



acam-messelectronic gmbH

is now

Member of the ams Group

The technical content of this acam-messelectronic document is still valid.

Contact information:

Headquarters:

ams AG

Tobelbaderstrasse 30

8141 Unterpremstaetten, Austria

Tel: +43 (0) 3136 500 0

e-Mail: ams_sales@ams.com

Please visit our website at www.ams.com

PICOSTRAIN®

Data Sheet

PSØ9

Single Chip Solution for Strain Gauges
Volume 2: CPU

5th November 2014

Document-No: DB_PSØ9_Vol2_en V1.0

Published by acam-messelectronic gmbh

©acam-messelectronic gmbh 2014

Disclaimer / Notes

“Preliminary” product information describes a product which is not in full production so that full information about the product is not available yet. Therefore, acam-messelectronic gmbh (“acam”) reserves the right to modify this product without notice. The information provided by this data sheet is believed to be accurate and reliable. However, no responsibility is assumed by acam for its use, nor for any infringements of patents or other rights of third parties that may result from its use. The information is subject to change without notice and is provided “as is” without warranty of any kind (expressed or implied). Picocap is a registered trademark of acam. All other brand and product names in this document are trademarks or service marks of their respective owners.

Support / Contact

For a complete listing of Direct Sales, Distributor and Sales Representative contacts, visit the acam web site at:

<http://www.acam.de/sales/distributors/>

For technical support you can contact the acam support team in the headquarters in Germany or the Distributor in your country. The contact details of acam in Germany are:

support@acam.de

or by phone

+49-7244-74190.

Content

| | | |
|-----|--|------|
| 1 | Overview | 1-3 |
| 1.1 | General | 1-3 |
| 1.2 | Functional Block Diagram..... | 1-3 |
| 2 | CPU & Memory | 2-4 |
| 2.1 | Block Diagram | 2-4 |
| 2.2 | Memory Organization | 2-5 |
| 2.3 | Arithmetic Logic Unit (ALU) | 2-14 |
| 2.4 | Status and Result Registers | 2-15 |
| 3 | General Functions..... | 3-19 |
| 3.1 | System Reset, Sleep Mode and Auto-configuration..... | 3-19 |
| 3.2 | CPU Clock Generation | 3-20 |
| 3.3 | Watchdog Counter and Single Conversion Counter | 3-21 |
| 3.4 | Timer..... | 3-21 |
| 4 | Instruction Set..... | 4-22 |
| 4.1 | Branch instructions | 4-22 |
| 4.2 | Arithmetic operations | 4-22 |
| 5 | Assembly Programs | 5-46 |
| 5.1 | Directives | 5-47 |
| 5.2 | Sample Code | 5-48 |
| 6 | Miscellaneous | 6-1 |
| 6.1 | Bug Report | 6-1 |
| 6.2 | Document History | 6-1 |

1 Overview

1.1 General

The PS09 is a system-on-chip for ultra low-power and high resolution applications. It was designed especially for weight scales but fits also to any kind of force or torque measurements based on metal strain gages. It takes full advantage of the digital measuring principle of PICO STRAIN. Thus, it combines the performance of a 28-Bit signal converter with a 24-Bit microprocessor. This volume 2 datasheet describes the PS09 CPU and the instruction set for programming the CPU. In stand-alone operation it is mandatory to have a program running in the CPU, but also in front-end mode, when operated as pure resistance-to-digital converter, the CPU might be used to implement additional data post-processing on chip.

For a general description of the converter front-end, configuration and electrical characteristics please refer to datasheet volume 1.

1.2 Functional Block Diagram

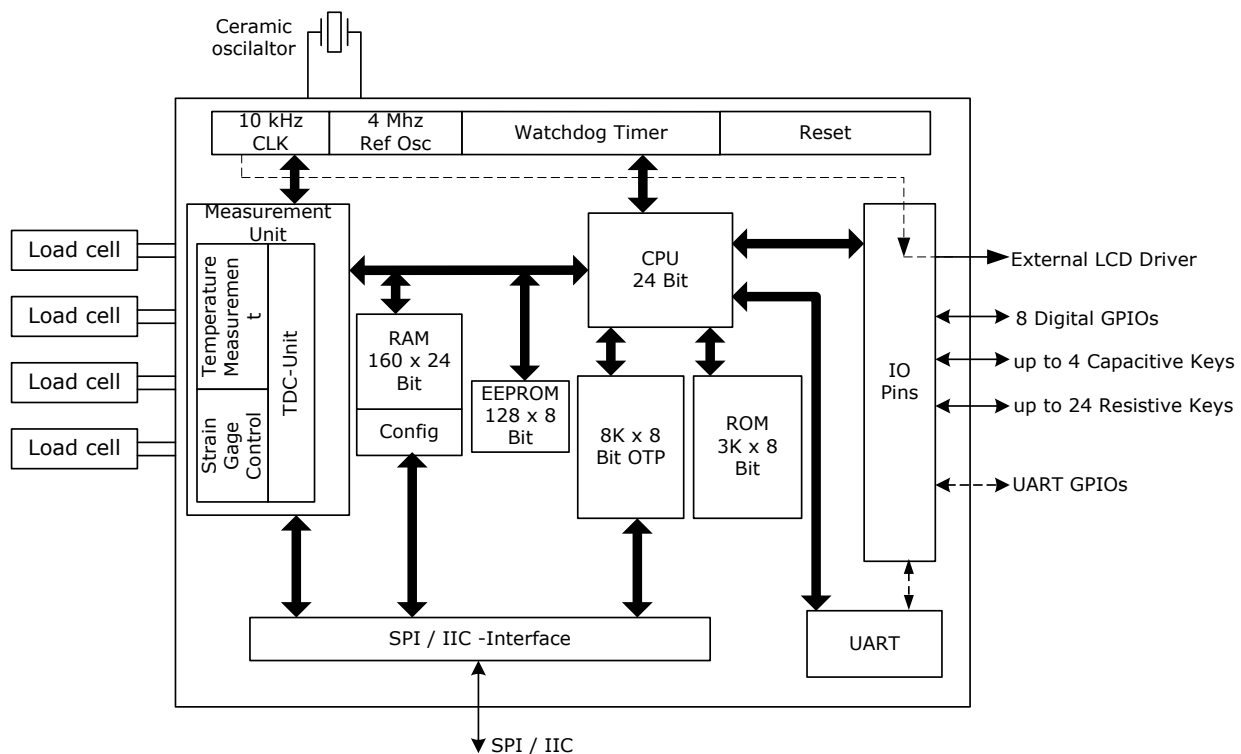


Figure 1-1: PS09 block diagram

2 CPU & Memory

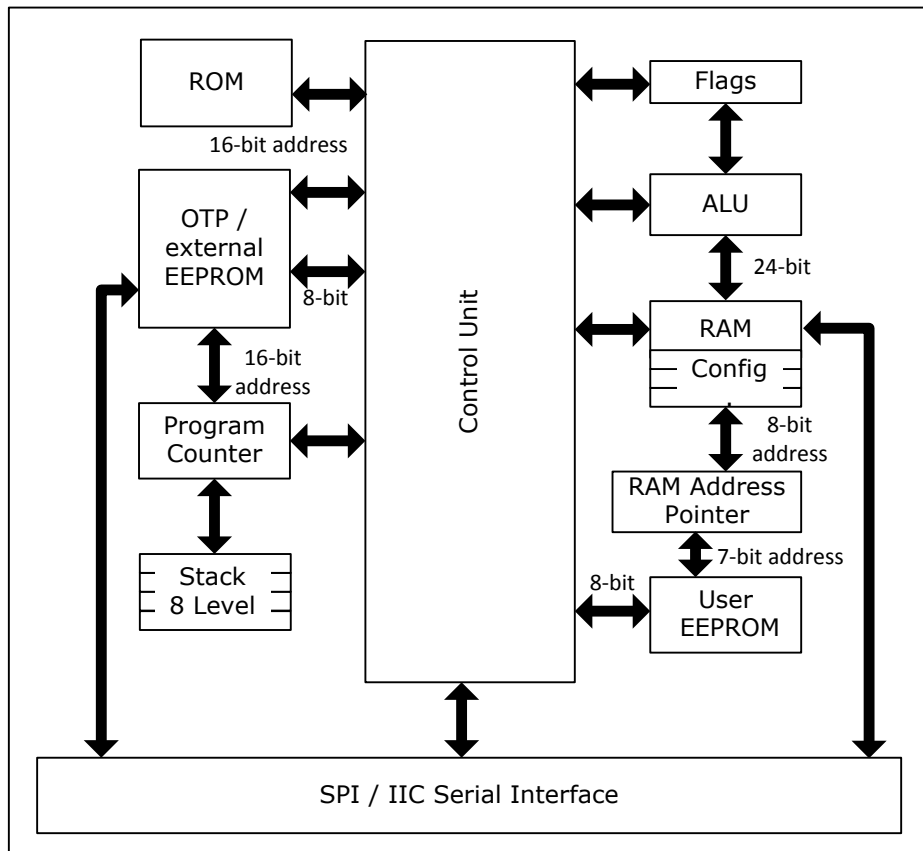
acam designed its own proprietary 24-bit central processing unit. It combines calculation power with ultra-low power operation. Only this special design made it possible to build a system that runs with a few μA current only, but offers complex post-processing of the high-resolution measurement data.

The program itself is stored in an 8k OTP. During development it can be stored alternatively in an external EEPROM.

For effective programming, acam implemented already some special functions like the 48-bit multiplication and division in ROM code.

