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**

acam mess•electronic

PICOCAP®

Data Sheet

PCapØ2Ax DSP

Single Chip Solution for Capacitance Measurement
Volume 2: Digital Signal Processor

August 16th, 2013, Version 0.2
Document-No: DB_PCapØ2A_Vol2_en.pdf

Published by acam-messelectronic gmbh

Disclaimer / Notes

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 System Overview

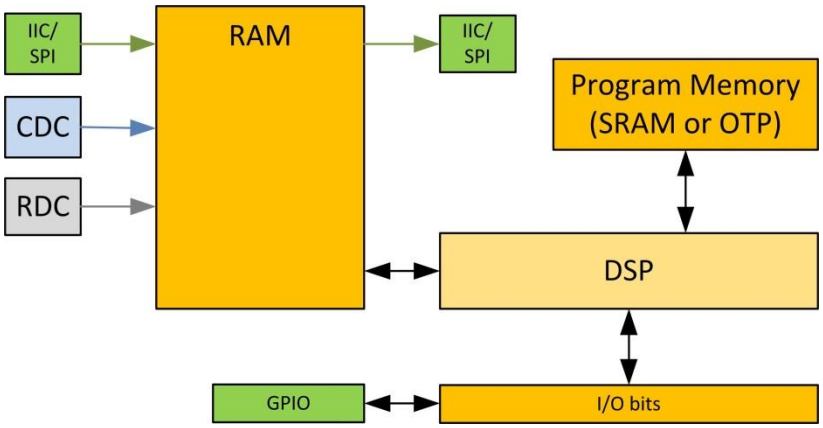
This volume 2 datasheet describes the 48-DSP of the PCapØ2A. It describes only the items related to the DSP itself. For all other issues please refer to the volume 1.

A 48-Bit digital signal processor (DSP) in Harvard architecture has been integrated to the PCapØ2. It is responsible for taking the information from the CDC and RDC measuring units, for processing the data and making them available to the user interface. Both, the CDC/RDC raw data as well as the data processed by the DSP are stored in the RAM. The program for the DSP is stored either in the OTP or the SRAM. The DSP can collect various status information from a set of 64 I/O Bits and write back 16 of those. This way the DSP can react on and also control the GPIO pins of PCapØ2. The DSP is internally clocked at approximately 100 MHz. The internal clock is stopped through a firmware command, to save power. The DSP can also be clocked by other sources (e.g. a low power clock). The DSP starts again upon a GPIO signal or an "end of measurement" condition.

In its simplest form, the DSP transfers the pure time measurement information from the CDC/RDC to the read registers without any further processing (PCapØ2_Raw_values.hex). The next higher step is to calculate the capacitance ratios including the information from the compensation measurements, as it is provided in acam's standard firmware version PCapØ2_standard.hex.

The DSP is acam proprietary to cover low-power tasks as well as very high data rates. It is programmed in Assembler. A user-friendly assembler software with a graphical interface, helptext pop-ups as well as sample code sustain programming efforts.

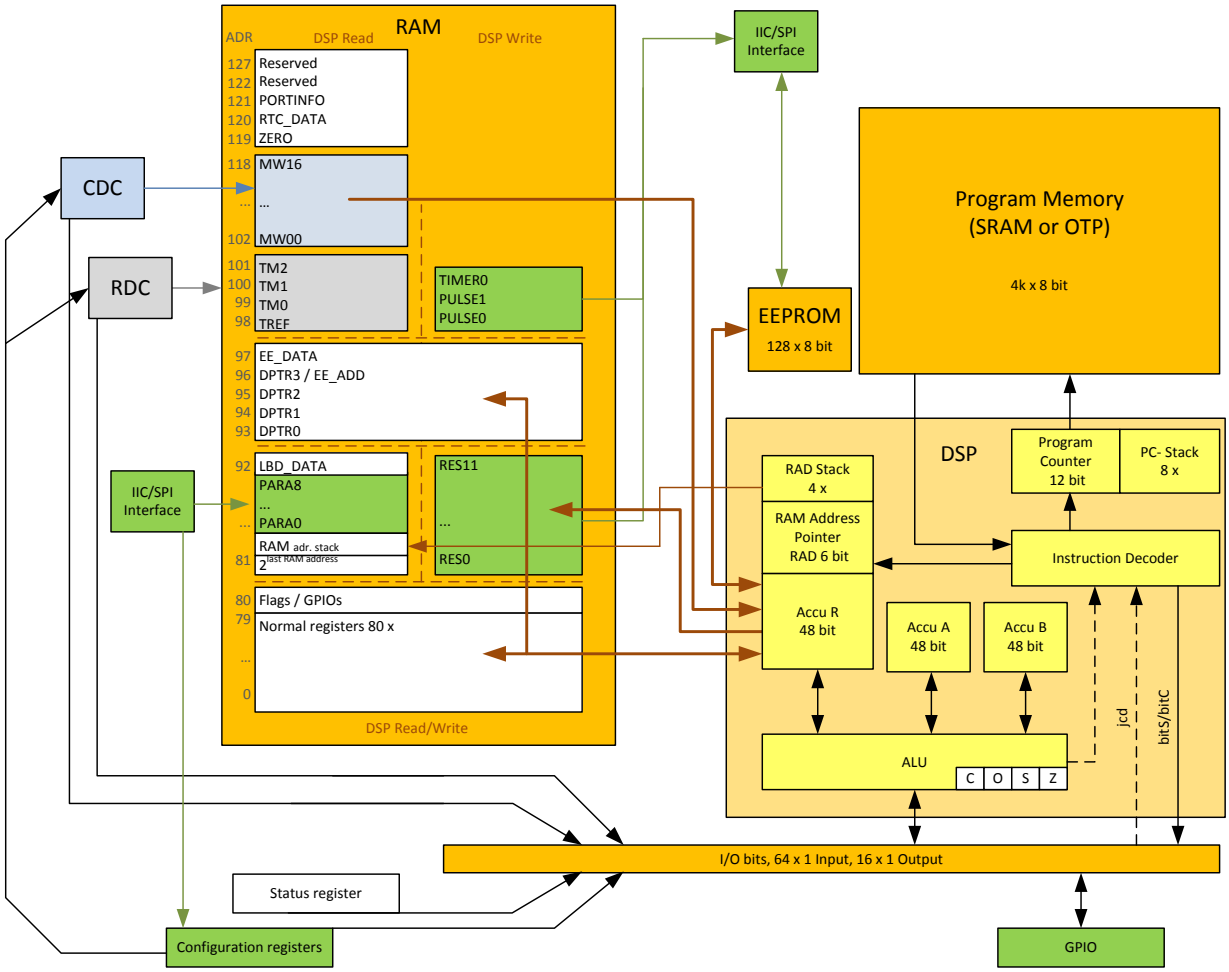
Figure 1-1 DSP Embedding



2 DSP & Environment

The detailed structure of how the DSP is implemented into the PCapØ2 is shown in figure 2-1.

Figure 2-1 DSP Environment



This Harvard DSP for 48 bit wide parallel data processing is coupled to a 128 x 48 bit RAM, 80 x 48 bit thereof free accessible. In read access, the DSP can get the CDC measurement raw data from address space 102 to 118, the RDC raw data from address space 98 to 101. By write access the DSP provides the output data to either the serial interfaces (addresses 81 to 92) or to the PDM/PWM interfaces (addresses 98, 99).

A detailed description of the RAM is given is section 2.1. The DSP operates with two accumulators A and B and has direct access to the RAM, which can be seen as a third accumulator. The RAM address pointer is of 6 bit size, and there is a 4-fold stack for RAM addresses.

The program code for the DSP is in the OTP or the SRAM. During evaluation, the program is typically in the SRAM. In production it will be in the OTP. For fast applications it is also possible that after power-on reset the program is copied from the OTP into the SRAM automatically. The program counter has 12 bit and there is an 8-fold stack for the program counter.

Finally, the DSP can get a lot of information from the 64 I/O bits. The read information covers the ALU status, trigger information, some of the configuration bits and the information about the status of the GPIOs. 16 of those bits can be used as outputs, setting the GPIOs and also some internal information. For details see section 2.5. The DSP can read these bits by means of instruction jcd (conditional jump) and set those bits by means of instructions bitS/bitC (bit Set/Clear).

The ALU flags overflow, carry, equal/not equal and pos./neg. are used directly as condition for the jcd instructions and are also mirrored in the I/O bits.

2.1 RAM Structure

The RAM plays a key role. It is made of 128 words with size of maximum 48 bit. The DSP has free write and read access to registers address 80 of those words, all 48 bits wide. The RAM space addresses 81 to 92 and 98 and higher is different for read and write.

The main data in the read section are the raw data as they come from the CDC and the RDC as well as the parameters. The parameters are part of the configuration registers and set via the serial interface or copied from the OTP.

The DSP reads those raw data, does the data processing and writes back the results into the write section of the RAM. From there, the user can read the final results through the serial interface.

Some of the RAM cells are dedicated to special functions and will be described in the following in detail.

Table 2-1 gives a full overview of the RAM write and read registers.

Table 2-1 RAM Structure in Detail

RAM: DSP Read				RAM: DSP Write		
Addr	**Description**	**Bits**		**Addr**	**Description**	**Bits**
127	Reserved	-				
…		-				
122	Reserved	-				
121	PORTINFO	24				
120	RTC_DATA	16				
119	ZERO	48				
118	MW16	37				
	…	37				
102	MW00	37				
101	TM2	37				
100	TM1	37		100	TIMER0	16
99	TM0	37		99	PULSE1	16
98	TREF	37		98	PULSE0	16
97	EE_DATA					8
96	DPTR3 / EE_ADD					7
95	DPTR2					7
94	DPTR1					7
93	DPTR0					7
92	LBD_DATA	6		92	RES11	24
91	PARA8	24		91	RES10	24
…	…	24		…	…	…
83	PARA0	24		83	RES2	24
82	RAM_adr_Stack	24		82	RES1	48/24
81	$2^{last\_RAM\_address}$	48		81	RES0	48/24
80	Flags / GPIO's					48
79	Free RAM					48
…	…					48
0	Free RAM					48

2.1.1 Registers 0 to 79

This is normal RAM space without any special functions. It is readable and writable via instruction rad.

Example:

Add content of RAM address 12 and 13 and write the result into RAM address 13

```
rad   12
move  a, r
rad   13
add   r, a
```

This RAM space can be used as a normal register.

