Pins

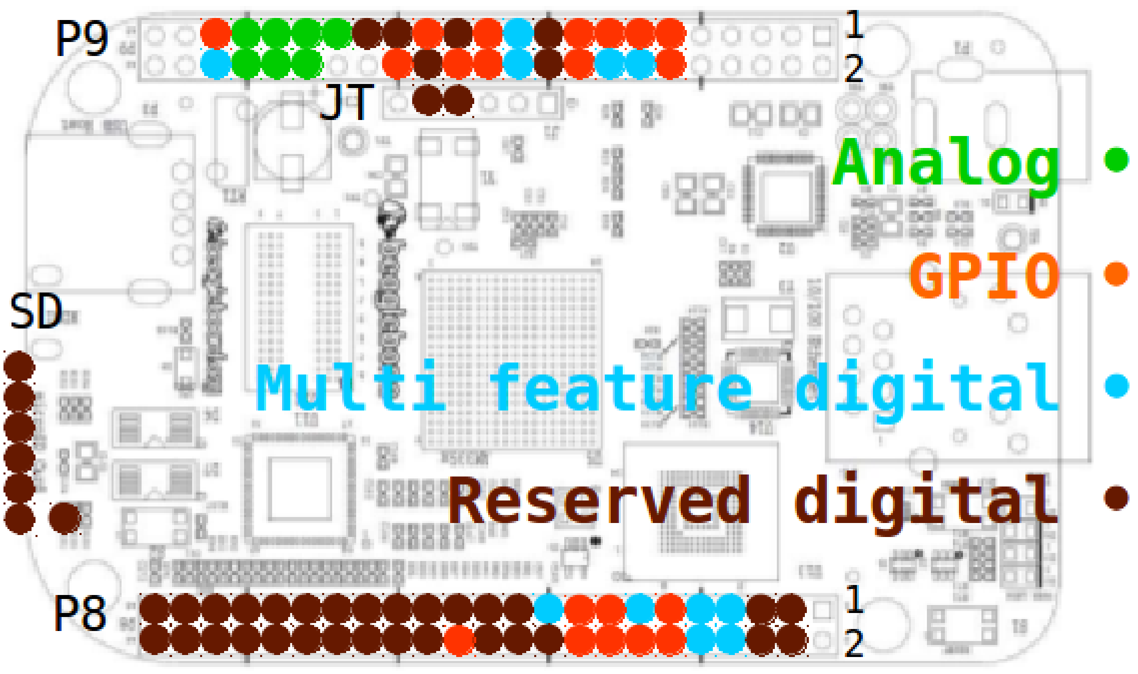
The Beaglebone hardware contains two header connectors, each with 46 pins, 92 pins in total. Just a few are related to special features (such as RESET, GND or power supply lines), while most of them are lines with input / output capabilities, either for analog or digital signals. Some of them are free, others are unfree (used by the system in default configuration). All input / output header pins can get controlled by *libpruio*, as well as further pins on the J1 header and the SD card slot. Furthermore *libpruio* can controll internal lines, not wired to any connector (ie. the onboard user leds, example **sos** demonstrates that).

Table of Contents

Analog
Digital
GPIO
PWM
TIMER
CAP
QEP
Pinmuxing
Custom overlay
Universal overlay
LKM

Analog lines always operate as input. In contrast, digital lines can either operate as input or output, ie. as a simple switch (called general purpose input/output = GPIO). Some digital lines have further features at the same pin, ie. for generating a pulse train. In order to use that features, the pin has to get configured to the matching mode before usage (pinmuxing).

Here's an overview of the Beaglebone Black pins. The settings for other Beaglebone hardware (White, Green, Blue, Pocket-, ...) are different (have more free pins). See the related TRM (technical reference manual) for details.



Header pins on BeagleBone Black boards

The pins are sorted into four categories:

1. Green pins are analog input lines.
2. Orange pins are GPIO only pins. They may provide other features like UART, MMC, HDMI, ... But *libpruio* doesn't support them, so from our point of view they are GPIO only.
3. Blue pins provide beside GPIO further features, like PWM, TIMER, CAP, QEP. Those are rare and valuable, arrange them carefully.

4. Brown pins are used by the BeagleboneBlack operating system, try to avoid them.

For pinmuxing (digital pins category 2 to 4) the connected CPU ball needs to get adressed. Therefor a set of header files are shipped with *libpruio*, which define the CPU ball numbers by the connectors location. Ie. it's more easy for the user specify P8_07 (pin 7 on header P8) then looking up the related ball number 36 (since the ball numbers are not sorted in any order). Here's a table of the pin headers:

Name (Type)	Headers	FreeBASIC	C
White, Green, Black	2x46 headers	**src/pruio/pruio_pins.bi**	src/c_include/pruio_pins.h
PocketBeagle	2x36 headers	**src/pruio/pruio_pins_pocket.bi**	src/c_include/pruio_pins_pocket.h
Blue	indiv. conn.	**src/pruio/pruio_pins_blue.bi**	src/c_include/pruio_pins_blue.h

For Python programming language all pin defines are included in the binding file `src/python/libpruio/pruio.py`.

