .
        bit map data ….
};
```

Where the type *CALYPSOIMAGE* is defined as:

```
typedef struct calypsoimage {
    ULONG size_bytes;                   /* image size bytes */
    WORD color8;                        /* 1=8bit 0=16bit */
    WORD palette_size;                  /* number of palette entries */
    WORD width;                         /* width in pixels */
    WORD height;                    /* height in lines */
    const PALETTE_RGB *palette;         /* palette data */
```

```
        const WORD *data;                       /* image data */
} CALYPSOIMAGE;
```

MANDATORY:
Image width MUST be an *even number.*

2.11.29.2 Parameters

2.11.29.2.1 Parameter – video_addr

Display Layer start address (offset from start of video memory)

2.11.29.2.2 Parameter – image

Pointer to image structure

2.11.29.3 Return Value

Next free address.

2.11.29.4 Example

```
/*makefile */
……..

objects = game.o config.o devices.o custom.o idram.o \
sysram.o fpga.o vdemo.o \
bgnd.o

…….

/* demo.c */
extern const CALYPSOIMAGE bgnd;
LONG next_video_memory address;

/* load background image to start of video memory */
next_video_memory address  = CalypsoLoadImage (0, &bgnd);

/* bgnd.c - generated by imagconv */
#include          "hardware.h"

#define SIZE_BYTES              0x00079e00
#define COLOR8                  1
#define PALETTE_SIZE            256
#define WIDTH                   832
#define HEIGHT                   600
static const PALETTE_RGB palette[] = {
{0x80,0x00,0x00,0x00}, // Col   0

/* more palette data... */

{0x80,0x00,0x00,0x00}, // Col   255
{0,0,0,0}
};
static const WORD data[] = {
0x1111,

/* more image data... */

0
}
;
const CALYPSOIMAGE bgnd = {
```

```
        (ULONG) SIZE_BYTES,
        (WORD) COLOR8,
        (WORD) PALETTE_SIZE,
        (WORD) WIDTH,
        (WORD) HEIGHT,
        palette,
        data
};
```

2.11.30 CalypsoLoadImageLayer( )

LONG CalypsoLoadImageLayer(CALYPSOLAYER *layer, BYTE doublebuffer_frame, const CALYPSOIMAGE *image);

2.11.30.1 Description

Load image from system to specified Display Layer in video memory.
When loading a full screen background images, the *image width* must equal the *display layer width*

Works identically for 8/16 bit images.

2.11.30.2 Parameters

2.11.30.2.1 Parameter – layer

Display Layer

2.11.30.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.30.2.3 Parameter – image

Pointer to image structure.

2.11.30.3 Return Value

Next free address.

2.11.30.4 Example

```
/* demo.c */
extern const CALYPSOIMAGE bgnd;
LONG next;

/* load background image to Base Left Layer */
next  = CalypsoLoadImageLayer(&CalypsoLayer[BL_LAYER], DOUBLEBUFFER_FRAME0, &bgnd);
```

2.11.31 CalypsoLoadImageConvert8to16( )

LONG CalypsoLoadImageConvert8to16(LONG video_addr, const CALYPSOIMAGE* image);

2.11.31.1 Description

Load image from system to video memory with 8-bit to 16-bit colour conversion.
This function is a special version of CalypsoLoadImage( ). It converts on the fly from 8-bit to 16-bit colour using the palette in the image structure.

The image loading process is slow (due to conversion), but gives 50 per cent saving in EPROM space.

2.11.31.2 Parameters

2.11.31.2.1 Parameter – video_addr

Display Layer start address (offset from start of video memory)

2.11.31.2.2Parameter – image

Pointer to image structure

2.11.31.3 Return Value

Next free address.

2.11.31.4 Example

```
/* demo.c */
extern const CALYPSOIMAGE bgnd;     /* 8-bit image */
LONG , bgnd_start , next_video_memory_address;

bgnd_start = 0;
if(ColorMode == (BYTE)DIRECT_COLOR)
        next_video_memory_address = CalypsoLoadImageConvert8to16(bgnd_start, &bgnd);
else
        next_video_memory_address = CalypsoLoadImage(bgnd_start, &bgnd);
```

2.11.32 CalypsoLoadImageLayerConvert8to16 ( )

LONG CalypsoLoadImageLayerConvert8to16 (CALYPSOLAYER *layer, BYTE doublebuffer_frame, const CALYPSOIMAGE *image);

2.11.32.1 Description

Load image from system to Display Layer with 8-bit to 16-bit colour conversion. This function is a special version of CalypsoLoadImageLayer( ). It converts on the fly from 8-bit to 16-bit colour using the palette in the image structure.

The image loading process is slow (due to conversion), but gives 50 per cent saving in EPROM space.

2.11.32.2 Parameters

2.11.32.2.1Parameter – layer

Display Layer

2.11.32.2.2Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.32.2.3Parameter – image

Pointer to image structure

2.11.32.3 Return Value

Next free address.

2.11.32.4 Example

```
/* demo.c */
extern const CALYPSOIMAGE bgnd;     /* 8-bit image */
LONG next;

/* load background image to Base Layer */
next = CalypsoLoadImageLayer (CalypsoLayer[BL_LAYER], DOUBLEBUFFER_FRAME0, &bgnd);
```

2.11.33 CalypsoLoadSprite( )

LONG CalypsoLoadSprite(LONG video_addr, const CALYPSOIMAGE *image);

2.11.33.1 Description

Load *image* from system to video memory, then -
Create and load *image mask* to next video memory address

The *image mask* is required by function *CremsonPutSprite()* for *transparent* blitting

***Note*:** The image width MUST be an *even number*.

2.11.33.2 Parameters

2.11.33.2.1 Parameter – video_addr

Display Layer start address (offset from start of video memory)

2.11.33.2.2 Parameter – image

Pointer to image structure

2.11.33.3 Return Value

Next free memory address.

2.11.33.4 Example

LONG sprite_addr1, sprite_addr2;
sprite_addr2 = CalypsoLoadSprite(sprite_addr1, &sprite_image);

2.11.34 CalypsoPutSprite( )

WORD CalypsoPutSprite(CALYPSOLAYER *layer, BYTE doublebuffer_frame, LONG x, LONG y, LONG image_start, LONG xsize, LONG ysize, LONG mode);

2.11.34.1 Description

Blit sprite from undisplayed video memory to Display Layer

2.11.34.2 Parameters

2.11.34.2.1 Parameter – layer

Display Layer

2.11.34.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.34.2.3 Parameter – x

Layer x position.

2.11.34.2.4 Parameter – y

Layer y position.

2.11.34.2.5 Parameter – image_start

Start address of sprite image data in video memory.

2.11.34.2.6Parameter – xsize

Sprite width in pixels.

2.11.34.2.7Parameter – ysize

Sprite height in lines.

2.11.34.2.8Parameter – mode

NORMAL_MODE
TRANSPARENT_MODE

2.11.34.3 Return Value

None.

2.11.34.4 Example

2.11.35 CalypsoSaveSprite( )

WORD CalypsoSaveSprite(CALYPSOLAYER *layer, BYTE doublebuffer_frame, LONG x, LONG y, LONG save_address, LONG xsize, LONG ysize);

2.11.35.1 Description

Save Block (sprite) from Display Layer to undisplayed video memory.

2.11.35.2 Parameters

2.11.35.2.1Parameter – layer

Display Layer

2.11.35.2.2Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.35.2.3Parameter – x

Layer x position.

2.11.35.2.4Parameter – y

Layer y position.

2.11.35.2.5Parameter – image_start

Destination address in video memory.

2.11.35.2.6Parameter – xsize

Sprite width in pixels.

2.11.35.2.7Parameter – ysize

Sprite height in lines.

2.11.35.3 Return Value

None.

2.11.36 CalypsoClearSprite( )

WORD CalypsoClearSprite(CALYPSOLAYER *layer, BYTE doublebuffer_frame, LONG x, LONG y, LONG xsize, LONG ysize);

2.11.36.1 Description

Clear Block (sprite) in Display Layer

2.11.36.2 Parameters

2.11.36.2.1 Parameter – layer

Display Layer

2.11.36.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.36.2.3 Parameter – x

Layer x position.

2.11.36.2.4 Parameter – y

Layer y position.

2.11.36.2.5 Parameter – xsize

Sprite width in pixels.

2.11.36.2.6 Parameter – ysize

Sprite height in lines.

2.11.36.3 Return Value

None.

2.11.37 CalypsoFillSprite ( )

WORD CalypsoFillSprite(CALYPSOLAYER *layer, BYTE doublebuffer_frame, LONG x, LONG y, LONG xsize, LONG ysize, UWORD color, UBYTE mode);

2.11.37.1 Description

Fill Block (sprite) in Display Layer with colour value.

2.11.37.2 Parameters

2.11.37.2.1 Parameter – layer

Display Layer

2.11.37.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:

DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame
DOUBLEBUFFER_AUTO - currently non-displayed double buffer frame

This parameter is ignored for console and window layers, which are not double buffered.

2.11.37.2.3Parameter – x

Layer x position

2.11.37.2.4Parameter – y

Layer y position

2.11.37.2.5Parameter – xsize

Sprite width in pixels.

2.11.37.2.6Parameter – ysize

Sprite height in lines.

2.11.37.2.7Parameter – color

Fill colour.
The colour depth (8-bit or 16-bit) should match the Display Layer colour depth.

2.11.37.2.8Parameter – mode

Fill mode:
BLT_COPY - overwrite current pixels
BLT_AND - logical AND with current pixels
BLT_OR - logical OR with current pixels

2.11.37.3 Return Value

None.

2.11.38 CalypsoSetAsciiFont( )

void CalypsoSetAsciiFont(const CALYPSOFONT *font);

2.11.38.1 Description

Set the current ASCII text font.

Font source files can be generated from TrueType font files using Heber utility: *fontconv.bat*
(Refer to quick start guide in the font section for more information)

These will produce source files *font_name.c* containing font data in the following structure:

const CALYPSOFONT font_name = {
font parameters …
bit-map data…
};

2.11.38.2 Parameters

2.11.38.2.1Parameter – font

Pointer to font structure.

2.11.38.3 Return Value

None.

2.11.38.4 Example

CalypsoSetAsciiFont(&tempo16_0);

2.11.39 CalypsoSetTextColor( )

WORD CalypsoSetTextColor(UWORD foreground_color, UWORD background_color);

2.11.39.1 Description

Set current text colour

2.11.39.2 Parameters

2.11.39.2.1 Parameter – foreground_color

Foreground colour (8/16 bit)

2.11.39.2.2 Parameter – background_color

Background colour (8/16 bit)

2.11.39.3 Return Value

None.

2.11.39.4 Example

CalypsoSetTextColor(RED,WHITE); /* 8-bit */
CalypsoSetTextColor(0x1234, 0x0000); /* 16-bit */

2.11.40 CalypsoSetTextPosition( )

void CalypsoSetTextPosition(CALYPSOLAYER *layer, WORD x, WORD y);

2.11.40.1 Description

Set Display Layer current text writing position.

2.11.40.2 Parameters

2.11.40.2.1 Parameter – layer

Display layer

2.11.40.2.2 Parameter – x

Screen x co-ordinate in pixels.
x=0 - screen left

2.11.40.2.3 Parameter – y

Screen y co-ordinate in display lines.
y=0 – screen top

2.11.40.3 Return Value

None.

2.11.40.4 Example

See below.

2.11.41 CalypsoPutChar( )

WORD CalypsoPutChar(CALYPSOLAYER *layer, BYTE doublebuffer_frame, UBYTE ch);

2.11.41.1 Description

Write text character to Display Layer at current layer writing position.
Current writing position advances after character is written.

***Note*:** writing position is BOTTOM LEFT of character (refer to quick start guide : introduction to video programming)

2.11.41.2 Parameters

2.11.41.2.1 Parameter – layer

Display Layer
Note: It is recommended that text should be written to a *non double-buffered* display layers (Console or window layer). If the base or middle layer is used, they should be in *single-buffered* mode as this will avoid the complication of writing the same text twice (once to each double buffer frame)

2.11.41.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:
DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.41.2.3 Parameter – ch

ASCII Character code 0x00-0x7F

2.11.41.3 Return Value

None.

2.11.41.4 Example

```
WORD x,y;

x=0; y=32;      /* Note: writing position is BOTTOM LEFT of character */

/* start writing at top left of screen*/
CalypsoSetTextPosition(&CalypsoLayer[C_LAYER], x, y);
CalypsoPutChar(&CalypsoLayer[C_LAYER], 'A');
```

2.11.42 CalypsoPutStr( )

WORD CalypsoPutStr(CALYPSOLAYER *layer, BYTE doublebuffer_frame, BYTE *str);

2.11.42.1 Description

Write an ACII text string to the display layer at current layer writing position. The current writing position advances after string is written.

2.11.42.2 Parameters

2.11.42.2.1 Parameter – layer

Display Layer.

2.11.42.2.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:
DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.42.2.3 Parameter – str

Null terminated text string.

2.11.42.3 Return Value

None.

2.11.42.4 Example

```
WORD x,y;

x=0; y=32;      /* Note: writing position is BOTTOM LEFT of character */
/* start writing at top left of screen*/
CalypsoSetTextPosition(&CalypsoLayer[C_LAYER], x, y);
```

CalypsoPutStr(&CalypsoLayer[C_LAYER], 0, "----Calypso 16---");

2.11.43 CalypsoSetUnicodeFont( )

void CalypsoSetUnicodeFont(const CALYPSOFONT *font, UWORD unicode_page)

2.11.43.1 Description

Assign Unicode font to code page. Multiple fonts can be assigned to cover different Unicode code ranges. (Refer to quick start guide in the font section for more information).

These will produce source files *font_name.c* containing font data in the following structure:

const CALYPSOFONT font_name = {
font parameters …
bit-map data…
};

2.11.43.2 Parameters

2.11.43.2.1 Parameter – font

Pointer to Calypso Unicode font

2.11.43.2.2 Parameter – unicode_page

Start of corresponding Unicode code page

2.11.43.3 Return Value

None.

2.11.43.4 Example

See below.

2.11.44 CalypsoGetUnicodeFont( )

CALYPSOFONT *CalypsoGetUnicodeFont(UWORD unicode_page)

2.11.44.1 Description

Returns the Unicode Font assigned to Unicode code page

2.11.44.2 Parameters

2.11.44.2.1 Parameter – unicode_page

2.11.44.3 Return Value

Pointer to font assigned to page.
Returns NULL if no font assigned.

2.11.44.4 Example

See below.

2.11.45 CalypsoPutUnicodeChar( )

WORD CalypsoPutUnicodeChar(CALYPSOLAYER* layer, BYTE doublebuffer_frame, UWORD ch)

2.11.45.1 Description

Write Unicode character to display layer

Character is in UTF-16 big-endian format.
Character must be in Unicode range 0000 - 0xFFFF (64k characters)

2.11.45.1.1 Parameter – layer

Display Layer

Note: It is recommended that text should be written to a *non double-buffered* display layers (Console or window layer). If the base or middle layer is used, they should be in *single-buffered* mode as this will avoid the complication of writing the same text twice (once to each double buffer frame).

2.11.45.1.2 Parameter – doublebuffer_frame

Display Layer double buffer frame:
DOUBLEBUFFER_FRAME0 - first double buffer frame
DOUBLEBUFFER_FRAME1 - second double buffer frame

2.11.45.1.3 Parameter – ch

Unicode character 0x0000 - 0x7FFF. Encoding: UTF16 big endian

2.11.45.2 Return Value

Normally 0, returns 1 if display list full.

2.11.45.3 Example

```
/* Assign and Display Unicode Fonts */
#define N_UNICODE_PAGES   512

extern const CALYPSOFONT example16_0;
extern const CALYPSOFONT example16_80;
extern const CALYPSOFONT example16_100;
extern const CALYPSOFONT example16_180;
extern const CALYPSOFONT example16_200;
extern const CALYPSOFONT example16_380;

UWORD ch;     /* Unicode character UTF16 */
UWORD codepage;

CalypsoSetUnicodeFont(&example16_0, 0x0);
CalypsoSetUnicodeFont(&example16_80, 0x80);
CalypsoSetUnicodeFont(&example16_100, 0x100);
CalypsoSetUnicodeFont(&example16_180, 0x180);
CalypsoSetUnicodeFont(&example16_200, 0x200);
CalypsoSetUnicodeFont(&example16_380, 0x380);

for(n=0, codepage=0; n < N_UNICODE_PAGES;  n++) {
        /* check if font has been assigned to this unicode page */
        if(CalypsoGetUnicodeFont(codepage) != NULL) {
                for(ch=0; ch<128; ch++)
                        CalypsoPutUnicodeChar(&CalypsoLayer[C_LAYER], 0, codepage + ch);
        }
        codepage += 0x80;
}
```

2.11.46 CalypsoPutUnicodeStr( )

WORD CalypsoPutUnicodeStr(CALYPSOLAYER* layer, BYTE doublebuffer_frame, UWORD *str)

2.11.46.1 Description

Write Unicode string to display layer. Each character in string (array of 16-bit unsigned WORDS) is in UTF-16 big-endian format. Characters must be in Unicode range 0000 - 0xFFFF (64K characters)

2.11.46.1.1 Parameter – layer

Display Layer.

2.11.46.1.2Parameter – doublebuffer_frame

Display Layer double buffer frame:
DOUBLEBUFFER_FRAME0 - first double buffer frame.
DOUBLEBUFFER_FRAME1 - second double buffer frame.

2.11.46.1.3Parameter – str

Pointer to null-terminated array of 16 bit Unicode characters.

2.11.46.2 Return Value

Normally 0, returns 1 if display list full.

2.11.47 CalypsoDefineAnimation( )

void CalypsoDefineAnimation(ANIMATION* anim, CALYPSOLAYER* layer,
UWORD n_frames, WORD x, WORD y, WORD dx, WORD dy,
LONG image_start, LONG image_offset, WORD n_images, WORD image_index,
LONG width, LONG height, BYTE type, LONG mode,
LONG buffer1, LONG buffer2, void (*update)(ANIMATION* anim))

2.11.47.1 Description

Define in full an *animation sequence*.

An animation sequence is any defined succession of graphics images. Examples are moving sprites, and video clips.

The existing driver (CremsonDriver) requires the developer to explicitly write code to handle timing, double buffering and synchronisation with vertical interrupt.

The enhanced driver allows all parameters of an animation sequence to be specified in advance. The animation can then be started with a single command: all screen updates will be taken care of automatically by the driver. This greatly simplifies animation code.

Each *animation* defined relates to a single screen object, and has an associated ANIMATION structure. A maximum of 8 animations can be active at any one time.

2.11.47.2 Parameters

2.11.47.2.1Parameter – anim

Pointer to animation structure associated with the animation.

2.11.47.2.2Parameter – layer

The display layer in which the animation will run.

2.11.47.2.3Parameter – n_frames

n_frames - total number of display frames in the animation. (0 < n_frames < 32768)
- run forever if n_frames= ANIMATION_RUN_FOREVER

2.11.47.2.4Parameter –x

Start position – x coordinate in pixels.

2.11.47.2.5Parameter –y

Start position – y coordinate in display.

2.11.47.2.6Parameter –dx

Initial x speed - pixels per animation frame.

2.11.47.2.7Parameter –dy

Initial y speed – lines per animation frame.

2.11.47.2.8Parameter – image_start

Start address in video memory of first image in sequence.

2.11.47.2.9Parameter – image_offset

Multiple image sequence: video memory offset between images for consecutive frames.

2.11.47.2.10 Par
ameter – n_images

Multiple image sequence: number of separate images
Parameter – image_index
Multiple image sequence: current image index

2.11.47.2.11 Par
ameter – width

Image width in pixels. All images in an animation must be the same size.

2.11.47.2.12 Par
ameter – height

Image height in lines. All images in an animation must be the same size.

2.11.47.2.13 Par
ameter – type

Set the type of animation to be performed.

ANIMATION_TYPE1 - update position, put next image
ANIMATION_TYPE2 - clear previous image, update position, put next image
ANIMATION_TYPE3 - restore image background, update position,save image background,put next image

2.11.47.2.14 Par
ameter – mode

Blit mode:
NORMAL_MODE
TRANSPARENT_MODE

2.11.47.2.15 Par
ameter – buffer1

If type =ANIMATION_TYPE3 - address of buffer in video memory for storing image background. (Double buffer frame 0) Otherwise set to zero.

2.11.47.2.16 Par
ameter – buffer2

If type =ANIMATION_TYPE3 - address of buffer in video memory for storing image background. (Double buffer frame 1) Otherwise set to zero.

2.11.47.2.17 Par
ameter – update

Pointer to user supplied update function - calculates next x,y position

2.11.47.3 Example

