Circle

Rene Stange

Mar 11, 2022

CONTENTS

Note: The latest information refers to the current development version on the *develop* branch.

CHAPTER

ONE

LICENSE

CHAPTER

TWO

FOREWORD

This documentation has been written to help Circle users to write great bare metal applications. If you want to help to improve this documentation, please give feedback in the Project issues or send a Pull request, if you have corrections or improvements.

CHAPTER

THREE

INTRODUCTION

The Circle project provides a C++ bare metal environment for the Raspberry Pi single-board computers (SBC). This is a framework for developing applications, which run on the bare hardware, without using an operating system, which is somewhat equivalent to programming a very powerful micro-controller. Frequent areas of application for the bare metal system model are:

- High-speed data acquisition (DAQ)
- Retro computer emulation with accurate timing
- Low latency, high performance audio processing

Characteristics of bare metal solutions can be:

- Low interrupt latency
- Full system control[1]
- Light-weighted software architecture[2]
- Direct hardware access[3]
- Quick system start (boot)
- Can power off the system at any time[4]

This documentation provides the necessary information for developing bare metal applications using Circle.

[1] Secondary CPU cores can be dedicated to a specific task.
[2] Common operating systems work with many software layers instead.
[3] Do not need to write a device driver to access hardware interfaces.
[4] While the green Activity LED is off.

CHAPTER

FOUR

GETTING STARTED

To start using Circle, you need to download the project and a toolchain[1], configure Circle for your target platform, build the Circle libraries and your application[2], and install the built binary image (the kernel image)[3] on a SD card, along with a number of firmware files. In some cases an additional configuration file *config.txt* is needed on the SD card. The following notes require a x86_64 PC running Linux as development host. The file doc/windows-build.txt describes, how Windows can be used instead.

4.1 Download

The Circle project can be downloaded using *Git* as follows:

```
cd /path/to/your/projects
git clone https://github.com/rsta2/circle.git
```

The recommended toolchains for building Circle applications can be downloaded from here. Please note that there are different toolchains for 32-bit (AArch32, normally *arm-none-eabi-*) and for 64-bit (AArch64, normally *aarch64-none-elf-*) targets.

4.2 Configuration

Circle is configured using the file *Config.mk* in the project's root directory. This file can be created using the `configure` script, which provides these options:

```
-r <number>, --raspberrypi <number>
                    Raspberry Pi model number (1, 2, 3, 4, default: 1)
-p <string>, --prefix <string>
                    Prefix of the toolchain commands (default: arm-none-eabi-)
--multicore         Allow multi-core applications
--realtime          Enable real time mode to improve IRQ latency
--keymap <country> Set default USB keymap (DE, ES, FR, IT, UK, US)
--qemu              Build for running under QEMU
-d <option>, --define <option>
                    Define additional system option
--c++17             Use C++17 standard for compiling (default C++14)
```

(continues on next page)

[1] A toolchain in this context is cross compiler with additional tools and libraries, which runs on a specific platform and builds binaries for another (normally different) platform.

[2] For a start this can be one of the provided sample programs.

[3] Depending on the Raspberry Pi model and the target architecture (32- or 64-bit) a binary image has the filename *kernel.img*, *kernel7.img*, *kernel7l.img*, *kernel8.img* or *kernel8-rpi4.img*.

(continued from previous page)

```
-f, --force        Overwrite existing Config.mk file
-h, --help         Show usage message
```

If you want to configure Circle for a Raspberry Pi 3 with the default toolchain prefix `arm-none-eabi-`, with the toolchain path in the system `PATH` variable, from Circle's project root enter simply:

```
./configure -r 3
```

The file *Config.mk* can also be created by yourself. A typical 32-bit configuration looks like this:

```
PREFIX = /path/to/your/toolchain/bin/arm-none-eabi-
AARCH = 32
RASPPI = 3
```

This sets the path and name of your toolchain, and the architecture and model of your Raspberry Pi[4] computer.

Note: The configurable system options, described in the file include/circle/sysconfig.h, can be defined there or in the *Config.mk* file, like that:

```
DEFINE += -DOPTION_NAME
```

System options, which are enabled by default, can be disabled with:

```
DEFINE += -DNO_OPTION_NAME
```

A typical 64-bit configuration looks like that:

```
PREFIX64 = /path/to/your/toolchain/bin/aarch64-none-elf-
AARCH = 64
RASPPI = 3
```

64-bit operation is possible on the Raspberry Pi 3, 4 and Zero 2 only.

4.3 Building

After configuring Circle, go to the root directory of the Circle project and enter:

```
./makeall clean
./makeall
```

By default only the latest sample (with the highest number) is build. The ready build kernel image file should be in its subdirectory of *sample/*. If you want to build another sample after `./makeall` go to its subdirectory and do `make`.

[4] For the Raspberry Pi Zero and Zero W the target `RASPPI = 1` has to be configured. The Raspberry Pi Zero 2 W requires the target `RASPPI = 3`.

4.4 Installation

Copy the Raspberry Pi firmware (from *boot/* subdirectory, do `make` there to get them) files along with the *kernel*.img* (from *sample/* subdirectory) to a SD card with FAT file system.

The *config32.txt* file, provided in the *boot/* subdirectory, is needed to enable FIQ use in 32-bit mode on the Raspberry Pi 4 and has to be copied to the SD card in this case (rename it to *config.txt*). Furthermore the additional file *armstub7-rpi4.bin* is required on the SD card then. Please see boot/README for information on how to build this file.

The *config64.txt* file, provided in the *boot/* directory, is needed to enable 64-bit mode and has to be copied to the SD card in this case (rename it to *config.txt*). FIQ support for 64-bit mode on the Raspberry Pi 4 requires an additional file *armstub8-rpi4.bin* on the SD card. Please see boot/README for information on how to build this file.

Put the SD card into your Raspberry Pi and power it on.

CHAPTER

FIVE

HELLO WORLD!

Now we want to start developing our first Circle program. It may look like this:

Listing 1: main.cpp

```
#include <circle/startup.h>     // for EXIT_HALT
#include <circle/actled.h>
#include <circle/timer.h>

int main (void)
{
        CActLED ActLED;

        for (unsigned i = 1; i <= 10; i++)
        {
                ActLED.On ();
                CTimer::SimpleMsDelay (200);

                ActLED.Off ();
                CTimer::SimpleMsDelay (500);
        }

        return EXIT_HALT;
}
```

The program should be self-explanatory. `CTimer::SimpleMsDelay()` is a static delay function, which can be used, when there is no instance of the class `CTimer` in the system.

For a first test create a subdirectory in the *app/* directory and save this program as *main.cpp* there. Furthermore you need the following *Makefile* in the same directory:

Listing 2: Makefile

```
CIRCLEHOME = ../..

OBJS    = main.o

LIBS    = $(CIRCLEHOME)/lib/libcircle.a

include $(CIRCLEHOME)/Rules.mk

-include $(DEPS)
```

Now enter `make` in this directory and copy the resulting *kernel*.img* file to the SD card. When you power on your Raspberry Pi, the green Activity LED should blink ten times. Then the system halts.

5.1 The CKernel class

Normally an application is not that simple and we should apply some structure to our program, which can be used for any Circle application. In C++ the means of abstraction is a class and we want to define our application's main class now. In Circle it is usually called `CKernel`. It is a good practice to separate class definitions from its implementation, so we define the class in the header file *kernel.h*:

Listing 3: kernel.h

```
#ifndef _kernel_h
#define _kernel_h

//#include <circle/memory.h>
#include <circle/actled.h>
#include <circle/types.h>

enum TShutdownMode
{
        ShutdownNone,
        ShutdownHalt,
        ShutdownReboot
};

class CKernel
{
public:
        CKernel (void);
        ~CKernel (void);

        boolean Initialize (void);

        TShutdownMode Run (void);

private:
        //CMemorySystem m_Memory;       // not needed any more
        CActLED         m_ActLED;
};

#endif
```

You should create a new subdirectory under *app/* and save this file there. Beside the class constructor `CKernel()` and destructor `~CKernel()` there are the methods `Initialize()` and `Run()`. This implements a three step initialization for the class members, which is common throughout Circle:

1. The constructor `CKernel()` does some basic initialization for the class member variables.
2. The method `Initialize()` completes the initialization of the class members and returns `TRUE`, if the initialization was successful.
3. The method `Run()` is entered to start the execution of the application. When it returns, the application halts or the system reboots, depending of the returned value of type `TShutdownMode`. Many applications never return

from `Run()`.

Note: Circle uses the type `boolean` with the possible values `TRUE` and `FALSE` for historical reasons. You can use `bool`, `true` and `false` instead, which is equivalent.

Note: Earlier Circle versions required a member of the class `CMemorySystem` in `CKernel`, which initializes and manages the system memory. An instance of `CMemorySystem` is created now, before the function `main()` is called, so that there is no need to add it to `CKernel` any more. For compatibility `CMemorySystem` may still be instantiated in `CKernel`, but this is deprecated.

A possible class implementation for `CKernel`, with the same function as the "Hello world!" program before, looks as follows:

Listing 4: kernel.cpp

```
#include "kernel.h"
#include <circle/timer.h>

CKernel::CKernel (void)
{
}

CKernel::~CKernel (void)
{
}

boolean CKernel::Initialize (void)
{
        return TRUE;
}

TShutdownMode CKernel::Run (void)
{
        for (unsigned i = 1; i <= 10; i++)
        {
                m_ActLED.On ();
                CTimer::SimpleMsDelay (200);

                m_ActLED.Off ();
                CTimer::SimpleMsDelay (500);
        }

        return ShutdownHalt;
}
```

The class constructor `CKernel()` and destructor `~CKernel()` and the method `Initialize()` are not really used here, but this will change in real applications. Please note, that the constructor of the member variable `m_ActLED` is implicitly called in `CKernel()`. This call is automatically generated by the compiler.

Now that we have defined and implemented the class `CKernel`, we still have to provide a `main()` function, which implements the three step procedure given above for our class. This can be done as follows:

Listing 5: main.cpp

```
#include "kernel.h"
#include <circle/startup.h>

int main (void)
{
        CKernel Kernel;
        if (!Kernel.Initialize ())
        {
                halt ();
                return EXIT_HALT;
        }

        TShutdownMode ShutdownMode = Kernel.Run ();

        switch (ShutdownMode)
        {
        case ShutdownReboot:
                reboot ();
                return EXIT_REBOOT;

        case ShutdownHalt:
        default:
                halt ();
                return EXIT_HALT;
        }
}
```

This *main.cpp* file is part of most Circle programs without changes.

Note: Because some destructors used in `CKernel` may not be implemented, `main()` never really returns, but calls `halt()` or `reboot()` instead. Because we want to provide a common implementation of *main.cpp* here, we have to accept this little flaw here. In fact with the described `CKernel` implementation, it would be possible to return from `main()`, but this need not be the case in other Circle applications.

Finally we have to add *kernel.o* to the *Makefile* listed above:

Listing 6: Makefile

```
CIRCLEHOME = ../..

OBJS    = main.o kernel.o

LIBS    = $(CIRCLEHOME)/lib/libcircle.a

include $(CIRCLEHOME)/Rules.mk

-include $(DEPS)
```

That's all. Now we have the basic structure of a Circle application and you should be able to build it using `make`.

CHAPTER

SIX

A MORE COMPLEX PROGRAM

Now that we know, how the basic structure of a Circle application looks like, we want to add some more often used classes and thus functionality. The following program is based on the *sample/04-timer*. You will need a HDMI display or a serial terminal, connected to your Raspberry Pi, to try it out.

First create a new subdirectory below *app/* and copy the files *main.cpp* and *Makefile* from the previously discussed program. These files remain unchanged. Only our `CKernel` class will be modified and extended. The class definition looks now as follows:

Listing 1: kernel.h

```
#ifndef _kernel_h
#define _kernel_h

#include <circle/actled.h>
#include <circle/koptions.h>
#include <circle/devicenameservice.h>
#include <circle/screen.h>
#include <circle/serial.h>
#include <circle/exceptionhandler.h>
#include <circle/interrupt.h>
#include <circle/timer.h>
#include <circle/logger.h>
#include <circle/types.h>

enum TShutdownMode
{
        ShutdownNone,
        ShutdownHalt,
        ShutdownReboot
};

class CKernel
{
public:
        CKernel (void);
        ~CKernel (void);

        boolean Initialize (void);

        TShutdownMode Run (void);
```

(continues on next page)

(continued from previous page)

```
private:
        static void TimerHandler (TKernelTimerHandle hTimer,
                                  void *pParam, void *pContext);

private:
        CActLED                 m_ActLED;
        CKernelOptions          m_Options;
        CDeviceNameService      m_DeviceNameService;
        CScreenDevice           m_Screen;
        CSerialDevice           m_Serial;
        CExceptionHandler       m_ExceptionHandler;
        CInterruptSystem        m_Interrupt;
        CTimer                  m_Timer;
        CLogger                 m_Logger;
};

#endif
```

We add the following classes as member objects to `CKernel`:

Class	Purpose
CKernelOptions	Provides command line options from *cmdline.txt*
CDeviceNameService	Maps device names to a pointer to the device object
CScreenDevice	Access to the HDMI display (screen)
CSerialDevice	Access to the serial interface (UART)
CExceptionHandler	Reports system faults (abort exceptions) for debugging
CInterruptSystem	Interrupt (IRQ and FIQ) handling
CTimer	Provides several time services
CLogger	System logging facility

Furthermore a private static `TimerHandler()` callback function is added, which is used to show the function of kernel timers, implemented by the `CTimer` class. The file *kernel.cpp* has been updated like this:

Listing 2: kernel.cpp

```
#include "kernel.h"

static const char FromKernel[] = "kernel";

CKernel::CKernel (void)
:       m_Screen (m_Options.GetWidth (), m_Options.GetHeight ()),
        m_Timer (&m_Interrupt),
        m_Logger (m_Options.GetLogLevel (), &m_Timer)
{
        m_ActLED.Blink (5);
}

CKernel::~CKernel (void)
{
}
```

In the constructor of `CKernel` the `CScreenDevice` member is explicitly initialized using the display width and height

from the configuration file *cmdline.txt* on the SD card. The display resolution can be selected in the first line of this file for example like this: `width=640 height=480`. The `CTimer` member uses interrupts (IRQ) to implement a system tick of 100 Hz and hence gets a pointer to the `CInterruptSystem` member object.

Note: All Circle options for *cmdline.txt* are listed in doc/cmdline.txt. All options must be specified in the first line, separated with a space.

The system logging facility `CLogger` is initialized with the wanted logging level and a pointer to the timer, so that it can log the system time. The logging level can be set in *cmdline.txt* by adding `loglevel=N`, where N is a number between 0 (panic) and 4 (debug, default). Only the log messages with a severity of smaller or equal then this value will be logged.

Listing 3: kernel.cpp (continued)

```
boolean CKernel::Initialize (void)
{
	boolean bOK = TRUE;

	if (bOK)
	{
		bOK = m_Screen.Initialize ();
	}

	if (bOK)
	{
		bOK = m_Serial.Initialize (115200);
	}

	if (bOK)
	{
		CDevice *pTarget = m_DeviceNameService.GetDevice (
					m_Options.GetLogDevice (), FALSE);
		if (pTarget == 0)
		{
			pTarget = &m_Screen;
		}

		bOK = m_Logger.Initialize (pTarget);
	}

	if (bOK)
	{
		bOK = m_Interrupt.Initialize ();
	}

	if (bOK)
	{
		bOK = m_Timer.Initialize ();
	}

	return bOK;
}
```

In the `Initialize()` method the second step of the class member initialization is done. The call to `m_Logger.Initialize()` gets a pointer to the logging device as a parameter, which is `&m_Screen` by default. If you add `logdev=ttyS1` to *cmdline.txt* you can read the messages on a connected serial terminal. The mapping from device name to device object pointer takes place in `m_DeviceNameService.GetDevice()`, which returns 0, if the device name is not found.

Important: The order of initialization is important. The same applies to the constructor and the order of member objects in the class definition in *kernel.h*.

Listing 4: kernel.cpp (continued)

```
TShutdownMode CKernel::Run (void)
{
        m_Logger.Write (FromKernel, LogNotice,
                        "An exception will occur after 15 seconds from now");

        m_Timer.StartKernelTimer (15 * HZ, TimerHandler);

        unsigned nTime = m_Timer.GetTime ();
        while (1)
        {
                while (nTime == m_Timer.GetTime ())
                {
                        // just wait a second
                }

                nTime = m_Timer.GetTime ();

                m_Logger.Write (FromKernel, LogNotice, "Time is %u", nTime);
        }

        return ShutdownHalt;
}
```

`m_Logger.Write()` writes a message of the given severity to the system log. `FromKernel` names the source of the message (see definition above). `m_Timer.StartKernelTimer()` triggers, that the `TimerHandler()` gets called after 15 seconds. `m_Timer.GetTime()` returns the current local system time in seconds since 1970-01-01 00:00:00. Because we do not use a real-time clock, the actual time is equal to the uptime of the system. The program generates a log message every second on the screen or serial terminal, if it is selected as logging device.

Listing 5: kernel.cpp (continued)

```
void CKernel::TimerHandler (TKernelTimerHandle hTimer,
                            void *pParam, void *pContext)
{
        void (*pInvalid) (void) = (void (*) (void)) 0x500000;

        (*pInvalid) ();
}
```

After 15 seconds the `TimerHandler()` is called and generates a “Prefetch abort” exception by jumping to the address 0x500000, because the memory region at this address is marked as “not executable”.

The Appendix *Analyzing exceptions* explains using this program, how the information can be analyzed, which is dis-

played, when an abort exception occurs.

CHAPTER

SEVEN

BASIC SYSTEM SERVICES

This section describes the basic system services, which are provided for applications by the Circle base library *libcircle.a*. Only those classes are discussed here, which are directly used by applications. All Circle classes are listed in doc/classes.txt.

The Circle project does not provide a single centralized C++ header file. Instead the header file(s), which must be included for a specific class, function or macro definition are specified in the related subsection.

7.1 System information

This section describes the classes `CMachineInfo` and `CKernelOptions`, which provide information about the Raspberry Pi model, on which the application is running, and the runtime options, which can be defined in the file *cmdline.txt* on the SD card.

7.1.1 CMachineInfo

Normally there is exactly one instance of the class `CMachineInfo` in the system, which is created by the Circle system initialization code. If another instance is created, it acts as an alias for the first instance.

```
#include <circle/machineinfo.h>
```

class **CMachineInfo**

static *CMachineInfo* **CMachineInfo::Get**(void)
Returns a pointer to the first instance of `CMachineInfo`.

Model information

TMachineModel *CMachineInfo*::**GetMachineModel**(void) const
Returns the Raspberry Pi model, the application is running on. Possible values are:

- MachineModelA
- MachineModelBRelease1MB256
- MachineModelBRelease2MB256
- MachineModelBRelease2MB512
- MachineModelAPlus
- MachineModelBPlus

- MachineModelZero
- MachineModelZeroW
- MachineModelZero2W
- MachineModel2B
- MachineModel3B
- MachineModel3APlus
- MachineModel3BPlus
- MachineModelCM
- MachineModelCM3
- MachineModelCM3Plus
- MachineModel4B
- MachineModel400
- MachineModelCM4
- MachineModelUnknown

const char *CMachineInfo::**GetMachineName**(void) const
Returns the name of the Raspberry Pi model, the application is running on.

unsigned CMachineInfo::**GetModelMajor**(void) const
Returns the major version (1-4) of the Raspberry Pi model, the application is running on, or zero if it is unknown.

unsigned CMachineInfo::**GetModelRevision**(void) const
Returns the revision number (1-) of the Raspberry Pi model, the application is running on, or zero if it is unknown.

TSoCType CMachineInfo::**GetSoCType**(void) const
Returns the type of the SoC (System on a Chip), the application is running on. Possible values are:

- SoCTypeBCM2835
- SoCTypeBCM2836
- SoCTypeBCM2837
- SoCTypeBCM2711
- SoCTypeUnknown

unsigned CMachineInfo::**GetRAMSize**(void) const
Returns the size of the SDRAM in MBytes of the Raspberry Pi model, the application is running on, or zero if it is unknown.

const char *CMachineInfo::**GetSoCName**(void) const
Returns the name of the SoC (System on a Chip), the application is running on.

u32 CMachineInfo::**GetRevisionRaw**(void) const
Returns the raw revision code of the Raspberry Pi model, the application is running on.

Clocks and peripherals

unsigned *CMachineInfo*::**GetActLEDInfo**(void) const
Returns the information, about how the green Activity LED is connected to the system. The result has to be masked with ACTLED_PIN_MASK to extract the GPIO pin number. If the result masked with ACTLED_ACTIVE_LOW is not zero, the LED is on, when the value 0 is written to the GPIO pin. If the result masked with ACTLED_VIRTUAL_PIN is not zero, the LED is connected to a GPIO expander, which is controlled by the firmware.

unsigned *CMachineInfo*::**GetClockRate**(u32 nClockId) const
Returns the current frequency in Hz of the system clock, selected by nClockId, which can have the following values:

- CLOCK_ID_CORE
- CLOCK_ID_ARM
- CLOCK_ID_UART
- CLOCK_ID_EMMC
- CLOCK_ID_EMMC2

unsigned *CMachineInfo*::**GetGPIOPin**(TGPIOVirtualPin Pin) const
Returns the physical GPIO pin number of the PWM audio pins. Pin can have the values GPIOPinAudioLeft or GPIOPinAudioRight.

unsigned *CMachineInfo*::**GetGPIOClockSourceRate**(unsigned nSourceId)
This method allows to enumerate the different clock sources for GPIO clocks. It returns the frequency in Hz of the GPIO clock source with the ID nSourceId, which can be zero to GPIO_CLOCK_SOURCE_ID_MAX. The returned value is GPIO_CLOCK_SOURCE_UNUSED, if the clock source is unused.

unsigned *CMachineInfo*::**GetDevice**(TDeviceId DeviceId) const
Returns the device number of the default I2C master in the system. DeviceId has to be set to DeviceI2CMaster.

boolean *CMachineInfo*::**ArePWMChannelsSwapped**(void) const
Returns TRUE, if the left PWM audio channel is PWM1 (not PWM0).

DMA channels

unsigned *CMachineInfo*::**AllocateDMAChannel**(unsigned nChannel)
Allocates an available DMA channel from the platform DMA controller. nChannel can be DMA_CHANNEL_NORMAL (normal DMA engine requested), DMA_CHANNEL_LITE (lite (or normal) DMA engine requested), DMA_CHANNEL_EXTENDED ("large address" DMA4 engine requested, on Raspberry Pi 4 only) or an explicit channel number (0-15). Returns the allocated channel number or DMA_CHANNEL_NONE on failure.

void *CMachineInfo*::**FreeDMAChannel**(unsigned nChannel)
Release an allocated DMA channel. nChannel is the channel number (0-15).

7.1.2 CKernelOptions

The class `CKernelOptions` provides the values of runtime options, which can be defined in the file *cmdline.txt* on the SD card. The supported options are listed in doc/cmdline.txt. There is exactly one or no instance of this class in the system. Only relatively simple programs can work without an instance of `CKernelOptions`.

```
#include <circle/koptions.h>
```

class **CKernelOptions**

static *CKernelOptions* ***CKernelOptions**::**Get**(void)
Returns a pointer to the only instance of `CKernelOptions`.

unsigned *CKernelOptions*::**GetWidth**(void) const

unsigned *CKernelOptions*::**GetHeight**(void) const
Return the requested width and height of the screen, or zero if not specified. These values will normally handed over to the constructor for the class `CScreenDevice`.

const char **CKernelOptions*::**GetLogDevice**(void) const

unsigned *CKernelOptions*::**GetLogLevel**(void) const
Return the name of the target device for the system log (default `tty1`) and the log level (default `LogDebug`), to be handed over to the constructor of the class `CLogger`.

const char **CKernelOptions*::**GetKeyMap**(void) const
Returns the country code of the requested keyboard map (option `keymap=`). The default can be set with the system option `DEFAULT_KEYMAP`.

unsigned *CKernelOptions*::**GetUSBPowerDelay**(void) const
Returns the requested USB power-on delay in milliseconds, or zero to use the default value.

boolean *CKernelOptions*::**GetUSBFullSpeed**(void) const
Returns `TRUE`, if the option `usbspeed=full` is given in *cmdline.txt*.

const char **CKernelOptions*::**GetSoundDevice**(void) const
Returns the configured sound device (option `sounddev=`). Defaults to an empty string.

unsigned *CKernelOptions*::**GetSoundOption**(void) const
Returns the value configured with the option `soundopt=` in *cmdline.txt* (0-2, default 0).

TCPUSpeed *CKernelOptions*::**GetCPUSpeed**(void) const
Returns `CPUSpeedMaximum`, if the option `fast=true` is given in *cmdline.txt*, or `CPUSpeedLow` otherwise.

unsigned *CKernelOptions*::**GetSoCMaxTemp**(void) const
Returns the enforced maximal temperature of the SoC (option `socmaxtemp=`) in degrees Celsius (default 60).

const unsigned **CKernelOptions*::**GetTouchScreen**(void) const
Returns the calibration parameters for the touchscreen. The returned pointer refers to an array with four elements (min-x, max-x, min-y, max-y). It is `nullptr`, if the option `touchscreen=` is not set.

7.2 Memory

Circle enables the Memory Management Unit (MMU) to be able to use the data cache of the CPU to speed up operation, but it does not make use of virtual memory to implement specific system features. The physical-to-virtual address mapping is one-to-one over the whole used memory space.[1] The memory layout for the different system configurations can be found in doc/memorymap.txt.

7.2.1 new and delete

Circle supports system heap memory. Memory can be allocated with the normal C++ `new` operator and freed with the `delete` operator. Allocating and freeing memory blocks is supported from `TASK_LEVEL` and `IRQ_LEVEL`, but not from `FIQ_LEVEL`.[2] Allocated memory blocks are always aligned to the maximum size of a cache-line in the system.[3]

Note: Circle keeps a number of linked lists to manage memory blocks of different sizes. The supported block sizes are defined by the system option `HEAP_BLOCK_BUCKET_SIZES`. By default the maximum manageable block size is 512 KByte. Larger memory blocks can be allocated, but not re-used after `delete`.

The `new` operator can have a parameter, which specifies the type of memory to be allocated:

Parameter	Description
HEAP_LOW	memory below 1 GByte
HEAP_HIGH	memory above 1 GByte (on Raspberry Pi 4 only)
HEAP_ANY	memory above 1 GB (if available) or memory below 1 GB (otherwise)
HEAP_DMA30	30-bit DMA-able memory (alias for HEAP_LOW)

This is especially important on the Raspberry Pi 4, which supports different SDRAM memory regions. For instance one can specify to allocate a 256 byte memory block above 1 GByte:

```
#include <circle/new.h>

unsigned char *p = new (HEAP_HIGH) unsigned char[256];
```

Further information on using memory type parameters is available in doc/new-operator.txt.

7.2.2 CMemorySystem

```
#include <circle/memory.h>
```

class **CMemorySystem**

The class `CMemorySystem` implements most of the memory management function inside Circle. There is normally exactly one instance of this class in each Circle application, which is created by the Circle system initialization code. Earlier versions of Circle required to explicitly create this instance in `CKernel`. This is deprecated now, but does not disturb either. If another instance of `CMemorySystem` is created, it is an alias for the first created instance.

[1] There is one exception from this rule. On the Raspberry Pi 4 the memory mapped I/O register space of the xHCI USB controller, which is connected using a PCIe interface, is re-mapped into the 4 GByte 32-bit address space, because it is physically located above the 4 GByte boundary, and would not be accessible in 32-bit mode otherwise.

[2] System execution levels (e.g. `TASK_LEVEL`) are described in the section *Synchronization*.

[3] 32 bytes on the Raspberry Pi 1 and Zero, 64 bytes otherwise

Methods callable from applications are:

size_t *CMemorySystem*::**GetMemSize**(void) const
: Returns the total memory size available to the application, as reported by the firmware.

size_t *CMemorySystem*::**GetHeapFreeSpace**(int nType) const
: Returns the free space on the heap of the given type, according to the memory type `HEAP_LOW`, `HEAP_HIGH` or `HEAP_ANY`. Does not cover memory blocks, which have been freed.

static *CMemorySystem* **CMemorySystem*::**Get**(void)
: Returns a pointer to the instance of `CMemorySystem`.

static void *CMemorySystem*::**DumpStatus**(void)
: Dumps some memory allocation status information. Requires `HEAP_DEBUG` to be defined.

7.2.3 CClassAllocator

The class `CClassAllocator` allows to define a class-specific allocator for a class, using a pre-allocated store of memory blocks. This can speed up memory allocation, if the maximum number of instances of the class is known and a class instance does not occupy too much memory space. If you want to use this technique for your own class, the class definition has to look like this:

Listing 1: myclass.h

```
#include <circle/classallocator.h>

class CMyClass
{
...

        DECLARE_CLASS_ALLOCATOR
};
```

You have to add the following to the end of the class implementation file:

Listing 2: myclass.cpp

```
#include "myclass.h"

...

IMPLEMENT_CLASS_ALLOCATOR (CMyClass)
```

Before an instance of your class can be created, one of these (macro-) functions have to be executed:

```
#include "myclass.h"

INIT_CLASS_ALLOCATOR (CMyClass, Number);                        // or:

INIT_PROTECTED_CLASS_ALLOCATOR (CMyClass, Number, Level);
```

The second variant initializes a class-specific allocator, which is protected with a spin-lock for concurrent use. *Number* is the number of pre-allocated memory blocks and *Level* the maximum execution level, from which `new` or `delete` for this class will be called.[?]

7.2.4 C functions

Circle provides the following C standard library functions for memory allocation:

```
#include <circle/alloc.h>

void *malloc (size_t nSize);
void *calloc (size_t nBlocks, size_t nSize);
void *realloc (void *pBlock, size_t nSize);
void free (void *pBlock);
```

7.3 Synchronization

This section discusses the different system execution levels of code inside a Circle application and how they can be synchronized. Furthermore the class `CSpinLock` will be introduced, which is the main synchronization object in multi-core environments, but also in single-core environments, because all Circle code should be prepared to run on multiple cores, at least where it is possible.

7.3.1 Execution levels

```
#include <circle/synchronize.h>
```

The current execution level is determined by the type of interrupt requests, which are enabled (i.e. allowed to occur) or active (i.e. currently handled) at a given time. Circle defines the following execution levels:

Level[1]	Currently running	Enabled interrupts
TASK_LEVEL	normal application code or task[2]	IRQ, FIQ
IRQ_LEVEL	IRQ handler or callback[3]	FIQ
FIQ_LEVEL	FIQ handler	

Interrupt requests of the same type (i.e. IRQ or FIQ) cannot be nested. That means, when for example an IRQ handler is running for one device, a triggered IRQ of another device has to wait for the execution of its IRQ handler, until the previous IRQ handler has been completed.

The execution level (e.g. `TASK_LEVEL`) of the currently running code is returned by the following function:

unsigned **`CurrentExecutionLevel`**`(void)`

The current execution level can be explicitly raised with this function:

void **`EnterCritical`**`(unsigned nTargetLevel = IRQ_LEVEL)`

`EnterCritical()` can be called with the same as the current execution level or with a higher level, but not with a lower one. Reducing the current execution level is possible with this function:

void **`LeaveCritical`**`(void)`

[1] These symbols are defined as C macros.
[2] Tasks are discussed in the section *Multitasking*.
[3] A number of callback functions in an Circle application (e.g. kernel timer handler) will be called directly from an IRQ handler.

In summary EnterCritical() is called to enter a critical code region, which must not be interrupted by an IRQ, or by both IRQ and FIQ, depending on the target level. This critical region will be left with LeaveCritical(). Calls to EnterCritical() can be nested with the same or increasing target level. Every EnterCritical() has its corresponding LeaveCritical().

Important: In a multi-core environment using EnterCritical() for synchronization (e.g. protecting data structures in a critical region) is not recommended or does not work at all. You should use spin locks (see below) instead. Furthermore, because Circle source code should be able to run in any environment, where possible, it is good practice to use spin locks also for code, which is developed for a single-core environment. If the system option ARM_ALLOW_MULTI_CORE is disabled, all spin lock operations mutate to calls of EnterCritical() and LeaveCritical() automatically.

