This product, formerly sold by ams AG, and before that optionally by either Applied Sensors GmbH, acam-messelectronic GmbH or Cambridge CMOS Sensors, is now owned and sold by

ScioSense

The technical content of this document under ams / Applied Sensors / acam-messelectronic / Cambridge CMOS Sensors is still valid.

Contact information

Headquarters:

ScioSense B.V.

High Tech Campus 10

5656 AE Eindhoven

The Netherlands

info@sciosense.com

www.sciosense.com

acam

member of the ams group

Data Sheet

TDC-GP30Y

System-Integrated Solution for Ultrasonic Flow Meters
Volume 2: CPU, Memory and Firmware

September 19 , 2019
Document-No: DB_GP30Y_Vol2_en V0.2

acam-messelectronic gmbh is now a member of ams group

Copyrights & Disclaimer

Copyright acam-messelectronic gmbh, Friedrich-List-Str. 4, 76297 Stutensee, Germany-Europe. Trademarks Registered. All rights reserved. The material herein may not be reproduced, adapted, merged, translated, stored, or used without the prior written consent of the copyright owner.

Devices sold by acam-messelectronic gmbh are covered by the warranty and patent indemnification provisions appearing in its General Terms of Trade. acam-messelectronic gmbh makes no warranty, express, statutory, implied, or by description regarding the information set forth herein. acam-messelectronic gmbh reserves the right to change specifications and prices at any time and without notice. Therefore, prior to designing this product into a system, it is necessary to check with acam-messelectronic gmbh for current information. This product is intended for use in commercial applications. Applications requiring extended temperature range, unusual environmental requirements, or high reliability applications, such as military, medical life-support or life-sustaining equipment are specifically not recommended without additional processing by acam-messelectronic gmbh for each application. This product is provided by acam-messelectronic gmbh “AS IS” and any express or implied warranties, including, but not limited to the implied warranties of merchantability and fitness for a particular purpose are disclaimed.

acam-messelectronic gmbh shall not be liable to recipient or any third party for any damages, including but not limited to personal injury, property damage, loss of profits, loss of use, interruption of business or indirect, special, incidental or consequential damages, of any kind, in connection with or arising out of the furnishing, performance or use of the technical data herein. No obligation or liability to recipient or any third party shall arise or flow out of acam-messelectronic gmbh rendering of technical or other services.

“Preliminary” product information describes a product which is not in full production so that full information about the product is not yet available. Therefore, acam-messelectronic gmbh (“acam”) reserves the right to modify this product without notice.

Support / Contact

For direct sales, distributor and sales representative contacts, visit the acam web site at:

www.acam.de www.ams.com

For technical support you can contact the acam support team: support@acam.de

or by phone +49-7244-74190.

Notational Conventions

Throughout the GP30 documentation, the following stile formats are used to support efficient reading and understanding of the documents:

- Hexadecimal numbers are denoted by a leading 0x, e.g. 0xAF = 175 as decimal number. Decimal numbers are given as usual.
- Binary numbers are denoted by a leading 0b, e.g. 0b1101 = 13. The length of a binary number can be given in bit (b) or Byte (B), and the four bytes of a 32b word are denoted B0, B1, B2 and B3 where B0 is the lowest and B3 the highest byte.
- Abbreviations and expressions which have a special or uncommon meaning within the context of GP30 application are listed and shortly explained in the list of abbreviations, see following page. They are written in plain text. Whenever the meaning of an abbreviation or expression is unclear, please refer to the glossary at the end of this document.
- **Variable names** for hard coded registers and flags are in bold. Meaning and location of these variables is explained in the datasheet (see registers CR, SRR and SHR).
- ***Variable names*** which represent memory or code addresses are in bold italics. Many of these addresses have a fixed value inside the ROM code, others may be freely defined by software. Their meaning is explained in the firmware and ROM code description, and their physical addresses can be found in the header files. These variable names are defined by the header files and thus known to the assembler as soon as the header files are included in the assembler source code. Note that different variable names may have the same address, especially temporary variables.
- *Physical variables* are in italics (real times, lengths, flows or temperatures).