2.1 Block Diagram

Figure 2-1: Block Diagram



2.2 Memory Organization

Figure 2-2: PS09 Memory Organization

| | | | |
|--|--|---|----|
| FFFF h F000 h | 65535 61440 | ROM Program memory | 4k |
| EEEE h 2000 h | 61439 8192 | Reserved | |
| 1FFF h 2F h 0000 h | 8191 47 0 | User program Memory 8192 bytes of OTP / External EEPROM Configuration, optional (mirrored to RAM) | 8k |

2.2.1 OTP

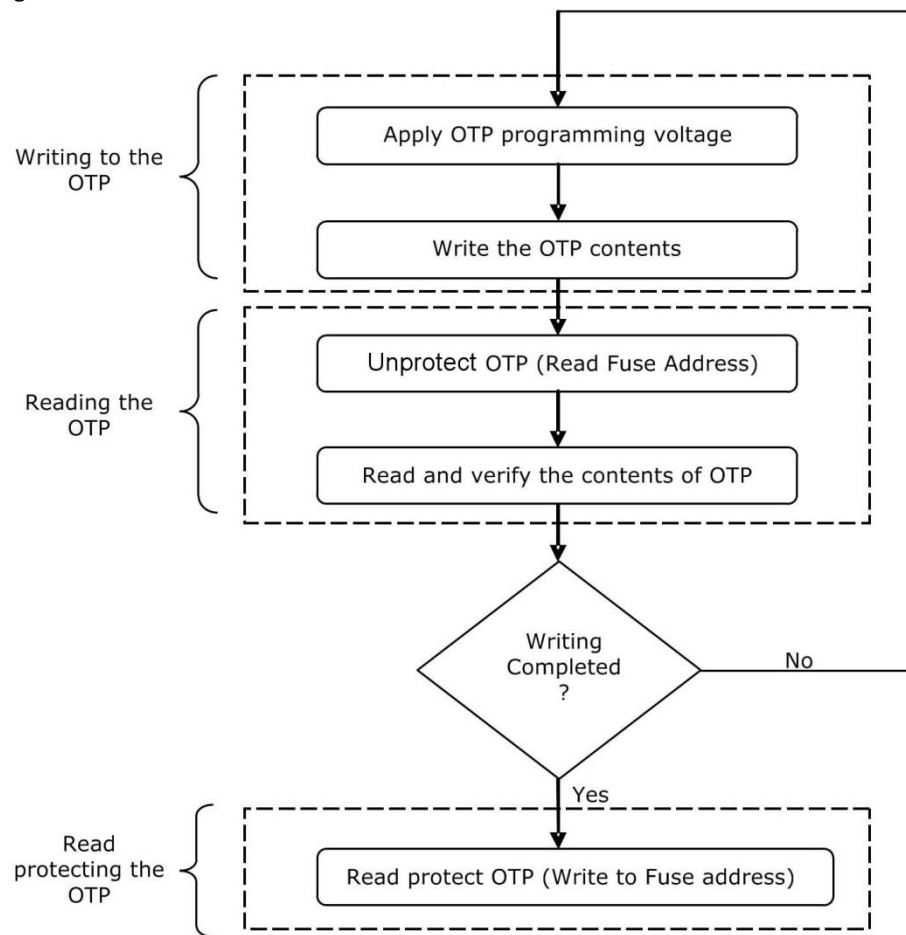
The user program memory in PS09 available for user programming is 8 kbyte in size. This 8 kB user program memory is implemented by an on-chip one time programmable ROM, the OTP. As the name suggests, this memory is writable only once. Hence for development of the user program, the PS09 supports an erasable and re-programmable external EEPROM, maximum 8 kB in size. Once the application program development is complete with the external EEPROM, then the same program can be downloaded into the OTP and it will function in the same manner with the OTP.

(Except prolonged code execution time as described further in 2.2.3).

The first 48 bytes of the OTP from location 0 – 47 are reserved for the configuration data. In order to enable programming of the OTP, an external programming voltage of 6.5 V must be available on pin VPP_OTP of the PS09.

The following flow diagram shows how the OTP is generally handled, details follow in subsequent sections.

Figure 2-3: Using the OTP

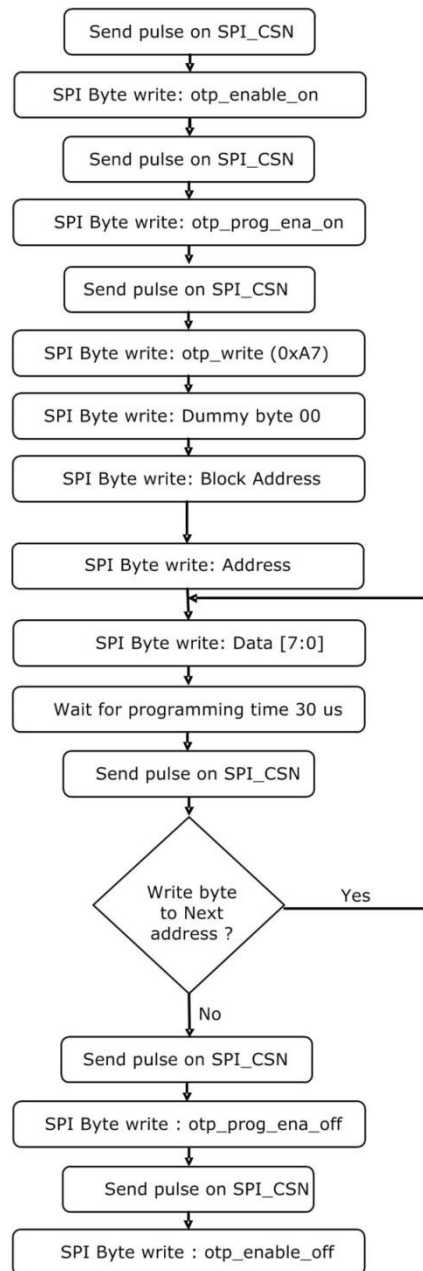


2.2.1.1 Writing to the OTP

The OTP needs an external voltage of 6.5 V on the VPP_OTP pin of the PS09 in order to enable programming. In addition to enabling the OTP, there are op codes to enable and disable the PROG (Enable Programming) signal of the OTP.

The following is a flowchart that shows the SPI command sequence to write a byte to the OTP.

Figure 2-4: Writing to the OTP



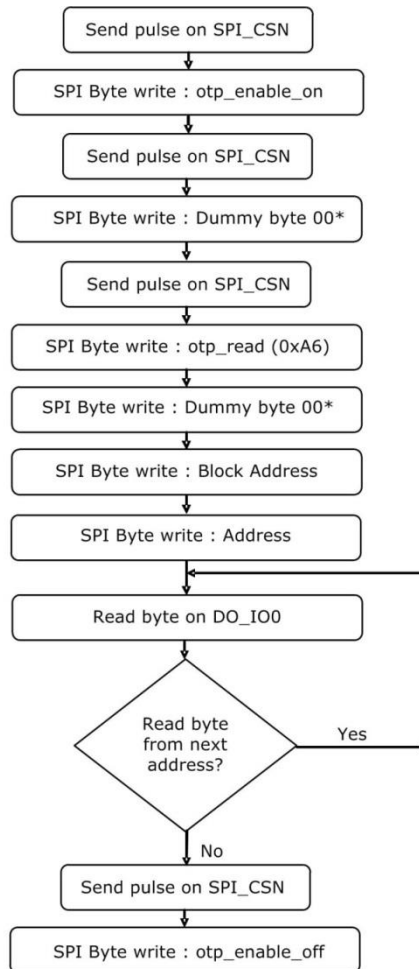
For a list of all op codes pertaining to accessing the OTP through the SPI / IIC interface, refer to Vol.1, Chapter 4, section 4.5.3.3 OTP Access.

2.2.1.2 Reading the OTP

On power on reset, the OTP is by default read protected. An un-programmed OTP content is all 0s. To enable the OTP, the Address 8143, called the Fuse Address must be read first. When the content of the Fuse address is all 0s indicating an un-programmed OTP, then the OTP is enabled for reading, i.e. the OTP is unprotected. Hence this de-protection is the first step in working with the OTP.

The following is a flowchart that shows the general sequence of sending SPI commands to read a byte from the OTP. This is the sequence to be used when controlling the PS09 by an external microcontroller, through the SPI / IIC interface.

Figure 2-5: Reading the OTP



* the Dummy byte (0x00) is required to be sent as it is needed because of timing purposes

For a list of values of all op codes for accessing the OTP through the SPI / IIC interface, refer to Vol.1, Chapter 4, section 4.5.3.3 OTP Access.

2.2.1.3 Read protecting the OTP

Once the OTP has been programmed with the user program and when the code development is complete, the code can be read protected with the Fuse address. For read protecting the OTP, the fuse address 8143 must be written with a non-zero value. The read protection process is completed by reading the address 8143 after writing it with the non-zero value.

2.2.2 External EEPROM

An external EEPROM of up to 8 kB size is supported as user program memory by the PS09 with the sole purpose of supporting user program development. The final program will be written to the on-chip OTP. It is to be noted that the program will be executed in

exactly the same manner, irrespective of whether the user program memory is the OTP or the external EEPROM.

The programming sequence to write a byte into the external EEPROM and to read a byte from the external EEPROM through the SPI / IIC interface can be found under Vol.1, Chapter 4, section 4.5.3.4 External EEPROM Access.

Remark: If no EEPROM is connected, pin 8 (EE_DATA) must be terminated with a capacitance of 100 pF to GND.

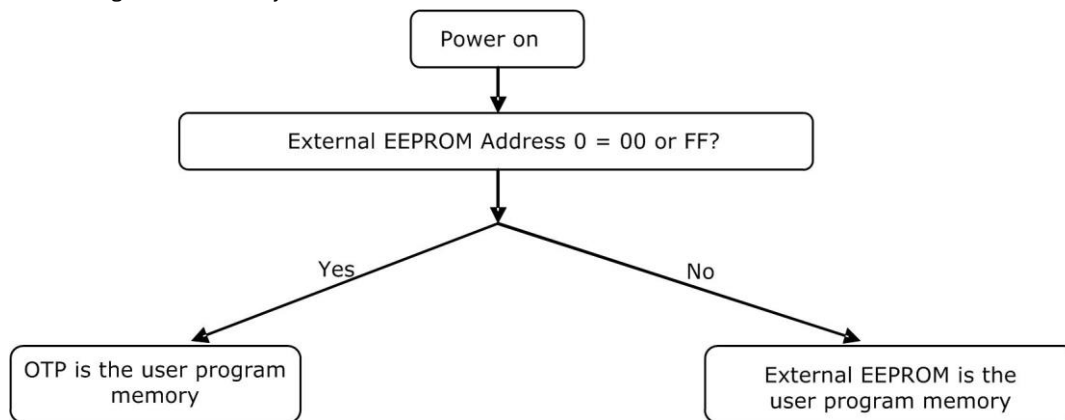
2.2.3 User Program development using the external EEPROM

This section describes how the program can be developed by the user using the external EEPROM as the program memory.

As already stated, basically a user program is executed in the same manner, irrespective of whether the user program memory used is the OTP or the external EEPROM. However the PS09 has to know, which of the two has to be used as the user program memory. For this purpose, as a standard operation on power-up, the PS09 checks for the presence of an external EEPROM by reading address 0 of the external EEPROM. When 00 or FF is read back from address 0 of the EEPROM, then the PS09 takes the internal OTP as the user program memory and executes the code from the OTP. When a value other than 00 and FF is read from the Address 0 of the external EEPROM, then the EEPROM is considered to be the user program memory by the chip and user code in the external EEPROM is executed.