When planing a new project, concentrate on the category 1 to 3 pins, and try to avoid the brown pins. Some of them are used to control the boot sequence and mustn't be connected at boot-time. Others are reserved to be used by the system (ie. for HDMI or MCASP), but they can get freed by re-configurations of the boot sequence, so that you can use them. It's beyond the scope of this documentation to cover all details. Find further information in the SRM (system reference manual) shipped with your board.

Note

In the following tables the column named BB contains pin specifications for BeagleBone boards while PB contains the similar PocketBeagle board specification.

Some header pins (`P9_41` and `P9_42`) are connected to two CPU balls. Both CPU balls must not be set in contrary output states, in order to avoid hardware damages.

In the first part this chapter explains the different features available on the header pins, and serves detailed information about the hardware limits. In the second part in section **Pinmuxing** you'll learn how to get a pin in your custom mode.

Analog

Analog lines work always as input line. Analog output isn't supported by the Beaglebone hardware (but can get achieved by a combination of a PWM output and a hardware filter).

Analog inputs operate in the range of 0 to 1V8 on Beaglebone hardware. On PocketBeagle boards some pins operate in the range of 0 to 3V3. The header pins are directly connected to the CPU connectors. There's no overvoltage protection, so in order to avoid hardware damages you mustn't trespass the maximum voltage range.

In addition to the seven analog lines available on the header pins (see table below), *libpruio* can also receive samples from the AIN-7 line, which is internal connected to the board power line (3V3) by a 50/50 voltage divider. This may be useful to measure the on board voltage, in oder to control the power supply.

BB	PB	Description
P9_39	P1_19	AIN-0 (default configuration in step 1
P9_40	P1_21	AIN-1 (default configuration in step 2

P9_37	P1_23	AIN-2 (default configuration in step 3
P9_38	P1_25	AIN-3 (default configuration in step 4
P9_33	P1_27	AIN-4 (default configuration in step 5
P9_36	P2_35 3V3	AIN-5 (default configuration in step 6
P9_35	P1_02 3V3	AIN-6 (default configuration in step 7
	P2_36	AIN-7 (default configuration in step 8

The ADC subsystem samples the input with 12 bit resolution. The minimum input (0V) gets sampled as 0 (zero) and the maximum input (1V8) gets sampled as 4095. In order to make this raw data comparable with other ADC input, *libpruio* can encode to different bit formats. The default encoding is 16 bit. This means the maximum input of 1V8 gets measured as 65520 (= 4095 * 16).

libpruio offers full control over the ADC subsystem configuration. Up to 16 ADC steps can get configured to switch the required input line, specify individual delay values and maybe apply avaraging. The ADC subsystem supports analog input up to a frequency of 200 kHz. This works for up to eight steps. The maximum frequency shrinks when

- avaraging of the samples gets applied, or
- more than eight steps are active, or
- delays (open or sample) are required, or
- a clock devider is active (see **AdcSet::ADC_CLKDIV**).

Here's the strategy to fetch analog input (after calling the constructor **PruIo::PruIo()**)

1. configure ADC steps by calling **AdcUdt::setStep()** (or use default configuration generated by the constructor **PruIo::PruIo()**).
2. specify the run mode, step mask and bit encoding by calling function **PruIo::config()**. In case of RB or MM mode, also set the number of samples (*Samp* > 1) and the measurement frequency (*Tmr*) in this call.
3. Only for RB or MM mode (when *Samp* > 1): start the sampling by calling either **PruIo::rb_start()** or **PruIo::mm_start()**.
4. Get the samples in array **AdcUdt::Value**.

Find further details on analog lines and the ADC subsystem configurations in ARM Reference Guide, chapter 12. Find example code in **io_input.bas**, **oszi.bas**, **rb_file.bas**, **rb_oszi.bas** and trigger.bas.

Digital

Each digital header pin can get configured either in GPIO mode or in one of up to seven alternative modes. The matrix of the possible connections is hard-coded in the CPU logic. Find the subset of the Beaglebone headers described in the BBB Software Reference Guide, chapter 7. Or find the complete description in the CPU documentation Sitara AM335x ARM Cortex-A8 Microprocessors Guide, chapter 2.2 .