Abbrevations

AM	Amplitude measurement
CD	Configuration Data
CPU	Central Processing Unit
CR	Configuration Register
CRC	Cyclic Redundancy Check
DIFTOF, DIFTOF_ALL	Difference of up and down ->TOF
DR	Debug Register
FEP	Frontend Processing
FDB	Frontend data buffer
FHL	First hit level
FW	Firmware, software stored on the chip
FWC	Firmware Code
FWD	Firmware Data
FWD-RAM	Firmware Data memory
GPIO	General purpose input/output
Hit	Stands for a detected wave period
HSO	High speed oscillator
INIT	Initialization process of ->CPU or -> FEP
IO	Input/output
I2C	Inter-Integrated Circuit bus
LSO	Low speed oscillator
MRG	Measurement Rate Generator
NVRAM, NVM	Programmable Non-Volatile Memory
PI	Pulse interface
PP	Post Processing
PWR	Pulse width ratio
R	RAM address pointer of the CPU, can also stand for the addressed register
RAA	Random Access Area
RAM	Random Access Memory
RI	Remote Interface
ROM	Read Only Memory
ROM code	Hard coded routines in ROM
SHR	System Handling Register
SPI	Serial Peripheral Interface
SRAM	Static RAM
SRR	Status & Result Register
SUMTOF	Sum of up and down TOF
Task	Process, job
TDC	Time-to-digital-converter
TOF, TOF_ALL	Time of Flight
TS	Task Sequencer
TM	Temperature measurement
UART	Universal Asynchronous Receiver & Transmitter
USM	Ultrasonic measurement
Vref	Reference voltage
X,Y,Z	Internal registers of the CPU
ZCD	Zero cross detection
ZCL	Zero cross level

For details see the glossary in section 9.

Content

Copyrights & Disclaimer 1-2
1 Introduction 1-1
1.1 CPU & Environment 1-1
2 Program Area 2-1
3 Random Access Area (RAA) 3-1
3.1 RAM 3-2
3.2 Direct mapped register 3-3
3.3 NVRAM 3-3
4 CPU 4-1
4.1 Registers and Accumulators 4-1
4.2 CPU Flags 4-1
4.3 Arithmetic Operations 4-2
4.4 Branch Instructions 4-2
4.5 Instruction Set 4-3
4.6 Detailed Description of Commands 4-4
5 Libraries and pre-defined routines 5-1
5.1 common.h 5-2
6 CPU Handling 6-1
6.1 CPU Handling 6-1
7 Assembler Software 7-1
7.1 Assembly Programs 7-4
7.2 Basic Structure 7-6
7.3 Example 1: Simple TOF Difference via Pulse Interface 7-6
8 Miscellaneous 8-1
8.1 Bug Report 8-1
8.2 Last Changes 8-1

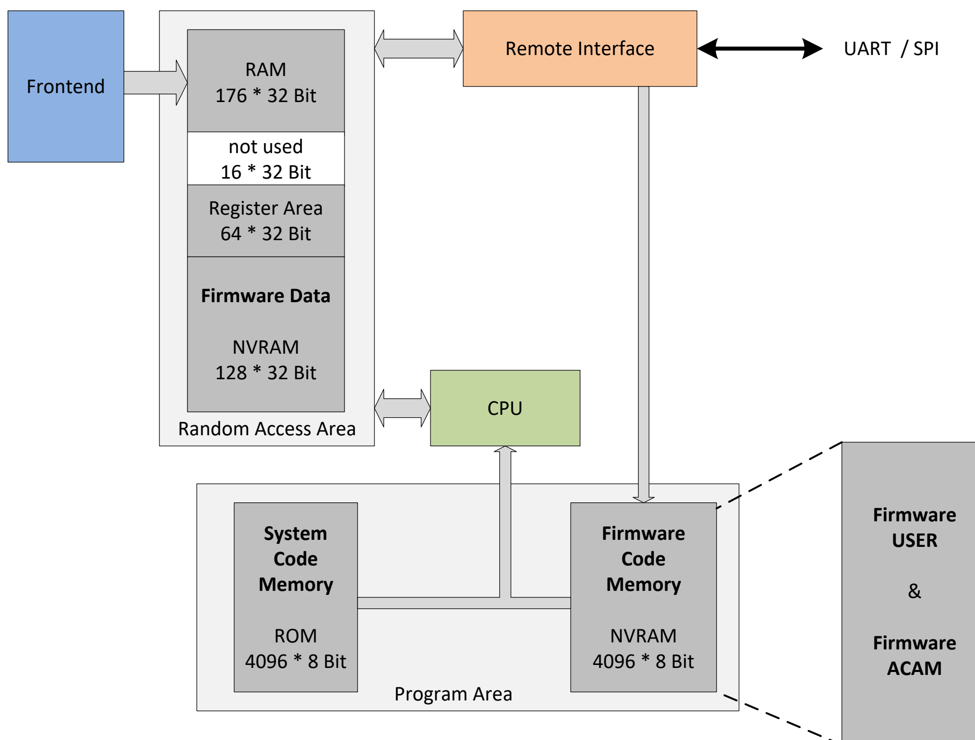
1 Introduction

TDC-GP30 stands for a new generation of ultrasonic flow converters. Besides an improved ToF front end it also includes a 32-bit CPU and memory for on-chip data post-processing and, in a final stage, the complete flow calculation. This volume 2 of TDC-GP30 datasheet describes the CPU, the instruction set and the memory organization. Further, it describes the assembler and basics in software development. This will enable customers to write their own programs. The description of the general hardware and the analog front end basics is given in volume 1.

