acam mess • electronic

PICOCAP®

Datasheet

PCapØ1Ax DSP

Single-chip Solution for Capacitance Measurement

Description of the Digital Signal Processor

August 1st, 2011

Document-No.: DB_PCapØ1_DSP_e V0.0

Published by acam-messelectronic gmbh

© copyright 2011 by acam-messelectronic gmbh, Stutensee, Germany

Legal note

The present manual (data sheet and guide) is still under development, which may result in corrections, modifications or additions. acam cannot be held liable for any of its contents, neither for accuracy, nor for completeness. The compiled information is believed correct, though some errors and omissions are likely. We welcome any notification, which will be integrated in succeeding releases.

The acam recommendations are believed useful, the firmware proposals and the schematics operable, nevertheless it is of the customer's sole responsibility to modify, test and validate them before setting up any production process.

acam products are not designed for use in medical, nuclear, military, aircraft, spacecraft or life-support devices. Nor are they suitable for applications where failure may provoke injury to people or heavy material damage. acam declines any liability with respect to such non-intended use, which remains under the customer's sole responsibility and risk. Military, spatial and nuclear use subject to German export regulations.

acam do not warrant, and it is not implied that the information and/or practice presented here is free from patent, copyright or similar protection. All registered names and trademarks are mentioned for reference only and remain the property of their respective owners. The acam logo and the PicoCap logo are registered trademarks of acam-messelectronic gmbh, Germany.

Support / Contact

For a complete listing of direct sales contacts, distributors and sales representatives visit the acam website at:

http://www.acam.de

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

sales@acam.de   or by phone   +49 7244 7419 0.

Table of Contents

1 System Overview 1-1

2 DSP & Environment
- 2.1 RAM Structure 2-3
- 2.2 SRAM / OTP 2-7
- 2.3 DSP Inputs & Outputs 2-8
- 2.4 ALU Flags 2-12
- 2.5 DSPOUT – GPIO Assignment 2-13
- 2.6 DSP Configuration 2-16

3 Instruction Set
- 3.1 Instructions 3-2
- 3.2 Instruction Details 3-12

4 Writing Assembly Programs
- 4.1 Directives 4-3
- 4.2 Sample Code 4-4

5 Libraries
- 5.1 standard.h 5-3
- 5.2 pcap01a.h 5-4
- 5.3 cdc.h 5-4
- 5.4 rdc.h 5-5
- 5.5 signed24_to_signed48.h 5-6
- 5.6 dma.h 5-6
- 5.7 pulse.h 5-7
- 5.8 sync.h 5-7
- 5.9 median.h 5-8

6 Examples
- 6.1 Standard Firmware, 6-2
- 6.2 xx 6-2

7 Miscellaneous
- 7.1 Bug Report 7-1
- 7.2 Document History 7-1

1 System Overview

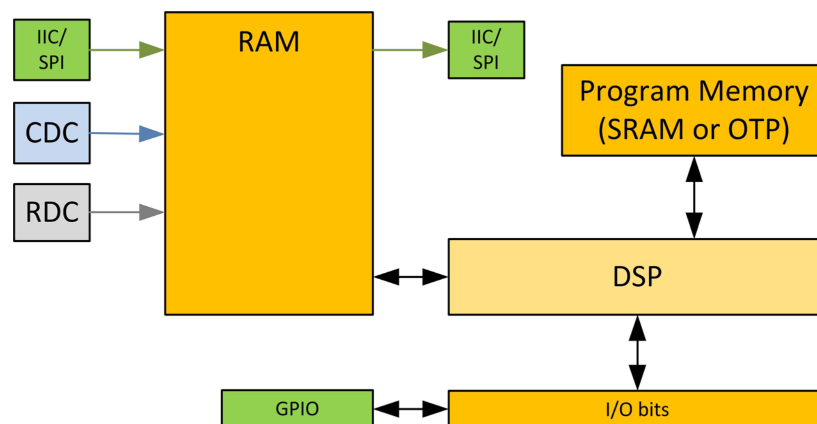
This datasheet describes the 48-DSP of the PCapØ1A. It describes only the items related to the DSP itself. For all other issues please refer to the PCapØ1A-0301 datasheet.

A 48-Bit digital signal processor (DSP) in Harvard architecture has been integrated to the PCap01. 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 PCap01. 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 simpliest form, the DSP transfers the pure time measurement information from the CDC/RDC to the read registers without any further processing. 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 03.01.xx.

