Contents

Introduction

LibK is a lightweight firmware development library that is designed for flash based, resource constrained microcontrollers that are not able to run a full modern operating system. LibK uses protothread-based, cooperative multitasking to make all I/O operatins non-blocking and to make the most of the already constrained CPU by never wasting any clock cycles waiting for an I/O operation to complete.

Included in LibK you will find many device drivers that make it possible to use the supported devices in your project directly without having to implement the device drivers yourself. The device drivers have also been rewritten by me to use cooperative multitasking so that no device can ever slow down your application just because it is slow.

Using LibK you can build portable applications for AVR-8, STM32 and PC.

Gettings started: compiling and flashing

LibK build system is originally designed for building on Ubuntu linux so it may require modifications if you want to build on a different system. The build process uses Makefiles that it includes from all of the subfolders. Each subfolder has it's own Makefile, KConfig file and a README.md (docs) file. All of these files are aggregated by make and then compiled into their respective targets.

To compile LibK you will also need appropriate gcc toolchain for your hardware target.

- avr-gcc and avr-libc for compiling for atmega, arduino, avr32
- arm-none-eabi-gcc for compiling for sam3, stm32 and other arm architectures
- pandoc for compiling this PDF from all readme files (although you should have this PDF included)
- clang (scan-build) for enabling static analysis of the code during the build
- lib-curses for compiling menuconfig (for editing libk configurations)
- avrdude for flashing avr chips

On Ubuntu:

```
sudo apt-get install gcc-avr gcc-arm-none-eabi avr-libc pandoc libncurses5-dev avrdude
```

Build everything:

```
make buildall
```

This will build all supported architectures, produce corresponding libraries and compile examples as well.

The above will in fact invoke build commands for each target. For building a specific target library use BUILD variable that you pass to MAKE like this:

```
make BUILD=arm-stm32f103
```

The parameter to this command should be the same string as the config file name of the corresponding config in the "configs" folder. The build script will then include appropriate source files relevant only for that target and compile them into a corresponding static library.

Installing on your hardware

Note: Depending on how your target flashing process works, these instructions may not apply to you. In that case, find out how to actually burn a binary file on your architecture from the documentation of your target device.

All libk programs compile to binary elf files that are then converted to either a binary blob or a hex file using appropriate objdump utility for the target platform. This resulting hex or bin file can then be directly passed to the flashing software and transferred to the target chip. Usually this transfer is done over a usb to serial converter.

- Avr platform uses tool avrdude and expects avr to be connected to usbasp programmer. For flashing using a different programmer you will need to modify the makefiles (in examples folders).
- STM32 uses builtin ST bootloader and the stm32flash utility. You will need to explicitly boot the chip into the bootloader mode by holding boot0 low at poweron. Some firmwares also support booting into the bootloader from software (check the fst6-demo example for this can be done. It implements a firmware upgrade command).

When building your own software, you will usually link libk into your project and then flash using your standard flashing process. For the examples provided inside libk, there are however makefile targets specified for flashing each of the examples. All examples are built using “make BUILD=(target) build-(example name)” pattern and installed using the corresponding “make BUILD=(target) build-(example name)” pattern.

```
make BUILD=arm-stm32f100mdvl build-fst6-demo
make BUILD=arm-stm32f100mdvl install-fst6-demo
```

Note: if you get a “permission denied” when trying to access your serial port as normal user in linux, then make sure to add yourself to the “dialout” group.

Supported architectures

Manufacturer	Chip	status
Atmel	ATMega328p	my primary focus right now
Atmel	AT91SAM3 ARM	my secondary focus but now yet fully supported
ST	STM32F103	peripheral library is included, but device interfaces not implemented yet

Device driver support

Device drivers in libk operate on a higher level than architecture code. So they are largely architecture agnostic. Device drivers typically use interfaces to access services provided by the architecture and they can also export interfaces to other drivers in order to eliminate dependencies between drivers on each other.