1.1 CPU & Environment

There is a 32-bit CPU in Harvard architecture integrated in TDC-GP30 which is acam proprietary design. It is optimized for ultra-low power operation with the target to do the flow calculation. Figure 1-1 shows the memory organization and how the frontend, the CPU and the remote interface interact.

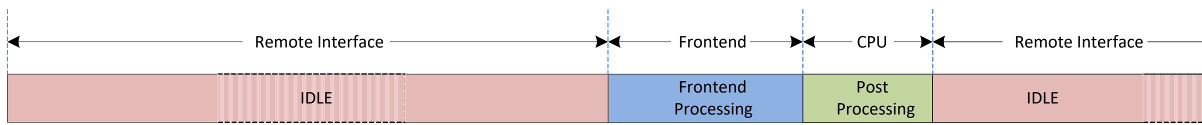
Figure 1-1



After completion of a measurement, the frontend writes the various results for time-of-flight or temperature, for amplitude, pulse width and voltage into the front end data buffer (FDB), which is part of the RAM. From there, the user can read directly the raw data via the remote interface. This would be the situation in time conversion mode.

In the case of flow meter mode, the frontend processing is followed by a CPU processing. The CPU post-processing is activated by setting bit PP_EN in configuration register CR_MRG_TS and CPU_REQ_EN_PP in configuration register CR_IEH.

Figure 1-2 Flow Meter Mode



The programmable firmware will be in a non-volatile 4k NVRAM. Additionally, many functions are already implemented as ROM routines. The CPU uses a separate 176x32 bit RAM to do its calculations and to write back the final results. Configuration data is stored in the register area of the RAM and a special firmware data are in the RAM is reserved for firmware specific data.

The firmware code memory and the firmware data memory are zero static power NVRAMs. It is not necessary to switch them down to save operation current.

The various main elements will be described in detail in the following.

2 Program Area

The program area consists of two memory parts:

- A 4-kbyte NVRAM, called **Firmware Code Memory** for re-programmable program code
- A 4-kbyte ROM, called **System Code Memory** with read-only program code.

The firmware code consists of:

- A USER part which can be programmed by customer (green colored)
- An optional acam part, pre-programmed by acam including general subroutines addressable by customers. For details on this option please contact acam.

The NVRAM in GP30 is a combination of a volatile SRAM and a non-volatile FLASH memory. Access to/from NVRAM is only given via the SRAM part, where volatile data can be read and written in an unlimited number of times, while non-volatile data resides in FLASH part.

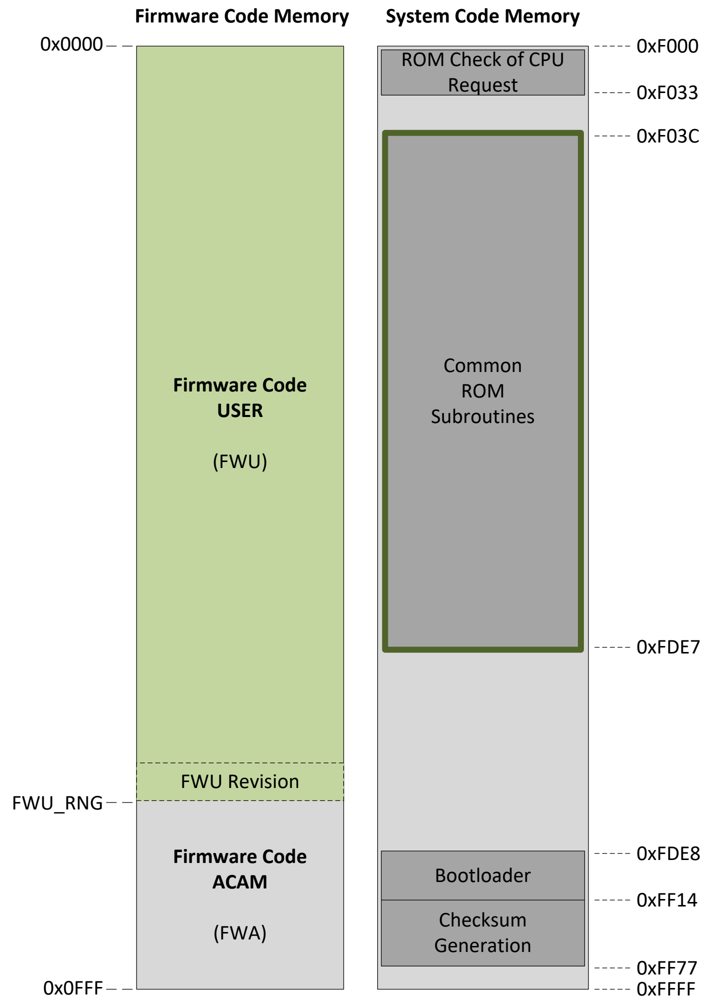
The complete data transfer from SRAM to FLASH is performed by a STORE. From FLASH to SRAM the data are completely transferred by a RECALL. The execution of both transactions has to be enabled first.