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



1 System Overview

2 DSP & Environment 2-2

2.1 RAM Structure **2-3**

2.1.1 Registers 0 to 43 2-3

2.1.2 Registers 44 to 47 2-4

2.1.3 Register 48, Shift Register 2-5

2.1.4 Read Registers 49 to 61 2-5

2.1.5 Read Register 62 2-6

2.1.6 Read Register 63 2-6

2.1.7 Write Registers 50, 51 2-6

2.1.8 Write Registers 54 to 61 2-6

2.2 SRAM / OTP **2-7**

2.2.1 Memory Management 2-7

2.2.2 OTP 2-8

2.3 DSP Inputs & Outputs **2-8**

2.4 ALU Flags **2-12**

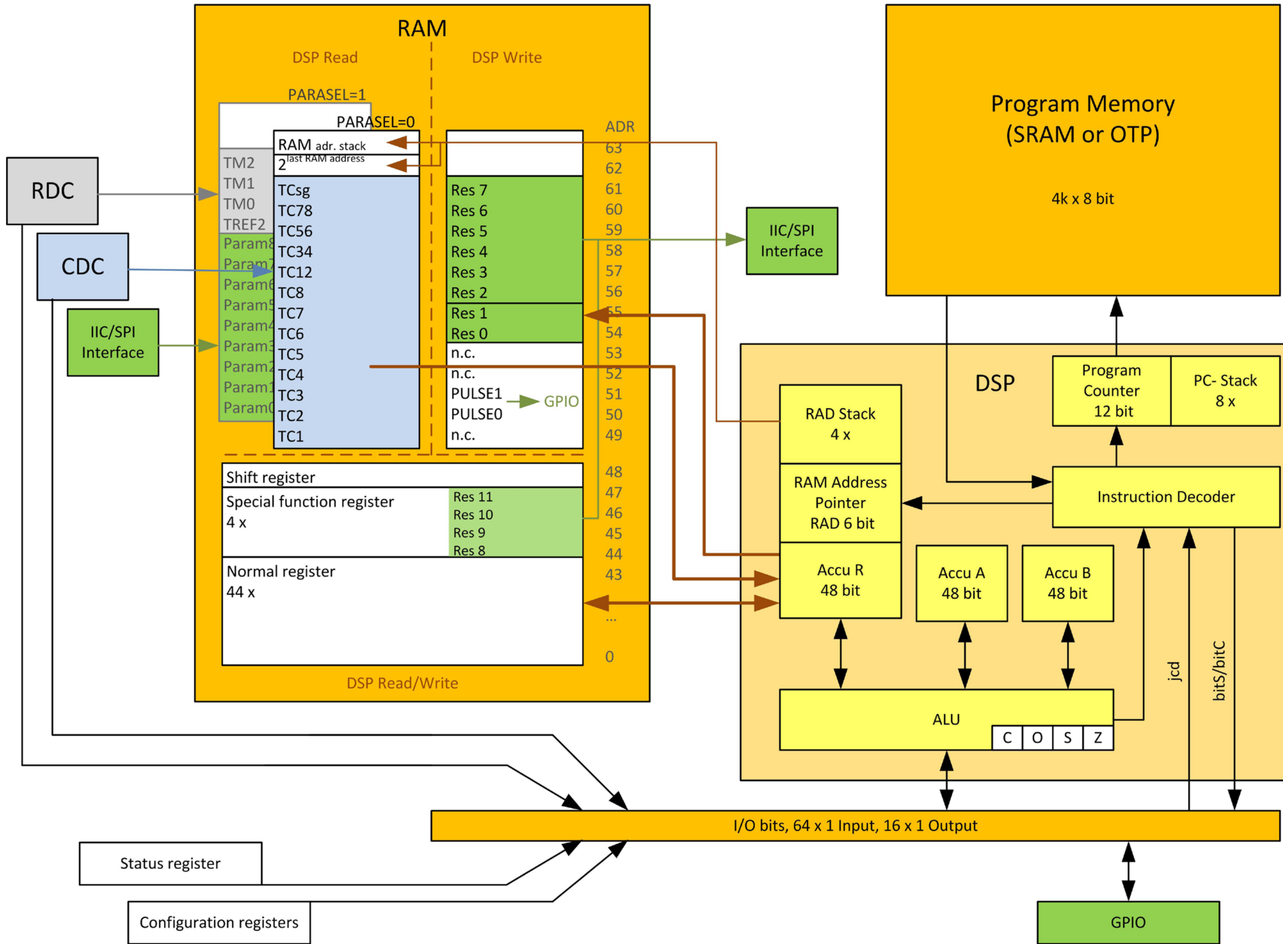
2.5 DSPOUT – GPIO Assignment **2-13**

2.6 DSP Configuration **2-16**

2 DSP & Environment

The detailed structure of how the DSP is implemented into the PCap01 is shown in figure 2-1.

Figure 2-1: DSP Environment



The 48-bit DSP has read/write access to the RAM. While the lowest 44 registers can be used free, the higher ones are used for special information. In read access, the DSP can get the measurement raw data from address space 49 to 61. If PARASEL = 0 this is the CDC data, if PARASEL = 1 it is the RDC data and the parameter registers' content. By write access the DSP provides the output data to either the serial interfaces (addresses54 to 61 and 44 to 47) or to the PDM/PWM interfaces (addresses 50 and 51).

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 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

2 DSP & Environment

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 2.3.1. The DSP can read these bits by means of instruction jcd (conditional jump) and set those bts 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 64 registers with size of maximum 48 bit. The DSP has free write and read access to registers address 0 to 48, all 48 bits wide. The RAM space address 49 to 63 is different for read and write. The read section itself is doubled and the selection between the two is made by setting parameter PARASEL. The data in the read section are the raw data as they come from the CDC (PARASEL=0), and data from the RDC as well as the parameters (PARASEL=1). 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.

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

In the following we explain the various RAM sections in detail.