7.3.2 CSpinLock

The class CSpinLock implements a spin lock, which is a synchronization object in a multi-core environment. It can be used to protect a data structure, which is shared between multiple cores, from destruction, when multiple cores are trying to access this data structure at the same time. The spin lock serializes the access, so that only one core can write or read the data at a time.

```
#include <circle/spinlock.h>
```

class **CSpinLock**

In Circle a spin lock is initialized with this constructor:

CSpinLock::**CSpinLock**(unsigned nTargetLevel = IRQ_LEVEL)
: nTargetLevel is the maximum execution level from which the spin lock is acquired and released.

void *CSpinLock*::**Acquire**(void)
: This method tries to acquire the spin lock. It also raises the execution level to the level given to the constructor. If the spin lock is currently acquired by another core, the execution will be stalled, until the spin lock is released by the other core.

void *CSpinLock*::**Release**(void)
: Releases the spin lock.

Important: Calls to Acquire() cannot be nested for the same spin lock. If doing so, the execution will freeze. Multiple spin locks can be acquired in a row, but must be released in the opposite order. Otherwise a system deadlock may occur randomly.

7.3.3 CGenericLock

This class is used for mutual exclusion (critical sections) from TASK_LEVEL, at places where it is not clear, if the scheduler (see *Multitasking*) is in the system and mutual exclusion must work between tasks or between multiple CPU cores otherwise. If the scheduler is available and the system option NO_BUSY_WAIT is defined, this lock is implemented by the class CMutex, otherwise by the class CSpinLock.

```
#include <circle/genericlock.h>
```

class **CGenericLock**

void *CGenericLock*::**Acquire**(void)
Acquires the lock. Execution blocks, if another task or CPU core has already acquired the lock.

void *CGenericLock*::**Release**(void)
Releases the lock. Execution of another task or CPU core, which is waiting for the lock, continues.

7.3.4 Memory barriers

Memory barriers are system control CPU instructions, which influence the access to the main memory. They can be important especially in multi-core applications to ensure, that data has been written to or read from memory at a given place in the code.

When a variable is written by one CPU core in a multi-core environment, this is normally recognized by the other CPU cores, but for synchronization purposes barriers may be required, if a write or read operation must be completed at a specific place in code.

```
#include <circle/synchronization.h>
```

Circle defines the following memory barriers:

DataSyncBarrier()
This barrier (also known as *DSB*) ensures, that all memory read and write operations have been completed, at the place where it is inserted in the code. It may be required to insert this barrier, after an application has written data from one CPU core, which will be read from an other CPU core afterwards.

DataMemBarrier()
This barrier (also known as *DMB*) ensures, that all memory read operations have been completed, at the place where it is inserted in the code. It may be required to insert this barrier, before an application will read data, which has been written by an other CPU core before.

7.4 System log

Circle uses a system log facility throughout the system to report status information from other system facilities or devices to the user. It is recommended to use this log facility from application code too and this is implemented in all Circle sample programs. While an application may write information messages directly to a device, the log facility provides additional services and is a standard tool for collecting status information in Circle. The system log is implemented by the class `CLogger`.

7.4.1 CLogger

```
#include <circle/logger.h>
```

class **CLogger**

There is exactly one or no instance of `CLogger` in the system. Only relatively simple programs can work without an instance of `CLogger`.

Initialization

CLogger::**CLogger**(unsigned nLogLevel, *CTimer* *pTimer = 0, boolean bOverwriteOldest = TRUE)
Creates the instance of CLogger. nLogLevel (0-4) determines, which log messages are included in the system log. Only messages with a log level smaller or equal to nLogLevel are considered. pTimer is a pointer to the system timer object. The time is not logged, if pTimer is zero. The following log levels are defined:

Level	Severity	Description
0	LogPanic	Halt the system after processing this message
1	LogError	Severe error in this component, system may continue to work
2	LogWarning	Non-severe problem, component continues to work
3	LogNotice	Informative message, which is interesting for the system user
4	LogDebug	Message, which is only interesting for debugging this component

Set bOverwriteOldest to FALSE, if you want to keep old log messages for Read(), even when the text ring buffer is full (see *Read the log*).

boolean *CLogger*::**Initialize**(*CDevice* *pTarget)
Initializes the system log facility. Returns TRUE on success. pTarget is a pointer to the device, to which the log messages will be written.

void *CLogger*::**SetNewTarget**(*CDevice* *pTarget)
Sets the target for the log messages to a new device.

static *CLogger* **CLogger*::**Get**(void)
Returns a pointer to the only instance of CLogger.

Write the log

void *CLogger*::**Write**(const char *pSource, TLogSeverity Severity, const char *pMessage, ...)
Writes a message from the module pSource with Severity (see table above) to the log. The message can be composed using format specifiers as supported by *CString*::Format().

void *CLogger*::**WriteV**(const char *pSource, TLogSeverity Severity, const char *pMessage, va_list Args)
Same function as Write(), but the message parameters are given as va_list.

Read the log

CLogger has a 16K (LOGGER_BUFSIZE) sized text ring buffer, which saves the written log messages. If this buffer is full, old messages will be overwritten by default. This behavior can be changed with the parameter bOverwriteOldest of the constructor.

int *CLogger*::**Read**(void *pBuffer, unsigned nCount, boolean bClear = TRUE)
Reads and deletes maximal nCount characters from the log buffer. The read characters will be returned in pBuffer. Set bClear to TRUE to remove the returned bytes from the buffer or to FALSE to keep them. Returns the number of characters actually read.

boolean *CLogger*::**ReadEvent**(TLogSeverity *pSeverity, char *pSource, char *pMessage, time_t *pTime, unsigned *pHundredthTime, int *pTimeZone)
Returns the next log event (message) from a log event queue with maximal 50 entries or FALSE, if the queue is empty. The buffers at pSource and pMessage must have the sizes LOG_MAX_SOURCE and LOG_MAX_MESSAGE. This queue is normally used by the class *CSysLogDaemon*, which sends log messages to a syslog server.

Log event notification

void *CLogger*::**RegisterEventNotificationHandler**(TLogEventNotificationHandler *pHandler)
Registers a callback function, which is executed, when a log event (message) arrives. This is normally used by the class *CSysLogDaemon*, which sends log messages to a syslog server. `TLogEventNotificationHandler` has the following prototype:

```
void TLogEventNotificationHandler (void);
```

void *CLogger*::**RegisterPanicHandler**(TLogPanicHandler *pHandler)
Registers a callback function, which is executed, before a system halt, which is triggered by a log message with severity `LogPanic`. This is normally used by the class *CSysLogDaemon*, which sends log messages to a syslog server. If `CSysLogDaemon` is not in the system, `RegisterPanicHandler()` can be used for other application purposes. `TLogPanicHandler` has to following prototype:

```
void TLogPanicHandler (void);
```

Quick access

The following macros allow a quick access to the system log.

LOGMODULE(name)
Defines the C-string `name` as a name for this source module for generating log messages with the macros below.

LOGPANIC(format, ...)

LOGERR(format, ...)

LOGWARN(format, ...)

LOGNOTE(format, ...)

LOGDBG(format, ...)
Writes a message with the given severity, `format` and optional parameters to the system log using the module name defined with `LOGMODULE()`.

7.5 Interrupts

This section describes the low-level hardware-interrupt support in Circle. This should be only of interest, if one wants to develop its own device driver or driver-like functions.

In the ARM architecture there are two types of interrupt request, IRQ and FIQ. The IRQ is the basic interrupt request type, can have multiple active sources on all Raspberry Pi models and is used to control most interrupt-driven devices. The FIQ (Fast Interrupt Request) is used for low-latency interrupts and can have only one active interrupt source at a time on the Raspberry Pi 1-3 and Zero. The Raspberry Pi 4 has an new interrupt controller (GIC-400) and may theoretically support multiple simultaneous FIQ sources, but for a homogeneous solution this currently not supported in Circle. Therefore there are normally multiple active interrupt sources in a system, which use the IRQ, but only up to one, which uses the FIQ.

7.5.1 CInterruptSystem

The class `CInterruptSystem` is the provider of hardware-interrupt support in Circle. Hardware-interrupt support is not mandatory in an application, but if it is used, there is exactly one instance of this class in the system.

```
#include <circle/interrupt.h>
```

class **CInterruptSystem**

boolean *CInterruptSystem*::**Initialize**(void)
Initializes the interrupt system. Returns `TRUE` on success.

Note: There is a two step initialization required for the interrupt system. Step one is done in the constructor of `CInterruptSystem`, step two in `Initialize()`.

---

void *CInterruptSystem*::**ConnectIRQ**(unsigned nIRQ, TIRQHandler *pHandler, void *pParam)
Connects an interrupt handler to an IRQ source (vector). The known interrupt sources are defined in <circle/bcm2835int.h> for the Raspberry Pi 1-3 and Zero and in <circle/bcm2711int.h> for the Raspberry Pi 4. An IRQ handler has the following prototype:

```
void IRQHandler (void *pParam);
```

`pParam` can be any user parameter and gets the value specified in the call to `ConnectIRQ()` for this IRQ source.

void *CInterruptSystem*::**DisconnectIRQ**(unsigned nIRQ)
Disconnects the interrupt handler from the given IRQ source.

void *CInterruptSystem*::**ConnectFIQ**(unsigned nFIQ, TFIQHandler *pHandler, void *pParam)
Connects an interrupt handler to a FIQ source. Only one active FIQ source is allowed at a time. An FIQ handler has the same prototype as an IRQ handler (see above).

void *CInterruptSystem*::**DisconnectFIQ**(void)
Disconnects the interrupt handler of the active FIQ source.

static *CInterruptSystem* **CInterruptSystem*::**Get**(void)
Returns a pointer to the only instance of `CInterruptSystem`.

Important: If one or more IRQ handlers in a system make use of floating point registers, the system option `SAVE_VFP_REGS_ON_IRQ` has to be enabled. The same applies accordingly to `SAVE_VFP_REGS_ON_FIQ` for FIQ handlers.

7.6 Time

This section describes the services, which are provided by Circle regarding time. This concerns:

- The current non-consecutive system time (local and UTC) in seconds since 1970-01-01 00:00:00
- The timezone, expressed in minutes +/- from UTC
- The current consecutive system up-time in seconds
- A coarse grained, consecutive system tick counter in 1/100 seconds
- A fine grained, consecutive system tick counter in microseconds

- A possibly greater number of kernel timers, which elapse after a given number of coarse system ticks, resulting in a callback function to be executed
- Up to four periodic timer handlers, called 100 times per second
- Delaying program execution for a number of milli-, micro- or nanoseconds
- One fine grained periodic user timer, executing a callback function in an interval down to microseconds
- Converting time values (seconds since 1970-01-01 00:00:00) into time components (i.e. year, month, day, hour, minute, seconds) or reversed or into a string representation

These services are implemented in the classes `CTimer`, `CUserTimer` and `CTime`.

Note: "Consecutive" in this context means, that the time does never "jump". For example the current local system time may be updated, while the system is running, from an external time source (e.g. NTP server). This may cause a step back or forward in time. Consecutive time sources ensure, that this does not happen, which is important, e.g. when a program waits for an amount of time to pass, by calculating the difference between the current time and a start time.

7.6.1 CTimer

This class is the main provider of time services in Circle.

```
#include <circle/timer.h>
```

class **CTimer**

There is exactly one or no instance of this class in the system. Only relatively simple programs can work without an instance of `CTimer`. The only static timer functions, which can be called before this instance of `CTimer` is created and initialized, are:

static unsigned *CTimer*::**GetClockTicks**(void)
: Returns the current value of the fine grained, consecutive system tick counter in microseconds. It does not necessarily start at zero and may overrun after a while. It continues to count from zero then.

CLOCKHZ
: Frequency of the fine grained, consecutive system tick counter (1000000).

static void *CTimer*::**SimpleMsDelay**(unsigned nMilliSeconds)

static void *CTimer*::**SimpleusDelay**(unsigned nMicroSeconds)
: Delay the program execution by the given amount of time.

Initialization

CTimer::**CTimer**(*CInterruptSystem* *pInterruptSystem)
: Creates the instance of `CTimer`.

boolean *CTimer*::**Initialize**(void)
: Initializes and activates the system timer services. Returns `TRUE` on success.

Note: `CTimer::Initialize()` may generate log messages. Therefore it requires an initialized instance of the class `CLogger` in the system.

`CTimer::Initialize()` determines the CPU speed by calibrating a delay loop by default. This can be suppressed with the system option `NO_CALIBRATE_DELAY` (e.g. to reduce boot time).

---

static *CTimer* *`CTimer::Get`(void)
Returns a pointer to the single instance of `CTimer`.

Local time and UTC

boolean *CTimer*::**SetTimeZone**(int nMinutesDiff)

int *CTimer*::**GetTimeZone**(void) const
Sets or returns the current timezone in minutes difference to UTC.

boolean *CTimer*::**SetTime**(unsigned nTime, boolean bLocal = TRUE)
Sets the current system time in seconds since 1970-01-01 00:00:00. The time is given according to the timezone by default or in UTC, if the parameter `bLocal` is FALSE. Returns `TRUE`, if the time is valid.

unsigned *CTimer*::**GetTime**(void) const

unsigned *CTimer*::**GetLocalTime**(void) const

boolean *CTimer*::**GetLocalTime**(unsigned *pSeconds, unsigned *pMicroSeconds)
Returns the current local system time in seconds since 1970-01-01 00:00:00. The third variant always returns `TRUE`.

unsigned *CTimer*::**GetUniversalTime**(void) const

boolean *CTimer*::**GetUniversalTime**(unsigned *pSeconds, unsigned *pMicroSeconds)
Returns the current universal system time (UTC) in seconds since 1970-01-01 00:00:00. This value may be invalid, if the time was not set and the timezone difference is greater than zero. The third variant returns `FALSE` in this case.

CString *`CTimer::GetTimeString`(void)
Returns the current local system time as a string (format `"[MMM dD ]HH:MM:SS.ss"`). Returns zero, when `Initialize()` has not been called yet. The resulting `CString` object must be deleted by the caller.

Coarse system tick and up-time

unsigned *CTimer*::**GetTicks**(void) const
Returns the current value of the coarse grained, consecutive system tick counter in 1/100 seconds units.

Note: `CTimer::GetTicks()` reads the ticks variable only and returns quickly. `CTimer::GetClockTicks()` reads a hardware register (on Raspberry Pi 1 and Zero) or has to do some calculations (in 64-bit mode). Therefore calling `CTimer::GetTicks()` does normally cost less CPU cycles. You should use `CTimer::GetTicks()`, if its precision is sufficient for your purpose, or `CTimer::GetClockTicks()` otherwise.

HZ
Frequency of the coarse grained, consecutive system tick counter (100).

unsigned *CTimer*::**GetUptime**(void) const
Returns the system up-time in seconds, since the class `CTimer` has been initialized.

Kernel timers

TKernelTimerHandle *CTimer*::**StartKernelTimer**(unsigned nDelay, TKernelTimerHandler *pHandler, void *pParam = 0, void *pContext = 0)

Start a kernel timer, which elapses after `nDelay` coarse system ticks (100 Hz). Call `pHandler` on elapse with the given values of `pParam` and `pContext`. Returns a handle to the started timer. `TKernelTimerHandler` has the following prototype:

```
void TKernelTimerHandler (TKernelTimerHandle hTimer, void *pParam, void *pContext);
```

MSEC2HZ(msecs)

A macro, which converts milliseconds into coarse system ticks.

void *CTimer*::**CancelKernelTimer**(TKernelTimerHandle hTimer)

Cancel (remove) the kernel timer given with the handle `hTimer`. It will not elapse any more.

Periodic timers

void *CTimer*::**RegisterPeriodicHandler**(TPeriodicTimerHandler *pHandler)

Register a periodic timer handler, which is called `HZ` times (100) per second. Up to four handlers are allowed. `TPeriodicTimerHandler` has the following prototype:

```
void TPeriodicTimerHandler (void);
```

Update time handler

void *CTimer*::**RegisterUpdateTimeHandler**(TUpdateTimeHandler *pHandler)

Register a handler, which is called when `SetTime()` is invoked. This allows the application to apply additional checks, before the new time is set.

typedef boolean **TUpdateTimeHandler**(unsigned nNewTime, unsigned nOldTime)

The handler gets the `nNewTime` to be set and the current `nOldTime` in seconds since 1970-01-01 00:00:00 UTC, and returns `TRUE`, if the new time can be set or `FALSE`, if the time is invalid. The call to `SetTime()` is ignored then.

Delay

void *CTimer*::**MsDelay**(unsigned nMilliSeconds)

void *CTimer*::**usDelay**(unsigned nMicroSeconds)

void *CTimer*::**nsDelay**(unsigned nNanoSeconds)

Delay the program execution by the given amount of time. These functions should be used, when an instance of `CTimer` is available in the system (i.e. instead of `SimpleMsDelay()` and `SimpleusDelay()`).

Note: The actual delay may deviate from the requested value to some degree, but is never smaller than requested.

7.6.2 CUserTimer

This class implements a fine grained, user programmable interrupt timer. It uses the system timer 1 hardware, which must not be used for other purposes in the application then.

```
#include <circle/usertimer.h>
```

class **CUserTimer**

CUserTimer::**CUserTimer**(*CInterruptSystem* *pInterruptSystem, TUserTimerHandler *pHandler, void *pParam = 0, boolean bUseFIQ = FALSE)
Creates an instance of `CUserTimer`. Only one is allowed. `pHandler` is a pointer to the callback function, which is executed, when the user timer elapses. By default the IRQ is used to trigger the interrupt. `bUseFIQ` has to be set to `TRUE` to use the FIQ instead (e.g. for high frequencies). `TUserTimerHandler` has this prototype:

```
void TUserTimerHandler (CUserTimer *pUserTimer, void *pParam);
```

boolean *CUserTimer*::**Initialize**(void)
Initializes the user timer. Returns `TRUE` on success. Automatically starts the user timer with a delay of 1 hour.

void *CUserTimer*::**Stop**(void)
Stops the user timer. It has to be re-initialized to be used again.

void *CUserTimer*::**Start**(unsigned nDelayMicros)
(Re-)starts the user timer to elapse after the given number of microseconds (> 1). This method must be called from the user timer handler to a set new delay. It can be called on a running user timer to update the delay.

USER_CLOCKHZ
Frequency of the user timer (1000000).

7.6.3 CTime

This class converts the time into different representations.

```
#include <circle/time.h>
```

type **time_t**
Time representation (normally) in seconds since 1970-01-01 00:00:00.

class **CTime**

CTime::**CTime**(void)
Creates an instance of `CTime`.

CTime::**CTime**(const *CTime* &rSource)
Creates an instance of `CTime` from a different `CTime` object (copy constructor).

void *CTime*::**Set**(time_t Time)
Sets the time to the number seconds since 1970-01-01 00:00:00.

boolean *CTime*::**SetTime**(unsigned nHours, unsigned nMinutes, unsigned nSeconds)
Sets the time from its components hours (0-23), minutes (0-59) and seconds (0-59). Returns `TRUE` if the time is valid.

boolean *CTime*::**SetDate**(unsigned nMonthDay, unsigned nMonth, unsigned nYear)
Sets the date from its components month-day (1-31), month (1-12) and year (1970-). Returns `TRUE` if the date is valid.

time_t *CTime*::**Get**(void) const
Returns the time in the number seconds since 1970-01-01 00:00:00.

unsigned *CTime*::**GetSeconds**(void) const

unsigned *CTime*::**GetMinutes**(void) const

unsigned *CTime*::**GetHours**(void) const

unsigned *CTime*::**GetMonthDay**(void) const

unsigned *CTime*::**GetMonth**(void) const

unsigned *CTime*::**GetYear**(void) const
Return the components of the time. See `SetTime()` and `SetDate()` for the possible value ranges.

unsigned *CTime*::**GetWeekDay**(void) const
Returns the weekday (0-6, Sun-Sat).

const char **CTime*::**GetString**(void)
Returns a string representation of the time. The format is `"WWW MMM DD HH:MM:SS YYYY"`, where "WWW" is the weekday.

7.7 Direct Memory Access (DMA)

Circle supports Direct Memory Access (DMA) using the platform DMA controller of the Raspberry Pi. This is implemented in the class `CDMAChannel`.

7.7.1 CDMAChannel

```
#include <circle/dmachannel.h>
```

class **CDMAChannel**

CDMAChannel::**CDMAChannel**(unsigned nChannel, *CInterruptSystem* *pInterruptSystem = 0)
Creates an instance of `CDMAChannel` and allocates a channel of the platform DMA controller. `nChannel` must be `DMA_CHANNEL_NORMAL` (normal DMA engine), `DMA_CHANNEL_LITE` (lite (or normal) DMA engine), `DMA_CHANNEL_EXTENDED` ("large address" DMA4 engine, on Raspberry Pi 4 only) or an explicit channel number (0-15). `pInterruptSystem` is a pointer to the instance of `CInterruptSystem` and is only needed for interrupt operation.

void *CDMAChannel*::**SetupMemCopy**(void *pDestination, const void *pSource, size_t nLength, unsigned nBurstLength = 0, boolean bCached = TRUE)
Setup a DMA memory copy transfer from `pSource` to `pDestination` with length `nLength`. `nBurstLength` > 0 increases the speed, but may congest the system bus. `bCached` determines, if the source and destination address ranges are in cached memory.

void *CDMAChannel*::**SetupIORead**(void *pDestination, u32 nIOAddress, size_t nLength, TDREQ DREQ)
Setup a DMA read transfer from the I/O port `nIOAddress` to `pDestination` with length `nLength`. `DREQ` paces the transfer from these devices:

- DREQSourceNone (no wait)
- DREQSourceEMMC
- DREQSourcePCMRX
- DREQSourceSMI
- DREQSourceSPIRX
- DREQSourceUARTRX

void *CDMAChannel*::**SetupIOWrite**(u32 nIOAddress, const void *pSource, size_t nLength, TDREQ DREQ)
Setup a DMA write transfer to the I/O port `nIOAddress` from `pSource` with length `nLength`. `DREQ` paces the transfer to these devices:

- DREQSourceNone (no wait)
- DREQSourceEMMC
- DREQSourcePCMTX
- DREQSourcePWM
- DREQSourcePWM1 (on Raspberry Pi 4 only)
- DREQSourceSMI
- DREQSourceSPITX
- DREQSourceUARTTX

void *CDMAChannel*::**SetupMemCopy2D**(void *pDestination, const void *pSource, size_t nBlockLength, unsigned nBlockCount, size_t nBlockStride, unsigned nBurstLength = 0)
Setup a 2D DMA memory copy transfer of `nBlockCount` blocks of `nBlockLength` length from `pSource` to `pDestination`. Skip `nBlockStride` bytes after each block on destination. Source is continuous. The destination cache, if any, is not touched. `nBurstLength` > 0 increases speed, but may congest the system bus. This method can be used to copy data to the framebuffer and is not supported with `DMA_CHANNEL_LITE`.

void *CDMAChannel*::**SetCompletionRoutine**(TDMACompletionRoutine *pRoutine, void *pParam)
Sets a DMA completion routine for interrupt operation. `pRoutine` is called, when the transfer is completed. `pParam` is a user parameter, which is handed over to the completion routine. `TDMACompletionRoutine` has the following prototype:

```
void TDMACompletionRoutine (unsigned nChannel, boolean bStatus, void *pParam);
```

`nChannel` is the channel number. `bStatus` is `TRUE`, if the transfer completed successfully.

void *CDMAChannel*::**Start**(void)
Starts the DMA transfer, which has been setup before.

boolean *CDMAChannel*::**Wait**(void)
Waits for the completion of the DMA transfer (for synchronous non-interrupt operation without completion routine). Returns `TRUE`, if the transfer was successful.

boolean *CDMAChannel*::**GetStatus**(void)
Returns `TRUE`, if the last completed transfer was successful.

7.7.2 DMA buffers

```
#include <circle/synchronize.h>
```

DMA_BUFFER(type, name, num)
Defines a buffer with `name` and `num` elements of `type` to be used for DMA transfers.

See doc/dma-buffer-requirements.txt for more information on DMA buffers.

7.7.3 Cache maintenance

```
#include <circle/synchronize.h>
```

void **CleanAndInvalidateDataCacheRange**(uintptr nAddress, size_t nLength)
Cleans and invalidates a memory range in the data cache.

7.8 GPIO access

This section presents the Circle classes, which implement digital input/output operations, using the pins exposed on the 40-pin GPIO (General Purpose Input/Output) header of the Raspberry Pi (26-pin on older models). This covers writing and reading a single or all exposed GPIO pin(s), providing GPIO clocks for different purposes (e.g. for Pulse Width Modulation (PWM) output), triggering interrupts from GPIO pins, using I2C and SPI interfaces, and switching the (green) Act LED.

Please see these documents for a description of the GPIO hardware:

- BCM2835 ARM Peripherals (for Raspberry Pi 1 and Zero)
- BCM2711 ARM Peripherals (for Raspberry Pi 4)

The first document is also valid for the Raspberry Pi 2, 3 and Zero 2 with some modifications (e.g. I/O base address).

Important: The Circle documentation (including all READMEs) uses SoC (BCM) numbers (0-53), when referring to specific GPIO pins. These numbers are different from the physical pin position numbers (1-40) on the GPIO header. Please see pinout.xyz for the mapping of SoC numbers to the header position.

7.8.1 CGPIOPin

This class encapsulates a GPIO pin, which can be read, write or inverted. A GPIO pin can trigger an interrupt, when a specific GPIO event occurs.

```
#include <circle/gpiopin.h>
```

class **CGPIOPin**

GPIO_PINS
Number of available GPIO pins (54).

Initialization

CGPIOPin::**CGPIOPin**(void)
Default constructor. Object must be initialized afterwards using `AssignPin()`, `SetMode()` and optionally `SetPullMode()`.

CGPIOPin::**CGPIOPin**(unsigned nPin, TGPIOMode Mode, *CGPIOManager* *pManager = 0)
Creates and initializes a `CGPIOPin` instance for GPIO pin number `nPin`, set pin mode `Mode`. `pManager` must be specified only, if this pin will trigger interrupts (IRQ). `nPin` can have a numeric value (0-53) or these special values:

- GPIOPinAudioLeft (GPIO pin, which gates the left PWM audio channel)
- GPIOPinAudioRight (GPIO pin, which gates the right PWM audio channel)

`Mode` can have these values:

- GPIOModeInput
- GPIOModeOutput
- GPIOModeInputPullUp
- GPIOModeInputPullDown
- GPIOModeAlternateFunction0
- GPIOModeAlternateFunction1
- GPIOModeAlternateFunction2
- GPIOModeAlternateFunction3
- GPIOModeAlternateFunction4
- GPIOModeAlternateFunction5

void *CGPIOPin*::**AssignPin**(unsigned nPin)
Assigns a GPIO pin number to the object. To be used together with the default constructor and `SetMode()`. See `CGPIOPin::CGPIOPin()` for the possible values for `nPin`.

void *CGPIOPin*::**SetMode**(TGPIOMode Mode, boolean bInitPin = TRUE)
Sets GPIO pin to `Mode`. See `CGPIOPin::CGPIOPin()` for the possible values. If `bInitPin` is `TRUE`, this method initializes the pull-up/down mode and output level (LOW) too. To be used together with the default constructor and `AssignPin()` or for dynamic changes of the direction for input/output pins.

void *CGPIOPin*::**SetPullMode**(TGPIOPullMode Mode)
Sets the pull-up/down mode to one of the following values:

- GPIOPullModeOff
- GPIOPullModeDown
- GPIOPullModeUp

Input / Output

void *CGPIOPin*::**Write**(unsigned nValue)
Sets the GPIO pin to `nValue` (output), which can be `LOW` (0) or `HIGH` (1).

unsigned *CGPIOPin*::**Read**(void) const
Returns the value, read from the GPIO pin (input). Can be `LOW` (0) or `HIGH` (1).

void *CGPIOPin*::**Invert**(void)
Sets the GPIO pin to the inverted value. For output pins only.

static void *CGPIOPin*::**WriteAll**(u32 nValue, u32 nMask)
Sets the GPIO pins 0-31 at once. `nValue` specifies the levels of GPIO pins 0-31 in the respective bits to be written, where `nMask` is a bit mask for the written value. Only those GPIO pins are affected, for which the respective bit is set in `nMask`. The other pins are not touched.

static u32 *CGPIOPin*::**ReadAll**(void)
Returns the level of the GPIO pins 0-31 in the respective bits.

Interrupts

A GPIO pin can trigger an interrupt (IRQ) under certain conditions. The `CGPIOPin` object must be initialized with a pointer to an instance of the class `CGPIOManager` for this purpose. There is maximal one instance of `CGPIOManager` in the system.

void *CGPIOPin*::**ConnectInterrupt**(TGPIOInterruptHandler *pHandler, void *pParam, boolean bAutoAck = TRUE)
Connects the interrupt handler function `pHandler` to the GPIO pin, to be called on a GPIO event. `pParam` is a user parameter, which will be handed over to the interrupt handler. If `bAutoAck` is TRUE, the GPIO event detect status will be automatically acknowledged, when the interrupt occurs. Otherwise, the interrupt handler must call `AcknowledgeInterrupt()`. The GPIO interrupt handler has the following prototype:

```
void TGPIOInterruptHandler (void *pParam);
```

void *CGPIOPin*::**DisconnectInterrupt**(void)
Disconnects the interrupt handler from the GPIO pin. The interrupt source(s) must be disabled before using `DisableInterrupt()` and `DisableInterrupt2()`, if they were enabled before.

void *CGPIOPin*::**EnableInterrupt**(TGPIOInterrupt Interrupt)
Enables a specific event condition to trigger an interrupt for this GPIO pin. `Interrupt` can be:

- GPIOInterruptOnRisingEdge
- GPIOInterruptOnFallingEdge
- GPIOInterruptOnHighLevel
- GPIOInterruptOnLowLevel
- GPIOInterruptOnAsyncRisingEdge
- GPIOInterruptOnAsyncFallingEdge

void *CGPIOPin*::**DisableInterrupt**(void)
Disables a previously enabled event condition from triggering an interrupt.

void *CGPIOPin*::**EnableInterrupt2**(TGPIOInterrupt Interrupt)
Same function as `EnableInterrupt()` for a second interrupt source.

void *CGPIOPin*::**DisableInterrupt2**(void)
Same function as `DisableInterrupt()` for a second interrupt source.

void *CGPIOPin*::**AcknowledgeInterrupt**(void)
Manually acknowledges the GPIO event detect status. To be called from the from interrupt handler, if `bAutoAck` was `FALSE`, when calling `ConnectInterrupt()`.

7.8.2 CGPIOPinFIQ

This class encapsulates a special GPIO pin, which is using the FIQ (Fast Interrupt Request) to handle GPIO interrupts with low latency. There is only one GPIO pin of this type allowed in the system.

```
#include <circle/gpiopinfiq.h>
```

class **CGPIOPinFIQ** : public *CGPIOPin*

`CGPIOPinFIQ` is derived from `CGPIOPin` and inherits its methods. For initialization it provides this special constructor:

CGPIOPinFIQ::**CGPIOPinFIQ**(unsigned nPin, TGPIOMode Mode, *CInterruptSystem* *pInterrupt)
The parameters are the same as for `CGPIOPin::CGPIOPin()`, with one exception: `pInterrupt` is a pointer the single interrupt system object in the system. A `CGPIOPinFIQ` object does not need an instance of `CGPIOManager` to generate interrupts.

7.8.3 CGPIOManager

This class implements an interrupt multiplexer for `CGPIOPin` instances. There must be exactly one instance of `CGPIOManager` in the system, if at least one GPIO pin triggers interrupts using the IRQ.

```
#include <circle/gpiomanager.h>
```

class **CGPIOManager**

CGPIOManager::**CGPIOManager**(*CInterruptSystem* *pInterrupt)
Creates a `CGPIOManager` instance. `pInterrupt` is a pointer to the interrupt system object.

boolean *CGPIOManager*::**Initialize**(void)
Initializes the `CGPIOManager` object. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if the initialization was successful.