Before a digital header pin gets used, *libpruio* checks its mode. When the current setting matches the required feature, *libpruio* just continues and operates the pin. Otherwise it tries to change the pinmuxing appropriately, first. This needs pinmuxing capabilities, see section **Pinmuxing** for details. Otherwise you've to ensure that all used digital header pins are in the appropriate mode before you start the program (so that the initial checks succeed).

In any case, all digital lines operates in the range of 0 to 3V3. An input pin returns low in the range of 0 to 0V8 and high in the range of 1V8 to 3V3. In the range inbetween the state is undefined. The maximum current on an output pin shouldn't exceed 6 mA.

Mode	LOW	HIGH	Notice
output	0	3V3	max. 6 mA
input	0 to 0V8	1V8 to 3V3	undefined from 0V8 to 1V8

On BeagleBone boards two of the header pins are connected to multiple CPU balls. Those are P9_41 and P9_42. When changing the pinmuxing of any of the related CPU balls, it must be ensured that the second CPU ball doesn't operate in a contrary output state. You need not care about that, *libpruio* handles that accordingly.

On PocketBeagle boards there are also double pins, but one of the CPU balls is a AIN input. There is no safety issue here, but you may measure by the analog input some digital output from the connected digital pin.

Depending on the pin mode the pin can act either as input or output pin. Here's the strategy to fetch digital input (after calling the constructor **PruIo::PruIo()**)

- configure the pin by the related function (**GpioUdt::config()**, CapUdt::config() or QepUdt::config()).
- start the main loop by calling **PruIo::config()**
- read the current pin state by calling the related function (**GpioUdt::Value()**, CapUdt::Value() or QepUdt::Value()).

And here's the strategy to set digital output (after calling the constructor **PruIo::PruIo()**)

- start the main loop by calling **PruIo::config()**
- set the required pin state by calling the related function (**GpioUdt::setValue()** or PwmUdt::setValue()).

Note
Some pins are used to control the boot sequence and mustn't be connected at boot-time.

GPIO

GPIO stands for General Purpose Input or Output. In output mode the program can switch any connected hardware on or off. In input mode the program can detect the state of any connected hardware. *libpruio* can configure and use any digital header pin in one of the following five GPIO modes. (Therefor the universal device tree overlay `libpruio-00A0.dtbo` has to be loaded and the program has to be executed with admin privileges.) Find details on GPIO hardware in ARM Reference Guide, chapter 25.

PinMuxing Enumerator	Function	GpioUdt::Value()
PRUIO_GPIO_OUT0	output pin low (no resistor)	0
PRUIO_GPIO_OUT1	output pin high (no resistor)	1
PRUIO_GPIO_IN	input pin with no resistor	undefined
PRUIO_GPIO_IN_0	input pin with pulldown resistor	0
PRUIO_GPIO_IN_1	input pin with pullup resistor	1

An input pin can get configured with pullup or pulldown resistor, or none of them. Those resistors (about 10 k) are incorporated in the CPU. In contrast, output pins get always configured with no CPU resistor connection by *libpruio* (to minimize power consumption). So the first two modes (PRUIO_GPIO_OUT0 and PRUIO_GPIO_OUT1) use the same pinmuxing. Those modes are predefined in the universal overlay `libpruio-00A0.dtbo` for each claimed header pin.

In order to set a GPIO output just set its state by calling function Gpio::setValue() (after calling the constructor **PruIo::PruIo()**). When you do this after the call to **PruIo::config()**, the state will change immediately. Otherwise it changes after the **PruIo::config()** call.

In order to get a GPIO input configure the pin first by calling function **GpioUdt::config()** (after calling the constructor **PruIo::PruIo()** and before the call to **PruIo::config()**). Then check its state by calling function **GpioUdt::Value()** (after the call to **PruIo::config()**).

Find example code in **button.bas** (input) or **sos.bas** and **stepper.bas** (output).

PWM

PWM stands for Pulse Width Modulated output. This is generating a digital pulse train with a given frequency and duty cycle. Usualy PWM is used to control actuators, ie. to control the speed of a DC engine, the power of an AC motor, or the position of a servo.

PWM is available on a subset of header pins. A PWM pin is configured as an output pin without resistor connection. This pin gets auto-set to high or low state, depending on a counter running on a certain clock rate. When the counter matches specified values, the state of the output toggles. Since PWM output can get generated by different subsystems (and *libpruio* supports many of them), the resolution and the frequency range vary between the pins.