2.1.1 Registers 0 to 43

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
```

2 DSP & Environment

Table 2-1: RAM Structure in Detail

RAM Read				
Addr.	Description			Bits
63	RAM Address Stack			24
62	2^(last RAM Address)			48
	PARASEL = 0	Bits	PARASEL = 1	
61	TCsg	37	TM2	37
60	TC78	37	TM1	37
59	TC56	37	TM0	37
58	TC34	37	TMREF	37
57	TC12	37	PARA8	24
56	TC8	37	PARA7	24
...	...		...	
49	TC1	37	PARA0	24
48	Shift register			48
47	RQ47/RES11			48/24
46	RQ46/RES10			48/24
45	RQ45/RES9/DPTR1			48/24
44	RQ44/RES8/DPTR0			48/24
43	Register 43			48
...	...			
0	Register 0			48

RAM Write		
Addr.	Description	Bits
63	n.c.	
62	n.c.	–
61	RES7	24
...	...	...
56	RES2	24
55	RES1	48
54	RES0	48
53	n.c.	
52	n.c.	
51	PULSE1	11
50	PULSE0	11
49	n.c.	
48	Shift register	48
47	RQ47/RES11	48
46	RQ46/RES10	48
45	RQ45/RES9/DPTR1	48
44	RQ44/RES8/DPTRØ	48
43	Register 43	48
...	...	...
0	Register 0	48

2.1.2 Registers 44 to 47

This RAM space can be used as normal register.

As a specialty, registers 44 and 45 can be used as data pointers DPTRØ and DPTR1. They can be used for indirect addressing. See section 3.2.1 for further information.

Additionally, the lower 24 bit of those four registers can be read via SPI/IIC. These lower 24 bit is the content of read registers Res 8 to Res 11 (read address 11 to 14). In case registers 44 and 45 are used as pointers the reading from those registers via interface has to be synchronized with the firmware. This is the case e.g. with the standard firmware.

2 DSP & Environment

2.1.3 Register 48, Shift Register

This register can be used as a normal RAM cell. Additionally, it can be used for 8/24/40 bit right shift operations. The right shift is selected by bits SHI48M00_OUT and SHI48M01_OUT. Those are write bits 12 and 13 out of the 16 DSP output bits.

Table 2-2: Shift register

Bit 13	Bit 12	Right shift
SHI48M01_OUT	SHIM00_OUT	
0	0	0
0	1	1 byte / 8 bit
1	0	3 byte / 24 bit
1	1	5 byte / 40 bit

Example: Make a one byte right shift

```
bitS   SHI48M00_OUT
bitC   SHI48M01_OUT
rad    48
move   r,a
move   a,r
```

2.1.4 Read Registers 49 to 61

The content depends on the setting of the DSP I/O bit 58, PARASEL.

PARASEL = 0:

Discharge time data of the capacitance measurement. TC1 = discharge time at port PC0, TC2 = discharge time at port PC1, ... TC78 = discharge time at ports PC6 and PC7 (external compensati-on on), TCsg = discharge time with all ports off (internal comensation on).

The data have 37 bit.

PARASEL = 1:

Registers 49 to 57 show the 24-bit parameter data as they are given by configuration registers 11 to 19, Param0 to Param9. They can be used in the DSP program e.g. to set the offset and slope of the pulsed ouputs.

Registers 58 to 61 contain the 24 bit discharge time data of the temperature measurement.

2 DSP & Environment

2.1.5 Read Register 62

This register is the two's exponent of the RAD stack. The exponent needs to be written into the RAM address. The result can be read from register 62. In the assembler the necessary 3 instrunctions are merged into one:

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

Is the same as

```
rad 10
rad 62
move a,r
```

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

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

2.1.6 Read Register 63

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 63
move a,r
```

2.1.7 Write Registers 50, 51

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 11 bit wide.

2.1.8 Write Registers 54 to 61

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 from address 0 to 7.

Addresses 54 and 55 are 48 bit while the others are 24 bit wide only.

2 DSP & Environment

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.

2 DSP & Environment

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 graphical 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 not be set back.

2.2.2 OTP

The PCap device 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 controlleds by means of instructions bitS/bitC (bit set/bit clear).

Table 2.4: DSP Inputs/Outputs

Bit Name	Description	Type	Read Bit #	Write Bit #
FLAG1	Flags for free use. A typical example would be to use this flag to indicate an initialization or calibration state.	Flag	63	15
FLAG0		Flag	62	14
FLAG0SHIM-48M01	Control bits for the shift register. See 2.1.3 for the desciption of how to use them.	Out	61	13
FLAG0SHIM-48M00		Out	60	12
DSP_SNC	This bit starts a new CDC measurement (Start-New-Cycle by DSP).	Out	59	11

2 DSP & Environment

Bit Name	Description	Type	Read Bit #	Write Bit #
PARASEL	This parameter selects between the two Read RAM blocks address 49 to 63 PARASEL = 0: CDC data PARASEL = 1: RDC data and parameters	Out	58	10
TFLAGRES	Temperature reset. This flag has to be set 1 and back 0 after each RDC measurement. Otherwise a new RDC measurement is not possible.	Out	57	9
Interrupt	Sets the interrupt (pin INTN)	Out	56	8
DSP_OUT<7...0>	Status feedback of the 8 general DSP outputs (Write bits 0 to 7).	IN	55...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	
ALU_NEG		Status	40	
DSP_IN_TRIGN	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	39	
n.c.			38,37	
DSPOVLSTAN*	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	36	
MUP_ERRN*	Flag = LOW indicates that a multi-pulse error has started the DSP. This flag is reset by a STOP instruction at the end of the firmware. For acam internal use only.	Start trigger	35	
DSPTEMPSTAN*	Flag = LOW indicates that an RDC measurement has started the DSP. Therefore, DSP_STARTON-TEMP has to be set (configuration register 8). This flag is reset by a STOP instruction at the end of the firmware.	Start trigger	34	