7.8.4 CGPIOClock

A GPIO clock is a programmable digital clock generator. A Raspberry Pi computer provides several of these clocks. Their output is used for special system purposes (e.g. for the PWM and PCM / I2S devices) or can be directly connected to some GPIO pins. GPIO clocks are driven by an internal clock source with a specific clock frequency.

```
#include <circle/gpioclock.h>
```

class **CGPIOClock**

CGPIOClock::**CGPIOClock**(TGPIOClock Clock, TGPIOClockSource Source = GPIOClockSourceUnknown)
Creates a `CGPIOClock` instance for GPIO clock `Clock` with clock source `Source`. `Clock` can be:

Clock	Connected to
GPIOClock0	GPIO4 (ALT0) or GPIO20 (ALT5)
GPIOClock1	GPIO5 (ALT0) or GPIO21 (ALT5), Raspberry Pi 4 only
GPIOClock2	GPIO6 (ALT0)
GPIOClockPCM	PCM / I2S device
GPIOClockPWM	PWM device

The respective GPIO pin has to be set to the given `GPIOModeAlternateFunctionN` (ALTn), using a `CGPIOPin` object, so that the signal can be accessed at the GPIO header. `Source` can be:

Source	Raspberry Pi 1-3	Raspberry Pi 4
GPIOClockSourceOscillator	19.2 MHz	54 MHz
GPIOClockSourcePLLC	1000 MHz (varies)	1000 MHz (may vary)
GPIOClockSourcePLLD	500 MHz	750 MHz
GPIOClockSourceHDMI	216 MHz	unused

If `Source` is set to `GPIOClockSourceUnknown`, the clock source is selected automatically, when `StartRate()` is called.

void *CGPIOClock*`::`**`Start`**`(`unsigned nDivI, unsigned nDivF = 0, unsigned nMASH = 0`)`
Starts the clock using the given integer divider `nDivI` (1-4095). The MASH modes with `nDivF > 0` are described in the BCM2835 ARM Peripherals document.

boolean *CGPIOClock*`::`**`StartRate`**`(`unsigned nRateHZ`)`
Starts the clock with the given target frequency `nRateHZ` in Hertz. Assigns the clock source automatically. Returns `FALSE`, if the requested rate cannot be generated.

void *CGPIOClock*`::`**`Stop`**`(`void`)`
Stops the clock.

7.8.5 CPWMOutput

This class provides access to the Pulse Width Modulator (PWM) device, which can be used to generate (pseudo) analog signals on the GPIO pins 18 and 19 (two channels). These pins have to be set to `GPIOModeAlternateFunction5` using the class `CGPIOPin` for that purpose.

```
#include <circle/pwmoutput.h>
```

class **`CPWMOutput`**

CPWMOutput`::`**`CPWMOutput`**`(`TGPIOClockSource Source, unsigned nDivider, unsigned nRange, boolean bMSMode`)`
Creates a `CPWMOutput` object with clock source `Source` and the divider `nDivider` (equivalent to `nDivI`, 1-4095). See `CGPIOClock` for these parameters. For the parameters `nRange` (Range) and `bMSMode` (M/S mode) see the BCM2835 ARM Peripherals document.

void *CPWMOutput*`::`**`Start`**`(`void`)`
Starts the PWM clock and device.

void *CPWMOutput*`::`**`Stop`**`(`void`)`
Stops the PWM clock and device.

void *CPWMOutput*`::`**`Write`**`(`unsigned nChannel, unsigned nValue`)`
Write `nValue` (0-Range) to PWM channel `nChannel` (1 or 2).

PWM_CHANNEL1

PWM_CHANNEL2
Macros to be used for the `nChannel` parameter.

7.8.6 CI2CMaster

This class is a driver for the I2C master devices of the Raspberry Pi computer. The GPIO pin mapping for the I2C master devices is as follows:

nDevice	nConfig 0 (SDA SCL)	nConfig 1 (SDA SCL)	Raspberry Pi boards
0	GPIO0 GPIO1		Rev. 1
1	GPIO2 GPIO3		All other
2			None
3	GPIO2 GPIO3	GPIO4 GPIO5	Raspberry Pi 4 only
4	GPIO6 GPIO7	GPIO8 GPIO9	Raspberry Pi 4 only
5	GPIO10 GPIO11	GPIO12 GPIO13	Raspberry Pi 4 only
6	GPIO22 GPIO23		Raspberry Pi 4 only

The `Read()` and `Write()` methods (see below) may return the following error codes as a negative value:

Value	Description
I2C_MASTER_INALID_PARM	Invalid parameter
I2C_MASTER_ERROR_NACK	Received a NACK
I2C_MASTER_ERROR_CLKT	Received clock stretch timeout
I2C_MASTER_DATA_LEFT	Not all data has been sent / received

```
#include <circle/i2cmaster.h>
```

class **CI2CMaster**

CI2CMaster::**CI2CMaster**(unsigned nDevice, boolean bFastMode = FALSE, unsigned nConfig = 0)
Creates a `CI2CMaster` object for I2C master `nDevice` (0-6), with configuration `nConfig` (0 or 1). See the mapping above for these parameters. The default I2C clock is 100 KHz or 400 KHz, if `bFastMode` is `TRUE`. This can be modified with `SetClock()` for a specific transfer.

boolean *CI2CMaster*::**Initialize**(void)
Initializes the `CI2CMaster` object. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if the initialization was successful.

void *CI2CMaster*::**SetClock**(unsigned nClockSpeed)
Modifies the default clock before a specific transfer. `nClockSpeed` is the wanted I2C clock frequency in Hertz.

int *CI2CMaster*::**Read**(u8 ucAddress, void *pBuffer, unsigned nCount)
Reads `nCount` bytes from the I2C slave device with address `ucAddress` into `pBuffer`. Returns the number of read bytes or < 0 on failure. See the error codes above.

int *CI2CMaster*::**Write**(u8 ucAddress, const void *pBuffer, unsigned nCount)
Writes `nCount` bytes to the I2C slave device with address `ucAddress` from `pBuffer`. Returns the number of written bytes or < 0 on failure. See the error codes above.

7.8.7 CI2CSlave

This class is a driver for the I2C slave device. The GPIO pin mapping is as follows:

Raspberry Pi	SDA	SCL
1-3, Zero	GPIO18	GPIO19
4	GPIO10	GPIO11

```
#include <circle/i2cslave.h>
```

class **CI2CSlave**

CI2CSlave::**CI2CSlave**(u8 ucAddress)
Creates the `CI2CSlave` object and assigns the I2C address `ucAddress`.

boolean *CI2CSlave*::**Initialize**(void)
Initializes the `CI2CSlave` object. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if the initialization was successful.

int *CI2CSlave*::**Read**(void *pBuffer, unsigned nCount)
Reads `nCount` bytes from the I2C master into `pBuffer`. Returns the number of read bytes or < 0 on failure.

int *CI2CSlave*::**Write**(const void *pBuffer, unsigned nCount)
Writes `nCount` bytes to the I2C master from `pBuffer`. Returns the number of written bytes or < 0 on failure.

7.8.8 CSPIMaster

The class `CSPIMaster` is a driver for SPI master devices, with these features:

- SPI non-AUX devices only
- Standard mode (3-wire) only
- Chip select lines (CE0, CE1) are active low
- Polled operation only

The GPIO pin mapping is as follows:

nDevice	MISO	MOSI	SCLK	CE0	CE1	Support
0	GPIO9	GPIO10	GPIO11	GPIO8	GPIO7	All boards
1						class CSPIMasterAUX
2						None
3	GPIO1	GPIO2	GPIO3	GPIO0	GPIO24	Raspberry Pi 4 only
4	GPIO5	GPIO6	GPIO7	GPIO4	GPIO25	Raspberry Pi 4 only
5	GPIO13	GPIO14	GPIO15	GPIO12	GPIO26	Raspberry Pi 4 only
6	GPIO19	GPIO20	GPIO21	GPIO18	GPIO27	Raspberry Pi 4 only

GPIO0 and GPIO1 are normally reserved for the ID EEPROM of hat boards.

```
#include <circle/spimaster.h>
```

class **CSPIMaster**

CSPIMaster::**CSPIMaster**(unsigned nClockSpeed = 500000, unsigned CPOL = 0, unsigned CPHA = 0, unsigned nDevice = 0)
Creates an `CSPIMaster` instance for access to SPI master `nDevice` (see table above), with default SPI clock frequency `nClockSpeed` in Hertz, clock polarity `CPOL` (0 or 1) and clock phase `CPHA` (0 or 1).

boolean *CSPIMaster*::**Initialize**(void)
Initializes the SPI master. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if initialization was successful.

void *CSPIMaster*::**SetClock**(unsigned nClockSpeed)
Modifies the default SPI clock frequency before a specific transfer. `nClockSpeed` is the SPI clock frequency in Hertz. This method is not protected by an internal spin lock for multi-core operation.

void *CSPIMaster*::**SetMode**(unsigned CPOL, unsigned CPHA)
Modifies the default clock polarity / phase before a specific transfer. `CPOL` is the clock polarity (0 or 1) and `CPHA` is the clock phase (0 or 1). These parameters must match the SPI slave settings. This method is not protected by an internal spin lock for multi-core operation.

void *CSPIMaster*::**SetCSHoldTime**(unsigned nMicroSeconds)
Sets the additional time, CE# stays active after the transfer. The set value is valid for the next transfer only. Normally CE# goes inactive very soon after the transfer, this sets the additional time, CE# stays active.

int *CSPIMaster*::**Read**(unsigned nChipSelect, void *pBuffer, unsigned nCount)
Reads `nCount` bytes into `pBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or `ChipSelectNone`). Returns the number of read bytes or < 0 on failure.

int *CSPIMaster*::**Write**(unsigned nChipSelect, const void *pBuffer, unsigned nCount)
Writes `nCount` bytes from `pBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or `ChipSelectNone`). Returns the number of written bytes or < 0 on failure.

int *CSPIMaster*::**WriteRead**(unsigned nChipSelect, const void *pWriteBuffer, void *pReadBuffer, unsigned nCount)
Simultaneous writes and reads `nCount` bytes from `pWriteBuffer` and to `pReadBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or `ChipSelectNone`). Returns the number of transferred bytes or < 0 on failure.

7.8.9 CSPIMasterAUX

The class `CSPIMasterAUX` is a polling driver for the auxiliary SPI master (SPI1). The GPIO pin mapping is as follows:

MISO	MOSI	SCLK	CE0	CE1	CE2
GPIO19	GPIO20	GPIO21	GPIO18	GPIO17	GPIO16

The CE# signals are active low.

```
#include <circle/spimasteraux.h>
```

class **CSPIMasterAUX**

CSPIMasterAUX::**CSPIMasterAUX**(unsigned nClockSpeed = 500000)
Creates a `CSPIMasterAUX` object. Sets the default SPI clock frequency to `nClockSpeed` in Hertz.

boolean *CSPIMasterAUX*::**Initialize**(void)
Initializes the SPI1 AUX master. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if initialization was successful.

void *CSPIMasterAUX*::**SetClock**(unsigned nClockSpeed)
Modifies the default SPI clock frequency before a specific transfer. `nClockSpeed` is the SPI clock frequency in Hertz. This method is not protected by an internal spin lock for multi-core operation.

int *CSPIMasterAUX*::**Read**(unsigned nChipSelect, void *pBuffer, unsigned nCount)
Reads `nCount` bytes into `pBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or 2). Returns the number of read bytes or < 0 on failure.

int *CSPIMasterAUX*::**Write**(unsigned nChipSelect, const void *pBuffer, unsigned nCount)
Writes `nCount` bytes from `pBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or 2). Returns the number of written bytes or < 0 on failure.

int *CSPIMasterAUX*::**WriteRead**(unsigned nChipSelect, const void *pWriteBuffer, void *pReadBuffer, unsigned nCount)
Simultaneous writes and reads `nCount` bytes from `pWriteBuffer` and to `pReadBuffer`. Activates chip select `nChipSelect` (CE#, 0, 1 or 2). Returns the number of transferred bytes or < 0 on failure.

7.8.10 CSPIMasterDMA

The class `CSPIMasterDMA` is a driver for the SPI0 master device. It implements an asynchronous DMA operation. Optionally one can do synchronous polling transfers (e.g. for small amounts of data). The GPIO pin mapping of the SPI0 master device is as follows:

MISO	MOSI	SCLK	CE0	CE1
GPIO9	GPIO10	GPIO11	GPIO8	GPIO7

```
#include <circle/spimasterdma.h>
```

class **CSPIMasterDMA**

CSPIMasterDMA::**CSPIMasterDMA**(*CInterruptSystem* *pInterruptSystem, unsigned nClockSpeed = 500000, unsigned CPOL = 0, unsigned CPHA = 0, boolean bDMAChannelLite = TRUE)
Creates a `CSPIMasterDMA` object. Sets the default SPI clock frequency to `nClockSpeed` in Hertz, the clock polarity to `CPOL` (0 or 1) and the clock phase to `CPHA` (0 or 1). `pInterruptSystem` is a pointer to the interrupt system object. Set `bDMAChannelLite` to `FALSE` for very high speeds or transfer sizes >= 64K.

boolean *CSPIMasterDMA*::**Initialize**(void)
Initializes the SPI0 master. Usually called from `CKernel::Initialize()`. Returns `TRUE`, if initialization was successful.

void *CSPIMasterDMA*::**SetClock**(unsigned nClockSpeed)
Modifies the default SPI clock frequency before a specific transfer. `nClockSpeed` is the SPI clock frequency in Hertz.

void *CSPIMasterDMA*::**SetMode**(unsigned CPOL, unsigned CPHA)
Modifies the default clock polarity / phase before a specific transfer. `CPOL` is the clock polarity (0 or 1) and `CPHA` is the clock phase (0 or 1). These parameters must match the SPI slave settings.

void *CSPIMasterDMA*::**SetCompletionRoutine**(TSPICompletionRoutine *pRoutine, void *pParam)
Sets a completion routine `pRoutine` to be called, when the next transfer completes. `pParam` is a user parameter, which is handed over to the completion routine. The prototype of the completion routine looks like this:

```
void TSPICompletionRoutine (boolean bStatus, void *pParam);
```

bStatus is TRUE on success.

void *CSPIMasterDMA*::**StartWriteRead**(unsigned nChipSelect, const void *pWriteBuffer, void *pReadBuffer, unsigned nCount)

Starts a simultaneous write and read transfer of nCount bytes from pWriteBuffer and to pReadBuffer. Chip select nChipSelect (CE#, 0, 1 or ChipSelectNone) will be activated during the transfer. The buffers must be aligned to the size of a data-cache-line (see *DMA buffers*).

int *CSPIMasterDMA*::**WriteReadSync**(unsigned nChipSelect, const void *pWriteBuffer, void *pReadBuffer, unsigned nCount)

Simultaneous writes and reads nCount bytes from pWriteBuffer and to pReadBuffer. Activates chip select nChipSelect (CE#, 0, 1 or ChipSelectNone). Returns the number of transferred bytes or < 0 on failure. Synchronous (polled) operation for small amounts of data.

7.8.11 CSMIMaster

The class CSMIMaster is a driver for the Secondary Memory Interface (SMI) device of the Raspberry Pi. It supports the following features:

- Drives any combination of SMI data lines (GPIO8 to GPIO25)
- May also drive SMI address lines (GPIO0 to GPIO5)
- Does not use SOE / SWE lines on GPIO6 / GPIO7
- Read / Write operation in direct mode, or Write-only in DMA mode

---

Note: One must first call *CSMIMaster::SetupTiming()* with suitable timing information. The device bank to use and the address to assert on the *SAx* lines may then optionally be set with *CSMIMaster::SetDeviceAndAddress()*. Then direct mode may be used with *CSMIMaster::Read()* or *CSMIMaster::Write()*, or for DMA mode one must first call *CSMIMaster::SetupDMA()* with a suitable buffer, then *CSMIMaster::WriteDMA()* to flush the buffer to SMI.

```
#include <circle/smimaster.h>
```

class **CSMIMaster**

CSMIMaster::**CSMIMaster**(unsigned nSDLinesMask = 0x3FFFF, boolean bUseAddressPins = TRUE)

Creates a CSMIMaster object. There can be only one. nSDLinesMask is a bit mask, which determines which *SDx* lines should be driven. For example (1 << 0) | (1 << 5) for SD0 (GPIO8) and SD5 (GPIO13). bUseAddressPins enables the use of the address pins GPIO0 to GPIO5, if it is set to TRUE.

unsigned *CSMIMaster*::**GetSDLinesMask**(void)

Returns the nSDLinesMask, handed over to the constructor.

void *CSMIMaster*::**SetupTiming**(TSMIDataWidth Width, unsigned nCycle_ns, unsigned nSetup, unsigned nStrobe, unsigned nHold, unsigned nPace, unsigned nDevice = 0)

Sets up the SMI cycle. nWidth is the length of the data bus (see below). nCycle_ns is the clock period for the setup/strobe/hold cycle (in nanoseconds). nSetup is the setup time, that is used to decode the address value (in units of nCycle_ns). nStrobe is the width of the strobe pulse, that triggers the transfer (in units of nCycle_ns). nHold is the hold time, that keeps the signals stable after the transfer (in units of nCycle_ns). nPace is the pace time in between two cycles (in units of nCycle_ns). nDevice is the settings bank to use (0 .. 3).

enum **TSMIDataWidth**

Values for specifying the width of the SMI data bus:

- SMI8Bits
- SMI9Bits
- SMI16Bits
- SMI18Bits

void *CSMIMaster*::**SetupDMA**(void *pDMABuffer, unsigned nLength)
Sets up DMA for (potentially multiple) SMI cycles of data from the buffer `pDMABuffer` (must be DMA-aligned). `nLength` is the length of the buffer in bytes.

void *CSMIMaster*::**SetDeviceAndAddress**(unsigned nDevice, unsigned nAddr)
Defines the device and address to use for the next Read/Write operation. `nDevice` is the settings bank to use (0 .. 3). `nAddr` is the value to be asserted on the address pins *SAx*.

unsigned *CSMIMaster*::**Read**(void)
Issues a single SMI read cycle from the *SDx* lines, and returns the read value.

void *CSMIMaster*::**Write**(unsigned nValue)
Issues a single SMI write cycle, i.e. writes the value `nValue` to the (enabled) *SDx* lines.

void *CSMIMaster*::**WriteDMA**(boolean bWaitForCompletion)
Triggers a DMA transfer of a few cycles with the buffer/length specified in *SetupDMA()*. `bWaitForCompletion` specifies whether to wait for DMA completion before returning.

7.8.12 CActLED

This class switches the (green) Act(ivity) LED on or off. It automatically determines the Raspberry Pi model to use the right LED pin for the model.

```
#include <circle/actled.h>
```

class **CActLED**

CActLED::**CActLED**(boolean bSafeMode = FALSE)
Creates the `CActLED` object. Safe mode works with LEDs connected to GPIO expander and chain boot, but is not as quick.

void *CActLED*::**On**(void)
Switches the Act LED on.

void *CActLED*::**Off**(void)
Switches the Act LED off.

void *CActLED*::**Blink**(unsigned nCount, unsigned nTimeOnMs = 200, unsigned nTimeOffMs = 500)
Blinks the Act LED `nCount` times. The LED is `nTimeOnMs` milliseconds on and `nTimeOffMs` milliseconds off.

static *CActLED* **CActLED*::**Get**(void)
Returns a pointer to the single `CActLED` instance in the system (if any).

7.9 Multi-core support

Beginning with the Raspberry Pi 2, four cores are provided by a Cortex-A CPU. Circle distinguishes between the primary core 0 and the secondary cores 1 to 3 in a way, that all system operations including interrupt handling are running on the primary core 0. The secondary cores are free to be used by the application. This allows to implement time-critical or time-consuming operations on the secondary cores, without being disturbed by interrupts or other system functions. The optional scheduler and all tasks are running on core 0 too (see *Multitasking*).

Circle supports multi-core applications by handling the start-up of the secondary cores with the class *CMultiCoreSupport*, with the synchronization class *CSpinLock* and with *Memory barriers*.

The system option `ARM_ALLOW_MULTI_CORE` has to be defined to use multi-core support with the class `CMultiCoreSupport`. For performance reasons this system option should not be defined for single core applications.

Further information on using the multi-core support is available in the file doc/multicore.txt.

The sample programs *17-fractal* and *26-cpustress* can be build with multi-core support. A more complex multi-core example is the project MiniSynth Pi.

7.9.1 CMultiCoreSupport

```
#include <circle/multicore.h>
```

class **CMultiCoreSupport**

If you want to use the secondary CPU cores in your application, you have to define a user class, which is derived from the class `CMultiCoreSupport`.

CMultiCoreSupport`::`**CMultiCoreSupport**`(`*CMemorySystem* *pMemorySystem`)`
Creates the instance of `CMultiCoreSupport`. Must be invoked in the first place of the initializer list of the defined user class. The parameter `pMemorySystem` must be set to `CMemorySystem::Get()`, which can be included from `<circle/memory.h>`.

boolean *CMultiCoreSupport*`::`**Initialize**`(`void`)`
Initializes the multi-core support and starts the secondary cores. It is important, that this method is called, when the other system initialization is already done. Normally it is invoked at the last method in `CKernel::Run()`.

virtual void *CMultiCoreSupport*`::`**Run**`(`unsigned nCore`)` = 0
Overload this virtual method to define the entry for the secondary cores (1 to 3) into your application. It is invoked three times (once on each secondary core) with `nCore` being the number of the executing CPU core (1, 2 or 3).

Important: When a secondary core returns from `Run()`, the CPU core is automatically halted and will sleep. For unused cores you can simply return from this method.

Note: This method is not executed from the primary CPU core 0 by default. If you want to handle all CPU cores at the same place, you have to explicitly call the `Run()` method of your user defined multi-core class from `CKernel::Run()` with the parameter 0.

---

static unsigned *CMultiCoreSupport*`::`**ThisCore**`(`void`)`
Returns the number of the CPU core (0, 1, 2 or 3), which called this method.

static void *CMultiCoreSupport*::**HaltAll**(void)
In a multi-core environment this method halts all CPU cores. The current execution will be interrupted using an Inter-Processor Interrupt (IPI) and each core calls the `halt()` function in turn.

static void *CMultiCoreSupport*::**SendIPI**(unsigned nCore, unsigned nIPI)
Sends an Inter-Processor Interrupt (IPI) with the number `nIPI` to the core `nCore` (0, 1, 2 or 3). If this technique is used for application purposes, `nIPI` can have a user defined value from `IPI_USER` to `IPI_MAX`.

virtual void *CMultiCoreSupport*::**IPIHandler**(unsigned nCore, unsigned nIPI)
Overload this virtual method to receive Inter-Processor Interrupts (IPI) from other CPU cores. `nCore` is the number of the CPU core, which received the IPI and which is executing `IPIHandler()`. `nIPI` is the IPI number specified in the call to `CMultiCoreSupport::SendIPI()`.

Important: Be sure to pass calls to this method further to `CMultiCoreSupport::IPIHandler()` with the same parameters, if `nIPI < IPI_USER`. Otherwise the `CMultiCoreSupport::HaltAll()` method will not work, which is also invoked on a system panic condition (abort exception, assertion failed).

7.10 CPU clock rate management

Most Raspberry Pi models require a CPU clock rate management by the bare metal application to reach the maximum performance. This management continuously measures the current temperature of the CPU (actually the SoC) and regulates the clock rate of the ARM CPU, so that it is decreased, when the temperature is getting too high.

The absolute maximum of the allowed CPU temperature is 85 degrees Celsius. The firmware automatically ensures, that this limit is not exceeded. If the temperature comes near to this value, the firmware shows a warning icon in the upper right corner of the screen. Please read the Frequency management and thermal control documentation page to get more information on this.

The different Raspberry Pi models allow different maximum CPU clock rates and the the frequency of the ARM CPU, which is set after boot, is also different:

Raspberry Pi	Maximum CPU clock rate	Boot CPU clock rate	Remarks
1	700 MHz	700 MHz	No management required
2	900 MHz	600 MHz	
Zero	1000 MHz	700 MHz	
Zero 2	1000 MHz	600 MHz	
3 Model B	1200 MHz	600 MHz	
3 Model A+/B+	1400 MHz	600 MHz	
4	1500 MHz	600 MHz	Rev. 1.4: 1800 MHz
400	1800 MHz	600 MHz	Head sink included

Circle uses the class `CCPUThrottle` to implement a CPU clock rate management, which is described below. The sample program *26-cpustress* demonstrates its usage.

Important: After boot the CPU clock rate of the ARM CPU is not set to the allowed maximum on most Raspberry Pi models. Without further action, the bare metal application will not operate with maximum performance.

If you need the maximum performance at any time in your application and cannot handle, when the CPU is clocked down, you may need a head sink and/or fan installed.

`CCPUThrottle` should not be used together with code doing I2C or SPI transfers. Because clock rate changes to the CPU clock may also effect the CORE clock, this could result in changing transfer speeds.

Note: To keep the CPU performance at the maximum level, it is possible to use a Case Fan, which is especially available for the official Raspberry Pi 4 case. This fan has a control line, which has to be connected to a GPIO pin. To use such a fan with the class `CCPUThrottle`, you have to add the option `gpiofanpin=PIN` to the file cmdline.txt, where PIN is the GPIO pin number (SoC number, not header position) to which the control line of the fan is connected. The CPU speed is not throttled, when this option is used.

7.10.1 CCPUThrottle

```
#include <circle/cputhrottle.h>
```

class **CCPUThrottle**

CCPUThrottle::**CCPUThrottle**(TCPUSpeed InitialSpeed = CPUSpeedUnknown)
Creates the class `CCPUThrottle`. `InitialSpeed` is the CPU speed to be set initially, with these possible values:

- CPUSpeedLow
- CPUSpeedMaximum
- CPUSpeedUnknown

If `CPUSpeedUnknown` is selected as initial speed and the parameter `fast=true` is set in the file cmdline.txt, the resulting setting will be `CPUSpeedMaximum`, or `CPUSpeedLow` if not set.

static *CCPUThrottle* **CCPUThrottle*::**Get**(void)
Returns a pointer to the only `CCPUThrottle` object in the system (if any).

boolean *CCPUThrottle*::**IsDynamic**(void) const
Returns if CPU clock rate change is supported. Other Methods can be called in any case, but may be nop's or return invalid values, if `IsDynamic()` returns `FALSE`.

unsigned *CCPUThrottle*::**GetClockRate**(void) const
Returns the current CPU clock rate in Hz or zero on failure.

unsigned *CCPUThrottle*::**GetMinClockRate**(void) const
Returns the minimum CPU clock rate in Hz.

unsigned *CCPUThrottle*::**GetMaxClockRate**(void) const
Returns the maximum CPU clock rate in Hz.

unsigned *CCPUThrottle*::**GetTemperature**(void) const
Returns the current CPU (SoC) temperature in degrees Celsius or zero on failure.

unsigned *CCPUThrottle*::**GetMaxTemperature**(void) const
Returns the maximum CPU (SoC) temperature in degrees Celsius.

TCPUSpeed *CCPUThrottle*::**SetSpeed**(TCPUSpeed Speed, boolean bWait = TRUE)
Sets the CPU speed. `Speed` selects the speed to be set and overwrites the initial value. Possible values are:

- CPUSpeedLow
- CPUSpeedMaximum

`bWait` must be `TRUE` to wait for new clock rate to settle before return. Returns the previous setting or `CPUSpeedUnknown` on error.

boolean *CCPUThrottle*::**SetOnTemperature**(void)
Sets the CPU speed depending on current SoC temperature. Call this repeatedly all 2 to 5 seconds to hold the temperature down! Throttles the CPU down when the SoC temperature reaches 60 degrees Celsius Returns TRUE if the operation was successful.

Note: The default temperature limit of 60 degrees Celsius may be too small for continuous operation with maximum performance. The limit can be increased with the parameter socmaxtemp in the file cmdline.txt.

boolean *CCPUThrottle*::**Update**(void)
Same function as SetOnTemperature(), but can be called as often as you want, without checking the calling interval. Additionally checks for system throttled conditions, if a system throttled handler has been registered with RegisterSystemThrottledHandler(). Returns TRUE if the operation was successful.

Important: You have to repeatedly call SetOnTemperature() or Update(), if you use this class!

void *CCPUThrottle*::**RegisterSystemThrottledHandler**(unsigned StateMask, TSystemThrottledHandler *pHandler, void *pParam = 0)
Registers the callback pHandler, which is invoked from Update(), when a system throttled condition occurs, which is given in StateMask. pParam is any user parameter to be handed over to the callback function. StateMask can be composed from these bit masks by or'ing them together:

- SystemStateUnderVoltageOccurred
- SystemStateFrequencyCappingOccurred
- SystemStateThrottlingOccurred
- SystemStateSoftTempLimitOccurred

void *CCPUThrottle*::**DumpStatus**(boolean bAll = TRUE)
Dumps some information on the current CPU status to the *System log*. Set bAll to TRUE to dump all information. Only the current clock rate and temperature will be dumped otherwise.

7.11 Firmware access

In order for a device driver to make certain settings (e.g. size of the frame buffer), it sometimes needs to communicate with the firmware, which runs on the VPU co-processor. This may be necessary from application code too, if specific settings should be made, which are not supported by Circle. The firmware can be accessed using the Mailbox property interface. This is supported in Circle by the class CBcmPropertyTags.

Note: This Mailbox property interface wiki article does not describe all supported functions. Information on more functions can only be retrieved from the Linux source code.

7.11.1 CBcmPropertyTags

```
#include <circle/bcmpropertytags.h>
```

class **CBcmPropertyTags**

boolean *CBcmPropertyTags*::**GetTag**(u32 nTagId, void *pTag, unsigned nTagSize, unsigned nRequestParmSize = 0)

Makes a mailbox property call to the firmware with a single tag. `nTagId` is the tag identifier. The identifiers, used by Circle, are listed in the header file circle/bcmpropertytags.h. `pTag` points to the tag structure and `nTagSize` is the size of this structure. This header file defines the tag structure for a number of mailbox property functions too. The parameter `nRequestParmSize` specifies the number of bytes in the tag structure, which are passed as input parameters to the firmware, where the `TPropertyTag` header does not count. This parameter may be zero for property tags, which do not pass input parameters to the firmware. `GetTag()` returns `TRUE`, if the call succeeds.

boolean *CBcmPropertyTags*::**GetTags**(void *pTags, unsigned nTagsSize)

Makes a mailbox property call to the firmware with multiple tags at once. `pTags` points to the tags structure, which is a concatenation of multiple property tag structures. `nTagsSize` is the total size of this structure. `GetTags()` returns `TRUE`, if the call succeeds.

Example

The following code retrieves the current clock rate of the ARM CPU from the firmware with the `GetTag()` method:

```
#include <circle/bcmpropertytags.h>

unsigned CMyClass::GetClockRate (void)
{
        CBcmPropertyTags Tags;                  // this class

        TPropertyTagClockRate TagClockRate;     // the tag structure

        TagClockRate.nClockId = CLOCK_ID_ARM;   // input parameter

        if (!Tags.GetTag (PROPTAG_GET_CLOCK_RATE,
                          &TagClockRate, sizeof TagClockRate,
                          sizeof TagClockRate.nClockId))
        {
                return 0;                       // return 0 on failure
        }

        return TagClockRate.nRate;              // return the clock rate
}
```

An example for using the `GetTags()` method is available in the frame buffer driver.

7.12 Direct hardware access

Circle applications may need to directly read or write registers of hardware devices, if a driver for the respective device does not exist yet. This subsection describes the Circle support for accessing hardware registers.

7.12.1 Functions

```
#include <circle/memio.h>
```

u8 **read8**(uintptr nAddress)

u16 **read16**(uintptr nAddress)

u32 **read32**(uintptr nAddress)
Reads a value with the specified bit size from the memory-mapped I/O address `nAddress` and returns it.

void **write8**(uintptr nAddress, u8 uchValue)

void **write16**(uintptr nAddress, u16 usValue)

void **write32**(uintptr nAddress, u32 nValue)
Writes a value with the specified bit size to the memory-mapped I/O address `nAddress`. `uchValue`, `usValue` and `nValue` are the respective values.

Note: An access to a memory-mapped I/O device register must normally be aligned to the access size.

7.12.2 Macros

The detailed definitions for the different hardware devices of the Raspberry Pi cannot be listed here. Please read the respective header file for details.

```
#include <circle/bcm2835.h>
```

This header file provides macro definitions of memory-mapped I/O addresses for all Raspberry Pi models, described in the BCM2835 ARM Peripherals document, especially:

ARM_IO_BASE
Base address of the 16 MB sized main memory-mapped I/O block, valid on the ARM CPU of the respective Raspberry Pi model. This address is normally used from the Circle application.