2.1.2 Register 80, Flags & Internal Control Signals

Table 2-2 Flags

Bit	Flag Name	Default (after Reset)	Description
0	EE_ON_BY_DSP	0	Disjunction (OR) with EE_ON from CFG
1	CFG_BANK_SEL	0	Switches config Bank for alternated settings of R_TRIG_SEL, C_TRIG_SEL, CONV_TIME, C_AVRG
2	C_SELFTEST_BY_DSP	0	Antivalence (XOR) with C_SELFTEST from config
3	RDCHG_COM_INT_SEL	0	0 := use RDCHG_IN_SEL0 1 := use RDCHG_IN_SEL1 for internal compensation
4..7	free to use	0	
8	RST_RDC	pulsed	Temperature reset. This flag has to be set 1, after each RDC measurement. Otherwise a new RDC measurement is not possible. This flag is set back to 0 automatically
16..47	free to use	unknown	

2.1.3 Read Register 81

This register is there to get the N-th power of 2. The exponent N needs to be written to the RAD stack. The result can be read from register 81. In the assembler, the necessary three instructions are merged into one:

```
load2exp    a, 10 ; a = 2^10 = 1024
```

is the same as

```
rad   10
rad   81
move  a, r
```

A very simple and efficient method to set an accumulator = 1 is

```
load2exp    b, 0  ; b = 2^0 = 1
```

2.1.4 Read Register 82

This register contains the content of the RAM address stack. The 24 bit data is made of the 4 last 6-bit RAM addresses. This address can be used to load 24 bit constants from the program memory into the data space. The necessary 6 instructions are merged into one single instruction by the assembler.

```
load a, 1715956   ; a = 1715956
```

is the same as

```
rad 'h06          ; 'h06 * 2^18
rad 'h22          ;+ 'h22 * 2^12
rad 'h3b          ;+ 'h3b * 2^6
rad 'h34          ;+ 'h34 = 1715956
rad 82
move a, r
```

2.1.5 Read Register 83 to 91, Parameters

The content of the configuration registers addresses 50 to 76, the 9 parameters, is copied into this RAM space and made available to the DSP this way.

The parameters are used to provide variable and firmware specific configuration data. Typically, e.g. PARAMETER8 is used in the standard firmware and others for the gain correction factor.

2.1.6 Read Register 92, Low battery detection

This register shows the result of the Low-voltage detection measurement, LBD_DATA. The value has 6 bit. The result depends on the trim of the Bandgap (recommended = 7).

BG_TRIM1 = 7: Voltage = 2.026 V + LBD_DATA * 24.4 mV

2.1.7 Read/Write Registers 93 to 96, Data Pointer

These registers may be used for indirect addressing. They are 7 bits wide. DPTR3 is simultaneously used as address pointer for the EEPROM.

Load a register with the address you want to manipulate:

`load a, <myaddress>`
`rad DPTR0`
`move r, a`
	`load a, <myaddress>`
	`rad 93`
	`move r, a`

Load a RAM address pointer with content of DPTRO:

```
rad _at_DPTR0          ; now ram address pointer is set to content of DPTR0
```

!! Hint: in the PCapØ2x.h "_at_DPTRO" to "_at_DPTR3" are set to values of 284 to 287. These are no valid RAM addresses but just indicators to the assembler to generate the corresponding opcodes.

Example direct memory address: Copy a memory block from one address to another:

`__sub_dma__:`
`not b`
`inc b`
`__sub_dma_loop__:`
`rad _at_DPTR1`
	; DPTR1 := source_address; DPTRO := destination address; b:= length of dma

```
move a, r
rad _at_DPTR0
move r, a

rad DPTR0
inc r
rad DPTR1
inc r
inc b

jNE __sub_dma_loop__
jrt
```

2.1.8 Read/Write Register 97, EEPROM DATA

This register named EE_DATA is used to write or read data to or from EEPROM.

Read:

To read data from the EEPROM the read address has to be written to DPTR3/EE_ADD, register 96 and a read strobe (bitS EE_RD) must be generated. The DSP has to wait until the data on Register 97 are valid. Afterwards, the value can be fetched from register 97:

```
load a, <myaddress>
rad EE_ADD
move r, a
bitS EE_RD

while_ee_rd_loop:
jcd EE_BUSY while_ee_rd_loop

rad EE_DATA
move a, r
```

Write:

For writing into the EEPROM it has to be activated and the EE_WR_EN has to be set.

To write to the EEPROM the address has to be loaded to DPTR3/ EE_ADD (register 96) and the value has to be written to EE_DATA (register 97). No further action is necessary.

Before each write-process ensure that the EEPROM is ready.

Erase:

To erase the EEPROM it has to be activated and writing has to be enabled. To erase a databyte, the address has to be set to DPTR3/EE_ADD (Register 96) and an erase-pulse has to be generated (bitS EE_ER).

```
load2exp a, EE_ON
rad FLAGREG
or r, a

load a, <myaddress>
rad EE_ADD
move r, a

while_ee_busy:
jcd_EE_BUSY, while_ee_busy

bitS EE_ER

while_ee_erasing:
jcd EE_BUSY, while_ee_erasing

load b, <mycontent>

rad EE_DATA
move r, b

while ee_writing:
jcd EE_BUSY, while ee_writing

load2exp a, EE_ON
not a
rad FLAGREG
and r, a
```

2.1.9 Read Register 98 to 101, RDC Results

Those register hold the resistance discharge time measurement raw data of 37 bit. The will be used by library rdc.h to calculate the resistance ratios.

2.1.10 Read Registers 102 to 118, CDC Results

Those register hold the capacitance discharge time measurement raw data of 37 bit. The will be used by library cdc.h to calculate the resistance ratios, taking into account the compensation methods selected.

2.1.11 Read Register 119, ZERO

This register a default zero value for easy programming.

2.1.12 Read Register 120, RTC_DATA

There is a real time counter which can be used to have long-term timing information. The used demands an external 32.768 kHz oscillator. The RTC is a Gray-counter with 2^{17} pre-divider, which gives a base period of 4 seconds and a measurement range of 3 days and 49 minutes. The count is given in Gray-code. Library file gray2bin.h supports the conversion into binary data format.

2.1.13 Read Register 121, PORTINFO

The low 8 bits mirror the port enable setting as defined by configuration parameter C_PORT_EN in register 12.

Bits 8 to 17 are error flags for the capacitance ports including the internal reference ports.

2.1.14 Write Registers 81 to 92

These are the result registers to which the DSP has to write the output data so that the user can read those through the SPI/IIC interface as Res 0 to Res 7.

Addresses 81 and 82 are 48 bit, while the others are 24 bit wide only.

Attention: These Registers are write only! You can't read from these Registers!

2.1.15 Write Registers 98, 99

These registers contain the data that is used to generate the PWM/PDM output signals. After the DSP has calculated and scaled the output data, it writes those into these two registers. The data are 14 bit wide.

2.1.16 Write Register 100, TIMER0

The DSP has a 16bit Timer based on the OLF clock. This Timer may be used to generate long delays while the DSP is halted. Bit #1 (timer) in DSP_START_EN must be set!

By writing a value to Register 100 the timer starts to count up from 0 each OLF-clock cycle until the written value has been reached. Then a DSP_START_TRIG is generated.

If the DSP is not halted the TIMER0_IRQ_N Flag could be tested anyway.

Example 1 (without halting DSP):

```
CONST wait_time_1ms 50        ; 50*20µs (@50kHz)
…
load a, wait_time
rad TIMER0
move r, a
```

```
timer_wait_loop:
jcd TIMER0_IRQ_N, timer_wait_loop
```

Example 2 (with halting DSP, DSP run on internal oscillator)

```
CONST wait_time_1ms 50          ; 50*20µs (@50kHz)
…
ORG 0
jcd TIMER0_IRQ_N, Skip_Timer0_process
      jsb Triggered_by_Timer0
Skip_Timer0_process:
…
load a, wait_time
rad TIMER0
move r, a
stop

Triggered_by_Timer0:
…
```

2.2 SRAM / OTP

Table 2-3 Memory organization

		SRAM		OTP					
Address				direct/single		double		quad	
dec.	hex.	Contents	Length [Byte]	Contents	Length [Byte]	Contents	Length [Byte]	Contents	Length [Byte]
4095	FFF			Unused	1	Test byte	1	Test byte	1
4094 .. 4032	FFE .. FC0			Config. Registry	63	Config. Registry	63	Config. Registry	63
4031 .. 2048	FBF .. 800h							Program code	1984
2047	7FF							Test byte	1
2046 .. 1984	7FE .. 7C0							Config. Registry	63
1983 .. 0	7BF .. 0	Program code	4096	Program code	4032	Program code	4032	Program code	1984

2.2.1 Memory Management

The DSP can be operated from SRAM (for maximum speed, 100 MHz max.) or from OTP (for low power, 10 MHz max. with error correction, 40 MHz max. without error correction). When the firmware has been copied from the OTP into the SRAM and the DSP runs from the SRAM, it is possible to use an SRAM-to-OTP data integrity monitor. It can be activated setting parameter MEMCOMP in register 0. This has to be disabled for operation directly from the OTP and needs the DSP to run with the internal ring oscillator.

Memory integrity ("ECC") mechanisms survey the OTP contents internally and correct faulty bits (as far as possible).

MEMLOCK, the memory readout blocker, is activated by special OTP settings performed when loading down the firmware (see the graphi-cal user interface existing for firmware development). MEMLOCK contributes to the protection of your intellectual property. MEMLOCK gets active earliest after it was written to the OTP and the chip got a power-on reset. MEMLOCK is write-only, it can't be set back.

2.2.2 OTP

The PCapØ2 is equipped with a 4 kB permanent program memory space, which is one-time programmable, called the OTP memory. In fact, the OTP is total 8 kB in size but 4 kB are used for ECC mechanism (error correction code). The default state of all the bits of the OTP memory in an un-programmed state is 'high' or 1. Programming a bit means changing its state from High to Low. Once a bit is programmed to 0, it cannot be programmed back to 1. Data retention is given for 10 years at 95°C. MEMLOCK is fourfold protected.

2.3 DSP Inputs & Outputs

The DSP has access to 64 bits of information on ALU status, start trigger, configuration, input/output pins.

This information can be interpreted by means of instruction jcd, conditional jump.

Instruction conditional jump:

```
jcd   p1,p2: if p1 ==1 then jump to p2
```

16 of those bits can be set by the DSP, e.g. to set a GPIO or to select between RDC and CDC data. The bits are controlled by means of instructions bitS/bitC (bit Set/bit Clear).