2 DSP & Environment

Bit Name	Description	Type	Read Bit #	Write Bit #
BANK1VALIDN*	Flag = LOW indicates that a CDC measurement has started the DSP. The CDC writes the measurement data TC1, TC2 etc. into two banks alternately. Those two flags indicate which bank is currently active. Therefore, in the firmware both flags have to be checked (see sample firmware PCap01-standard).	Start trigger	33	
BANK0VALIDN*		Start trigger	32	
PORTMASKN <7…0>	These bits show the content of CMEAS_PORT_EN, configuration register 2. The DSP can read these bits to decide which calculations need to be done.	Config Reg	31…24	
PORTERRN <7…0>	These bits indicate an error on one of the capacitance ports. They are the same as the CAP_ERROR_PC bits in the status register.	Status	23…16	
MR2N	Indicates whether measure mode 2 is set or not. 0 = MR2, 1 = MR1	Config Reg	15	
FULLCOMPN	Indicates whether full compensation in capacitance measurement is on or not. 0 = on, 1 = off.	Config Reg	14	
DIS_COMPN	Flag = 0 indicates that the internal capacitance compensation is switched off (CMEAS_BITS, configuration register 2).	Config Reg	13	
DIS_COMPFN	Flag = 0 indicates that the full capacitance compensation is switched off (CMEAS_BITS, configuration register 2).	Config Reg	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 forntend is active.	Status	9	
PS_CAL_ERRN	acam internal use only	Status	8	
PS_UNLOCKEDN 1	acam internal use only	Status	7	

2 DSP & Environment

Bit Name	Description	Type	Read Bit #	Write Bit #
PS_UNLOCKEDN 0	acam internal use only	Status	6	
TEMPERRN	Flag = bit 3 of status register. Indicates whether an error occurred during the temperature measurement. 0 = error, 1 = no error	Status	5	
TENDFLAGN	Flag = bit 22 of status register. Indicates the end of the temperature measurement. 0 = measurement done, 1 = measurement running.	Status	4	
DSP_7	Those two outputs are used by the DSP for - Reset watchdog - INI_RESET by DSP	Out		7
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 assigment 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 DSP & Environment

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	jOvlC, jOvlS	>= -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. i.e. for A - B: if $A \geq B \rightarrow C = 1$; if $A < B \rightarrow C = 0$.

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

2 DSP & Environment

2.5 DSPOUT – GPIO Assignment

PCap01 is very flexible with assignment of the various GPIO pins to the DSP inputs/outputs. The following table shows the possible combinations.