BB	PB	Subsystem	Frequency Range	Notice BBB
P8_07		TIMER-4	0.000010914 to 6e6 Hz	free
P8_09	P1_28	TIMER-5	0.000010914 to 6e6 Hz	free
P8_10	P1_26	TIMER-6	0.000010914 to 6e6 Hz	free
P8_08	P2_28	TIMER-7	0.000010914 to 6e6 Hz	free
P9_22	P1_08	PWMSS-0, PWM A	0.42 to 50e6 Hz	free
P9_21	P1_10	PWMSS-0, PWM B	0.42 to 50e6 Hz	free
P9_31	P1_36	PWMSS-0, PWM A	0.42 to 50e6 Hz	MCASP0
P9_29	P1_33	PWMSS-0, PWM B	0.42 to 50e6 Hz	MCASP0
P9_14	P2_01	PWMSS-1, PWM A	0.42 to 50e6 Hz	free
P9_16		PWMSS-1, PWM B	0.42 to 50e6 Hz	free
P8_36		PWMSS-1, PWM A	0.42 to 50e6 Hz	HDMI
P8_34		PWMSS-1, PWM B	0.42 to 50e6 Hz	HDMI
P8_19		PWMSS-2, PWM A	0.42 to 50e6 Hz	free
P8_13	P2_03	PWMSS-2, PWM B	0.42 to 50e6 Hz	free
P8_46		PWMSS-2, PWM A	0.42 to 50e6 Hz	HDMI
P8_45		PWMSS-2, PWM B	0.42 to 50e6 Hz	HDMI
P9_42		PWMSS-0, CAP	0.0233 to 50e6 Hz	free (double pin)

JT_05		PWMSS-1, CAP	0.0233 to 50e6 Hz	JTag (UART0_TXD)
SD_10	SD_10	PWMSS-1, CAP	0.0233 to 50e6 Hz	SD card in button
P9_28	P2_30	PWMSS-2, CAP	0.0233 to 50e6 Hz	MCASP0

The TIMER and PWMSS-CAP subsystems use a 32 bit counter and generate a single pulse train. The signal goes high at the beginning of the period and low when the compare value is reached. In the TIMER subsystems the counter clock can get pre-scaled, so very long periods (low frequencies) are possible. The maximum is ??? days.

In contrast the PWMSS-PWM subsystems use a 16 bit counter with 17 bit duty resolution in up-down mode. The frequency range can get extended by a clock pre-scaler. *libpruio* auto-configures the mode and the pre-scaler. For high frequencies (low counter periods) the counter runs in up-count mode (16 bit resolution). For frequencies below 1526 Hz (= 100e6 / 65536) the counter runs in up-down mode (17 bit duty resolution, 16 bit frequency resolution). This up-down mode can be forced for high frequencies also by masking the subsystems bit in member variable **PwmMod::ForceUpDown** to 1.

A PWMSS-PWM subsystem handles two outputs at the same frequency. Two Action Qualifiers are used to set the states of the outputs A and B in case of six different events, see ARM Reference Guide, chapter 5.2.4.3 for details. By default *libpruio* configures the Action Qualifiers to set both outputs to high state at the beginning of a period and switch to low state when the counter matches the duty value. The default configuration can get overriden in array PwmMod::AcCtl. This three dimensional array contains the configurations for the Action Qualifiers.

Index	Value	Description
1	0,1	Output (0 = A, 1 = B)
2	0-2	Subsystem
3	0-2	Usecase (0 = up count, 1 = up count up-down mode, 2 = down count up-down mode)

Additionally the PWMSS-PWM modules can get synchronized. ???

The output gets specified by calling function **PwmMod::setValue()**. Since the output frequency may vary from the specified parameters due to resolution issues, the real values can get computed by calling function **PwmMod::Value()**.

Find example code in **pwm_adc.bas** or **pwm_cap.bas**.

TIMER

A TIMER output is very similar to PWM output, but the pulse train starts in low state for `Dur1` duration, and then toggles to high state for the `Dur2` period of time. Both durations are specified directly in [mSec] (instead of setting frequency and duty cycle). Additionally a TIMER can operate in one shot mode, stopping after the first pulse, or after a number of pulses in the range of 1 to 255.

The TIMER feature is available on a subset of header pins. By default a TIMER pin is configured as an output pin without resistor connection. This pin toggles its state, depending on a counter running on a certain clock rate. When the counter matches specified values, the state of the output changes. Since TIMER output can get generated by different subsystems (and *libpruio* supports some of them), the duration range in [mSec] vary between the pins. In any case the counter resolution is 32 bit.

BB	PB	Subsystem	Max Dur	Min Dur	Notice
P8_07		TIMER-4	45812984	0.000167	free
P8_09	P1_28	TIMER-5	45812984	0.000167	free
P8_10	P1_26	TIMER-6	45812984	0.000167	free
P8_08	P2_28	TIMER-7	45812984	0.000167	free
P9_42		PWMSS-0, CAP	42949	0.00002	free (double pin)
SD_10	SD_10	PWMSS-1, CAP	42949	0.00002	uSD slot
JT_05		PWMSS-1, CAP	42949	0.00002	JTag (UART0_TXD)
P9_28	P2_30	PWMSS-2, CAP	42949	0.00002	MCASP0

The output gets specified by calling function **TimerUdt::setValue()**. Since the output timing may vary from the specified parameters due to resolution issues, the real values can get computed by calling function **TimerUdt::Value()**.