Table 2-4 DSP Inputs/Outputs

Bit Name	Description	Type	Read Bit #	Write Bit #
DSP_OUT<7…0>	Status feedback of the 8 general DSP outputs (Write bits 0 to 7).	IN	56 to 63	
SPI_TRIGGERED_N	Flag = LOW indicates that a falling edge at a pin or an SPI/IIC opcode has started the DSP. This flag is reset by a STOP instruction at the end of the firmware.	Start trigger	55	
PIN_TRIGGERED_N	Flag = LOW indicates a GPIO has started the DSP		53	
TDC_OVFL_TRIGGERED_N*	Flag = LOW indicates that a TDC overflow has started the DSP. This flag is reset by a STOP instruction at the end of the firmware.	Start trigger	52	
INTN_TRIGGERED_N	Flag = LOW indicates the DSP is started by rising edge of INTN-Signal	Start trigger	51	
RDC_TRIGGERED_N *	Flag = LOW indicates that an RDC measurement has started the DSP. Therefore, DSP_STARTONTEMP has to be set (configuration register 8). This flag is reset by a STOP instruction at the end of the firmware.	Start trigger	50	
TIMER0_IRQ_N	Flag = LOW indicates the DSP is started by the internal timer	Start trigger	49	
CDC_TRIGGERED_N	Indicates the DSP is started by the end of the capacitance conversion.	Start trigger	48	
ALU_OFL_N	ALU flags for overflow, carry, equal and sign. The ALU flags are used by the jump instruction of the assembler	Status	47	
ALU_OFL		Status	46	
ALU_CAR_N		Status	45	
ALU_CAR		Status	44	
ALU_EQ		Status	43	
ALU_NE		Status	42	
ALU_POS		Status	41	
POR_FLAG_Wdog	status bit 7	Status	28	
POR_FLAG_CONFIG_N	status bit 6	Status	27	
POR_FLAG_SRAM_N	status bit 5	Status	26	
TIMER0_Busy	Indicates that timer0 is still running	Status	25	
DCHG_DUM_EN		Config Reg	19	
MR2_N	Indicates whether measure mode 2 is set or not. 0 = MR2, 1 = MR1	Config Reg	18	
C_COMP_FORCE_N		Config Reg	17	

C_COMP_R_N		Config Reg	16	
C_COMP_EXT_N		Config Reg	15	
C_COMP_IN_N		Config Reg	14	
C_SINGLE / C_DIFFERENTIAL_N		Config Reg	13	
C_GROUNDED / C_FLOATING_N		Config Reg	12	
TRIG_BG	This parameter starts the Bandgap (to synchronize with measurement) (pulse, automatically set to 0)	Out		15
TRIG_LBD	This parameter starts the "Low Bat Detection" (pulse, automatically set to 0)	Out		14
EE_ER	EEPROM erase strobe (pulse, automatically set to 0)	Out		13
EE_RD	EEPROM read strobe (pulse, automatically set to 0)	Out		12
ERR_OVFLN	Flag = bit 16 of status register. Indicates an overflow or other error in the CDC.	Status	11	
COMB_ERRN	Flag = bit 16 of status register. This is a combined condition of all known error conditions.	Status	10	
CYC_ACTIVEN	Flag = bit 23 of status register. Indicates that the CDC frontend is active.	Status	9	
LBD_BUSY	Indicates Low-Bat-Detection is busy	Status	8	
EE_BUSY	Indicates, EEPROM is busy	Status	7	
Interrupt_In	Port INTN will be reseted by a positive edge on SSN (SPI) or a stop condition (I2C), whit this the current status of INTN could be detected		6	
TEMPERRN	Flag = bit 3 of status register. Indicates whether an error occurred during the temperature measurement. 0 = error, 1 = no error	Status	5	
RDC_BUSY	Flag = bit 22?? of status register. Indicates RDC unit is busy. 0 = measurement done, 1 = measurement running.	Status	4	
Interrupt_Out	Sets the interrupt (pin INTN) (pulse, automatically set to 0)	Out		11
(PAGE)	Reserved, do not use	Out		10
TRIG_RDC	This bit starts a new RDC measurement. (pulsed, automatically set to 0)	Out		9
TRIG_CDC	This bit starts a new CDC measurement (pulsed, automatically set to 0)	Out		8
DSP_7	Those two outputs are used by the DSP for	Out		7

DSP_6	- Reset watchdog (DSP_7) - INI_RESET by DSP (DSP_6)	Out		6
DSP_5	Sets the general purpose output pin PG5	Out		5
DSP_4	Sets the general purpose output pin PG4	Out		4
DSP_3	When the Pulse1 is switched OFF then this bit can be used to set and clear the general purpose output pin PG3. When the Pulse1 is ON then this bit must be cleared so that the Pulse1 output appears on PG3.	In/Out	3	3
DSP_2	When the Pulse0 is switched OFF then this bit can be used to set and clear the general purpose output pin PG2. When the Pulse0 is ON then this bit must be cleared so that the Pulse0 output appears on PG2	In/Out	2	2
DSP_1	Set or read the general purpose I/Os at pins PG0 & PG1. The assignment is programmable and shown in detail below.	In/Out	1	1
DSP_0		In/Out	0	0

* A positive edge on those inputs start the DSP. The status of the start trigger is memorized till the next reset or stop of the DSP. The start trigger information can be read from inputs 32 to 36 by jcd.

2.4 ALU Flags

With each ALU operation flags are set. The ALU has four flags: overflow, carry, equal and sign. The following table shows an overview:

Table 2-5 ALU Flags

Flag	Description	Format	Modified by Instructions:	Interpreted by Instructions:	Range
ON	No Overflow	signed	add, sub, mult, div	jOvIC, jOvIS	>= -2^{47} and <= $2^{47} - 1$
O	Overflow				< -2^{47} and > $2^{47} - 1$
CN	No Carry*	unsigned	add, sub, mult, div	jCarC, jCarS	< 2^{48}
C	Carry*				>= 2^{48}
Z	Equal / Zero	signed / unsigned	add, sub, mult, div, move, shiftL, shiftR	jEQ, jNE	== 0
ZN	Not Equal / not Zero				!=0
S	Positive	signed	add, sub, mult, div, move, shiftL, shiftR	jPos, jNeg	>= 0
SN	Negative				< 0

* During addition, the carry C is set when a carry-over takes place from the most significant bit, else C remains at 0.

During subtraction, carry C is by default 1. Carry C is cleared only when the minuend < subtrahend.

E.g. for A - B: if A ⩾ B → C = 1; if A < B → C = 0.

In other words, the carry C is actually the status of the carry of the addition operation A+ 2's complement (B).

2.5 DSPOUT – GPIO Assignment

PCapØ2 is very flexible with assignment of the various GPIO pins to the DSP inputs/outputs. The following table shows the possible combinations.

Table 2-6 Pin Assignment

External Port	Description	In/Out
PG0	SSN (in SPI-Mode)	in
	DSP_x_0 or DSP_x_2	in* / out
	FF0 or FF2	in*
	Pulse0	out
PG1	MISO (in SPI-Mode)	out
	DSP_x_1 or DSP_x_3	in* / out
	FF1 or FF3	in*
	Pulse1	out
PG2	DSP_x_0 or DSP_x_2	in* / out
	FF0 or FF2	in*
	Pulse0	out
	INTN	out
PG3	DSP_x_1 or DSP_x_3	in* / out
	FF1 or FF3	in*
	Pulse1	out
PG4	DSP_OUT_4 (output only)	out
PG5	DSP_OUT_5 (output only)	out

* These ports provide an optional debouncing filter and an optional pull-up resistor.

Figure 2-2 GPIO Assignment

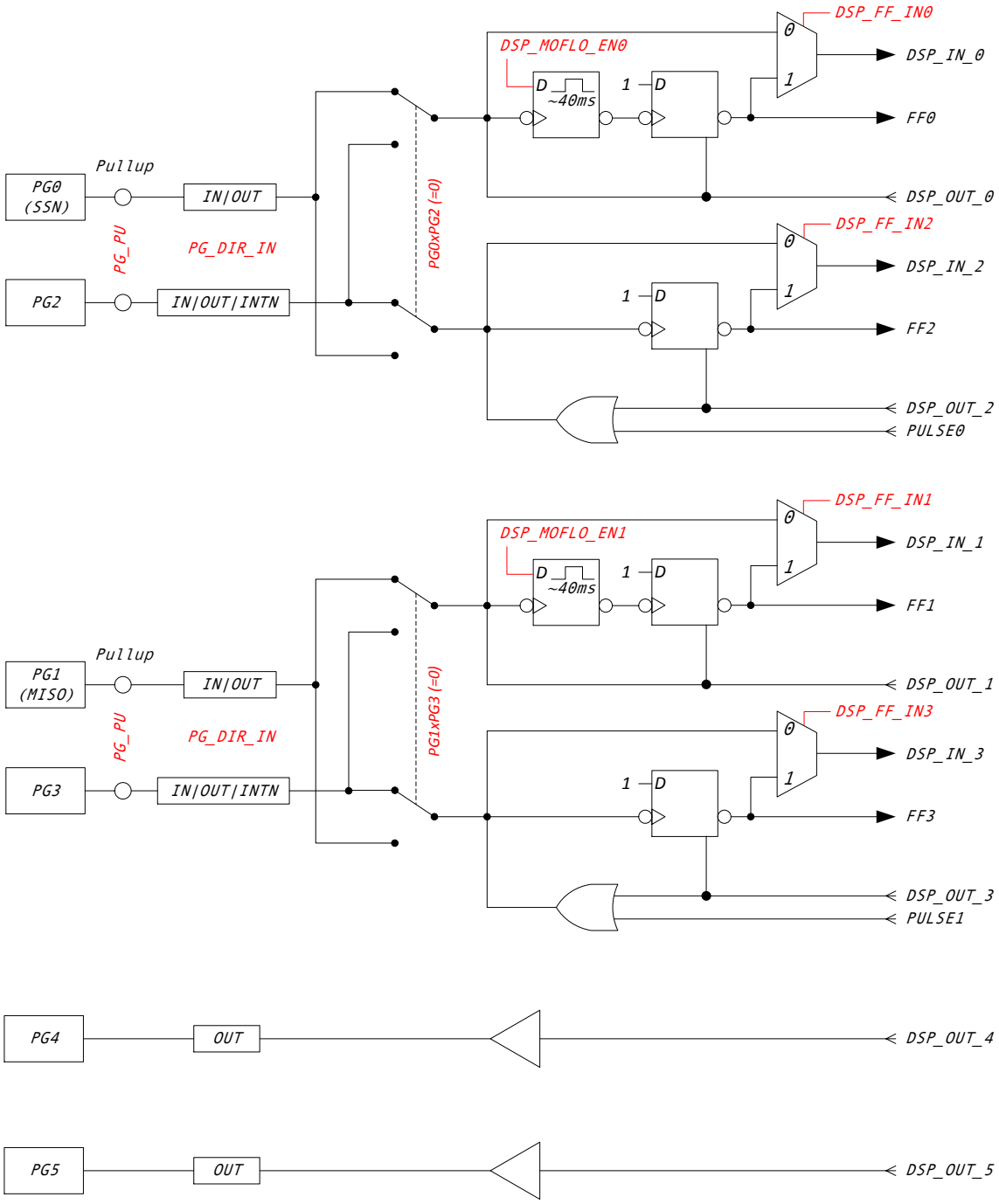
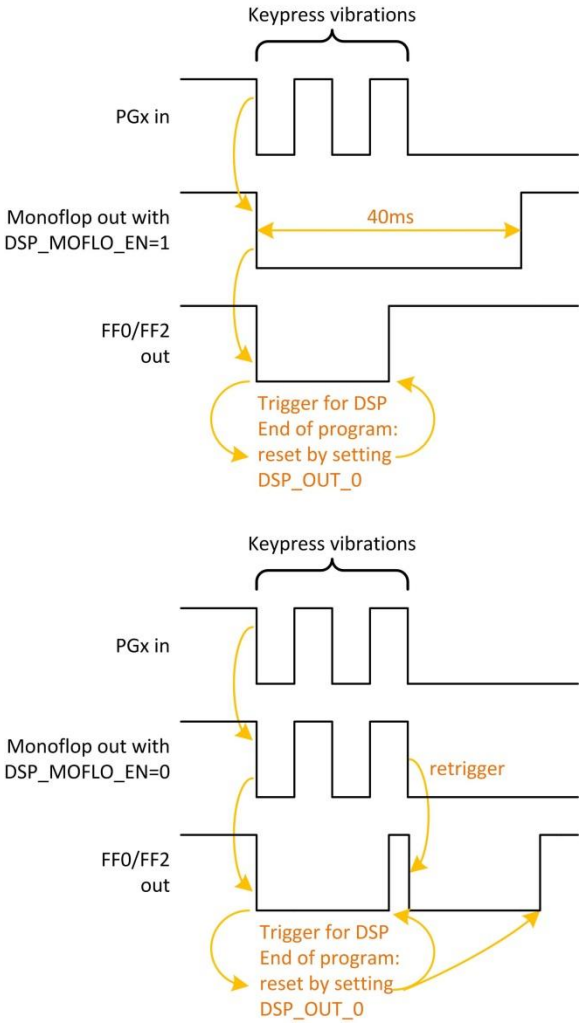