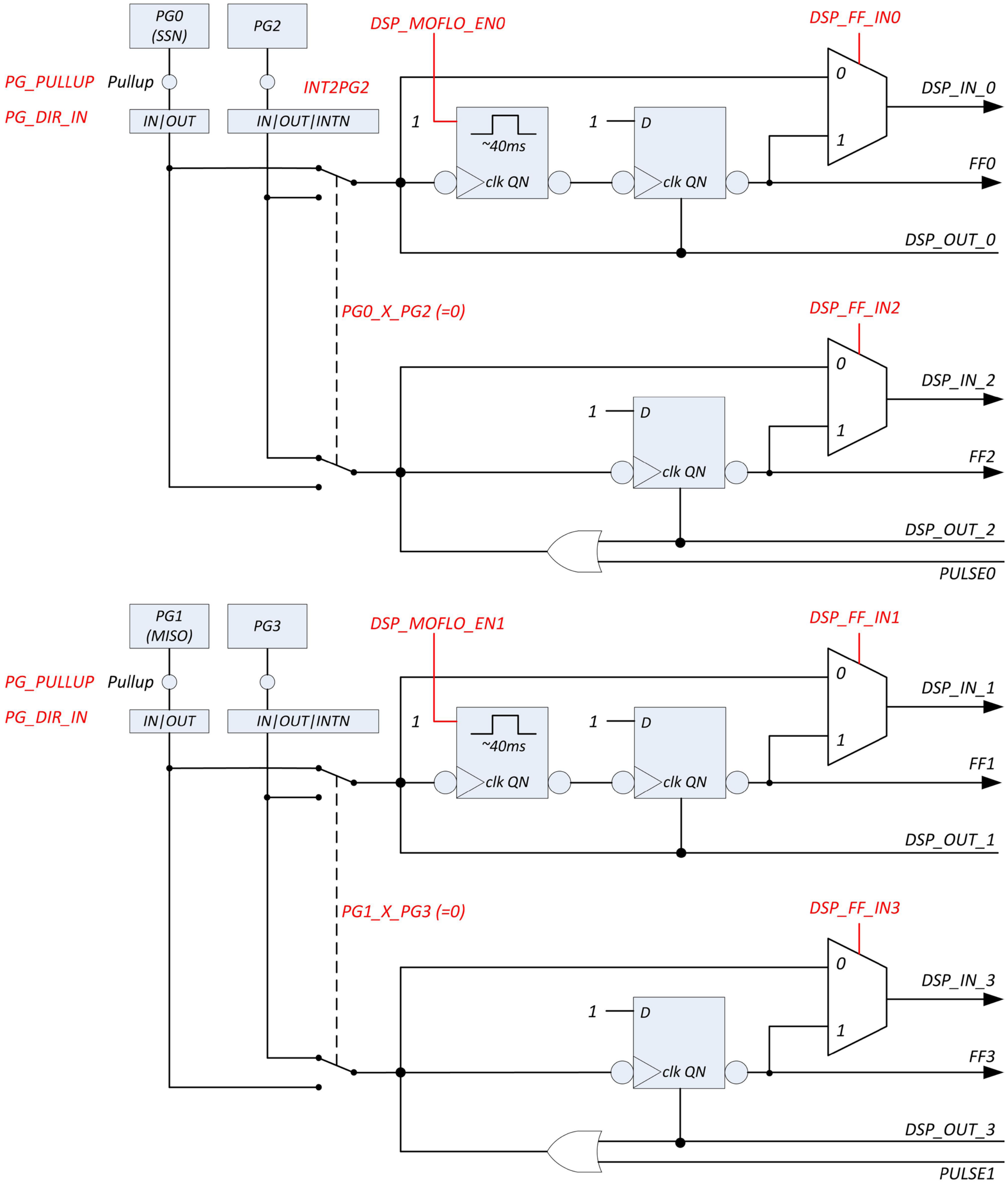
Table 2-6: Pin Assignment

External Port	Description	In/Out
PG0	SSN (in SPI-Mode)	in
	DSPØ or DSP2	in* / out
	FF0 or FF2	in*
	Pulse0	out
PG1	MISO (in SPI-Mode)	out
	DSP1 or DSP3	in* / out
	FF1 or FF3	in*
	Pulse1	out
PG2	DSPØ or DSP2	in* / out
	FF0 or FF2	in*
	Pulse0	out
	INTN	out
PG3	DSP1 or DSP3	in* / out
	FF1 or FF3	in*
	Pulse1	out
PG4	DSP4 (output only)	out
PG5	DSP5 (output only)	out

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

2 DSP & Environment

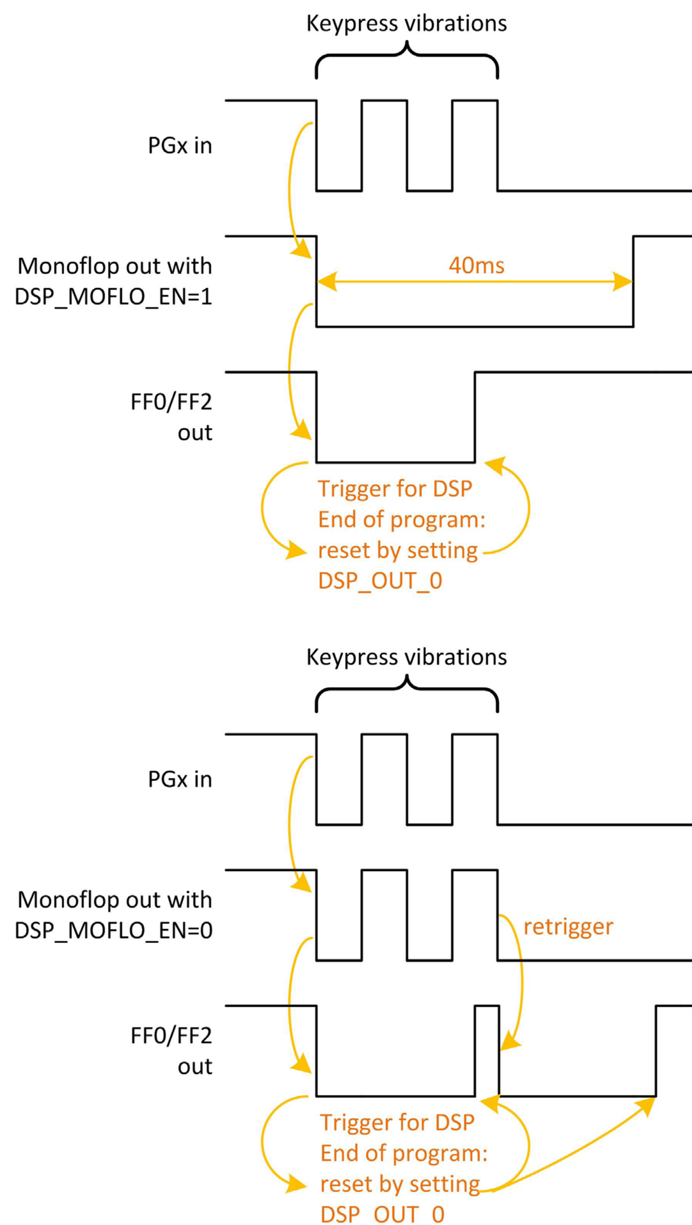
Figure 2-2: GPIO Assignment



2 DSP & Environment

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

Figure 2-3: Port trigger timing



2 DSP & Environment

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

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_SRMA_SEL, DSP_START, DSP_STARTONOVL, DSP_STARTONTEMP, DSP_STARTPIN, DSP_ WATCHDOG_LENGTH, DSP_SPEED

Parameter	Description	Settings
DSP_SRAM_SEL	Selects the program memory for the processor	0 = OTP 1 = SRAM
DSP_START	Start 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!	