CAP

CAP stands for Capture and Analyse a digital Pulse train. So it means measuring the frequency and the duty cycle of a digital signal input. It's a kind of reverse PWM. Usualy CAP is used to read sensor inputs, ie. from a speed sensor.

CAP is available on a subset of header pins. The specified pin gets configured as input with pulldown resistor. A counter is running on a certain clock rate. Each transition of the input triggers the capture of the counter value. The frequency gets measured as the difference between two positive transitions (a period). The duty cycle gets measured as the ratio between a period and the on-time of the signal. A positive transition resets the couter. Since CAP input can get analysed by different subsystems (and *libpruio* support some of them), the frequency range vary between the pins.

BB	PB	Subsystem	Frequency Range	Notice
P9_42		PWMSS-0, CAP	0.0233 to 50e6 Hz	free (double pin)
SD_10	SD_10	PWMSS-1, CAP	0.0233 to 50e6 Hz	SD card in button
P9_28	P2_30	PWMSS-2, CAP	0.0233 to 50e6 Hz	MCASP0
JT_04		PWMSS-2, CAP	0.0233 to 50e6 Hz	JTag (UART0_RXD)

The measurement results get available by calling function **CapMod::Value()**. Before, you have to configure the pin for CAP input by calling function **CapMod::config()** once.

Find example code in **pwm_cap.bas**.

QEP

QEP stands for Quadrature Encoder Pulse measurement. So it means measuring the position and the speed of a quadrature encoder. Encoders for rotary (ie. for an electrical drive) and linear (ie. for a printer head) movement are supported.

QEP is available on a subset of the header pins. Each of the three PWMSS subsystems contains a QEP module. (The QEP module of PWMSS-2 can get connected on two sets of to the header pins.)

The module can operate in different modes, and depending on the mode it operates on a different set of input signals (header pins). Function **QepMod::config()** is used to specify the operational mode of the module. It configures one or more header pins, depending on the first parameter *Ball*. And depending on the mode (and the number of input pins), either

- speed, or
- speed, direction and position

information is available when calling function **QepMod::Value()**. The accuracy of the position information can get improved by using an index signal that resets the position counter.

BB	PB	Type	Subsystem	Speed	Direction	Position	Index	Further Pins	Notice BBB
P9_42		A input	PWMSS-0	X	-	-	-		free (double pin)
P9_27	P2_34	B input	PWMSS-0	X	X	X	-	P9_42	free
P9_41		I input	PWMSS-0	X	X	X	X	P9_42, P9_27	free (double pin)
P8_35		A input	PWMSS-1	X	-	-	-		HDMI
P8_33		B input	PWMSS-1	X	X	X	-	P8_35	HDMI
P8_31		I input	PWMSS-1	X	X	X	X	P8_35, P8_33	HDMI
P8_12		A input	PWMSS-2	X	-	-	-		free
P8_11		B input	PWMSS-2	X	X	X	-	P8_12	free
P8_16		I input	PWMSS-2	X	X	X	X	P8_11, P8_12	free
P8_41		A input	PWMSS-2	X	-	-	-		HDMI
P8_42		B input	PWMSS-2	X	X	X	-	P8_41	HDMI
P8_39		I input	PWMSS-2	X	X	X	X	P8_41, P8_42	HDMI
	P2_24	A input	PWMSS-2	X	-	-	-		free
	P2_33	B input	PWMSS-2	X	X	X	-	P2_24	free
	P2_22	I	PWMSS-2	X	X	X	X	P2_24,	free

		input						P2_33	

Direction information is derived from two different signals that "look" at the sensor lines with a mechanical shift of 1 / 4 of the pitch. So to get direction information at least an B input (or an I input) needs to get specified as *Ball* parameter to the function **QepMod::config()**.

The speed is defined as the rate of change of position with respect to time v = Δx / ΔT. The QEP modules Capture Unit works at a certain frequency, specified by parameter *VHz* in the call to function **QepMod::config()**. At this frequency the current position and the time between the last transitions get stored in the array PwmSS::Raw. Function **QepMod::Value()** uses these data to compute the speed. By default speed values are scaled in transitions per second. Ie. for a rotary sensor turning at 1 revolution per second you'll get twice the number of lines for an A input and four times the number of lines for a B or I input.