```
ANIMATION example_anim;

/* Load fifteen images of spinning dice to video memory */
dice_start[0]    = sprite_start;
dice_start[1]    = CalypsoLoadSprite(dice_start[0], &dice1);
dice_start[2]    = CalypsoLoadSprite(dice_start[1], &dice2);
/* etc */
/* etc */
dice_start[13]   = CalypsoLoadSprite(dice_start[12],&dice13);
```

```
dice_start[14]    = CalypsoLoadSprite(dice_start[13],&dice14);

save_buffer[0]   = CalypsoLoadSprite(dice_start[14],&dice15);
dice_size_bytes = dice_start[1] - dice_start[0];
save_buffer[1]   = save_buffer[0] + dice_size_bytes;

/* define animation */
CalypsoDefineAnimation( &example_anim,
        &CalypsoLayer[BL_LAYER],                    /* Base Left Layer */
        ANIMATION_RUN_FOREVER,                      /* run forever */
        0,0,2,4,                                    /* startx=0, starty=0, x_speed=2, y_speed=4 */
        dice_start[0], dice_size_bytes, 15, 0,      /* 15 images of spinning dice in video memory */
        dice1.width, dice1.height,                  /* image sizes */
        ANIMATION_TYPE3, TRANSPARENT_MODE,          /* restore/update/save/put, transparent blit */
        save_buffer[0], save_buffer[1],             /* background save buffers */
        edgebounce );                               /* update function */


/* add animation to list */
CalypsoAddToAnimationList(&example_anim);

/* start animation */
CalypsoSetFrameRate(FRAME_30HZ);
CalypsoStartAnimation(&example_anim);

/*** After the animation has started, any of the following commands can be used subsequently…***/

/* pause animation */
CalypsoPauseAnimation(&example_anim);

/* single step animation */
CalypsoStepAnimation(&example_anim);

/* stop animation */
CalypsoStopAnimation(&example_anim);

/* clear list */
CalypsoClearAnimationList();


/* EXAMPLE: update function */
static void edgebounce(ANIMATION *anim)
{
WORD xlo, ylo, xhi, yhi;            /* screen limits */

/* define screen limits */
xlo = 0;
ylo = 0;
xhi = H_SIZE - anim->width;
yhi = V_SIZE - anim->height;

/* new position = old position + speed */
anim->x += anim->dx;
anim->y += anim->dy;

/* reverse direction at screen limits */
if(anim->x < xlo) {anim->x = xlo; anim->dx = -anim->dx;}
if(anim->y < ylo) {anim->y = ylo; anim->dy = -anim->dy;}
if(anim->x > xhi) {anim->x = xhi; anim->dx = -anim->dx;}
if(anim->y > yhi) {anim->y = yhi; anim->dy = -anim->dy;}

}
```

2.11.48 CalypsoAddToAnimationList( )

WORD CalypsoAddToAnimationList(ANIMATION* anim)

2.11.48.1 Description

Add an animation to the list of currently active animations.

After an animation has been defined with function *CalypsoDefineAnimation( )*, it must be added to the *Animation List.* This is a list of currently active animations.

The animation can then be run, paused single-stepped or stopped using functions:
CalypsoStartAnimation( ), CalypsoStopAnimation( ), CalypsoPauseAnimation( ), CalypsoStepAnimation( )

A maximum of EIGHT animations can be active at any time.

2.11.48.2 Parameters

2.11.48.2.1 Parameter – anim

Pointer to animation.

2.11.48.3 Return Value

Returns 1 if successful.
Returns 0 if *Animation List* full.

2.11.48.4 Example

See example for CalypsoDefineAnimation( )

2.11.49 CalypsoRemoveFromAnimationList( )

WORD CalypsoRemoveFromAnimationList(ANIMATION* anim)

2.11.49.1 Description

Remove an animation from the list of currently active animations.

2.11.49.2 Parameters

2.11.49.2.1 Parameter – anim

Pointer to animation.

2.11.49.3 Return Value

Returns 1 if successful.
Returns 0 if animation not found.

2.11.50 CalypsoClearAnimationList( )

void CalypsoClearAnimationList()

2.11.50.1 Description

Remove all animations from the *Animation List.*

2.11.50.2 Parameters

None.

2.11.50.3 Return Value

None.

2.11.50.4 Example

See example for CalypsoDefineAnimation( )

2.11.51 CalypsoStartAnimation( )

void CalypsoStartAnimation(ANIMATION* anim)

2.11.51.1 Description
Start an animation.

2.11.51.2 Parameters

2.11.51.2.1 Parameter – anim
Pointer to animation.

2.11.51.3 Return Value
None.

2.11.51.4 Example
See example for CalypsoDefineAnimation( )

2.11.52 CalypsoStopAnimation( )

void CalypsoStopAnimation(ANIMATION* anim)

2.11.52.1 Description
Stop (finish) an animation.

2.11.52.2 Parameters

2.11.52.2.1 Parameter – anim
Pointer to animation.

2.11.52.3 Return Value
None.

2.11.52.4 Example
See example for CalypsoDefineAnimation( )

2.11.53 CalypsoPauseAnimation( )

void CalypsoPauseAnimation(ANIMATION* anim)

2.11.53.1 Description
Pause an animation.

The animation can be resumed with *CalypsoStartAnimation( )*

2.11.53.2 Parameters

2.11.53.2.1 Parameter – anim
Pointer to animation.

2.11.53.3 Return Value
None.

2.11.53.4 Example
See example for CalypsoDefineAnimation( )

2.11.54 CalypsoStepAnimation( )

void CalypsoStepAnimation(ANIMATION* anim)

2.11.54.1 Description

Single step an animation.

Each call to *CalypsoStepAnimation( )* will advance the animation by one frame.

2.11.54.2 Parameters

2.11.54.2.1 Parameter – anim

Pointer to animation.

2.11.54.3 Return Value

None.

2.11.54.4 Example

See example for CalypsoDefineAnimation( )

2.11.55 CalypsoPoint( )

void CalypsoPoint(CALYPSOLAYER *layer, LONG x, LONG y, UWORD color);

2.11.55.1 Description

Draw pixel to current Display Layer.

2.11.55.2 Parameters

2.11.55.2.1 Parameter – layer

Display Layer.

2.11.55.2.2 Parameter – x

X position in display layer.

2.11.55.2.3 Parameter – y

Y position in display layer.

2.11.55.2.4 Parameter – color

Pixel Colour 8 or 16 bit.

2.11.55.3 Return Value

None.

2.11.56 CalypsoLine( )

void CalypsoLine(CALYPSOLAYER *layer, LONG Ax, LONG Ay, LONG Bx, LONG By, UWORD color);

2.11.56.1 Description

Draw line of given colour from co-ordinates Ax, Ay to co-ordinates Bx, By.

2.11.56.2 Parameters

2.11.56.2.1 Parameter – layer

Display Layer.

2.11.56.2.2 Parameter – Ax

Line start x co-ordinate.

2.11.56.2.3Parameter – Ay
Line start y co-ordinate.

2.11.56.2.4Parameter – Bx
Line end x co-ordinate.

2.11.56.2.5Parameter – By
Line end y co-ordinate.

2.11.56.2.6Parameter – color
Line colour.

2.11.56.3 Return Value
None.

2.11.57 CalypsoCircle( )

void CalypsoCircle(CALYPSOLAYER *layer, LONG xcentre, LONG ycentre, LONG radius, UBYTE color);

2.11.57.1 Description
Draw a circle of a given colour and radius, with its centre at co-ordinates xcentre, ycentre.

2.11.57.2 Parameters

2.11.57.2.1Parameter – layer
Display Layer

2.11.57.2.2Parameter – xcentre
X co-ordinate of centre of circle.

2.11.57.2.3Parameter – ycentre
Y co-ordinate of centre of circle.

2.11.57.2.4Parameter – radius
Radius of circle.

2.11.57.2.5Parameter – color
Colour of circle.

2.11.57.3 Return Value
None.

3 PERIPHERAL DRIVER LAYER

This section describes the functions provided by the various drivers in the peripheral layer. Some drivers have no public services available and are not documented; these drivers are accessed via modules in the interface layer which make calls to the services via the device structure. For example the elector mechanical meters driver.

3.1 Driver Templates

Support for various hoppers and coin / bank note acceptors can be added by using the generic hopper and money acceptor functions. This can be done using the driver template files found in \heber\templates and adding suitable configuration data for the particular device to be supported. Hopper configuration and acceptor configuration structures can be found in the example game project in config.c.

Support for different reels can be added by using the reel profile template files in \heber\templates. These should be edited and the appropriate ramp tables inserted. Contact the manufacturer of your particular reels for recommended ramp tables.

3.2 Timers

The timer module is a device driver for controlling timers.

3.2.1 SetTimer

BYTE SetTimer(TIMER *timer ,TIME time)

3.2.1.1 Description

The timer whose address is held in ‘timer’ is loaded with the given time. The timer is then installed into the timer list and will decrement every 10ms until it is zero; it will then be removed from the timer list and will not be decremented further. If the timer is already active when ‘SetTimer’ is called it will not be loaded into the timer list for a second time but will be loaded with the new value and it will be decremented from there.
N.B. Timers set with ‘SetTimer’ continue to run when the system is shut down in the event of an error.

3.2.1.2 Parameters

3.2.1.2.1 Parameter 1 – timer

Address of the timer to set.

3.2.1.2.2 Parameter 2 – time

The time (in 10ms units) the timer is to be loaded with.

3.2.1.3 Return Value

Returns TRUE on success.
Returns FALSE if there is no room in the timer list.

3.2.1.4 Example

```
TIMER  timer1;
SetTimer(&timer1,100);
while(timer1);    // wait for 1 second
```

3.2.2 SetFreezeTimer

BYTE SetFreezeTimer(TIMER *timer ,TIME time)

3.2.2.1 Description

The timer whose address is held in ‘timer’ is loaded with the given time. The timer is then installed into the timer list and will decrement every 10ms until it is zero; it will then be removed from the timer list and will not be decremented further. If the timer is already active when ‘SetTimer’ is called it will not be loaded into the timer list for a second time but will be loaded with the new value and it will be decremented from there.

***Note*:** Timers set with 'SetFreezeTimer' stop decrementing when the system is shut down in the event of an error and restart when the system comes out of shutdown.

3.2.2.2 Parameters

3.2.2.2.1 Parameter 1 – timer

Address of the timer to set.

3.2.2.2.2 Parameter 2 – time

Time, in 10ms units, the timer is to be loaded with.

3.2.2.3 Return Value

Returns TRUE on success.
Returns FALSE if there is no room in the timer list.

3.2.2.4 Example

```
TIMER  timer1;
SetFreezeTimer(&timer1,100);              // wait for 1 second
while(timer1);
```

3.2.3 Wait

void Wait(TIME time)

3.2.3.1 Description

Wait for the specified time before returning.
N.B. This function uses SetTimer and as such the time continues to count down during shutdown.

3.2.3.2 Parameters

3.2.3.2.1 Parameter 1 – time

The time (in 10ms units) the function is to wait before returning.

3.2.3.3 Return Value

None.

3.2.3.4 Example

```
Wait(100);        // wait for 1 second
```

3.2.4 WaitF

void WaitF(TIME time)

3.2.4.1 Description

Wait for the specified time before returning.
N.B. This function uses SetFreezeTimer and as such the time stops counting down during shutdown.

3.2.4.2 Parameters

3.2.4.2.1 Parameter 1 – time

The time (in 10ms units) the function is to wait before returning.

3.2.4.3 Example

```
WaitF(100);       // wait for 1 second.
```

3.3 Stepping Motor Profiles

3.3.1 Overview

The generic stepper motor driver will typically be used to control reels and can handle many different types of reel. There is no limit in the software to the number of reels (limit = 6 reels for hardware) or the number of different types of reel that can be controlled simultaneously.
All the information defining a particular reel type is contained in a REEL_TYPE structure.

```
typedef struct reel_type
{
        WORD  steps_per_rev;            // total number of steps in a full rotation
        WORD  steps_per_symbol;         // number of steps for one symbol
        BYTE    min_opto;       // number of time an opto is seen before counting a tab
        BYTE    tolerance;      // Maximum error in motor steps before correcting an error in position
        BYTE    revpos;         // Do we correct the position if the reel spins backward??
        const RAMP      *ramps[MAX_RAMPS];  // list of ramps the reel can use
} REEL_TYPE;
```

3.3.2 Defining a New Reel Type

The type declaration for the REEL_TYPE structure can be found in stepper.h, refer to this when defining a new type structure. A full definition of a reel will look as below:

```
const  REEL_TYPE      ReelType=
        {
        48,                     /* nof steps per rev */
        3,                      /* nof steps per symbol */
        1,                      /* time to see tab */
        2,                      /* allowable error */
        TRUE,                   /* correct position for reverse spin */
        &Ramp_I,                /* ramp - INI   */
        &Ramp_S,                /* ramp - SYS   */
        &Ramp_69,               /* ramp - 69    */
        &Ramp_78,               /* ramp - 78    */
        &Ramp_89,               /* ramp - 89    */
        &Ramp_96,               /* ramp - 96    */
        &Ramp_104,              /* ramp - 104   */
        &Ramp_HSN,              /* ramp - nudge1 */
        &Ramp_HSN,              /* ramp - nudge2 */
        &Ramp_HSN,              /* ramp - nudge3 */
        &Ramp_wobble,           /* ramp - Wobble */
        &Ramp_HSwobble,         /* ramp - Half Step Wobble */
        &Ramp_skillslow,        /* ramp - Skill stop  */
        &Ramp_skillmed,         /* ramp - Skill stop  */
        &Ramp_skillfast,        /* ramp - Skill stop  */
        };
```

3.3.2.1 steps_per_rev

The number of physical motor steps required to move the reel one complete revolution. The more steps a reel has, the smaller the distance between two steps.

3.3.2.2 steps_per_symbol

The number of physical motor steps required to move the reel one whole reel symbol.

$$\frac{\text{steps\_per\_rev}}{\text{steps\_per\_symbol}} \equiv Number\_of\_symbols$$

The distance to move the reel of one symbol will be called one motor position for the whole description.
One motor position = steps_per_symbol physical motor steps.

3.3.2.3 min_opto

On each reel, there is a tab of varying width. The number of consecutives times that the tab is seen by the sensor depends on: the step distance, the type of reel and the accuracy of the RAMP table.
However, a tab position (equilibrium) must be defined by one physical position of the reel.
This parameter defines the number of times the position sensor must be active before counting a tab position. A count on a tab position will activate the position correction software.

3.3.2.4 Tolerance

This is the maximum inaccuracy in number of physical motor steps allowable before activating the position correction software. This inaccuracy is measured every rotation.

3.3.2.5 revpos

This should be set TRUE if the position correction software is to be active whilst the reel is spinning backwards. This position correction software is always active whilst the reel is spinning forwards.

3.3.2.6 ramps

This is a list of pointers to RAMP structures. A RAMP structure contains all the information required to define a particular type of reel movement. The demo software uses a list of RAMPS as follow:
There are two ramps for system use, ‘system’ and ‘initialisation’ ramps, In addition there are ramps for spinning at 69RPM, 78RPM, 89RPM, 96RPM, and 104RPM.There is provision for three different nudge ramps and three different ‘skill stop’ ramps.

3.3.3 The RAMP Structure

The RAMP structure allows the stepper driver to be very flexible in the way it can control the reels. A RAMP structure can be defined to make a reel do almost anything.

3.3.3.1 pattern_tbl

The pattern table defines the energising sequence of the four phases of the stepper motor. To move one step the reel increments its index into this table and writes the new pattern out to the motor coils.

3.3.3.2 pixmask

Each time the pattern table index is incremented (or decremented for reverse spin) it is logically ANDed with this value to within the table.

3.3.3.3 vector_tbl

This is a list of function pointers to the functions, which will be called by the stepper driver in order to control the reel. The standard vector tables provided will suffice for most situations but for any special requirements a new vector table may be defined and new functions written.

3.3.3.4 delay_lst

A list of delays (in milliseconds) required to time the various steps. The vector functions will access these delays and the meaning of any particular delay is dependant on the vector functions used and the number of steps controlled by each vector function. For the standard vector functions (RampUp, Run, and RampDown) the first number indicates the number of steps to be controlled by that vector function and that is followed by the a delay for each step. It is left to the vector functions to manipulate the index into this delay list. This information is derived from the RAMP tables provided by the manufacturer.

3.3.4 Specification of the driver:

The stepper driver provides information on the reel position, the number of errors in asserting the position and how many times these errors were fixed by the correction software.

The position of the reel can only be checked when the tab is passing in front of the sensor. However, the position of the reel will be tracked independently of the vector function. The position will be checked during RampUp, and Run.

RampDown vector is not checked anymore as in some occasions it could end up in an extra rotation of the reel

To go from one position to another, the reels travel a distance. The position of the reel can be corrected all along this distance except during the number of steps defined in a ramp down vector.

3.4 StepMotors

This module contains wrapper functions, which call the appropriate functions from the given stepper driver.

3.4.1 GetNofMotors

WORD GetNofMotors(const DEVICE* dev)

3.4.1.1 Description

Return the number of stepper motors controlled by the given device driver.

3.4.1.2 Parameters

3.4.1.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.1.3 Return Value

Return the number of stepping motors controlled by the given device.

3.4.1.4 Example

```
n = GetNofMotors(&StepperDevice);
```

3.4.2 InitialiseStepMotors

void InitialiseStepMotors(const DEVICE* dev)

3.4.2.1 Description

Initialise all the stepper motors controlled by the given device driver. The stepper diver is usually defined in the file config.c. InitialiseStepMotors spins all motors to their reference position (tab position) and detect missing or faulty motors/position sensors. If errors are detected the system shuts down and the error handler is called.

3.4.2.2 Parameters

3.4.2.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.2.3 Return Value

None.

3.4.2.4 Example

```
InitialiseStepMotors(&StepperDevice);
```

3.4.3 StepMotorsIdle

BYTE StepMotorsIdle(const DEVICE* dev);

3.4.3.1 Description

Test the status of all motors controlled by the device and return TRUE if they are all idle.

3.4.3.2 Parameters

3.4.3.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.3.3 Return Value

Return TRUE if all motors are idle.
Return FALSE if any motor is spinning.

3.4.3.4 Example

```
while(!StepMotorsIdle(&StepperDevice));          // Wait for all motors to finish
```

3.4.4 DetectStepMotorOptos

LONG DetectStepMotorOptos(const DEVICE* dev)

3.4.4.1 Description

Read the status of the optical position sensor of each motor.

3.4.4.2 Parameters

3.4.4.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.4.3 Return Value

Return a 32 bit integer with each bit representing the status of each motor. A 1 means the motor is at its reference position (the sensor is read as active).The least significant bit represents the first motor (motor index = 0). Unused bits are set to zero.

3.4.4.4 Example

```
LONG  rmask;
rmask=DetectStepMotorOptos(&StepperDevice);        // read status of each motor.
nb_reels=cpt=GetNofMotors(&StepperDevice);         // number of reels controlled by driver
while(cpt--)                                       // display result of init.
      {
        if(rmask & 1) sprintf(str,"OK ");
        else sprintf(str,"BAD ");
        TxString(dev,channel,str);
        rmask>>=1;
      }
```

3.4.5 SpinStepMotorsToTab

void SpinStepMotorsToTab(const DEVICE* dev, WORD rampidx)

3.4.5.1 Description

Spin all motors controlled by the device to their reference positions.

3.4.5.2 Parameters

3.4.5.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.5.2.2 Parameter 2 – rampidx

Index to the ramp table to use. The ramp table specifies how the motor is controlled and the delays between each step. This information is found in the motor type structure.

3.4.5.3 Return Value

None.

3.4.5.4 Example

```
SpinStepMotorsToTab(&StepperDevice,RAMP_104);
while(!StepMotorsIdle(&StepperDevice));          // ensure stepper on idle mode before doing anything else
```

3.4.6 SpinStepMotor

Void SpinStepMotor(const DEVICE* dev, WORD motidx, WORD dist, WORD rampidx)

3.4.6.1 Description

The motor is set up to spin the given distance in motor position. Control returns from the function immediately and the reel spins and stops in its own time. If the motor is not in the idle state when this function is called it first waits for the motor to enter the idle state and then starts the next spin and returns.

3.4.6.2 Parameters

3.4.6.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.6.2.2 Parameter 2 – motidx

Motor to spin. This is the index to the motor in the device motor list, which is found in the device configuration structure.

3.4.6.2.3 Parameter 3 – dist

Distance to spin the motor. The distance is in motor positions.
Negative value will cause the motor to spin in reverse.
Distance will not be considered of ramp table RAMP_NUDGE1, RAMP_NUDGE2 and RAMP_NUDGE3

3.4.6.2.4 Parameter 4 – rampidx

Index to the ramp table to use. The ramp table specifies how the motor is controlled and the delays between each step.

3.4.6.3 Return Value

None.

3.4.6.4 Example

Spin motor 0 of 'StepperDevice' 12 motor positions using the 96RPM control ramp.