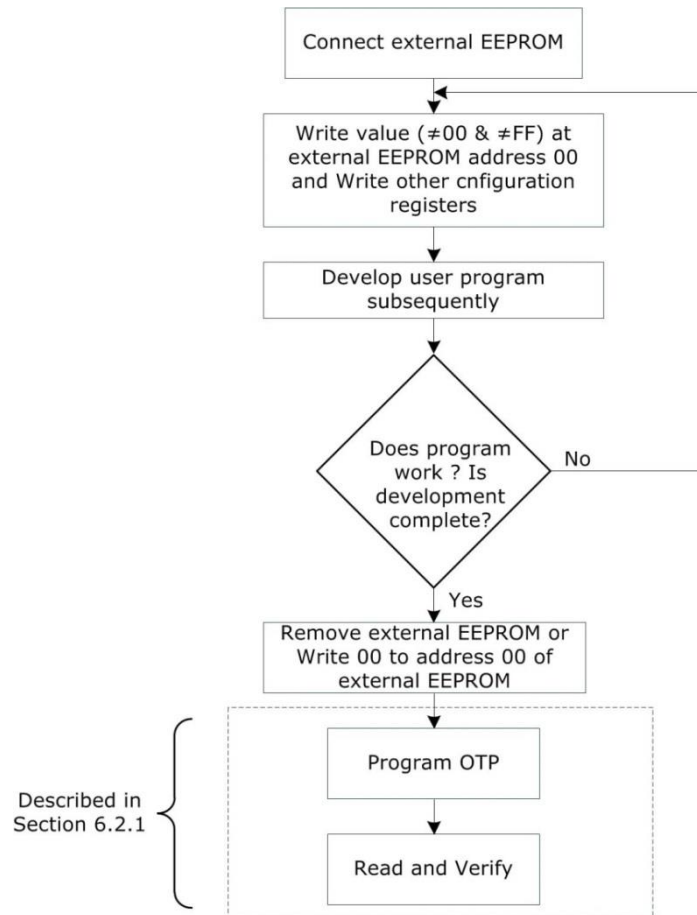
The content of address 00 corresponds to value of the bits 23:16 of Configuration register 0 (tdc_conv_cnt).

Figure 2-6: Program Memory on POR



Once the user program development is completed using the external EEPROM then, the final program is ready to be written to the OTP. Then the external EEPROM is either removed physically or it is made inactive to the PS09 by writing the address 00 of the external EEPROM with 00 or FF. The following flowchart gives an overview of how the user program is developed using the EEPROM and transferred to the OTP finally.

Figure 2-7: User Programm Development



The PS09 Assembler Software which is used for user program development supports downloading the developed program to the external EEPROM or to the on chip OTP. The target for downloading the program can be selected from a drop down list on the Download page of the assembler.

The lower 48 bytes in the user program memory are reserved for an automatic configuration of the PS09 during a power-on reset. 3 successive bytes are added to a 24 bit word. So there are 16 words of 24 bit each that are used for configuration register 0 to 15. During a power- on reset they are copied into RAM addresses 48 to 63.

Generally the code execution from the external EEPROM takes longer than from the internal OTP. This fact needs to be considered when delay routines are realized using incr/decr opcodes in loops as the delay will be longer when executed from the EEPROM in comparison with the OTP. The code execution from the external EEPROM is approx. 10 to 15 times slower than from the internal OTP.

2.2.4 ROM Program memory

In PS09, 4 kbytes is reserved for the ROM starting at address F000 h. All computation routines needed for the PICOSTRAIN measuring method reside here. The program can jump back from the ROM to the OTP/external EEPROM.

2.2.5 User EEPROM

The user EEPROM in PS09 is 128 bytes of 8 bits each. This user EEPROM can be used to store calibration data that can be accessed from the user program. The processor can write to and read from this EEPROM, byte-wise using the putepr and getepr op-codes. This EEPROM hangs on the same address bus as the RAM. Hence the RAM address pointer is used to address both the user EEPROM and the RAM. See section 2.2.7 to get more details with code snippets on how the RAM address pointer is used to address both the user EEPROM and the RAM.

2.2.6 RAM Organization

Table 2-1: RAM address organization

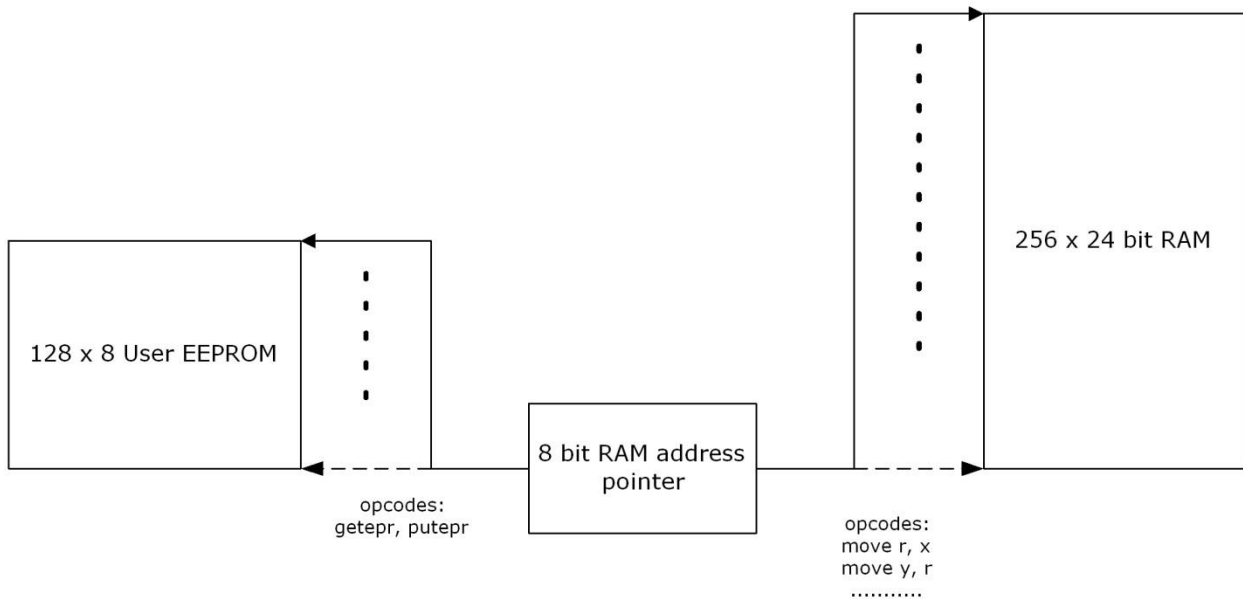
| | | |
|--|--|--|
| 255 ... 240 | Status and Result registers in stand-alone mode (same content as 31 – 16) (e.g. using the DSP Instruction Set) | |
| 239 ... 208 | System RAM | |
| 207 ... 96 | User RAM 207 ... User RAM 96 | |
| 95 ... 92 | Reserved | |
| 91 ... 86 | UART Config / status reg | |
| 85 ... 81 | Internal registers | |
| 80 | UART Config / status reg | |
| 79... 64 | Reserved for internal use | |
| 63 48 | Config reg 15 Config reg 0 | |
| 47 32 | User RAM 47 User RAM 32 | |
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 | Modrspar result Timer I/O status – falling, rising and pressed status of the 8 GPIO s Status of the 24 Multi Input keys, Pressed or Released Status : rising edge on the 24 Multi Input keys Status : falling edge on the 24 Multi Input keys UBATT CAL HB1+ Status flags TMP $HBO = 1/4 * (HB1+HB2+HB3+HB4)$ $HB4 = (G-H) / (G+H) *$ $HB3 = (E-F) / (E+F) *$ $HB2 = (C-D) / (C+D) *$ $HB1 = (A-B) / (A+B) *$ | User RAM 31 ... 16 Status and Result registers in front end mode; (e.g. using external µC) |
| 15 0 | User RAM 15 User RAM 0 | |

* Parameters A..H represent the discharging times at the different ports, see section 2.4.1 Result Registers for more details

2.2.7 RAM Address Pointer

The RAM has its own address bus with 256 addresses. The width of 24 bit corresponds to the register width of the ALU. By means of the RAM address pointer a single RAM address is mapped into the ALU. It then acts as a fourth accumulator register. Changing the RAM address pointer does not affect the content of the addressed RAM. The RAM address pointer itself is modified by separate opcodes (ramadr, increadr, ...). As explained in the previous section, the RAM address bus is additionally used to address 128 bytes of user EEPROM with particular op codes.

Figure 2-8: RAM Address Pointer



When the RAM address pointer is set to a value and op codes putepr and getepr are used, the RAM address pointer points to the respective byte in the user EEPROM. Hence operations are carried out with the respective user EEPROM byte. All other op codes like move r, x set the RAM address pointer to point to the RAM, hence the operation is performed in the RAM.

The following sample code illustrates how the RAM address pointer is used to access the user EEPROM and the RAM, based on the op code used.

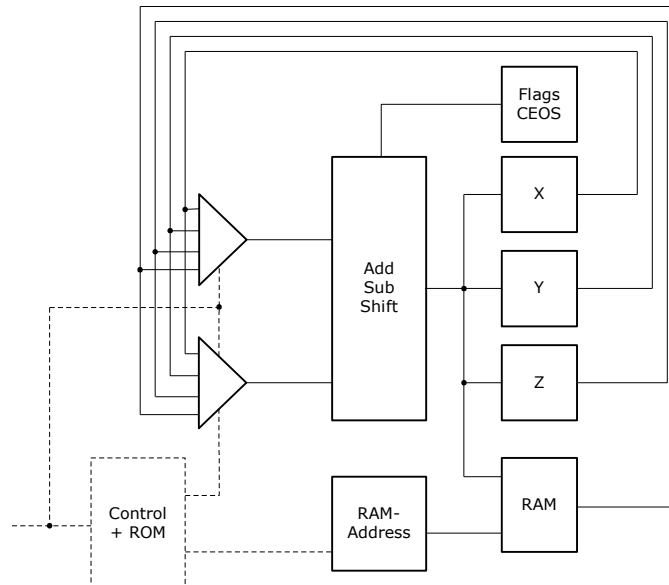
Sample code:

```
Ramadr      3      // Sets the RAM address pointer to address 3
Move        r, x    // Moves the content of the X accumulator to the RAM address 3
                // RAM Address Pointer is pointing to the RAM
```


| | | |
|---------|---|---|
| Ramadr | 4 | // Sets the RAM address pointer to address 4 |
| Geteptr | x | // Gets the content of the user EEPROM address 4 into the X // accumulator |
| | | // RAM Address Pointer is pointing to the user EEPROM |
| Ramadr | 3 | // Sets the RAM address pointer to address 3 |
| Putepr | x | // Moves the content of the X accumulator to the user EEPROM // address 3 |
| | | // RAM Address Pointer is pointing to the user EEPROM |
| Clear | r | // Clears the content of RAM address 3 |
| | | // RAM Address Pointer is pointing to the RAM |

2.3 Arithmetic Logic Unit (ALU)

Figure 2-9: ALU block diagram



2.3.1 Accumulators

The ALU has three 24-Bit accumulators, X, Y and Z. The RAM is addressed by the RAM address pointer and the addressed RAM cell is used as forth accumulator. A single RAM address is mapped into the ALU by the ram address pointer. So in total there are 4 accumulators. All transfer operations (move, swap) and arithmetic-operations (shift, add, mult24, ...) can be applied to all accumulators.

2.3.2 Flags

The processor controls 4 flags with each operation. Not-Equal and Sign flags are set with each write access to one of the accumulators (incl. RAM). Additionally, the Carry and Overflow flags are set in case of a calculation (Add / Sub / shiftR). It is possible to query each flag in a jump instruction.

