Tutorial

This tutorial is still incomplete. Please contribute by clicking on the `Edit this page on GitHub` link at the bottom of this page.

Table of Contents

Registers

All registers are in undefined state at startup. The “Reset” column in the datasheets is misleading (and only emulated by the original IDE which generates code to set all registers to 0). You have to initialize all registers yourself.

> *Source: https://www.eevblog.com/forum/blog/eevblog-1144-padauk-programmer-reverse-engineering/msg3129442/#msg3129442*

Clock Sources and Setup

TODO: This section is still missing and should explain the following terms:

_sdcc_external_startup, IHRC, ILRC, external crystal, SYSCLK, maximum possible clk frequency, factory calibrated values, easy pdk calibration, …

Code Options

Padauk µCs have several options that are called "code option" in the datasheet. There is usually an overview of all code options a µC supports in a chapter towards the end of the datasheet. The datasheet does not provide any detailed information on how to set the code options, because Padauk expects users to use their IDE which takes care of that for you. Some code options are set as fuses, whereas others are set in undocumented registers.

To figure out whether a code option is set as a fuse or in an undocumented register, you can consult the following table. For µCs not listed in the table, you need to look into the PDK include files as described in the next two sections, or into the original `.INC` files that come with the Padauk IDE.

µC	**Fuse**				**Undocumei**					
					ROP					
	Security	**Pin Drive**	**Startup Speed**	**3-bit LVR**	**INT0 Source**	**INT1 Source**	**TMX Bits**	**TMX CLK**	**PWM Type**	**P\ C**
PFS154	x	x	x							
PFS172[1]	x	x	x		x	x	x	x	x	
PFS173	x	x	x		x	x	x	x	x	
PMS150C	x	x	x	x						

µC	Fuse				Undocumen					
					ROP					
	Security	Pin Drive	Startup Speed	3-bit LVR	INT0 Source	INT1 Source	TMX Bits	TMX CLK	PWM Type	P\ C
PMS15A	x	x	x	x						
PMS152	x		x		x	x	x	x	x	x
PMS154C[2]	x	x	x	x						
PMS171B[3]	x	x	x		x	x	x	x	x	

Click on an individual table cell to get the code that controls the corresponding code option.

Fuses

Some of the code options are configured by setting bits in a magic word towards the end of the ROM. These can be set using the `PDK_SET_FUSE(...)` macro. The factory-default fuse settings differ depending on the µC model, which is why it is best to always set all supported fuses. Be aware that your program must only contain a single call to `PDK_SET_FUSE`.

```
// Example of setting fuses on a PFS173
// The available fuses vary depending on the µC model!
#define PFS173

#include <pdk/device.h>

unsigned char _sdcc_external_startup(void)
{
    PDK_SET_FUSE(FUSE_SECURITY_ON | FUSE_PB4_PB5_NORMAL | FUSE_BOOTUP_SLOW
}

void main(void)
{
    // ...
}
```

Undocumented Registers

Other code options are configured in undocumented registers:

- `ROP`: Many µCs have a `ROP` register that configures PWM, timer, and external interrupt related code options.
- `MISC2`: Some µCs have a `MISC2` register that supports selecting the comparator edge(s) that trigger an interrupt.
- `MISCLVR`: Some µCs configure the LVR and sometimes bandgap related code options in a `MISCLVR` register.

> It is unclear whether you are allowed to change these code options while the µC is running, or whether you are supposed to only set them once and leave them as they are.

```c
// Example of using the ROP register on a PFS173
// The available settings and undocumented registers vary depending on the
#define PFS173

#include <pdk/device.h>

unsigned char _sdcc_external_startup(void)
{
    // ...
}

void main(void)
{
    // Use PA4 for INT1 instead of PB0
    ROP = ROP_INT_SRC_PA4;
}
```

Digital I/O

Digital I/O is organized in ports of (at most) 8 pins. The ports are named `A`, `B`, and `C`. I/O pins are controlled by the following registers (replace `x` by `A`, `B`, or `C`):

- `PxC` **Control Register**: Controls whether the pin is used as an input (`0`) or output (`1`).
- `Px` **Data Register**: Sets the output low (`0`) or (`high`). Has no effect on inputs.
- `PxPH` **Pull-up ("pull-high") Register**: Enables a pull-up resistor. The resistor is also active when the pin is used as an output and driven high (!)
- `PxPL` **Pull-down ("pull-low") Register**: Enables a pull-down resistor (only very few ports and pins have pull-down resistors). It is unknown whether the resistor is also

active when the pin is used as an output and driven high (!)

- `PxDIER` **Digital Input Enable Register**: For pins to work as digital inputs, this register needs to be set to `1`. Otherwise the input signal is cut off from the digital circuitry (including the external interrupt hardware + wake up functionality). This is recommended when using the pin as an analog input to the ADC or comparator, and required when using the pin as input of an external crystal.

The following table provides an overview of how the registers work together to control an I/O pin (this example uses PB.0).