Figure 2-3 Port trigger timing



There is a possibility to activate a 40 ms debounce filter ("monoflop") for the ports in case these are used as inputs. This might be useful especially in case the DSP is started by the pins (signals FF0, FF2). Figure 2-3 shows the effect of the monoflop filter.

The settings herefore are made in configuration registers 8 and 9. The relevant parameters are:

Table 2-7

Parameter	Description	Settings
INT2PG2	Useful with QFN24 packages, where no INTN pin is available. Permits rerouting the interrupt signal to the PG2 port. If INT2PG2 =1 then all other settings for PG2 are ignored.	
PG1_X_G3	The pulse codes can be output at ports PGØ & PG1 or PG2 & PG3. In I2C mode of communication, they can be optionally given out on PG2 and PG3, instead of PGØ and PG1.	0 = PG1 1 = PG3
PG0_X_G2		0 = PG0 1 = PG2
PG_DIR_IN	toggles outputs to inputs (PG3/bit23 to PG0/bit20).	0 = output 1 = input
PG_PULL_UP	Activates pull-up resistors in PG0 to PG3 lines; useful for mechanical switches.	Bit 16 = PG0 Bit 17 = PG1 Bit 18 = PG2 Bit 19 = PG3
DSP_FF_IN	Pin mask for latching flip-flop activation	Bit 12 = PG0 Bit 13 = PG1 Bit 14 = PG2 Bit 15 = PG3
DSP_MOFLO_EN	Activates anti-bouncing filter in PG0 and PG1 lines	Bit 9 for PG1 Bit 8 for PG0

2.6 DSP Configuration

The configuration of the DSP is done in configuration register 8. Relevant bits are:

DSP_SRAM_SEL, DSP_START, DSP_STARTONOVL, DSP_STARTONTEMP, DSP_STARTPIN, DSP_WATCHDOG_LENGTH, DSP_SPEED

Table 2-8

Parameter	**Description**	**Settings**
DSP_SRAM_SEL	Selects the program memory for the processor	0 = OTP 1 = SRAM
DSP_START	Startbit. Command for the processor; processor clock is started, program counter jumps to address zero and processing begins. After firmware completion, DSP stops its own clock! As the DSP is triggered by rising	0→1(rising edge) = start DSP

	edge, this bit is to be set '0' first and then '1'.	
DSP_START_EN<4..0>	see Vol1	
DSP_STARTPIN	Pin mask for DSP trigger	0 = FF0 1 = FF1 2 = FF2 3 = FF3
DSP_SPEED	Setting the DSP speed	1 = Standard (fast) 3 = Low-current (slow)

DSP Start

There are various options to trigger the DSP.

In slave operation:

- Trigger by external controller. This is done by successive clearing and setting the startbit DSP_START in configuration register 8.

In stand-alone operation:

- Trigger by pin. The trigger pin is selected between pins PG0 to PG3 by configuration parameters DSP_STARTPIN and PG0_X_PG2/PG1_X_PG3. Signal FFx triggers the DSP. FFx has to be reset in the firmware by setting DSP_x, e.g.
  BitS DSP_2
  BitC DSP_2
- Trigger by the end of a temperature measurement. This option is selected by configuration bit DSP_STARTONTEMP and is recommended for stand-alone operation with temperature measurement.
- Trigger on error. This option is enabled by setting configuration bit DSP_STARTONOVL. It should be used only if error handling is implemented in the software.

Watchdog

The watchdog is (now) based on the constant clock (5 kHz) and counts always, even if the DSP is halted. If the DSP doesn't reset the Watchdog within the configured watchdog time a power-on reset is generated => auto-boot. Status Flag POR_FLAG_Wdog is set.

The watchdog is implemented to handle situations where no CDC or RDC is running.

In slave applications the watchdog should be disabled. If the watchdog is used disarm the watchdog in advance to any SIF-Communication.

System Reset

In case the PCapØ2 is operated as a slave, not in self-boot mode, it is necessary to do the following actions after applying power:

1. Send opcode Power-up Reset via the serial interface, opcode 'h88.
2. Write the firmware into the SRAM by means of opcode "Write to SRAM".
3. Write the configuration registers by means of opcode "Write Config". Register 20 with the RUNBIT has to be the last one in order.
4. Send a partial reset, opcode 'h8A
5. Send a start command, opcode 'h8C

3 Instruction Set

The complete instruction set of the PCapØ2 consists of 29 core instructions that have unique op-code decoded by the CPU. Further, acam offers a set of libraries including common constant definitions and mathematical operations

The library family is intended to be continuously expanded and be a great help during software development.

Table 3-1 Instruction set

Simple Arithmetic	Miscellaneous	RAM access	Bitwise operation
add	resetWDG	rad	not
sign	powerOnReset	clear	and
sub	nop	load	or
inc	stop	load2exp	xor
		move	

Complex Arithmetic	Shift & Rotate	Unconditional jump	Bitwise
div	shiftL	jsb	bitC
mult	shiftR	jrt	bitS

Conditional jump
jcd
jCarC
jCarS
jEQ
jNE
jNeg
jOflC
jOflS
jPOS

3.1 Instructions

and	Bitwise AND
Syntax:	and p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r] p1 != p2
Calculus:	p1 := p1 & p2
Flags affected:	C O S Z
Bytes:	1
Description:	Bitwise AND (conjunction)
Category:	Bitwise operation

add	Addition
Syntax:	add p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r]
Calculus:	p1 := p1 + p2
Flags affected:	C O S Z
Bytes:	1
Description:	Addition of two registers
Category:	Simple arithmetic

bitC	Clear single bit
Syntax:	bitC p1
Parameters:	p1 = number 0 to 15
Calculus:	Set bit number p1 of the DSP output bits bit = 1
Flags affected:	-
Bytes:	1
Description:	Clear a single bit in the DSP output bits
Category:	Bitwise

bitS	Set single bit
Syntax:	bitS p1
Parameters:	p1 = number 0 to 15
Calculus:	Set bit number p1 of the DSP output bits bit = 1
Flags affected:	-
Bytes:	1
Description:	Set a single bit in the DSP output bits

Category:	Bitwise

clear	**Clear register**
Syntax:	clear p1
Parameters:	p1 = ACCU [a,b,r]
Calculus:	p1 := 0
Flags affected:	S Z
Bytes:	2
Description:	Clear addressed register to 0
Category:	RAM access

div	**Unsigned division**
Syntax:	div
Parameters:	-
Calculus:	Single div code: b := (a/r), a := Remainder * 2 N div codes: b := (a/r)*2^(N-1), a := Remainder * (2^N)
Flags affected:	S Z
Bytes:	1
Description:	Unsigned division of two 48-bits registers. When the div opcode is used once, the resulting quotient is assigned to register 'b'. The remainder can be calculated from 'a'. When N div opcodes are used one after another, the result in b := (a/r)*2^(N-1). Before executing the first division step, the following conditions must be satisfied: 'b' = 0, and 0<'a'<2*'r'. If this condition is not satisfied, you can shift 'a' until this is satisfied. After shifting, if a -> a* (2^ea) and r -> r* (2^er), then the resulting quotient b for N division steps is b:= (a/r) * 2^(1+ea-er-N) a = Remainder * (2^N)
Category:	Complex arithmetic

inc	**Increment register**
Syntax:	inc p1
Parameters:	p1 = ACCU [a,b,r]
Calculus:	p1 := p1 + 1
Flags affected:	C O S Z
Bytes:	1
Description:	Increment register
Category:	Simple arithmetic

jCarC	Jump on Carry Clear
Syntax:	jCarC p1
Parameters:	p1 = jumplabel
Calculus:	if (carry == 0) PC := p1
Flags affected:	-
Bytes:	2

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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt.
	If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization:
	`jCarS new_label`
	`jsb p1`
	`jrt`
	`new_label: ……..`
	In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.

Category:	Conditional jump

jCarS	Jump on Carry Set
Syntax:	jCarS p1
Parameters:	p1 = jumplabel
Calculus:	if (carry == 1) PC := p1
Flags affected:	-
Bytes:	2

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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt.
	If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization:
	`jCarSC new_label`
	`jsb p1`
	`jrt`
	`new_label: ……..`
	In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.