GPU_IO_BASE
Base address of the 16 MB sized main memory-mapped I/O block, valid on the GPU co-processor. This address is used for operations, which are executed by the GPU or connected devices (e.g. DMA controllers).

Note: A Raspberry Pi has several processing units. We only distinguish here between the ARM CPU, where the Circle application is running on, and all other processing units, where the firmware, accelerated graphics processing and more is executed. We call these processors the GPU or VPU. Please note that from the point of view of the boot order, the ARM CPU is the secondary co-processor.

GPU_MEM_BASE
Base address of the lower (starting at address 0x0 on the ARM CPU) 1 GB memory address range, valid on the GPU and connected devices (e.g. DMA controllers). The legacy platform DMA controller, for instance, can only access this address space for data transfers.

BUS_ADDRESS(address)
Converts the memory address `address`, valid on the ARM CPU, to a GPU bus address, valid on the GPU and connected devices.

```
#include <circle/bcm2836.h>
```

This header file provides macro definitions of memory-mapped I/O addresses for the Raspberry Pi 2 to 4 and compatible models, described in the ARM Quad A7 core document, especially:

ARM_LOCAL_BASE
Base address of the 256 MB sized local memory-mapped I/O block. A number of registers from this block are local to the respective ARM CPU core.

```
#include <circle/bcm2711.h>
```

This header file provides macro definitions of memory-mapped I/O addresses for the Raspberry Pi 4 and compatible models, described in the BCM2711 ARM Peripherals document.

7.12.3 I/O barriers

The following I/O barriers are especially required on the Raspberry Pi 1 and Zero. On other Raspberry Pi models they have no function.

```
#include <circle/synchronization.h>
```

PeripheralEntry()
If your code directly accesses memory-mapped hardware registers, you should insert this special barrier before the first access to a specific hardware device.

PeripheralExit()
If your code directly accesses memory-mapped hardware registers, you should insert this special barrier after the last access to a specific hardware device.

Note: Most programs would work without `PeripheralEntry()` and `PeripheralExit()`, but to be sure, it should be used as noted. In a few tests there have been issues on the Raspberry Pi 1, where invalid data was read from hardware registers, without these barriers inserted.

You do not need to care about this, when you access hardware devices using a Circle device driver class, because this is handled inside the driver.

7.13 Utilities

This section lists utility classes and functions, which help to implement Circle applications.

7.13.1 CString

```
#include <circle/string.h>
```

class **CString**

This class encapsulates a character string and allows different manipulations on it. The methods of this class are not reentrant.

CString::**CString**(void)
Creates an empty string object ("");

CString::**CString**(const char *pString)
Creates a string object. Sets the string initially to `pString`.

CString::**CString**(const *CString* &rString)
Copy constructor. Creates a new string object. Sets the string initially to the value of `rString`. `rString` remains unchanged.

CString::**CString**(*CString* &&rrString)
Move constructor. Creates a new string object. Sets the string initially to the value of `rrString`. `rrString` is set to an empty string.

CString::**operator const char***(void) const
Returns a pointer to the string buffer, which is terminated with a zero-character.

const char **CString*::**operator=**(const char *pString)
Assigns a new string. Returns a pointer to the string buffer, which is terminated with a zero-character.

CString &*CString*::**operator=**(const *CString* &rString)
Assigns a new string. Returns a reference to the string object.

CString &*CString*::**operator=**(*CString* &&rrString)
Move assignment. Assigns a new string. `rrString` is set to an empty string. Returns a reference to the string object.

size_t *CString*::**GetLength**(void) const
Returns the length of the string in number of characters (zero for empty string).

void *CString*::**Append**(const char *pString)
Appends `pString` to the string.

int *CString*::**Compare**(const char *pString) const
Compares `pString` with the string. Returns:

- zero, if the strings are identical
- a negative value, if the string is smaller than `pString`
- a positive value, if the string is greater than `pString`

int *CString*::**Find**(char chChar) const
Searches for `chChar` in the string. Returns the zero-based index of the character or -1, if it is not found.

int *CString*::**Replace**(const char *pOld, const char *pNew)
Replaces all occurrences of `pOld` with `pNew` in the string. Returns the number of occurrences.

void *CString*::**Format**(const char *pFormat, ...)
Formats a string as known from `sprintf()`. Does support only a subset of the known format specifiers:

`%[#][[-][0]len][.prec][l|ll]{c|d|f|i|o|p|s|u|x|X}`

Field	Description
#	insert prefix 0, 0x or 0X for %o, %x or %X
-	left justify output
0	insert leading zeros
len	decimal number specifying the length of the field
.prec	decimal number specifying the precision for %f
l	type is `long`
ll	type is `long long` (with STDLIB_SUPPORT >= 1 only)
c	insert `char`
d	insert decimal `int`, `long` or `long long` (maybe with sign)
f	insert `double`
i	same as %d
o	insert octal `unsigned`, `unsigned long` or `unsigned long long`
p	same as %x
s	insert string (type is `const char *`)
u	insert decimal `unsigned`, `unsigned long` or `unsigned long long`
x	insert hex `unsigned`, `unsigned long` or `unsigned long long` (lower case)
X	insert hex `unsigned`, `unsigned long` or `unsigned long long` (upper case)

void *CString*::**FormatV**(const char *pFormat, va_list Args)
Same as `Format()`, but `Args` are given as `va_list`.

7.13.2 CPtrArray

```
#include <circle/ptrarray.h>
```

class **CPtrArray**

This class implements a dynamic array of pointers. The methods of this class are not reentrant.

CPtrArray::**CPtrArray**(unsigned nInitialSize = 100, unsigned nSizeIncrement = 100)
Creates a `CPtrArray` object with initially space for `nInitialSize` elements. The memory allocation will be increased by `nSizeIncrement` elements, when the array is full.

unsigned *CPtrArray*::**GetCount**(void) const
Returns the current number of used elements in the array.

void **CPtrArray*::**operator[]**(unsigned nIndex) const
Returns the pointer for the array element at `nIndex` (based on zero). `nIndex` must be smaller than the value returned from `GetCount()`.

void *&*CPtrArray*::**operator[]**(unsigned nIndex)
Returns a reference to the pointer for the array element at `nIndex` (based on zero). `nIndex` must be smaller than the value returned from `GetCount()`.

unsigned *CPtrArray*::**Append**(void *pPtr)
Appends `pPtr` to end of the array.

void *CPtrArray*::**RemoveLast**(void)
Removes the last element from the array.

7.13.3 CPtrList

```
#include <circle/ptrlist.h>
```

class **CPtrList**

This class implements a linked list of pointers. The methods of this class are not reentrant.

type **TPtrListElement**
Opaque type definition.

TPtrListElement *CPtrList::**GetFirst**(void)
Returns the first element, or 0 if list is empty.

TPtrListElement *CPtrList::**GetNext**(TPtrListElement *pElement)
Returns the next element following `pElement`, or 0 if nothing follows.

void *CPtrList::**GetPtr**(TPtrListElement *pElement)
Returns the pointer for `pElement`.

void CPtrList::**InsertBefore**(TPtrListElement *pAfter, void *pPtr)
Inserts `pPtr` before the element `pAfter`, which must not be 0.

void CPtrList::**InsertAfter**(TPtrListElement *pBefore, void *pPtr)
Inserts `pPtr` after the element `pBefore`. Use `pBefore == 0` to set the first element in the list (list must be empty).

void CPtrList::**Remove**(TPtrListElement *pElement)
Removes the element `pElement` from the list.

TPtrListElement *CPtrList::**Find**(void *pPtr)
Searches the element, whose pointer is equal to `pPtr` and returns it, or 0 if `pPtr` was not found.

7.13.4 CNumberPool

```
#include <circle/numberpool.h>
```

class **CNumberPool**

This class implements an allocation pool for numbers. The methods of this class are not reentrant.

static const unsigned **Limit** = 63
Allowed maximum of an allocated number.

static const unsigned **Invalid** = *Limit* + 1
Returned by `AllocateNumber()` on failure.

CNumberPool::**CNumberPool**(unsigned nMin, unsigned nMax = *Limit*)
Creates a number pool. `nMin` is the minimal returned number by `AllocateNumber()`. `nMax` is the maximal returned number.

unsigned CNumberPool::**AllocateNumber**(boolean bMustSucceed, const char *pFrom = "numpool")
Allocates a number from the number pool and returns it. If there are no more numbers available, this method returns `CNumberPool::Invalid`, if `bMustSucceed` is `FALSE`, or the system halts with a panic message otherwise. This message has the prefix `pFrom`.

void CNumberPool::**FreeNumber**(unsigned nNumber)
Returns `nNumber`, which has been allocated earlier, to the number pool for reuse.

7.13.5 Atomic memory access

```
#include <circle/atomic.h>
```

This header file defines some functions, which implement an atomic access to an aligned `int` variable in memory. These functions can be useful for synchronization purposes, especially for multi-core applications, where using a spin lock would be too time consuming. All accesses to such a variable must use one of the following functions, to ensure them being atomic.

int **AtomicGet**(volatile const int *pVar)
: Returns the value of the `int` variable at `pVar`.

int **AtomicSet**(volatile int *pVar, int nValue)
: Sets the `int` variable at `pVar` to `nValue` and returns `nValue`.

int **AtomicExchange**(volatile int *pVar, int nValue)
: Sets the `int` variable at `pVar` to `nValue` and returns the previous value.

int **AtomicCompareExchange**(volatile int *pVar, int nCompare, int nValue)
: Sets the `int` variable at `pVar` to `nValue`, if the previous value of the variable was `nCompare`, and returns the previous value of the variable.

int **AtomicAdd**(volatile int *pVar, int nValue)
: Adds `nValue` to the `int` variable at `pVar`. Returns the result of the operation.

int **AtomicSub**(volatile int *pVar, int nValue)
: Subtracts `nValue` from the `int` variable at `pVar`. Returns the result of the operation.

int **AtomicIncrement**(volatile int *pVar)
: Increments the `int` variable at `pVar` by 1. Returns the result of the operation.

int **AtomicDecrement**(volatile int *pVar)
: Decrements the `int` variable at `pVar` by 1. Returns the result of the operation.

7.13.6 C standard library functions

```
#include <circle/util.h>
```

This header file defines some functions, known from the C standard library.

Memory functions

void ***memset**(void *pBuffer, int nValue, size_t nLength)

void ***memcpy**(void *pDest, const void *pSrc, size_t nLength)

void ***memmove**(void *pDest, const void *pSrc, size_t nLength)

int **memcmp**(const void *pBuffer1, const void *pBuffer2, size_t nLength)

String functions

size_t **strlen**(const char *pString)

int **strcmp**(const char *pString1, const char *pString2)

int **strcasecmp**(const char *pString1, const char *pString2)

int **strncmp**(const char *pString1, const char *pString2, size_t nMaxLen)

int **strncasecmp**(const char *pString1, const char *pString2, size_t nMaxLen)

char ***strcpy**(char *pDest, const char *pSrc)

char ***strncpy**(char *pDest, const char *pSrc, size_t nMaxLen)

char ***strcat**(char *pDest, const char *pSrc)

char ***strchr**(const char *pString, int chChar)

char ***strstr**(const char *pString, const char *pNeedle)

char ***strtok_r**(char *pString, const char *pDelim, char **ppSavePtr)

Number conversion

unsigned long **strtoul**(const char *pString, char **ppEndPtr, int nBase)

unsigned long long **strtoull**(const char *pString, char **ppEndPtr, int nBase)

int **atoi**(const char *pString)

7.13.7 Other functions

```
#include <circle/util.h>
```

u16 **bswap16**(u16 usValue)

u32 **bswap32**(u32 ulValue)
: Swaps the byte order of a 16- or 32-bit value.

int **parity32**(unsigned nValue)
: Returns the number of 1-bits in `nValue` modulo 1.

7.13.8 Macros

```
#include <circle/macros.h>
```

PACKED
Packs a `struct` definition. The members will be stored tightly, not aligned as usual.

ALIGN(n)
Aligns a variable or member to a boundary of `n` in memory.

NORETURN
Append this to the prototype of a function, which never returns.

BIT(n)
Returns the bit mask `(1U << (n))`.

likely(exp)

unlikely(exp)
In time critical code this gives the compiler a hint, which result of the boolean expression `exp` is normally expected. This can result in faster code.

7.14 Debugging support

Circle provides a number of classes, functions and macros, which support the debugging of applications. This section describes the tools, which can be included in a program itself. Debugging a Circle application with an external debugger is described in doc/debug-jtag.txt and doc/debug.txt in the Circle repository.

Note: Beside the tools, which are described here, you can also use the *System log* to write debug messages to the screen or serial interface.

7.14.1 Assertions

Assertions are a common technique to insert runtime checks into the code. This is frequently used in the Circle libraries itself and is also recommended for application code. Assertions will be included in a checked build of Circle only and are ignored, when the macro symbol `NDEBUG` is defined.

```
#include <assert.h>
```

assert(expr)
Inserts a runtime check into the code. `expr` must be a true boolean expression, otherwise the system is halted with an "Assertion failed" message, which contains the filename and the source code line of the failed assertion, and with a stack trace.

ASSERT_STATIC(expr)
This is a static assertion, which will be evaluated at build time. It will be placed outside of a function, e.g. to check the size of a structure definition. The compiler generates an error message, if the expression `expr` is false.

7.14.2 Functions

The following functions are only available, when NDEBUG is not defined.

```
#include <circle/debug.h>
```

void **debug_hexdump**(const void *pStart, unsigned nBytes, const char *pSource = 0)
Writes a hexdump of nBytes, starting at pStart to the *System log*. pSource is used as prefix of the log messages ("debug" if omitted).

void **debug_stacktrace**(const uintptr *pStackPtr, const char *pSource = 0)
Writes a stack trace to the *System log*. This function tests 64 numbers starting at pStackPtr, if they point into the program code and logs them in this case.

void **debug_click**(unsigned nMask = DEBUG_CLICK_ALL)
This function can be used to debug events, which occur frequently, so that writing a log message would destroy the timing of the system. The function generates an audio click, which can be heard via the headphone jack of the Raspberry Pi. Frequent events generate a tone, very frequent events may generate a frequency, which is not hear-able. nMask can be DEBUG_CLICK_LEFT, DEBUG_CLICK_RIGHT or DEBUG_CLICK_ALL and selects the audio channel to be used. On some Raspberry Pi models these channels may be swapped.

Note: The macro DEBUG_CLICK must be defined, when you want to use debug_click(). The PWM audio device cannot be used in this case.

7.14.3 CExceptionHandler

```
#include <circle/exceptionhandler.h>
```

class **CExceptionHandler**
This class handles abort exceptions, which occur on different program errors. The exception handler displays a stack trace and logs some important register values. An instance of this class should be added to each more complex program, which includes a CLogger instance too. Usually it will be added as a member to CKernel. This class does not have methods, which can be called from application code.

7.14.4 CTracer

```
#include <circle/tracer.h>
```

class **CTracer**
This class can be used to trace the program execution, without changing the timing too much. The class maintains a ring buffer, which is filled with trace events and dumped later, when the execution of the critical program parts has been completed.

CTracer::**CTracer**(unsigned nDepth, boolean bStopIfFull)
Creates an instance of this class. nDepth is the size of the ring buffer in number of events. If bStopIfFull is TRUE, the tracing stops automatically, when the ring buffer is full. Otherwise a new event overwrites the oldest event.

void *CTracer*::**Start**(void)
Starts the tracing and the tracing clock. Arriving events will be written to the ring buffer now.

void *CTracer*::**Stop**(void)
Stops the tracing. If an event arrives afterwards, it is ignored.

void *CTracer*::**Event**(unsigned nID, unsigned nParam1 = 0, unsigned nParam2 = 0, unsigned nParam3 = 0, unsigned nParam4 = 0)
Sends an event to the tracer. Insert this into your program code, where something important happens to catch an issue. `nID` is any number, except 0, which is the stop event. `nParamN` is any parameter of the event. This method is not reentrant. You have to use a spin lock, if `Event()` may be called concurrently.

void *CTracer*::**Dump**(void)
Writes the entire tracing buffer to the *System log*. If the tracing was not stopped before, it is stopped automatically before the dump.

static *CTracer* **CTracer*::**Get**(void)
Returns a pointer to the `CTracer` object.

7.14.5 CLatencyTester

```
#include <circle/latencytester.h>
```

class **CLatencyTester**
This class can be used to measure the IRQ latency of the running code. The class continuously triggers an IRQ and measures the delay between the time, the IRQ was triggered and the time, the IRQ handler is called. This delay can be important for real-time applications. This is demonstrated in the sample program *40-irqlatency*.

Note: The class `CLatencyTester` blocks the system timer 1, which is used by the class `CUserTimer` too. You can use only one of both classes at a time.

CLatencyTester::**CLatencyTester**(*CInterruptSystem* *pInterruptSystem)
Creates a `CLatencyTester` object. `pInterruptSystem` is a pointer to the interrupt system object.

void *CLatencyTester*::**Start**(unsigned nSampleRateHZ)
Starts the measurement. `nSampleRateHZ` is the sample rate in Hz.

void *CLatencyTester*::**Stop**(void)
Stops the measurement.

unsigned *CLatencyTester*::**GetMin**(void) const
Returns the minimum IRQ latency in microseconds. Can be called, while the test is running.

unsigned *CLatencyTester*::**GetMax**(void) const
Returns the maximum IRQ latency in microseconds. Usually this is the most interesting value. Can be called, while the test is running.

unsigned *CLatencyTester*::**GetAvg**(void)
Returns the average IRQ latency in microseconds. Can be called, while the test is running. Please note that the accumulated IRQ latency may overrun after some time. This method will return 0xFFFFFFFFU then.

void *CLatencyTester*::**Dump**(void)
Writes the results to the *System log*.

CHAPTER

EIGHT

SUBSYSTEMS

This section describes the Circle subsystems. A subsystems is a group of classes, which implements services for a specific purpose, which are different from the *Basic system services* and are normally provided by its own library (see *Libraries*). Only those classes are discussed here, which are directly used by applications. All Circle classes are listed in doc/classes.txt.

The Circle project does not provide a single centralized C++ header file. Instead the header file(s), which must be included for a specific class, function or macro definition are specified in the related subsection.

8.1 Multitasking

Circle provides an optional cooperative non-preemptive scheduler, which allows to solve programming problems based on the process concept. Because in Circle there is only one flat address space with a one-to-one physical-to-virtual address mapping a process in Circle is similar to a thread. In Circle the name "task" is used instead.

Because the scheduler is optional, a Circle application can work without it. The scheduler was introduced to implement TCP/IP networking support, which required many threads of execution at the same time to be implemented even on the one-core Raspberry Pi models. Later porting the VCHIQ driver and HDMI sound support, the accelerated graphics support (Raspberry Pi 1-3 and Zero only) and Wireless LAN support had similar requirements. It can be useful to use the scheduler also for modeling complex user problems in Circle.

The scheduler library provides the following classes:

Class	Function
CScheduler	Cooperative non-preemptive scheduler
CTask	Representation of a thread of execution, a task
CMutex	Mutual exclusion (critical sections) across tasks
CSemaphore	Implements the well-known semaphore concept
CSynchronizationEvent	Synchronizes the execution of task(s) with an event

Important: In a multi-core environment (see *Multi-core support*) all tasks and the scheduler run on CPU core 0.

8.1.1 CScheduler

This class implements a cooperative non-preemptive scheduler, which controls which task runs at a time. Because the scheduler is non-preemptive, a running task has to explicitly release the CPU by sleeping, waiting for a synchronization object (mutex, semaphore, synchronization event) or by calling `CScheduler::Get()->Yield()` after a short time, so that the other tasks are able to run. This relatively simple scheduler implements the round-robin policy without task priorities (and without much overhead).

```
#include <circle/sched/scheduler.h>
```

class **CScheduler**

static boolean *CScheduler*::**IsActive**(void)
Returns `TRUE` if the scheduler is available in the system. The scheduler is optional in Circle.

static *CScheduler* **CScheduler*::**Get**(void)
Returns a pointer to the only scheduler object in the system. It must not be called, if the scheduler is not available.

CTask **CScheduler*::**GetCurrentTask**(void)
Returns a pointer to the `CTask` object of the currently running task.

CTask **CScheduler*::**GetTask**(const char *pTaskName)
Returns a pointer to the `CTask` object of the task with the name `pTaskName` or 0, if the task was not found.

boolean *CScheduler*::**IsValidTask**(*CTask* *pTask)
Returns `TRUE`, if `pTask` is referencing a CTask object of a currently known task.

void *CScheduler*::**Yield**(void)
Switch to the next task. The currently running task releases the CPU and the next task in round-robin order, which is not blocked, gets control.

Important: A task should call this from time to time, if it does longer calculations.

---

void *CScheduler*::**Sleep**(unsigned nSeconds)
The current task pauses execution for `nSeconds` seconds. The next ready task gets control.

void *CScheduler*::**MsSleep**(unsigned nMilliSeconds)
The current task pauses execution for `nMilliSeconds` milliseconds. The next ready task gets control.

void *CScheduler*::**usSleep**(unsigned nMicroSeconds)
The current task pauses execution for `nMicroSeconds` microseconds. The next ready task gets control.

void *CScheduler*::**SuspendNewTasks**(void)
Causes all new tasks to be created in a suspended state. You can achieve the same, if you set the parameter `bCreateSuspended` to `TRUE`, when calling `new` for a task. Nested calls to `SuspendNewTasks()` and `ResumeNewTasks()` are allowed.

void *CScheduler*::**ResumeNewTasks**(void)
Stops causing new tasks to be created in a suspended state and starts any tasks that were created suspended. Nested calls to `SuspendNewTasks()` and `ResumeNewTasks()` are allowed.

void *CScheduler*::**ListTasks**(*CDevice* *pTarget)
Writes a task listing to the device `pTarget`.

void *CScheduler*::**RegisterTaskSwitchHandler**(TSchedulerTaskHandler *pHandler)
`pHandler` is called on each task switch. This method is normally used by the Linux kernel driver and Pthreads emulation. The handler is called with a pointer to the `CTask` object of the task, which gets control now. The prototype of the handler is:

```
void TSchedulerTaskHandler (CTask *pTask);
```

void *CScheduler*::**RegisterTaskTerminationHandler**(TSchedulerTaskHandler *pHandler)
pHandler is called, when a task terminates. This method is normally used by the Linux kernel driver and Pthreads emulation. The handler is called with a pointer to the CTask object of the task, which terminates. See RegisterTaskSwitchHandler() for the prototype of the handler.

8.1.2 CTask

Derive this class, define the Run() method to implement your own task and call new on it to start it.

```
#include <circle/sched/task.h>
```

class **CTask**

CTask::**CTask**(unsigned nStackSize = TASK_STACK_SIZE, boolean bCreateSuspended = FALSE)
Creates a task. nStackSize is the stack size for this task. By default a new task is immediately ready to run and its Run() method can be called. If you have to do more initialization, before the task can run, set bCreateSuspended to TRUE. The task has to be started explicitly by calling Start() on it then.

virtual void *CTask*::**Run**(void)
Override this method to define the entry point for your own task. The task is automatically terminated, when Run() returns.

void *CTask*::**Start**(void)
Starts a task, that was created with bCreateSuspended = TRUE or restarts it after Suspend().

void *CTask*::**Suspend**(void)
Suspends a task from running, until Resume() is called for this task.

void *CTask*::**Resume**(void)
Alternative method to (re-)start a suspended task.

boolean *CTask*::**IsSuspended**(void) const
Returns TRUE, if the task is currently suspended from running.

void *CTask*::**Terminate**(void)
Terminates the execution of the task. This method can only be called by the task itself. The task terminates on return from Run() too.

void *CTask*::**WaitForTermination**(void)
Waits for the termination of the task. This method can only be called by an other task.

void *CTask*::**SetName**(const char *pName)
Sets the specific name pName for this task.

const char **CTask*::**GetName**(void) const
Returns a pointer to 0-terminated name string of this task. The default name of a task is constructed from the address of its task object (e.g. "@84abc0"). The main application task has the name "main".

void *CTask*::**SetUserData**(void *pData, unsigned nSlot)
Sets a user pointer for this task. If you have to associate some data with a task, you can call this method with nSlot = TASK_USER_DATA_USER. pData is any user pointer to be set.

void **CTask*::**GetUserData**(unsigned nSlot)
Returns a user pointer for this task, which has previously been set using SetUserData(). nSlot must be TASK_USER_DATA_USER for application usage.

8.1.3 CMutex

Provides a method to provide mutual exclusion (critical sections) across tasks.

```
#include <circle/sched/mutex.h>
```

class **CMutex**

void *CMutex*::**Acquire**(void)
: Acquires the mutex. The current task blocks, if another task already acquired the mutex. The mutex can be acquired multiple times by the same task.

void *CMutex*::**Release**(void)
: Releases the mutex. Another task, which was waiting for the mutex to acquire, will be waken.

8.1.4 CSemaphore

Implements the well-known semaphore synchronization concept, which was initially defined by Dijkstra. The class maintains a non-negative counter, which is decremented with the `Down()` operation. When this is not possible, because the counter is already zero, the calling task waits, until the counter is incremented again. This is possible with the `Up()` operation. Semaphores can be used to control the access to a limited number of resources.

```
#include <circle/sched/semaphore.h>
```

class **CSemaphore**

CSemaphore::**CSemaphore**(unsigned nInitialCount = 1)
: Creates a semaphore. `nInitialCount` is the initial count of the semaphore.

unsigned *CSemaphore*::**GetState**(void) const
: Returns the current count of the semaphore.

void *CSemaphore*::**Down**(void)
: Decrements the semaphore count. Blocks the calling task, if the count is already zero.

void *CSemaphore*::**Up**(void)
: Increments the semaphore count. Wakes another waiting task, if the count was zero. Can be called from interrupt context.

boolean *CSemaphore*::**TryDown**(void)
: Tries to decrement the semaphore count. Returns `TRUE` on success or `FALSE`, if the count is zero.

8.1.5 CSynchronizationEvent

Provides a method to synchronize the execution of tasks with an event. The event can be set or cleared. If a task is waiting for the event, it is blocked, when the event is cleared (unset) and will continue execution, when the event is set again. Multiple tasks can wait for the event at the same time.

```
#include <circle/sched/synchronizationevent.h>
```

class **CSynchronizationEvent**

CSynchronizationEvent::**CSynchronizationEvent**(boolean bState = FALSE)
: Creates the synchronization event. `bState` is the initial state of the event (default cleared).

boolean *CSynchronizationEvent*::**GetState**(void)
Returns the current state for the synchronization event.

void *CSynchronizationEvent*::**Clear**(void)
Clears the synchronization event.

void *CSynchronizationEvent*::**Set**(void)
Sets the synchronization event. Wakes all tasks currently waiting for the event. Can be called from interrupt context.

void *CSynchronizationEvent*::**Wait**(void)
Blocks the calling task, if the synchronization event is cleared. The task will wake up, when the event is set later. Multiple tasks can wait for the event to be set.

boolean *CSynchronizationEvent*::**WaitWithTimeout**(unsigned nMicroSeconds)
Blocks the calling task for `nMicroSeconds` microseconds, if the synchronization event is cleared. The task will wake up, when the event is set later. Multiple tasks can wait for the event to be set. This method returns `TRUE`, if `nMicroSeconds` microseconds have elapsed, before the event has been set. To determine, what caused the method to return, use `GetState()` to see, if the event has been set. It is possible to have timed out and the event is set anyway.

8.2 USB

The USB (Universal Serial Bus) subsystem provides services and device drivers, which support the access to USB 2.0 and USB 3.0 (on the Raspberry Pi 4 only) devices. Essentially, this concerns drivers for the DWHCI OTG USB controller of the Raspberry Pi 1-3 and Zero (host mode only) and the xHCI USB controller(s) of the Raspberry Pi 4 (400 and Compute Module 4 too), USB device class drivers, some vendor specific USB device drivers and support classes for all these drivers.

Most of the operations in this subsystem are hidden from applications behind device driver interfaces, which will be described in the *Devices* section. An application, which uses the USB, has especially to deal with the initialization of the USB support at system startup and optionally with detecting newly attached USB devices, while the system is running (USB plug-and-play). This section is limited to these topics.

Please read the file doc/usb-plug-and-play.txt for general information about the (optional) USB plug-and-play support in Circle.

8.2.1 CUSBHCIDevice

```
#include <circle/usb/usbhcidevice.h>
```

class **CUSBHCIDevice** : public *CUSBHostController*
This class is the base of the USB support in a Circle application. To use USB, you should create a member of this class in the `CKernel` class of your application.

Note: Actually there is not really a class `CUSBHCIDevice` available in Circle. Instead, two classes `CDWHCIDevice` and `CXHCIDevice` (both derived from `CUSBHostController`) exist for the respective USB host controllers of the target Raspberry Pi model, and `CUSBHCIDevice` is only an alias for these class names, defined as macro. To ensure, that an application can be built for each Raspberry Pi model, you should use the name `CUSBHCIDevice` only.

Some methods available via `CUSBHCIDevice` are defined in its base class *CUSBHostController* and can be called using a pointer to a `CUSBHostController` object too.

CUSBHCIDevice::**CUSBHCIDevice**(*CInterruptSystem* *pInterruptSystem, *CTimer* *pTimer, boolean bPlugAndPlay = FALSE)

Creates an instance of this class. `pInterruptSystem` is a pointer to the interrupt system object and `pTimer` a pointer to the system timer object. `bPlugAndPlay` must be set to `TRUE` to enable the USB plug-and-play support. This is optional and requires further support by the application.

boolean *CUSBHCIDevice*::**Initialize**(boolean bScanDevices = TRUE)

Initializes the USB subsystem. Normally this includes a bus scan and the initialization of all attached USB devices, which takes some time. To speed-up the USB initialization, `bScanDevices` can be set to `FALSE`, if USB plug-and-play was enabled in the constructor of this class (`bPlugAndPlay = TRUE`). The device initialization will be deferred to a later call of `UpdatePlugAndPlay()` then.

void *CUSBHCIDevice*::**ReScanDevices**(void)

This method can be invoked to re-scan the USB for newly attached devices, in case USB plug-and-play support has not been enabled, when calling the constructor of this class (`bPlugAndPlay = FALSE`).

8.2.2 CUSBHostController

```
#include <circle/usb/usbhostcontroller.h>
```

class **CUSBHostController**

This is the base class of `CDWHCIDevice` and `CXHCIDevice` (aka `CUSBHCIDevice`). The following methods can be called for an instance of these classes too.

static boolean *CUSBHostController*::**IsPlugAndPlay**(void)

Returns `TRUE`, if USB plug-and-play is supported by the USB subsystem.

boolean *CUSBHostController*::**UpdatePlugAndPlay**(void)

If USB plug-and-play is enabled, this method must be called continuously from `TASK_LEVEL`, so that the internal USB device tree can be updated, if new devices have been attached or devices have been removed from the USB. Returns `TRUE`, if the USB device tree might have been changed. The application should test for the existence of devices, which it supports, by invoking `CDeviceNameService::GetDevice()` then. `UpdatePlugAndPlay()` always returns `TRUE` on its first call.

static boolean *CUSBHostController*::**IsActive**(void)

Returns `TRUE`, if the USB subsystem is available.

static *CUSBHostController* **CUSBHostController*::**Get**(void)

Returns a pointer to the only instance of `CUSBHostController` (aka `CUSBHCIDevice`) in the system.