PBC.0	PB.0	PBPH.0	PBPL.0	PBDIER.0	Result
0	x	x	x	0	Digital I/O disabled (can only use pin for analog input and crystal)
0	x	0	0	1	Input without pull resistors
0	x	1	0	1	Input with pull-up resistor
0	x	0	1	1	Input with pull-down resistor
0	x	1	1	1	*not explicitly mentioned in datasheet*
1	0	x	x	x	Output low wihout pull resistors
1	1	0	0	x	Output high wihout pull resistors
1	1	1	0	x	Output high with pull-up resistor
1	1	1	1	x	*not explicitly mentioned in datasheet*
1	1	0	1	x	*not explicitly mentioned in datasheet*

Maximum Current

The maximum current an I/O pin can drive and sink varies by pin and µC. Take a look at the interactive pinout diagram of an individual µC and enable “Maximum Sink/Drive Current” to learn more.

TODO: Many µCs can increase the maximum current on two of their pins.

- How do I enable that?
- Is there any downside to enabling that?

Interrupts

Padauk µCs provide up to 8 interrupt sources. Interrupts are configured with the following registers:

- `INTEGS`: Interrupt Edge Select Register
- `INTRQ`: Interrupt Request Register
- `INTEN`: Interrupt Enable Register

Whenever an interrupt occurs, the corresponding bit in the `INTRQ` register is set to `1` by hardware. The µC never sets the bit back to `0`. This has to be done by your program.

If you want to trigger an interrupt service routine (ISR) when an interrupt occurs, then you have to individually enable it for each interrupt source by setting the corresponding bit to `1` in the `INTEN` register. In addition, you need to enable interrupts globally by calling `__engint()`.

All interrupts share a single ISR, which is denoted by adding the `__interrupt(0)` attribute to the function definition. Inside the ISR, you have to check which interrupt(s) triggered it by checking the corresponding bits in the `INTRQ` register. It is possible for multiple interrupts to occur at the same time

If you handle multiple interrupts in your ISR **and** you disable an interrupt conditionally by setting its corresponding bit in `INTEN` to `0` at some point in your program, you also have to check the `INTEN` register when the ISR is called. This is because the bit in `INTRQ` is still set even if the interrupt is disabled in `INTEN`. The alternative is to disable **all** interrupts whenever you want to disable a single interrupt by calling `__disgint()`.

The following example shows how to use the two external interrupts on a PFS173:

```
#include <stdint.h>
#include <pdk/device.h>

volatile uint16_t counter;

void interrupt(void) __interrupt(0)
{
```

```
  if (
    INTEN & INTEN_PA0  // Is the interrupt enabled?
    &&
    INTRQ & INTRQ_PA0  // Did the interrupt occur?
  ) {
    // Reset interrupt request bit
    INTRQ &= ~INTRQ_PA0;

    // Handle interrupt
    counter++;
  }

  if (
    INTEN & INTEN_PB0  // Is the interrupt enabled?
    &&
    INTRQ & INTRQ_PB0  // Did the interrupt occur?
  ) {
    // Reset interrupt request bit
    INTRQ &= ~INTRQ_PB0;

    // Handle interrupt
    // ...
  }
}

void main(void)
{
  INTEN = INTEN_PA0 | INTEN_PB0;
  INTEGS = INTEGS_PA0_BOTH | INTEGS_PB0_BOTH;

  // Further initialization code

  // Enable interrupts.
  INTRQ = 0;
  __engint();

  for(;;) {
    // Make sure to temporarily disable the interrupt when
    // reading the variable from the main program, since access
    // to 16 bit variables is not atomic.
    uint16_t i;

    INTEN &= ~INTEN_PA0;
    // The ISR is now no longer called when an interrupt at PA0 occurrs.
    // However, the INTRQ_PA0 bit in INTRQ will still be set if an interru

    // -> Let's say at this moment the signal at PA0 changes
    // -> INTRQ & INTRQ_PA0 is set to 1 by hardware
    // -> The ISR is _not_ called, because INTEN & INTEN_PA0 is not set.

    i = counter;

    // -> Let's say at this moment the signal at PB0 changes
    // -> INTRQ & INTRQ_PB0 is set to 1 by hardware
    // -> The ISR is called, because INTEN & INTEN_PB0 is set AND interrup
    //
    // Important: The INTRQ_PA0 bit is still set to 1!
```

```
    // If we don't check INTEN & INTEN_PA0 in the ISR, the code for PA0 wo
    // though we disabled the interrupt.

    // Re-enable the PA0 interrupt now that we have read the counter varia
    INTEN |= INTEN_PA0;

    // ...
  }
}

unsigned char _sdcc_external_startup(void)
{
  // ...
}
```

VDD/2 LCD Bias Voltage Generator

Some µCs have the ability to output `VDD/2` on some of their pins. This functionality is enabled in the `MISC` register. After enabling the LCD bias voltage generator, the corresponding pins can no longer be used as input pins, because configuring them as input enables the `VDD/2` output. You should disable pull-up, pull-down resistors and also disable the `PxDIER` bit for the corresponding pins when using the LCD bias voltage generator.