2.3.2.1 Carry

Shows the carry over in an addition or subtraction. With shift operations (shiftL, rotR, etc.) it shows the bit that has been shifted out.

2.3.2.2 Not-Equal zero

This flag is set to zero in case a new result not equal to zero is written into an accumulator (add, sub, move ,swap, etc.).

2.3.2.3 Sign

The sign is set when a new result is written into an accumulator (add, sub, move, swap, etc.) and the highest bit (MSB) is 1.

2.3.2.4 Overflow

Indicates an overflow during an addition or subtraction of two numbers in two's complement representation.

2.4 Status and Result Registers

2.4.1 Result Registers

Content of the RAM result registers at the end of a measurement:

| | | |
|------------|-------------------------------|--|
| ram = 16 : | HB1 = (A-B) / (A+B) | HB1 un-compensated |
| ram = 17 : | HB2 = (C-D) / (C+D) | HB2 un-compensated |
| ram = 18 : | HB3 = (E-F) / (E+F) | HB3 un-compensated |
| ram = 19 : | HB4 = (G-H) / (G+H) | HB4 un-compensated |
| ram = 20 : | HBO = 1/4 * (HB1+HB2+HB3+HB4) | HBO compensated sum |
| ram = 21 : | TMP = RTemp / Rsg | Temperature measurement value, see Vol.1, Chapter 3, Section 3.5.10 Internal Temp. Measurement |
| ram = 22 : | Status flags | See section 2.4.2 Status Register |
| ram = 23 : | HB1+ | Time measurement TDC at SG_A1, Pin11 |

| | | |
|------------|----------------|---|
| ram = 24 : | CAL | Resolution TDC |
| ram = 25 : | UBATT | Measured supply voltage |
| ram = 26 : | Status_Multi_F | Indicates falling edge occurrence on 24 possible Multi Input keys |
| ram = 27 : | Status_Multi_R | Indicates rising edge occurrence on 24 possible Multi Input keys |
| ram = 28 : | Status_Multi_P | Status of the 24 Multi Input keys, Pressed or Released |
| ram = 29 : | Status_IO | Falling, Rising and Current Status of 8 GPIO pins |
| ram = 30 : | Timer | Status of the timer on measurement completion |
| ram = 31 : | Modrspan | Rspan value on measurement completion. For load cells with Rspan, the ratio Rspan/Rsg when bit mod_rspan = 1 in Config_reg1. |

Descriptions:

| | |
|---------|--|
| A : | Discharge time measurement at SG_A1 |
| B : | Discharge time measurement at SG_A2 |
| C : | Discharge time measurement at SG_B1 |
| D : | Discharge time measurement at SG_B2 |
| E : | Discharge time measurement at SG_C1 |
| F : | Discharge time measurement at SG_C2 |
| G : | Discharge time measurement at SG_D1 |
| H : | Discharge time measurement at SG_D2 |
| RTemp : | Discharge time measurement through the combination of Integrated Rspan and strain gage resistor at SG_D1 and SG_C2 |
| Rsg : | Discharge time measurement at SG_D1 SG_C2 |

Formats:

| | |
|----------|---|
| HB1 : | Result in $\frac{1}{100} ppm$ |
| HB2 : | Result in $\frac{1}{100} ppm$ |
| HB3 : | Result in $\frac{1}{100} ppm$ |
| HB4 : | Result in $\frac{1}{100} ppm$ |
| HBO : | Result in $\frac{1}{100} ppm$ |
| TMP : | current ratio CR by $1 + \frac{TMP}{2^{20}}$ |
| Status : | See above |
| HB1+ : | Result in $250 * \frac{SG_{A1}}{2^{14}} ns$ @ 4 MHz clock |
| CAL : | Calculation of Resolution by $\frac{250,000}{CAL} ps$ @ 4 MHz clock |
| UBATT : | Calculation of Supply Voltage by $2.0 + 1.6 * \frac{UBATT}{64} V$ |

HB1, HB2, HB3, HB4, HBO and TMP are given as two's complement. MSB = 1 indicates a negative value. To get the positive value calculate $2^{24} - X$.

Explanation:

Based on a standard extension of a load cell (2 mV/V) the resistance variation is 0.2 %, e.g. 2 Ω at a 1000 Ω load cell. The change of 0.2 % corresponds to 2000 ppm. For reasons of internal calculations and accuracy, the result is given in x100 of 2000 ppm (= 200,000 ppm). Please note that the value in this register depends not only on the load cell's sensitivity but also on the Mult_HBx setting in PS09. This explanation is based on Mult_HBx = 1.

Examples:

1.5 mV/V load cell, PICOSTRAIN wiring, Mult_HBx = 1:

1.5 mV/V = 1500 ppm → result in PS09 at maximum strain: 150,000 (0x0249F0)

2 mV/V load cell, Wheatstone wiring, Mult_HBx = 1:

2 mV/V means 1.333 mV/V in Wheatstone = 1333 ppm (due to a reduction in strain) →
result in PS09 at maximum strain: 133,333 (0x0208D5)

1 mV/V load cell, PICOSTRAIN wiring, Mult_HBx = 4:

1 mV/V = 1000 ppm → result in PS09 at maximum strain: 400,000 (0x061A80)

2.4.2 Status Register

Table 2-2: Status Register (RAM Address 246)

| Bit | Description |
|--------------------------------|---|
| Status[23] = flg_status_cport4 | Status flag of capacitive port 4 |
| Status[22] = flg_status_cport3 | Status flag of capacitive port 3 |
| Status[21] = flg_status_cport2 | Status flag of capacitive port 2 |
| Status[20] = flg_status_cport1 | Status flag of capacitive port 1 |
| Status[19] = flg_rstpwr | 1 = Power-on reset caused jump into OTP / ext. EEPROM |
| Status[18] = flg_rstssn | 1 = Pushed button caused jump into OTP / ext. EEPROM |
| Status[17] = flg_wdtalt | 1 = Watchdog interrupt caused jump into OTP / ext. EEPROM |
| Status[16] = flg_endavg | 1 = End of measurement caused jump into OTP / ext. EEPROM |
| Status[15] = flg_intavO | 1 = Jump into OTP / ext. EEPROM in sleep mode |
| Status[14] = flg_ub_low | 1 = Low voltage |
| Status[13] = flg_errtdc | 1 = TDC error |
| Status[12] = reserved | 1 = reserved |
| Status[11] = flg_err_cport | 1 = Error at capacitive ports |
| Status[10] = flg_errprt | 1 = Error at strain gauge ports |
| Status[09] = flg_timeout | 1 = Timeout TDC |
| Status[08] = flg_ext_interrupt | 1 = DSP start by external interrupt |
| Status[07] = flg_cport4_r | 1 = Rising edge at capacitive port 4, 0 = no edge |
| Status[06] = flg_cport3_r | 1 = Rising edge at capacitive port 3, 0 = no edge |
| Status[05] = flg_cport2_r | 1 = Rising edge at capacitive port 2, 0 = no edge |
| Status[04] = flg_cport1_r | 1 = Rising edge at capacitive port 1, 0 = no edge |
| Status[03] = flg_cport4_f | 1 = Falling edge at capacitive port 4, 0 = no edge |
| Status[02] = flg_cport3_f | 1 = Falling edge at capacitive port 3, 0 = no edge |
| Status[01] = flg_cport2_f | 1 = Falling edge at capacitive port 2, 0 = no edge |
| Status[00] = flg_cport1_f | 1 = Falling edge at capacitive port 1, 0 = no edge |

The status of the inputs can be queried from the status registers at RAM address 250 to 252. Please see Vol.1, Chapter 4, Section 4.3.3 Multi-input keys for more details.

3 General Functions

3.1 System Reset, Sleep Mode and Auto-configuration

ALU activity is requested by a reset (power-on, watchdog), the end of measurement or in sleep mode the end of the conversion counter. A reset has priority over the other two items. First the ALU jumps into the ROM code starting with address 0000 h. There a first check is done whether the ALU was activated after a reset or not.

In case of a reset, the flag `otp_pwr_cfg` is checked to decide whether the auto-configuration data from the OTP/external EEPROM have to be copied into the RAM or not.

Subsequently, the flag `otp_pwr_prg` is checked to decide whether OTP/ external EEPROM user code (starting at address 48) ought to be executed. In stand-alone operation this is reasonable and `otp_pwr_cfg` bit should be 1. In front end operation this is unlikely and with `otp_pwr_cfg = 0` the μ P is stopped.

In case the ALU is started not by a reset the TDC unit starts a measurement or, in sleep mode, the conversion counter is started without a measurement. Afterwards the flag `otp_usr_prg` is checked to decide whether a jump into the user code in OTP/external EEPROM (address 48) must be performed or not. Again, in stand-alone operation `otp_usr_prg = 1` is reasonable, in front-end operation `otp_usr_prg = 0` will be more likely.

In the user code in the OTP / external EEPROM first the flag `flg_rstpwr` should be checked to see whether the reason for the jump was a reset. If yes, a detailed check is recommended to see whether the reset comes from a power-on reset, a pushed button, the watchdog interrupt.

Otherwise a check of flag `flg_intav0` will indicate if the chip is still in sleep mode or if an active strain measurement is running.

At the end the ALU is stopped. This implements a complete reset of the ALU including the start flags. Also the program stack is reset. Only the RAM data remain unchanged.

3.1.1 Power-On Reset

When applying the supply voltage to the chip a power-on reset is generated. The whole chip is reset, only the RAM remains unchanged.

In case `otp_pwr_prg = 1` the user code at EEPROM address 48 is started.

3.1.2 Watchdog Reset

A power-on reset can also be triggered by the watchdog timer. This happens in case the microprocessor is started four times without being reset by the opcode "clrwtd". Status bit `flg_wdtalt` in register 224+22; bit 17 indicates a timeout of the watchdog timer.

In case `otp_pwr_prg = 1` the user code at EEPROM address 48 is started.

3.1.3 External Reset on Pin 6

In stand-alone mode (if Mode pin is unconnected) it is possible to apply an external power-on at pin 6 (SPI_CSN_RST). This can be used as a reset button. The status of the button can be requested from status bit `flg_rstssn` in register 224+22, bit 18.

In case `otp_pwr_prg = 1` the user code at EEPROM address 48 is started.

3.1.4 Sleep Mode

In sleep mode only the 10 kHz oscillator is running. At regular intervals the microprocessor is waked up but without doing a measurement. In this phase it can check the I/Os. A start-up of the microprocessor from sleep mode is indicated by status bit `flg_intav0` in register 224+22, bit 22.

| | | |
|----------------|---------------------------------|---------------------------|
| Configuration: | <code>tdc_sleepmode</code> | Register 1, Bit 17 |
| | <code>tdc_conv_cnt [7:0]</code> | Register 0, Bits 23 to 16 |

Note : The sleep mode works only in combination with `Single_conversion = 1` in `Configreg_02`

Sleep mode is activated by setting `tdc_sleepmode = 1`. This is equivalent to set `avrate = 0`.