8.3 Filesystems

Circle provides two different implementations for FAT filesystem support:

- A native filesystem subsystem with FAT16 and FAT32 support using C++ classes, which has several limitations: short filenames (8.3) only, access to the root directory only, sequential file access only, subset of common file operations only, relatively slow
- A port of FatFs - Generic FAT Filesystem Module (by ChaN), written in C, but with full FAT filesystem support and much faster

8.3.1 CFATFileSystem

```
#include <circle/fs/fat/fatfs.h>
```

class **CFATFileSystem**
This class provides the API of the native FAT filesystem support in Circle, which has several limitations (see above). If you want to use this variant, you should instantiate this class as a member of the class `CKernel`.

int *CFATFileSystem*::**Mount**(*CDevice* *pPartition)
Mounts the block device `pPartition` as FAT filesystem. Returns non-zero on success. A pointer to the block device can be requested using `CDeviceNameService::GetDevice()`. This method is usually invoked with the partition name `"emmc1-1"` (first partition of the SD card) or `"umsd1-1"` (first partition of an USB mass-storage device, e.g. USB flash drive) for this purpose.

void *CFATFileSystem*::**UnMount**(void)
Un-mounts the filesystem.

void *CFATFileSystem*::**Synchronize**(void)
Flushes the buffer cache, without un-mounting the filesystem.

unsigned *CFATFileSystem*::**RootFindFirst**(TDirentry *pEntry, TFindCurrentEntry *pCurrentEntry)
Finds the first file entry in the root directory and returns non-zero, if it was found. `pEntry` is a pointer to a buffer, which receives the information about the found file. `pCurrentEntry` points to the current entry variable, which must be maintained, until the directory scan has been completed. `TDirentry` is defined as follows:

```
#define FS_TITLE_LEN            12      // length of a file name (may include a dot)

struct TDirentry
{
        char            chTitle[FS_TITLE_LEN+1];        // 0-terminated
        unsigned        nSize;                          // number of bytes
        unsigned        nAttributes;
#define FS_ATTRIB_NONE          0x00    // no attribute set
#define FS_ATTRIB_SYSTEM        0x01    // HIDDEN attribute set
#define FS_ATTRIB_EXECUTABLE    0x02    // file is executable (always set for FAT)
};
```

unsigned *CFATFileSystem*::**RootFindNext**(TDirentry *pEntry, TFindCurrentEntry *pCurrentEntry)
Finds the next file entry in the root directory and returns non-zero, if it was found, or zero, if there are no more entries. `pEntry` is a pointer to a buffer, which receives the information about the found file. `pCurrentEntry` points to the current entry variable, which must be maintained, until the directory scan has been completed. See `RootFindFirst()` for the definition of TDirentry.

unsigned *CFATFileSystem*::**FileOpen**(const char *pTitle)
Opens the file with the filename `pTitle` (8.3 name without path, may include a dot) for read. Returns the file handle or zero on failure.

unsigned *CFATFileSystem*::**FileCreate**(const char *pTitle)
Creates a new file with the filename `pTitle` (8.3 name without path, may include a dot) for write. Returns the file handle or zero on failure (e.g. read-only file with this name exists).

> **Warning:** This method truncates the file, if it already exists with the filename `pTitle`.

unsigned *CFATFileSystem*::**FileClose**(unsigned hFile)
Closes the file with the file handle `hFile`. This handle has been returned by a previous call to `FileOpen()` or `FileCreate()`. Returns non-zero on success.

unsigned *CFATFileSystem*::**FileRead**(unsigned hFile, void *pBuffer, unsigned nCount)
Reads sequentially up to `nCount` bytes from the file with file handle `hFile` into `pBuffer`. Returns the number of bytes read, zero when the end of file has been reached, or FS_ERROR on general failure (e.g. invalid parameter).

unsigned *CFATFileSystem*::**FileWrite**(unsigned hFile, const void *pBuffer, unsigned nCount)
Writes sequentially `nCount` bytes from `pBuffer` to the file with file handle `hFile`. Returns the number of bytes written, or FS_ERROR on general failure (e.g. invalid parameter).

int *CFATFileSystem*::**FileDelete**(const char *pTitle)
Deletes the file with the name `pTitle` from the root directory. Returns a positive value on success, zero, if the file was not found, or a negative value, if the file has the read-only attribute.

8.3.2 FatFs library

The FatFs - Generic FAT Filesystem Module (by ChaN) has been ported to Circle, to provide a full function support for the FAT filesystem. The related files can be found in the subdirectory *addon/fatfs*. The associated sample program demonstrates some basic features of FatFs. Please see the subsection "Application Interface" on this website for a description of the different functions of this library.

The Circle port of FatFs supports the following volume ID strings for logical drives:

ID	Drive	Partition	Device
SD:	0:	first FAT partition	SD card
USB:	1:	first FAT partition	first USB mass-storage device
USB2:	2:	first FAT partition	second USB mass-storage device
USB3:	3:	first FAT partition	third USB mass-storage device

Important: FatFs may support the exFAT filesystems too. This support has been disabled in Circle for legal reasons. You have to read the subsection "exFAT Filesystem" on the page FatFs Module Application Note first, if you want to use exFAT support! This may require a license fee.

8.4 TCP/IP networking

This section describes the TCP/IP networking support in Circle. This covers initialization and configuration, the socket API, the available upper layer protocol clients and servers and a few utility classes. The TCP/IP networking support requires the scheduler in the system (see *Multitasking*).

The following sample programs demonstrate TCP/IP networking features:

Sample	Description
18-ntptime	Setting the system time from a NTP server
20-tcpsimple	TCP echo server
21-webserver	Simple HTTP webserver
31-webclient	Simple HTTP client (only for reference)
33-syslog	Send log messages to an UDP syslog server
35-mqttclient	MQTT client
38-bootloader	HTTP- and TFTP-based bootloader

8.4.1 CNetSubSystem

```
#include <circle/net/netsubsystem.h>
```

class **CNetSubSystem**

This class represents the TCP/IP support in Circle. There can be only one instance of this class.

CNetSubSystem::**CNetSubSystem**(const u8 *pIPAddress = 0, const u8 *pNetMask = 0, const u8 *pDefaultGateway = 0, const u8 *pDNSServer = 0, const char *pHostname = "raspberrypi", TNetDeviceType DeviceType = NetDeviceTypeEthernet)

Creates the `CNetSubSystem` instance. The parameters `pIPAddress`, `pNetMask`, `pDefaultGateway` and `pDNSServer` point to 4-byte arrays, which define the network configuration (e.g. IP address {192, 168, 0, 42}). Set all these pointers to 0 to enable the dynamic network configuration using DHCP instead. `pHostname` specifies the host name, which is sent to the DHCP server (0 to disable). `DeviceType` can be `NetDeviceTypeEthernet` (default) or `NetDeviceTypeWLAN`.

Note: Setting `DeviceType` to `NetDeviceTypeWLAN` is not enough to access a WLAN. Instead you have to instantiate and initialize the classes `CBcm4343Device` (WLAN hardware driver), `CNetSubSystem` and `CWPASupplicant` (support task for secure WLAN access) in this order. Please see the subsection *WLAN support* and the WLAN sample for details!

---

boolean *CNetSubSystem*::**Initialize**(boolean bWaitForActivate = TRUE)

Initializes the network subsystem. Usually this method returns after the network configuration has been assigned, if DHCP is enabled. This requires that the DHCP server can be reached and takes some time. If you want to speedup network initialization, you can set the parameter `bWaitForActivate` to `FALSE`. Then this method will return immediately after initialization, but you have to test on your own, if the network is available using the method `IsRunning()`, before accessing the network.

boolean *CNetSubSystem*::**IsRunning**(void) const

Returns `TRUE`, when is TCP/IP network is available and configured. If DHCP is enabled, this means that an IP address is already bound.

CNetConfig **CNetSubSystem*::**GetConfig**(void)

Returns a pointer to the network configuration, which is saved in an instance of the class `CNetConfig`. This is usually used to inform the user about the dynamically assigned configuration. You should not try to manipulate the configuration using this pointer.

static *CNetSubSystem* **CNetSubSystem*::**Get**(void)

Returns a pointer to the instance of `CNetSubSystem`.

8.4.2 CNetConfig

```
#include <circle/net/netconfig.h>
```

class **CNetConfig**

An instance of this class holds the configuration of the TCP/IP networking subsystem. A pointer to this instance can be requested using `CNetSubSystem::GetConfig()`. The following methods can be used to get the different configuration items.

boolean *CNetConfig*::**IsDHCPUsed**(void) const

Returns `TRUE` if the network is configured dynamically using DHCP.

const *CIPAddress* **CNetConfig*::**GetIPAddress**(void) const

Returns our own IP address.

const u8 *CNetConfig::**GetNetMask**(void) const
Returns the net mask of the local network, we are connected to.

const CIPAddress *CNetConfig::**GetDefaultGateway**(void) const
Returns the IP address of the default gateway into the Internet.

const CIPAddress *CNetConfig::**GetDNSServer**(void) const
Returns the IP address of the Domain Name Service server.

const CIPAddress *CNetConfig::**GetBroadcastAddress**(void) const
Returns the (directed) broadcast address, which is valid in the local network, we are connected to.

8.4.3 CSocket

```
#include <circle/net/socket.h>
#include <circle/net/in.h>              // for IPPROTO_*, MSG_DONTWAIT
#include <circle/netdevice.h>           // for FRAME_BUFFER_SIZE
```

class **CSocket** : public CNetSocket
This class forms the API for TCP/IP network access in Circle.

Note: Port numbers at the Circle socket API are in host byte order. This means you do not need to swap the byte order to network order, when you specify a little endian number to an API function.

Operations can be blocking or non-blocking. Blocking operations wait for the completion, before the function returns. Non-blocking operations return immediately, which means that you have to ensure on your own, that the system is not congested, e.g. if sending much data.

CSocket::**CSocket**(CNetSubSystem *pNetSubSystem, int nProtocol)
Creates a `CSocket` object, which represents one TCP/IP connection in Circle. `pNetSubSystem` is a pointer to the network subsystem. `nProtocol` can be `IPPROTO_TCP` or `IPPROTO_UDP`.

CSocket::**~CSocket**(void)
Destroys a `CSocket` object and terminates an active connection.

int CSocket::**Bind**(u16 usOwnPort)
Binds the port number `usOwnPort` to this socket. Returns 0 on success or < 0 on error.

int CSocket::**Connect**(CIPAddress &rForeignIP, u16 usForeignPort)
Connects to a foreign host/port (TCP) or setup a foreign host/port address (UDP). `rForeignIP` is the IP address of the host to be connected. `usForeignPort` is the number of the port to be connected. Returns 0 on success or < 0 on error.

int CSocket::**Listen**(unsigned nBackLog = 4)
Listens for incoming connections (TCP only). You must call `Bind()` before. `nBackLog` is the maximum number of simultaneous connections, which may be accepted in a row before `Accept()` is called (up to 32). Returns 0 on success or < 0 on error.

CSocket *CSocket::**Accept**(CIPAddress *pForeignIP, u16 *pForeignPort)
Accepts an incoming connection (TCP only). You must call `Listen()` before. `pForeignIP` points to a `CIPAddress` object, which receives the IP address of the remote host. The remote port number will be returned in `*pForeignPort`. Returns a newly created socket to be used to communicate with the remote host, or 0 on error.

int CSocket::**Send**(const void *pBuffer, unsigned nLength, int nFlags)
Sends a message to a remote host. `pBuffer` is a pointer to the message and `nLength` is its length in bytes. `nFlags` can be `MSG_DONTWAIT` (non-blocking operation) or 0 (blocking operation). Returns the length of the

sent message or < 0 on error.

int *CSocket*::**Receive**(void *pBuffer, unsigned nLength, int nFlags)
Receives a message from a remote host. `pBuffer` is a pointer to the message buffer and `nLength` is its size in bytes. `nLength` should be at least `FRAME_BUFFER_SIZE`, otherwise data may get lost. `nFlags` can be `MSG_DONTWAIT` (non-blocking operation) or 0 (blocking operation). Returns the length of received message, which is 0 with `MSG_DONTWAIT` if no message is available, or < 0 on error.

int *CSocket*::**SendTo**(const void *pBuffer, unsigned nLength, int nFlags, *CIPAddress* &rForeignIP, u16 nForeignPort)
Sends a message to a specific remote host. `pBuffer` is a pointer to the message and `nLength` is its length in bytes. `nFlags` can be `MSG_DONTWAIT` (non-blocking operation) or 0 (blocking operation). `rForeignIP` is the IP address of the host to be sent to (ignored on TCP socket). `nForeignPort` is the number of the port to be sent to (ignored on TCP socket). Returns the length of the sent message or < 0 on error.

int *CSocket*::**ReceiveFrom**(void *pBuffer, unsigned nLength, int nFlags, *CIPAddress* *pForeignIP, u16 *pForeignPort)
Receives a message from a remote host, returns host/port of remote host. `pBuffer` is a pointer to the message buffer and `nLength` is its size in bytes. `nLength` should be at least `FRAME_BUFFER_SIZE`, otherwise data may get lost. `nFlags` can be `MSG_DONTWAIT` (non-blocking operation) or 0 (blocking operation). `pForeignIP` is a pointer to a `CIPAddress` object, which receives the IP address of the host, which has sent the message. The number of the port from which the message has been sent will be returned in `*pForeignPort`. Returns the length of the received message, which is 0 with `MSG_DONTWAIT` if no message is available, or < 0 on error.

int *CSocket*::**SetOptionBroadcast**(boolean bAllowed)
`bAllowed` specifies weather sending and receiving broadcast messages is allowed on this socket (default `FALSE`). Call this with `bAllowed = TRUE` after `Bind()` or `Connect()` to be able to send and/or receive broadcast messages (ignored on TCP socket). Returns 0 on success or < 0 on error.

const u8 **CSocket*::**GetForeignIP**(void) const
Returns a pointer to the IP address of the connected remote host (4 bytes) or 0, if the socket is not connected.

8.4.4 Clients

CDNSClient

```
#include <circle/net/dnsclient.h>
```

class **CDNSClient**
This class supports the resolve of an Internet domain host name to an IP address.

CDNSClient::**CDNSClient**(*CNetSubSystem* *pNetSubSystem)
Creates a `CDNSClient` object. `pNetSubSystem` is a pointer to the network subsystem.

boolean *CDNSClient*::**Resolve**(const char *pHostname, *CIPAddress* *pIPAddress)
Resolves the host name `pHostname` to an IP address, returned in `*pIPAddress`. `pHostname` can be a dotted IP address string (e.g. "192.168.0.42") too, which will be converted. Returns `TRUE` on success.

CHTTPClient

```
#include <circle/net/httpclient.h>
#include <circle/net/http.h>            // for THTTPStatus
```

class **CHTTPClient**
This class can be used to generate requests to a HTTP server.

Note: In the Internet of today there are only a few webservers any more, which provide plain HTTP access. For HTTPS (HTTP over TLS) access with Circle you can use the circle-stdlib project, which includes Circle as a submodule.

CHTTPClient::**CHTTPClient**(*CNetSubSystem* *pNetSubSystem, *CIPAddress* &rServerIP, u16 usServerPort = HTTP_PORT, const char *pServerName = 0)
Creates a `CHTTPClient` object. `pNetSubSystem` is a pointer to the network subsystem. `rServerIP` is the IP address of the server and `usServerPort` the server port to connect. `pServerName` is the host name of the server, which is required for the access to virtual servers (multiple websites with different host names, hosted on the same server).

THTTPStatus *CHTTPClient*::**Get**(const char *pPath, u8 *pBuffer, unsigned *pLength)
Sends a GET request to the server. `pPath` is the absolute path of the requested document, optionally with appended parameters:

```
/path/filename.ext[?name=value[&name=value...]]
```

The received content will be returned in `pBuffer`. `*pLength` is the buffer size in bytes on input and the received content length on output. Returns the HTTP status (`HTTPOK` on success).

THTTPStatus *CHTTPClient*::**Post**(const char *pPath, u8 *pBuffer, unsigned *pLength, const char *pFormData)
Sends a POST request to the server. `pPath` is the absolute path of the requested document, optionally with appended parameters:

```
/path/filename.ext[?name=value[&name=value...]]
```

The received content will be returned in `pBuffer`. `*pLength` is the buffer size in bytes on input and the received content length on output. `pFormData` are the posted parameters in this format:

```
name=value[&name=value...]
```

Returns the HTTP status (`HTTPOK` on success).

CNTPClient

```
#include <circle/net/ntpclient.h>
```

class **CNTPClient**
This class can be used to request the current time from a Network Time Protocol server.

CNTPClient::**CNTPClient**(*CNetSubSystem* *pNetSubSystem)
Creates a `CNTPClient` object. `pNetSubSystem` is a pointer to the network subsystem.

unsigned *CNTPClient*::**GetTime**(*CIPAddress* &rServerIP)
Requests the current time from a NTP server. `rServerIP` is the IP address from the NTP server, which can be resolved using the class `CDNSClient`. Returns the current time in seconds since 1970-01-01 00:00:00 UTC, or 0 on error.

CNTPDaemon

```
#include <circle/net/ntpdaemon.h>
```

class **CNTPDaemon** : public *CTask*

This class is a background task, which continuously (all 15 minutes) updates the Circle system time from a NTP server. It uses the class `CNTPClient`.

CNTPDaemon`::`**CNTPDaemon**(const char *pNTPServer, *CNetSubSystem* *pNetSubSystem)

Creates the `CNTPDaemon` task. `pNTPServer` is the host name of the NTP server (e.g. "pool.ntp.org"). `pNetSubSystem` is a pointer to the network subsystem. This object must be created using the `new` operator.

CMQTTClient

```
#include <circle/net/mqttclient.h>
```

class **CMQTTClient** : public *CTask*

This class is a client for the MQTT protocol, according to the MQTT v3.1.1 specification. It is implemented as a task. To use this class, you have to derive a user defined class from `CMQTTClient` and override its virtual methods. The task must be created with the `new` operator.

Warning: This implementation does not support multi-byte-characters in strings.

CMQTTClient`::`**CMQTTClient**(*CNetSubSystem* *pNetSubSystem, size_t nMaxPacketSize = 1024, size_t nMaxPacketsQueued = 4, size_t nMaxTopicSize = 256)

Creates a `CMQTTClient` task. `pNetSubSystem` is a pointer to the network subsystem. `nMaxPacketSize` is the maximum allowed size of a MQTT packet sent or received (topic size + payload size + a few bytes protocol overhead). `nMaxPacketsQueued` is the maximum number of MQTT packets queue-able on receive. If processing a received packet takes longer, further packets have to be queued. `nMaxTopicSize` is the maximum allowed size of a received topic string.

boolean *CMQTTClient*`::`**IsConnected**(void) const

Returns `TRUE` if an active connection to the MQTT broker exists.

void *CMQTTClient*`::`**Connect**(const char *pHost, u16 usPort = MQTT_PORT, const char *pClientIdentifier = 0, const char *pUsername = 0, const char *pPassword = 0, u16 usKeepAliveSeconds = 60, boolean bCleanSession = TRUE, const char *pWillTopic = 0, u8 uchWillQoS = 0, boolean bWillRetain = FALSE, const u8 *pWillPayload = 0, size_t nWillPayloadLength = 0)

Establishes a connection to the MQTT broker `pHost` (host name or IP address as a dotted string). `usPort` is the port number of the MQTT broker service (default 1883). `pClientIdentifier` is the identifier string of this client (if 0 set internally to `raspiNNNNNNNNNNNNNNN`, N = hex digits of the serial number). `pUsername` is the user name for authorization (0 if not required). `pPassword` is the password for authorization (0 if not required). `usKeepAliveSeconds` is the duration of the keep alive period in seconds (default 60). `bCleanSession` specifies, if this should be a clean MQTT session. (default TRUE).

`pWillTopic` is the topic string for the last will message (no last will message if 0). `uchWillQoS` is the QoS setting for last will message (default unused). `bWillRetain` is the retain parameter for last will message (default unused). `pWillPayload` is a pointer to the last will message payload (default unused). `nWillPayloadLength` is the length of the last will message payload (default unused).

void *CMQTTClient*`::`**Disconnect**(boolean bForce = FALSE)

Closes the connection to a MQTT broker. `bForce` forces a TCP disconnect only and does not send a MQTT DISCONNECT packet.

void *CMQTTClient*::**Subscribe**(const char *pTopic, u8 uchQoS = MQTT_QOS2)
Subscribes to the MQTT topic `pTopic` (may include wildchars). `uchQoS` is the maximum QoS value for receiving messages with this topic (default QoS 2).

void *CMQTTClient*::**Unsubscribe**(const char *pTopic)
Unsubscribes from the MQTT topic `pTopic`.

void *CMQTTClient*::**Publish**(const char *pTopic, const u8 *pPayload = 0, size_t nPayloadLength = 0, u8 uchQoS = MQTT_QOS1, boolean bRetain = FALSE)
Publishes the MQTT topic `pTopic`. `pPayload` is a pointer to the message payload (default unused). `nPayloadLength` is the length of the message payload (default 0). `uchQoS` is the QoS value for sending the PUBLISH message (default QoS 1). `bRetain` is the retain parameter for the message (default FALSE).

virtual void *CMQTTClient*::**OnConnect**(boolean bSessionPresent)
This is a callback entered when the connection to the MQTT broker has been established. `bSessionPresent` specifies, if a session was already present on the server for this client.

virtual void *CMQTTClient*::**OnDisconnect**(TMQTTDisconnectReason Reason)
This is a callback entered when the connection to the MQTT broker has been closed. `Reason` is the reason for closing the connection, which can be:

```
enum TMQTTDisconnectReason
{
        MQTTDisconnectFromApplication                   = 0,

        // CONNECT errors
        MQTTDisconnectUnacceptableProtocolVersion       = 1,
        MQTTDisconnectIdentifierRejected                = 2,
        MQTTDisconnectServerUnavailable                 = 3,
        MQTTDisconnectBadUsernameOrPassword             = 4,
        MQTTDisconnectNotAuthorized                     = 5,

        // additional errors
        MQTTDisconnectDNSError,
        MQTTDisconnectConnectFailed,
        MQTTDisconnectFromPeer,
        MQTTDisconnectInvalidPacket,
        MQTTDisconnectPacketIdentifier,
        MQTTDisconnectSubscribeError,
        MQTTDisconnectSendFailed,
        MQTTDisconnectPingFailed,
        MQTTDisconnectNotSupported,
        MQTTDisconnectInsufficientResources,

        MQTTDisconnectUnknown
};
```

virtual void *CMQTTClient*::**OnMessage**(const char *pTopic, const u8 *pPayload, size_t nPayloadLength, boolean bRetain)
This is a callback entered when a PUBLISH message has been received for a subscribed topic. `pTopic` is the topic of the received message. `pPayload` is a pointer to the payload of the received message. `nPayloadLength` is the length of the payload of the received message. `bRetain` is the retain parameter of the received message.

virtual void *CMQTTClient*::**OnLoop**(void)
This is a callback regularly entered from the MQTT client task.

CSysLogDaemon

```
#include <circle/net/syslogdaemon.h>
```

class **CSysLogDaemon** : public *CTask*

This class is a background task, which sends the messages from the *System log* to a RFC5424/RFC5426 syslog server via UDP.

CSysLogDaemon::**CSysLogDaemon**(*CNetSubSystem* *pNetSubSystem, const *CIPAddress* &rServerIP, u16 usServerPort = SYSLOG_PORT)

Creates the `CSysLogDaemon` task. `pNetSubSystem` is a pointer to the network subsystem. `rServerIP` is the IP address of the syslog server. `usServerPort` is the port number of the syslog server (default 514). This object must be created using the `new` operator.

8.4.5 Servers

CHTTPDaemon

```
#include <circle/net/httpdaemon.h>
#include <circle/net/http.h>            // for THTTPStatus
```

class **CHTTPDaemon** : public *CTask*

This class implements a simple HTTP server as a task. You have to derive a user class from it, override the virtual methods and create it using the `new` operator to start it.

Note: This class uses a listener/worker model. The initially created task listens for incoming requests (listener) and spawns a child task (worker), which processes the request and terminates afterwards.

CHTTPDaemon::**CHTTPDaemon**(*CNetSubSystem* *pNetSubSystem, *CSocket* *pSocket = 0, unsigned nMaxContentSize = 0, u16 nPort = HTTP_PORT, unsigned nMaxMultipartSize = 0)

Creates the `CHTTPDaemon` task. `pNetSubSystem` is a pointer to the network subsystem. `pSocket` is 0 for first created instance (listener). `nMaxContentSize` is the buffer size for the content of the created worker tasks. Set this parameter to the maximum length in bytes of a webpage, which is generated by your server. `nPort` is the port number to listen on (default 80). `nMaxMultipartSize` is the buffer size for received multipart form data. If your server receives requests, which include multipart form data, this parameter must be set to the maximum length of this data, which you want to process.

virtual *CHTTPDaemon* **CHTTPDaemon*::**CreateWorker**(*CNetSubSystem* *pNetSubSystem, *CSocket* *pSocket) = 0

Creates a worker instance of your derived webserver class. `pNetSubSystem` is a pointer to the network subsystem. `pSocket` is the socket that manages the incoming connection. Both parameters have to be handed over to the constructor of your derived webserver class, to be passed to `CHTTPDaemon::CHTTPDaemon`. See this example:

Listing 1: mywebserver.h

```
class CMyWebServer : public CHTTPDaemon
{
public:
        CMyWebServer (CNetSubSystem *pNetSubSystem,
                      CActLED       *pActLED,      // some user data
                      CSocket       *pSocket = 0); // 0 for first instance

        CHTTPDaemon *CreateWorker (CNetSubSystem *pNetSubSystem,
```

(continues on next page)

(continued from previous page)

```
                                    CSocket       *pSocket);

        ...
};
```

Listing 2: mywebserver.cpp

```
#define MAX_CONTENT_SIZE        4000    // maximum content size of your pages

CMyWebServer::CMyWebServer (CNetSubSystem *pNetSubSystem,
                            CActLED       *pActLED,
                            CSocket       *pSocket)
:       CHTTPDaemon (pNetSubSystem, pSocket, MAX_CONTENT_SIZE),
        m_pActLED (pActLED)
{
}

CHTTPDaemon *CMyWebServer::CreateWorker (CNetSubSystem *pNetSubSystem,
                                         CSocket       *pSocket)
{
        return new CMyWebServer (pNetSubSystem, m_pActLED, pSocket);
}
```

virtual THTTPStatus *CHTTPDaemon*::**GetContent**(const char *pPath, const char *pParams, const char *pFormData, u8 *pBuffer, unsigned *pLength, const char **ppContentType) = 0

Define this method to provide your own content. `pPath` is the path of the file to be sent (e.g. "/index.html", can be "/" too). `pParams` are the GET parameters ("" for none). `pFormData` contains the parameters from the form data from POST ("" for none). Copy your content to `pBuffer`. `*pLength` is the buffer size in bytes on input and the content length on output. `*ppContentType` must be set to the MIME type, if it is not "text/html". This method has to return the HTTP status (`HTTPOK` on success).

virtual void *CHTTPDaemon*::**WriteAccessLog**(const *CIPAddress* &rRemoteIP, THTTPRequestMethod RequestMethod, const char *pRequestURI, THTTPStatus Status, unsigned nContentLength)

Overwrite this method to implement your own access logging. `rRemoteIP` is the IP address of the client. `RequestMethod` is the method of the request and `pRequestURI` its URI. `Status` and `nContentLength` specify the returned HTTP status number and the length of the sent content. The default implementation of this method writes a message to the *System log*.

boolean *CHTTPDaemon*::**GetMultipartFormPart**(const char **ppHeader, const u8 **ppData, unsigned *pLength)

This method can be called from `GetContent()` and returns the next part of multipart form data (`TRUE` if available). This data is not available after returning from `GetContent()` any more. `*ppHeader` returns a pointer to the part header. `*ppData` returns a pointer to part data. `*pLength` returns the part data length.

CTFTPDaemon

```
#include <circle/net/tftpdaemon.h>
```

class **CTFTPDaemon** : public *CTask*
This class provides a server task for the TFTP protocol. You have to implement the pure virtual methods in a derived class, start the task with the `new` operator and will be able to receive and handle TFTP requests. This server can handle only one connection at a time, and works in binary mode only. The TFTP fileserver sample demonstrates the usage of this class.

CTFTPDaemon`::`**CTFTPDaemon**`(`*CNetSubSystem* *pNetSubSystem`)`
Creates the `CTFTPDaemon` task. `pNetSubSystem` is a pointer to the network subsystem.

virtual boolean *CTFTPDaemon*`::`**FileOpen**`(`const char *pFileName`) = 0`
Virtual method entered to open a file for read to be sent via TFTP. `pFileName` is the file name sent by the client. Returns `TRUE` on success.

virtual boolean *CTFTPDaemon*`::`**FileCreate**`(`const char *pFileName`) = 0`
Virtual method entered to create a file for write to be received via TFTP. `pFileName` is the file name sent by the client. Returns `TRUE` on success.

virtual boolean *CTFTPDaemon*`::`**FileClose**`(`void`) = 0`
Virtual method entered to close the currently open file. Returns `TRUE` on success.

virtual int *CTFTPDaemon*`::`**FileRead**`(`void *pBuffer, unsigned nCount`) = 0`
Virtual method entered to read `nCount` bytes from the currently open file into `pBuffer`. Returns the number of bytes read, or < 0 on error.

virtual int *CTFTPDaemon*`::`**FileWrite**`(`const void *pBuffer, unsigned nCount`) = 0`
Virtual method entered to write `nCount` bytes from `pBuffer` into the currently open file. Returns the number of bytes written, or < 0 on error.

8.4.6 Utilities

CIPAddress

```
#include <circle/net/ipaddress.h>
```

IP_ADDRESS_SIZE
The size of an IP (v4) address (4 bytes).

class **CIPAddress**
This class encapsulates an IP (v4) address.

CIPAddress`::`**CIPAddress**`(`void`)`

CIPAddress`::`**CIPAddress**`(`u32 nAddress`)`

CIPAddress`::`**CIPAddress**`(`const u8 *pAddress`)`

CIPAddress`::`**CIPAddress**`(`const *CIPAddress* &rAddress`)`
Creates an `CIPAddress` object. Initialize it from different address formats.

boolean *CIPAddress*`::`**operator==**`(`const *CIPAddress* &rAddress2`)` const

boolean *CIPAddress*::**operator!=**(const *CIPAddress* &rAddress2) const

boolean *CIPAddress*::**operator==**(const u8 *pAddress2) const

boolean *CIPAddress*::**operator!=**(const u8 *pAddress2) const

boolean *CIPAddress*::**operator==**(u32 nAddress2) const

boolean *CIPAddress*::**operator!=**(u32 nAddress2) const
Compares this IP address with a second IP address in different formats.

CIPAddress &*CIPAddress*::**operator=**(u32 nAddress)
Assign a new IP address `nAddress`.

void *CIPAddress*::**Set**(u32 nAddress)

void *CIPAddress*::**Set**(const u8 *pAddress)

void *CIPAddress*::**Set**(const *CIPAddress* &rAddress)
Sets the IP address in different formats.

void *CIPAddress*::**SetBroadcast**(void)
Sets the IP address to the broadcast address (255.255.255.255).

CIPAddress::**operator u32**(void) const
Returns the IP address as `u32` value.

const u8 **CIPAddress*::**Get**(void) const
Returns a pointer to the IP address as an array with 4 bytes.

void *CIPAddress*::**CopyTo**(u8 *pBuffer) const
Copy the IP address to a buffer (4 bytes).

boolean *CIPAddress*::**IsNull**(void) const
Returns `TRUE`, if the IP address components are all zero (0.0.0.0).

boolean *CIPAddress*::**IsBroadcast**(void) const
Returns `TRUE` if the IP address is the broadcast address (255.255.255.255).

unsigned *CIPAddress*::**GetSize**(void) const
Returns the size of an IP (v4) address (4).

void *CIPAddress*::**Format**(*CString* *pString) const
Sets `*pString` to the dotted string representation of the IP address.

boolean *CIPAddress*::**OnSameNetwork**(const *CIPAddress* &rAddress2, const u8 *pNetMask) const
Returns `TRUE`, if this IP address is on the same network as `rAddress2` with `pNetMask` applied.

CMACAddress

```
#include <circle/macaddress.h>
```

Note: This class is belongs to the Circle base library, because it is needed there to implement non-USB network device drivers.

MAC_ADDRESS_SIZE
The size of an (Ethernet) MAC address (6 bytes).

class **CMACAddress**
This class encapsulates an (Ethernet) MAC address.

CMACAddress::**CMACAddress**(void)
Creates an CMACAddress object. The address is initialized as "invalid" and must be set, before it can be read.