Category:	Conditional jump

jcd	Conditional Jump
Syntax:	jcd p1, p2
Parameters:	p1 = Flag or input port bit [63…0]. See section 2.3 for DSP Inputs.

	p2 = jumplabel
Calculus:	If (p1 == 1) PC := p2
Flags affected:	-
Bytes:	2
Description:	Program counter is set to target address if the bit given by p1 is set to one. The target address is given by using a jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt.
Category:	Conditional jump

jEQ	**Jump on Equal**
Syntax:	jEQ p1
Parameters:	p1 = jumplabel
Calculus:	if (Z == 0) PC := p1
Flags affected:	-
Bytes:	2

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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt.
	If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization:
	`jNE new_label`
	`jsb p1`
	`jrt`
	`new_label: ……..`
	In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.

Category:	Conditional jump

jNE	**Jump on Not Equal**
Syntax:	jNE p1
Parameters:	p1 = jumplabel
Calculus:	if (Z == 1) PC := p1
Flags affected:	-
Bytes:	2
Description:	Jump on not equal to 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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt. If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization: `jEQ new_label`

	jsb p1 jrt new_label: …….. In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.
Category:	Conditional jump

jNeg	Jump on Negative
Syntax:	jNeg p1
Parameters:	p1 = jumplabel
Calculus:	if (S == 1) PC := p1
Flags affected:	-
Bytes:	2
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 jumplabel. If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization: jPos new_label jsb p1 jrt new_label: …….. In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.
Category:	Conditional jump

jOvlC	Jump on Overflow Clear
Syntax:	jOvlC p1
Parameters:	p1 = jumplabel
Calculus:	if (O == O) PC := p1
Flags affected:	-
Bytes:	2
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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt. If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization: jOflS new_label jsb p1 jrt

	new_label: …..... In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.
Category:	Conditional jump

jOvIS	**Jump on Overflow Set**
Syntax:	jOvIC p1
Parameters:	p1 = jumplabel
Calculus:	if (O == 1) PC := p1
Flags affected:	-
Bytes:	2
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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt. If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization: jOflC new_label jsb p1 jrt new_label: …..... In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.
Category:	Conditional jump

jPos	**Jump on Positive**
Syntax:	jPos p1
Parameters:	p1 = jumplabel
Calculus:	if (S == O) PC := p1
Flags affected:	-
Bytes:	2
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 jumplabel. The conditional jump does not serve the stack. Therefore it is not possible to return by jrt. If the target address is beyond the range of current address (PC) +-128 bytes, then the assembler software will substitute this opcode for the following optimization: jNeg new_label jsb p1

	jrt new_label: …….. In this case the stack will be loaded with p1, and therefore the stack capacity will be reduced by one.
Category:	Conditional jump

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

jsb	**Unconditional Jump**
Syntax:	jsb p1
Parameters:	p1 = jumplabel
Calculus:	PC := PC from jsub-call
Flags affected:	-
Bytes:	2
Description:	Jump to subroutine without condition. The programm counter is loaded by the address given through the jumplabel. The subroutine is processed until the keyword 'jrt' 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). The stack is incremented by 1.
Category:	Unconditional Jump

load	**Load Accumulator**
Syntax:	load p1,p2
Parameters:	p1 = ACCU [a,b] p2 = 24-bit number
Calculus:	p1 := p2
Flags affected:	S Z

Bytes:	6

Description:	Move constant to p1 (p1=ACCU, p2=NUMBER)
	The following instruction is not allowed:
	load r, NUMBER
	This instruction is a macro that is replaced by the following opcodes:
	`rad NUMBER[23:18]`
	`rad NUMBER[17:12]`
	`rad NUMBER[11:6]`
	`rad NUMBER[5:0]`
	`rad 63`
	`move [a, b], r`
	Here the 24-bits number is split into four pieces, the symbol [xx:yy] indicates the individual bit range belonging to each piece. Please notice that the ram address pointer is changed during the operations, keep this in mind while coding.

Category:	RAM access

load2exp	**Load Accumulator with 2exp**
Syntax:	load2exp p1,p2
Parameters:	p1 = ACCU [a,b] p2 = 6-bit number
Calculus:	p1 := 2^p2
Flags affected:	S Z
Bytes:	2

Description:	Move 2^(p2) to p1(p1=ACCU, p2=NUMBER)
	The following instruction is not allowed:
	`load r, NUMBER`
	This instruction is a macro that is replaced by the following opcodes:
	`rad NUMBER[5:0]`
	`rad 62`
	`move [a, b], r`

Category:	RAM access

move	**Move**
Syntax:	move p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r]
Calculus:	p1 := p2
Flags affected:	S Z
Bytes:	1
Description:	Move content of p2 to p1
Category:	RAM access

mult	Multiply
Syntax:	mult
Parameters:	-
Calculus:	ab := (b * r)
Flags affected:	S Z
Bytes:	1
Description:	Unsigned multiplication of the content of ab and r registers. ab is the composition of the registers a and b, forming an 96-bits long register, where 'a' takes the most significant bits, and register 'b' takes the less significant ones. The result is stored in the composed register a and b. The register 'a' must be previously cleared. This instruction only executes one multiplication step, to execute a full 48-bits multiplication, this instruction must be executed 48 times. This has the disadvantage of being tedious to code, but also has the advantage of executing only the amount of arithmetic needed, if you don't need a 48-bits multiplication but N, where N<48, then you have only to execute N multiplication steps in order to complete the full N-bits multiplication. After one multiplication step, register 'a' contains ((a+(b[0]*r))>>1), and register 'b' contains { a[0], b[47:1] }. For example: lets denote the individual bits of register 'a' as a[47], a[46], a[45].....a[2], a[1], a[0], and lets denote a range of bits of 'a' as: a[3:0], meaning the 4 less significant bits of register 'a'. Then, after one multiplication step, a[46:0] = (a[47:0] + r[47:0] * b[0]) >> 1, where >> 1, means right shift by one position; the value of a[47] is zero, and b[47] = (a[0] + r[0] * b[0]), and b[46:0] = b[47:1]. The register r remains unchanged.
Category:	Complex arithmetic

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

not	Bitwise NOT
Syntax:	not p1
Parameters:	p1 = ACCU [a,b,r]
Calculus:	p1 := ~ p1
Flags affected:	C O S Z

Bytes:	1
Description:	Invert register
Category:	Bitwise operation

or	**Bitwise OR**
Syntax:	or p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r] p1 != p2
Calculus:	p1 := p1 | p2
Flags affected:	C O S Z
Bytes:	1
Description:	Bitwise OR (disjunction)
Category:	Bitwise operation

powerOnReset	**Power On Reset**
Syntax:	powerOnReset
Parameters:	-
Calculus:	-
Flags affected:	S Z
Bytes:	5

Description:	This is a symbolic opcode which is equivalent to the following instruction sequence:
	`bitC 54`
	`bitC 55`
	`bitS 55`
	`bitS 54`
	`bitC 55`

Category:	Miscellaneous

rad	**Set RAM Address Pointer**
Syntax:	rad p1
Parameters:	p1 = NUMBER [6-bit]
Calculus:	-
Flags affected:	1
Bytes:	1
Description:	Set pointer to ramaddress (range: 0..63) Note: rad _at_DPTR0 and rad _at_DPTR1 are instructions that will be seen in the firmware. With these opcodes, the address in the Data Pointer (DPTR0 &1 at RAM address 44 and 45) is taken as the address for

	the RAM address pointer. _at_DPTR0 is at address 285, _at_DPTR1 is at address 287.
	`rad _at_DPTR0`
	`move a, r`
	will move the contents of the address stored in DPTR0 to the A register.
	See also section 3.2.1.

Category:	RAM access

resetWDG	**Clear watch dog timer**
Syntax:	resetWDG
Parameters:	-
Calculus:	-
Flags affected:	-
Bytes:	5

Description:	Clear watchdog timer.
	This is a symbolic opcode which is equivalent to the following instruction sequence:
	`bitC 54`
	`bitC 55`
	`bitS 54`
	`bitS 55`
	`bitC 54`

Category:	Miscellaneous

shiftL	**Shift Left**
Syntax:	shiftL p1
Parameters:	p1 = ACCU [a, b]
Calculus:	p1 := p1<< 1
Flags affected:	S Z
Bytes:	1
Description:	Shift p1 left --> shift p1 register to the left, fill LSB with 0, MSB is placed in carry register
Category:	Shift and rotate

shiftR	**Shift Right**
Syntax:	shiftR p1
Parameters:	p1 = ACCU [a, b]
Calculus:	p1 := p1>> 1
Flags affected:	S Z
Bytes:	1
Description:	Signed shift right of p1 --> shift p1 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 [a,b]
Calculus:	When S = 0 => p1 := |p1|, S := (1 - p1/|p1|)/2 When S = 1 => p1 := - |p1|, S := (1 - p1/|p1|)/2
Flags affected:	S Z
Bytes:	1
Description:	The Signum flag takes the sign of accumulator, 0 when positive or 1 when negative. The accumulator changes its sign after the execution of this opcode, if and only if the Signum flag (before the execution) is 1. Zero is assumed to be positive.
Category:	Simple arithmetic

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

sub	**Subtraction**
Syntax:	sub p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r]
Calculus:	p1:= p1 – p2
Flags affected:	C O S Z
Bytes:	1
Description:	Subtraction of 2 registers. The following instructions are not allowed: add a,a. add b,b. add r,r
Category:	Simple arithmetic

xor	Bitwise XOR
Syntax:	xor p1,p2
Parameters:	p1 = ACCU [a,b,r] p2 = ACCU [a,b,r] p1 != p2
Calculus:	p1 := p1 ^ p2
Flags affected:	C O S Z
Bytes:	1
Description:	Bitwise XOR (antivalence)
Category:	Bitwise operation

3.2 Instruction Details

3.2.1 rad

Sets the RAM address. Typical example:

```
rad   12
move  a, r
```

Pointer

rad _at_DPTR0 and rad _at_DPTR1 are special instructions for indirect addressing. _at_DPTR0 and _at_DPTR1 are special RAM addresses 285 and 287 that have been defined in the firmware.

RAM addresses 44 and 45 are used as data pointers, named DPTR0 and DTPTR1.

By means of

```
rad DPTR0
move r, a
```

an address is loaded into DPTR0. With

```
rad _at_DPTR0
```

the address in DPTR0 is loaded.

Example: copy sequently RAM-content from one address-space to another

```
load a, C0_ratio
rad DPTR1
move r, a
load a, RES0
rad DPTR0
move r, a
```

```
load b, 8
jsb __sub_dma__

__sub_dma__:
; DPTR1 := source_address
; DPTR0 := destination address
; b:= length of dma
rad _at_DPTR1
move a, r
rad _at_DPTR0
move r, a
rad ONE
move a, r
rad DPTR0
add r, a
rad DPTR1
add r, a
sub b, a
jNE __sub_dma__
jrt
#endif
```

3.2.2 mult

The instruction “mult” is just a single multiplication step. To do a complete 48-bit multiplication this instruction has to be done 48 times. The multiplicands are in accumulators b and r. Every step takes the lowest bit of b. If it is one, r is added to accumulator a, else nothing is added. Thereafter a and b are shifted right. The lowest bit of a becomes the highest bit of b. Before the first step of the multiplication, a has to be cleared. The final result is spread over both accumulators a and b.