Device class	Device model	Support	Interfaces used	Interfaces expo
Board	Multiwii V2.5	Supported	`flight_controller` interface	
Board	Arduino Pro Mini	Supported	Architecture ATMega328p	
Board	Stm32f103 development board	planned		
Board	Crius ATmega256 flight control	planned		
Crypto	AES 256	Supported		
Display	ILI9340	Supported	parallel_interface and serial_interface	
Display	Parallel LCD	Supported	parallel_interface	
Display	7 segment led	Supported	parallel_interface	
Display	8x8 Led matrix	Supported	parallel_interface	
Display	ssd1306 OLED	Supported	i2c_interface	
Filesystem		Planned		
HID	WiiNunchuck	Supported	i2c_interface	
IO	74HC165	Supported	serial_interface, parallel_interface	
IO	74HC4051	Supported	parallel_interface	
IO	74HC595	Supported	serial_interface, parallel_interface	
IO	PCF8574	Supported	i2c_interface	planned: parall

Device class	Device model	Support	Interfaces used	Interfaces expo
NET	ENC28J60 Supported	serial_interface	planned: i2c_interface	
NET	TCPIP	Supported	i2c_interface	planned: serial_
RADIO	NRF24L01	Supported	serial_interface	planned: i2c_i
SENSOR	ACS712	Supported	analog_interface	
SENSOR	ADXL345	Supported	i2c_interface	
SENSOR	AMT345	Supported	analog_interface	
SENSOR	BH1750	Supported	i2c_interface	
SENSOR	BMP085	Supported	i2c_interface	
SENSOR	DHT11	Supported	parallel_interface	
SENSOR	DS18B20	Supported	parallel_interface	
SENSOR	FS300A	Supported		
SENSOR	HCSR04	Supported	parallel_interface	
SENSOR	HMC5883L	Supported	i2c_interface	
SENSOR	L3G4200D	Supported	i2c_interface	
SENSOR	LDR	Supported	analog_interface	
SENSOR	MMA7455	Supported	i2c_interface	
SENSOR	MPU6050	Supported	i2c_interface	
SENSOR	NTCTEMP	Supported	analog_interface	
SENSOR	TSL235	Supported		
TERMINAL	VT100	Supported	framebuffer_interface	serial_interface

Drivers are tested on avr. Other architectures may not have full drivers support until there is stable architecture abstraction layer written for that architecture.

(devices marked as quarantine are drivers that have not been updated yet after changes to the core api. Code has been included in the source tree but it has not yet been updated to work nicely with other facilities of libk)

Design goals

- To maintain small memory footprint and use minimal amount of ram
- To take advantage of as much of the (limited) CPU power as possible
- To maintain high portability across all of the supported platforms
- To minimize the amount of code inside the application that needs to be changed to support a new platform.

How the design goals are currently implemented:

- LibK places as much constant data as possible into flash and thus frees up a lot of ram for use in the application
- LibK uses cooperative multitasking inside the kernel and all drivers are written to never block the cpu while waiting for an external event or I/O.

- LibK uses asynchronous I/O at all times, meaning that no read or write operation is ever blocking. This allows for other tasks to happen while a task is waiting for io.
- LibK uses protothreads instead of conventional threads - this means that threads never have any stack and all threads execute inside the same stack space of the main application. It also have many other advantages that are discussed further in this document.

Kernel Architecture

LibK is a lightweight kernel that implements cooperative multitasking using stackless threads. Stackless threads have been chosen because they are the most lightweight kind of threads available and they have very small overhead. There are however both advantages and also some disadvantages to using stackless protothreads.

Advantages of stackless protothreads:

- Allow device drivers to be written such that no CPU cycles are ever wasted waiting for I/O. The device driver can simply return control to the application and wait until the next time it has the chance to run.
- Writing asynchronous tasks becomes a lot easier because they can be written linearly instead of being organized as a complicated state machine.
- No thread is ever interrupted while it is in the middle of some operation. Threads always run to completion. Meaning that we can design our code without having to think about byte level synchronization (device level syncrhonization is required though).
- No stack also means that we never run out of stack space inside a thread.
- Context switching is very fast - current thread method saves it's resume point, returns to libk scheduler, libk scheduler loads the address of the next thread function to run, and calls it.
- Not a problem to have tens of threads for each asynchronous action. Since threads are just normal methods minus the stack, we can have many threads without experiencing significant slowdown.
- Synchronization is much easier because all code that you "see" is atomic until it explicitly releases control to the scheduler. We thus do not have to worry about non atomic memory access.