```
SpinStepMotor(&StepperDevice,0,12,RAMP_96);
while(!StepMotorsIdle);          // Wait for all motors to finish
```

3.4.7 StopStepMotor

Void StopStepMotor(const DEVICE* dev,WORD motidx)

3.4.7.1 Description

Bring the motor to rest on the first possible motor position.
N.B. For this function to work the motor must be spinning with one of the 'skill' control ramps. The spin must be given a non-zero distance which translates to more physical steps than the 'ramp up' phase of the control ramp. Having reached full speed the motor will spin until it is stopped using this function.

3.4.7.2 Parameters

3.4.7.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.7.2.2 Parameter 2 – motidx

Motor to spin. This is the index to the motor in the device motor list.

3.4.7.3 Return Value

None.

3.4.7.4 Example

Start the motor spinning and stop it when input IP20 becomes active.

```
SpinStepMotor(&StepperDevice,0,12, RAMP_SKILLMED);
while(!MotorsIdle(&StepperDevice))
        if(Test(IP20)) StopMotor(StepperDevice,0);
```

3.4.8 GetStepMotorPosn

WORD GetStepMotorPosn(const DEVICE* dev, WORD motidx)

3.4.8.1 Description

Return the current motor position. This value represents the distance in motor position from the current position of the reel to the tab position.

3.4.8.2 Parameters

3.4.8.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.8.2.2 Parameter 2 – motidx

Specifies the motor whose position is to be returned. This is the index to the motor in the device motor list.

3.4.8.3 Return Value

Current motor position in the range 0 to N-1.
(Where N is the number of motor position in a complete revolution).

3.4.8.4 Example

```
Pos0 = GetStepMotorPosn(&StepperDevice,0);
```

3.4.9 StepMotorIdle

BYTE StepMotorIdle(const DEVICE* dev, WORD motidx)

3.4.9.1 Description

Test the status of the motor and return TRUE if the motor is idle. When the motor is in the idle state the coils are duty cycled to prevent overheating. There is a period of time between the motor stopping and the motor entering the idle state. A motor must be in the idle state before it can start spinning.

3.4.9.2 Parameters

3.4.9.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.9.2.2 Parameter 2 – motidx

Motor to test. This is the index to the motor in the device motor list.

3.4.9.3 Return Value

Return TRUE if specific motor is idle.
Return FALSE if motor is spinning.

3.4.9.4 Example

```
while(!StepMotorIdle(&StepperDevice,0));          // wait for motor to stop
SpinStepMotor(&StepperDevice,0,12, RAMP_96);      // then spin it again
```

3.4.10 StepMotorSpinning

BYTE StepMotorSpinning(const DEVICE* dev, WORD motidx)

3.4.10.1 Description

Test the status of the motor and return TRUE if the motor is spinning. The motor will leave the spinning state some time before entering the idle state. This function should be used to synchronise events with the motor stopping.

3.4.10.2 Parameters

3.4.10.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.10.2.2 Parameter 2 – motidx

Motor to test. This is the index to the motor in the device motor list.

3.4.10.3 Return Value

Return TRUE if the motor is spinning. Return FALSE otherwise.

3.4.10.4 Example

```
while(StepMotorSpinning(&StepperDevice,0));    // Wait for motor to stop
Playback(2,Tune1,10);  // Play tune as motor stops
```

3.4.11 GetStepMotorTabsMissed

WORD GetStepMotorTabsMissed(const DEVICE* dev, WORD motidx)

3.4.11.1 Description

Get the number of times the motor has passed through the reference position without the position sensor being read as active. The count is kept internally and cleared by ClearStepMotorErrors().

3.4.11.2 Parameters

3.4.11.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.11.2.2 Parameter 2 – motidx

Motor to test. This is the index to the motor in the device motor list.

3.4.11.3 Return Value

Return the number of times the position sensor has been expected but not detected.

3.4.11.4 Example

```
Err = GetStepMotorTabsMissed(&StepperDevice,0);
```

3.4.12 GetStepMotorPosnCorrected

WORD GetStepMotorPosnCorrected(const DEVICE* dev, WORD motidx)

3.4.12.1 Description

Get the number of times the motors internal position variable has been corrected as a result of the position sensor being read as active when the position variable has not shown the reference position. The count is kept internally and cleared by ClearStepMotorErrors().

3.4.12.2 Parameters

3.4.12.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.12.2.2 Parameter 2 – motidx

Motor to test. This is the index to the motor in the device motor list.

3.4.12.3 Return Value

Return the number of times the motor position has been corrected.

3.4.12.4 Example

Err = GetStepMotorPosnCorrected(&StepperDevice,0);

3.4.13 ClearStepMotorErrors

void ClearStepMotorErrors(const DEVICE* dev, WORD motidx)

3.4.13.1 Description

Clear the internal position detection variables.

3.4.13.2 Parameters

3.4.13.2.1 Parameter 1 – dev

Address of the device driver structure.

3.4.13.2.2 Parameter 2 – motidx

Specifies the motor whose errors are to be cleared. This is the index to the motor in the device motor list, which is found in the device configuration structure in config.c.

3.4.13.3 Return Value

None.

3.4.13.4 Example

ClearStepMotorErrors(&StepperDevice,0);

3.5 Stepper Driver Definition

The stepper driver definition can be found inside the file “reel.c” usually within a project. The duty cycle timing is controlled all along the functioning of the reels by the two last delays of a RAMP:

Example:

```
static const   WORD    Ramp_96_delay_tbl[]=
                  {
                  10                        /* Stabilize       */
                  4,     37,26,14,23,        /* Ramp Up      */
                  1,                13,      /* Run             */
                  5,     15,25,16,41,        /* Ramp Down   */
                  100,                       /* Settle time before duty cycle    */

                  20,20                     /* Duty cycle */
                  };
```

Depending on the reels used, the duty cycle can be reviewed by updating these parameters.

It has been noticed that during the initialization of the stepper driver, the duty cycle parameters used are within our driver. As they are part of the source code, they cannot be seen or modified. It has also been noticed that on particular occasions, the reels stutter during the initialization process. Therefore, two new functions have been created to update the duty cycle specific to the initialization process:

- SetWakeUpDelay
- GetWakeUpDelay

3.5.1 SetWakeUpDelay

void setWakeUpDelay(UWORD setval);

3.5.1.1 Description

Creates the value of the new delay of the duty cycle within initialization.

3.5.1.2 Parameter - setval

The value of the delay.

3.5.1.3 Return Value

None.

3.5.1.4 Example

See below.

3.5.2 GetWakeUpDelay

UWORD getWakeUpDelay(void)

3.5.2.1 Description

Returns the current value of the delay of duty cycle within initialization.

3.5.2.2 Parameter – void

3.5.2.3 Return Value

None.

3.5.2.4 **Example:** The following example illustrates the use of both functions:

```
/* display the current initialization duty cycle)
sprintf(str,"wakeup delay = %d\r",getWakeUpDelay());
TxString(dev,channel,str);
Wait(2);

/* change the delay to initialization duty cycle to 15*/
        setWakeUpDelay(15);

/*reinitialize the steppers */
InitialiseStepMotors(&StepperDevice);// spin all the motors to their reference position.
```

Security PIC

The security PIC is optionally fitted to the Pluto 5 Casino board and has two functions. Firstly it operates as a real time clock and secondly it records and time stamps door transitions. The PIC has its own battery circuit, which enables it to operate when the Pluto 5 board is powered down. The processor communicates with the PIC via the I2CPICDevice driver.
The PIC maintains a log of the last twelve door transitions and with each is recorded the time the event occurred. In addition to this log a record is kept of door transition from open to close and from close to open.

3.5.3 GetPICTime

BYTE GetPICTime(const DEVICE* picdev, RTIME* rtime);

3.5.3.1 Description

Read the current time and date from the PIC to the RTIME structure.

3.5.3.2 Parameters

3.5.3.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.3.2.2 Parameter 2 – rtime

Address of the RTIME structure the data is to be copied to.

3.5.3.3 Return Value

Returns TRUE if the data is read successfully.
Returns FALSE if the read is not completed successfully.

3.5.3.4 Example

```
if(GetPICTime(&PICDevice, &time))
        'time' contains valid time and date read from PIC
else
        read failed. 'time' is invalid
```

3.5.4 GetPICResetTime

BYTE GetPICResetTime(const DEVICE* picdev, RTIME* rtime)

3.5.4.1 Description

Writes into the RTIME structure the time at which the PIC was last reset

3.5.4.2 Parameters

3.5.4.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.4.2.2 Parameter 2 – rtime

Address of the RTIME structure the data is to be copied to.

3.5.4.3 Return Value

Returns TRUE if the data is read successfully.
Returns FALSE if the read is not completed successfully.

3.5.4.4 Example

```
if(GetPICResetTime(&PICDevice, &time))
        'time' contains valid time and date read from PIC
else
        read failed. 'time' is invalid
```

3.5.5 SetPICTime

BYTE SetPICTime(const DEVICE* picdev, const RTIME* time)

3.5.5.1 Description

Set the PIC clock to the time and date specified in the RTIME structure

3.5.5.2 Parameters

3.5.5.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.5.2.2 Parameter 2 – rtime

Address of the RTIME structure containing the time and date to be set.

3.5.5.3 Return Value

Returns TRUE if the PIC clock is successfully set.
Returns FALSE if the PIC clock was not set.

3.5.5.4 Example

```
RTIME real_time;
real_time.year = 2000;
real_time.month = 1;
real_time.weekday = 1;
real_time.date = 1;
real_time.hours = 0;
real_time.minutes = 0;

if(SetPICTime(&PICDevice,&real_time))
          TxString(&SerialDevice,1,"PIC RTC RESET\r");
        else
          TxString(&SerialDevice,1,"PIC RTC WRITE FAIL\r");
```

3.5.6 GetPICStatus

BYTE GetPICStatus(const DEVICE* dev, BYTE* sr)

3.5.6.1 Description

Read the PIC status byte into the byte variable 'sr'.
If Bit 0 =1 then the PIC has detected a power reset.
If Bit 1 =1 then the PIC has detected data corruption.

3.5.6.2 Parameters

3.5.6.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.6.2.2 Parameter 2 – sr

Address of the byte variable the status byte is to be copied to.

3.5.6.3 Return Value

Returns TRUE if the PIC status byte was successfully read.
Returns FALSE if the PIC was not successfully read.

3.5.6.4 Example

```
if(GetPICStatus(&PICDevice, &sr))
        'sr' holds PIC status byte
else
        'sr' is undefined
```

3.5.7 GetPICSwO2C

BYTE GetPICSwO2C(const DEVICE* dev, BYTE* dst)

3.5.7.1 Description

Reads the PIC door status byte showing door transitions from open to close since the last time the PIC switch monitor log was cleared. (See ClearPICLog()). The PIC monitors seven switches (P20 pins 1-7). Pin eight is the switch common, switches must be wired to this pin NOT ground.

3.5.7.2 Parameters

3.5.7.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.7.2.2 Parameter 2 – dst

Address of the byte variable the door status byte is to be copied to.

3.5.7.3 Return Value

Returns TRUE if the PIC door status byte was successfully read.
Returns FALSE if the PIC was not successfully read.

3.5.7.4 Example

```
if(GetPICSwO2C(&PICDevice, &st))
{
        'st' holds PIC door status byte
}
else
{
        'st' is undefined
}
```

3.5.8 GetPICSwC2O

BYTE GetPICSwC2O(const DEVICE* dev, BYTE* dst)

3.5.8.1 Description

Reads the PIC door status byte showing door transitions from closed to open since the last time the PIC switch monitor log was cleared. (See ClearPICLog()).The PIC monitors seven switches (P20 pins 1-7). Pin eight is the switch common, switches must be wired to this pin NOT ground.

3.5.8.2 Parameters

3.5.8.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.8.2.2 Parameter 2 – dst

Address of the byte variable the door status byte is to be copied to.

3.5.8.3 Return Value

Returns TRUE if the PIC door status byte was successfully read.
Returns FALSE if the PIC was not successfully read.

3.5.8.4 Example

```
if(GetPICSwC2O(&PICDevice, &st))
        'st' holds PIC door status byte
else
        'st' is undefined
```

3.5.9 GetPICLog

BYTE GetPICLog(const DEVICE* dev,BYTE idx,PICLOG* dst)

3.5.9.1 Description

Copies the PIC door switch log entry from the given position in the log to the address defined by 'dst'

It is advised that all PIC access should still have a single retry in case of failure.

It should be noted that communication with the PIC is a very slow, due to the low clock speed of the PIC. We strongly advise not to retrieve alarms from the PIC in an interrupt or a video animation loop as it will probably cause unacceptable execution delays (screen freezing, interrupt overrun)

3.5.9.2 Parameters

3.5.9.2.1 Parameter 1 – picdev

Address of the PIC device driver structure.

3.5.9.2.2 Parameter 2 – idx

Log entry to be copied: THIS ENTRY IS A NEGATIVE NUMBER. Entry -1 is the most recent; entry –2 is before that, down to entry –12 being the oldest.

3.5.9.2.3 Parameter 3 – dst

Address of the PICLOG variable the log information is to be copied to.

3.5.9.3 Return Value

Returns TRUE if the log was read correctly.
Returns FALSE otherwise.

3.5.9.4 Example

This example will demonstrate how to read the 12 alarms consecutively with a retry in case of failure for each log.

```
PICLOG power_off_reading;
BYTE text_string[16];
WORD j;

BYTE str[50];
BYTE retry;
BYTE status;

retry=0;
j=1;
status=TRUE;
while (status==TRUE)
        {
          sprintf(text_string,"RD -%d",j);
          TxString(&SerialDevice,1,text_string);
          Wait(50);                          //THIS DELAY IS RECOMMENDED BUT NOT MANDATORY
          if(GetPICLog(&PICDevice,-j ,&power_off_reading))
```

```
        {
                TxString(&SerialDevice,1,"\t SUCCESS\r");
                retry=0;
                if(j<12)
                        j++;
                else
                        status=FALSE;
        }
        else
        {
                TxString(&SerialDevice,1,"\t FAIL\r");
                if(retry==0)
                {
                        sprintf(text_string,"retry ",j);
                        TxString(&SerialDevice,1,text_string);
                        retry=1;
                }
                else
                {
                        retry=0;
                        if(j<12)
                                j++;
                        else
                                status=FALSE;
                }
        }
}
```

3.5.10 ConvertPICTime

void ConvertPICTime(LONG pic, RTIME* time)

3.5.10.1 Description

Convert the 24 bit PIC time into the RTIME format.

3.5.10.2 Parameters

3.5.10.2.1 Parameter 1 – pic

24 bit time used by the PIC and stored in the least significant 24 bits of a PIC log entry.

3.5.10.2.2 Parameter 2 – time

Pointer to the RTIME structure to write the time to.

3.5.10.3 Return Value

None.

3.5.10.4 Example

```
PICLOG          log;
if(GetPICLog(&PICDevice,0,&log))
        ConvertPICTime(log & 0x00FFFFFF, &time);
```

3.5.11 ClearPICLog

BYTE ClearPICLog(const DEVICE* dev)

3.5.11.1 Description

The PIC log and door status bytes are cleared.

3.5.11.2 Parameters

3.5.11.2.1 Parameter 1 – picdev
Address of the PIC device driver structure.

3.5.11.3 Return Value
Returns TRUE if the log was cleared correctly - Returns FALSE otherwise.

3.5.11.4 Example
ClearPICLog(&PICDevice);

3.5.12 ClearPICResetTime
BYTE ClearPICResetTime(const DEVICE* dev)

3.5.12.1 Description
The PIC time of last reset is cleared.

3.5.12.2 Parameters

3.5.12.2.1 Parameter 1 – picdev
Address of the PIC device driver structure.

3.5.12.3 Return Value
Returns TRUE if the time was cleared correctly - Returns FALSE otherwise.

3.5.12.4 Example
ClearPICResetTime(&PICDevice);

3.5.13 ClearPICStatus
BYTE ClearPICStatus(const DEVICE* dev);

3.5.13.1 Description
The PIC status byte is cleared.

3.5.13.2 Parameters

3.5.13.2.1 Parameter 1 – picdev
Address of the PIC device driver structure.

3.5.13.3 Return Value
Returns TRUE if the status byte was cleared correctly - Returns FALSE otherwise.

3.5.13.4 Example
ClearPICStatus(&PICDevice);

E2Rom

This module is a driver for accessing a E2Rom via the internal or external I2c bus.
The device configuration data specifies the address and type of device and the device driver for the I2C bus (internal or external)

3.5.14 ReadE2Rom
BYTE ReadE2Rom(const DEVICE* dev,WORD offset,WORD n,BYTE *dst)

3.5.14.1 Description

Read the given number of bytes from the given device and write them to the destination address in system memory.

3.5.14.2 Parameters

3.5.14.2.1 Parameter1 – dev

A pointer to the device structure for the E2Rom device.

3.5.14.2.2 Parameter2 – offset

The byte offset into the device where the bytes are to be read from.

3.5.14.2.3 Parameter3 – n

The number of bytes to be read.

3.5.14.2.4 Parameter4 – dst

The address in system memory where the bytes are to be copied.

3.5.14.3 Return Value

TRUE on success.
FALSE otherwise.

3.5.14.4 Example

The following code writes a string to the E2Rom and reads it back in order to test for the presence of the device.

```
const char*       E2RStr[]={“E2Rom detected”};
char* str[64];
BYTE sr;

sr=WriteE2Rom(E2RomStr,sizeof(E2RomStr),&E2RomDevice,0);
sr|=ReadE2Rom(&E2RomDevice,0,sizeof(E2RomStr),str);
if(sr) TxString(device,channel,E2RStr);
else TxString(device,channel,"E2Rom not detected...\r");
```

3.5.15 WriteE2Rom

BYTE WriteE2Rom(const BYTE *src,WORD n,const DEVICE* dev,WORD offset)

3.5.15.1 Description

Write the given number of bytes from system memory to the E2Rom device.

3.5.15.2 Parameters

3.5.15.2.1 Parameter1 – src

The address in system memory of the start of the data to be written to the E2Rom device.

3.5.15.2.2 Parameter2 – n

The number of bytes to be written to the E2Rom device.

3.5.15.2.3 Parameter3 – dev

A pointer to the device structure of the E2Rom device driver.

3.5.15.2.4 Parameter4 – offset

The byte offset into the device where the data is to be written

3.5.15.3 Return Value

Return TRUE on success.

3.5.15.4 Example

The following code writes a string to the E2Rom and reads it back in order to test for the presence of the device.