CMACAddress::**CMACAddress**(const u8 *pAddress)
Creates an CMACAddress object. Set it from pAddress, which points to an array with 6 bytes.

boolean *CMACAddress*::**operator==**(const *CMACAddress* &rAddress2) const

boolean *CMACAddress*::**operator!=**(const *CMACAddress* &rAddress2) const
Compares this MAC address with a second MAC address.

void *CMACAddress*::**Set**(const u8 *pAddress)
Sets the MAC address to pAddress, which points to an array with 6 bytes.

void *CMACAddress*::**SetBroadcast**(void)
Sets the MAC address to the (Ethernet) broadcast address (FF:FF:FF:FF:FF:FF).

const u8 **CMACAddress*::**Get**(void) const
Returns a pointer to the MAC address as an array with 6 bytes.

void *CMACAddress*::**CopyTo**(u8 *pBuffer) const
Copy the MAC address to a buffer (6 bytes).

boolean *CMACAddress*::**IsBroadcast**(void) const
Returns TRUE if the MAC address is the (Ethernet) broadcast address (FF:FF:FF:FF:FF:FF).

unsigned *CMACAddress*::**GetSize**(void) const
Returns the size of an (Ethernet) MAC address (6).

void *CMACAddress*::**Format**(*CString* *pString) const
Sets *pString to the string representation of the MAC address.

8.4.7 WLAN support

The WLAN support in Circle is based on three elements:

1. Driver class *CBcm4343Device* for the WLAN hardware
2. *TCP/IP networking* subsystem, which is instantiated with the class *CNetSubSystem*
3. *WPA Supplicant* library, which is built from the submodule hostap, and is instantiated via the wrapper class *CWPASupplicant*

To enable WLAN support in Circle, these elements have to be created and initialized in this order. This is demonstrated in the WLAN sample. The third element is only required to use secure WLAN networks.

Note: The TCP/IP networking subsystem must be configured to use the WLAN device (`NetDeviceTypeWLAN`) and must be initialized, without waiting for an IP address from the DHCP server (with the parameter `FALSE`). Because the DHCP protocol requires *WPA Supplicant* to work, *CNetSubSystem::Initialize()* would never return otherwise.

CWPASupplicant

```
#include <wlan/hostap/wpa_supplicant/wpasupplicant.h>
```

class **CWPASupplicant**
This class is a wrapper for the well-known *WPA Supplicant* application, which has been ported to Circle as a library. An instance of this class is required for connecting to secure (i.e. WPA2) WLAN networks. The WLAN hardware driver *CBcm4343Device* and the *TCP/IP networking* subsystem must already running, when *WPA Supplicant* is initialized.

CWPASupplicant::**CWPASupplicant**(const char *pConfigFile)
Creates an instance of this class. `pConfigFile` is the path to the configuration file (e.g. "SD:/wpa_supplicant.conf").

boolean *CWPASupplicant*::**Initialize**(void)
Initializes the *WPA Supplicant* module and automatically starts to connect to one of the WLAN networks, which have been configured in the configuration file.

boolean *CWPASupplicant*::**IsConnected**(void) const
Returns `TRUE`, if a connection to a configured WLAN network is currently active.

8.5 Graphics

Circle provides several options for implementing graphical user interfaces (GUI) and for generating pixel and vector graphics on an attached HDMI or composite TV display. These options are described in this section.

8.5.1 C2DGraphics

The class `C2DGraphics` is part of the Circle base library and can be used to generate pixel graphics on a frame buffer, which is provided by the class *CBcmFrameBuffer*.

```
#include <circle/2dgraphics.h>
```

class **C2DGraphics**
This class is a software graphics library with VSync and hardware-accelerated double buffering.

C2DGraphics::**C2DGraphics**(unsigned nWidth, unsigned nHeight, boolean bVSync = TRUE, unsigned nDisplay = 0)
Creates on instance of this class. `nWidth` is the screen width in pixels (0 to detect). `nHeight` is the screen height in pixels (0 to detect). Set `bVSync` to `TRUE` to enable VSync and HW double buffering. `nDisplay` is the zero-based display number (for Raspberry Pi 4).

boolean *C2DGraphics*::**Initialize**(void)
Initializes the screen. Returns `TRUE` on success.

unsigned *C2DGraphics*::**GetWidth**(void) const
Returns the screen width in pixels.

unsigned *C2DGraphics*::**GetHeight**(void) const
Returns the screen height in pixels.

void *C2DGraphics*::**ClearScreen**(TScreenColor Color)
Clears the screen. `Color` is the color used to clear the screen (see *CScreenDevice::SetPixel()*).

void *C2DGraphics*::**DrawRect**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, TScreenColor Color)
Draws a filled rectangle. `nX` is the start X coordinate. `nY` is the start Y coordinate. `nWidth` is the rectangle width. `nHeight` is the rectangle height. `Color` is the rectangle color.

void *C2DGraphics*::**DrawRectOutline**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, TScreenColor Color)
Draws an unfilled rectangle (inner outline). `nX` is the start X coordinate. `nY` is the start Y coordinate. `nWidth` is the rectangle width. `nHeight` is the rectangle height. `Color` is the rectangle color.

void *C2DGraphics*::**DrawLine**(unsigned nX1, unsigned nY1, unsigned nX2, unsigned nY2, TScreenColor Color)
Draws a line. `nX1` is the start position X coordinate. `nY1` is the start position Y coordinate. `nX2` is the end position X coordinate. `nY2` is the end position Y coordinate. `Color` is the line color.

void *C2DGraphics*::**DrawCircle**(unsigned nX, unsigned nY, unsigned nRadius, TScreenColor Color)
Draws a filled circle. `nX` is the circle X coordinate. `nY` is the circle Y coordinate. `nRadius` is the circle radius. `Color` is the circle color.

void *C2DGraphics*::**DrawCircleOutline**(unsigned nX, unsigned nY, unsigned nRadius, TScreenColor Color)
Draws an unfilled circle (inner outline). `nX` is the circle X coordinate. `nY` is the circle Y coordinate. `nRadius` is the circle radius. `Color` is the circle color.

void *C2DGraphics*::**DrawImage**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, TScreenColor *PixelBuffer)
Draws an image from a pixel buffer. `nX` is the image X coordinate. `nY` is the image Y coordinate. `nWidth` is the image width. `nHeight` is the image height. `PixelBuffer` is a pointer to the pixels.

void *C2DGraphics*::**DrawImageTransparent**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, TScreenColor *PixelBuffer, TScreenColor TransparentColor)
Draws an image from a pixel buffer with transparent color. `nX` is the image X coordinate. `nY` is the image Y coordinate. `nWidth` is the image width. `nHeight` is the image height. `PixelBuffer` is a pointer to the pixels. `TransparentColor` is the color to use for transparency.

void *C2DGraphics*::**DrawImageRect**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, unsigned nSourceX, unsigned nSourceY, TScreenColor *PixelBuffer)
Draws an area of an image from a pixel buffer. `nX` is the image X coordinate. `nY` is the image Y coordinate. `nWidth` is the image width. `nHeight` is the image height. `nSourceX` is the source X coordinate in the pixel buffer. `nSourceY` is the source Y coordinate in the pixel buffer. `PixelBuffer` is a pointer to the pixels.

void *C2DGraphics*::**DrawImageRectTransparent**(unsigned nX, unsigned nY, unsigned nWidth, unsigned nHeight, unsigned nSourceX, unsigned nSourceY, unsigned nSourceWidth, unsigned nSourceHeight, TScreenColor *PixelBuffer, TScreenColor TransparentColor)
Draws an area of an image from a pixel buffer with transparent color. `nX` is the image X coordinate. `nY` is the image Y coordinate. `nWidth` is the image width. `nHeight` is the image height. `nSourceX` is the source X coordinate in the pixel buffer. `nSourceY` is the source Y coordinate in the pixel buffer. `nSourceWidth` is the source image width. `nSourceHeight` is the source image height. `PixelBuffer` is a pointer to the pixels. `TransparentColor` is the color to use for transparency.

void *C2DGraphics*::**DrawPixel**(unsigned nX, unsigned nY, TScreenColor Color)
Draws a single pixel. `nX` is the pixel X coordinate. `nY` is the pixel Y coordinate. `Color` is the pixel color.

Note: If you need to draw a lot of pixels, consider using *C2DGraphics::GetBuffer()* for better speed.

TScreenColor **C2DGraphics::GetBuffer**(void)
Gets raw access to the drawing buffer. Returns a pointer to the buffer.

void *C2DGraphics*::**UpdateDisplay**(void)
Once everything has been drawn, updates the display to show the contents on screen. If VSync is enabled, this method is blocking until the screen refresh signal is received (every 16ms for 60 FPS refresh rate).

8.5.2 LVGL

The Light and Versatile Graphics Library (LVGL) v8.2.0 can be used with Circle. This library provides an API, which is based on the C language. See the LVGL documentation for details.

```
#include <lvgl/lvgl.h>
```

class **CLVGL**
This class is a wrapper for LVGL and has to be instantiated to use this graphics library. The wrapper class supports USB mouse or touchscreen input.

CLVGL::**CLVGL**(*CScreenDevice* *pScreen, *CInterruptSystem* *pInterrupt)

CLVGL::**CLVGL**(*CBcmFrameBuffer* *pFrameBuffer, *CInterruptSystem* *pInterrupt)
Create an instance of this class. `pScreen` or `pFrameBuffer` reference the display to be used. `pInterrupt` is a pointer to the system interrupt object.

boolean *CLVGL*::**Initialize**(void)
Initializes to LVGL support. Returns `TRUE` on success.

void *CLVGL*::**Update**(boolean bPlugAndPlayUpdated = FALSE)
Updates the display. This has to be called continuously from the application main loop at `TASK_LEVEL`. `bPlugAndPlayUpdated` must be set to `TRUE`, if the application supports USB plug-and-play and *CUSBHostController::UpdatePlugAndPlay()* returned `TRUE` too.

8.5.3 µGUI

The µGUI library can be used with Circle. This library provides an API, which is based on the C language. Download the Reference Guide for details.

```
#include <ugui/uguicpp.h>
```

class **CUGUI**
This class is a wrapper for µGUI and has to be instantiated to use this graphics library. The wrapper class supports USB mouse or touchscreen input.

CUGUI::**CUGUI**(*CScreenDevice* *pScreen)
Creates an instance of this class. `pScreen` references the display to be used.

boolean *CUGUI*::**Initialize**(void)
Initializes to µGUI support. Returns `TRUE` on success.

void *CUGUI*::**Update**(boolean bPlugAndPlayUpdated = FALSE)
Updates the display. This has to be called continuously from the application main loop at TASK_LEVEL. bPlugAndPlayUpdated must be set to TRUE, if the application supports USB plug-and-play and *CUSBHostController::UpdatePlugAndPlay()* returned TRUE too.

8.5.4 Accelerated graphics

The accelerated graphics support is described in the *VC4* subsystem section.

8.6 VC4

The VC4 subsystem in addon/vc4 provides the VCHIQ driver as an interface to the audio and accelerated graphics services, which are offered by the Raspberry Pi firmware. The accelerated graphics support is not available on the Raspberry Pi 4 and with AARCH = 32 only. This section describes the components of the VC4 subsystem.

8.6.1 VCHIQ driver

```
#include <vc4/vchiq/vchiqdevice.h>
```

class **CVCHIQDevice** : public CLinuxDevice
This class is a driver for the VC host interface queue, which implements an interface to a number of service processes, which are running on the video processing unit (VPU) of the Raspberry Pi computers. Because this driver has been ported from Linux, it is based on the Linux kernel device driver emulation code in addon/linux. The API of the VCHIQ driver is based on the C language, and is not covered by this documentation.

CVCHIQDevice::**CVCHIQDevice**(*CMemorySystem* *pMemory, *CInterruptSystem* *pInterrupt)
Creates an instance of the VCHIQ driver class. There can be only one. pMemory and pInterrupt are pointers to the Circle memory and interrupt system objects.

boolean *CVCHIQDevice*::**Initialize**(void)
Initializes the VCHIQ driver. Returns TRUE on success. This method is inherited from the base class CLinuxDevice.

8.6.2 VCHIQ sound

The VCHIQ sound driver class *CVCHIQSoundBaseDevice* is described in the *Audio devices* section.

8.6.3 Accelerated graphics

The accelerated graphics support in addon/vc4/interface has been ported from the Raspberry Pi OS (former Raspbian) userland libraries, which implement the following APIs:

- EGL 1.4
- OpenGL ES 1.1 and 2.0
- OpenVG 1.1
- Dispmanx (proprietary)

Please see this website for detailed information about the first three APIs, which are not specific to Circle and are based on the C language.

Note: The accelerated graphics support is not available on the Raspberry Pi 4 and with `AARCH = 32` only.

CHAPTER

NINE

DEVICES

This section describes the interfaces of the different device driver classes in Circle.

The Circle project does not provide a single centralized C++ header file. Instead the header file(s), which must be included for a specific class, function or macro definition are specified in the related subsection.

9.1 Device management

In Circle most devices are represented by two things:

- By a device specific object, an instance of a class, which is derived from the class `CDevice`.
- By a device name, a C-string, which allows to retrieve a pointer to the device object, using the Circle device name service, which is implemented by the class `CDeviceNameService`.

Note: The I/O system of Circle is not as uniform as that of Linux, for example. Some device classes have specific interfaces, different from the well-known `Read()` and `Write()` interface and are not derived from `CDevice`.

9.1.1 CDevice

```
#include <circle/device.h>
```

class **CDevice**
This class is the base class for most device classes in Circle.

virtual int *CDevice*::**Read**(void *pBuffer, size_t nCount)
Performs a read operation of up to `nCount` bytes from a device to `pBuffer`. Returns the number of read bytes or < 0 on failure.

virtual int *CDevice*::**Write**(const void *pBuffer, size_t nCount)
Performs a write operation of up to `nCount` bytes to a device from `pBuffer`. Returns the number of written bytes or < 0 on failure.

virtual u64 *CDevice*::**Seek**(u64 ullOffset)
Sets the position of the read/write pointer of a device to the byte offset `ullOffset`. Returns the resulting offset, or `(u64) -1` on failure. This method is only implemented by block devices, character devices always return failure.

virtual boolean *CDevice*::**RemoveDevice**(void)
Requests the remove of a device from the system for pseudo plug-and-play. This is only implemented for USB devices (e.g. for USB mass-storage devices). Returns `TRUE` on the successful removal of the device.

void **RegisterRemovedHandler**(TDeviceRemovedHandler *pHandler, void *pContext = 0)
Registers a callback, which is invoked, when this device is removed from the system in terms of hot-plugging. `pHandler` gets called, before the device object is deleted. `pHandler` can be 0 to unregister a previously set handler. `pContext` is a user pointer, which is handed over to the handler.

```
void TDeviceRemovedHandler (CDevice *pDevice, void *pContext);
```

Note: See the file doc/usb-plug-and-play.txt for detailed information on USB plug-and-play support in Circle!

9.1.2 CDeviceNameService

```
#include <circle/devicenameservice.h>
```

class **CDeviceNameService**
In Circle devices can be registered by name and retrieved later using the same name. This is implemented in the class `CDeviceNameService`.

Note: A device name usually consists of an alpha name prefix, followed by a decimal device index number, which is >= 1. Partitions on block devices have another partition index, which is >= 1 too. Sound devices do not have a device index number. Examples:

Device name	Description
tty1	First screen device
ukbd1	First USB keyboard device
umsd1	First USB mass-storage device (e.g. flash drive)
umsd1-1	First partition on the first USB mass-storage device
sndpwm	PWM sound device
null	Null device

static *CDeviceNameService* **CDeviceNameService::Get**(void)
Returns a pointer to the single `CDeviceNameService` instance in the system.

CDevice **CDeviceNameService::GetDevice**(const char *pName, boolean bBlockDevice)
Returns a pointer to the device object of the device, with the name `pName` and the device type `bBlockDevice`, or 0 if the device is not found. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device.

CDevice **CDeviceNameService::GetDevice**(const char *pPrefix, unsigned nIndex, boolean bBlockDevice)
Returns a pointer to the device object of the device, with the name prefix `pName`, the device index `nIndex` and the device type `bBlockDevice`, or 0 if the device is not found. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device. The resulting name consists of the name prefix followed by the decimal device index (e.g. `umsd1` for the first USB mass-storage device).

void *CDeviceNameService* **::ListDevices**(*CDevice* *pTarget)
Generates a textual device name listing and writes it to the device `pTarget`.

void *CDeviceNameService* **::AddDevice**(const char *pName, *CDevice* *pDevice, boolean bBlockDevice)
Adds the pointer `pDevice` to a device object with the name `pName` to the device name registry. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device. This method is usually only used by device driver classes.

void *CDeviceNameService*::**AddDevice**(const char *pPrefix, unsigned nIndex, *CDevice* *pDevice, boolean bBlockDevice)

Adds the pointer `pDevice` to a device object with the name prefix `pName` and device index `nIndex` to the device name registry. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device. The resulting name consists of the name prefix followed by the decimal device index (e.g. `umsd1` for the first USB mass-storage device). This method is usually only used by device driver classes.

void *CDeviceNameService*::**RemoveDevice**(const char *pName, boolean bBlockDevice)

Removes the device with the name `pName` and the device type `bBlockDevice` from the device name registry. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device. This method is usually only used by device driver classes.

void *CDeviceNameService*::**RemoveDevice**(const char *pPrefix, unsigned nIndex, boolean bBlockDevice)

Removes the device with the name prefix `pPrefix`, the device index `nIndex` and the device type `bBlockDevice` from the device name registry. `bBlockDevice` is `TRUE`, if this is a block device, otherwise it is a character device. The resulting name consists of the name prefix followed by the decimal device index (e.g. `umsd1` for the first USB mass-storage device). This method is usually only used by device driver classes.

9.2 Character devices

Character devices usually accept and/or deliver a stream of characters via `Write()` and `Read()` calls. In Circle some character devices use a register-able callback handler instead of the `Read()` method, to deliver the received data. This makes time-consuming polling operations superfluous for these devices.

9.2.1 CScreenDevice

```
#include <circle/screen.h>
```

class **CScreenDevice** : public *CDevice*

This class can be used to write characters to the (usually HDMI) screen, which is connected to the Raspberry Pi computer. The screen is treated like a terminal and provides a number of control sequences (see `Write()`). This device has the name `"ttyN"` (N >= 1) in the device name service.

CScreenDevice::**CScreenDevice**(unsigned nWidth, unsigned nHeight, boolean bVirtual = FALSE, unsigned nDisplay = 0)

Constructs an instance of `CScreenDevice`. `nWidth` is the screen width and `nHeight` the screen height in number of pixels. Set both parameters to 0 to auto-detect the default resolution of the screen, which is usually the maximum resolution of the used monitor. `bVirtual` should be set to `FALSE` in any case. The Raspberry Pi 4 supports more than one display. `nDisplay` is the zero-based display number here. Multiple instances of `CScreenDevice` are possible here.

Note: You have to set `max_framebuffers=2` in *config.txt* to use two displays on the Raspberry Pi 4.

boolean *CScreenDevice*::**Initialize**(void)

Initializes the instance of `CScreenDevice` and clears the screen. Returns `TRUE` on success.

unsigned *CScreenDevice*::**GetWidth**(void) const

Returns the screen width in number of pixels.

unsigned *CScreenDevice*::**GetHeight**(void) const

Returns the screen height in number of pixels.

unsigned *CScreenDevice*::**GetColumns**(void) const
Returns the screen width in number of character columns.

unsigned *CScreenDevice*::**GetRows**(void) const
Returns the screen height in number of character rows.

int *CScreenDevice*::**Write**(const void *pBuffer, size_t nCount)
Writes `nCount` characters from `pBuffer` to the screen. Returns the number of written characters. This method supports several escape sequences:

Sequence	Description	Remarks
\E[B	Cursor down one line	
\E[H	Cursor home	
\E[A	Cursor up one line	
\E[%d;%dH	Cursor move to row %1 and column %2	starting at 1
^H	Cursor left one character	
\E[D	Cursor left one character	
\E[C	Cursor right one character	
^M	Carriage return	
\E[J	Clear to end of screen	
\E[K	Clear to end of line	
\E[%dX	Erase %1 characters starting at cursor	
^J	Carriage return/linefeed	
\E[0m	End of bold, half bright, reverse mode	
\E[1m	Start bold mode	
\E[2m	Start half bright mode	
\E[7m	Start reverse video mode	
\E[27m	Same as \E[0m	
\E[%dm	Set foreground color	%d = 30-37 or 90-97
\E[%dm	Set background color	%d = 40-47 or 100-107
^I	Move to next hardware tab	
\E[?25h	Normal cursor visible	
\E[?25l	Cursor invisible	
\E[%d;%dr	Set scroll region from row %1 to %2	starting at 1

^X = Control character, \E = Escape (\x1b), %d = Numerical parameter (ASCII)

void *CScreenDevice*::**SetPixel**(unsigned nPosX, unsigned nPosY, TScreenColor Color)
Sets the pixel at position `nPosX, nPosY` (based on `0, 0`) to `Color`. The color value depends on the macro value `DEPTH`, which can be defined as 8, 16 (default) or 32 in *include/circle/screen.h* or *Config.mk*. Circle defines the following standard color values:

- BLACK_COLOR (black)
- NORMAL_COLOR (white)
- HIGH_COLOR (red)
- HALF_COLOR (dark blue)

The following specific color values are defined:

- RED_COLOR
- GREEN_COLOR
- YELLOW_COLOR
- BLUE_COLOR

- MAGENTA_COLOR
- CYAN_COLOR
- WHITE_COLOR
- BRIGHT_BLACK_COLOR
- BRIGHT_RED_COLOR
- BRIGHT_GREEN_COLOR
- BRIGHT_YELLOW_COLOR
- BRIGHT_BLUE_COLOR
- BRIGHT_MAGENTA_COLOR
- BRIGHT_CYAN_COLOR
- BRIGHT_WHITE_COLOR

COLOR16(r, g, b)
Defines a color value for `DEPTH == 16`. r/g/b can be 0-31.

COLOR32(r, g, b, alpha)
Defines a color value for `DEPTH == 32`. r/g/b can be 0-255. alpha is usually 255.

TScreenColor *CScreenDevice*::**GetPixel**(unsigned nPosX, unsigned nPosY)
Returns the pixel color value at position `nPosX, nPosY` (based on `0, 0`).

void *CScreenDevice*::**Rotor**(unsigned nIndex, unsigned nCount)
Displays a rotating symbol in the upper right corner of the screen. `nIndex` is the index of the rotor to be displayed (0..3). `nCount` is the phase (angle) of the current rotor symbol (0..3).

CBcmFrameBuffer **CScreenDevice*::**GetFrameBuffer**(void)
Returns a pointer to the member of the type `CBcmFrameBuffer`, which can be used to directly manipulate the frame buffer.

9.2.2 CSerialDevice

```
#include <circle/serial.h>
```

class **CSerialDevice** : public *CDevice*
This class is a driver for the PL011-compatible UART(s) of the Raspberry Pi. The Raspberry Pi 4 provides five of these serial devices, the other models only one. This driver cannot be used for the Mini-UART (AUX). The GPIO mapping is as follows (SoC numbers):

nDevice	TXD	RXD	Support
0	GPIO14	GPIO15	All boards
0	GPIO32	GPIO33	Compute Modules
0	GPIO36	GPIO37	Compute Modules
1			None (AUX)
2	GPIO0	GPIO1	Raspberry Pi 4 only
3	GPIO4	GPIO5	Raspberry Pi 4 only
4	GPIO8	GPIO9	Raspberry Pi 4 only
5	GPIO12	GPIO13	Raspberry Pi 4 only

GPIO32/33 and GPIO36/37 can be selected with system option `SERIAL_GPIO_SELECT`. GPIO0/1 are normally reserved for the ID EEPROM. Handshake lines CTS and RTS are not supported.

This device has the name "ttySN" (N >= 1) in the device name service, where N = nDevice+1.

Note: This driver can be used in two modes: polling or interrupt driven. The mode is selected with the parameter pInterruptSystem of the constructor.

SERIAL_BUF_SIZE
This macro defines the size of the read and write ring buffers for the interrupt driver (default 2048). If you want to increase the buffer size, you have to specify a value, which is a power of two.

CSerialDevice::**CSerialDevice**(*CInterruptSystem* *pInterruptSystem = 0, boolean bUseFIQ = FALSE, unsigned nDevice = 0)
Constructs a CSerialDevice object. Multiple instances are possible on the Raspberry Pi 4. nDevice selects the used serial device (see the table above). pInterruptSystem is a pointer to interrupt system object, or 0 to use the polling driver. The interrupt driver uses the IRQ by default. Set bUseFIQ to TRUE to use the FIQ instead. This is recommended for higher baud rates.

boolean *CSerialDevice*::**Initialize**(unsigned nBaudrate = 115200, unsigned nDataBits = 8, unsigned nStopBits = 1, TParity Parity = ParityNone)
Initializes the serial device and sets the baud rate to nBaudrate bits per second. nDataBits selects the number of data bits (5..8, default 8) and nStopBits the number of stop bits (1..2, default 1). Parity can be CSerialDevice::ParityNone (default), CSerialDevice::ParityOdd or CSerialDevice::ParityEven. Returns TRUE on success.

int *CSerialDevice*::**Write**(const void *pBuffer, size_t nCount)
Writes nCount bytes from pBuffer to be sent out via the serial device. Returns the number of bytes, successfully sent or queued for send, or < 0 on error. The following errors are defined:

SERIAL_ERROR_BREAK

SERIAL_ERROR_OVERRUN

SERIAL_ERROR_FRAMING

SERIAL_ERROR_PARITY
Returned from Write() and Read() as a negative value. Please note, that these defined values are positive. You have to precede them with a minus for comparison.

int *CSerialDevice*::**Read**(void *pBuffer, size_t nCount)
Returns a maximum of nCount bytes, which have been received via the serial device, in pBuffer. The returned int value is the number of received bytes, 0 if data is not available, or < 0 on error (see Write()).

unsigned *CSerialDevice*::**GetOptions**(void) const
Returns the current serial options mask.

void *CSerialDevice*::**SetOptions**(unsigned nOptions)
Sets the serial options mask to nOptions. These options are defined:

SERIAL_OPTION_ONLCR
Translate NL to NL+CR on output (default)

void *CSerialDevice*::**RegisterMagicReceivedHandler**(const char *pMagic, *TMagicReceivedHandler* *pHandler)
Registers a magic received handler pHandler, which is called, when the string pMagic is found in the received data. pMagic must remain valid after return from this method. This method does only work with interrupt driver.

typedef void *CSerialDevice*::**TMagicReceivedHandler**(void)

9.2.3 CUSBKeyboardDevice

```
#include <circle/usb/usbkeyboard.h>
```

class **CUSBKeyboardDevice** : public CUSBHIDDevice

This class is a driver for USB standard keyboards. An instance of this class is automatically created, when a compatible USB keyboard is found in the USB device enumeration process. Therefore only the class methods needed to use the keyboard by an application are described here, not the methods used for initialization. This device has the name "ukbdN" (N >= 1) in the device name service.

Note: This driver class supports two keyboard modes: cooked and raw mode. In cooked mode the driver reports ISO-8859-1 character strings and the keyboard LEDs are handled automatically. There are six available keyboard maps (DE, ES, FR, IT, UK, US), which can be selected with the DEFAULT_KEYMAP configurable system option or the keymap= setting in the file *cmdline.txt* on the SD card.

In raw mode the driver reports the raw USB keyboard codes and modifier information and the LEDs have to be set manually by the application.

---

void *CUSBKeyboardDevice*::**RegisterKeyPressedHandler**(TKeyPressedHandler *pKeyPressedHandler)

Registers a function, which gets called, when a key is pressed in cooked mode:

typedef void **TKeyPressedHandler**(const char *pString)

pString points to a C-string, which contains the ISO-8859-1 code of the pressed key. This is normally only one character, but can be one of the following control sequences for special purpose keys:

Sequence	Key
\E	Escape
\177	Backspace
^I	Tabulator
^J	Return
\E[2~	Insert
\E[1~	Home
\E[5~	PageUp
\E[3~	Delete
\E[4~	End
\E[6~	PageDown
\E[A	Up
\E[B	Down
\E[D	Left
\E[C	Right
\E[[A	F1
\E[[B	F2
\E[[C	F3
\E[[D	F4
\E[[E	F5
\E[17~	F6
\E[18~	F7
\E[19~	F8
\E[20~	F9
\E[G	KP_Center

^X = Control character, \E = Escape (\x1b), \nnn = Octal code

void *CUSBKeyboardDevice*::**RegisterSelectConsoleHandler**(TSelectConsoleHandler *pSelectConsoleHandler)

Registers a function, which gets called, when the *Alt* key is pressed together with a function key *F1* to *F12* in cooked mode. This is used to select the console in some systems.

typedef void **TSelectConsoleHandler**(unsigned nConsole)

`nConsole` is the number of the console to select (0-11).

void *CUSBKeyboardDevice*::**RegisterShutdownHandler**(TShutdownHandler *pShutdownHandler)

Registers a function, which gets called, when the *Ctrl*, *Alt* and *Del* keys are pressed together in cooked mode. This is used to shutdown or reboot some systems.

typedef void **TShutdownHandler**(void)

void *CUSBKeyboardDevice*::**UpdateLEDs**(void)

In cooked mode this method has to be called from TASK_LEVEL from time to time, so that the status of the keyboard LEDs can be updated.

u8 *CUSBKeyboardDevice*::**GetLEDStatus**(void) const

Returns the LED status mask of the keyboard in cooked mode, with the following bit masks:

- LED_NUM_LOCK
- LED_CAPS_LOCK
- LED_SCROLL_LOCK

boolean *CUSBKeyboardDevice*::**SetLEDs**(u8 ucStatus)

Sets the keyboard LEDs according to the bit mask values listed under `GetLEDStatus()`. This method can be called on TASK_LEVEL only. It works in cooked and raw mode.

void *CUSBKeyboardDevice*::**RegisterKeyStatusHandlerRaw**(TKeyStatusHandlerRaw *pKeyStatusHandlerRaw, boolean bMixedMode = FALSE)

Registers a function, which gets called to report the keyboard status in raw mode. If `bMixedMode` is `FALSE`, then the cooked mode handlers are ignored. You can set it to `TRUE` to be able to use cooked mode and raw mode handlers together.

Note: It depends on the used USB keyboard, if the raw status handler gets called on status changes only or repeatedly after some delay too. The application must be able to handle both cases.

---

typedef void **TKeyStatusHandlerRaw**(unsigned char ucModifiers, const unsigned char RawKeys[6])

`RawKeys` contains up to six raw USB keyboard codes or zero in each byte. `ucModifiers` contains a mask of the pressed modifier keys, with the following bit masks:

- LCTRL
- LSHIFT
- ALT
- LWIN
- RCTRL
- RSHIFT
- ALTGR
- RWIN

9.2.4 CMouseDevice

```
#include <circle/input/mouse.h>
```

class **CMouseDevice** : public *CDevice*

This class is the generic mouse interface device. An instance of this class is automatically created, when a compatible USB mouse or USB gamepad with touchpad is found in the USB device enumeration process. Therefore only the class methods, needed to use the mouse by an application, are described here, not the method used for creation. This device has the name `"mouseN"` (N >= 1) in the device name service.

Note: This class supports two mouse modes: cooked and raw mode. In cooked mode a mouse cursor is shown on the screen and automatically controlled by the driver, which reports several mouse events (down, up, move, wheel).

In raw mode the driver directly reports the raw mouse displacement, button and wheel information.