Disadvantages of this approach:

- Stackless means that no variable on the thread method stack is valid after a thread returns and then resumes again. Although we can easily solve this by maintaining the context inside a separate object to which the thread is attached.
- No preemption also means that a thread can keep CPU to itself for as long as it wishes. It is up to the programmer to release the CPU as quickly as possible. Since device drivers usually use interrupt requests to respond in realtime anyway, this limitation has not proven to be a problem so far.
- No thread can ever call another thread or spawn a new thread. Threads are only single level. All other code called from inside the thread can be considered to execute atomically (except for when it is interrupted by an ISR).
- Data that needs to be saved across multiple thread switches must be stored in memory (this is a problem with all multitasking though).
- Longer response times for tasks - a task can lose cpu for as long as it takes the heavies task to finish. This is solved by programmer explicitly designing threads such that control is periodically released to the scheduler.

A detailed evaluation of libk threading

Stackless threads in libk are designed to solve one specific problem: busy waits. I settled for this approach because it has been the most lightweight solution to this problem. Almost 98% of all embedded code uses busy waits - the standard way to solve this problem is by implementing a scheduler that is able to interrupt a currently running task in the middle of it's busy loop and switch to a different task. This however also comes with a lot of subtle problems that result in much more synchronization code all around the application to ensure atomic access to shared data.

With protothreads, it is instead possible to minimize the amount of locking and synchronization necessary, while at the same time to enjoy a healthy degree of multitasking where CPU rarely is just idly spinning inside some delay loop.

The main area where this kind of multitasking really is useful is device drivers that do a lot of I/O. I/O operations are by far the greatest bottleneck in most embedded systems that don't use multitasking and instead resort to idly waiting for an I/O operation to complete. LibK solves this problem by doing minimal caching of data and also by never waiting for an I/O operation to complete and instead letting another task run while I/O is in progress.

Another attractive feature of libk threading is that it is completely implemented in software - meaning that it will work the same on all hardware. It is after all just an array of "update" methods that the kernel schedules periodically.

I have found that I could improve performance with protothreading almost 100x. When I eliminated all busy delay loops in the device drivers I have found that my application was able to run a lot faster and also it has become much more responsive. I have not howerver noticed a significant memory overhead. By far the main memory overhead (which is also a necessary evil) is caching data in memory so that it can be retained while another thread has control of the CPU. Most drivers use caching in one way or another. Device interfaces =================

How to device access works

For the sake of being light weight, the on chip devices all have static methods that are used for accessing each device. You can use macros to construct code for accessing a low level on chip device and most of these macros are device in arch/*.h files.

Thus we can write spi0_init() and spi0_writereadbyte() to interact with the spi0 interface on the chip. All this is fine, until we want to configure another device driver to use a speciffic spi device. Of course we could write the driver so that it has the spi device hardcoded and always uses one speciffic output device - this is fine if we only have one instance of the driver - but boards can have several instances of the same device connected to completely different spi ports. And this needs to be configured.

This does require some kind of way to reference an on chip device that can be specified dynamically in code. Generating separate code for each instance of the device no longer provides much benefit - so we need to use some extra memory to do this. The way this is solved right now is through the use of interfaces. Every type of low level device has an interface struct that can be optionally used to access that device from a driver. This interface we can construct dynamically. The overhead is a little memory used for storing the function pointers and a small overhead of calling a function pointer but this is negligible unless you have some really really small device. And also the library allows you to actually not use this interface at all if you don't want to - however higher level drivers use it because it adds a configurability benefit to the driver so that we can easily define multiple instances of a driver for a board.

All device instance data is stored in a device specific struct along with any interfaces to lower level devices that the driver uses to talk to the device. This way we can configure a device in the board file to use i2c port X and address Y to talk to a device and then the driver code can use the constructed interface to access the device without knowing exactly which i2c port and address it is using. The driver does not need this data - it only needs a reference to an i2c interface that is already preconfigured to use specific settings for that board.

The kind of code structure we are aiming to achieve is that all on board devices have to be configured in the board driver. All application code then uses structs defined in the board struct to talk to all devices accessible to it.

Architecture layer

- Martin K. Schröder - info@fortmax.se

Currently supported architectures

VENDOR	Model	Status
ATMEL	ATMega328p	yes
ATMEL	AT91SAM3	yes
ST	STM32F103	part

When you want to implement support for a new chip, what you would need to do is implement all of the interfaces defined in arch/interface.h file. We will take a look at these interfaces in turn below. But first a little on how things are put together.