```
const char*     E2RStr[]={“E2Rom detected”};
char* str[64];
BYTE sr;

sr=WriteE2Rom(E2RomStr,sizeof(E2RomStr),&E2RomDevice,0);
sr|=ReadE2Rom(&E2RomDevice,0,sizeof(E2RomStr),str);
if(sr) TxString(device,channel,E2RStr);
else TxString(device,channel,"E2Rom not detected...\r");
```

3.6 Real Time Clock

This module contains a device drive for accessing the real time clock.

3.6.1 StartRTC

BYTE StartRTC(const DEVICE* dev)

3.6.1.1 Descriptions

Start the RTC running.

3.6.1.2 Parameters

3.6.1.2.1 Parameter1 – dev

A pointer to the RTC device driver structure.

3.6.1.3 Return Value

TRUE on success.
FALSE otherwise.

3.6.1.4 Example

```
if(StartRTC(&RTCDevice))
        {
        // Clock is now running
}
```

3.6.2 StopRTC

BYTE StopRTC(const DEVICE* dev)

3.6.2.1 Descriptions

Stop the RTC.

3.6.2.2 Parameters

3.6.2.2.1 Parameter1 – dev

A pointer to the RTC device driver structure.

3.6.2.3 Return Value

TRUE on success. FALSE otherwise.

3.6.2.4 Example

```
if(StopRTC(&RTCDevice))
        { // Clock is now running }
```

3.6.3 GetRTCTime

BYTE GetRTCTime(const DEVICE* rtcdev,RTIME* time)

3.6.3.1 Descriptions

Read the time from the RTC and write it to the RTIME structure.

3.6.3.2 Parameters

3.6.3.2.1 Parameter1 – dev

A pointer to the RTC device driver structure.

3.6.3.2.2 Parameter2 – time

A pointer to the RTIME structure where the time is to be written

3.6.3.3 Return Value

TRUE on success. FALSE otherwise.

3.6.3.4 Example

```
if(GetRTCTime(&RTCDevice,&time))
{       // 'time' contains the valid time }
```

3.6.4 SetRTCTime

BYTE SetRTCTime(const DEVICE* rtcdev,const RTIME* time)

3.6.4.1 Descriptions

Set the given time to the RTC.

3.6.4.2 Parameters

3.6.4.2.1 Parameter1 – dev

A pointer to the RTC device driver structure.

3.6.4.2.2 Parameter2 – time

A pointer to the RTIME structure containing the time to be written to the RTC.

3.6.4.3 Return Value

TRUE on success. FALSE otherwise.

3.6.4.4 Example

```
if(SetRTCTime(&RTCDevice,&time));
        {
        // Time is set
        }
```

3.7 Video

This module contains wrapper functions, which call the appropriate functions from the given video driver. In addition it provides some higher level functions which are device independent because they act directly on video memory. Basic explanation about the video can be found on the quick start guide.

3.7.1 VideoOn

void VideoOn(const DEVICE* dev)

3.7.1.1 Description

Enable the monitor. After using this function, At least one layer will need to be re-enabled.

3.7.1.2 Parameters

3.7.1.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.1.3 Return Value

None.

3.7.1.4 Example

```
VideoOn();
EnableLayer(dev, BASE_ENABLE);
```

3.7.2 VideoOff

void VideoOff(const DEVICE* dev)

3.7.2.1 Description

Blank the display by turning off the video and disabling all layers.

3.7.2.2 Parameters

3.7.2.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.2.3 Example

```
VideoOff();
```

3.7.3 InitLayer

void InitLayer(const DEVICE* dev, void* layer, BYTE colour_mode, BYTE flip_mode, WORD transparent_colour, WORD width, WORD height, LONG orig_adr0, LONG disp_adr0, LONG orig_adr1, LONG disp_adr1)

3.7.3.1 Description

Define the initial parameters of a screen layer.

3.7.3.1.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.3.1.2 Parameter2 – layer

Pointer to the video device display layer structure.

3.7.3.1.3 Parameter3 – colour_mode

INDIRECT_COLOR	8 bits per pixel paletted.
DIRECT_COLOR	16 bits per pixel.

Window Layer: value ignored. DIRECT_COLOR mode only.

3.7.3.1.4 Parameter4 – flip_mode

LAYER_FRAME0	single buffered frame 0
LAYER_FRAME1	single buffered frame 1

Window and Console Layers: value ignored. Single buffered only.

3.7.3.1.5 Parameter5 – transparent_colour

Transparent colour for the layer. In 16 bit colour or in 8 bit colour, colour = 0 is always transparent. Thus when clearing a layer, the layer will be initially transparent.

This parameter allows for an additional colour to also be declared as transparent

Base Layer: value ignored. Lowest priority layer.

3.7.3.1.6 Parameter6 – width

Width of logical graphics field in pixels.
Must be multiple of 64 bytes.
Must be greater than or equal to HSIZE, the horizontal display width.

3.7.3.1.7 Parameter7 – height

Height of logical graphics field in rasters (pixels)

3.7.3.1.8 Parameter8 – orig_adr0

Base Address of logical graphics field: Memory address of top left pixel.
(Expressed as offset from the start of video memory)

Base and Middle Layers: Applies to frame 0 when double buffering
(Swapping between two frames)

3.7.3.1.9 Parameter9 – disp_adr0

Base Address of display frame: Top left position address.

Base and Middle Layers: Applies to frame 0 when double buffering
(Swapping between two frames)

3.7.3.1.10 Parameter10 – orig_adr1

Base address of logical graphics field. (Expressed as offset from the start of video memory)

Base and Middle Layers: Applies to frame 1 when double buffering
(Swapping between two frames)

Console and Window Layers: Value ignored.

3.7.3.1.11 Parameter11 – disp_adr1

Base Address of display frame: Top left position address.

Base and Middle Layers: Applies to frame 1 when double buffering
(Swapping between two frames)

Console and Window Layers: Value ignored.

3.7.3.2 Return Value

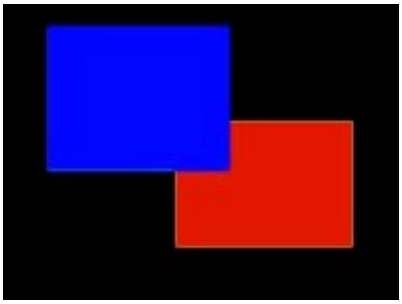
None.

3.7.3.3 Example

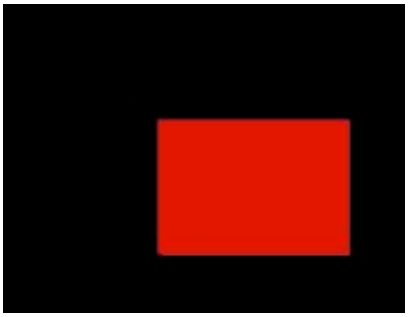
```
screen0=0;                  // 8 bit colour rectangles which will be set into the console layer
screen1=HSIZE*VSIZE;        // 8 bit colour in the middle layer using all colors
/* initialize the console layer to have black transparent */
InitLayer(dev, &CremsonLayer[C_LAYER], INDIRECT_COLOR, LAYER_FRAME0, XXXXXXX, HSIZE,
VSIZE, screen0, screen0, screen0, screen0);
ClearLayer(dev, &CremsonLayer[C_LAYER], 0, HSIZE);
/* initialize the middle layer */
InitLayer(dev, &CremsonLayer[ML_LAYER], INDIRECT_COLOR, LAYER_FRAME0, BLACK, HSIZE,
VSIZE, screen1, screen1, screen1, screen1);
ClearLayer(dev, &CremsonLayer[ML_LAYER], 0, HSIZE);
EnableLayer(dev, CONSOLE_ENABLE);
EnableLayer(dev, MIDDLE_ENABLE);
```

Then a BLUE rectangle is drawn on the console layer and a RED rectangle on the middle layer. A small area of the rectangles is overlaid.

1)
XXXXXX is replaced by BLACK causing BLACK to be the ONLY transparent colour. The result is the following:



2)
XXXXXX is replaced by BLUE causing BLACK and BLUE to be the transparent colours. The result is the following:



3.7.4 ClearLayer

void ClearLayer(const DEVICE* dev,void* layer, BYTE scrn, LONG width)

3.7.4.1 Description

Fast clear of logical Layer to colour=0x0000 (usually transparent black)

3.7.4.2 Parameters

3.7.4.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.4.2.2 Parameter2 – layer

Pointer to the video device display layer structure.

3.7.4.2.3 Parameter3 – scrn

0 Clear frame 0 of double buffer memory.
1 Clear frame 1 of double buffer memory.
When doing double buffering, this function will have to be called twice to clear the two areas of memory.

3.7.4.2.4 Parameter4 – width

Width in pixels of the area to be cleared. Usually width = layer width, but smaller strips from left hand edge can be specified for clearing.

Width must satisfy 8 <= width <= layer width

3.7.4.3 Return Value

None.

3.7.4.4 Example

ClearLayer((&CremsonDevice, &CremsonLayer[BL_LAYER], 0, HSIZE);

3.7.5 clear_video_memory

void clear_video_memory(WORD color, ULONG offset, ULONG size);

3.7.5.1 Description

Slower clear of logical layer to the given colour. This function can be used on 8 bit colours only.

3.7.5.2 Parameters

3.7.5.2.1 Parameter1 - colour

Colour to fill the memory with. This is a colour index (0-255) into and array of RGB values.

3.7.5.2.2 Parameter2 - offset

Offset from the base of video memory to start clearing. The graphics board driver sets the base address when it is installed.

3.7.5.2.3 Parameter1 - size

Number of BYTES/pixels to clear.

3.7.5.3 Return Value

None.

3.7.6 SetWindowLayer

void SetWindowLayer(const DEVICE* dev, WORD xpos, WORD ypos, WORD width, WORD height)

3.7.6.1 Description

Controls the size and position of the window layer:

This layer is slightly different to the other layers. It has variable screen position and size, unlike the other layers which are all full screen.

3.7.6.2 Parameters

3.7.6.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.6.2.2 Parameter1 – xpos

Window Top left Screen X Position in pixels.

3.7.6.2.3 Parameter1 – ypos

Window Top left Screen Y Position in pixels.

3.7.6.2.4 Parameter1 – width

Window screen width in pixels.

3.7.6.2.5 Parameter1 – height

Window screen height in pixels.

The ***window layer*** is initialised with **InitLayer()** which has slightly different parameters in this case
Parameters:

- dev: device
- layer: layer identifier
- colour_mode: INDIRECT_COLOR/ DIRECT_COLOR
- flip_mode: (*don't care)*
- transparent_colour: (*don't care)*

- width: (*don't care)*
- height: (*don't care)*
- orig_adr0: screen origin address
- disp_adr0: display start address
- orig_adr1: *(don't care)*
- disp_adr1: (*don't care)*

```
InitLayer(&CremsonDevice, &CremsonLayer[W_LAYER],INDIRECT_COLOR, 0, 0, HSIZE, 0, 0, 0, 0,0);
EnableLayer(&CremsonDevice, WINDOW_ENABLE);
xpos=ypos=100;
width=120; height=200;
SetWindowLayer(&CremsonDevice, xpos, ypos, width, height);

/* gives 120x200 pixel window at screen coordinates x=100, y=100*/
```

3.7.7 ScreenPartitionX

void ScreenPartitionX (const DEVICE* dev, WORD x)

3.7.7.1 Description

Base and Middle Layers: Specify the position of the left field – right field partition.

3.7.7.2 Parameters

3.7.7.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.7.2.2 Parameter2 – x

Partition position. X must satisfy: 1 <= x <= HSIZE

3.7.7.3 Return Value

None.

3.7.7.4 Example

```
ScreenPartitionX(&CremsonDevice, HSIZE/2);    // Divide screen vertically in two)
```

3.7.8 ScrollLayer

void ScrollLayer(const DEVICE* dev,void* layer, ULONG xoff, ULONG yoff)

3.7.8.1 Description

Scroll a display layer by applying horizontal or vertical offset to display base address.
The offset is applied at the next vertical interrupt. The current x and y display offsets are stored within the driver.

3.7.8.2 Parameters

3.7.8.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.8.2.2 Parameter2 – layer

Pointer to the video device display layer structure.

3.7.8.2.3 Parameter2 – xoff

Horizontal offset in bytes.

8 bit colour mode: 1 pixel = 1 byte
16 bit colour mode: 1 pixel = 2 bytes

xoff must satisfy:

0 <= xoff <= layer width (8 bit colour)
0 <= xoff <= 2*layer width (16 bit colour)

3.7.8.2.4 Parameter2 – yoff

Vertical offset in rasters (pixels)

3.7.8.3 Return Value

None.

3.7.8.4 Example

```
for (xoff=0, yoff=0; xoff < HSIZE; xoff++)
        ScrollLayer(&CremsonDevice, &CremsonLayer[BL_LAYER], xoff, yoff);
```

3.7.9 EnableLayer

void EnableLayer(const DEVICE* dev,UBYTE select)

3.7.9.1 Description

Enable a screen layer on the display.

3.7.9.2 Parameters

3.7.9.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.9.2.2 Parameter2 – select

The layer to be enabled:
CONSOLE_ENABLE
WINDOW_ENABLE
MIDDLE_ENABLE
BASE_ENABLE

3.7.9.3 Return Value

None.

3.7.9.4 Example

```
EnableLayer(&CremsonDevice, BASE_ENABLE);        // enables the BASE Layer (Left and Right Fields)
```

3.7.10 DisableLayer

void DisableLayer(const DEVICE* dev,UBYTE select)

3.7.10.1 Description

Disable a screen layer on the display.

3.7.10.2 Parameters

3.7.10.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.10.2.2 Parameter2 – select

Layer to be disabled:
CONSOLE_ENABLE
WINDOW_ENABLE
MIDDLE_ENABLE
BASE_ENABLE

3.7.10.3 Return Value

None.

3.7.10.4 Example

DisableLayer(&CremsonDevice, CONSOLE_ENABLE);

3.7.11 LoadFont

void LoadFont(const DEVICE* dev,const ULONG* font)

3.7.11.1 Description

Load the given font into video memory. This becomes the font used by the text functions. If changing the font in the process of writing, the display will move to the next line but keep track of horizontal position.

3.7.11.2 Parameters

3.7.11.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.11.2.2 Parameter2 – font

Pointer to the font data to load. Font files can be generated using the utilities provided (refer to quick start guide for more information).

3.7.11.3 Return Value

None.

3.7.11.4 Example

LoadFont(dev,tempo16); //load font into the driver

3.7.12 TextColour

void TextColour(const DEVICE* dev,UWORD forcol,UWORD bakcol)

3.7.12.1 Description

Set the foreground and background colour for the text. Text written after calling this will use the colour specified. Text already written will be unchanged.

3.7.12.2 Parameters

3.7.12.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.12.2.2 Parameter2 – forcol

Foreground colour for the text. This value is the position in the palette of the colour to be used (0 - 255)

3.7.12.2.3 Parameter3 – bakcol

Background colour for the text. This value is the position in the palette of the colour to be used (0 - 255)

3.7.12.3 Return Value

None.

3.7.12.4 Example

Set text colours to black on white.
TextColour(&CremsonDevice,BLACK,WHITE);

If the standard VGA palette is loaded, colours 0-15 are the standard IBM colours as follows:
- (defined in graphics.h)
0 BLACK
1 BLUE
2 GREEN
3 CYAN
4 RED

5 MAGENTA
6 BROWN
7 LIGHTGRAY
8 DARKGRAY
9 LIGHTBLUE
10 LIGHTGREEN
11 LIGHTCYAN
12 LIGHTRED
13 LIGHTMAGENTA
14 YELLOW
15 WHITE

3.7.13 InitDrawframe

void InitDrawframe(const DEVICE* dev, WORD mode, LONG base, WORD xres)

3.7.13.1 Description

Initialise the drawing frame which is used to define the screen memory location and dimensions when Drawing and Blit Transfer commands are used

3.7.13.2 Parameters

3.7.13.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.13.2.2 Parameter1 – mode

Mode= 0x8000 - 16 bit colour
Mode= 0x0000 - 8 bit colour

3.7.13.2.3 Parameter1 – base

Base address of drawing frame relative to start of video memory.

3.7.13.2.4 Parameter1 – xres

Horizontal resolution of drawing frame in pixels.

3.7.13.3 Return Value

None.

3.7.13.4 Example

Screen0 = 0; HSIZE = 640;
InitDrawframe(&CremsonDevice, 0, screen0, HSIZE);

3.7.14 SetCursor

void SetCursor(const DEVICE* dev, LONG screenbase,WORD x,WORD y)

3.7.14.1 Description

Place the cursor at the given co-ordinates relative. This function will mostly be used when writing text into screen.

3.7.14.2 Parameters

3.7.14.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.14.2.2 Parameter2 – screenbase

This parameter is always 0. To ensure user is setting the cursor at the correct video memory representing the start of the screen it is strongly recommended to use InitDrawframe before. Then the cursor will be set from the start of this screen.

3.7.14.2.3 Parameter3– x

X co-ordinate of the cursor.

3.7.14.2.4 Parameter4– y

Y co-ordinate of the cursor.

3.7.14.3 Return Value

None.

3.7.14.4 Example

Set cursor to top left corner of a screen define to be at HSIZE*VSIZE in the video memory.

```
LONG screen1=HSIZE*VSIZE;
InitDrawframe(dev, 0, screen1, HSIZE);  //init the drawing frame to be medium layer.
SetCursor(dev,0,0,16);          // Set the cursor at x=0;Y=16 of the current drawing frame
LoadFont(dev,tempo16);          //load font into the driver
VPutStr(dev,"hello");           //write “hello” at cursor
```

3.7.15 VPutChar

void VPutChar(const DEVICE* dev,UBYTE ch)

3.7.15.1 Description

Write the given character at the current cursor position and advance the cursor to the next position. Character \r and \n need to be used to move to the beginning of a new line.

***Note*:** writing position is BOTTOM LEFT of character (refer to quick start guide: introduction to video programming).

3.7.15.2 Parameters

3.7.15.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.15.2.2 Parameter2 – ch

Character to write.

3.7.15.3 Return Value

None.

3.7.15.4 Example

Write the letter ‘A’ at the current cursor position.

```
VPutChar(&CremsonDevice,’A’);
```

3.7.16 VputStr

void VPutStr(const DEVICE* dev,const char* s)

3.7.16.1 Description

Write the text at the current cursor position. Text automatically wraps to the next line if necessary.

3.7.16.2 Parameters

3.7.16.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.16.2.2 Parameter1 – s

String to write to the screen.

3.7.16.3 Return Value

None.

3.7.16.4 Example

Write to the screen at current cursor position.
VPutStr(&CremsonDevice,"Hello World");

3.7.17 Loadpalette, LoadCpalette, loadMBPalette

void LoadCPalette(const DEVICE* dev,const RGB* palette)
void LoadMBPalette(const DEVICE* dev,const RGB* palette)
void LoadPalette(const DEVICE* dev,const RGB* palette)

3.7.17.1 Description

Load the given colour palette. Colour palettes are used in 256 colour mode. The Cremson Device has two colour palettes: one for the Middle/ Base Layers, and one for the Console Layer. There are three similar functions:
LoadCPalette - loads the Console Layer palette
LoadMBPalette - loads the Middle/Base Layer palette
LoadPalette - loads both palettes identically

3.7.17.2 Parameters

3.7.17.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.17.2.2 Parameter2 – palette

Pointer to an array of 256 RGB structures which define the colours of the palette.

3.7.17.3 Return Value

None.

3.7.17.4 Example

LoadPalette(palette_table);

3.7.18 GetPalette, GetMBPalette, GetCPalette

void GetPalette(const DEVICE* dev,UBYTE entry,RGB* col)
void GetMBPalette(const DEVICE* dev,UBYTE entry,RGB* col)
void GetCPalette(const DEVICE* dev,UBYTE entry,RGB* col)

3.7.18.1 Description

Read the red, green, blues values of the colour at a given palette index.

GetPalette (obsolete) reads the Middle/Base Layer palette
GetMBPalette reads the Middle/Base Layer palette
GetCPalette reads the Console Layer palette

3.7.18.2 Parameters

3.7.18.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.18.2.2 Parameter3 – entry

Position in the palette the colour values are to be read from.

3.7.18.2.3 Parameter3 – col

Pointer to an RGB structure which the RGB values are to be written to.

3.7.18.3 Return Value

None.

3.7.18.4 Example

Read palette entry 16 into variable ‘col’
RGB col;
GetPalette(&CremsonDevice,16,&col)

3.7.19 SetPalette SetMBPalette SetCPalette

void SetPalette(const DEVICE* dev,UBYTE entry,RGB* col)
void SetMBPalette(const DEVICE* dev,UBYTE entry,RGB* col)
void SetCPalette(const DEVICE* dev,UBYTE entry,RGB* col)

3.7.19.1 Description

Set the RGB values for the colour at the given position in the palette.

SetPalette sets both palettes.
SetMBPalette sets the Middle/Base Layer palette.
SetCPalette sets the Console Layer palette.

3.7.19.2 Parameters

3.7.19.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.19.2.2 Parameter2 – entry

Position in the palette the colour values are to be written to.

3.7.19.2.3 Parameter3 – col

Pointer to an RGB structure which contains the RGB values to be written to the palette entry.

3.7.19.3 Return Value

None.

3.7.19.4 Example

Lighten the colour at position 16 in the palette.

GetPalette(&CremsonDevice,16,&col);
Col.red+=20;
Col.green+=20;
Col.blue+=20;
SetPalette(&CremsonDevice,16,&col);

3.7.20 SetColorMode

void SetColorMode(const DEVICE* *dev*, UBYTE *mode*)

3.7.20.1 Description

Set the current colour mode

3.7.20.2 Parameters:

3.7.20.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.20.2.2 Parameter2 – mode

INDIRECT_COLOR 8 bit per pixel colour depth
DIRECT_COLOR 16 bit per pixel colour depth

3.7.20.3 Return Value

None.

3.7.20.4 Example

```
/* Initialise BASE layer:    screen0, 16 bit colour */
SetColorMode(dev, DIRECT_COLOR);
InitLayer(dev, &CremsonLayer[BL_LAYER], DIRECT_COLOR, LAYER_FRAME0, BLACK, HSIZE, VSIZE,
screen0, screen0, screen0, screen0);
ClearLayer(dev, &CremsonLayer[BL_LAYER], LAYER_FRAME0, HSIZE);

/* Initialise CONSOLE layer: screen1,  8 bit colour */
SetColorMode(dev, INDIRECT_COLOR);
InitLayer(dev, &CremsonLayer[C_LAYER], INDIRECT_COLOR, LAYER_FRAME0, BLACK, HSIZE,
VSIZE, screen1, screen1, screen1, screen1);
ClearLayer(dev, &CremsonLayer[C_LAYER], LAYER_FRAME0, HSIZE);

/* 16 bit color BLIT: cherry sprite to the Base Layer, screen 0 at x=200, y=200 */
SetColorMode(dev, DIRECT_COLOR);
PutSprite(dev, screen0,   200, 200, sprite1, cherry_width, cherry_height, 1);

/* 8 bit color BLIT:  coin sprite to the Console Layer, screen 1  at x=200, y=0 */
SetColorMode(dev, INDIRECT_COLOR);
PutSprite(dev, screen1, 200, 0, sprite2, coin_width, coin_height, 1);
```

3.7.21 GetColorMode

UBYTE GetColorMode(const DEVICE* dev)

3.7.21.1 Description

Return the current colour mode

3.7.21.2 Parameters:

3.7.21.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.21.3 Return Value

INDIRECT_COLOR	8 bit per pixel colour depth
DIRECT_COLOR	16 bit per pixel colour depth

3.7.22 LoadSprite

LONG LoadSprite(const DEVICE* dev,const WORD* startaddr,LONG offset,LONG size)

3.7.22.1 Description

Load the sprite image into video memory at the given offset address. (More information on how to generate an image file format which can be use by the pluto5 board from a standard image format can be found in the Quick Start guide).

3.7.22.2 Parameters

3.7.22.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.22.2.2 Parameter2 – startaddr

Address in system memory of the image data.

3.7.22.2.3 Parameter3 – offset

Offset into video memory where the image data is to be loaded.

3.7.22.2.4 Parameter3 – size

Size in **BYTE** of the image data.

3.7.22.3 Return Value

Offset into video memory of the next free BYTE after the image has loaded.
Due to constraints of alignment this may vary from 'offset+size'

3.7.22.4 Example

Load image1 and image2 into off screen memory.