The use of mult is simplified by using the standard.h library. This library includes function calls for multiplications with arbitrary number of multiplication steps. E.g., a call of function mult_24 will do a 24-step multiplication.

Example 1: r= 5, b=5; 48-bit integer multiplication

Steps		a	b	r	
		'b0..0000	'b000000..000101	'b0..0101	b= 5; r = 5
1	+, →	'b0..0010	'b100000..000010	'b0..0101	r is added to a, a & b shifted right
2	→	'b0..0001	'b010000..000001	'b0..0101	a & b shifted right
3	+, →	'b0..0011	'b001000..000000	'b0..0101	r is added to a, a & b shifted right
4	→	'b0..0001	'b100100..000000	'b0..0101	a & b shifted right
5	→	'b0..0000	'b110010..000000	'b0..0101	a & b shifted right
6	→	'b0..0000	'b011001..000000	'b0..0101	a & b shifted right
47	→	'b0..0000	'b000000.0100110	'b0..0101	a & b shifted right
48	→	'b0..0000	'b000000..010011	'b0..0101	a & b shifted right

In many cases it will not be necessary to do the full 48 multiplication steps but much fewer. The necessary number of steps is given by the number of significant bits of b and also the necessary significant number of bits of the result.

But, if the multiplication steps are fewer than 48, the result might be spread between accumulators a and b. Doing an appropriate right shift of the multiplicand in r, and the appropriate number of multiplication steps, it is possible to ensure that the result is either fully in a or in b.

Example 2: 24-bit fractional number multiplication, result in a

Let's assume that multiplicand b is 12.5, given as 24-bit number with 4 integer and 20 fractional digits, and b has to be multiplied by 1.5. The result shall have 24 significant bits, too.

To have the final result fully in a, it is best to shift r as far as possible to the left. Therefore, r is shifted 46 bit to the left, r = 'h600000 000000. This left shift is easily done for constants.

The minimum number of multiplication steps is then given by the number of significant bits of b.

12.5*1.5 = b*2^expB * r*2^expR = b*2^-20 * r*2^-46; b='hC80000; r='h600000000000

Steps		a	b	r
		'h000000000000	'h000000C80000	'h600000000000
8	→	'h000000000000	'h00000000C800	'h600000000000
16	→	'h000000000000	'h0000000000C8	'h600000000000
19	→	'h000000000000	'h000000000019	'h600000000000
20	+,→	'h300000000000	'h00000000000C	'h600000000000
21	→	'h180000000000	'h000000000006	'h600000000000
22	→	'h0C0000000000	'h000000000003	'h600000000000
23	+,→	'h360000000000	'h000000000001	'h600000000000
24	+,→	'h4B0000000000	'h000000000000	'h600000000000

After 24 multiplication steps the full 24-bit result stands in a, starting at the highest significant bit. In many cases the result can be used in this form to do further mathematical processing, e.g. as parameter r in a further multiplication.

In case the true decimal value has to be calculated from the result, this is done by following formula:

product = a*2^steps+expR+expB = a*2^24+(-20)+(-46) = a*2^-42

'h4B0000000000*2-42 = 'h4B*2^-2 = 75*2^-2 = 18.75

Result in A:
Steps = expRes – expB – expR
Note: Steps >= Number of significant bits in B

Result in B:
Steps = expRes – expB – expR – 48
Note: Steps >= Number of significant bits in B

3.2.4 div

The instruction “div” is, like the multiplication, just a sinlge step of a complete division. The necessary number of steps for a complete division depends on the accuracy of the result. The dividend is in accumulator a, the divisor is in accumulator r. Every division step contains following actions:

leftshift b

compare a and r. If a is bigger or equal to r then r is subtracted from a and One is added to b

leftshift a

Start Conditions: 0 < a < 2*r, b = 0

Again, multiple division steps are implemented in the standard.h library to be easily used by customers. A call of function e.g. div_24 out of this library will do a sequence of 24 division steps. The result is found in b, the remainder in a.

With N division steps the result in b:= (a/r)+2^(N-1), a:= remainder*2^N.

Example 1: a = 2, r = 6, Integer division

Steps	a = 2	b	r = 6	
	000000..000010	0..00000	0..0110	a < r, leftshift b, a
1	000000..000100	0..00000	0..0110	a < r, leftshift b, a
2	000000..001000	0..00000	0..0110	leftshift b, a >= r: a-=r, b+=1, leftshift a
3	000000..000100	0..00001	0..0110	a < r, leftshift b, a
4	000000..001000	0..00010	0..0110	leftshift b, a >= r: a-=r, b+=1, leftshift a
5	000000..000100	0..00101	0..0110	

*Quotient = b * $2^{(1\text{-steps})}$ = 0.3125, Remainder = a*$2^{(\text{-steps})}$ = 4*2^{5} = 0.125*

The following two, more complex examples show a nice advantage of division over multiplication: The resolution in bit is directly given by the number of multiplication steps. With this knowledge, assembly programs can be written very effectively. It is easy to use only the number of division steps that is necessary.

Example 2: A = 8.75, R = 7.1875, Fractional number division, A & R with 4 fractional digits each.

8.75/7.1875 = a*2^{expA} / r*2^{expR} = a*2^{-4} / r*2^{-4}

Steps	a = 140	b	r = 115	
	1000 1100	0000 0000	0111 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
1	0011 0010	0000 0001	0111 0011	a < r, leftshift b, a
2	0110 0100	0000 0010	0111 0011	a < r, leftshift b, a
3	1100 1000	0000 0100	0111 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
4	1010 1010	0000 1001	0111 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
5	0110 1110	0001 0011	0111 0011	a < r, leftshift b, a
6	1101 1100	0010 0110	0111 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
7	1101 0010	0100 1101	0111 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
8	1011 1110	1001 1011	0111 0011	

*Quotient = b * $2^{(1+expA-expR-steps)}$ = 155 * $2^{(1-4+4-8)}$ = 1.2109*

Remainder = a$2^{(-steps-expR)}$ = 190*2^{12} = 0.0463*

Example 3: A = 20, R = 1.2, Fractional number division, R < A.

A and R are left shifted to display the fractional digits of R. Further, R has to be leftshifted till it is bigger than A/2.

20/1.2 = a*2^{expA} /r*2^{expR} = a*2^{-4} /r *2^{-8}

Steps	a = 320	b	r = 307	
	0001 0100 0000	0000 0000 0000	0001 0011 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
1	0000 0001 1010	0000 0000 0001	0001 0011 0011	a < r, leftshift b, a
2	0000 0011 0100	0000 0000 0010	0001 0011 0011	a < r, leftshift b, a
3	0000 0110 1000	0000 0000 0100	0001 0011 0011	a < r, leftshift b, a
4	0000 1101 0000	0000 0000 1000	0001 0011 0011	a < r, leftshift b, a
5	0001 1010 0000	0000 0001 0000	0001 0011 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
6	0000 1101 1010	0000 0010 0001	0001 0011 0011	a < r, leftshift b, a
7	0001 1011 0100	0000 0100 0010	0001 0011 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a

8	0001 0000 0010	0000 1000 0101	0001 0011 0011	a < r, leftshift b, a
9	0010 0000 0100	0001 0000 1010	0001 0011 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
10	0001 1010 0010	0010 0001 0101	0001 0011 0011	leftshift b, a >= r: a-=r, b+=1, leftshift a
11	0000 1101 1110	0100 0010 1011	0001 0011 0011	a < r, leftshift b, a
12	0001 1011 1100	1000 0101 0110	0001 0011 0011	

*Quotient = b * $2^{(1+expA-expR-steps)}$ = 2134 * $2^{(1-4+8-12)}$ = 16.6719*

The remainder is, as always, smaller than the denominator divided by 2^{steps} e.g. in the present case, remainder < 1.2 / 2^{12} = 0,0003

Steps = 1 + expA – expB – expRes

4 Assembly Programs

The PCapØ2 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. The user can write his own header files but also integrate the library 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 3 of this datasheet. In the following sections 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

Example:

```
CONST value 1
equal ((value + 3)/2)
```

4.1 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, CONST [name] [value] 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.	`jsb LABEL1`
		`LABEL1:`
		`...`

;	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.	`; this is a comment`

org	Sets a new origin in program memory for subsequent statements.	`org 0x23`
		`equal 0x332211`
		`; write 0x11 to address 0x23,`
		`; 0x22 to address 0x24 ...`

equal	Insert three bytes of user defined data in program memory, starting at the address as defined by **org**.	
#include	Include the header or library file named in the quotation marks **""** or brackets < >. The code will be added at the line of the include command. In quotation marks there might be just the file name in case the file is in the same folder as the program, but also the complete path. Names in brackets refer to the acam library with the fixed path \Programs\acam PCapØ2\lib.	`#include <rdc.h>` `#include "rdc.h"`

#ifdef
#elseif
#endif
	Directive to implement code or not, dependig on the value of the symbol following the #ifdef directive. Use e.g. to include header files only once into a program.	`#ifdef __standard_h__`
		`#else`
		`#define __standard_h__`
		`...`
		`#endif`

#define	Defines a symbol that will be interpreted as true when being analysed by the #ifdef directive	

4.2 Sample Code

In the following we show some sample code for programming loops in the various kinds, for the use of the load instruction and the rotate instruction.