2 DSP & Environment

DSPSTARTONOVL	This setting assures that processor starts upon error condition	0 = default is mandatory
DSP_STARTONTEMP	This setting assures that processor starts upon normal temperature measurement completion. Depends very much on firmware. In standard firmware 03.01.xx temperature values are determined after CDC-determination.	0 = default, mandatory with standard firmware 03.01.xx
DSP_STARTPIN	Pin mask for DSP trigger	0 = FF0 1 = FF1 2 = FF2 3 = FF3
DSP_WATCHDOG_LENGTH	Processor watchdog to be defined in connection with the software developper	0 = off 1 = 512 2 = 1024 3 = 2048
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 setting configuration bit DSP_START.

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 frimware by setting DSP_OUT_x.
- 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 DSPSARTONOVL. It should be used only if error handling is implemented in the software.

2 DSP & Environment

Watchdog

The watchdog is made of a counter based on the DSP clock. Parameter DSP_WATCHDOG_LENGTH defines the maximum number of DSP clock cycles. Within this period the watchdog has to be reset by instruction resetWDG. Otherwise a power-on reset will be triggered.

The watchdog is implemented to handle situations where the software hangs and doesn't serve the awatchdog.

In slave applications the watchdog should be disabled.

System Reset

In case the PCap01 is operated as 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 RUN-BIT 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........ 3-2

3.1 Instructions........ 3-2

3.2 Instruction Details........ 3-12

3.2.1 rad........ 3-12

3.2.2 mult........ 3-13

3.2.3 div........ 3-15

3 Instruction Set

The complete instruction set of the PCap01 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 1-0: Instruction set

Simple Arithmetic	Miscellaneous	Bitwise	RAM access
add	resetWDG	bitC	rad
sign	powerOnReset	bitS	clear
sub	nop		load
	stop		load2exp
			move

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

Conditional jump
jcd
jCarC
jCarS
jEQ
jNE
jNeg
jOflC
jOflS
jPOS

3.1 Instructions

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

3 Instruction Set

bitC	Clear single bit
Syntax:	bitC p1
Parameters:	p1 = number 0 to 15
Calculus:	Bit number p1 of the DSP outputs is cleared
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 p1od 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* (2er), 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

3 Instruction Set

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

3 Instruction Set

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

3 Instruction Set

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 == 0) 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

jOvlS	Jump on Overflow Set
Syntax:	jOvlC p1
Parameters:	p1 = jumplabel
Calculus:	if (O == 1) PC := p1
Flags affected:	-
Bytes:	2

3 Instruction Set

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 == 0) 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 sub-routine with jrt - otherwise there will be no jump back. The stack is decremented by 1.
Category:	Unconditional Jump

3 Instruction Set

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

3 Instruction Set

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

3 Instruction Set

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

powerOnReset	LPower 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 Addess 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 firm-ware. 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. 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

3 Instruction Set

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 = 0 => 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, when the Signum flag (before the execution) is 1. Zero is assumed to be positive.
Category:	Simple arithmetic

3 Instruction Set

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	Substraction
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

3.2 Instruction Details

3.2.1 rad

Sets the RAM address. Typical example:

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

Pointer

rad _at_DPTRØ and rad _at_DPTR1 are special instructions for indirect addressing. _at_DPTRØ /_at_DPTR1 are special RAM addresses that have been defined in the firmware.

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

By means of

```
rad DPTRØ
move r, a
```

an address is loaded into DPTRØ. With

```
rad _at_DPTRØ
```

the address in DPTRØ is loaded.

3 Instruction Set

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.

3 Instruction Set

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 less. 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 less than 48 the result might be spread between accumulators a and b. Doing an apropriate right shift of the multiplicand in r, and the apropriate 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:

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

$'h4B0000000000*2^{-42} = 'h4B*2^{-2} = 75*2^{-2} = 18.75$

3.2.3 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 to 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
- leftshfit 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.

3 Instruction Set

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

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 effective. 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 = $b * 2^{(1+expA-expR-steps)} = 155 * 2^{(1-4+4-8)} = 1.2109$, *Rest* = $a*2^{(\text{-steps})} = 190*2^{-8} = 0.7421$

3 Instruction Set

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 $= b * 2^{(1+expA-expR-steps)} = 2134 * 2^{(1-4+8-12)} = 16.6719$, *Rest* $= a*2^{(-steps)} = 28*2^{-12} = 0.109$