```
free=HSIZE*VSIZE;
free=LoadSprite(&CremsonDevice,&image1,free,image1_width*image1_height);
free=LoadSprite(&CremsonDevice,&image2,free,image2_width*image2_height);
```

3.7.23 LoadImage

LONG LoadImage(const DEVICE* dev,const WORD* startaddr,LONG offset,LONG size)

3.7.23.1 Description

Load the image into video memory at the given offset address. . (More information on how to generate an image file format which can be use by the pluto5 board from a standard image format can be found in the Quick Start guide).

This function should be used for images which are not moved around the screen (e.g background images) Use 'LoadSprite' for images which need to move around the screen.

3.7.23.2 Parameters

3.7.23.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.23.2.2 Parameter2 – startaddr

Address in system memory of the image data.

3.7.23.2.3 Parameter3 – offset

Offset into video memory where the image data is to be loaded.

3.7.23.2.4 Parameter3 – size

Size in BYTE of the image data.

3.7.23.3 Return Value

Offset into video memory of the next free BYTE after the image has loaded.
Due to constraints of alignment this may vary from 'offset+size'

3.7.23.4 Example

Load image1 and image2 into off screen memory.

```
free=HSIZE*VSIZE;
free=LoadImage(&CremsonDevice,&image1,free,image1_width*image1_height);
free=LoadImage(&CremsonDevice,&image2,free,image2_width*image2_height);
```

3.7.24 CremsonByteOrder

void CremsonByteOrder(const DEVICE* dev, UBYTE order)

3.7.24.1 Description

When loading 8 bit image files built with Image Alchemy (.raw format), the byte order is incorrect: the high byte and low byte is reversed. Issue 23 of the Peripherals Library has a new function: **CremsonByteOrder()**
This sets the byte order when loading video memory.

3.7.24.1.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.24.1.2 Parameter2 – order

NORMAL 0 => 16 bit pictures
REVERSE 1 => 8 bit pictures

3.7.24.2 Example

```
SetColorMode(&CremsonDevice, INDIRECT_COLOR);
CremsonByteOrder(&CremsonDevice, REVERSE);

if(FileOpen(&ATADevice, command.parameters, &file[0], READ_ONLY) != NULL)
{
FileLoad(&ATADevice, &file[0], (UWORD*)(VideoMemBase+(offscreen/SIZEOFWORD)));
PutImage(&CremsonDevice,screen0,0,0,(UWORD*)(VideoMemBase+(offscreen/SIZEOFWORD)));
CremsonByteOrder(&CremsonDevice, NORMAL);
}
```

3.7.25 GetImageX

LONG GetImageX(const WORD* image)

3.7.25.1 Description

Return the horizontal size of the given image.

3.7.25.2 Parameters

3.7.25.2.1 Parameter1 – image

Absolute address in memory of the image data. The image data should be in alchemy .RAW format, which is the format used to store images on CDRom.

3.7.25.3 Return Value

The horizontal size in pixels of the given image.

3.7.25.4 Example

```
Xsize=GetImageX(filedata);      //'filedata' is the address of the image in memory.
```

3.7.26 GetImageY

LONG GetImageY(const WORD* image)

3.7.26.1 Description

Return the vertical size of the given image.

3.7.26.2 Parameters

3.7.26.2.1 Parameter1 – image

Absolute address in memory of the image data. The image data should be in alchemy .RAW format, which is the format used to store images on CDRom.

3.7.26.3 Return Value

Vertical size in pixels of the given image.

3.7.26.4 Example

Get the size of the images from a .raw file loaded from compact flash card. This image has a 800 bytes header.

```
FileLoad(&ATADevice, &file[0], (UWORD*)(VideoMemBase+offscreen/SIZEOFWORD));
image_width=GetImageX(VideoMemBase+offscreen/SIZEOFWORD);
image_height=GetImageY(VideoMemBase+offscreen/SIZEOFWORD);
```

3.7.27 GetImageStart

LONG GetImageStart(const WORD* image)

3.7.27.1 Description

Return the start address of the actual image data. This skips the file header present in image alchemy .RAW format.

3.7.27.2 Parameters

3.7.27.2.1 Parameter1 – image

Absolute address in memory of the image data. The image data should be in alchemy .RAW format, which is the format used to store images on CDRom.

3.7.27.3 Return Value

Start address of the image data.

3.7.28 PutImage

void PutImage(const DEVICE* dev,LONG screenbase, LONG x, LONG y, const WORD* image)

3.7.28.1 Description

Put an image onto the screen at the given co-ordinates.

3.7.28.2 Parameters

3.7.28.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.28.2.2 Parameter2 – screenbase

Offset into video memory which represents the start of the screen the co-ordinates refer to.

3.7.28.2.3 Parameter3 – x

X co-ordinate of the on screen point where the top left corner of the image will be.

3.7.28.2.4 Parameter4 – y

Y co-ordinate of the on screen point where the top left corner of the image will be.

3.7.28.2.5 Parameter5 - image

The absolute address in video memory where the image data resides.

3.7.28.3 Example

```
PutImage(&CremsonDevice,0,x,y,filedata);        // Put background on screen
```

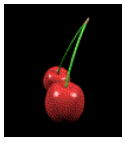
3.7.29 PutSprite

void PutSprite(const DEVICE* dev,LONG screenbase,LONG x, LONG y,LONG image_start,LONG xsize,LONG ysize,LONG mode)

3.7.29.1 Description

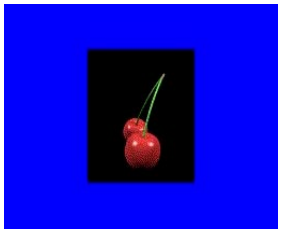
Copy the sprite image from off screen video memory to the defined screen co-ordinates.
The last parameter of the function controls sprite transparency.
This is transparency within a screen layer , and is useful for blitting irregular shapes, for example:

The background colour was set to be blue and a user wants to load this sprite to the display screen.

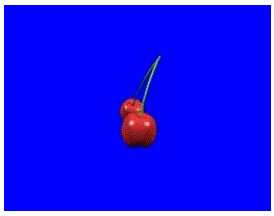


The picture represents a cherry on a black background.

If loading this picture without transparency the result is the following where the black background was loaded into the screen:



If loading this picture with transparency the result is the following where the black background of the picture is not modifying the blue background of the layer:



Note: The transparent colour within the sprite image is always colour=0

Colour=0 is usually black, but in 8-bit colour mode it can be mapped to a different display colour by changing the palette.

3.7.29.2 Parameters

3.7.29.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.29.2.2 Parameter2 – screen

Offset into video memory which represents the start of the screen the co-ordinates refer to.

3.7.29.2.3 Parameter3 – x

X co-ordinate of top left corner of screen area.

3.7.29.2.4 Parameter4 – y

Y co-ordinate of top left corner of screen area.

3.7.29.2.5 Parameter4 – imagestart

Offset into video memory where the sprite image has previously been loaded.

3.7.29.2.6 Parameter6 – xsize

Horizontal size in pixels of the sprite image.

3.7.29.2.7 Parameter7 – ysize

Vertical size in pixels of the sprite image.

3.7.29.2.8 Parameter7 – mode

Mode of copy. Use zero for normal mode or non zero for transparency within the layer.

3.7.29.3 Example

Copy the image from off screen memory immediately following the screen memory to the defined screen co-ordinates using transparent mode. In transparent mode colour 0 will be transparent.
*LONG screen=HSIZE*VSIZE; // Video memory addresses*
PutSprite(dev,screen, 150, 0,0x0100000,cherry1_width,cherry1_height,1);

3.7.30 ClearSprite

void ClearSprite(const DEVICE* dev, LONG screenbase, LONG x, LONG y, LONG xsize, LONG ysize)

3.7.30.1 Description

Clear area previously written to by sprite

3.7.30.2 Parameters

3.7.30.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.30.2.2 Parameter2 – screenbase

Offset into video memory which represents the start of the screen the co-ordinates refer to.

3.7.30.2.3 Parameter3 – x

X co-ordinate of the on screen point where the top left corner of the image will be.

3.7.30.2.4 Parameter4 – y

Y co-ordinate of the on screen point where the top left corner of the image will be.

3.7.30.2.5 Parameter5 – xsize

Horizontal size in pixels of the sprite image.

3.7.30.2.6 Parameter6 – ysize

Vertical size in pixels of the sprite image.

3.7.30.3 Return Value

None.

3.7.30.4 Example

Copy the image from off screen memory immediately following the screen memory to the defined screen co-ordinates using transparent mode. In transparent mode colour 0 will be transparent.
*LONG screen=HSIZE*VSIZE; // Video memory addresses*
PutSprite(dev, screen, 150, 0,0x0100000,cherry1_width,cherry1_height,1);
And then delete it:
PutSprite(dev, screen, 150, 0,cherry1_width,cherry1_height);

3.7.31 MakeSprite

LONG MakeSprite(const DEVICE* dev, WORD* startaddr,LONG offset)

3.7.31.1 Description

Create sprite image in memory from downloaded Image Alchemy raw file.

When loading sprites from **eprom** to video memory, function LoadSprite() is used. This loads the image data, and a *mask* (header) of the image data into video memory. This is required for sprites that are transparent *within* a screen layer.

Issue 24 of the Peripherals Library has a new function: **MakeSprite()**. This will create a sprite in video memory from a *.raw* image file loaded from **Compact Flash**. This function creates a transparent sprite (image + mask) from the image data

3.7.31.2 Parameters

3.7.31.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.31.2.2 Parameter2 – startaddr

Pointer on a **WORD** where the sprite was downloaded before. (header+image)

3.7.31.2.3 Parameter3 – offset

Offset into video memory which represents the start where the image will be moved (only image now)
Horizontal size in pixels of the screen area to save.

3.7.31.3 Return Value

Next free byte of video memory after that used by this sprite.

3.7.31.4 Example

```
if(FileOpen(&ATADevice, fname, &file, READ_ONLY) != NULL)
{

FileLoad(&ATADevice, &file, (UWORD*)(VideoMemBase+screen2/SIZEOFWORD));//we load the sprite always at the same place: screen2 careful there is the header
//From there two ways to create your sprite:
add_sprite=MakeSprite(&CremsonDevice,(UWORD*)(VideoMemBase+(screen2/SIZEOFWORD)),offset);
        //function considers already the header (implicit)

// or the equivqlent with a known function:
//add_sprite=LoadImage(&CremsonDevice,(UWORD*)(VideoMemBase+(SIZE_HEADER+screen2)/SIZEOFWORD),offset,width*height);    //function does not consider the header. You need to remove header manually

}
```

3.7.32 SaveBackground

void SaveBackground(const DEVICE* dev,LONG screenbase,LONG x,LONG y,LONG saveaddress, LONG xsize,LONG ysize)

3.7.32.1 Description

Save an area of screen memory defined by the screen co-ordinates and the xsize, ysize parameters to the given save address. This function is normally used to save a background before drawing a sprite on the screen so that the background can be restored when the sprite moves.

3.7.32.2 Parameters

3.7.32.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.32.2.2 Parameter2 – screen

Offset into video memory which represents the start of the screen the co-ordinates refer to.

3.7.32.2.3 Parameter3 – x

X co-ordinate of top left corner of screen area.

3.7.32.2.4 Parameter4 – y

Y co-ordinate of top left corner of screen area.

3.7.32.2.5 Parameter5 – saveaddress

Offset into video memory where the screen area is to be saved.

3.7.32.2.6 Parameter6 – xsize

Horizontal size in pixels of the screen area to save.

3.7.32.2.7 Parameter7 – ysize

Vertical size in pixels of the screen area to save.

3.7.32.3 Return Value

None.

3.7.32.4 Example

Save a 100x100 square at the top left (x=0,y=0) of the screen whose start is at offset zero from the video memory start to the ofscreen memory immediatedly following the screen.
SaveBackground(&CremsonDevice,0,0,0,640*480,100,100*);*

3.7.33 RestoreBackground

void RestoreBackground(const DEVICE* dev,LONG screenbase,LONG x, LONG y,LONG saveaddress,LONG xsize,LONG ysize)

3.7.33.1 Description

Restore a previously saved area of screen memory defined by the screen co-ordinates and the xsize, ysize parameters to the given save address. Normally used to restore a background after a sprite has moved.

3.7.33.2 Parameters

3.7.33.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.33.2.2 Parameter2 – screen

Offset into video memory which represents the start of the screen the co-ordinates refer to.

3.7.33.2.3 Parameter3 – x

X co-ordinate of top left corner of screen area.

3.7.33.2.4 Parameter4 – y

Y co-ordinate of top left corner of screen area.

3.7.33.2.5 Parameter5 – saveaddress

Offset into video memory where the screen area has previously been saved.

3.7.33.2.6 Parameter6 – xsize

Horizontal size in pixels of the screen area to restore.

3.7.33.2.7 Parameter7 – ysize

Vertical size in pixels of the screen area to restore.

3.7.33.3 Return Value

None.

3.7.33.4 Example

Restore the screen area saved in the last example.
RestoreBackground(&CremsonDevice,0,0,0,640*480,100,100*);*

3.7.34 BltBusy

UBYTE BltBusy(const DEVICE* dev)

3.7.34.1 Description

Test the status of the 2D accelerator (Bit Blatter).

3.7.34.2 Parameters

3.7.34.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.34.3 Return Value

TRUE if the bit blatter is busy.
FALSE otherwise.

3.7.35 Line

void Line(LONG screenbase,LONG Ax, LONG Ay, LONG Bx, LONG By, UWORD color)

3.7.35.1 Description

Draw a line between the defined screen points A & B in the given colour.

3.7.35.2 Parameters

3.7.35.2.1 Parameter1 – screenbase

Offset into video memory which defines the start of the screen to draw on.

3.7.35.2.2 Parameter1 – Ax

X co-ordinate of point A

3.7.35.2.3 Parameter1 – Ay

Y co-ordinate of point A

3.7.35.2.4 Parameter1 – Bx

X co-ordinate of point B

3.7.35.2.5 Parameter1 – By

Y co-ordinate of point B

3.7.35.2.6 Parameter1 – colour

Colour to draw line.

3.7.35.3 Return Value

None.

3.7.35.4 Example

Draw a red diagonal line.

```
Line(0,0,0,640,480,RED);
```

3.7.36 Circle

void Circle(LONG screen,LONG xcentre, LONG ycentre, LONG radius, UBYTE color)

3.7.36.1 Description

Draw an empty circle at the defined screen co-ordinate with the given radius and using the given colour.

3.7.36.2 Parameters

3.7.36.2.1 Parameter1 – screenbase

Offset into video memory which defines the start of the screen to draw on.

3.7.36.2.2 Parameter2 – xcentre

X co-ordinate of centre point of circle

3.7.36.2.3 Parameter3 – ycentre

Y co-ordinate of point of circle

3.7.36.2.4 Parameter4 – radius

Radius of circle.

3.7.36.2.5 Parameter5 – colour

Colour to draw line.

3.7.36.3 Return Value

3.7.36.4 Example

Draw a blue circle at the centre of the screen having a radius of 200
Circle(0,HZISE/2,VSIZE/2,200,BLUE);

3.7.37 Rectangle

void Rectangle(LONG screen,LONG left,LONG top,LONG right,LONG bottom,UBYTE color)

3.7.37.1 Description

Draw an empty rectangle defined by the above parameters.

3.7.37.2 Parameters

3.7.37.2.1 Parameter1 – screenbase

Offset into video memory which defines the start of the screen to draw on.

3.7.37.2.2 Parameter2 – left

X co-ordinate of the left edge of the rectangle.

3.7.37.2.3 Parameter3 – top

Y co-ordinate of the top edge of the rectangle.

3.7.37.2.4 Parameter4 – right

X co-ordinate of the right edge of the rectangle.

3.7.37.2.5 Parameter5 – bottom

Y co-ordinate of the bottom edge of the rectangle.

3.7.37.2.6 Parameter5 – colour

Colour to draw.

3.7.37.3 Return Value

None.

3.7.37.4 Example

Draw a blue 100x100 square at the top left corner of the screen
Rectangle(0,0,0,100,100 ,BLUE);

3.7.38 PutPixel

void PutPixel(const DEVICE* dev,LONG screen,LONG x, LONG y, UWORD color)

3.7.38.1 Description

Write the pixel the defined screen address with the given colour.

3.7.38.2 Parameters

3.7.38.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.38.2.2 Parameter2 – screen

Offset into video memory which represents the start of the screen to write the pixel on?

3.7.38.2.3 Parameter3 – x

X co-ordinate of pixel.

3.7.38.2.4 Parameter4 – y

Y co-ordinate of pixel.

3.7.38.2.5 Parameter5 – colour

Colour to write to pixel, this is the position in the palette (0-255) which holds the colour values required.

3.7.38.3 Return Value

None.

3.7.38.4 Example

Draw a white pixel at screen co-ordinates x=100,y=120, on the screen starting at offset zero from the start of video memory.
PutPixel(&CremsonDevice,0,100,120,WHITE);

3.7.39 GetPixel

UWORD GetPixel(const DEVICE* dev,LONG screen,LONG x, LONG y)

3.7.39.1 Description

Get the colour of the pixel at the defined screen address.

3.7.39.2 Parameters

3.7.39.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.39.2.2 Parameter2 – screen

Offset into video memory which represents the start of the screen to write the pixel on?

3.7.39.2.3 Parameter3 – x

X co-ordinate of pixel.

3.7.39.2.4 Parameter4 – y

Y co-ordinate of pixel.

3.7.39.3 Return Value

Colour of the pixel, this is the position in the palette (0-255) which holds the colour values of the pixel.

3.7.39.4 Example

Lighten the colour of the pixel at screen co-ordinates x=100, y=120, on the screen starting at offset zero from the start of video memory.

```
RGB     col;
col=GetPixel(&CremsonDevice,0,100,120);
col.red+=20;
col.green+=20;
col.blue+=20;
SetPixel(&CremsonDevice,0,0,0,&col)
```

3.7.40 GetScreen

LONG GetScreen(const DEVICE* dev,BYTE scrn)

3.7.40.1 Description

Return the offset into video memory which represents the start address of the given screen.

3.7.40.2 Parameters

3.7.40.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.40.2.2 Parameter2 – scrn

Screen number (one or zero).

3.7.40.3 Return Value

Offset to the given screen base.

3.7.40.4 Example

```
Screen0=GetScreen(&CremsonDevice,0);
```

3.7.41 ScreenFree

BYTE ScreenFree(const DEVICE* dev, BYTE scrn)

3.7.41.1 Description

Test the status of the given screen to see if it is free to write to. This function will mostly be used when double buffering data.

3.7.41.2 Parameters

3.7.41.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.41.2.2 Parameter2 – scrn

Screen index of the screen to test. This function will only return a valid result if an automatic double buffering has been started. The only valid values of this parameter are zero and one.

3.7.41.3 Return Value

TRUE if the given screen is free for writing.

3.7.41.4 Example

Wait for screen zero to become free for writing.