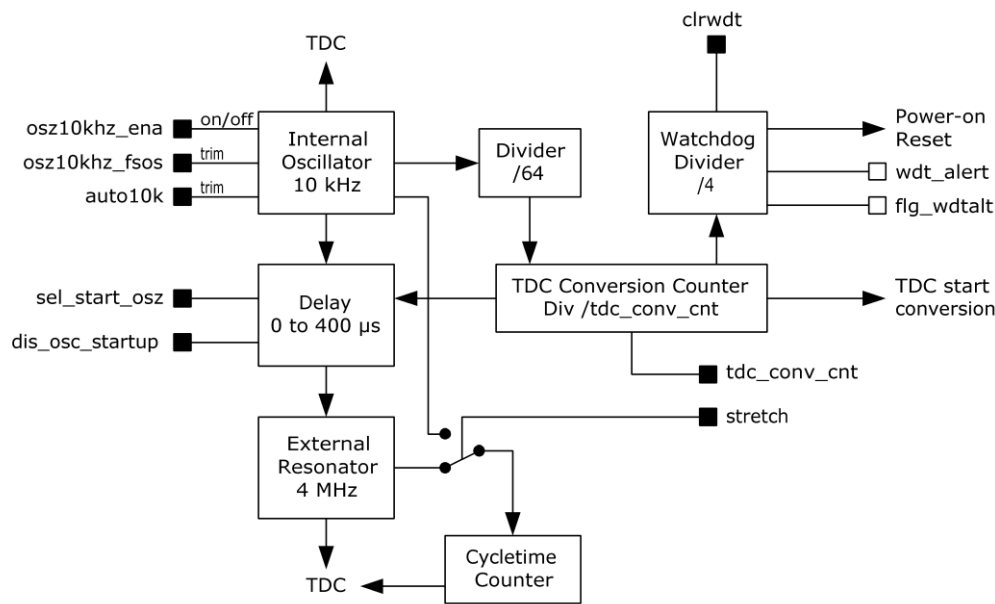
In sleep mode the conversion counter `tdc_cnv_cnt` is running to the end and then immediately starting the user program beginning at address 48 in the EEPROM.

After running in sleep mode the TDC has to be reinitialized for measurements.

3.2 CPU Clock Generation

The basic clock for the system is the internal, low-current 10 kHz oscillator. It is used to trigger measurements in single conversion mode for the TDC unit in measurement range 2 as pre-counter as basis for the cycle time in stretched modes.

Figure 3-1: Clock Generation



3.3 Watchdog Counter and Single Conversion Counter

The TDC conversion counter starts a measurement in single conversion mode. It is running continuously. The single conversion rate is given by $10 \text{ kHz} / 64 / \text{tdc_conv_cnt}$.

With the beginning of a measurement the watchdog counter is increased. The watchdog counts the conversions. At the end of a measurement the microprocessor starts to run the user code. In normal operation the watchdog has to be reset by CLRWDT before the user code ends. The watchdog causes a power-on reset in case the TDC doesn't finish its measurement because of an error or the user code does not run to end.

It is possible to switch off the watchdog when controlling the PSØ9 by the SPI interface (Mode pin is connected to 0) sending SPI opcode watch_dog_off. Further the watchdog is reset by each signal edge at the SPI_CSN_RST pin.

3.4 Timer

PSØ9 has a real time counter that counts automatically after a power-on reset in periods of 12.8 ms. The value of this timer can be read out at address 254, it is updated at the end of each measurement. The counter rolls over at 2^{24} bit, which corresponds to a period of 46 hours

4 Instruction Set

The complete instruction set of the PS09 consists of 69 core instructions that have unique op-codes decoded by the CPU.

4.1 Branch instructions

There are 3 principles of jumping within the code:

Jump. Absolute addressing within the whole address space of 8 kB.

Branch. Relative to the actual address, jump within the address range of -128 to +127.

Skip. Jump ahead up to 3 op-codes (3 to 15 bytes).

The assembler puts together jump and branch into goto-instructions.

It is possible to jump into subroutines only by means of absolute jumps and without any condition.

4.2 Arithmetic operations

The RAM is organized in 24 Bit words. All instructions are based on two's complement operations. An arithmetic command combines two accumulators and writes back the result into the first mentioned accumulator. The RAM address pointer points to the RAM address that is handled in the same way as an accumulator. Each operation on the accumulator affects the four flags. The status of the flags refers to the last operation.

Table 4-1: Instruction set

| Simple Arithmetic | Complex Arithmetic | Shift & Rotate | RAM access |
|-------------------|--------------------|----------------|------------|
| abs | div24 | clrC | clear |
| add | divmod | rotl | decramadr |
| compare | mult24 | rotR | incramadr |
| compl | mult48 | setC | move |
| decr | | shiftL | ramadr |
| getflag | | shiftR | swap |
| incr | | | |
| sign | | | |
| sub | | | |

| Logic | Bitwise | EEPROM access OTP/external EPROM |
|--------|---------|-------------------------------------|
| and | bitclr | equal |
| eor | bitinv | getepr |
| nor | bitset | putepr |
| invert | | addepr |
| nand | | |
| nor | | |
| or | | |

| Unconditional jump | Skip on Flag | Miscellaneous |
|--------------------|--------------|---------------|
| skip | | |
| goto | | clk10kHz |
| gotoBitC | skipBitC | clrwdt |
| gotoBitS | skipBitS | nop |
| gotoCarC | skipCarC | stop |
| gotoCarS | skipCarS | initTDC |
| gotoEQ | skipEQ | newcyc |
| gotoNE | skipNE | |
| gotoNeg | skipNeg | |
| gotoOvrC | skipOvrC | |
| gotoOvrS | skipOvrS | |
| gotoPos | skipPos | |
| jsub | | |
| jsubret | | |

| abs | Absolute value of register |
|-----------------|----------------------------|
| Syntax: | abs p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | $p1 = p1 $ |
| Flags affected: | C O S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Absolute value of register |
| Category: | Simple arithmetic |

| add | Addition |
|-----------------|---|
| Syntax: | add p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | $p1 = p1 + p2$ |
| Flags affected: | C O S Z |
| Bytes: | 1 (p2 = ACCU) 4 (p2 = number) |
| Cycles: | 1 (p2 = ACCU) 4 (p2 = number) |
| Description: | Addition of two registers or addition of a constant to a register |
| Category: | Simple arithmetic |

| addepr | |
|---------------|---|
| Syntax: | addepr x |
| Parameters: | ACCU[x] |
| Calculus: | $x = x + \text{Value (EEPROM(rampointer))}$ |
| Flags: | Z S C O |
| Bytes: | 2 |
| Cycles: | 100..200 |
| Description: | Adds the value from the content of the EEPROM register, currently addressed by the ram address pointer, to the X-Accumulator. |
| Category: | EEPROM access |

| and | Logic AND |
|-----------------|---|
| Syntax: | and p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | $p1 = p1 \text{ AND } p2$ |
| Flags affected: | S Z |
| Bytes: | 2 (p2 = ACCU) |
| 5 (p2 = number) | |
| Cycles: | 3 (p2 = ACCU) |
| 6 (p2 = number) | |
| Description: | Logic AND of 2 registers or Logic AND of register and constant |
| Category: | Logic |

| bitclr | Clear single bit |
|-----------------|--|
| Syntax: | bitclr p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = number 0 to 23 |
| Calculus: | $p1 = p1 \text{ and not } (1 \ll p2)$ |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Clear a single bit in the destination register |
| Category: | Bitwise |

| bitinv | Invert single bit |
|-----------------|---|
| Syntax: | bitinv p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = number 0 to 23 |
| Calculus: | $p1 = p1 \text{ eor } (1 \ll p2)$ |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Invert a single bit in the destination register |
| Category: | Bitwise |

| bitset | Set single bit |
|-----------------|---|
| Syntax: | bitset p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = number 0 to 23 |
| Calculus: | $p1 = p1 \text{ or } (1 \ll p2)$ |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Set a single bit in the destination register |
| Category: | Bitwise |

| clear | Clear register |
|-----------------|------------------------|
| Syntax: | clear p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | $p1 = 0$ |
| Flags affected: | S Z |
| Bytes: | 1 |
| Cycles: | 1 |

| | |
|--------------|-------------------------------|
| Description: | Clear addressed register to 0 |
| Category: | RAM access |

| | |
|-----------------|--|
| clk10khz | Clock source 10 kHz |
| Syntax: | clk10khz p1 |
| Parameters: | p1 = number 0 or 1 |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 2 |
| Cycles: | 3 |
| Description: | Change clock source of processor to 10 kHz. The clock of the processor is switched to the slower 10 kHz clock instead of the 40 MHz. The 10 kHz clock is still stable to variations in temperature and supply voltage. If p1 is set to 1 the 10 kHz clock is on, if p1 == 0 the 10 kHz clock is off. With the 10 kHz clock beeper application at the IO-Port may programmed with the microcontroller. Do not switch directly between CLK4MHz and CLK10kHz. |
| Category: | Miscellaneous |

| | |
|-----------------|--------------------------------|
| clrC | Clear flags |
| Syntax: | clrC |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | C 0 |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Clear Carry and Overflow flags |
| Category: | Shift and Rotate |

| | |
|-----------------|---|
| clrwdt | Clear watchdog |
| Syntax: | clrwdt |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 2 |
| Cycles: | |
| Description: | Clear watchdog. This opcode is used to clear the watchdog at the end of a program run. Apply this opcode right before ,stop'. |
| Category: | Miscellaneous |

| | |
|----------------|---------------------------|
| compare | Compare two values |
|----------------|---------------------------|

| | |
|-----------------|--|
| Syntax: | compare p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | $- = p2 - p1$ only the flags are changed but not the registers |
| Flags affected: | C O S Z |
| Bytes: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Description: | Comparison of 2 registers by subtraction. Comparison of a constant with a register by subtraction The flags are changed according to the subtraction result, but not the registers contents themselves |
| Category: | Simple arithmetic |

| | |
|-----------------|----------------------------------|
| compl | Complement |
| Syntax: | compl p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | $p1 = - p1 = \text{not } p1 + 1$ |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | two's complement of register |
| Category: | Simple arithmetic |

| | |
|-----------------|------------------------|
| decr | Decrement |
| Syntax: | decr p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | $p1 = p1 - 1$ |
| Flags affected: | C O S Z |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Decrement register |
| Category: | Simple arithmetic |

| | |
|------------------|--------------------------------------|
| decramadr | Decrement RAM address pointer |
| Syntax: | decramadr |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 1 |

| | |
|--------------|--------------------------------------|
| Cycles: | 1 |
| Description: | Decrement RAM address pointer by one |
| Category: | Ram Access |

| | |
|-----------------|--|
| div24 | Signed division 24 Bit |
| Syntax: | div24 p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] |
| Calculus: | $p1 = (p1 \ll 24) / p2$ (if $ p1 < p2/2 $) |
| Flags affected: | S & Z of p1 |
| Bytes: | 2 |
| Cycles: | 20 |
| Description: | Signed division of 2 registers, 24 bits of the division of 2 registers, result is assigned to p1 |
| Category: | Complex arithmetic |

| | |
|---------------------|--|
| divmodSigned | modulo division |
| Syntax: | divmod p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] |
| Calculus: | $p1 = p1 / p2$ and $p2 = p1 \% p2$ |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | |
| Description: | Signed modulo division of 2 registers, 24 higher bits of the division of 2 registers, result is assigned to p1, the rest is placed to p2 |
| Category: | Complex arithmetic |

| | |
|-----------------|---|
| eor | Exclusive OR |
| Syntax: | eor p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | $p1 = p1 \text{ xor } p2$, bit combination 0 / 0 and 1 / 1 returns 0, bit combination 0 / 1 and 1 / 0 returns 1 |
| Flags affected: | S Z |
| Bytes: | 2 (p1 = ACCU, p2 = ACCU) 5 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 3 (p1 = ACCU, p2 = ACCU) 6 (p1 = ACCU, p2 = NUMBER) |
| Description: | Logic XOR (exclusive OR, antivalence) of the 2 given registers Logic XOR (exclusive OR, antivalence) of register with constant |