For low speed values transitions period time is used in equation v = n / ΔT, where n is the number of transitions counted (n = 2 for A input, n = 4 for B or I input). The higher the speed value, the smaller the speed resolution. So in contrast, high speed values get computed by counting the number of input transitions in the given period of time (T), using equation v = Δx / T. *libpruio* auto-switches between both equations and auto-computes the optimal speed value for switching (array **QepMod::Prd**), in order to get maximum speed resolution.

The speed limits are depending on the hardware in use. Linear sensors don't have an upper speed limit, unless the counter clock rate doesn't get exceeded (100e6 Hz). But in case of a rotary sensor the maximum speed is limited by the number of sensor lines per revolution. When the sensor turns more than one revolution in the measurement period (parameter *VHz*) the number of transitions (Δx) will be incorrect. So for rotary encoders you've to make sure that the measurement frequency is higher than the maximum sensor frequency, ie. *VHz* = 25 limits the shaft speed to 1500 rpm (= 25 [1/s] * 60 [s/min]). An alternative approach is to specify a multiple of the sensor lines as parameter *PMax* and correct the position information accordingly. Parameter *Scale* can be used to apply a custom scaling factor to speed computations.

The minimum speed is limited by the measurement frequency (*VHz*). When no transition period (2 transitions for A input, 4 transitions for B or I input) occurs in the measurement time (*VHz*), the speed gets computed to zero. That's also the case when the direction changes during the measurement period. By default *libpruio* doesn't count each transition, because time measurement in this case requires high accuracy of the sensor mechanics. If either the duty cycle of a signal is not exactly 50 % or the phase shift between A and B signals is not exactly 1 / 4 (90 °), the computed speed values will change erratic. Since most sensors don't fulfill those requirements, *libpruio* measures frequency on just one transition of a single signal. This means at least two (in case of A input) or four position counts (in case of B and I input) are necessary to measure a non-zero speed value.

Note

A direction change during the time period always results in zero-speed computation.

Position information is computed by counting the transitions of the input signals considering the direction information (A and B inputs are necessary). In case of just one input the position counter runs in upwart direction.

An index signal (I input) can be used to reset the counter. By default the counter is set to zero on the positive transition of the index signal when counting in upward direction. And the counter is set to value *PMax* on the negative transition of the index signal when counting in downward direction.

The measurement results get available by calling function **QepMod::Value()**. Before, you have to configure the pin(s) for QEP input by calling function **QepMod::config()** once.

Find example code in **qep.bas**.

Pinmuxing

At boot time the operating system sets all pins in a save mode. You can list the default settings by executing the example **analyse** after power on reset.

When you configure a header pin in your code, *libpruio* first checks its current mode, and just continues in case of a match. Otherwise action is required to get you what you need. In order to set a header pin in an other mode, or just configure a pull-up or pull-down resistor, there're three methods supported:

1. **Custom overlay** -> configure at boot time the desired mode by an custom overlay
2. **Universal overlay** -> configure at boot time multiple custom modes in an overlay, and switch at runtime
3. **LKM** -> load the kernel module, and switch at runtime

Each method has its advantages and downsides. The first two are well known and save, but unflexible. There is no fixed border between them. Ie. you can prepare a mixed device tree blob with multiple settings for some pins, and single settings for the others. Anyway, you have to know and specify each pin configuration before boot. And the necessary tools (device tree compiler, capemgr) are not very reliable. Debugging is a mess. Any change requires rebooting.

In contrast the loadable kernel module (LKM, since version 0.6) is flexible and fast in executation speed. You need not prepare the pins before boot. Instead you can access all pins at runtime, and you can switch any feature at run-time. This method has a short boot time and a small memory footprint. But you can more easy raise conflicts with other systems.

libpruio checks in the constructor **PruIo**:**PruIo()** if the LKM is available, and uses it when found. It's the prefered pinmuxing method. Otherwise it checks for settings provided by an universal overlay. If neither of them are present, it works with the current pin settings and throughs error messages when the programm tries to change pinmuxing.

Note

It's best practice to configure (pinmux) the pins before starting the *libpruio* main loop (by calling **PruIo::config()**). Also it's possible to pinmux later. But in the later case the user has to make sure that the PRU finished configuration and the main loop is running. In FreeBASIC syntax use `WHILE .DRam[0] > PRUIO_MSG_IO_OK : .WaitCycles += 1 : WEND` before the pinmuxing command.

Custom overlay

A custom setting contains a single configuration for each pin. Once you finished your development and burned the design on a printed circuit board, this is the prefered method for setup.

- The pinmux matches your design,
- it's loaded safely by the kernel, and
- the software can run from user space (no administrator privileges).

But this is the final state, and it's a long way to get there.