```
while(!ScreenFree(dev,0)); //wait until one of the buffered screen is fully filled
```

3.7.42 SetDoubleBuffer

void SetDoubleBuffer(const DEVICE* dev,LONG scrn0,LONG scrn1,BYTE rate)

3.7.42.1 Description

Set up the double buffering parameters for all the layers. As a result parameter 2 and parameter 3 are not used anymore.

3.7.42.2 Parameters

3.7.42.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.42.2.2 Parameter2 – scrn0

NOT USED.

3.7.42.2.3 Parameter3 – scrn1

NOT USED.

3.7.42.2.4 Parameter4 – rate

The rate at which to toggle between screen screens. Use on of the following system #defines.
SINGLE_BUFFER
FRAME60HZ
FRAME30HZ
FRAME20HZ
FRAME15HZ
FRAME12HZ
FRAME10HZ
FRAME5HZ
FRAME2HZ

3.7.42.3 Return Value

None.

3.7.42.4 Example

```
screen0=0;                  // Middle Left Layer screen 0 (double buffered)
screen1=HSIZE*VSIZE;        // Middle Left Layer screen 1 (double buffered)
InitLayer(dev, &CremsonLayer[ML_LAYER], INDIRECT_COLOR, LAYER_FRAME0, BLACK, HSIZE, VSIZE, screen0, screen0, screen1, screen1); //double buffered using screen 0 and screen 1
ClearLayer(dev, &CremsonLayer[ML_LAYER], 0, HSIZE);      //clear memory area from origin address0
ClearLayer(dev, &CremsonLayer[ML_LAYER], 1, HSIZE);      //clear memory area from origin address1
SetDoubleBuffer(dev,0,0,FRAME60HZ);
```

3.7.43 ScreenToggle

void ScreenToggle(const DEVICE* dev, WORD enable)

3.7.43.1 Description

Enable/ disable screen toggling in automatic double buffer mode.
This function is useful to prevent toggling until screen updates are complete.

3.7.43.2 Parameters

3.7.43.2.1 Parameter1 – dev

Pointer to the video device driver structure.

3.7.43.2.2 Parameter2 – enable

1 enable
0 disable

3.7.43.3 Return Value

None.

3.8 CD-ROM

This module contains wrapper functions, which call the appropriate functions from the given CD-ROM driver.

3.8.1 InitialiseDrive

CDROMSTS InitialiseDrive(const DEVICE* dev)

3.8.1.1 Description

This function initialises the given CDRom drive. The device is polled waiting for an OK status then the CD is locked into the drive ready for access. If the device does not return an OK status within the timeout (specified in the driver configuration data) then after the timeout the appropriate error condition is returned.

3.8.1.2 Parameters

3.8.1.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.1.3 Return Value

Return the CD-ROM status. For details of the CDROMSTS type refer to cdrom.h.

3.8.1.4 Example

```
if(InitialiseDrive(&CDRomDevice)==OK)
{
	//access CD drive
}
else
{
	// Report error
}
```

3.8.2 DriveStatus

CDROMSTS DriveStatus(const DEVICE* dev)

3.8.2.1 Description

Test the drive status.

3.8.2.2 Parameters

3.8.2.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.2.3 Return Value

Return the CDRom status. For details of the CDROMSTS type refer to cdrom.h

3.8.2.4 Example

```
if(DriveStatus(&CDRomDevice)==OK)
{
	//access CD drive
}
else
```

```
{
        // Report error
}
```

3.8.3 DriveEnquiry

CDROMSTS DriveEnquiry(const DEVICE* dev,char* s)

3.8.3.1 Description

Issue an enquiry command to the drive and build a text string containing the drive information reported (e.g. manufacturer etc).

3.8.3.2 Parameters

3.8.3.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.3.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.8.3.4 Example

```
char str[128];

if(DriveEnquiry(&CDRomDevice)==OK)
{
        TxString(&SerialDevice,1,(BYTE*)str);    // report drive information
}
```

3.8.4 LockDrive

CDROMSTS LockDrive(const DEVICE* dev)

3.8.4.1 Description

Lock the CD into the drive. This prevents the CD from being ejected and prepares it to be accessed.

3.8.4.2 Parameters

3.8.4.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.4.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.8.4.4 Example

```
if(LockDrive(&CDRomDevice)==OK)
{
        // access CD
}
```

3.8.5 UnlockDrive

CDROMSTS UnlockDrive(const DEVICE* dev)

3.8.5.1 Description

Allow disk to be ejected.

3.8.5.2 Parameters

3.8.5.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.5.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.8.5.4 Example

```
if(UnlockDrive(&CDRomDevice)==OK)
{
        // CD may be ejected by user
}
```

3.8.6 ReadVolumeDesc

CDROMSTS ReadVolumeDesc(const DEVICE* dev)

3.8.6.1 Description

Read the volume descriptor information from the CDRom in the drive into a structure. The volume descriptor contains information such as the physical disk address of the root directory.

3.8.6.2 Parameters

3.8.6.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.6.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.8.6.4 Example

```
char str[128];

if(ReadVolumeDesc(&CDRomDevice)==OK)
{
      GetVolumeDesc(dev, str);          // format info as text
      TxString(&SerialDevice,1, str);   // report volume info
}
```

3.8.7 GetVolumeDesc

void GetVolumeDesc(const DEVICE* dev,char* s)

3.8.7.1 Description

Format the volume descriptor information previously read by ‘ReadVolumeDesc’ as a text string.

3.8.7.2 Parameters

3.8.7.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.7.3 Example

```
char str[128];
if(ReadVolumeDesc(&CDRomDevice)==OK)
{
      GetVolumeDesc(dev, str);          // format info as text
      TxString(&SerialDevice,1, str);   // report volume info
}
```

3.8.8 ReadBlock

CDROMSTS ReadBlock(const DEVICE* dev,LONG sector,LONG length,UWORD* info)

3.8.8.1 Description

Read the given number of sectors of data from the CD starting at the sector defined by 'sector'. The data is copied to the address in system/video memory pointed to by 'info'.

3.8.8.2 Parameters

3.8.8.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.8.2.2 Parameter2 – sector

Start sector on the disk to read from.

3.8.8.2.3 Parameter3 – length

Number of sectors to read (1 sector = 2048 bytes).

3.8.8.2.4 Parameter4 – info

Pointer into system/video memory where the data from the disk is to be copied.

3.8.8.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.8.8.4 Example

```
If(ReadBlock(&CDRomDevice,startsector,1,info)==OK)
{
        //process data from CD
}
```

3.8.9 DriveBusy

UBYTE DriveBusy(const DEVICE* dev)

3.8.9.1 Description

Tests the drive to see if it is ready to accept a command and returns immediately the result without waiting for the usual timeout.

3.8.9.2 Parameters

3.8.9.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.9.3 Return Value

TRUE if the device is busy (i.e. not ready to receive a command).
FALSE if the drive is ready to receive a command.

3.8.9.4 Example

This allows the user to test the drive without locking up the system in the event of the drive being not ready or missing.

```
SetTimer(&timer,100);                   // wait just 1 sec for drive
while(DriveBusy(&CDRomDevice))
        if(timer==0)        return;
```

3.8.10 ReadPktCmd

CDROMSTS ReadPktCmd(const DEVICE* dev,const PKTCMD* packet,UWORD* info)

3.8.10.1 Description

This allows the user access to low level function of the drive as defined in the ATAPI specification.

3.8.10.2 Parameters

3.8.10.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

3.8.10.2.2 Parameter2 – packet

Command packet; please refer to the ATAPI specification for the details of the command packet and IDE registers.

3.8.10.2.3 Parameter3 – info

Address for any data returned by the command to be copied to.

3.8.10.3 Return

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h.

3.9 Touchscreen Driver

3.9.1 Touchscreen Controllers Supported

The Touch Screen Driver supports the following serial interface controllers:

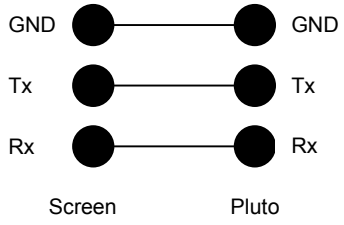
MicroTouch:
- Serial SMT2
- Serial SMT3
- Serial SMT3V
- Serial SMT3R
- Serial SMT3RV

ELO SmartSet Family:
- E271-2200 AccuTouch, DuraTouch
- E211-2210 AccuTouch, DuraTouch
- E281-2310 IntelliTouch
- 2500S IntelliTouch

3.9.2 Connecting Supported Touchscreen to the Pluto 5 Board

MicroTouch serial touchscreens: connect with a standard 9W serial cable.

ELO serial touchscreens: cable must not connect any handshake signals. The only signals connected should be RX, TX and GND:



3.9.3 References

- Ref 1. MicroTouch Touch Controllers Reference Guide 19-213 V2.2
- Ref 2. ELO SmartSet TouchScreen Controller Family: Technical Reference Manual V1.0
- Ref 3. ELO Addendum for E281-2310 Controller
- Ref 4. ELO Addendum for IntelliTouch 2500S Controller

3.9.4 Driver Structure

The Touch Screen Driver forms part of the Peripheral Library and has the following components:

- Generic Touch Screen Driver
- ELO Device Driver
- MicroTouch Device Driver

Either the ELO driver or MicroTouch driver is installed depending on the manufacturer of the touch screen to be used.

The *Generic Touch Screen Driver* provides a **Generic Touch Screen API**.

The functions provided by this API will allow developers to write *device independent touchscreen code* that can be used with all supported models of touch screen controller.

The *Generic Touch Screen Driver* uses the services of either the *ELO Device Driver* or *the MicroTouch Device Driver.*

These drivers in turn make use of the services provided by the various *serial drivers* within the Pluto 5 libraries.

The ELO and MicroTouch Device Drivers also provide **low level API's** which directly map their command sets to API functions. The command sets are:

- ELO Smartset Command Set (ref 2)
- MicroTouch Recommended Development Firmware Commands (ref 1)

It is recommended that developers use the Generic Touch Screen API. The low level API's are provided for developers familiar with the command sets who wish to implement functions not provided by the Generic API.

3.9.5 Driver Installation

The device driver to be installed should be listed in the file *devices.c* in the project directory:

```
const  INSTALLDEVICE DeviceList[ ] =
{
        …
/*      uncomment ONE of the following      */
        {&MicroTouchDevice, 0, 0, 0},
/*      {&EloDevice, 0, 0, 0},              */
        …
        {0, 0, 0}
};
```

The application code should reference the driver loaded. For example, in the file *your_game.c* in the project directory:

```
        const DEVICE* dev;
/*      comment or uncomment ONE of the following   */

        dev = &MicroTouchDevice;
/*      dev = &EloDevice;                   */
        …
/*      touch screen commands       */
        TouchScreenInit(dev);
        TouchScreenSetMode(dev, …);
/*      etc
```

3.9.6 Generic API Functions.

The following is a complete list of the functions provided. They are described in detail in later sections.

The following functions initialise and send commands to the touch screen controller:

- *TouchScreenReset()*
- *TouchScreenInit()*
- *TouchScreenIdentify()*
- *TouchScreenSetSerial()*
- *TouchScreenCommsMode()*
- *TouchScreenSetMode()*
- *TouchScreenCalibrate()*

The following functions return touch data:

- *TouchScreenGetNextData()*
- *TouchScreenGetX()*
- *TouchScreenGetY()*
- *TouchScreenGetZ()*
- *TouchScreenGetTouchStatus()*
- *TouchScreenSetScale*()

3.9.7 Configuration: Serial Port Assignment

The driver can be configured to use any of the serial ports in a Pluto 5/Pluto 5 Casino/Calypso 16 system.

The **assigned serial port** is defined in *config.c* in the project directory. For example:

```
const MICROTOUCHCFG MicroTouchCfg =
{
/*      Serial port, channel assigned to touchscreen      */
        &DUART0Device, 1
};

const ELOCFG EloCfg =
{
/*      Serial port, channel assigned to touchscreen      */
        &DUART0Device, 1
        };
```

It is recommended that the assigned serial port *receive buffer size* should be increased from the default value, because of the high throughput of incoming touch data. The recommended minimum size is 64 bytes.

For example, also in *config.c* in the project directory:

```
/****    Serial Communications - DUART0 configuration and buffers */
…
static BYTE     Duart0RxBBuffer[1<<8]={0};      /* bigger buffer (256 bytes) for touchscreen */
…
```

3.9.8 Driver Operation Overview

Serial communication with the touchscreen controller operates in two possible modes.
The current mode is set by the function *TouchScreenCommsMode()* in the Generic Driver API.

3.9.8.1 Command Mode

This bi-directional mode is used for **initialising** and **sending commands** to the touch screen controller.

ELO and MicroTouch controllers both use a simple <*command*> <*response*> protocol.
These protocols are implemented by the respective device drivers.

The Generic API provides functions for initialising and sending commands to the controllers:

- *TouchScreenReset()*
- *TouchScreenInit()*

- *TouchScreenIdentify()*
- *TouchScreenSetSerial()*
- *TouchScreenCommsMode()*
- *TouchScreenSetMode()*
- *TouchScreenCalibrate()*

Note*:** These functions are ***non-reentrant. They should only be invoked from main loop code, and should not be called from an interrupt service routine. The functions expect a response from the touch screen controller, and terminate with a timeout error if no response is received.

3.9.8.2 Touch Data Mode

In this mode, the assigned serial port **receives a continual stream of touch data** (XY co-ordinates) from the touch screen controller.

The Generic Driver installs a service routine in the *10ms timer interrupt routine.* This routine:

- Reads incoming *touch data packets* (in a number of different formats) from the serial receive buffer.
- Converts the *touch data packets* into a **standard format** *TOUCHDATA record.*
- Stores the *TOUCHDATA records* in the buffer *TouchDataBuffer[].*

The size of the touch data buffer is defined in config.c in the project directory, and must be a power of two. The buffer is circular, and overwrites the oldest record on overflow. A buffer of size *n* records will therefore contain the last *n-1* touch records.

The following declaration should be added to config.c in the project directory. This declares a touch data buffer of 256 entries:

```
(config.c)
/* Touchscreen: Touch data buffer */
TOUCHDATA  TouchDataBuffer[1<<8]={0};
const WORD    TouchDataBufferMask=(1<<8)-1;
```

The **standard format** TOUCHDATA record structure is as follows:

```
typdef struct {
        UWORD x;
        UWORD y;
        UWORD z;
        UBYTE status;
} TOUCHDATA;
```

where:

- x, y are *unscaled* and *uncalibrated* 10-bit touch co-ordinates (0< x <1024 and 0< y <1024)
- z is 8-bit touch pressure data, where supported by the touchscreen controller (0 < z < 256)
- status bit0 = 1 Touch Down Event
- status bit1 = 1 Stream Event (continuous touch)
- status bit2 = 1 Touch Up Event (lift off)

The Generic API provides a number of functions for reading touch data packets from the touch data buffer, and returning *scaled* and *calibrated* touch positions. See the next section for example code.

- *TouchScreenGetNextData()* - Get the next touch data packet from the buffer
- *TouchScreenGetX()* - Return the *scaled* and *calibrated* X position from the touch data packet
- *TouchScreenGetY()* - Return the *scaled* and *calibrated* Y position from the touch data packet
- *TouchScreenGetZ()* – Return the touch pressure from the touch data packet
- *TouchScreenGetTouchStatus()* – Returns the touch status from the touch data packet
- *TouchScreenSetScale*() – Set the scaling and direction

3.9.9 Example User Code Structure

The following is the basic structure of the user code in a touch screen application:

```
dev = &MicroTouchDevice;

/*** initialise touch screen controller ***/
TouchScreenCommsMode(dev, COMMAND_MODE);
TouchScreenInit(dev);
TouchScreenSetMode(dev, DOWN_REPORT|STREAM_REPORT|UP_REPORT);
TouchScreenSetScale(x_resolution, y_resolution, x_invert, y_invert);


/*** GET TOUCH DATA ***/
TouchScreenCommsMode(dev, TOUCH_DATA_MODE);

If  (TouchScreenGetNextData (&packet) != NULL )        /* Touch Data available in buffer */
{
        x       = TouchScreenGetX(&packet);
        y       = TouchScreenGetY(&packet);
        status  = TouchScreenGetTouchStatus(&packet);

        do_something(x, y, status);
}
```

3.9.10 Calibration

All touchscreens require calibrating, in order to ensure correspondence between **touch layer** co-ordinates and **display screen** co-ordinates.

A generic API function *TouchScreenCalibrate()* is provided for this purpose, which is used identically for ELO and MicroTouch screens.

A **two point calibration sequence** is used, which involves the user touching a target near the bottom left of the screen, followed by a target at the top right of the screen. If the display screen has dimensions HSIZE and VSIZE, the targets should be at the following locations:

Target 1 (x, y) = (HSIZE/8 , VSIZE - VSIZE/8)
Target 2 (x, y) = (HSIZE-HSIZE/8 , VSIZE/8)

Although the generic API function can be used identically for ELO and MicroTouch screens, the underlying mechanism is different:

The MicroTouch Driver uses **hardware calibration** within the MicroTouch controller. After calibration, the calibration data is stored in non-volatile memory in the MicroTouch controller, and calibration corrections are applied to touch data *before* it is transmitted to the Pluto 5 system.

The ELO driver uses **software calibration**, with calibration data stored in a structure non-volatile RAM in the **Pluto system**, and calibration corrections applied by the Pluto 5 ELO driver to incoming touch co-ordinate data. If using ELO touchscreens, the following should be declared in *sysram.c* in the project directory:

```
/*  Touchscreen calibration data: NOT cleared on power up */
TOUCHCALIBDATA touchscreen_calib_data;
```

Calibration is only necessary at system installation. Re-calibration should not be necessary unless the display screen size and position are modified.

3.9.10.1 Generic API Example

The following is an **example user function** using *TouchScreenCalibrate()*;

```
LONG screen0, target_sprite;

/* The target sprite is a small bitmapped image of a touch target, for example crosshairs */
extern const WORD      target[];
extern const WORD      target_width;
extern const WORD      target_height;

WORD demo_calibrate(const DEVICE* dev)
{
        WORD x,y;
        /* Middle layer - screen 0 */
        InitDrawframe(&CremsonDevice, 0, screen0, HSIZE);
        LoadSprite(&CremsonDevice, target, target_sprite, target_width*target_height);

        if (TouchScreenCalibrate(dev, 1))
                return(1);

        /*************** STEP 1: Touch Calibration Mark at bottom left of screen ****************/

        x = (HSIZE/8);
        y = VSIZE-(VSIZE/8);
        ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], screen0, HSIZE);
        PutSprite(&CremsonDevice, screen0, x - target_width/2, y - target_height/2, target_sprite,
        target_width, target_height, 0);

        if (TouchScreenCalibrate(dev, 2))
                return(1);

        /*************** STEP 2: Touch Calibration Mark at top right of screen ****************/

        x = HSIZE-(HSIZE/8);
        y = (VSIZE/8);
        ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], screen0, HSIZE);
        PutSprite(&CremsonDevice, screen0, x - target_width/2, y - target_height/2, target_sprite,
        target_width, target_height, 0);

        if (TouchScreenCalibrate(dev, 3))
         {
                ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], screen0, HSIZE);
                return(1);
        }

        ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], screen0, HSIZE);
        return(0);
}
```

3.9.11 TouchScreenInit

UWORD TouchScreenInit(const DEVICE *dev)

3.9.11.1 Description

Initialise the Touch Screen Controller.

3.9.11.2 Parameters

3.9.11.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers

MicroTouchDevice MicroTouch Touch Screen Controllers

3.9.11.3 Return Value

None.

3.9.11.4 Example

```
// dev = &EloDevice;
dev = &MicroTouchDevice;
TouchScreenInit(dev);
```

3.9.12 TouchScreenReset

UWORD TouchScreenReset(const DEVICE *dev, UBYTE reset_type)

3.9.12.1 Description

Reset the Touch Screen Controller.

3.9.12.2 Parameters

3.9.12.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers
MicroTouchDevice	MicroTouch Touch Screen Controllers

3.9.12.2.2 Parameter 2 - reset_type

(EL O only. MicroTouch: don't care)

HARD_RESET	(=0)	Full Hardware Reset
SOFT_RESET	(=1)	Software Reset

3.9.12.3 Return Value

None.

3.9.12.4 Example