The whole arch layer for an architecture can be included into your application by including arch/soc.h file. It will automatically include needed files according to include/autoconfig.h file (generated by 'make menuconfig')

```
#include <arch/soc.h>
```

This layer consists of following devices:

- ADC - analog to digital converters
- DAC - digital to analog converters
- GPIO - on chip general purpose io driver
- PWM - pulse width modulation peripherals

- SPI - on board serial peripheral interface peripherals
- I2C/TWI - i2c interfaces

- UART - asynchronous serial interfaces
- CAN - controller area network interfaces

The arch layer is built around the idea of being really fast. Because this is the absolutely lowest layer that is closest to hardware, the methods of this layer get called a lot of times. This layer is responsible for providing a human readable interface to chip register operations.

We would typically implement this layer by wrapping register operations in macro expressions. You can take a look at how this is done in one of the header files found in for instance arch/avr/m328p folder. All cpu operations are wrapped in macros. You don't have to use macros on higher layers, but this layer is usually very simple (ie all "method" calls in this layer really just write values to some memory location) so we actually want to do this with macros. You can for instance look at the implementation of gpio interface in arch/avr/m328p/gpio.h

Devices and their interfaces

There are typically two ways to access devices on the soc. You can either call the arch layer directly (ie call the macros) or you can use generic interfaces that the arch layer can create for you in order to access the devices generically. Generic interfaces are designed to allow interchangable use of many different types of devices that can provide the same kind of service. For example, libc FILE* handle is a type of generic interface. It has get() and put() function pointers that get called by getc/putc. This way you can use FILE* to access a file on any filesystem, you can access a device, you can access a fifo pipe - all thought exactly the same interface.

This is exactly what we want to have for our device drivers as well!

But there is one problem: the devices that we are working with are all flash based controllers that usually have little memory - both flash and ram. We can not use really rich interfaces like the linux kernel. And we don't want to force the user to use interfaces. We want to provide interfaces simply as an essential feature for enabling driver reuse and for adding a great deal of flexibility for how device drivers are used.

You should think of interfaces as "services" that the device provides as opposed to identifying a specific device. If we make all of our devices "service"-oriented, we can reuse device drivers in many different contexts and pipe data from one driver to another. This is exactly the same philosophy that is used by linux commands where you can create more complex commands easily by simply piping results of one command to the next.

Interface	Methods	Usage
analog_interface	read_pin	Used for any device th
parallel_interface	write_pin, write_word, read_pin, read_word, configure_pin, get_pin_status	GPIO interface is for i
pwm_interface	set_channel_duty, set_channel_period	PWM interface is for c
packet_interface	begin, end, write, read, sync, packets_available	A packet interface is si
serial_interface	get, put, getn, putn, flush, waiting	A serial interface is an

Here are some examples of what interfaces different core devices can export:

Click on the device name to read more!

Device	Interfaces exported
ADC	analog_interface
GPIO	parallel_interface
PWM	pwm_interface
SPI	serial_interface
I2C	packet_interface
UART	serial_interface

Analog to digital converters

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/adc.h
- arch/adc.h

These devices are expected to provide a service of reading an analog signal on one of available pins and reporting the value as a digital 16 bit number. Adc implementation is limited to 256 channels (although can easily be extended in the future if one needs more). An adc driver should typically only control one type of device or a few types of devices that are very similar to each other. If a device differs too much, a separate driver should be created.

Public interface

| struct analog_interface method | Description | |——————————————|————-| int16_t (*read_pin)(adc, pin_number) | reads adc pin and returns it's integer representation.

Implementation macro guidelines

Method	Description
adc0_enable()	provides means to turn on the adc peripheral
adc0_disable()	will turn off adc to save power
adc0_init_default()	configures adc to default settings (typically internal vref, left alignment and slow
adc0_set_vref(adcref)	must provide means of setting voltage reference source for the conversion.
adc0_set_prescaler(adc_clock)	sets adc clock prescaler
adc0_set_alignment(adc_align)	must provide means of setting alignment of result (left or right)
adc0_interrupt_on()	turns on adc interrupt
adc0_interrupt_off()	turns off adc interrupt
adc0_set_channel(adc_chan)	provides means to tell adc peripheral which channel to sample
adc0_get_channel()	evaluates to whatever channel we are currently sampling
adc0_start_conversion()	kicks off adc conversion
adc0_conversion_in_progress()	checks if a conversion is currently in provress
adc0_wait_for_completed_conversion()	returns when current conversion has finished
adc0_read() (uint16_t)	simply waits for previous conversion to complete first and then returns sampled
adc0_read_immediate(chan)	kicks off a conversion, waits for it to complete and returns the value for supplied
adc0_set_mode(adc_mode)	provides access for setting different adc modes (such as manual or automatic con

Macros should be provided for every peripheral separately if multiple devices are available - ie adc0, adc1 etc..