---

boolean *CMouseDevice*::**Setup**(unsigned nScreenWidth, unsigned nScreenHeight)

Setup mouse device in cooked mode. `nScreenWidth` and `nScreenHeight` are the width and height of the screen in pixels. Returns `FALSE` on failure. This method must be called first in the setup process for a mouse in cooked mode.

void *CMouseDevice*::**RegisterEventHandler**(TMouseEventHandler *pEventHandler)
Registers an mouse event handler in cooked mode. `pEventHandler` is a pointer to the event handler with the following prototype:

typedef void **TMouseEventHandler**(*TMouseEvent* Event, unsigned nButtons, unsigned nPosX, unsigned nPosY, int nWheelMove)
`nPosX` is the X-coordinate of the current mouse cursor position in pixels (0 is on the left border). `nPosY` is the Y-coordinate of the position in pixels (0 is on the top border). These parameters are always valid (also in button and wheel events). `Event` is the reported mouse event with these possible values:

enum **TMouseEvent**

Event	Reports	Parameter
MouseEventMouseDown	one button, which has been pressed	`nButtons`
MouseEventMouseUp	one button, which has been released	`nButtons`
MouseEventMouseMove	a mouse move to a new screen position	`nPosX`, `nPosY`
MouseEventMouseWheel	a wheel move (raw displacement -/+)	`nWheelMove`

MOUSE_BUTTON_LEFT

MOUSE_BUTTON_RIGHT

MOUSE_BUTTON_MIDDLE

MOUSE_BUTTON_SIDE1

MOUSE_BUTTON_SIDE2
Bit masks for the `nButtons` parameter.

boolean *CMouseDevice*::**SetCursor**(unsigned nPosX, unsigned nPosY)
Sets the mouse cursor to a specific screen position in cooked mode. `nPosX` is the X-coordinate of the position in pixels (0 is on the left border). `nPosY` is the Y-coordinate of the position in pixels (0 is on the top border). Returns `FALSE` on failure.

boolean *CMouseDevice*::**ShowCursor**(boolean bShow)
Switches the mouse cursor on the screen on or off in cooked mode. Set `bShow` to `TRUE` to show the mouse cursor. Returns the previous state.

void *CMouseDevice*::**UpdateCursor**(void)
This method must be called frequently from `TASK_LEVEL` in cooked mode to update the mouse cursor on screen.

void *CMouseDevice*::**RegisterStatusHandler**(TMouseStatusHandler *pStatusHandler)
Registers the mouse status handler in raw mode. `pStatusHandler` is a pointer to the status handler with the following prototype:

typedef void **TMouseStatusHandler**(unsigned nButtons, int nDisplacementX, int nDisplacementY, int nWheelMove)
`nButtons` is the raw button mask reported from the mouse device. Use the same bit masks `MOUSE_BUTTON_LEFT` etc. listed above. `nDisplacementX` and `nDisplacementY` are the raw displacement values reported from the mouse device, with these limits:

MOUSE_DISPLACEMENT_MIN

MOUSE_DISPLACEMENT_MAX

unsigned *CMouseDevice*::**GetButtonCount**(void) const
Returns the number of supported buttons for this mouse device.

boolean *CMouseDevice*::**HasWheel**(void) const
Returns TRUE, if the mouse supports a mouse wheel.

9.2.5 CUSBGamePadDevice

```
#include <circle/usb/usbgamepad.h>
```

class **CUSBGamePadDevice** : public CUSBHIDDevice
This class is the base class for USB gamepad drivers and the generic application interface for USB gamepads. There are a number of different derived classes, which implement the drivers for specific gamepads. Circle automatically creates an instance of the right class, when a compatible USB gamepad is found in the USB device enumeration process. Therefore only the class methods, needed to use the gamepad by an application, are described here, not the methods used for initialization. This device has the name "upadN" (N >= 1) in the device name service.

Note: Circle supports gamepads, which are compatible with the USB HID-class specification and some other gamepads. To use a specific gamepad an application must normally know the mapping of the gamepad controls (buttons, axes etc.) to the gamepad report items. This mapping is not defined by the specification, but known for some widely available gamepads. A supported gamepad with a known mapping is called a "known gamepad" here and its driver offers additional services. The properties of a gamepad can be requested using GetProperties().

The *sample/27-usbgamepad* is working with all supported gamepads, but has limited function. The *sample/37-showgamepad* is only working with with known gamepads with more function.

---

struct **TGamePadState**
This structure is used to report the current state of the gamepad controls to the application. It can be fetched using GetInitialState() or by registering a status handler using RegisterStatusHandler().

```
#define MAX_AXIS        16
#define MAX_HATS        6

struct TGamePadState
{
        int naxes;              // Number of available axes and analog buttons
        struct
        {
                int value;      // Current position value
                int minimum;    // Minimum position value (normally 0)
                int maximum;    // Maximum position value (normally 255)
        }
        axes[MAX_AXIS];         // Array of axes and analog buttons

        int nhats;              // Number of available hat controls
        int hats[MAX_HATS];     // Current position value of hat controls

        int nbuttons;           // Number of available digital buttons
        unsigned buttons;       // Current status of digital buttons (bit mask)
```

(continues on next page)

(continued from previous page)

```
        int acceleration[3];    // Current state of acceleration sensor (x, y, z)
        int gyroscope[3];       // Current state of gyroscope sensor (x, y, z)
};

#define GAMEPAD_AXIS_DEFAULT_MINIMUM    0
#define GAMEPAD_AXIS_DEFAULT_MAXIMUM    255
```

enum **TGamePadButton**

Defines bit masks for the `TGamePadState::buttons` field for known gamepads. If the digital button is pressed, the respective bit is set. The following buttons are defined:

Digital button	Alias	Comment
GamePadButtonGuide	GamePadButtonXbox, GamePadButtonPS, GamePadButtonHome	
GamePadButtonLT	GamePadButtonL2, GamePadButtonLZ	
GamePadButtonRT	GamePadButtonR2, GamePadButtonRZ	
GamePadButtonLB	GamePadButtonL1, GamePadButtonL	
GamePadButtonRB	GamePadButtonR1, GamePadButtonR	
GamePadButtonY	GamePadButtonTriangle	
GamePadButtonB	GamePadButtonCircle	
GamePadButtonA	GamePadButtonCross	
GamePadButtonX	GamePadButtonSquare	
GamePadButtonSelect	GamePadButtonBack, GamePadButtonShare, GamePadButtonCapture	
GamePadButtonL3	GamePadButtonSL	left axis button
GamePadButtonR3	GamePadButtonSR	right axis button
GamePadButtonStart	GamePadButtonOptions	optional
GamePadButtonUp		
GamePadButtonRight		
GamePadButtonDown		
GamePadButtonLeft		
GamePadButtonPlus		optional
GamePadButtonMinus		optional
GamePadButtonTouchpad		optional

GAMEPAD_BUTTONS_STANDARD

GAMEPAD_BUTTONS_ALTERNATIVE

GAMEPAD_BUTTONS_WITH_TOUCHPAD

Number of digital buttons (19, 21 or 22) for known gamepads with different properties.

enum **TGamePadAxis**

Defines indices for the `TGamePadState::axes` field for known gamepads. This field covers the state information of axes and analog buttons. The following axes are defined:

Axes	Alias
GamePadAxisLeftX	
GamePadAxisLeftY	
GamePadAxisRightX	
GamePadAxisRightY	
GamePadAxisButtonLT	GamePadAxisButtonL2
GamePadAxisButtonRT	GamePadAxisButtonR2
GamePadAxisButtonUp	
GamePadAxisButtonRight	
GamePadAxisButtonDown	
GamePadAxisButtonLeft	
GamePadAxisButtonL1	
GamePadAxisButtonR1	
GamePadAxisButtonTriangle	
GamePadAxisButtonCircle	
GamePadAxisButtonCross	
GamePadAxisButtonSquare	

unsigned *CUSBGamePadDevice*::**GetProperties**(void)
Returns the properties of the gamepad as a bit mask of `TGamePadProperty` constants, which are:

Constant	Description
GamePadPropertyIsKnown	is a known gamepad
GamePadPropertyHasLED	supports `SetLEDMode()`
GamePadPropertyHasRGBLED	if set, `GamePadPropertyHasLED` is set too
GamePadPropertyHasRumble	supports `SetRumbleMode()`
GamePadPropertyHasGyroscope	provides sensor info in `TGamePadState`
GamePadPropertyHasTouchpad	has touchpad with button
GamePadPropertyHasAlternativeMapping	has additional +/- buttons, no START button

const TGamePadState **CUSBGamePadDevice*::**GetInitialState**(void)
Returns a pointer to the current gamepad state. This allows to initially request the information about the different gamepad controls. The control's state fields may have some default value, when a report from the gamepad has not been received yet. `GetReport()` is an deprecated alias for this method.

void *CUSBGamePadDevice*::**RegisterStatusHandler**(TGamePadStatusHandler *pStatusHandler)
Registers a handler function to be called on gamepad state changes. `pStatusHandler` is a pointer to this function, with this prototype:

typedef void **TGamePadStatusHandler**(unsigned nDeviceIndex, const *TGamePadState* *pGamePadState)
`nDeviceIndex` is the zero-based device index of this gamepad. The gamepad with the name `"upadN"` (N >= 1) in the device name service has the device index `N-1`. `pGamePadState` is a pointer to the current gamepad state.

boolean *CUSBGamePadDevice*::**SetLEDMode**(TGamePadLEDMode Mode)
Sets LED(s) on gamepads with multiple uni-color LEDs. `Mode` selects the LED mode to be set. Returns `TRUE` if the LED mode is supported and was successfully set. A gamepad may support only a subset of the defined `TGamePadLEDMode` modes:

- GamePadLEDModeOff
- GamePadLEDModeOn1
- GamePadLEDModeOn2
- GamePadLEDModeOn3

- GamePadLEDModeOn4
- GamePadLEDModeOn5
- GamePadLEDModeOn6
- GamePadLEDModeOn7
- GamePadLEDModeOn8
- GamePadLEDModeOn9
- GamePadLEDModeOn10

boolean *CUSBGamePadDevice*::**SetLEDMode**(u32 nRGB, u8 uchTimeOn, u8 uchTimeOff)
: Sets the LED on gamepads with a single flash-able RGB-color LED. The property bit `GamePadPropertyHasRGBLED` is set, if this method is supported by a gamepad. `nRGB` is the color value to be set (0x00rrggbb). `uchTimeOn` is the duration, while the LED is on in 1/100 seconds. `uchTimeOff` is the duration, while the LED is off in 1/100 seconds. Returns `TRUE`, if the operation was successful.

boolean *CUSBGamePadDevice*::**SetRumbleMode**(TGamePadRumbleMode Mode)
: Sets the rumble mode `Mode`, if the gamepad supports it (`GamePadPropertyHasRumble` is set). Returns `TRUE`, if the operation was successful. The following modes are defined:

- GamePadRumbleModeOff
- GamePadRumbleModeLow
- GamePadRumbleModeHigh

9.2.6 CUSBSerialDevice

```
#include <circle/usb/usbserial.h>
```

class **CUSBSerialDevice** : public CUSBFunction
: This class is the base class for USB serial device (aka interface, adapter) drivers and the generic application interface for USB serial devices. There are a number of different derived classes, which implement the drivers for specific devices. Circle automatically creates an instance of the appropriate class, when a compatible USB serial device is found in the USB device enumeration process. Therefore only the class methods, needed to use the USB serial device by an application, are described here, not the methods used for initialization. This device has the name "`uttyN`" (N >= 1) in the device name service.

Note: Circle currently supports USB serial devices, which are compatible with the USB CDC-class specification (interfaces 2-2-0 and 2-2-1) and other devices, which use the following controllers: CH341, CP2102, FT231x, PL2303.

There are many different combinations of USB vendor and device IDs for these devices and Circle supports only a small subset of these combinations, which were available for tests. If you have a USB serial device, which is not detected, there is still some chance, that the device can work with a Circle driver. You have to add the vendor/device ID combination of your device to the array `DeviceIDTable[]` at the end of the respective source file *lib/usb/usbserial*.cpp* and test it. Please report newly found vendor/device ID combinations and the used driver!

---

int *CUSBSerialDevice*::**Read**(void *pBuffer, size_t nCount)

int *CUSBSerialDevice*::**Write**(const void *pBuffer, size_t nCount)
: Reads/writes data from/to the USB serial device (see *CDevice*).

boolean *CUSBSerialDevice*::**SetBaudRate**(unsigned nBaudRate)
Sets the interface speed to a specific baud (bit) rate. `nBaudRate` is the rate in bits per second. Returns `TRUE` on success.

boolean *CUSBSerialDevice*::**SetLineProperties**(TUSBSerialDataBits DataBits, TUSBSerialParity Parity, TUSBSerialStopBits StopBits)
Sets the communication parameters number of data bits (`DataBits`), parity (`Parity`) and number of stop bits (`StopBits`) to the following values. Returns `TRUE` on success.

```
enum TUSBSerialDataBits
{
        USBSerialDataBits5 = 5,
        USBSerialDataBits6 = 6,
        USBSerialDataBits7 = 7,
        USBSerialDataBits8 = 8,
};

enum TUSBSerialStopBits
{
        USBSerialStopBits1 = 1,
        USBSerialStopBits2 = 2,
};

enum TUSBSerialParity
{
        USBSerialParityNone,
        USBSerialParityOdd,
        USBSerialParityEven,
};
```

9.2.7 CUSBPrinterDevice

```
#include <circle/usb/usbprinter.h>
```

class **CUSBPrinterDevice** : public CUSBFunction
This class is a simple driver for printers with USB interface. Only printers are supported, which are by default able to print ASCII characters on their own, not GDI printers. There is only one method of interest for applications, which writes the characters out to the printer. The printer device has the name `"uprnN"` (N >= 1) in the device name service.

int *CUSBPrinterDevice*::**Write**(const void *pBuffer, size_t nCount)
See *CDevice::Write()*.

9.2.8 CTouchScreenDevice

```
#include <circle/input/touchscreen.h>
```

class **CTouchScreenDevice** : public *CDevice*
This class is the generic touchscreen interface device. An instance of this class is automatically created, when a compatible USB touchscreen is found in the USB device enumeration process. When the class *CRPiTouchScreen* is manually instantiated, it is created too. This device has the name `"touchN"` (N >= 1) in the device name service.

void *CTouchScreenDevice*::**Update**(void)
: This method must be called about 60 times per second. This is required for the Raspberry Pi official touchscreen only, but to be prepared for any touchscreen, you should call it in any case.

void *CTouchScreenDevice*::**RegisterEventHandler**(TTouchScreenEventHandler *pEventHandler)
: Registers a handler function, which will be called on events from the touchscreen. The prototype of the handler is:

typedef void **TTouchScreenEventHandler**(*TTouchScreenEvent* Event, unsigned nID, unsigned nPosX, unsigned nPosY)
: `Event` specifies the received event. `nID` is an zero based identifier of the finger (for multi-touch). This first finger has always the ID zero. `nPosX` and `nPosY` specify the pixel position of the finger on the screen (for finger down and move events), where `(0,0)` is the top left position. The following touchscreen events are defined:

enum **TTouchScreenEvent**

- TouchScreenEventFingerDown
- TouchScreenEventFingerUp
- TouchScreenEventFingerMove

boolean *CTouchScreenDevice*::**SetCalibration**(const unsigned Coords[4], unsigned nWidth, unsigned nHeight)
: Sets the calibration parameters for the touchscreen. `Coords` are the usable coordinates (min-x, max-x, min-y, max-y) of the touchscreen. `nWidth` is the physical screen width and `nHeight` the height in number of pixels. Returns `TRUE`, if the calibration information is valid.

Note: The calibration parameters for a touchscreen can be determined with the Touchscreen calibrator.

9.2.9 CRPiTouchScreen

```
#include <circle/input/rpitouchscreen.h>
```

class **CRPiTouchScreen**
: This class is a driver for the official Raspberry Pi touchscreen. If you want to use this touchscreen, you have to create an instance of this class and initialize it. For the further use of this touchscreen an instance of the class *CTouchScreenDevice* is automatically created.

RPITOUCH_SCREEN_MAX_POINTS
: The maximum number of detected fingers on the touchscreen (10).

boolean *CRPiTouchScreen*::**Initialize**(void)
: Initializes the driver. Returns `TRUE` on success.

Note: The driver cannot detect, if an official Raspberry Pi touchscreen is actually connected. Normally it returns `TRUE` in any case.

9.2.10 CConsole

```
#include <circle/input/console.h>
```

class **CConsole** : public *CDevice*

This class implements a console with input and output stream and a line editor using the screen `tty1` and USB keyboard `ukbd1` devices or alternate device(s) (e.g. serial interface). The console device itself has the name `console` in the device name service. The sample/32-i2cshell demonstrates, how this class can be used to implement a simple shell.

Note: This class does not create instances of the devices, which are used for input and output. This has to be done by the application. The device `ukbd1` is created in the USB device enumeration process, when an USB keyboard is found.

CConsole`::`**CConsole**(*CDevice* *pAlternateDevice = 0, boolean bPlugAndPlay = FALSE)

Creates an instance of this class. `pAlternateDevice` is an alternate device to be used, if the USB keyboard is not attached (default none). `bPlugAndPlay` must be set to `TRUE` to enable USB plug-and-play support for the console. This constructor is mandatory for USB plug-and-play operation.

CConsole`::`**CConsole**(*CDevice* *pInputDevice, *CDevice* *pOutputDevice)

Creates an instance of this class. `pInputDevice` is the device used for input (instead of the USB keyboard) and `pOutputDevice` is the device used for output (instead of the screen).

boolean *CConsole*`::`**Initialize**(void)

Initializes the console class. Returns `TRUE`, if the operation has been successful.

void *CConsole*`::`**UpdatePlugAndPlay**(void)

Updates the USB plug-and-play configuration. This method must be called continuously, if the USB-plug-and-play support has been enabled in the constructor.

boolean *CConsole*`::`**IsAlternateDeviceUsed**(void) const

Returns `TRUE`, if the alternate device is used instead of screen/USB keyboard?

int *CConsole*`::`**Read**(void *pBuffer, size_t nCount)

See *CDevice::Read()*. This method does not block! It has to be called until `!= 0` is returned.

int *CConsole*`::`**Write**(const void *pBuffer, size_t nCount)

See *CDevice::Write()*.

unsigned *CConsole*`::`**GetOptions**(void) const

Returns the console options bit mask.

void *CConsole*`::`**SetOptions**(unsigned nOptions)

Sets the console options bit mask to `nOptions`.

The following bits are defined:

CONSOLE_OPTION_ICANON

Canonic input using a line editor is enabled (default).

CONSOLE_OPTION_ECHO

Echo input to output is enabled (default).

9.2.11 CNullDevice

```
#include <circle/nulldevice.h>
```

class **CNullDevice** : public *CDevice*

This class implements the null device, which accepts all written characters and returns 0 (EOF) on read. It can be used instead of other character device classes, for instance as target for the *System log*. This device has the name "null" in the device name service.

int *CNullDevice*::**Read**(void *pBuffer, size_t nCount)

Returns always 0 (EOF).

int *CNullDevice*::**Write**(const void *pBuffer, size_t nCount)

Returns the number of written bytes, but ignores them.

9.3 Block devices

Block devices provide the access to physical or logical drives (e.g. SD card, USB flash drive). They allow to read and write consecutive blocks of bytes of a fixed block size. Circle supports only block devices with a size of 512 bytes. All block devices provide the following methods, which are derived from the class *CDevice*. The detailed class descriptions below list additional class-specific methods only.

Method	Purpose	Description
Read()	read block(s) from device	*CDevice::Read()*
Write()	write block(s) to device	*CDevice::Write()*
Seek()	set read/write pointer position	*CDevice::Seek()*

9.3.1 CEMMCDevice

```
#include <SDCard/emmc.h>
```

class **CEMMCDevice** : public *CDevice*

This class provides the physical access to SD cards and to embedded MMC memory on the Compute Module 4. This class has to be manually instantiated, if an application wants to access one of these devices. This is demonstrated in addon/SDCard/sample. There can be only one instance of this device, which has the name `emmc1` in the device name service.

This class has drivers for two different interfaces, the SDHOST interface and the EMMC interface. The SDHOST interface is enabled by default on the Raspberry Pi 1-3 and Zero, when the system option `REALTIME` is not enabled. On the Raspberry Pi 4 the EMMC interface is used in any case, but can be used on the earlier models with the system option `NO_SDHOST` too. This is not possible, when you want to access the on-board WLAN device at the same time. To access the embedded MMC on the Compute Module 4, the system option `USE_EMBEDDED_MMC_CM4` has to be enabled.

CEMMCDevice::**CEMMCDevice**(*CInterruptSystem* *pInterruptSystem, *CTimer* *pTimer, *CActLED* *pActLED = 0)

Creates the instance of this class. `pInterruptSystem` is a pointer to the system interrupt object. `pTimer` is a pointer to the system timer object. `pActLED` can be specified to use the green Activity LED to inform the user, when the SD card is currently accessed. This is optional.

boolean *CEMMCDevice*::**Initialize**(void)

Initializes the EMMC or SDHOST device and detects the inserted SD card. Returns `TRUE` on success.

const u32 *CEMMCDevice::**GetID**(void)
Returns a pointer to the 32 byte (four `u32` words) long identifier of the inserted SD card. This information can be used to recognize a specific SD card again and is only valid, when `Initialize()` was successfully called before.

9.3.2 CUSBBulkOnlyMassStorageDevice

```
#include <circle/usb/usbmassdevice.h>
```

class **CUSBBulkOnlyMassStorageDevice** : public CUSBFunction
This class provides the physical access to USB mass-storage devices (e.g. flash drive, hard disk), which support the USB Mass Storage Bulk Only 1.0 specification. An instance of this class is automatically created in the USB device enumeration process, when a compatible USB device (interface 8-6-50) is found. These devices have the name `umsdN` (N >= 1) in the device name service.

unsigned *CUSBBulkOnlyMassStorageDevice*::**GetCapacity**(void) const
Returns the capacity of the device in number of 512 Byte blocks.

Note: Circle supports USB mass-storage devices with up to 2 TBytes capacity.

9.3.3 CPartition

```
#include <circle/fs/partition.h>
```

class **CPartition** : public *CDevice*
This class encapsulates one primary partition of a block device with Master Boot Record (MBR). An instance of this class is automatically created, when a block device object is initialized and a primary partition is found, when the MBR is scanned. Extended partitions (partition types 0x05 and 0x0F) and EFI partitions (type 0xEF) will be ignored in this process. These partition devices have the name `DEV-N` (N >= 1) in the device name service, where DEV is the name of the physical block device. For example the first found partition on a SD card has the name `emmc1-1`. This class only supports the standard methods of the *CDevice* class.

Note: These partition devices are only accessed by the Circle-native FAT filesystem support (class *CFATFileSystem*), but not by the *FatFs library*, which implements its own MBR management.

9.4 Audio devices

Circle supports the generation of sound via several hardware (PWM, I2S, HDMI) and software (VCHIQ) interfaces. It allows to capture sound data via the I2S hardware interface. Furthermore it is able to exchange MIDI data via USB and via a serial interface (UART). The latter has to be implemented in the application using the class *CSerialDevice*.

The base class of all sound generating and capturing devices is `CSoundBaseDevice`. The following table lists the provided classes for the different interfaces. The higher level support provides an additional conversion function for sound data in different formats as an example, which can be easily adapted for other sound classes.

Interface	Connector	Low level support	Higher level support
PWM	3.5” headphone jack	CPWMSoundBaseDevice	CPWMSoundDevice
I2S	GPIO header	CI2SSoundBaseDevice	
HDMI	HDMI(0)	CHDMISoundBaseDevice	
VCHIQ	HDMI or headphone jack	CVCHIQSoundBaseDevice	CVCHIQSoundDevice

Several sample programs demonstrate functions of the different audio devices:

- sample/12-pwmsound (playback a short sound sample using the PWM sound device)
- sample/29-miniorgan (using the PWM, HDMI or I2S sound device, USB or serial MIDI)
- sample/34-sounddevices (integrating multiple sound devices in one application)
- sample/42-i2sinput (I2S to PWM sound data converter)
- addon/vc4/sound/sample (HDMI or PWM sound support via VCHIQ interface)

The separate project MiniSynth Pi is a more extensive example for an application, which generates sound via the PWM or I2S interfaces in a multi-core environment, controlled with an USB or serial MIDI stream.

9.4.1 CSoundBaseDevice

```
#include <circle/soundbasedevice.h>
```

class **CSoundBaseDevice** : public *CDevice*

This class is the base for all sound generating and capturing classes in Circle. Normally it is not used directly in applications, but instead the derived class for the used interface is instantiated. Because this base class defines the common interface for all sound classes, it is described here first.

This class provides methods to start and stop the sound output and input, and to setup and manipulate one sound queue for each direction. Applications can use these queue(s) to provide/retrieve sound data with `Write()` and/or `Read()`. Alternatively they can override the methods `GetChunk()` and/or `PutChunk()` to directly write/read the audio samples to/from a provided DMA buffer.

Important: In a multi-core environment all methods, except if otherwise noted, have to be called or will be called (for callbacks) on CPU core 0.

Device activation

virtual boolean *CSoundBaseDevice*::**Start**(void)

Starts the transmission of sound data and initializes the device at the first call. Returns `TRUE`, if the operation was successful?

virtual void *CSoundBaseDevice*::**Cancel**(void)

Cancels the transmission of sound data. Cancel takes effect after a short delay.

virtual boolean *CSoundBaseDevice*::**IsActive**(void) const

Returns `TRUE`, if sound data transmission is currently running? This method can be called on any CPU core.

Output queue

These methods are used to output sound using a write queue. They are not used, if `GetChunk()` is overwritten instead.

boolean *CSoundBaseDevice*`::`**`AllocateQueue`**`(`unsigned nSizeMsecs`)`
: Allocates the queue used for `Write()`. `nSizeMsecs` is the size of the queue in milliseconds duration of the stream.

boolean *CSoundBaseDevice*`::`**`AllocateQueueFrames`**`(`unsigned nSizeFrames`)`
: Allocates the queue used for `Write()`. `nSizeFrames` is the size of the queue in number of audio frames.

void *CSoundBaseDevice*`::`**`SetWriteFormat`**`(`TSoundFormat Format, unsigned nChannels = 2`)`
: Sets the format of sound data provided to `Write()` to `Format`. `nChannels` must be 1 (Mono) or 2 (Stereo). The following (interleaved little endian) write formats are allowed:

  - SoundFormatUnsigned8
  - SoundFormatSigned16
  - SoundFormatSigned24 (occupies 3 bytes)
  - SoundFormatSigned24_32 (occupies 4 bytes)

int *CSoundBaseDevice*`::`**`Write`**`(`const void *pBuffer, size_t nCount`)`
: Appends audio samples from `pBuffer` to the output queue. `nCount` is the size of the buffer in bytes and must be a multiple of the frame size. Returns the number of bytes from the buffer, which have to be consumed successfully. This value may be smaller than `nCount`, in which case some frames have been ignored. This method can be called on any CPU core.

unsigned *CSoundBaseDevice*`::`**`GetQueueSizeFrames`**`(`void`)`
: Returns the output queue size in number of frames. This method can be called on any CPU core.

unsigned *CSoundBaseDevice*`::`**`GetQueueFramesAvail`**`(`void`)`
: Returns the number of frames currently available in the output queue, which are waiting to be sent to the hardware interface. This method can be called on any CPU core.

void *CSoundBaseDevice*`::`**`RegisterNeedDataCallback`**`(`TSoundDataCallback *pCallback, void *pParam`)`
: Registers the callback function `pCallback`, which is called, when more sound data is needed, which means that at least half of the queue is empty. `pParam` is a user parameter to be handed over to the callback. The callback function has the following prototype:

typedef void **`TSoundDataCallback`**`(`void *pParam`)`
: `pParam` is the user parameter, which has been handed over to `RegisterNeedDataCallback()`.

Input queue

These methods are used to input sound data using a read queue. They are not used, if `PutChunk()` is overwritten instead.

boolean *CSoundBaseDevice*`::`**`AllocateReadQueue`**`(`unsigned nSizeMsecs`)`
: Allocates the queue used for `Read()`. `nSizeMsecs` is the size of the queue in milliseconds duration of the stream.

boolean *CSoundBaseDevice*`::`**`AllocateReadQueueFrames`**`(`unsigned nSizeFrames`)`
: Allocates the queue used for `Read()`. `nSizeFrames` is the size of the queue in number of audio frames.

void *CSoundBaseDevice*`::`**`SetReadFormat`**`(`TSoundFormat Format, unsigned nChannels = 2, boolean bLeftChannel = TRUE`)`
: Sets the format of sound data returned from `Read()` to `Format`. `nChannels` must be 1 (Mono) or 2 (Stereo). If `bLeftChannel` is `TRUE`, `Read()` returns the left channel, if `nChannels == 1`. The following (interleaved little endian) read formats are allowed:

- SoundFormatUnsigned8
- SoundFormatSigned16
- SoundFormatSigned24 (occupies 3 bytes)
- SoundFormatSigned24_32 (occupies 4 bytes)

int *CSoundBaseDevice*::**Read**(void *pBuffer, size_t nCount)
Moves up to `nCount` bytes of audio samples into `pBuffer` from the input queue and returns the number of returned bytes, which is a multiple of the frame size in any case, or 0 if no data is available. `nCount` must be a multiple of the frame size. This method can be called on any CPU core.

unsigned *CSoundBaseDevice*::**GetReadQueueSizeFrames**(void)
Returns the input queue size in number of frames. This method can be called on any CPU core.

unsigned *CSoundBaseDevice*::**GetReadQueueFramesAvail**(void)
Returns the number of frames currently available in the input queue, which are waiting to be read by the application. This method can be called on any CPU core.

void *CSoundBaseDevice*::**RegisterHaveDataCallback**(TSoundDataCallback *pCallback, void *pParam)
Registers the callback function `pCallback`, which is called, when enough sound data is available for `Read()`, which means that at least half of the queue is full. `pParam` is a user parameter to be handed over to the callback. The callback function has this prototype: *TSoundDataCallback()*.

Alternate interface

Optionally an application can bypass the output and/or input queues and can directly provide/consume the audio samples to/from a buffer, which is handed over to the callback methods `GetChunk()` and/or `PutChunk()`. This/These method(s) have to be overwritten to use the alternate interface. The format of the samples depends on the used hardware/software interface:

Interface	Format	Remarks
PWM	SoundFormatUnsigned32	range max. depends on sample rate and PWM clock rate
I2S	SoundFormatSigned24_32	occupies 4 bytes
HDMI	SoundFormatIEC958	special frame format (S/PDIF)
VCHIQ	SoundFormatSigned16	occupies 4 bytes

virtual int *CSoundBaseDevice*::**GetRangeMin**(void) const

virtual int *CSoundBaseDevice*::**GetRangeMax**(void) const
Return the minimum/maximum value of one sample. These methods can be called on any CPU core.

boolean *CSoundBaseDevice*::**AreChannelsSwapped**(void) const
Returns `TRUE`, if the application has to write the right channel first into buffer in `GetChunk()`.

virtual unsigned *CSoundBaseDevice*::**GetChunk**(s16 *pBuffer, unsigned nChunkSize)

virtual unsigned *CSoundBaseDevice*::**GetChunk**(u32 *pBuffer, unsigned nChunkSize)
You may override one of these methods to provide the sound samples. The first method is used for the VCHIQ interface, the second for all other interfaces. `pBuffer` is a pointer to the buffer, where the samples have to be placed. `nChunkSize` is the size of the buffer in words. Returns the number of words written to the buffer, which is normally `nChunkSize`, or 0 to stop the transfer. Each sample consists of two words (left channel, right channel), where each word must be between `GetRangeMin()` and `GetRangeMax()`. The HDMI interface requires a special frame format here, which can be applied using `ConvertIEC958Sample()`.

virtual void *CSoundBaseDevice*::**PutChunk**(const u32 *pBuffer, unsigned nChunkSize)
: You may override this method to consume the received sound samples. `pBuffer` is a pointer to the buffer, where the samples have been placed. `nChunkSize` is the size of the buffer in words. Each sample consists of two words (left channel, right channel).

u32 *CSoundBaseDevice*::**ConvertIEC958Sample**(u32 nSample, unsigned nFrame)
: This method can be called from `GetChunk()` to apply the framing on IEC958 (S/PDIF) samples. `nSample` is a 24-bit signed sample value as `u32`, where upper bits don't care. `nFrame` is the number of the IEC958 frame, this sample belongs to (0..191).

9.4.2 CPWMSoundBaseDevice

```
#include <circle/pwmsoundbasedevice.h>
```

class **CPWMSoundBaseDevice** : public *CSoundBaseDevice*
: This class is a driver for the PWM sound interface. The generated sound is available via the 3.5" headphone jack, provided by most Raspberry Pi models. Most of the methods, available for using this class, are provided by the base class *CSoundBaseDevice*. Only the constructor is specific to this class. This device has the name `"sndpwm"` in the device name service (character device).

Note: On the Raspberry Pi Zero, which does not have a headphone jack, the output from the PWM sound interface can be used via the GPIO header. You have to define the system option `USE_PWM_AUDIO_ON_ZERO` for this purpose. See the file include/circle/sysconfig.h for details!

CPWMSoundBaseDevice::**CPWMSoundBaseDevice**(*CInterruptSystem* *pInterrupt, unsigned nSampleRate = 44100, unsigned nChunkSize = 2048)
: Constructs an instance of this class. There can be only one. `pInterrupt` is a pointer to the interrupt system object. `nSampleRate` is the sample rate in Hz. `nChunkSize` is twice the number of samples (words) to be handled with one call to `GetChunk()` (one word per stereo channel). Decreasing this value also decreases the latency on this interface, but increases the IRQ load on CPU core 0.