```
// dev = &EloDevice;
dev = &MicroTouchDevice;
TouchScreenReset(dev, HARD_RESET);
```

3.9.13 TouchScreenIdentify

UWORD TouchScreenIdentify(const DEVICE *dev, TOUCHIDENTITY *touchidentity)

3.9.13.1 Description

Identify the Touch Screen Controller Hardware.

3.9.13.2 Parameters

3.9.13.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers
MicroTouchDevice	MicroTouch Touch Screen Controllers

3.9.13.2.2 Parameter 2 – touchidentity

The following structure:

```
typedef struct touchidentity {
        BYTE model[32];
        BYTE revision[32];
        BYTE features[32];
        BYTE status[32];
```

} TOUCHIDENTITY;

3.9.13.3 Return Value

zero	OK
non-zero	comms error code

comms error code - ELO:
0 ELO_COMMS_OK,
1 ELO_COMMS_FAIL,
2 ELO_COMMS_TIMEOUT,
3 ELO_COMMS_CHECKSUM_ERROR,
4 ELO_PROTOCOL_ERROR,
5 ELO_ACKNOWLEDGE_ERROR

comms error code - MicroTouch:
0 MICROTOUCH_COMMS_OK,
1 MICROTOUCH_COMMS_FAIL,
2 MICROTOUCH_COMMS_TIMEOUT,
3 MICROTOUCH_COMMS_INVALID_RESPONSE

3.9.13.4 Example

```
TOUCHIDENTITY touchidentity;
If( !TouchScreenIdentify(dev, &touchidentity))
{
sprintf(str, "%s\r\n", touchidentity.model);
TxString(&SerialDevice,1,str);
sprintf(str, "%s\r\n", touchidentity.revision);
TxString(&SerialDevice,1,str);
sprintf(str, "%s\r\n", touchidentity.features);
TxString(&SerialDevice,1,str);
sprintf(str, "%s\r\n", touchidentity.status);
TxString(&SerialDevice,1,str);
}
```

3.9.14 TouchScreenSetSerial

UWORD TouchScreenSetSerial(const DEVICE *dev, BYTE baud, BYTE bits, BYTE stop, BYTE parity, BYTE handshake)

3.9.14.1 Description

Set the Touch Screen Controller serial parameters.

WARNING: It is strongly recommended that the following ELO and MicroTouch default settings be used. These are also the default driver initialisation settings, so normally there is no need to call this function. Default settings (ELO and MicroTouch): 9600 baud, 8 data bits, 2 stop bit, no parity, no handshake.

3.9.14.2 Parameters

3.9.14.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers
MicroTouchDevice	MicroTouch Touch Screen Controllers

3.9.14.2.2 Parameter 2 – baud

Touch Screen Controller Baud Rate. The following are valid:

BAUD1200	(MicroTouch only)
BAUD2400	(MicroTouch only)
BAUD4800	(MicroTouch only)
BAUD9600	(Microtouch, ELO)
BAUD19200	(Microtouch, ELO)

3.9.14.2.3 Parameter 3 – bits

Touch Screen Controller data bits. The following are valid:
BITS7
BITS8

3.9.14.2.4 Parameter 4 – stop

Touch Screen Controller stop bits. The following are valid:
STOP1
STOP2

3.9.14.2.5 Parameter 5 – parity

Touch Screen Controller parity bits. The following are valid:
PARITYEVEN
PARITYODD
PARITYNONE

3.9.14.2.6 Parameter 6 – hardware handshake (ELO only. MicroTouch not used)

Touch Screen Handshake settings. The following is valid
No handshake 0

3.9.14.3 Return Value

zero OK
non-zero comms error code – see function TouchScreenIdentify()

3.9.14.4 Example

```
TouchScreenSetSerial(dev, BAUD9600, BITS8, STOP2, PARITYNONE, 0);          //touchscreen
ConfigureSerial(&DUART0Device,0,BAUD9600,BAUD9600,BITS8,STOP2,PARITYNONE,TXNONE,
RXNONE); //UART
```

3.9.15 TouchScreenSetMode

UWORD TouchScreenSetMode(const DEVICE *dev, UBYTE mode)

3.9.15.1 Description

Set the reporting mode for the Touch Screen Controller.

3.9.15.2 Parameters

3.9.15.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:
EloDevice ELO Touch Screen Controllers
MicroTouchDevice MicroTouch Touch Screen Controllers

3.9.15.2.2 Parameter 2 – mode

A logical OR combination of the following bits fields:
DOWN_REPORT Enable reporting of screen touch down events
STREAM_REPORT Enable reporting of continuous stream event data
UP_REPORT Enable reporting of stream lift off events

3.9.15.3 Return Value

zero OK
non-zero comms error code – see function TouchScreenIdentify()

3.9.15.4 Example

```
TouchScreenSetMode(dev, DOWN_REPORT|STREAM_REPORT|UP_REPORT);
```

3.9.16 TouchScreenSetScale

void TouchScreenSetScale(WORD xmax, WORD ymax, BYTE xinvert, BYTE yinvert)

3.9.16.1 Description

The default settings of the touchscreen are 1024*1024. A call to this function is mandatory in order to reflect the real resolution of the screen.

3.9.16.2 Parameters

3.9.16.2.1 Parameter 1 – xmax

Touch Screen X Resolution (0 < xmax <= 1024).

3.9.16.2.2 Parameter 2 – ymax

Touch Screen Y Resolution (0 < ymax <= 1024).

3.9.16.2.3 Parameter 3 – xinvert

0	Screen Left=0,	Screen Right= X max
1	Screen Left=Xmax,	Screen Right= 0

3.9.16.2.4 Parameter 4 - yinvert

0	Screen Top=0,	Screen Bottom= Y max
1	Screen Top=Ymax,	Screen Bottom= 0

3.9.16.3 Example

```
/* bottom left 0,0 top right 799,599 */
UWORD x_resolution=800;
UWORD y_resolution=600;
BYTE x_invert=0;
BYTE y_invert=1;
TouchScreenSetScale(x_resolution, y_resolution, x_invert, y_invert);
```

3.9.17 TouchScreenCalibrate

UWORD TouchScreenCalibrate(const DEVICE *dev, UWORD step_number)

3.9.17.1 Description

Calibrate the Touch Screen Controller.

3.9.17.2 Parameters

3.9.17.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers
MicroTouchDevice	MicroTouch Touch Screen Controllers

3.9.17.2.2 Parameter 2 - step_number

1-3 incrementing on successive calls to this function (multi-step calibration).

3.9.17.3 Return Value

zero	OK
non-zero	comms error code – see function TouchScreenIdentify()

3.9.17.4 Example

See section “Calibration”

3.9.18 TouchScreenCommsMode

void TouchScreenCommsMode(const DEVICE *dev, BYTE mode)

3.9.18.1 Description

Set the communication mode with the Touch Screen Controller:
COMMAND_MODE: Bi-directional, with ELO or MicroTouch command/response protocol, polled, in-line code. (blocking – not interrupt driven)
TOUCH_DATA_MODE: Uni-directional streaming touch data from controller to Pluto 5. Interrupt driven, with touch data packet buffer controlled by driver

3.9.18.2 Parameters

3.9.18.2.1 Parameter 1 – dev

Touch Screen Device. One of the following:

EloDevice	ELO Touch Screen Controllers
MicroTouchDevice	MicroTouch Touch Screen Controllers

3.9.18.2.2 Parameter 2 – mode

One of the following:
COMMAND_MODE
TOUCH_DATA_MODE

3.9.18.3 Return Value

None.

3.9.18.4 Example

See section 3.8.9 Example User Code Structure.

3.9.19 TouchScreenGetNextData

TOUCHDATA *TouchScreenGetNextData(TOUCHDATA *touch_packet)

3.9.19.1 Description

Get the next Touch Data Packet from the Touch Event Buffer.

3.9.19.2 Parameters

3.9.19.2.1 Parameter 1 - touch_packet

Touch Data Packets have the following structure:

```
typedef struct touchdata {
        UWORD x;
        UWORD y;
        UWORD z;
        UBYTE status;
} TOUCHDATA;
```

The x, y co-ordinate values are ***unscaled*** 10-bit values in the range 0-1023.

3.9.19.3 Return Value

Touch Data Packet or NULL if buffer empty.

3.9.19.4 Example

```
If (TouchScreenGetNextData (&packet) != NULL )
{
        x       = TouchScreenGetX(&packet);
        y       = TouchScreenGetY(&packet);
        status  = TouchScreenGetTouchStatus(&packet);
}
```

3.9.20 TouchScreenGetX

UWORD TouchScreenGetX(TOUCHDATA *touch_packet)

3.9.20.1 Description

Returns the scaled and calibrated screen touch X position from a Touch Data Packet
The scaling applied must be initially specified using function TouchScreenSetScale().

3.9.20.2 Parameters

3.9.20.2.1 Parameter 1 - touch_packet

3.9.20.3 Return Value

X position.

3.9.21 TouchScreenGetY

UWORD TouchScreenGetY(TOUCHDATA *touch_packet)

3.9.21.1 Description

Returns the scaled and calibrated screen touch Y position from a Touch Data Packet
The scaling applied must be initially specified using function TouchScreenSetScale().

3.9.21.2 Parameters

3.9.21.2.1 Parameter 1 - touch_packet

3.9.21.3 Return Value

Y position.

3.9.22 TouchScreenGetZ

UWORD TouchScreenGetZ(TOUCHDATA *touch_packet)

3.9.22.1 Description

Returns the screen touch Z value (touch pressure) from a Touch Data Packet (If supported by the Touch Screen hardware).

3.9.22.2 Parameters

3.9.22.2.1 Parameter 1 - touch_packet

3.9.22.3 Return Value

Z value (0-255) - 0 if unsupported.

3.9.23 TouchScreenGetTouchStatus

UBYTE TouchScreenGetTouchStatus(TOUCHDATA *touch_packet)

3.9.23.1 Description

Return the screen touch status from a Touch Data Packet.

3.9.23.2 Parameters

3.9.23.2.1 Parameter 1 - touch_packet

3.9.23.3 Return Value

Bit0=1	DOWN_EVENT	Touch down
Bit1=1	STREAM_EVENT	Continuous touch
Bit2=1	UP_EVENT	Lift up

4 INTERFACE LAYER

4.1 Sprite Engine

The sprite engine provides a convenient way of loading, displaying and moving images and animations around the screen. Think of a sprite as a container, which can hold an image (or a series of images to make up an animation sequence).

4.1.1 CreateSprite

LONG CreateSprite(const DEVICE* dev,SPRITE* sp,const WORD* image, LONG base, LONG mode, BYTE frames)

4.1.1.1 Description

Create a sprite container and associate it with an area of video memory. The video memory is used to hold background and image data.

4.1.1.2 Parameters

4.1.1.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.1.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory. This holds all the variables associated with the sprite.

4.1.1.2.3 Parameter3 – image

Start address of the images data in system memory. This data is loaded into video memory at the address defines by ‘base’.

4.1.1.2.4 Parameter4 – base

Offset into video memory where the sprite image data is to be loaded.

4.1.1.2.5 Parameter5 – mode

Copy mode used to copy the sprite from off screen memory to on screen memory. Use 0 for normal mode or TRANSPARENT_MODE. Devices, which do not support hardware accelerated transparent bit blatting, emulate this in software.

4.1.1.2.6 Parameter6 – frames

Number of animation frames to load. Images should be contiguous in system memory. For a still image load a single frame.

4.1.1.3 Return Value

Next free area of video memory after that used by this sprite.

4.1.1.4 Example

Load a twelve frame animation sequence into the sprite container

```
LONG spritebas;
SPRITE sprite1;

spritebase= HSIZE*VSIZE*2;     //load sprite above screen memory
spritebase= CreateSprite(&CremsonDevice,&sprite1,imagedata,spritebase,TRANSPARENT_MODE,12);
```

4.1.2 CloneSprite

LONG CloneSprite(const DEVICE* dev,SPRITE* mother,SPRITE* child, LONG base)

4.1.2.1 Description

Make a copy of the 'mother' sprite. The child sprite uses the same image data in video memory but has its own video memory to perform background save/restore sequences.

4.1.2.2 Parameters

4.1.2.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.2.2.2 Parameter2 – mother

Pointer to the SPRITE structure in system memory which is to be copied.

4.1.2.2.3 Parameter3 – child

Pointer to the SPRITE structure in system memory which is to become the copy.

4.1.2.2.4 Parameter4 – base

Offset into system memory, which defines the address where the child sprite allocates video memory for background save / restore.

4.1.2.3 Return Value

Next free area of video memory after that used by this sprite.

4.1.2.4 Example

Make a copy of the sprite created in the last example.

```
LONG spritebas;
SPRITE sprite1, sprite2;

spritebase= HSIZE*VSIZE *2;    //load sprite above screen memory
spritebase=CreateSprite(&CremsonDevice,&sprite1,imagedata,spritebase,TRANSPARENT_MODE,12);
spritebase=CloneSprite(&CremsonDevice,&sprite1,&sprite2,spritebase);
```

4.1.3 StartSprite

Void StartSprite(const DEVICE* dev,SPRITE* sp,LONG screen0, LONG screen1,WORD x,WORD y, XY (*upd)(SPRITE* sp,XY pos),void* motionvars, LONG startframe,LONG incr, UBYTE ratediv)

4.1.3.1 Description

Prepare the sprite for animation and/or motion around the screen. The sprite functions require two copies of screen memory, which are alternately displayed at a fixed rate (double buffering). This function requires that the two areas of screen memory be already allocated and any background images loaded.

4.1.3.2 Parameters

4.1.3.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.3.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory which is to be prepared.

4.1.3.2.3 Parameter3 – screen0

Offset into video memory of screen zero.

4.1.3.2.4 Parameter4 – screen0

Offset into video memory of screen one.

4.1.3.2.5 Parameter5 – x

The x co-ordinate of the sprite start position.

4.1.3.2.6 Parameter6 – y

The y co-ordinate of the sprite start position.

4.1.3.2.7 Parameter7 – upd

Pointer to a function which updates the sprite co-ordinates each time UpdateXY() is called. This function defines the motion of the sprite. Use 0 for a static sprite.

4.1.3.2.8 Parameter8 – motionvars

Address of a structure containing all the variables required by the update function.

4.1.3.2.9 Parameter9 – startframe

First frame of the animation sequence to be displayed.

4.1.3.2.10 Parameter10 – incr

Animation index increment. An increment of 1 shows all the animation frames in ascending order. An increment of -1 shows all the animation frames in descending order. An increment of 2 shows every other animation frame in ascending order etc.

4.1.3.2.11 Parameter11 – ratediv

Frame rate divisor. A value of 1 increments/decrements the animation index every time the sprite is re-drawn (The actual rate is set by the double buffering frame rate). A value of 2 increments / decrements the animation index every second time the sprite is re-drawn (half the double buffering frame rate) etc.

4.1.3.3 Return Value

None.

4.1.3.4 Example

Start an animated sprite from position x=500 y=150. The animation starts at frame 0 and cycles through the frames in ascending order each time the sprite is re-drawn.

```
LONG scr0;
LONG scr1;

scr0=0;
scr1= HSIZE*VSIZE;
StartSprite(dev,&sprite1,scr0,scr1,500,150,CVEdgeBounce,&cv,0,1,1);
```

4.1.4 UpdateSpriteXY

void UpdateSpriteXY(const DEVICE* dev,SPRITE* sp,BYTE scrn)

4.1.4.1 Description

Update the sprite co-ordinates for the given screen. A set of co-ordinates for the position of the sprite in both screens is maintained.

4.1.4.2 Parameters

4.1.4.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.4.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory.

4.1.4.2.3 Parameter3 – scrn

Screen index that defines which set of sprite co-ordinates is to be updated. (Either zero or one).

4.1.4.3 Return Value

None.

4.1.4.4 Example

```
//Update the screen 0 position of the sprite
UpdateSpriteXY(&CremsonDevice,&sprite,0);
```

4.1.5 SaveBack

void SaveBack(const DEVICE* dev,SPRITE* sp,LONG scrn)

4.1.5.1 Description

Save the background area occupied by the sprite on the given screen. This is done before drawing the sprite.

4.1.5.2 Parameters

4.1.5.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.5.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory.

4.1.5.2.3 Parameter3 – scrn

Screen index that defines which screen is to be operated on. (Either zero or one).

4.1.5.3 Return Value

None.

4.1.5.4 Example

Save the screen 0 background associated with the sprite.

```
SaveBack(&CremsonDevice,&sprite,0);
```

4.1.6 RestoreBack

void RestoreBack(const DEVICE* dev,SPRITE* sp,LONG scrn)

4.1.6.1 Description

Restore the previously saved background area occupied by the sprite on the given screen.

4.1.6.2 Parameters

4.1.6.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.6.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory.

4.1.6.2.3 Parameter3 – scrn

Screen index that defines which screen is to be operated on. (Either zero or one).

4.1.6.3 Return Value

None.

4.1.6.4 Example

```
//Restore the screen 0 background associated with the sprite.
RestoreBack(&CremsonDevice,&sprite,0);
```

4.1.7 DrawSprite

void DrawSprite(const DEVICE* dev,SPRITE* sp,BYTE scrn)

4.1.7.1 Description

Draw the sprite at its current position on the given screen.

4.1.7.2 Parameters

4.1.7.2.1 Parameter1 – dev

Pointer to the device structure.

4.1.7.2.2 Parameter2 – sp

Pointer to the SPRITE structure in system memory.

4.1.7.2.3 Parameter3 – scrn

Screen index that defines which screen is to be operated on. (Either zero or one).

4.1.7.3 Return Value

None.

4.1.7.4 Example

DrawSprite(&CremsonDevice,&sprite,0);

4.1.8 Example of double buffering with sprite engine.

A number of sprite engine functions have been modified to correctly support double buffering for both **middle and base layers** (release of interface library 5).
The functions corrected are: *StartSprite(), SaveBack(), RestoreBack(), and DrawSprite()*

```
LONG screen1, screen2, screen3, screen4, spritebuffer;

void            VDemo()
{
// Video memory addresses
screen1         = 0;                    // base layer frame 0
screen2         = HSIZE*VSIZE;          // base layer frame 1
screen3         = 2*HSIZE*VSIZE;        // middle layer frame 0
screen4         = 3*HSIZE*VSIZE;        // middle layer frame 1
spritebuffer    = 4*HSIZE*VSIZE;

LoadPalette(&CremsonDevice,stdpalette_table);

// initialise base layer
InitLayer(&CremsonDevice, &CremsonLayer[BL_LAYER], INDIRECT_COLOR, LAYER_FRAME0, BLACK,
HSIZE, VSIZE, screen1, screen1, screen2, screen2);

// initialise middle layer
InitLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], INDIRECT_COLOR, LAYER_FRAME0, BLACK,
HSIZE, VSIZE, screen3, screen3, screen4, screen4);

ClearLayer(&CremsonDevice, &CremsonLayer[BL_LAYER], LAYER_FRAME0, HSIZE);
ClearLayer(&CremsonDevice, &CremsonLayer[BL_LAYER], LAYER_FRAME1, HSIZE);
ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], LAYER_FRAME0, HSIZE);
ClearLayer(&CremsonDevice, &CremsonLayer[ML_LAYER], LAYER_FRAME1, HSIZE);
EnableLayer(&CremsonDevice, BASE_ENABLE);
EnableLayer(&CremsonDevice, MIDDLE_ENABLE);

SpriteDemo();
}
```

```
void    SpriteDemo()
{
    BYTE screen;
    static SPRITE sprites[2];       // Array of sprites
    static XY cv[2] = {             // Sprite motion X & Y increments
        {20,20},
        {15,15}
    };

    // sprite 1 animation frames: cherry1, cherry 2
    spritebuffer=CreateSprite(&CremsonDevice,&sprites[0],cherry1,spritebuffer,TRANSPARENT,2);
    // sprite 2 animation frames: cherry3, cherry 4
    spritebuffer=CreateSprite(&CremsonDevice,&sprites[1],cherry3,spritebuffer,TRANSPARENT,2);

    // sprite 1 base layer:   screen 1, screen 2
    StartSprite(&CremsonDevice,&sprites[0],screen1,screen2,  0,  0,CVEdgeBounce,&cv[0],0,1,1);
    // sprite 2 middle layer: screen 3, screen 4
    StartSprite(&CremsonDevice,&sprites[1],screen3,screen4,320,240,CVEdgeBounce,&cv[1],0,1,1);

    // set automatic double buffering running.
    // Toggle base and middle layers between frame 0 and frame 1....
    SetDoubleBuffer(&CremsonDevice,0,0,FRAME5HZ);

    // Sprite animation loop:
    screen=0;
    while(1)
    {
        while(!ScreenFree(&CremsonDevice,screen));

        UpdateSpriteXY(&CremsonDevice,&sprites[0],screen);
        UpdateSpriteXY(&CremsonDevice,&sprites[1],screen);

        RestoreBack(&CremsonDevice,&sprites[0],screen);
        RestoreBack(&CremsonDevice,&sprites[1],screen);

        SaveBack(&CremsonDevice,&sprites[0],screen);
        SaveBack(&CremsonDevice,&sprites[1],screen);

        DrawSprite(&CremsonDevice,&sprites[0],screen);
        DrawSprite(&CremsonDevice,&sprites[1],screen);

        screen=!screen;
    }
}

static XY CVEdgeBounce(SPRITE *sp,XY p1)
{
XY      p2;
XY*     p;

p=sp->motionvars;       // XY type holds x & y increments
if((p1.x+p->x>(HSIZE-sp->width)) || (p1.x+p->x<0))
        p->x=-p->x;

if((p1.y+p->y>(VSIZE-sp->height)) || (p1.y+p->y<0))
        p->y=-p->y;

p2.x=p1.x+p->x;
p2.y=p1.y+p->y;
return(p2);
}
```

NOTES:
The following lines from the above example define a screen layer, and assign a sprite to that layer.