In order to load a custom overlay, you need to create the device tree blob first. Therefor a source code file (suffix `*.dts`) is necessary, containing the pin claims and configurations. Also code to export the pins to SysFs is necessary, and the overlay should load the `uio_pruss` kernel module and enable the PRUSS.

libpruio ships with a tool that can handle all that stuff for you. You specify the bare minimum in a source code file, compile the code and execute the binary. It will create a matching device tree source code, and - when executed with `sudo` - will also compile the device tree blob to the correct place, so that you can load it as usual (either as uBoot overlay or by `capemgr`).

Therefor use the source code **`src/config/dts_custom.bas`** as a template, rename it and adapt it to your needs, by editing the part

```
'* The file name.
#DEFINE FILE_NAME "pruio_custom"
'* The version.
#DEFINE VERS_NAME "00A0"
'* The folder where to place the compiled overlay binary.
VAR TARG_PATH = "/lib/firmware"
'* The BB model.
VAR COMPATIBL = "ti,beaglebone-black"

'''''''''''''''''''''''''''' create settings for all required pins here
'M(P8_09) = CHR(7 + _I_)    ' example: pin  9 at header P8 in mode 7 (GPIO) as input
       (pulldown resistor)
'M(P9_21) = CHR(3 + PWMo)   ' example: pin 21 at header P9 in mode 3 (PWM) as output
       (no resistor)
M(P9_14) = CHR(6 + PWMo)   ' example: pin 14 at header P9 in mode 6 (PWMo) as output
       (no resistor)
M(P9_42) = CHR(0 + CAPi)   ' example: pin 42 at header P9 in mode 0 (eCAP) as input
       (pulldown resistor)
''''''''''''''''''''''''''''''''''''''''''''''''''''''' end of adaptions
```

The array named `M(...)` is of type `UBYTE` and contains the configuration for the header pins (or CPU balls). This code outputs a file named `pruio_custom-00A0.dts` in the current directory, ready to get compiled to a device tree blob that

- claims PRUSS and pins P9_14 + P9_42
- sets pin P9_14 in PWM output mode
- sets pin P9_42 in CAP input mode
- enables the PRUSS

When you execute the binary with `sudo`, the source code also gets compiled to `/lib/firmware/pruio_custom-00A0.dtbo`, ready to get loaded from there.

Note

For capemgr loading use `echo pruio_custom > ...` in this case (= drop the `-00A0.dtbo` part).

Universal overlay

This method is very similar to the previous **Custom overlay** method. You need source code for a device tree overlay, that gets compiled to a binary blob, which gets loaded by the kernel finally, either as uBoot overlay at boottime, or at runtime by `capemgr`.

But the source code does not contain only a single cofiguration per pin. Instead the blob prepares multiple configurations for each (or just some) pins, and the mode can get adapted at runtime by SysFs activities. Therefor the programm must be executed with `sudo`.

libpruio also ships with a tool to generate that kind of device tree blob, named **`src/config/dts_universal.bas`**. The same code is used to generate the source file `*.dts` and compile it. But in contrast the array named `M(...)` contains multiple configurations for each pin. All standard configurations are prepared in header files, and can get loaded by

```
' quick & dirty: first create settings for all pins ...
#INCLUDE ONCE "P8.bi"
#INCLUDE ONCE "P9.bi"
#INCLUDE ONCE "JT.bi"
#INCLUDE ONCE "SD.bi"
```

Afterwards you can (and should) reduce the configurations to the set of desired pins. Therefor just empty the entries in array `M(...)` for the pins your wont use, by code like

```
M(P8_25) = "" ' #4:BB-BONE-EMMC-2G
```

All remaining pins will get claimed and configured by the resulting device tree blob, that gets generated after compiling and executing the FreeBASIC code. When executing with `sudo`, the blob source gets compiled to folder `lib/firmware/libpruio-00A0.dtbo`, from where it can get loaded by the kernel.

Note

> The device tree compiler has a bug (effective up to kernel version 4.4.x). When the source file contains a certain number of pin configurations, the compiler stops compiling without any warning or error message. The resulting output file is valid, but contains just a subset of the source file configuration.
>
> When your blob tries to use a pin claimed by another subsystem (ie. HDMI), this pin configuration doesn't load. Check the output of `dmesg` for errors.

This method is pretty similar to the solution used by tool `config-pin`, but unfortunately it cannot be compatible. While `config-pin` uses human readable pin naming, *libpruio* uses byte coding instead. This is feasible since the user need not deal with numbers. Instead he can handle the byte code for pin numbers and modes by the prepared macros. The *libpruio* code can compute adaptions without containing long lists of text description. The code executes faster, consumes less memory, and is more easy to maintain.