Category: | Logic

| eorn | Exclusive NOR |
|-----------------|---|
| Syntax: | eorn p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | p1 = p1 xnor p2, bit combination 0 / 0 and 1 / 1 return 1, bit combination 0 / 1 and 1 / 0 return 0 |
| Flags affected: | S Z |
| Bytes: | 2 (p1 = ACCU, p2 = ACCU) 5 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 3 (p1 = ACCU, p2 = ACCU) 6 (p1 = ACCU, p2 = NUMBER) |
| Description: | Logic XNOR (exclusive NOR, equivalence) of the 2 given registers Logic XNOR (exclusive NOR, equivalence) of register with constant |
| Category: | Logic |

| equal | Write 3 Bytes to the OTP or the external EEPROM |
|-----------------|--|
| Syntax: | equal p1 |
| Parameters: | p1 = 24-Bit number |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 3 |
| Cycles: | |
| Description: | Write 3 bytes (p1) to configuration register of OTP/external EEPROM. The equal opcode is used to write 3 bytes of configuration data directly to a register. Therefore the opcode is simply used 16 times in the beginning of the assembler listing, fed with the configuration data given through p1. The configuration of the OTP/ external EEPROM is done in the lower area from byte 0.47, combined in 16x 24bit registers. From byte 48 upwards, the user code is written. Use this opcode to provide your own configuration instead of the standard configuration. |
| Category: | OTP/ External EEPROM access |

| getepr | Get EEPROM content |
|-----------------|---|
| Syntax: | getepr p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | p1 = EEPROM register content (addressed by RAM address pointer) |
| Flags affected: | S Z |
| Bytes: | 1 |
| Cycles: | 6 |
| Description: | Get EEPROM into register. The addressed register p1 gets the EEPROM |

| | |
|-----------|---|
| | register content which is addressed by the RAM address pointer. This opcode needs temporarily a place in the program counter stack (explanation see below). |
| Category: | EEPROM Access |

| | |
|-----------------|---|
| getflag | Set S and Z flags |
| Syntax: | getflag p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | signum = set if p1 < 0 notequalzero = set if p1 <> 0 |
| Flags affected: | S Z |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Set the signum and notequalzero flag according to the addressed register, content of the register is not affected |
| Category: | Simple arithmetic |

| | |
|-------------------|--|
| goto | jump without condition |
| Syntax: | goto p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) |
| 3 (absolute jump) | |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump without condition. Program counter is set to target address. The target address is given by using a jump label. Jump range: 0 < address < 8 kB See examples section for how to introduce a jump label. |
| Category: | Unconditional jump |

| | |
|-----------------|--|
| gotoBitC | Jump on bit clear |
| Syntax: | gotoBitC p1, p2, p3 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = NUMBER [0...23] p3 = JUMPLABEL |
| Calculus: | if (bit p2 of register p1 == 0) PC = p3 |
| Flags affected: | - |
| Bytes: | 3 |
| Cycles: | 4 |

| | |
|--------------|--|
| Description: | Jump on bit clear. Program counter will be set to target address if selected bit in register p1 is clear. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Bitwise |

| | |
|-----------------|--|
| gotoBitS | Jump on bit set |
| Syntax: | gotoBitS p1, p2, p3 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = NUMBER [0..23] p3 = JUMPLABEL |
| Calculus: | if (bit p2 of register p1 == 1) PC = p3 |
| Flags affected: | - |
| Bytes: | 3 |
| Cycles: | 4 |
| Description: | Jump on bit set. Program counter will be set to target address if selected bit in register p1 is set. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Bitwise |

| | |
|-----------------|--|
| gotoCarC | Jump on carry clear |
| Syntax: | gotoCarC p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (carry == 0) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump on carry clear. Program counter will be set to target address if carry is clear. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoCarS | Jump on carry set |
| Syntax: | gotoCarS p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (carry == 1) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) |

| | |
|--------------|--|
| | 4 (absolute jump) |
| Description: | Jump on carry set. Program counter will be set to target address if carry is set. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoEQ | Jump on equal zero |
| Syntax: | gotoEQ p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (Z == 0) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump on equal zero. Program counter will be set to target address if the foregoing result is equal to zero. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoNE | Jump on not equal zero |
| Syntax: | gotoNE p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (Z == 1) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump on not equal zero. Program counter will be set to target address if the foregoing result is not equal to zero. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoNeg | Jump on negative |
| Syntax: | gotoNeg p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (S == 1) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) |

| | |
|--------------|---|
| | 4 (absolute jump) |
| Description: | Jump on negative. Program counter will be set to target address if the foregoing result is negative. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoOvrC | Jump on overflow clear |
| Syntax: | gotoOvrC p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (O == 0) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump on overflow clear. Program counter will be set to target address if the overflow flag of the foregoing operation is clear. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoOvrS | Jump on overflow set |
| Syntax: | gotoOvrS p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (O == 1) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) 4 (absolute jump) |
| Description: | Jump on overflow set. Program counter will be set to target address if the overflow flag of the foregoing operation is set. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|--|
| gotoPos | Jump on positive |
| Syntax: | gotoPos p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | if (S == 0) PC = p1 |
| Flags affected: | - |
| Bytes: | 2 (relative jump) 3 (absolute jump) |
| Cycles: | 3 (relative jump) |

| | |
|--------------|---|
| | 4 (absolute jump) |
| Description: | Jump on positive. Program counter will be set to target address if the foregoing result is positive. The target address is given by using a jump label. See examples section for how to introduce a jump label. |
| Category: | Goto on flag |

| | |
|-----------------|------------------------|
| incr | Increment |
| Syntax: | incr p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | $p1 = p1 + 1$ |
| Flags affected: | C O S Z |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Increment register |
| Category: | Simple arithmetic |

| | |
|------------------|------------------------------------|
| incramadr | Increment RAM address |
| Syntax: | incramadr |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Increment RAM address pointer by 1 |
| Category: | RAM access |

| | |
|-----------------|---|
| initTDC | Initialize TDC |
| Syntax: | initTDC |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 2 |
| Cycles: | 3 |
| Description: | Initialization reset of the TDC (time-to-digital converter). Should be sent after configuration of registers. The initTDC preserves all configurations. |
| Category: | Miscellaneous |

| invert | Bitwise inversion |
|-----------------|-------------------------------|
| Syntax: | invert p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | p1 = not p1 |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Bitwise inversion of register |
| Category: | Logic |

| jsub | Unconditional jump |
|-----------------|---|
| Syntax: | jsub p1 |
| Parameters: | p1 = JUMPLABEL |
| Calculus: | PC = p1 |
| Flags affected: | C O S Z |
| Bytes: | 3 |
| Cycles: | 4 |
| Description: | Jump to subroutine without condition. The program counter is loaded by the address given through the jump label. The subroutine is processed until the keyword 'jsubret' occurs. Then a jump back is performed and the next command after the jsub-call is executed. This opcode needs temporarily a place in the program counter stack (explanation see below). Jump range: 0 < address < 8 kB |
| Category: | Unconditional Jump |

| jsubret | Return from subroutine |
|-----------------|--|
| Syntax: | jsubret |
| Parameters: | - |
| Calculus: | PC = PC from jsub-call |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 3 |
| Description: | Return from subroutine. A subroutine can be called via 'jsub' and exited by using jsubret. The program is continued at the next command following the jsub-call. You have to close a subroutine with jsubret - otherwise there will be no jump back. |
| Category: | Unconditional Jump |

| move | Move |
|-----------------|---|
| Syntax: | move p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-bit number |
| Calculus: | $p1 = p2$ |
| Flags affected: | S Z |
| Bytes: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Description: | Move content of p2 to p1 (p1 = ACCU, p2 = ACCU) Move constant to p1 (p1 = ACCU, p2 = NUMBER) |
| Category: | RAM access |

| mult24 | Signed 24-Bit multiplication |
|-----------------|--|
| Syntax: | mult24 p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] |
| Calculus: | $p1 = (p1 * p2) \gg 24$ |
| Flags affected: | S & Z of p1 |
| Bytes: | 2 |
| Cycles: | 30 |
| Description: | Signed multiplication of 2 registers like mult48, but only the 24 higher bits of the multiplication of 2 registers, result is stored in p1 |
| Category: | Complex arithmetic |

| mult48 | Signed 48-Bit multiplication |
|-----------------|--|
| Syntax: | mult48 p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] |
| Calculus: | $p1, p2 = p1 * p2$ |
| Flags affected: | S & Z of p1 |
| Bytes: | 2 |
| Cycles: | 30 |
| Description: | Signed multiplication of 2 registers. Higher 24 bits of the multiplication is placed to p1 Lower 24 bits of the multiplication is placed to p2 |
| Category: | Complex arithmetic |

| nand | Logic NAND |
|-----------------|---|
| Syntax: | nand p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p1 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | p1 = p1 nand p2 returns only 0 in case of bit combination 1 / 1 |
| Flags affected: | S Z |
| Bytes: | 2 (p1 = ACCU, p2 = ACCU) 5 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 3 (p1 = ACCU, p2 = ACCU) 6 (p1 = ACCU, p2 = NUMBER) |
| Description: | Logic NAND (negated AND) of the 2 given registers Logic NAND (negated AND) of register with constant |
| Category: | Logic |

| newcyc | Start TDC |
|-----------------|---|
| Syntax: | newcyc |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 2 |
| Cycles: | 3 |
| Description: | Start of TDC. This opcode can be used after configuration and initialization of the PSØ9 to start a new measurement cycle. Normally this is done by the PSØ81 ROM routines itself, but in case of custom-designed reset procedures this opcode can play a role. |
| Category: | Miscellaneous |