Typical direct usage

```
adc0_init_default();
printf("ADC value: %d\n", adc0_read_immediate(channel_id));
```

For other options it is best to check out implementation specific file in arch///adc.h

General purpose parallel interface

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/gpio.h
- arch/gpio.h

A general purpose io is any kind of parallel device where it is possible to address each bit separately and which can be one or more bytes wide. It must be possible to write individual bits separately and also to write full “words” which are as wide as the width of the gpio interface. For internal devices it may be convenient to group all gpio interfaces together so that the user can use a continuous pin indexing scheme (ie for 3x8 bit on board parallel ports the pins can be numbered 0-24 in order to provide a generic way to access the port).

Public interface

| parallel_interface | Description | |——————|————-| void (*write_pin)(struct parallel_interface* self, uint16_t pin_number, uint8_t value) | sets pin of the output port either high or low. uint8_t (*read_pin)(struct parallel_interface* self, uint16_t pin_number) | used to read an input pin. Always returns 0 when reading an output pin. uint8_t (*configure_pin)(struct parallel_interface* self, uint16_t pin_number, uint16_t flags) | configure pin to be input, output, pulled up, pulled down etc. Not all options may be supported by implementation. Function returns 0 on success and 1 on failure. uint8_t (*get_pin_status)(struct parallel_interface* self, timestamp_t *t_up, timestamp_t* t_down) | used to get status of the pin. Interface allows the implementation to track pin changes in the background and report values to the user of the interface. Returns pin status (went high/low etc.) and writes time in ticks for when pin went high, or went low. You should use time_ticks_to_us() to get interval in us. ticks is a value used for tracking cpu ticks. It only denotes a specific point in time. It is currently easier to keep track of time of the pin changes in the implementation rather than leaving this task to the user. This function is very useful for writing libraries that need to measure pulse intervals. The values of t_up and t_down must be updated by the implementation as way of measuring when the pin went high and when it goes low. Default value returned must be 0. uint8_t (*write_word)(struct parallel_interface* self, uint16_t addr, uint32_t value) | used to write byte or int to an io address. If you have pins PA0, PA1 .. PA7 and your registers are 8 bit long then writing to addr 0 should write all of PA pins at the same time. For implementations with larger registers, more bits may be written. This method is used to write multiple bits in one operation. uint8_t (*read_word)(struct parallel_interface* self, uint16_t addr, uint32_t *output) | user to read word from an io register. The size of the word depends on the implementation. It may be 8 bit or 16 bit or 32 bit. The size is equivalent to the full size of io registers of implementation. addr is the index of the io register. Implementation must check this value for a valid range and return error if it is invalid.

Implementation macro guidelines

gpio method	Description
gpio_init()	initializes the hardware gpio peri
gpio_configure(pin, fun)	provides means to configure pins
gpio_write_word(addr, value)	writes all bits at once on a port
gpio_read_word(addr, value)	reads all bits at once from port.
gpio_write_pin(pin, val)	writes 1 or 0 to a specific pin. G
gpio_read_pin(pin)	reads gpio pin
gpio_clear(pin)	clears a pin (some platforms can
gpio_set(pin)	sets a pin
gpio_enable_pcint(pin)	provides means to enable pin ch
gpio_disable_pcint(pin)	disable pin change interrupt for

gpio method	Description
uint8_t gpio_get_status(gpio_pin_t pin, timestamp_t *ch_up, timestamp_t* ch_down)	if implementation provides mean

Example usage

```
    gpio_init();
        while(1){
            gpio_write_pin(GPIO_PB0, 1);
            delay_ms(500);
            gpio_write_pin(GPIO_PB0, 0);
            delay_ms(500);
        }
    }
```

I2C / two wire interface

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/twi.h
- arch/twi.h

Public interface

The public interface used by the i2c bus is packet_interface. This is because i2c is a protocol based state machine that expects some standard data to operate. In this respect it operates just like any packet interface - it sends data buffers and receives data into a buffer. One special case that takes the point home is this: i2c devices sometimes expect a write operation followed by a repeated start signal and then a read operation. This sequence is used to read values from an i2c device. Using packet interface allows us to model this behaviour since it has begin(), end() write() and read() methods. Serial interface would be inappropriate here.