4.2.1 "for" Loop

Assembler	C-Equivalent	Comment
load a, max not a inc a rad index move r, a do: ;{..} rad index inc r jCarC do	for(index=-max; index < 0; index++)	max := number of repetitions 2nd complement for max (~max+1) store (-max) to index loop body loop increment repeat while index < 0

4.2.2 "while" Loop

Assembler	C-Equivalent	Comment
do: rad expression move a, r jEQ done ;{..} clear a jEQ do done;	while (expression) {..}	 activate Status Flags for „expression". Jump if expression == 0 loop body unconditional jump without writing to program counter stack

4.2.3 "do - while" Loop

Assembler	C-Equivalent	Comment
do: ;{..} rad expression move a, r jNE do	do {..} while (expression)	 loop body activate Status Flags jump if expression != 0

4.2.4 "do - while" with 2 pointers

Assembler	C-Equivalent	Comment

load a, MW7
rad loopLimit
move r, a
load a, MW0
rad DPTR0
move r, a
load a, RES0
rad DPTR1
move r, a
do:
  rad _at_DPTR0
  move a, r
  rad _at_DPTR1
  move r, a
  rad loopLimit
  move a, r
  rad DPTR1
  inc r
  rad DPTR0
  inc r sub a, r
jCarS do
	loopLimit = *MW7
	
	ptrSource = *MW0;
	
	ptrSink = *Res0;
	
	do { *ptrSink++ =
	*ptrSource++ }
	
	while (ptrSource <= MW7)
		load max-address for ptrSource
		
		load ptrSource with source address
		
		load ptrSink with sink address
		
		loop body
		  load value from source
		  write value to sink
		  write max-address to a
		  increment sink address
		  increment source address
		  limitLoop - ptrSource
		repeat loop if ptrSource <= max-address

4.2.5 Load Negative Values

How to load a negative 24 bit value from the program memory

Assembler	C-Equivalent	Comment
load a, 5 not a inc a	a = -5	a = 'h000000_000005 a = 'hffffff_fffffa (:=-6) a = 'hffffff_fffffb (:=-5)

4.2.6 Load Signed Values

How to load a signed 24 bit value from the program memory

Assembler	C-Equivalent	Comment

load2exp a, 23
load b, <S24bC>
rad 0
move r, b
sub b, a
jCarC positive
  sub b, a
  move r, b
positive:
  move b, r
	b = <S24bC>	a=2^23
		reg0 = <S24bC>
		
		if(<S24bC> >= 2^23)
		  reg0 = <S24bC> - 2^24

4.2.7 Rotate Right A to B

To rotate a value right from Akku A to Akku B, Akku B and R must be set to zero. Afterwards with each *mult* command a single „rotate right from A to B“ is done. This function could be used e.g. to shift a 8-bit value to the highest byte in the register.

Assembler	C-Equivalent	Comment

`load a, 0xa3`
`clear b`
`move r, b`
`mult ; (8x)`
`mult`
`..`
`mult`
	`A = <U8bC>`
	`b = a << 40`	

5 Libraries

The **PICO**CAP assembler comes with a set of ready-to-use library functions. With these libraries the firmware can be written in a modular manner. The standard firmware 03.01.xx is a good example for this modular programming.

When the DSP has to be programmed by the user for a specific application or when the firmware ought to be modified, these library functions can be simply integrated into the application program without any major tailoring. They save programming effort for known, repeatedly used, important functions. Some library files are interdependent on other file(s) from the library.

The library functions are called header files (they have *.h extension) in the assembler software and have to be included in the main *.asm program.

The following are the header files that are supplied with the **PICO**CAP assembler as part of the standard firmware.

- standard.h
- PCapØ2a.h
- cdc.h
- rdc.h
- signed24_to_signed48.h
- dma.h
- pulse.h
- sync.h
- median.h

The input parameters, output parameters, effect on RAM contents etc. for each of these library functions is explained in the tables below.

NOTE:
In the standard firmware and in all the library files, the notation “ufdN” is used as a comment. This shows if the parameter is signed or unsigned and the number of fractional digits in the number, N. For e.g. ufd21 indicates that the parameter is an unsigned number with 21 digits after the decimal point, 21 fractional digits. If the u at the beginning is missing, it is a signed number.

5.1 standard.h

Function:	Standard math library for implementing multiplication, division and shift operations.
Input parameters:	For shift right (1-48): parameter in accumulator B For shift left (1-48) : parameter in accumulator A Multiplication (1-48 steps) : parameter in Accumulators B and R Division (1-48 steps) : Dividend in Accumulator A, Divisor in R
Output/Return value:	For shift right (1-48): Output in B For shift left (1-48) : Output in A Multiplication (1-48 steps) : Output in AB Division (1-48 steps): Quotient in B, Remainder can be calculated from R
Prerequisites	-
Dependency on other header files	-
Function call	shiftR_B_48, ..., shiftR_B_01 shiftL_A_48, ..., shiftL_A_01 mult_48, ..., mult_01 div_49, ..., div_01 __div_variable__ __mult_variable__
Temporary memory usage	4 locations – all declared and used in the "__temporary_variables__" address range given as input parameter by the user.
Changes any RAM content permanently?	No

Function	__div_variable__ variable number of division steps
parameter	Akku A: dividend Akku B: no of division steps (rad) __sub_standard_divisor__ := divisor
return Value	Akku B := (dividend/divisor)
call	jsb __div_variable__
local / temporary ram	4x: __sub_standard_divisor__ __var_index0__ __var_index1__ __sub_standard_AkkuC__

Function	__mult_variable__ variable number of multiplication steps
parameter	Akku A: no of multiplication steps Akku B: multiplier 1 (rad) __sub_standard_multiplier__ := multiplier 2
return Value	Akku AB := (multiplier 1 *multiplier 2)
call	jsb __mult_variable__
local / temporary ram	4x: __sub_standard_multiplier__ __var_index0__ __var_index1__ __sub_standard_AkkuC__

5.2 PCapØ2a.h

Function:	This is a standard library for PCapØ2A firmware projects. This library contains the major address-mappings and constant names for the PCapØ2A. This file should be always included. It contains no commands, so no pro- gram space is wasted
Input parameters:	-
Output/Return value:	The constants in the file are declared, these can be used further in the program.
Prerequisites	-
Dependency on other header files	-
Function call	-
Temporary memory usage	-
Changes any RAM content permanently?	-

5.3 cdc.h

Function:	Function for **C**apacitance-to-**D**igital **C**onversion. This module contains the subroutine to determine the capacitor ratios, dependent on measurement scheme and the compensation mode	
Input parameters:	__sub_cdc_differential__ : __sub_cdc_gain_corr__ : __persistent_cdc_first__ : __temporary_variables__ :	0 = single sensor 1 = differential sensor Factor for TCsg ufd21 Address where CDC results are to be stored Define address space for temporary variables, address < 39!
	tbd result as (C1-C0)/(C1+C0)	
Switches	#define __CDC_INVERSE__	Results in C1_ratio to C7_ratio are the reversal values (C0/C1 etc.)
	#define __SUB_CDC_FPP_x__	x maybe a value between 19..25 select the fraction point position of the results. Default = 21
	#define __CDC_VARIABLE_AVERAGE__	Activate variable averaging by DSP. If enabled, A Value != 0 must be written to __sub_cdc_dsp_avr__ else declare CONST __sub_cdc_dsp_avr__ x where x is the number for DSP averaging
Output/Return value:	Capacitance ratios C0_ratio, …, C7_ratio CDC_BUSY signals if DSP-Averaging is complete (0:=false; 1:=true)	
Prerequisites	Declare a constant ONE = 1	
Dependency on other header files	#include <standard.h>	
Function call	jsb __sub_cdc__	
Temporary memory usage	5 locations – all declared and used in the "__temporary_variables__" address range given as input parameter by the user.	
Changes any RAM content permanently?	Yes – 10 locations updated with capacitance ratio results in the address range specified by the user in __persistent_cdc_first__ C0_ratio C1_ratio C2_ratio C3_ratio C4_ratio C5_ratio	

	C6_ratio C7_ratio DSP_C_AVR_CNT CDC_BUSY

5.4 rdc.h

Function:	Function for **R**esistance-to-**D**igital **C**onversion. This module contains the subroutine to determine the resistor ratios.
Input parameters:	__persistent_rdc_first__ : address where RDC results are to be stored __temporary_variables__ : define address space for temporary variables
Output/Return value:	Resistance ratios R0_ratio, R1_ratio, R2_ratio
Prerequisites	none
Dependency on other header files	#include <standard.h>
Function call	jsb __sub_rdc__
Switches	#define __SUB_RDC_FPP_x__ where x is the number of fraction (fix point position??) point position for results R0_ratio to R2_ratio. Default: 21
Temporary memory usage	1 location - declared and used in the "__temporary_variables__" address range given as input parameter by the user.
Changes any RAM content permanently?	Yes – 3 locations updated with resistance ratio results in the address range specified by the user in __persistent_rdc_first__ R0_ratio R1_ratio R2_ratio

5.5 signed24_to_signed48.h

Function:	This function is used to type-cast a 24-bit signed number to 48-bit signed value. For use e.g. with values transferred by PARA-Registers to a full 48-bit signed value.
Input parameters:	Accumulator B = signed 24bit value __temporary_variables__ : define address space for temporary variables
Output/Return value:	Accumulator B = signed 48bit Value

Prerequisites	-
Dependency on other header files	-
Function call	jsb __sub_signed24_to_signed48__
Temporary memory usage	1 location - declared and used in the "__temporary_variables__" address range given as input parameter by the user.
Changes any RAM content permanently?	Accumulator A is used in this subroutine, it will be overwritten.

5.6 dma.h

Function:	„Direct Memory Access" – This library file contains a subroutine to copy sequential RAM-content from one address-space to another. The number of RAM values to be copied can be specified.
Input parameters:	Accumulator B : number of values to be copied DPTR1 : source RAM block address DPTRO : destination RAM block address
Output/Return value:	The contents, i.e. the specified number of values are copied from the source RAM block to the destination RAM block.
Prerequisites	none
Dependency on other header files	-
Function call	jsb __sub_dma__
Temporary memory usage	-
Changes any RAM content permanently?	Yes, the destination RAM block

5.7 pulse.h

Function:	Linearization function specifically to determine the pulse-output value: Accumulator B = __sub_pulse_slope__ * Accu. B + __sub_pulse_offset__ Return Value is limited by 0 <= Akku B < 1024	
Input parameters:	Accumulator B : __sub_pulse_slope__ : __sub_pulse_offset__ : __temporary_variables__ :	input value, unsigned, 21 fractional digits constant factor, signed, 4 fractional digits constant summand, signed, 1 fractional digit define address space for temporary variables
Output/Return value:	The pulse output signals are generated	
Prerequisites	Declare a constant ONE = 1	
Dependency on other header files	-	
Function call	jsb __sub_pulse__	
Temporary memory usage	1 location - declared and used in the "__temporary_variables__" address range given as input parameter by the user.	
Changes any RAM content permanently?	No	

5.8 sync.h

Function:	The sync-filter (aka sin(x)/x) or rolling average filter is a filter function that determines the average for the last N values specified by the user in "__sub_sync_FilterOrder__ ".	
Input parameters:	Accumulator B : __sub_sync_FilterOrder__ : __persistent_sync_first__ : __temporary_variables__ :	input to be filtered filter order, depth of filtering address where the filtered results are stored define address space for temporary variables
Output/Return value:	The averaged value is passed back in Accumulator B. Additionally the filtered results are updated in the RAM.	
Prerequisites	Declare a constant ONE = 1 Filter must be initialized by -> jsub __sub_sync_initial__	
Dependency on other	-	

header files		
Function call	jsb __sub_sync__	
Temporary memory usag	1 location - declared and used in the "__temporary_variables__" address range given as input parameter by the user.	
Changes any RAM content permanently?	Yes –RAM locations updated with filtered results in the address range specified by the user in __persistent_sync_first__. Number of RAM locations depends on the filter order.	
	ringMemFirst : ringMemLast : FilterAkku : currentRingPos : AkkuDivider :	start of filter-memory last field of the filter memory sum of all memory-fields index Pointer; points to the current memory field 2^42 * FilterOrder