| nop | No operation |
|-----------------|---|
| Syntax: | - |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Placeholder code or timing adjust (no function) |
| Category: | Miscellaneous |

| nor | Logic NOR |
|-----------------|---|
| Syntax: | nor p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | p1 = p1 nor p2 returns only 1 in case of bit combination 0 / 0 |
| Flags affected: | S Z |
| Bytes: | 2 (p1 = ACCU, p2 = ACCU) 5 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 3 (p1 = ACCU, p2 = ACCU) 6 (p1 = ACCU, p2 = NUMBER) |
| Description: | Logic NOR (negated OR) of the 2 given registers Logic NOR (negated OR) of register with constant |
| Category: | Logic |

| or | Logic OR |
|-----------------|---|
| Syntax: | or p1, p2 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = ACCU [x, y, z, r] or 24-Bit number |
| Calculus: | p1 = p1 or p2 returns only 0 in case of bit combination 0 / 0 |
| Flags affected: | S Z |
| Bytes: | 2 (p1 = ACCU, p2 = ACCU) 5 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 3 (p1 = ACCU, p2 = ACCU) 6 (p1 = ACCU, p2 = NUMBER) |
| Description: | Logic OR of the 2 given registers Logic OR of register with constant |
| Category: | Logic |

| putep | Put lower 8 bits of register to internal EEPROM |
|-----------------|---|
| Syntax: | putep p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | EEPROM register (addressed by RAM address pointer) = p1 [7:0] |
| Flags affected: | - |
| Bytes: | 4 |
| Cycles: | ~12.5 ms |
| Description: | Put register into EEPROM. The lower 8 bits of the addressed register p1 is moved to the EEPROM (the EEPROM register address is set by the RAM address pointer). EEPROM bytes 0 to 127 are accessible via 'putep', bysetting the RAM address pointer to addresses 0 to 127 respectively. This opcode needs temporarily a place in the program counter stack (explanation see |

| | |
|-----------|--|
| | below). It is recommended not to use putepr in combination with the skip opcodes due to relatively longer execution times (~30ms). |
| Category: | EEPROM access |

| | |
|-----------------|---|
| ramadr | Set RAM address pointer |
| Syntax: | ramadr p1 |
| Parameters: | p1 = 8-Bit number |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Set pointer to RAM address (range: 0...255) |
| Category: | RAM access |

| | |
|-----------------|--|
| rotL | Rotate left |
| Syntax: | rotL p1[, p2] |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = 4-Bit number or none |
| Calculus: | p1 = p1 << 1 + carry; carry = MSB(x) (in case rotL p1, without p2) p1 = repeat (p2) rotL p1 (in case rotL p1, p2) |
| Flags affected: | C O S Z (of the last step) |
| Bytes: | 1 (p1 = ACCU, p2 = none) 2 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = none) 1+p2 (p1 = ACCU, p2 = NUMBER) |
| Description: | Rotate p1 left → shift p1 register to the left, fill LSB with carry, MSB is placed in carry register Rotate p1 left p2 times with carry → shift p1 register p2 times to the left, in each step fill LSB with the carry and place the MSB in the carry |
| Category: | Shift and rotate |

| | |
|-----------------|---|
| rotR | Rotate right |
| Syntax: | rotR p1[, p2] |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = 4-Bit number or none |
| Calculus: | p1 = p1 >> 1 + carry; carry: =MSB(x) (in case rotR p1, without p2) p1 = repeat (p2) rotR p1 (in case rotR p1, p2) |
| Flags affected: | C O S Z (of the last step) |
| Bytes: | 1 (p1 = ACCU, p2 = none) 2 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = none) |

| | |
|--------------|--|
| | 1 + p2 (p1 = ACCU, p2 = NUMBER) |
| Description: | Rotate p1 right → shift p1 register to the right, fill MSB with carry, LSB is placed in carry register Rotate p1 right p2 times with carry → shift p1 register p2 times to the right, in each step fill MSB with the carry and place the LSB in the carry |
| Category: | Shift and rotate |

| | |
|-----------------|--|
| round | Rounding |
| Syntax: | round p1, p2 |
| Parameters: | p1 = ACCU [x] p2 = NUMBER [half scale division] |
| Calculus: | p1 = round (p1, p2) |
| Flags affected: | |
| Bytes: | 7 |
| Cycles: | subroutine call |
| Description: | Rounds the number in x. Depending on the configured 'half scale division' the number stored in x will be rounded down or up (down < 5, up >= 5). |
| Category: | Miscellaneous |

| | |
|-----------------|--|
| setC | Set carry flag |
| Syntax: | setC |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | C 0 |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | Set carry flag and clear overflow flag |
| Category: | Shift and Rotate |

| | |
|-----------------|--|
| shiftL | Shift Left |
| Syntax: | shiftL p1[, p2] |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = 4-Bit number or none |
| Calculus: | p1 = p1 << 1; carry = MSB(x) (in case rotL p1, without p2) p1 = repeat (p2) shiftL p1 (in case rotL p1, p2) |
| Flags affected: | C 0 S Z |
| Bytes: | 1 (p1 = ACCU, p2 = none) 2 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = none) 1 + p2 (p1 = ACCU, p2 = NUMBER) |
| Description: | Shift p1 left → shift p1 register to the left, fill LSB with 0, MSB is placed in |

| | |
|-----------|--|
| | carry register Shift p1 left p2 times → shift p1 register p2 times to the left, in each step fill LSB with the 0 and place the MSB in the carry |
| Category: | Shift and rotate |

| shiftR | Shift right |
|-----------------|---|
| Syntax: | shiftR p1[, p2] |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = 4-Bit number or none |
| Calculus: | p1 = p1 >> 1; carry = MSB(x) (in case rotL p1, without p2) p1 = repeat (p2) shiftL p1 (in case rotL p1, p2) |
| Flags affected: | C O S Z |
| Bytes: | 1 (p1 = ACCU, p2 = none) 2 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = none) 1 + p2 (p1 = ACCU, p2 = NUMBER) |
| Description: | Signed shift right of p1 → shift p1 right, MSB is duplicated according to whether the number is positive or negative Signed shift p1 right p2 times → shift p1 register p2 times to the right, MSB is duplicated according to whether the number is positive or negative |
| Category: | Shift and rotate |

| sign | Sign |
|-----------------|---|
| Syntax: | sign p1 |
| Parameters: | p1 = ACCU [x, y, z, r] |
| Calculus: | p1 = p1 / p1 p1 = 1 = 0x000001 if p1 >= 0 p1 = -1 = 0xFFFFF if p1 < 0 |
| Flags affected: | S Z |
| Bytes: | 2 |
| Cycles: | 2 |
| Description: | Sign of addressed register in complement of two notations. A positive value returns 1, a negative value returns -1 Zero is assumed to be positive |
| Category: | Simple arithmetic |

| skip | Skip |
|-----------------|----------------------------------|
| Syntax: | skip p1 |
| Parameters: | p1 = NUMBER [1, 2, 3] |
| Calculus: | PC = PC + bytes of next p1 lines |
| Flags affected: | |
| Bytes: | 1 |

| | |
|--------------|----------------------------|
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 without conditions |
| Category: | Unconditional jump |

| | |
|-----------------|--|
| skipBitC | Conditional skip |
| Syntax: | skipBitC p1, p2,p3 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = NUMBER[0..23] p3 = NUMBER[1, 2, 3] |
| Calculus: | if (bit p2 of register p1 == 0) PC = PC + bytes of next p3 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p3 commands if bit p2 of register p1 is clear |
| Category: | Bitwise |

| | |
|-----------------|--|
| skipBitS | Conditional skip |
| Syntax: | skipBitS p1, p2,p3 |
| Parameters: | p1 = ACCU [x, y, z, r] p2 = NUMBER[0..23] p3 = NUMBER[1, 2, 3] |
| Calculus: | if (bit p2 of register p1 == 1) PC = PC + bytes of next p3 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p3 commands if bit p2 of register p1 is set |
| Category: | Bitwise |

| | |
|-----------------|---|
| skipCarC | Skip carry clear |
| Syntax: | skipCarC p1 |
| Parameters: | p1 = NUMBER [1, 2, 3] |
| Calculus: | if (carry == 0) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if carry clear |
| Category: | Skip on flag |
| skipCarS | Skip carry set |

| | |
|-----------------|---|
| Syntax: | skipCarS p1 |
| Parameters: | p1 = NUMBER [1, 2, 3] |
| Calculus: | if (carry == 1) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if carry set |
| Category: | Skip on flag |

| | |
|-----------------|---|
| skipEQ | Skip on zero |
| Syntax: | skipEQ p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (notequalzero == 0) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if result of previous operation is equal to zero |
| Category: | Skip on flag |

| | |
|-----------------|---|
| skipNE | Skip on non-zero |
| Syntax: | skipNE p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (notequalzero == 1) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if result of previous operation is not equal to zero |
| Category: | Skip on flag |

| | |
|-----------------|--|
| skipNeg | Skip on negative |
| Syntax: | skipNeg p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (signum == 1) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |

| | |
|--------------|---|
| Description: | Skip p1 commands if result of previous operation was smaller than 0 |
| Category: | Skip on flag |

| | |
|-----------------|--|
| skipOvrC | Skip on overflow |
| Syntax: | skipOvrC p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (overflow == 0) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if overflow is clear |
| Category: | Skip on flag |

| | |
|-----------------|--|
| skipOvrS | Skip on overflow |
| Syntax: | skipOvrS p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (overflow == 1) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if overflow is set |
| Category: | Skip on flag |

| | |
|-----------------|--|
| skipPos | Skip on positive |
| Syntax: | skipPos p1 |
| Parameters: | p1 = NUMBER[1, 2, 3] |
| Calculus: | if (signum == 0) PC = PC + bytes of next p1 lines |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 + skipped commands |
| Description: | Skip p1 commands if result of previous operation was greater or equal to 0 |
| Category: | Skip on flag |

| | |
|-------------|-------------|
| stop | Stop |
|-------------|-------------|

| | |
|-----------------|--|
| Syntax: | stop |
| Parameters: | - |
| Calculus: | - |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 1 |
| Description: | The DSP and clock generator are stopped, the converter and the EEPROM go to standby. A restart of the converter can be achieved by an external event like ,watchdog timer', ,external switch' or ,new strain measurement results'. Usually this opcode is the last command in the assembler listing. |
| Category: | Miscellaneous |

| | |
|-----------------|---|
| sub | Substraction |
| Syntax: | sub p1, p2 |
| Parameters: | p1 = NUMBER[1, 2, 3] p2 = NUMBER[1, 2, 3] or 24-Bit number |
| Calculus: | p1 = p2 - p1 |
| Flags affected: | C O S Z |
| Bytes: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Cycles: | 1 (p1 = ACCU, p2 = ACCU) 4 (p1 = ACCU, p2 = NUMBER) |
| Description: | Subtraction of 2 registers Subtraction of register from constant |
| Category: | Simple arithmetic |

| | |
|-----------------|---|
| swap | Swap |
| Syntax: | swap p1, p2 |
| Parameters: | p1 = ACCU [x, y, r] p2 = ACCU [x, y, r] |
| Calculus: | p1 = p2 and p2 = p1 |
| Flags affected: | - |
| Bytes: | 1 |
| Cycles: | 3 |
| Description: | Swap of 2 registers The value of two registers is exchanged between each other. Not possible with ACCU[z] |
| Category: | RAM Access |

5 Assembly Programs

The PS09 assembler is a multi-pass assembler that translates assembly language files into HEX files as they will be downloaded into the device. For convenience, the assembler can include header files to be then downloaded. The user can write his own header files but also integrate the library header files as they are provided by acam. The assembly program is made of many statements which contain instructions and directives. The instructions have been explained in the former section 4 of this datasheet. In the following we describe the directives and some sample code.