Interface method
void (*begin)(struct packet_interface* self)
void (*end)(struct packet_interface* self)
uint32_t (*write)(struct packet_interface* self, const uint8_t *data, uint16_t max_sz) | writes a packet to the device the max
void (*sync)(struct packet_interface* self)
uint16_t (*packets_available)(struct packet_interface* self)

Implementation macro guidelines

Hardware method
twi0_init(speed)

Hardware method
twi0_begin()
twi0_end(void)
twi0_start_write(uint8_t *data, uint8_t data_sz) | starts a write operation on the bus. This method may return directly bu*
twi0_busy(void)
twi0_success(void)

Example usage

```
char buffer[] = {DEVICE_ADDRESS, 'H', 'e', 'l', 'l', 'o'};
twi0_init(100000);
twi0_begin();
twi0_start_write(buffer, 6);
twi0_start_read(buffer, 6); // will read data into buffer starting at buffer[1]..
twi0_end()
```

PWM peripheral interface

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/pwm.h
- arch/pwm.h

Public interface

PWM interface is for controlling pwm hardware. Usually it would be built in, timer driven pwm channels, but the system does not limit you to just on board pwm channel You could just as easily have a driver for an i2c or spi pwm controller export this kind of interface and this would allow you to pass it to any component that requires a pwm interface (such as for example a motor speed controller).

The basic pulse width modulator interface consists of following methods:

Interface method	Corresponding twi action
uint16_t (*set_channel_duty)(struct pwm_interface* self, uint8_t channel, uint16_t value_us)	Sets channel “on” time to
uint16_t (*set_channel_period)(struct pwm_interface* self, uint8_t channel, uint16_t value_us)	Occasionally it may be b

Implementation macro guidelines

Hardware method	Description
pwm0_enable()	enable pwm channel and configure it to settings most commonly seen in hobby controller context: ie pu
pwm0_set(speed)	sets pulse width in microseconds.

Example usage

pwm0_init(); pwm1_init();

pwm0_set(1500); // half throttle pwm0_set(1000); // zero throttle pwm0_set(2000); // full throttle

SPI peripheral interface

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/spi.h
- arch/spi.h

Public interface

SPI peripherals can be accessed by using standard serial_interface. A method for each peripheral must be provided: spi0_get_serial_interface() that returns struct serial_interface.

Implementation macro guidelines

Hardware method	Description
hwspi0_set_clock(spi_rate)	sets spi clock rate. Implementation must provide appropriate devinces along the
hwspi0_set_mode(spi_mode)	sets spi mode (whether it's sampled on falling or leading edge of the clock. Valid
hwspi0_master()	configure this interface as spi master
hwspi0_slave()	configure this interface as spi slave
hwspi0_order_lsb_first()	sets order to lsb transmitted first (bit order)
hwspi0_order_msb_first()	sets order with msb first
hwspi0_interrupt_enable()	enables spi interrupt
hwspi0_interrupt_disable()	disables spi interrupt
hwspi0_enable()	enables spi interface
hwspi0_disable()	disables spi interface
hwspi0_config_gpio()	configures gpio pins for this spi interface (ie input/outputs etc).
hwspi0_wait_for_transmit_complete()	waits for transmission to finish
hwspi0_error_collision()	checks collision flag (set if data is written before previous data has been sent)
hwspi0_init_default()	initializes spi peripheral with default settings. Usually SPI_MODE0, interrupt d
hwspi0_putc(ch)	writes a character to spi, does not
hwspi0_getc(ch)	reads previously received character. since spi transmits and receives at the same
hwspi0_transfer(ch)	combines write and read into one op. Returns character received.

Example usage

```
hwspi0_init_default();

uint8_t data = hwspi0_transfer('X');
```

Timers and tick counters

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/time.h
- arch/time.h

Intro

It is often necessary for implementation to provide a timer/counter for counting cpu clock cycles because this is the only way to do fairly exact timing when interrupts are enabled. More complex chips usually have a dedicated timer for this and an instruction to read clock counter. But for architectures that do not have this functionality, one of the timers needs to be used.