3 Instruction Set

4 Writing Assembly Programs 4-2

4.1 Directives 4-3

4.2 Sample Code 4-4

4.2.1 “for” Loop 4-4

4.2.2 “while” Loop 4-4

4.2.3 “do - while” Loop 4-5

4.2.4 “do - while” with 2 pointers 4-5

4.2.5 Load Negative Values 4-6

4.2.6 Load Signed Values 4-6

4.2.7 Rotate Right A to B 4-6

4 Writing Assembly Programs

The PCap01 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 on 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 wert   1
equal  ((wert + 3)/2)
```

4 Writing Assembly Programs

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 the might be just the file name in case it 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 PCap01\lib.	`#include <rdc.h>` `#include "rdc.h"`

4 Writing Assembly Programs

#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 samle 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
load2exp b, n-1	for(i = 1 << n-1 ; i != 0;	n := number of repetitions
rad index	i =>> 1)	loop body
move r, b	{..}	
do:		loop increment
;{..}		repeat while b != 0
rad index		
move b, r		
sftR b		
move r, b		
jNE do		

4.2.2 "while" Loop

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

4.2.3 “do - while” Loop

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

4.2.4 “do - while” with 2 pointers

Assembler	C-Equivalent	Comment
load a, MW7	loopLimit = *MW7	load max-address for ptrSource
rad loopLimit		
move r, a		
load a, MW0	ptrSource = *MW0;	load ptrSource with source address
rad rad_ext1		
move r, a		
load a, RES0	ptrSink = *Res0;	load ptrSink with sink address
rad rad_ext3		
move r, a		
load2exp b, 0		initialize b with 1
do:	do { *ptrSink++ = *ptrSour-	loop body
rad ext1	ce++ }	load value from source
move a, r		
rad ext3		write value to sink
move r, a		
rad loopLimit		write max-address to a
move a, r		
rad rad_ext3		increment sink address
add r, b		
rad rad_ext1		increment source
add r, b		address
sub a, r		limitLoop – ptrSource
jCarS do		repeat loop if ptrSource <= max-
	while (ptrSource <= MW7)	address

4 Writing Assembly Programs

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	a = -5	a = 5
move b, a		b = 5
sub a, b		a = 0
sub a, b		a = -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	b = <S24bC>	a=2^23
load b, <S24bC>		reg0 = <S24bC>
rad 0		
move r, b		
sub b, a		
jCarC positive		if(<S24bC> >= 2^23)
sub b, a		reg0 = <S24bC> - 2^24
move r, b		
positive:		
move b, r		

4.2.7 Rotate Right A to B

To rotate a value right from Akku A to Akku B, AkkuB 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 to the highest byte in the register.

Assembler	C-Equivalent	Comment
load a, 0xa3	A = <U8bC>	
clear b	b = a << 40	
move r, b		
mult ; (8x)		
mult		
..		
mult		

5 Libraries 5-2

5.1 standard.h 5-3
5.2 pcap01a.h 5-4
5.3 cdc.h 5-4
5.4 rdc.h 5-5
5.5 signed24_to_signed48.h 5-6
5.6 dma.h 5-6
5.7 pulse.h 5-7
5.8 sync.h 5-8
5.9 median.h 5-9

5 Libraries

The PICOCAP 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 fimware 03.01.01 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. These library functions 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 Picocap assembler as part of the standard firmware.

- standard.h
- pcap01a.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 Libraries

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
Temporary memory usage	-
Changes any RAM content permanently?	No

5 Libraries

5.2 pcap01a.h

Function:	This is a standard library for PCap01A firmware projects. This library contains the major address-mappings and constant names for the PCap01A. This file should be always included. It contains no commands so no program 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__ :	0 = single sensor 1 = differential sensor
	__sub_cdc_gain_corr__ :	Factor for TCsg ufd21
	__persistent_cdc_first__ :	Address where CDC results are to be stored
	__temporary_variables__ :	Define address space for temporary variables, address < 39!
Output/Return value:	Capacitance ratios C0_ratio, …, C7_ratio	
Prerequisites	Declare a constant ONE = 1	
Dependency on other header files	#include <standard.h>	