Each line of the assembly program can contain only one directive or instruction statement. Statements must be contained in exactly one line.

Symbols

A symbol is a name that represents a value. Symbols are composed of up to 31 characters from the following list:

A - Z, a - z, 0 - 9, _

Symbols are not allowed to start with numbers. The assembler is case sensitive, so care has to be taken for this.

Numbers

Numbers can be specified in hexadecimal or decimal. Decimal have no additional specifier. Hexadecimals are specified by leading "0x".

Expressions and Operators

An expression is a combination of symbols, numbers and operators. Expressions are evaluated at assembly time and can be used to calculate values that otherwise would be difficult to be determined.

The following operators are available with the given precedence:

| Level | Operator | Description |
|-------|----------|--------------------------------------|
| 1 | () | Brackets, specify order of execution |
| 2 | * / | Multiplication, Division |
| 3 | + — | Addition, Subtraction |

5.1 Directives

The assembler directives define the way the assembly language instructions are processed. They also provide the possibility to define constants, to reserve memory space and to control the placement of the code. Directives do not produce executable code.

The following table provides an overview of the assembler directives.

| Directive | Description | Example |
|---|---|---|
| CONST | Constant definition, <code>CONST [name] [value]</code> value might be a number, a constant, a sum of both | CONST Slope 42 CONST Slope constant + 1 |
| LABEL : | Label for target address of jump instructions. Labels end with a colon. All rules that apply to symbol names also apply to labels. | jsub LABEL1 LABEL1: ... jsubret |
| ; | Comment, lines of text that might be implemented to explain the code. It begins with a semicolon character. The semicolon and all subsequent characters in this line will be ignored by the assembler. A comment can appear on a line itself or follow an instruction. | <i>; this is a comment</i> |
| <comment> <endcomment> | Comment, lines of text that might be implemented to explain the code. It begins with <code><comment></code> directive and ends with <code><endcomment></code> directive. All subsequent characters between these directives will be ignored by the assembler. | <i><comment></i> <i>this is ...</i> <i>a very long ...</i> <i>comment</i> <i><endcomment></i> |
| #include | Include the header or library file named in the quotation marks <code>"</code> . The code will be added at the line of the include command. In the quotation marks there might be just the file name in case it is in the same folder as the program, but also the complete path. | #include "rdc.h" |

5.2 Sample Code

The following code shows the basic structure of any PS09 program:

```

;-----
;      File:program_template.asm
;      This is a template for a standard user program that shows the various possible flags
;      that can be
;      read to find out what caused the DSP to jump into the user code. Some part of user code
;      needs
;      to be executed on POR, some on External interrupt etc. Those jumps that are relevant to
;      the user
;      can be retained, the rest can be commented.
;      Author: VK
;-----

#include "config.h"
#include "PS09_RAM_constants.h"

ramadr      224+22
skipBitC    r, 19, 1      ; Checking for power on reset flag, Bit 19 - in Status register
jsub       Routine_POR
goto       end

ramadr      224+22
skipBitC    r, 18, 1      ; Checking for SSN_RST (S6) Button Pressed : Bit 18 - in Status
                      register
jsub       Routine_Button_Press
goto       end

ramadr      224+22
skipBitC    r, 17, 1      ; Checking for Watchdog reset : Bit 17 - in Status register
jsub       Routine_watchdog
goto       end

ramadr      224+22
skipBitC    r, 16, 1      ; Checking for End of measurement : Bit 16 - in Status register
jsub       Routine_measurement_end
goto       end

ramadr      224+22
skipBitC    r, 15, 1      ; Checking for wakeup in Sleep mode : Bit 15 - in Status register
jsub       Routine_sleep_mode
goto       end

ramadr      224+22
skipBitC    r, 08, 1      ; Checking for DSP start due to External Pin Interrupt : Bit 08 -
                      in Status register
jsub       Routine_ext_interrupt
goto       end

ramadr      80
skipBitC    r, 08, 1      ; Check for jump into user code because of Receive Int from UART:
                      Bit 08 - Reg.80
jsub       Routine_uart_rec_int
goto       end

Routine_POR:
;-----Insert Power on reset routine here-----
nop
jsubret

```

```

;-----
Routine_Button_Press:
;----- Insert routine to be executed on Pushed button here-----
nop
jsubret
;-----

Routine_watchdog:
;-----Insert reset routine for watchdog reset here-----
nop
jsubret
;-----

Routine_measurement_end:
;-----Insert routine to be executed on measurement end here -----
nop
jsubret
;-----

Routine_sleep_mode:
;-----Insert routine for wakeup in Sleep mode here-----
nop
jsubret
;-----

Routine_ext_interrupt:
;-----Insert Interrupt service routine for External Interrupt from Pin here---
nop
jsubret
;-----

Routine_uart_rec_int:
;-----Insert interrupt routine for UART receive Interrupt here-----
nop
jsubret
;-----
end:
clrwdt
stop
;-----end of program-----

```

The following example from the Assembler program shows a simple program to display results on an LCD:

```

;-----
;      File: simple_meas_with_LCD.asm
;
;      Author: VK / UTG
;-----

;-----
;      Simple program to demonstrate calculation of Initial Offset after POR and after the Initial
;      Offset is taken, it goes to measure mode.
;      State 1: Take init offset value (ignore first 4 measurements) Then average over next 5.
;      State 2: Take measurement value and subtract init offset value, then scale to display
;      correct weight.
;

```

```
#include "config.h"
; The other include files are included at the end of the program

;----- Constants for measurement program -----
CONST      init_offset_for_measurement      121
CONST      count_measurements              122
CONST      temp_count                      123
CONST      init_offset_status              124
; To store the status of init_offset, 0 - if offset calculation is not yet complete
; 1 - if offset calculation is complete
;-----
start:
; On POR configure the PS09 to act as SPI master to communicate with the Holtek driver
ramadr      224+22
skipBitC    r, 19, 3      ; Checking for power on reset : flg_rstpwr bit
jsub        cfg_spi_master; Configures the SPI master lines on GPIO0, GPIO1 and GPIO2
; To use other pins for the SPI master, change in this include file accordingly
jsub        cfg_ht_driver ; Configure the HT1621 driver for display
jsub        init_values

ramadr      init_offset_status              ; Check status of init offset
skipBitS    r, 0, 2
jsub        get_init_offset
goto        end
; Refresh the displayed value on measurement completion
ramadr      22+224
skipBitS    r, 16, 1      ; Check for end of measurement - Bit 16
goto        end
;----- To display measurement values on LCD-----
; Reading measurement value HB0 into x Akku
ramadr      224+20
move        x, r
ramadr      init_offset_for_measurement
move        y, r
sub         x, y
abs         x
;----- Multiplication factor -----
shiftrl     x, 4          ; HB0 * 2^4
move        z, 0x8D5E5    ; With 2000 g Load and no multiplication factor (& with division
                           by 10 seen below): Meas.value = 3629
                           ; (2000 / 3629) * 2^20 = 0x8D15F
                           ; This factor is further corrected with again 500 g Load
                           ; (500/499) * 0x8D15F = 0x8D5E5
                           ; The above 2^4 and 2^20 factors are multiplied to adjust for the
                           following mult24 opcode
mult24      x, z          ; Implicit to opcode , result is / 2^24
move        z, 10         ; division by 10 to omit the LSB, only 4 digits needed
divmod      x, z
move        y, 0          ; Number of digits after the decimal point
jsub        notolcd       ; Routine to convert the display value to LCD format
move        z, 0x10       ; 0x10 - Code to display units as gm, Codes for other units are
                           present in notolcd.h
jsub        display_value_on_Holtek        ; Displaying the data with the Holtek driver
end:
clrwdt
stop
;-----end of main program-----

;----- Subroutines-----
init_values:
ramadr      count_measurements
```

```

clear            r

ramadr           init_offset_status
clear           r
ramadr           temp_count           ; Use a temporary counter
clear           r
jsubret
;=====
get_init_offset:
move            x, 0xFFFF00           ; setting all the segments
move            y, 0xFFFFF
move            z, 0
jsub            display_text_on_Holtek
;----- Get current measurement value and save it to x for further processing -----
ramadr           224+20           ;HB0 value
move            x, r
;----- Count Loops and dismiss first 4 measurements -----
ramadr           temp_count           ; Use a temporary counter
incr            r
compare         r,4           ; Is it higher than 4? -> Ignore first 4 measurements
gotoNeg         apply_roll_avg; From the 5th measurement, perform a rolling average
jsub            roll_avg_init5; Initialize rolling average filter with measurement value in X
goto            end_init_offset
;=====
apply_roll_avg:
;----- Use rolling average filter for init offset value -----
jsub            rolling_avg_5 ; 5 times rolling average
;----- Count Loops for Initial Offset -----
ramadr           count_measurements
incr            r
compare         r,5           ; Take 5 measurements
gotoPos         end_init_offset; Is it Lesser than 5? , then take more measurements else store
                        the offset
;---- After 5 valid measurements save filtered value to RAM as initial offset for measurement-
ramadr           init_offset_for_measurement ; Init offset value for measurement mode
move            r, x
ramadr           init_offset_status
incr            r           ; Set status of init_offset_status to 1 (init offset taken)
end_init_offset:
jsubret
;=====
#include "rollavg.h" ; This file is used to calculate the rolling average of the measurement
#include "cfg_spi_master.h"; These include files are used ONLY for using the Holtek LCD driver
#include "cfg_ht_driver.h"
#include "notolcd.h"
#include "display_value_on_Holtek.h"
#include "display_text_on_Holtek.h"

```

For details on programming with the assembler tool please refer to the PS09-EVA-KIT datasheet, which includes a description of the assembler software.

6 Miscellaneous

6.1 Bug Report

[See Data Sheet, Volume 1 “General Data and Front-end Description”]

6.2 Document History

05.11.2014 First release of Volume 2, Version 1.0



acam-messelectronic gmbh
Friedrich-List-Straße 4
76297 Stutensee-Blankenloch
Germany
Phone +49 7244 7419 - 0
Fax +49 7244 7419 - 29
E-Mail support@acam.de
www.acam.de