Usage of clock counter

In the rest of the system clock counter value can be used in many ways - from simple delays to asynchronous events. If you disable this functionality then some drivers may stop working, but ideally they would not even compile. There is a special config field called CONFIG_TIMESTAMP_COUNTER that enables timestamping functionality and drivers that need this functionality should be configured in the corresponding KConfig file to depend on it being set.

Public interface

Timer counter currently does not export interfaces. This may change when drivers would require an abstract timer object. For now, timers are mostly only used directly by drivers in the arch layer. While higher level drivers mostly only use timestamping to calculate delays and timeouts.

Implementation guidelines

Hardware method	Description
timestamp_init()	sets up timestamping counter to start counting. This configures one of the hard
timestamp_now()	gets current timestamp value in ticks. The value is ticks from cpu start. but rem
timestamp_ticks_to_us(ticks)	converts ticks value to microseconds.
timestamp_us_to_ticks(us)	converts microseconds to equivalent number of ticks
timestamp_before(unknown, known)	checks if unknown is before known. This function is required to handle overflow
timestamp_after(a,b)	checks if a is after b, also overflow safe
timestamp_from_now_us(us)	calculates a timeout value a number of microseconds in the future

Hardware method	Description
timestamp_expired(timeout)	checks if a previously set timeout has expired
timestamp_delay_us(timestamp_t usec)	equivalent to busy loop delay, but more exact when interrupts are enabled.
delay_us(usec)	a generic method that can be pinted directly to timestamp_delay_us
timestamp_ticks_since(timestamp)	returns ticks that have elapsed from some timestamp

There are more timer hardware specific methods but they will not be covered here for now. Instead, have a look at implementation specific time.h file to get an idea.

Example usage

```
timestamp_init();
timeout_t timeout = timeout_from_now(1000000); // 1s timeout
while(1){
    // do work
    if(timeout_expired(timeout))
        break; // will stop after 1 second
}
```

UART serial interface

Martin K. Schröder | info@fortmax.se

Files:

- arch/manufacturer/model/uart.h
- arch/uart.h

Intro

As with many other peripherals, a chip can have several uarts. All uarts should export a serial_interface for generic access.

Public interface

Check out definition of struct serial_interface in arch/interface.h

Implementation guidelines

Implementation is as always free to either use buffered interrupt driven uart or simple non buffered uart. The advantage of implementing a buffered uart is that all write operations can return to higher level code much quicker so that application can carry on while uart sends the data in the background. So it is always preferable to at least have a small buffer.

Hardware method	Description

Example usage

It is up to higher level code to provide printf functionality. However, uart code may export a file descriptor for the uart making it possible to use standard fprintf() functionality pretty easily. If this is done, then global uart file descriptors are exported and named uartX_fd for each available uart. This removes the need to write custom printf functions because we can just use libc.

```
uart0_init(32400);

// accessing directly
uart0_putc('X');

// using an interface
struct serial_interface serial = uart0_get_serial_interface();
char buffer[] = "Hello World!";
serial.putn(&serial, buffer, strlen(buffer));

// using file descriptor
fprintf(uart0_fd, "Hello %s\n", "World!");
```

Filesystem support

Currently supported filesystems

Name	What	Status
FAT	Fat filesystem	not ported

Display Drivers

Currently supported displays

Name	What	Status
ILI9340	TFT display	yes
LCDPFF8574	PCF8574 based lcd displays	yes
LEDMATRIX	Led matrix display	not tested
SEVSEG	seven segment display	not tested
SSD1306	monochrome OLED display	yes

Human interface devices

Currently supported input devices

Name	What	Status
WIINUNCHUCK	Wii nunchuck	not tested

External IO peripherals

Currently supported devices

Name	What	Status
L74HC165	Shift in register	yes
L74HC4051	Multiplexer	yes
L74HC595	Shift out register	yes
PCF8574	I2C GPIO expander	yes

Ethernet adapters

Currently supported devices

Name	What	Status
ENC28J60	Ethernet driver IC	not tested
TCPIP	Minimal IP stack	not tested
RFNET	Encrypted radio com	yes

Radio devices

Currently supported devices

Name	What	Status
NRF24L01	Radio module	not tested

TTY Emulation Support

Author: Martin K. Schröder, 2015

Currently supported tty interfaces

TTY drivers need to use generic display interface so that it can be used with any GLCD display.

Name	What	Status
VT100	VT100 interface for a text screen	yes