5 Libraries

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 – 8 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

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	Declare a constant ONE = 1
Dependency on other header files	#include <standard.h>
Function call	jsb __sub_rdc__
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 Libraries

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 e.g. 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?	No

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 contents to be copied can be specified.
Input parameters:	Accumulator B : number of values to copy DPTR1 : source RAM block address DPTR0 : 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	Declare a constant ONE = 1
Dependency on other header files	-
Function call	jsb __sub_dma__
Temporary memory usage	-
Changes any RAM content permanently?	Yes, the destination RAM block

5 Libraries

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 : input value, unsigned 21 fractional digits __sub_pulse_slope__ : constant factor, signed 4 fractional digits __sub_pulse_offset__ : constant summand, signed 1 fractional digit __temporary_variables__ : define address space for temporary variables
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	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 : input to be filtered __sub_sync_FilterOrder__ : filter order, depth of filtering __persistent_sync_first__ : address where the filtered results are stored __temporary_variables__ : define address space for temporary variables

5 Libraries

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 : start of filter-memory ringMemLast : last field of the filter memory FilterAkku : sum of all memory-fields currentRingPos : index Pointer; points to the current memory field AkkuDivider : 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. Afterwords the complete list is sorted. The value at the very middle of the list is returend as new median.

5 Libraries

Input parameters:	Accumulator B = input to be filtered __sub_ median _FilterOrder__ : filter order, depth of filtering __persistent_median_first__ : address where the filtered results are stored __temporary_variables__ : define address space for temporary variables
Output/Return value:	The new median is returned in Accumulator B.
Prerequisites	Declare a constant ONE = 1
Dependency on other header files	-
Function call	jsb __sub_median__
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__ : Start of filter memory __sub_median_list_middle__ : middle field of the filter memory __sub_median_list_last__ : last field of the filter memory

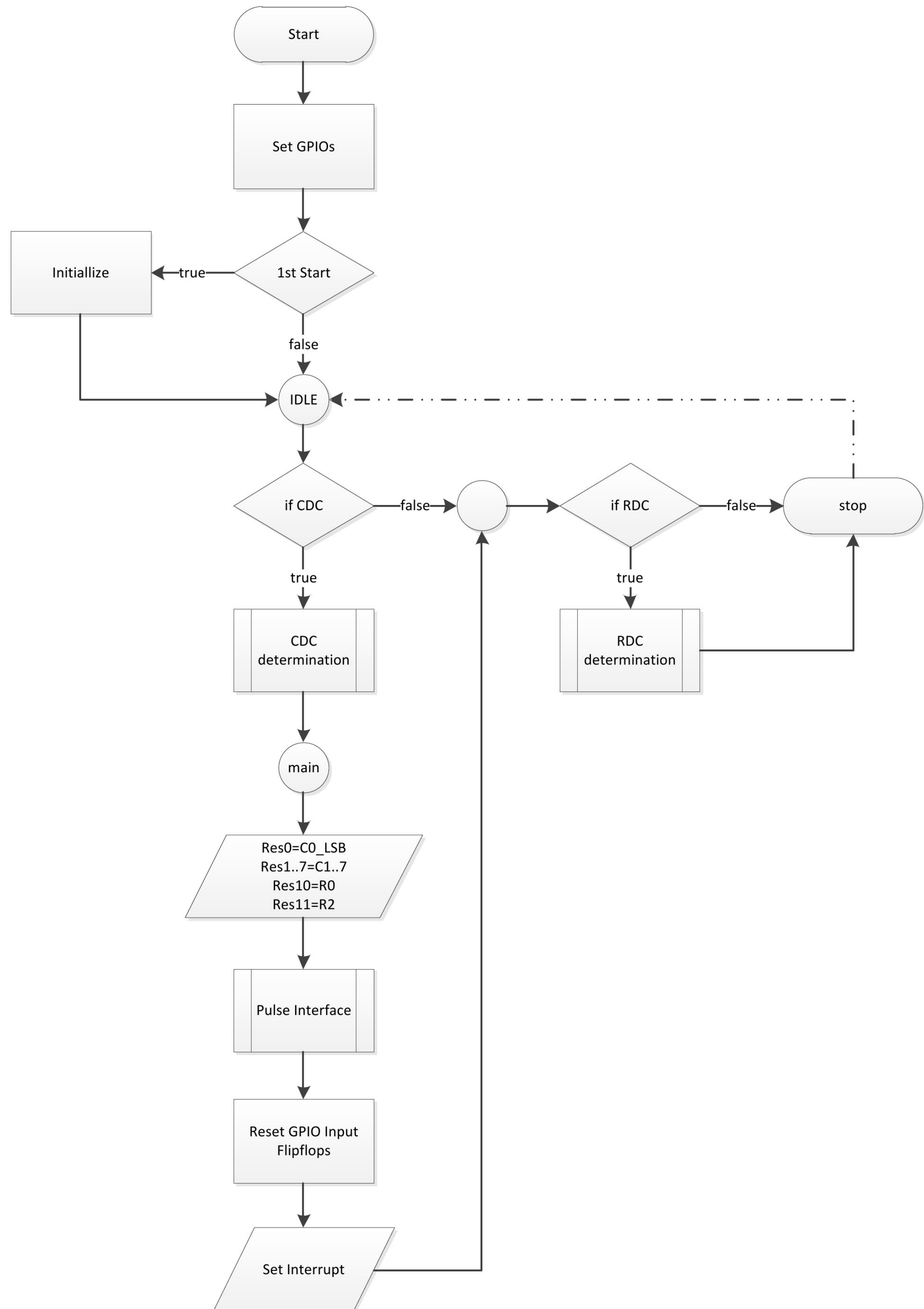
5 Libraries

6 Examples 6-2
6.1 Standard Firmware, Version 03.01.02 6-2

6 Examples

6.1 Standard Firmware, Version 03.01.02

Figure 6-1: Main Loop Flowchart



6 Examples

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_Capacitannce + 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 alternatly 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
```

d) Provide data to read registers

After the subroutines __sub_cdc__ and __sub_cdc__ had 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 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
```

6 Examples

```
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
```

e) Set the pulse interface

The offset and slope of the pulse outputs is typically defined in the parameter registers of PCap01.

```
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
```

6 Examples

```
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 scaled to the 0 to 1023 output of the PCap01 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                      ; PCap01 can output the value at PULSE0 output
```

6 Examples

7 Miscellaneous

7.1 Bug Report

7.2 Document History

01.08.2011 First release

acam
mess • electronic

acam-messelectronic gmbh

Am Hasenbiel 27

76297 Stutensee-Blankenloch

Germany

ph. +49 7244 7419 - 0

fax +49 7244 7419 - 29

e-mail: support@acam.de

www.acam.de