There's another difference. While `config-pin` allows to set double pins `P9_41` and `P9_42` in contrary output modes, *libpruio* handles that pins in a save manner, since there are no `P9_91` and `P9_92` pins.

Anyway, the desired mode for the pins in use has to be prepared before boot. All prepared modes get loaded at boot time. This increases down boot time and consumes a lot of kernel space memory. Big overlays needs splitting in multiple files.

LKM

This method has no restriction to pre-prepared pin configurations. Each pin (or CPU ball) can get configured at runtime to any mode. No device tree overlay with `pinctrl-single` or `bone-pinmux-helper` features is necessary at boot time. The LKM features can get enabled (loaded) or disabled (unloaded) at runtime during a session (without re-booting).

You can install the LKM either by the package management system, see chapter **Preparation** for details. Or you can compile and install it from the GitHub source tree, see section **LKM** for details. Both methods provide a systemd service named `libpruio-lkm.service`, which loads the LKM at boot time and makes the pinmuxing feature available for all users of the newly created system group `prui` (GID < 1000). So when you make yourself a member of that group by executing

```
sudo adduser <YourUserID> pruio
```

you can do pinmuxing from user space (without `sudo`, no administrator privileges necessary).

Note

> The user group `pruio` doesn't get removed when uninstalling. It's still existent after uninstall. To remove it edit the file `sudo nano /etc/group` and remove the related line `pruio:x:...` from that file (change is effective after next logout or boot). That way you can also remove a user from that group by deleting only his name in the line. Fancy things may happen when files or folders were created with the ownership of that group.

By default, the pinmuxing feature is limited to the free header pins, which are not claimed for other subsystems by the kernel. So you cannot configure pins in default configuration, because all pins are claimed by the cape-universal device tree blob. You can either disable the trigger file by (or by renaming that file, adapt the file name for pocket-beagle, BBGrenn or BBBlue)

```
sudo rm /boot/dtbs/`uname -r`/am335x-boneblack-uboot-univ.dtb
```

Or you can disable the *libpruio* safety feature by calling the constructor **PruIo::PruIo()** with PRUIO_ACT_FREMUX ored in first parameter.

Note

> On kernel 3.x there's no user access to files in folder `/sys/kernel/debug`. *libpruio* cannot determin the current kernel claims. For user space pinmuixing you have to disable that feature by seting bit 15 in your constructor **PruIo**:**PruIo()** call (parameter Act ored by PRUIO_ACT_FREMUX -> no safety feature).

The systemd service handles LKM loading and unloading. You can switch off the LKM at runtime for the current session by executing

```
sudo systemctl stop libpruio-lkm.service
```

The service gets stopped for the rest of the session and will start again after next boot. To re-start during the current session execute

```
sudo systemctl start libpruio-lkm.service
```

If you don't want auto-loading at boot time execute

```
sudo systemctl disable libpruio-lkm.service
```

And to re-enable the boot auto-load execute

```
sudo systemctl enable libpruio-lkm.service
```

Allthough it's not necessary, it's still recommended to load a minimal device tree blob along with the LKM, just to claim the used subsystems and set of pins. That way the kernel prohibits other software using that pins and subsystems. The source of such a minimal overlay may get named `libpruio-00A0.dts` and may look like

```
// dts file auto generated by pruio_config (don't edit)
/dts-v1/;
/plugin/;

/ {
    compatible = "ti,beaglebone-black", "ti,beaglebone";

    // identification
    board-name = "libpruio";
    manufacturer = "TJF";
    part-number = "PruIoBBB";
    version = "00A0";

    // state the resources this cape uses
    exclusive-use =
      "P8.07",
      ...
      "P9.42",
      "adc",
      "timer4",
      "timer5",
      "timer6",
      "timer7",
      "epwmss0",
      "epwmss1",
      "epwmss2",
      "pruss";

    fragment@0 {
      target = <&pruss>;
      __overlay__ {
          status = "okay";
      };
    };
  };
```

Replace the dots by further pins you want to use. But keep in mind, the overlays won't load if you try to reserve pins claimed previously by other systems, ie. like HDMI or MMC. The set of free pins vary on Beaglebone hardware. Find further information in the system reference manual shipped with your board.

Once you finished the overlay source, compile and install it by executing

```
sudo dtc -@ -I dts -O dtb -o /lib/firmware/libpruio-00A0.dtbo libpruio-00A0.d
```

Then load the overlay by the capemgr (add `libpruio` to file `/etc/default/capemgr`). In case of trouble check the kernel log output (`dmesg | grep uio`). (I'm not sure if loading as uBoot overlay in file `/boot/uEnv.txt` will be also sufficient, the capemgr full sure evaluates the `exclusive-use` tag.)