5.9 median.h

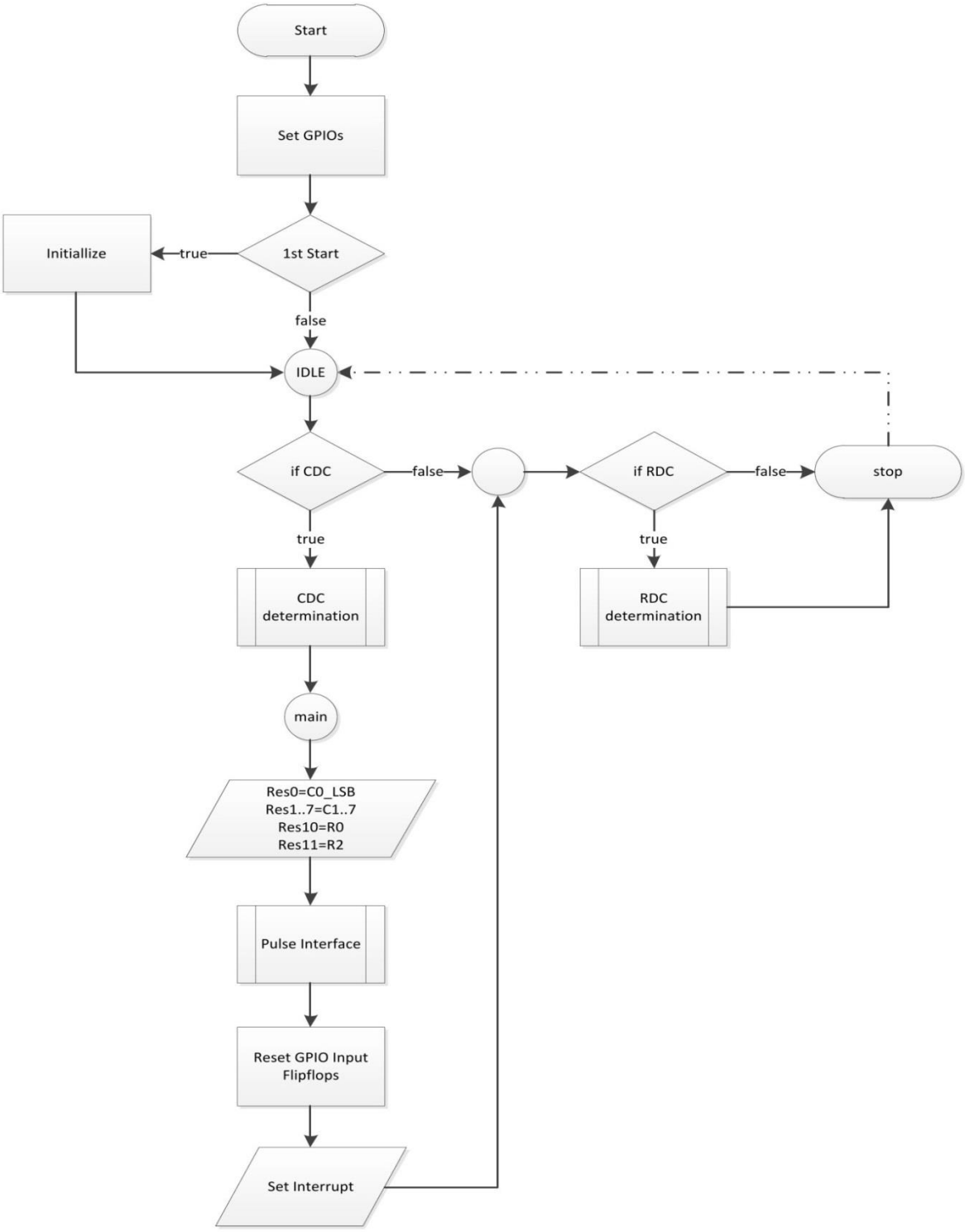
Function:	This is a quasi-median-filter. With __sub_median_FilterOrder__ the depth of the memory is defined. Each new Value (X) will be compared with the current median value, Is the new value smaller or equal to the median value, the last value in the list will be replaced by X. Otherwise the first value in the list will be replaced by X. Afterwards the complete list is sorted. The value at the very middle of the list is returnd as a new median.	
Input parameters:	Accumulator B : __sub_ median _FilterOrder__ : __persistent_median_first__ : __temporary_variables__ : __sub_median1_FilterOrder__: __persistent_median1_first__:	Input to be filtered. Filter order, depth of filtering Address where the filtered values are to be stored Address space for temporary variables Filter order for a second median Filter address where the second median filter values are to be stored
Output/Return value:	The new median is returned in Accumulator B.	
Prerequisites	Declare a constant ONE = 1	
Dependency on other header files	-	
Switches	#define __sub_median_filter1_enable__ if a second median filter will be used this is the switch to activate	

Function call	jsb __sub_median__ jsb __sub_median1__	
Temporary memory usage	2 locations - declared and used in the "__temporary_variables__" address range given as input parameter by the user.	
Changes any RAM content permanently?	Yes – RAM locations updated with filtered results in the address range specified by the user in __persistent_median_first__. Number of RAM locations depends on the filter order.	
	__sub_median_list_first__ : __sub_median_list_middle__ : __sub_median_list_last__ :	Start of filter memory middle field of the filter memory last field of the filter memory

6 Examples

6.1 Standard Firmware, Version 03.01.02

Figure 6-1: Main Loop Flowchart



Code snippets:

a) Identification of firmware

The following code writes the version of the firmware into a specific address of the program code:

```
org FW_VERSION
      equal FWG_Capacitance + FWT_Standard + 02
```

b) Check measurement status

These lines check whether measurement data are available or not. If they are, the program jumps into the sub routines given by the libraries. The CDC writes data alternately into two banks. Therefore, both banks have to be checked for valid data.

```
jcd    BANK0VALIDN, MK_QUERY_FL1 ;Jump if a CDC result is not yet available in Bank0
       jsb    __sub_cdc__        ;If result available - call subroutine for Capacitor
                                 ;to Digital conversion
      jsb    MK_main
MK_QUERY_FL1:                    ;Checking if CDC result is available in Bank1
jcd    BANK1VALIDN, MK_QUERY_FL2 ;Jump if a CDC result is not yet available in Bank1
       jsb __sub_cdc__           ;If result available - call subroutine for Capacitor
                                 ;to Digital conversion
       jsb MK_main
MK_QUERY_FL2:                    ;Checking if temperaure measurement (RDC) is running
jcd    TENDFLAGN, MK_RO_STOP     ;Jump if a meas. is still running & RDC result is
                                 ;not yet available
       jsb __sub_rdc__           ;If result available -  call subroutine Resistor to
                                 ;Digital conversion
```

c) Provide data to read-registers

After the subroutines __sub_cdc__ and __sub_cdc__ have been called, the results in form of Cs/Cref and Rs/Rref ratios are found in dedicated RAM space. With the following code the results are copied to the read registers. It is very simple thanks to subroutine __sub_dma__ from the acam library.

```
MK_main:                         ; Copying the CDC result to read-registers
load         a, C0_ratio         ; Loads the accumulator with first result
rad          DPTR1               ; Source address pointer
move         r, a
load         a, RES0             ; First result
rad          DPTR0               ; Destination address pointer
move         r, a
```

```
load        b, 8                ; 8 - No. of locations to be copied
jsb __sub_dma__                 ; This copies 8 address contents from the source
                                      ;location to the destination location

rad         R0_ratio            ; Copy the RDC results to the result registers
move        a, r                ; Copying only 2 results
rad         RES10
move        r, a
rad         R2_ratio
move        a, r
rad         RES11
move        r, a
```

d) Set the pulse interface

The offset and slope of the pulse outputs is typically defined in the parameter registers of PCapØ2.

```
CONST pulse_select  PARA2   ; bits<7..4> - pulse1_select
                            ; bits<3..0> - pulse0_select, add this bits to address C0_ratio
CONST pulse_slope0  PARA3   ;   signed 19 integer + fd4
CONST pulse_offset0 PARA4   ;   signed 22 integer + fd1
CONST pulse_slope1  PARA5   ;   signed 19 integer + fd4
CONST pulse_offset1 PARA6   ;   signed 22 integer + f1
```

The following is the calculation of linear function with the given slope and offset and thus scaling the pulse output to the necessary range.

```
; ---------- Pulse 0 -----------------------------
rad    pulse_slope0
move   b, r
jsb __sub_signed24_to_signed48__
rad    Slope
move   r, b                 ; Slope m

rad    pulse_offset0
move   b, r
jsb __sub_signed24_to_signed48__
rad    Offset
move   r, b                 ; Offset b

rad    _at_DPTR0            ; Getting the result x to be linearized
```

```
move   b, r
clear  a
rad    Slope
jsb    mult_24          ; Calculating m*x, result present in lower 24 bits of a and
                              ; upper 24 bits of b
rad    Offset           ; Taking only result in 'a' as final result
add    a, r             ; Calculating m*x + B
shiftR a                ; To account for only 1 digit after the decimal point
finally
rad    AkkuC
move   r, a

jPos MK_Pulse0_GE_Zero     ;  Scaling to minimum 0 : if( a < 0 ) a = 0
rad AkkuC
sub r, a
move a, r
```

After the result has been corrected by linearization it has to be clipped to the 0 to 1023 output range of the PCapØ2 pulse interface:

```
MK_Pulse0_GE_Zero:
load2exp b, 10           ; Scaling to maximum 1023 : if( a >= 1024) a = 1023
sub a, b
jNeg MK_Pulse0_s_1024
rad ONE
sub b, r                 ; b = 1023
rad AkkuC
move r, b
MK_Pulse0_s_1024:
rad AkkuC
move b, r
rad PULSE0
move r, b                ; PCapØ2 can output the value at PULSE0 output
```

7 Miscellaneous

7.1 Bug Report

7.2 Document History

17.01.2013 First release

16.07.2013 Version 0.1 released, section 2.5 GPIO table expanded

16.08.2013 Version 0.2 released, section 3 expanded with new opcodes

- Bitwise operation: not, and, or, xor
- Simple arithmetic: inc