```
// memory allocated to layer frames :
screen1     = 0;                // base layer frame 0
screen2     = HSIZE*VSIZE;      // base layer frame 1
…
// initialise base layer:
InitLayer(&CremsonDevice, &CremsonLayer[BL_LAYER], INDIRECT_COLOR, LAYER_FRAME0,
        BLACK, HSIZE, VSIZE, screen1, screen1, screen2, screen2);
…
// assign sprite[0] to base layer
StartSprite(&CremsonDevice, &sprites[0], screen1, screen2,  0,  0, CVEdgeBounce, &cv[0], 0, 1 ,1);
```

4.2 ISO9660

This module contains high level functions to address a CD-ROM whose format complies with ISO9660.

4.2.1 MountISO

UBYTE MountISO(const DEVICE* dev)

4.2.1.1 Description

Initialise the file system on the given drive. The current directory will be set to be the root directory and the file table for the current directory will be read into memory.

4.2.1.2 Parameters

4.2.1.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

4.2.1.3 Return Value

TRUE on success or FALSE in the event of an error.

4.2.1.4 Example

```
if(MountISO(&CDRomDevice))
{
        // OK to access file system
}
```

4.2.2 GetDirRecord

void GetDirRecord(const DIRRECORD* drp,char* s)

4.2.2.1 Description

Build a text string containing the information in the given directory record. The directory records for the files in the current directory may be obtained by reading the disk starting at the sector obtained from the path record pointed to by the global variable ‘currentdir’
e.g currentdir->dirsector

4.2.2.2 Parameters

4.2.2.2.1 Parameter1 – drp

Pointer to a directory record read from the disk.

4.2.2.2.2 Parameter2 – s

Pointer to the memory to write the string to.

4.2.2.3 Return Value

None.

4.2.2.4 Example

The following example shows how to obtain and print out the directory records for the current directory. For simplicity this code fragment assumes that the directory records for all files/directories in the current directory are contained with one disk sector.

```
sts=ReadBlock(&CDRomDevice,currentdir->dirsector,1, buffer);
if(sts!=OK) return(sts);                    // return error condition
drp=(const DIRRECORD*)buffer;               // pointer to directory record of first file

//      Print out the directory records
while(drp)
{
        GetDirRecord(drp,str);      // get directory record string
        TxString(dev,channel,str);// print it out
        drp=NextDRP(drp);           // point at next directory record
        if(drp==0) break;           // no more records to print
}
```

4.2.3 NextDRP

const DIRRECORD* NextDRP(const DIRRECORD* drp)

4.2.3.1 Description

Calculate the address of the next directory record. Having read a sector full of directory records for the current directory this function may be used to search for a particular directory record.

4.2.3.2 Parameters

4.2.3.2.1 Parameter1 – drp

Pointer to the current directory record.

4.2.3.3 Return Value

Pointer to the next directory record.

4.2.3.4 Example

See previous example.

4.2.4 ChangeDirectory

CDROMSTS ChangeDirectory(const DEVICE* dev,const char* path)

4.2.4.1 Description

Change the default directoy to the absolute directory specified by ‘path’.
When then default directory changes the file table is updated to reflect the change. The file table is an array of structures of type FILE containing information about the files in the current directory. This is used to locate the files on the disk.

4.2.4.2 Parameters

4.2.4.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

4.2.4.2.2 Parameter2 – path

Pointer to a text string defining the absolute path of the directory.

4.2.4.3 Return Value

Return the status reported by the drive. (OK if the command was successful) For details of the CDROMSTS type refer to cdrom.h

4.2.4.4 Example

```
If(ChangeDirectory(&CDRomDevice,"\\STREAM")==OK)
{
        // Current directory is \stream
}
```

4.2.5 LoadFile

UWORD* LoadFile(const DEVICE* dev,const char* filename,UWORD* info)

4.2.5.1 Description

Load the given file form the current directory to the address in system/video memory defined by 'info'.

4.2.5.2 Parameters

4.2.5.2.1 Parameter1 – dev

Pointer to the CDRom device driver structure.

4.2.5.2.2 Parameter2 – filename

Pointer to a text string specifying the file to load.

4.2.5.2.3 Parameter3 – info

Pointer into system/video memory where the file is to be loaded.

4.2.5.3 Return Value

Next free address in system/video memory after the file has been loaded. (Or zero in the event of an error).

4.2.5.4 Example

```
if(LoadFile(&CDRomDevice,"image1.raw",filedata))        // Load image off screen
{
        // put image on screen (offset 0 from base) at position x=150,y=100
        PutImage(&CremsonDevice,0,150,100,filedata);
        while(BltBusy(&CremsonDevice));
}
```

4.3 FAT32 Driver

This driver allows *read-only* access to a **FAT32 Format Filesystem** on an external ATA device.
External ATA devices can be connected to the IDE connector on the Calypso 16 Video Board.

Examples are:
- IDE Hard Drive.
- Compact Flash Card fitted to the Heber Compact Flash Adaptor Board.

If the drive is partioned, the FAT32 filesystem to be accessed must be on the **Primary Partition.**

Functions are provided to find and mount the filesystem; navigate the directory structure, list directory contents; and load a file from the drive to memory.

***Note*:** This driver uses the services of the **ATA Device Driver.** This driver must be loaded, and the drive initialised before the services of the FAT32 driver can be used. See section: *ATA Driver.*

The following functions use **file descriptors:**
- **FileOpen()**
- **FileGetChar()**
- **FileGetSectors()**

- **FileLoad()**
- **FileRewind()**
- **FileClose()**
- **GetFileSize()**

Files to be accessed are opened with function *FileOpen()* This function initialises a *file descriptor structure* containing information about the file.

Subsequent functions use the file descriptor information to quickly load data from the file on demand.

File Descriptors are statically declared, and are NOT dynamically allocated.
Each file descriptor requires 1810 bytes of RAM.
Each open file has a file descriptor. The maximum number of simultaneously open files is equal to the number of file descriptors declared, and will be limited by the variable RAM space available.
The application code should declare a fixed number of file descriptors.

```
/* config.c */
#define MAX_OPEN_FILES      8
FILE_DESCRIPTOR file[MAX_OPEN_FILES];
```

The maximum file size supported by the driver depends on the degree of fragmentation of the file.

For a completely unfragmented file, the maximum file size supported is 64 Mbytes.
This file size is reduced for fragmented files. It is recommended that CF cards that have been frequently modified should be defragmented, or re-formatted and re-created.

4.3.1 MountFAT32

UBYTE MountFAT32(const DEVICE *dev)

4.3.1.1 Description

Look for FAT32 filesystem on primary partition of drive. Mount the filesystem and initialise the *filesystem* structure.

4.3.1.2 Parameters

4.3.1.2.1 Parameter1 –dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.1.3 Return Value

Returns zero on success. Returns non-zero on failure.

4.3.1.4 Example

```
/* ATA Initilisation Code: See ATA Driver Section. */
…

/********** Mount FAT32 Filesystem *************/
TxString(&SerialDevice,1,"\r\nMounting Drive...\r\n");
DriveSpinup(&ATADevice);        /* needed for hard drives only */

if (MountFAT32(&ATADevice) != 0)
{
        TxString(&SerialDevice,1,"Failed to mount drive\r\n");
        while(1);
}
else
{
        TxString(&SerialDevice,1,"\r\nFAT32 Partition found on Primary Partition\r\n");
sprintf(str, "\r\nVolume: %s Size: %u Mbytes.\r\n\r\n", FAT32VolumeLabel(&ATADevice),
FAT32PartitionSize(&ATADevice));
        TxString(&SerialDevice,1,str);
```

}

4.3.2 ChangeDir

UBYTE ChangeDir(const DEVICE *dev, BYTE *path)

4.3.2.1 Description

Change current directory to new directory. New directory is root, parent or single level sub directory of current directory.

4.3.2.2 Parameters

4.3.2.2.1 Parameter1 – dev

Pointer to the ATA device This will be **&ATADevice.**

4.3.2.2.2 Parameter2 – path

New directory name string.

4.3.2.3 Return Value

Returns zero if successful. Returns non-zero if new directory not found.

4.3.2.4 Examples

```
ChangeDir(&ATADevice,  "\\");          /* Change to root directory */
ChangeDir(&ATADevice, "..");           /* Change to parent directory */
ChangeDir(&ATADevice, ".");            /* Change to current directory: no change! */
ChangeDir(&ATADevice, "images");       /* Change to sub-directory "images" */
```

4.3.3 NextDirRecord

UBYTE NextDirRecord(const DEVICE* dev, BYTE *path, DIR_ENTRY *dir_entry);

4.3.3.1 Description

This function reads a single record from the current directory.

Structure *dir_entry* is initialised with the record data, and can be interrogated by the functions: *DirEntryLongfilename, DirEntryAlias, DirEntryAttrib, DirEntryFilesize*

The function will increment the *directory record pointer* to next record in the current directory, until the end of the current directory is reached. Thus, the entire current directory can be listed by repeatedly calling this function.

The directory record pointer is initialised to the first record in the directory by function ChangeDir()

4.3.3.2 Parameters

4.3.3.2.1 Parameter1 – dev

Pointer to the ATA device This will be **&ATADevice.**

4.3.3.2.2 Parameter2 – dir_entry

A structure holding directory record data: filename, filesize, attribute bits.

4.3.3.3 Return Value

Returns true if further records in directory. Returns false if no more directory records: (at end of directory).

4.3.3.4 Example

See example in following section.

4.3.4 DirEntryLongfilename, DirEntryAlias, DirEntryAttrib, DirEntryFilesize

```
UBYTE       *DirEntryLongfilename(const DEVICE *dev, DIR_ENTRY *dir_entry)
UBYTE       *DirEntryAlias(const DEVICE *dev, DIR_ENTRY *dir_entry)
UBYTE       *DirEntryAttrib(const DEVICE *dev, DIR_ENTRY *dir_entry)
ULONG       DirEntryFilesize(const DEVICE *dev, DIR_ENTRY *dir_entry);
```

4.3.4.1 Description

These functions return the following information from the *dir_entry* structure:

- long filename string
- short 8.3 format filename.string
- Directory entry attribute bit string
- filesize in bytes.

4.3.4.2 Return Value

See above.

4.3.4.3 Example

```
/* List the current directory to the serial port */
BYTE str[256];
DIR_ENTRY entry;
/* initialise current directory structure and directory record pointer */
ChangeDir(&ATADevice, ".");

// get directory record and print it out
while (NextDirRecord(&ATADevice, &entry)) {
sprintf(str, "%8s %-12s %10d bytes %s\r\n", DirEntryAttrib(&ATADevice, &entry),
      DirEntryAlias(&ATADevice, &entry), DirEntryFilesize(&ATADevice, &entry),
      DirEntryLongfilename(&ATADevice, &entry)
);
TxString(&SerialDevice,1,str);
```

Example output:

```
           AUTOEXEC.BAT    1103   bytes
           CONFIG.SYS       811   bytes
           MODE.COM       29271   bytes
      sh   KEYBOARD.SYS   34566   bytes
      sh   KEYB.COM       19927   bytes
<dir>      IMAGES              0   bytes
     r     BOOTEX.LOG      1478   bytes
           SETUP.BMP     921656   bytes
           PSEUDO~1.TXT    2341   bytes   pseudocode.txt
```

4.3.5 GetDir

```
UBYTE *GetDir(const DEVICE *dev)
```

4.3.5.1 Description

Return current directory path string.

4.3.5.2 Parameters

4.3.5.2.1 Parameter1 – dev

Pointer to the ATA device This will be **&ATADevice.**

4.3.5.3 Return Value

Return current directory path string. This is full absolute directory path.
Examples:
“\”
“\images”

"\images\playing_cards"

4.3.5.4 Example

```
sprintf(str, "%s", GetDir(&ATADevice));
TxString(&SerialDevice,1,str);
```

4.3.6 FAT32LoadFile

UBYTE FAT32LoadFile(const DEVICE *dev, BYTE *path, UWORD *mem)

4.3.6.1 Description

Binary transfer from file *path* to memory at *mem* in System or Video RAM. The file to be transferred **must** be in the current directory.
Obsolete function, it is faster to use FileLoad describe further in this part

***Note*:** An integer number of sectors are transferred. Each sector contains 512 bytes. The total number of bytes transferred may therefore exceed the number of bytes in the file by up to 512 bytes.

4.3.6.2 Parameters

4.3.6.2.1 Parameter1 – dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.6.2.2 Parameter2 – path

Filename string of file to be transferred. This must be in the current directory.
Filename can be: **FAT32 long format**, or **short (8.3) format**.

Example: "acespades playing card" ***or*** "acespa~1"

4.3.6.2.3 Parameter3 – mem

Pointer to memory address to which the data is transferred. This can be in System RAM or Video RAM.

4.3.6.3 Return Value

Returns zero if successful. Returns non-zero if file not found.

4.3.6.4 Example

```
/* Navigate filesystem, load image file and display
Image to be loaded is: \images\playing_cards\acespa~1.bin */

/* change directory */
ChangeDir(&ATADevice, "\\");
if (ChangeDir(&ATADevice, "images") != 0)
        TxString(&SerialDevice,1,"\r\nDirectory not found.\r\n");
if (ChangeDir(&ATADevice, "playing_cards") != 0)
        TxString(&SerialDevice,1,"\r\nDirectory not found.\r\n");

/* load file to undisplayed video memory and blit to screen */
if ( !FAT32LoadFile(&ATADevice, "acespa~1.bin", (UWORD*) (VideoMemBase+offscreen) ) )
        PutSprite(&CremsonDevice, screen0, 0, 0, offscreen + 800,  card_xsize, card_ysize, 0);
else
        TxString(&SerialDevice,1,"\r\nFile not found.\r\n");;
```

4.3.7 FileOpen

FILE_DESCRIPTOR *FileOpen(const DEVICE *dev, BYTE *filename, FILE_DESCRIPTOR *file, UBYTE mode)

4.3.7.1 Description

Open file from ATA device. Initialise file pointer structure FILE_DESCRIPTOR.

4.3.7.2 Parameters

4.3.7.2.1 Parameter1 – dev

Pointer to the ATA device This will be **&ATADevice.**

4.3.7.2.2 Parameter2 –filename

Filename string of file to be transferred. This must be in the current directory.
Filename can be: **FAT32 long format**, or **short (8.3) format**.

Example: “acespades playing card” ***or*** “acespa~1”

4.3.7.2.3 Parameter3 – file

Pointer to file descriptor.

4.3.7.2.4 Parameter4 – mode

Current options: READ_ONLY open file for reading

4.3.7.3 Return Value

Return pointer to file descriptor.
Return NULL pointer on failure.

4.3.7.4 Example

```
/* config.c */
#define MAX_OPEN_FILES      8
FILE_DESCRIPTOR file[MAX_OPEN_FILES];
/* game.c */
ChangeDir(&ATADevice, "\\");
ChangeDir(&ATADevice, "images");

if (FileOpen(&ATADevice, “image.bin”, &file[0], READ_ONLY) != NULL) {
        …
}
else {
        TxString(&SerialDevice,1,"\r\nFile not found.\r\n");}
```

4.3.8 FileClose

UBYTE FileClose(const DEVICE *dev, FILE_DESCRIPTOR *file)

4.3.8.1 Description

Close the file.

4.3.8.2 Parameters

4.3.8.2.1 Parameter1 – dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.8.2.2 Parameter2 – file

Pointer to file descriptor.

4.3.8.3 Return Value

Not used.

4.3.8.4 Example

FileClose(&ATADevice, &file[0]);

4.3.9 FileGetChar

UWORD FileGetChar(const DEVICE *dev, FILE_DESCRIPTOR *file)

4.3.9.1 Description

(Character oriented file access)
Read character from current reading position in file.

One sector (512 bytes) of data is buffered in a buffer within statically declared File descriptor structure.

4.3.9.2 Parameters

4.3.9.2.1 Parameter1 – dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.9.2.2 Parameter2 – file

Pointer to file descriptor.

4.3.9.3 Return Value

Returns next character from file.
Returns EOF at end-of-file or failure.

4.3.9.4 Example

FileOpen(&ATADevice, “msg.txt”, &file[0], READ_ONLY);

while ((ch= FileGetChar(&ATADevice, &file[0])) != EOF)
TxChar(&SerialDevice, 1, (UBYTE) ch);

4.3.10 FileGetSectors

ULONG FileGetSectors(const DEVICE *dev, FILE_DESCRIPTOR *file, UWORD *mem, ULONG nsectors)

4.3.10.1 Description

(Block oriented file access).
Read n sectors of data from file at current reading position to buffer. (One sector = 512 bytes).

4.3.10.2 Parameters

4.3.10.2.1 Parameter1 – dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.10.2.2 Parameter2 – file

Pointer to file descriptor.

4.3.10.2.3 Parameter3 – mem

Pointer to memory buffer to which data will be loaded.

4.3.10.2.4 Parameter4 – nsectors

Number of sectors of data to be loaded.

4.3.10.3 Return Value

Returns number of sectors read.

4.3.10.4 Example

```
/* animation */
if (FileOpen(&ATADevice, "animation.bin", &file[0], READ_ONLY) != NULL)
```

```
 {
/* each animation frame occupies 258 sectors */
        while (FileGetSectors(&ATADevice, &file[0], (UWORD*)(VideoMemBase+offscreen), 258))
          PutSprite(&CremsonDevice, screen1, 0, 0, offscreen, width, height, 0);
}
```

4.3.11 FileLoad

UBYTE FileLoad(const DEVICE *dev, FILE_DESCRIPTOR *file, UWORD *mem)

4.3.11.1 Description

Load entire file to buffer. Rewinds to start of file on completion.

4.3.11.2 Parameters

4.3.11.2.1 Parameter1 – dev

Pointer to the ATA device This will be **&ATADevice.**

4.3.11.2.2 Parameter2 – file

Pointer to file descriptor.

4.3.11.2.3 Parameter3 – mem

Pointer to memory buffer to which data will be loaded.

4.3.11.3 Return Value

Returns 0 on success.
Returns1 if file not open.

4.3.11.4 Example

FileOpen(&ATADevice, “image.bin”, &file[0], READ_ONLY);
FileLoad(&ATADevice, &file[0], (UWORD)(VideoMemBase+offscreen));*

4.3.12 FileRewind

UBYTE FileRewind(const DEVICE *dev, FILE_DESCRIPTOR *file)

4.3.12.1 Description

Rewind current reading position to start of file.

4.3.12.2 Parameters

4.3.12.2.1 Parameter1 – dev

Pointer to the ATA device which will be **&ATADevice.**

4.3.12.2.2 Parameter2 – file

Pointer to file descriptor.

4.3.12.3 Return Value

Not used.

4.3.12.4 Example

FileRewind(&ATADevice, &file[0]);

4.3.13 GetFileSize

ULONG GetFileSize(FILE_DESCRIPTOR *file)

4.3.13.1 Description

Get file size in bytes.

4.3.13.2 Parameters

4.3.13.2.1 Parameter2 – file

Pointer to file descriptor.

4.3.13.3 Return Value

Return file size in bytes.

4.3.13.4 Example

```
sprintf(str, "Transfer speed %d bytes sec-1\r\n", GetFileSize(&file[0]) / transfer_time );
TxString(&SerialDevice,1,str);
```

Figure 1 – Pluto 5 Software Structure