9.4.3 CPWMSoundDevice

```
#include <circle/pwmsounddevice.h>
```

class **CPWMSoundDevice** : public *CPWMSoundBaseDevice*
: This class is a PWM playback device for sound data, which is available in main memory. It extents the class *CPWMSoundBaseDevice*, but has its own interface. The sample rate is fixed at 44100 Hz.

CPWMSoundDevice::**CPWMSoundDevice**(*CInterruptSystem* *pInterrupt)
: Constructs an instance of this class. There can be only one. `pInterrupt` is a pointer to the interrupt system object.

void *CPWMSoundDevice*::**Playback**(void *pSoundData, unsigned nSamples, unsigned nChannels, unsigned nBitsPerSample)
: Starts playback of the sound data at `pSoundData` via the PWM sound device. `nSamples` is the number of samples, where for Stereo the L/R samples are count as one. `nChannels` is 1 for Mono or 2 for Stereo. `nBitsPerSample` is 8 (unsigned char sound data) or 16 (signed short sound data).

boolean *CPWMSoundDevice*::**PlaybackActive**(void) const
: Returns `TRUE`, while the playback is active.

void *CPWMSoundDevice*::**CancelPlayback**(void)

Cancels the playback. The operation takes affect with a short delay, after which `PlaybackActive()` returns `FALSE`.

9.4.4 CI2SSoundBaseDevice

```
#include <circle/i2ssoundbasedevice.h>
```

class **CI2SSoundBaseDevice** : public *CSoundBaseDevice*

This class is a driver for the I2S sound interface. The generated sound is available via the GPIO header in the format: two 32-bit wide channels with 24-bit signed data. Most of the methods, available for using this class, are provided by the base class *CSoundBaseDevice*. Only the constructor is specific to this class. This device has the name `"sndi2s"` in the device name service (character device).

Note: The following GPIO pins have to be connected (SoC numbers, not header positions):

Name	Pin number	On early models	Description
PCM_CLK	GPIO18	GPIO28	Bit clock (output or input)
PCM_FS	GPIO19	GPIO29	Frame clock (output or input)
PCM_DIN	GPIO20	GPIO30	Data input (not for TX only mode)
PCM_DOUT	GPIO21	GPIO31	Data output (not for RX only mode)

The clock pins are outputs in master mode, or inputs in slave mode. On early models the signals are exposed on the separate P5 header.

Note: This driver class supports several I2S interfaces. Some interfaces require an additional I2C connection to work. The following interfaces are known to work:

- pHAT DAC (with PCM5102A DAC)
- PiFi DAC+ v2.0 (with PCM5122 DAC)
- Adafruit I2S Audio Bonnet (with UDA1334A DAC)
- Adafruit I2S 3W Class D Amplifier Breakout (with MAX98357A DAC)

CI2SSoundBaseDevice::**CI2SSoundBaseDevice**(*CInterruptSystem* *pInterrupt, unsigned nSampleRate = 192000, unsigned nChunkSize = 8192, boolean bSlave = FALSE, *CI2CMaster* *pI2CMaster = 0, u8 ucI2CAddress = 0, TDeviceMode DeviceMode = DeviceModeTXOnly)

Constructs an instance of this class. There can be only one. `pInterrupt` is a pointer to the interrupt system object. `nSampleRate` is the sample rate in Hz. `nChunkSize` is twice the number of samples (words) to be handled with one call to `GetChunk()` (one word per stereo channel). Decreasing this value also decreases the latency on this interface, but increases the IRQ load on CPU core 0.

`bSlave` enables the slave mode (PCM clock and FS clock are inputs). `pI2CMaster` is a pointer to an I2C master object (0 if no I2C DAC initialization is required). `ucI2CAddress` is the I2C slave address of the DAC (0 for auto probing the addresses 0x4C and 0x4D). `DeviceMode` selects, which transfer direction will be used, with this supported values:

- DeviceModeTXOnly (output)
- DeviceModeRXOnly (input)

- DeviceModeTXRX (output and input)

9.4.5 CHDMISoundBaseDevice

```
#include <circle/hdmisoundbasedevice.h>
```

class **CHDMISoundBaseDevice** : public *CSoundBaseDevice*

This class is a driver for HDMI displays with audio support. It directly accesses the hardware and does not require *Multitasking* support and the *VCHIQ driver* in the system. Most of the methods, available for using this class, are provided by the base class *CSoundBaseDevice*. This device has the name "sndhdmi" in the device name service (character device).

Note: This driver does not support HDMI1 on the Raspberry Pi 4 and 400 (HDMI0 only).

This driver supports a DMA and a polling mode. The latter is intended for very time critical and cache-sensitive applications, which cannot use interrupts.

CHDMISoundBaseDevice::**CHDMISoundBaseDevice**(*CInterruptSystem* *pInterrupt, unsigned nSampleRate = 48000, unsigned nChunkSize = 384 * 10)

Constructs an instance of this class to work in DMA mode. There can be only one. `pInterrupt` is a pointer to the interrupt system object. `nSampleRate` is the sample rate in Hz. `nChunkSize` is twice the number of samples (words) to be handled with one call to `GetChunk()` (one word per stereo channel, must be a multiple of 384). Decreasing this value also decreases the latency on this interface, but increases the IRQ load on CPU core 0.

CHDMISoundBaseDevice::**CHDMISoundBaseDevice**(unsigned nSampleRate = 48000)

Constructs an instance of this class to work in polling mode. There can be only one. `nSampleRate` is the sample rate in Hz.

boolean *CHDMISoundBaseDevice*::**IsWritable**(void)

Returns if the data FIFO has room for at least one sample to be written? This method can be called in polling mode only.

void *CHDMISoundBaseDevice*::**WriteSample**(s32 nSample)

Writes one sample to the data FIFO. `nSample` is the 24-bit signed sample to be written. This method can be called in polling mode only and only, when *IsWritable()* returned `TRUE` before. Must be called twice for each frame (for left and right channel).

9.4.6 CVCHIQSoundBaseDevice

```
#include <vc4/sound/vchiqsoundbasedevice.h>
```

class **CVCHIQSoundBaseDevice** : public *CSoundBaseDevice*

This class provides low-level access to the VCHIQ sound service, which is able to output sound via HDMI displays with audio support, or via the 3.5" headphone jack of Raspberry Pi models, which have it. This class requires, that the *Multitasking* support and the *VCHIQ driver* are available in the system. Most of the methods, available for using this class, are provided by the base class *CSoundBaseDevice*. This class description covers only the methods, which are specific to this class. This device has the name "sndvchiq" in the device name service (character device).

CVCHIQSoundBaseDevice::**CVCHIQSoundBaseDevice**(*CVCHIQDevice* *pVCHIQDevice, unsigned nSampleRate = 44100, unsigned nChunkSize = 4000, TVCHIQSoundDestination Destination = VCHIQSoundDestinationAuto)

Constructs an instance of this class. There can be only one. `pVCHIQDevice` is a pointer to the VCHIQ interface device. `nSampleRate` is the sample rate in Hz (44100..48000). `nChunkSize` is the number of samples transferred at once. `Destination` is the target device, the sound data is sent to (detected automatically, if equal to `VCHIQSoundDestinationAuto`), with these possible values:

enum **TVCHIQSoundDestination**

- VCHIQSoundDestinationAuto
- VCHIQSoundDestinationHeadphones
- VCHIQSoundDestinationHDMI
- VCHIQSoundDestinationUnknown

void *CVCHIQSoundBaseDevice*::**SetControl**(int nVolume, TVCHIQSoundDestination Destination = VCHIQSoundDestinationUnknown)

Sets the output volume to `nVolume` (-10000..400) and optionally the target device to `Destination` (not modified, if equal to `VCHIQSoundDestinationUnknown`). This method can be called, while the sound data transmission is running. The following macros are defined for specifying the volume:

VCHIQ_SOUND_VOLUME_MIN

VCHIQ_SOUND_VOLUME_DEFAULT

VCHIQ_SOUND_VOLUME_MAX

9.4.7 CVCHIQSoundDevice

```
#include <vc4/sound/vchiqsounddevice.h>
```

class **CVCHIQSoundDevice** : private *CVCHIQSoundBaseDevice*

This class is a VCHIQ playback device for sound data, which is available in main memory. It extents the class *CVCHIQSoundBaseDevice*, but has its own interface. The sample rate is fixed at 44100 Hz.

CVCHIQSoundDevice::**CVCHIQSoundDevice**(*CVCHIQDevice* *pVCHIQDevice, TVCHIQSoundDestination Destination = VCHIQSoundDestinationAuto)

Constructs an instance of this class. There can be only one. `pVCHIQDevice` is a pointer to the VCHIQ interface device. `Destination` is the target device, the sound data is sent to (see *TVCHIQSoundDestination* for the available options).

boolean *CVCHIQSoundDevice*::**Playback**(void *pSoundData, unsigned nSamples, unsigned nChannels, unsigned nBitsPerSample)

Starts playback of the sound data at `pSoundData` via the VCHIQ sound device. `nSamples` is the number of samples, where for Stereo the L/R samples are count as one. `nChannels` is 1 for Mono or 2 for Stereo. `nBitsPerSample` is 8 (unsigned char sound data) or 16 (signed short sound data). Returns `TRUE` on success.

boolean *CVCHIQSoundDevice*::**PlaybackActive**(void) const

Returns `TRUE`, while the playback is active.

void *CVCHIQSoundDevice*::**CancelPlayback**(void)
Cancels the playback. The operation takes affect with a short delay, after which `PlaybackActive()` returns `FALSE`.

void *CVCHIQSoundDevice*::**SetControl**(int nVolume, TVCHIQSoundDestination Destination = VCHIQSoundDestinationUnknown)
See *CVCHIQSoundBaseDevice::SetControl()*.

9.4.8 CUSBMIDIDevice

```
#include <circle/usb/usbmidi.h>
```

class **CUSBMIDIDevice** : public CUSBFunction
This class is a driver for USB Audio Class MIDI 1.0 devices. An instance of this class is automatically created, when a compatible device is found in the USB device enumeration process. Therefore only the class methods needed to use an USB MIDI device by an application are described here, not the methods used for initialization. This device has the name `"umidiN"` (N >= 1) in the device name service (character device).

Note: See the Universal Serial Bus Device Class Definition for MIDI Devices, Release 1.0 for information about USB MIDI packets and virtual MIDI cables!

---

void *CUSBMIDIDevice*::**RegisterPacketHandler**(TMIDIPacketHandler *pPacketHandler)
Registers a callback function, which is called, when a MIDI packet arrives. `pPacketHandler` is a pointer to the function, which has the following prototype:

typedef void **TMIDIPacketHandler**(unsigned nCable, u8 *pPacket, unsigned nLength)
`nCable` is the number of the virtual MIDI cable (0..15). `pPacket` is a pointer to one received MIDI packet. `nLength` is the number of valid bytes in the packet (1..3).

boolean *CUSBMIDIDevice*::**SendEventPackets**(const u8 *pData, unsigned nLength)
Sends one or more packets in the encoded USB MIDI event packet format. `pData` is a pointer to the packet buffer. `nLength` is the length of the packet buffer in bytes, which must be a multiple of 4. Returns `TRUE`, if the operation has been successful. This function fails, if `nLength` is not a multiple of 4 or the send function is not supported. The format of the USB MIDI event packets is not validated.

boolean *CUSBMIDIDevice*::**SendPlainMIDI**(unsigned nCable, const u8 *pData, unsigned nLength)
Sends one or more messages in plain MIDI message format. `nCable` is the number of the virtual MIDI cable (0..15). `pData` is a pointer to the message buffer. `nLength` is the length of the message buffer in bytes. Returns `TRUE`, if the operation has been successful. This function fails, if the message format is invalid or the send function is not supported.

9.5 Network devices

Network devices allow the low-level access to network interfaces, and provide methods for sending and receiving network frames and additional functions to manage the network access. Circle supports the access to IEEE 802.3 Ethernet and to IEEE 802.11 wireless LAN (WLAN) interfaces. All network device classes are derived from the base class *CNetDevice*, which defines the low-level API for network applications. Please note, that applications normally use the high-level TCP/IP socket interface, which is provided by the class *CSocket*.

9.5.1 CNetDevice

```
#include <circle/netdevice.h>
```

class **CNetDevice**
This class is the base class for all network device driver classes and defines the low-level API for specific network applications, which want to directly exchange frames via a network interface. Network devices are not registered in the device name service and can be found using the methods *CNetDevice::GetNetDevice()*.

FRAME_BUFFER_SIZE
This macro defines the maximum size of a sent or received frame on a network interface. Network buffers usually have this size in Circle.

virtual TNetDeviceType *CNetDevice*::**GetType**(void)
Returns the type of this network device, which is one of these:

type **TNetDeviceType**

- NetDeviceTypeEthernet
- NetDeviceTypeWLAN

virtual const *CMACAddress* **CNetDevice*::**GetMACAddress**(void) const
Returns a pointer to a MAC address object, which holds our own address at this network interface device.

virtual boolean *CNetDevice*::**IsSendFrameAdvisable**(void)
Returns `TRUE`, if it is advisable to call `SendFrame()`.

Note: `SendFrame()` can be called at any time, but may fail, when the TX queue is full. This method gives a hint, if calling `SendFrame()` is advisable.

---

virtual boolean *CNetDevice*::**SendFrame**(const void *pBuffer, unsigned nLength)
Sends a valid frame to the network. `pBuffer` is a pointer to the frame, which does not contain the frame checking sequence (FCS). `nLength` is the frame length in bytes. The frame does not need to be padded by the application.

virtual boolean *CNetDevice*::**ReceiveFrame**(void *pBuffer, unsigned *pResultLength)
Polls for a frame, which has been received via the network interface. `pBuffer` is a pointer to a buffer, where the frame will be placed, and must have the size *FRAME_BUFFER_SIZE*. `pResultLength` is a pointer to a variable, which receives the valid frame length. Returns `TRUE`, if a frame is returned in the buffer, `FALSE`, if nothing has been received.

virtual boolean *CNetDevice*::**IsLinkUp**(void)
Returns `TRUE`, if the physical link (PHY) is active.

virtual TNetDeviceSpeed *CNetDevice*::**GetLinkSpeed**(void)
Returns the speed of the physical link (PHY), if it is active, or `NetDeviceSpeedUnknown`, if it is not known. The following link speeds are defined:

type **TNetDeviceSpeed**

- NetDeviceSpeed10Half
- NetDeviceSpeed10Full
- NetDeviceSpeed100Half
- NetDeviceSpeed100Full
- NetDeviceSpeed1000Half

- NetDeviceSpeed1000Full
- NetDeviceSpeedUnknown

virtual boolean *CNetDevice*::**UpdatePHY**(void)
Updates the device settings according to physical link (PHY) status. Returns `FALSE`, if this function is not supported.

Note: This method is called continuously every two seconds by the PHY task of the *TCP/IP networking* subsystem. If you do not use this subsystem, you have to call this method on your own.

---

static const char **CNetDevice*::**GetSpeedString**(TNetDeviceSpeed Speed)
Returns a description for the link speed value `Speed`, which normally has been returned from `GetLinkSpeed()`.

static *CNetDevice* **CNetDevice*::**GetNetDevice**(unsigned nDeviceNumber)
Returns a pointer to the network device object for the zero-based number `nDeviceNumber` of a network device, or 0, if the device is not available.

static *CNetDevice* **CNetDevice*::**GetNetDevice**(TNetDeviceType Type)
Returns a pointer to the first network device object of the type `Type`, which is either a specific network device type (see *CNetDevice::GetType()*), or `NetDeviceTypeAny` to search for any network device.

9.5.2 CSMSC951xDevice

```
#include <circle/usb/smsc951x.h>
```

class **CSMSC951xDevice** : public CUSBFunction, *CNetDevice*
This class is a driver for the SMSC9512 and SMSC9514 Ethernet network interface devices, which are attached to the internal USB hub of Raspberry Pi 1, 2 and 3 Model B boards. This class is automatically instantiated in the USB device enumeration process, when a device of this type is found. This class does not provide specific methods, its API is defined by the base class *CNetDevice*.

9.5.3 CLAN7800Device

```
#include <circle/usb/lan7800.h>
```

class **CLAN7800Device** : public CUSBFunction, *CNetDevice*
This class is a driver for the LAN7800 Gigabit Ethernet network interface device, which is attached to an internal USB hub of the Raspberry Pi 3 Model B+ board. This class is automatically instantiated in the USB device enumeration process, when a device of this type is found. This class does not provide specific methods, its API is defined by the base class *CNetDevice*.

9.5.4 CUSBCDCEthernetDevice

```
#include <circle/usb/usbcdcethernet.h>
```

class **CUSBCDCEthernetDevice** : public CUSBFunction, *CNetDevice*
This class is a driver for the USB CDC Ethernet network interface device, which is supported by QEMU. This class is automatically instantiated in the USB device enumeration process, when a device of this type is found. This class does not provide specific methods, its API is defined by the base class *CNetDevice*.

9.5.5 CBcm54213Device

```
#include <circle/bcm54213.h>
```

class **CBcm54213Device** : public *CNetDevice*

This class is a driver for the BCM54213PE Gigabit Ethernet Transceivers of the Raspberry Pi 4, 400 and Compute Module 4. It is instantiated in the *TCP/IP networking* subsystem, but has to be manually instantiated by applications, which do not use this subsystem. This class does not provide specific methods, its API is defined by the base class *CNetDevice*.

9.5.6 CBcm4343Device

```
#include <wlan/bcm4343.h>
```

class **CBcm4343Device** : public *CNetDevice*

This class is a driver for the BCM4343x WLAN interface device of the Raspberry Pi 3, 4 and Zero (2) W. It has to be instantiated manually, and is normally used together with the class *CNetSubSystem* from the *TCP/IP networking* subsystem and the class *CWPASupplicant* from the submodule hostap. This class provides the interface, defined in its base class *CNetDevice*, and additional methods, which are needed to manage the association with a WLAN access point (AP). The following description covers only the methods, which are specific to this class.

CBcm4343Device::**CBcm4343Device**(const char *pFirmwarePath)

Creates an instance of this class. `pFirmwarePath` points to the path, where the firmware files for the WLAN controller are provided (e.g. "SD:/firmware/").

boolean *CBcm4343Device*::**Initialize**(void)

Initializes the WLAN controller and driver. Returns `TRUE` on success.

void *CBcm4343Device*::**RegisterEventHandler**(TBcm4343EventHandler *pHandler, void *pContext)

Registers the event handler `pHandler`, which is called on some specific WLAN events (e.g. disassociation from AP). `pContext` is a user pointer, which is handed over to the event handler. `pHandler` can be 0 to unregister the event handler.

boolean *CBcm4343Device*::**Control**(const char *pFormat, ...)

Sends the device specific control command `pFormat` with optional parameters to the WLAN device driver. Returns `TRUE` on success.

boolean *CBcm4343Device*::**ReceiveScanResult**(void *pBuffer, unsigned *pResultLength)

Polls for a received scan result message. `pBuffer` is a pointer to a buffer, where the message will be placed. The buffer must have the size *FRAME_BUFFER_SIZE*. `pResultLength` is a pointer to a variable, which receives the valid message length. Returns `TRUE`, if a message is returned in the buffer, or `FALSE` if nothing has been received.

const *CMACAddress* **CBcm4343Device*::**GetBSSID**(void)

Returns the BSSID of the associated AP.

boolean *CBcm4343Device*::**JoinOpenNet**(const char *pSSID)

Joins the open WLAN network with the SSID `pSSID`. Returns `TRUE` on success.

boolean *CBcm4343Device*::**CreateOpenNet**(const char *pSSID, int nChannel, bool bHidden)

Creates an open WLAN network (AP mode) with the SSID `pSSID` on channel `nChannel`. The SSID is hidden, if `bHidden` is `TRUE`. Returns `TRUE` on success.

9.6 Other devices

This section covers some device driver classes, which do not belong to other groups of devices. These classes have their own interface and are not derived from the class *CDevice*.

9.6.1 CBcmFrameBuffer

```
#include <circle/bcmframebuffer.h>
```

class **CBcmFrameBuffer**

This class is a driver for the frame buffer device(s), provided by the firmware of the Raspberry Pi. The Raspberry Pi 4, 400 and the Compute Module 4 support multiple frame buffer devices, all other models only one. A frame buffer is basically an address range in main memory, which is continuously read by the firmware in background, to be displayed on a HDMI or composite TV display. Writing to this memory address range modifies the displayed image. The Raspberry Pi firmware supports frame buffers with different widths, heights and depths of the pixel information. If one wants to display text in a frame buffer, the characters must be formed from a character generator in the software. The firmware does not support text displays on its own.

Note: To be able to use more than one frame buffer device, the option `max_framebuffers=N` (N > 1) is required in the file *config.txt* on the SD card.

CBcmFrameBuffer`::`**CBcmFrameBuffer**(unsigned nWidth, unsigned nHeight, unsigned nDepth, unsigned nVirtualWidth = 0, unsigned nVirtualHeight = 0, unsigned nDisplay = 0, boolean bDoubleBuffered = FALSE)

Constructs a frame buffer device object with `nWidth` * `nHeight` pixels. If both parameters are zero, the frame buffer is automatically created with the default size, which is normally the maximum supported size of the connected display. Each pixel has a depth of `nDepth` bits (4, 8, 16, 24 or 32).

The memory range of the frame buffer may be larger than the displayed physical display size. This can be used to quickly switch the displayed image (see *SetVirtualOffset()*). The optional virtual display size is `nVirtualWidth` * `nVirtualHeight` pixels. If `bDoubleBuffered` is `TRUE`, the virtual display height is automatically set to twice the physical display size, if `nVirtualWidth` and `nVirtualHeight` are specified as 0.

`nDisplay` is the zero-based ID number of the frame buffer device, which is transferred to the firmware to select a specific display on the Raspberry Pi 4, 400 and the Compute Module 4.

void *CBcmFrameBuffer*`::`**SetPalette**(u8 nIndex, u16 nRGB565)

void *CBcmFrameBuffer*`::`**SetPalette32**(u8 nIndex, u32 nRGBA)

Set the entry `nIndex` of the color palette to `nRGB565` or `nRGBA`. The color palette is only used in in 4-bit or 8-bit pixel depth mode. The color palette must be set before *Initialize()* is called, but can be updated later.

PALETTE_ENTRIES

The maximum number of entries in the color palette in 4-bit or 8-bit depth mode (256). `nIndex` must be below this.

boolean *CBcmFrameBuffer*`::`**Initialize**(void)

Initializes the frame buffer device and starts the display. Returns `TRUE` on success.

Note: This method does succeed on Raspberry Pi 1-3 and Zero, even when there is no display connected. On the Raspberry Pi 4, 400 and Compute Module 4 this method fails in this case.

---

u32 *CBcmFrameBuffer*::**GetWidth**(void) const

u32 *CBcmFrameBuffer*::**GetHeight**(void) const

u32 *CBcmFrameBuffer*::**GetVirtWidth**(void) const

u32 *CBcmFrameBuffer*::**GetVirtHeight**(void) const
Return the physical or virtual size of the frame buffer in number of pixels.

u32 *CBcmFrameBuffer*::**GetPitch**(void) const
Returns the size of one pixel line in memory in number of bytes and may contain padding bytes.

u32 *CBcmFrameBuffer*::**GetDepth**(void) const
Returns the size of one pixel in memory in number of bits.

u32 *CBcmFrameBuffer*::**GetBuffer**(void) const

u32 *CBcmFrameBuffer*::**GetSize**(void) const
Return the address and total size of the frame buffer in main memory.

boolean *CBcmFrameBuffer*::**UpdatePalette**(void)
Updates the color palette, after modifying it using *SetPalette()* or *SetPalette32()*. Returns TRUE on success. This method should be used only with a pixel depth of 4 or 8 bits.

boolean *CBcmFrameBuffer*::**SetVirtualOffset**(u32 nOffsetX, u32 nOffsetY)
Sets the offset of the top-left corner of the physically displayed image in a larger virtual frame buffer to [nOffsetX, nOffsetY]. Returns TRUE on success.

boolean *CBcmFrameBuffer*::**WaitForVerticalSync**(void)
Waits for the next vertical synchronization (VSYNC) blanking gap. Returns TRUE on success.

boolean *CBcmFrameBuffer*::**SetBacklightBrightness**(unsigned nBrightness)
Sets the backlight brightness level of the display to nBrightness. This has been tested with the Official 7" Raspberry Pi touchscreen only. The brightness level can be about 0..180 there. Returns TRUE on success.

static unsigned *CBcmFrameBuffer*::**GetNumDisplays**(void)
Returns to number of available displays, which is always 1 on models other than the Raspberry Pi 4, 400 or Compute Module 4.

9.6.2 CBcmRandomNumberGenerator

```
#include <circle/bcmrandom.h>
```

class **CBcmRandomNumberGenerator**
This class is a driver for the built-in hardware random number generator.

u32 *CBcmRandomNumberGenerator*::**GetNumber**(void)
Returns a 32-bit random number.

Note: Generating a random number takes a short while. For generating a large number of random numbers, you should use a polynomial random number generator, and seed it using this hardware random number generator.

9.6.3 CBcmWatchdog

```
#include <circle/bcmwatchdog.h>
```

class **CBcmWatchdog**
This class is a driver for the built-in watchdog device. It can be used to automatically restart a Raspberry Pi computer after program failure, or to restart it immediately from a specific partition.

void *CBcmWatchdog*::**Start**(unsigned nTimeoutSeconds = *MaxTimeoutSeconds*)
Starts the watchdog, to elapse after `nTimeoutSeconds` seconds. The system restarts after this timeout, if the watchdog is not re-triggered before.

const unsigned *CBcmWatchdog*::**MaxTimeoutSeconds** = 15
Is the maximum timeout in seconds.

void *CBcmWatchdog*::**Stop**(void)
Stops the watchdog. It will not elapse any more.

void *CBcmWatchdog*::**Restart**(unsigned nPartition = *PartitionDefault*)
Immediately restarts the system from the SD card partition with the number `nPartition`, with these special values:

const unsigned *CBcmWatchdog*::**PartitionDefault** = 0

const unsigned *CBcmWatchdog*::**PartitionHalt** = 63
`PartitionHalt` halts the system, instead of restarting it.

boolean *CBcmWatchdog*::**IsRunning**(void) const
Returns `TRUE`, if the watchdog is currently running.

unsigned *CBcmWatchdog*::**GetTimeLeft**(void) const
Returns the number of seconds left, until a restart will triggered.

CHAPTER

TEN

APPENDICES

10.1 Libraries

This appendix lists the libraries, which are provided by the Circle project.

10.1.1 Base libraries

The base libraries will be built using `./makeall` from Circle's project root.

Library lib/...	Description	Depends on lib
libcircle.a	Basic system services and drivers	
usb/libusb.a	USB host controller and class drivers	circle, input, fs
input/libinput.a	Generic input device services	circle
fs/libfs.a	Basic file system services (partition manager)	circle
fs/fat/libfatfs.a	FAT file system driver[1]	circle, fs
sched/libsched.a	Cooperative multi-tasking support	circle
net/libnet.a	TCP/IP networking	circle, sched

10.1.2 Add-on libraries

Add-on libraries will be built using `make` from the target directory. This appendix lists only a subset of the available add-on libraries. All provided add-on modules are listed here.

Library addon/...	Description
SDCard/libsdcard.a	EMMC and SDHOST SD card drivers
fatfs/libfatfs.a	FatFs file system modulePage 29, 1
Properties/libproperties.a	Property file (.ini) support
linux/liblinuxemu.a	Linux kernel driver and pthread emulation
vc4/vchiq/libvchiq.a	VCHIQ interface driver
vc4/sound/libvchiqsound.a	VCHIQ (HDMI) sound driver
ugui/libugui.a	uGUI graphics library
lvgl/liblvgl.a	LVGL graphics library

These libraries provide accelerated graphics support for the Raspberry Pi 1-3 and Zero (32-bit only) in *addon/vc4/interface/*:

[1] The file system support in the base libraries is restricted (no subdirectories, short file names). The FatFs file system module in *addon/fatfs/* provides full function support.

- bcm_host/libbcm_host.a
- khronos/libkhrn_client.a
- vmcs_host/libvmcs_host.a
- vcos/libvcos.a

10.2 System data types

This appendix lists the system data types, which are defined and used by Circle. You can also include <stdint.h> to use POSIX types. This file is provided by the toolchain and is not available, if your application is built with STDLIB_SUPPORT = 0.

```
#include <circle/types.h>
```

Type	Description
u8	8-bit unsigned value
u16	16-bit unsigned value
u32	32-bit unsigned value
u64	64-bit unsigned value
s8	8-bit signed value
s16	16-bit signed value
s32	32-bit signed value
s64	64-bit signed value
uintptr	unsigned value with the size of a pointer
intptr	signed value with the size of a pointer
size_t	count of bytes, result of the sizeof operator
ssize_t	count of bytes or an error value
boolean	can be TRUE or FALSE

Note: boolean is a synonym for the standard type bool, which can be used instead, with the values true or false. The definition of boolean has historical reasons, but is still used for an uniform source code.

10.3 Macros

This appendix lists the C-macros, which are defined globally by the Circle build system and which can be used in applications for conditional compiling:

Macro	Description
__circle__	Circle version number (e.g. 440400 for Circle 44.4, patch level 0)
AARCH	ARM architecture (32 or 64)
RASPPI	Major Raspberry Pi model version (1, 2, 3 or 4)
STDLIB_SUPPORT	Standard library support level[1] (0, 1, 2 or 3)
NDEBUG	Not defined in checked builds (default)

[1] See: doc/stdlib-support.txt

10.4 Analyzing exceptions

This appendix explains, how system abort exceptions can be analyzed. The output from the program in section *A more complex program* is used for this purpose. It looks like this:

```
logger: Circle 44.3 started on Raspberry Pi Zero W
00:00:01.00 timer: SpeedFactor is 1.00
00:00:01.00 kernel: An exception will occur after 15 seconds from now
00:00:02.00 kernel: Time is 2
00:00:03.00 kernel: Time is 3
00:00:04.00 kernel: Time is 4
...
00:00:14.00 kernel: Time is 14
00:00:15.00 kernel: Time is 15
00:00:16.00 except: stack[7] is 0xEFBC
00:00:16.00 except: stack[8] is 0xF04C
00:00:16.00 except: stack[11] is 0x11304
00:00:16.00 except: stack[13] is 0x113C4
00:00:16.00 except: stack[23] is 0x11408
00:00:16.00 except: stack[25] is 0x108E4
00:00:16.00 except: stack[31] is 0xE834
00:00:16.00 except: Prefetch abort (PC 0x500000, FSR 0xD, FAR 0x500000,
        SP 0x237F80, LR 0xEE7C, PSR 0x20000192)
```

If you want to detect the instruction, which caused the exception, you can open the file *kernel*.lst* and search for the address in *PC* (Program Counter). Because this is an invalid address outside the kernel image, you will not find it here, but *LR* (Link Register) specifies the address, from where `TimerHandler()` had been called (0xEE7C). The respective address is located at the beginning of a line in *kernel*.lst* with a trailing colon:

```
0000edd0 <CTimer::PollKernelTimers()>:
    edd0:       e92d41f0        push    {r4, r5, r6, r7, r8, lr}
...
    ee64:       e3530000        cmp     r3, #0
    ee68:       0a000011        beq     eeb4 <CTimer::PollKernelTimers()+0xe4>
    ee6c:       e1a00004        mov     r0, r4
    ee70:       e5942010        ldr     r2, [r4, #16]
    ee74:       e594100c        ldr     r1, [r4, #12]
    ee78:       e12fff33        blx     r3
==> ee7c:       e1a00004        mov     r0, r4
    ee80:       e3a01014        mov     r1, #20
...
```

Thus `TimerHandler()` had been called by the instruction "blx r3", the preceeding instruction of the given address.

The several listed addresses from the *stack[]* allow to do a backtrace, but not every shown address needs to be valid. Just search for the addresses in the *kernel*.lst* file, starting with the first one, and you will get the information, which function has been called from which other function (inside-out).

Note: Raspberry Pi is a trademark of Raspberry Pi Trading.

INDEX

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

W