The following table provides an overview of how the registers work together to control the LCD bias voltage generator (this example uses PB.0).

PBC.0	PB.0	PBPH.0	PBPL.0	PBDIER.0	Result
0	x	0	0	0	**Output** VDD/2
1	0	0	0	x	Output GND
1	1	0	0	x	Output VDD

Watchdog Timer

The watchdog timer continuously counts up and automatically resets the µC when it reaches its maximum value. It can be enabled in the `CLKMD` register (make sure to enable both the watchdog and `ILRC`). The timeout value is set in the `MISC` register and can be either `8k`, `16k`, `64k`, or `256k` `ILRC` ticks.

To avoid µC resets during normal operation, you should regularly call `__wdreset();` to reset the watchdog timer. The `ILRC` frequency and therefore watchdog timeout period can "drift a lot due to variation of manufacturing, supply voltage and temperature"[4]. You should also call `__wdreset();` right after reset and wakeup, because "the watchdog period will also be shorter than expected after reset or wakeup events"[4].

Low Voltage Reset (LVR)

LVR automatically resets and stops the µC while the supply voltage is below the configured LVR threshold. LVR is enabled by default and can be disabled in the `MISC` register. The threshold is configured as a code option; see the Code Options section for more information.

You should configure the LVR threshold to at least meet the minium voltage supported by your system clock setting. Consult the `f_SYS` row in the "Device Characteristics" table of the datasheet for more information.

Simulation and Emulation

Several options exist to simulate/emulate a Padauk µC. However, the official in-circuit emulator from Padauk currently is the only option that supports peripherals and interrupts. *Emulators* emulate a µC in hardware. They are usually realized using FPGAs and run in realtime. *Simulators* simulate a µC in software and are usually slower than the real µC.

µCsim

µCsim is part of SDCC and supports simulation of Padauk µCs, including breakpoints. The binary is called `spdk` and currently supports the following instruction sets and µCs (`spdk -H`):

Instruction Set	Simulated µC
PDK13	PMC153
PDK14	PMS132B
PDK15	PMS134

Interrupts are not currently supported.

µCsim Usage Example

This example invocation shows some of the commands supported by the simulator.

```
spdk main.ihx -X 8000000 -tPMS134

dump ram 0 9999   # dump RAM contents from byte 0 to 9999
dump regs8 0 9999 # dump register contents from register 0 to 9999
dump rom 0 9999   # dump ROM contents from word 0 to 9999

step 2            # step program execution 2 times

state             # print state of µC

break 0x26        # set breakpoint before the instruction at 0x26 is execu

run               # run program forever or until breakpoint is hit
```

Watchout: ROM addresses used by the emulator are word-based, whereas ROM addresses printed in the `main.rst` file are byte-based. To break at the following instruction defined in the `main.rst` file:

```
000040 F4 51                  122          sub     a, #0xf4
```

you have to divide the byte-based address (0x000040) by 2 when setting the breakpoint for the simulator:

```
break 0x20
```

Free PDK Simulator

A work in progress simulator for the PDK14 instruction set is available at free-pdk/fppa-pdk-tools. It currently does not support interrupts or peripherals.

pacmancoder/vpadauk

@pacmancoder has created a simulator for PDK13 that supports the PMS150C. The simulator is written in Rust and has support for interrupts, the timer, and digital IO.

Free PDK Emulator

A work in progress VHDL-based emulator for the PDK14 instruction set and PFS152 µC is available at free-pdk/fppa-pdk-emulator-vhdl.

Padauk ICE (in-circuit emulator)

Padauk offers in-circuit emulators that can be controlled from the Padauk IDE. A list of available emulators is available here. The list of µCs each emulator supports can be found here. The Padauk ICE supports most features of the µCs, and features not supported by ICE are usually called out in the datasheets.

> The Padauk ICE **cannot** be used to emulate programs created by the Free PDK toolchain.

---

1. Some additional options seem to exist which are not documented in the datasheet: https://github.com/free-pdk/pdk-includes/blob/f44fc2e7678b1ab72ed8bac6b9d408118f330ad8/device/pfs172.h#L153-L165 ↩
2. This µC has additional undocumented code options configured in `MISC2`: https://github.com/free-pdk/pdk-includes/blob/f44fc2e7678b1ab72ed8bac6b9d408118f330ad8/device/pms154c.h#L156-L158 ↩
3. Some additional options seem to exist which are not documented in the datasheet: https://github.com/free-pdk/pdk-includes/blob/f44fc2e7678b1ab72ed8bac6b9d408118f330ad8/device/pms171b.h#L145-L148 ↩
4. From the PFS173 datasheet ↩ ↩[2]

Edit this page on GitHub

Checkout /doc-style for more information on some of the special Markdown formatting features we use.

Free PDK

Free PDK	free-pdk	Free PDK is an effort to create an open source alternative to the proprietary Padauk µC programmer, as well as adding support to SDCC for Padauk µCs.