The Firmware Code is read protected all the time. Additionally, the GP30 has the ability of a firmware lock, which causes:

- A write protection for Firmware Code
- A read protection for Firmware Data
- A read protection for Configuration Register

The available size of USER Firmware (FWU) is defined in register SRR_FWU_RNG which can be read by customer. The USER firmware has also a 4-byte reserved area at the end of the code memory, which can be used to implement a revision number. The revision can be read via register SRR_FWU_REV. Additionally the revision of ACAM firmware can be read via SRR_FWA_REV.

The code in the ROM memory (read-only) includes system subroutines like bootloader, checksum generation and general subroutines which are also addressable by customer. It also handles the initial check of CPU requests set in SHR_CPU_REQ register.



The bootloader is always requested after a system reset has been occurred. However, bootloader actions are only performed if the bootloader release code is set. In a final initialization the bootloader also sets CPU request for “FW Init” and, if configured, a request for “Checksum Generation”. At minimum the bootloader clears its request in SHR_CPU_REQ and jumps back to ROM Check of CPU Request.

Note: For details about how to write firmware and to check it please see section 7 of the user manual, volume 3.

3 Random Access Area (RAA)

The random access area can be distinguished in 3 sections:

- Random access memory (**RAM**) storing volatile firmware data and including frontend data buffer
- Register area
- Non-volatile RAM (**NVRAM**) storing non-volatile firmware data

The RAA has the following structure:

<table>
<tr><th>IP</th><th>Address</th><th>DWORD</th><th>Section</th><th colspan="2">Description</th><th>RI</th></tr>
<tr><td rowspan="5">RAM
176*32</td><td>0x000–0x07F</td><td>128</td><td>FWV</td><td colspan="2">Firmware variables</td><td>RW</td></tr>
<tr><td>0x080–0x087</td><td>8</td><td>FDB</td><td colspan="2">Frontend data buffer</td><td>RW</td></tr>
<tr><td>0x088–0x09B</td><td>20</td><td>FDB /
(FWV)</td><td colspan="2">Frontend data buffer /
Firmware variables</td><td>RW</td></tr>
<tr><td>0x09C–0x09F</td><td>4</td><td>FWV</td><td colspan="2">Firmware variables</td><td>RW</td></tr>
<tr><td>0x0A0–0x0AF</td><td>16</td><td>FWV or
(TEMP)</td><td colspan="2">Firmware variables or
temporary variables</td><td>RW</td></tr>
<tr><td></td><td>0x0B0–0x0BF</td><td>16</td><td>NU</td><td colspan="2">not used</td><td>-</td></tr>
<tr><td rowspan="6">Direct
Mapped
Register</td><td>0x0C0–0x0CF</td><td>16</td><td>CR</td><td colspan="2">Configuration registers</td><td>RW</td></tr>
<tr><td>0x0D0–0x0DF</td><td>16</td><td>SHR</td><td colspan="2">System Handling registers</td><td>RW</td></tr>
<tr><td>0x0E0–0x0EF</td><td>16</td><td>SRR</td><td colspan="2">Status & result registers</td><td>RO</td></tr>
<tr><td>0x0F0–0x0F7</td><td>8</td><td>NU</td><td colspan="2">not used</td><td>-</td></tr>
<tr><td>0x0F8–0x0FB</td><td>4</td><td>DR</td><td colspan="2">Debug registers</td><td>RO</td></tr>
<tr><td>0x0FC–0x0FF</td><td>4</td><td>NU</td><td colspan="2">Not used</td><td>-</td></tr>
<tr><td rowspan="8">NVRAM
128*32</td><td>0x100–0x11F</td><td>32</td><td>FWD1</td><td colspan="2">Firmware data</td><td>RW</td></tr>
<tr><td>0x120–0x16B</td><td>76</td><td rowspan="3">FWD2</td><td colspan="2">Firmware data</td><td>RW</td></tr>
<tr><td>0x16C–0x17A</td><td>15</td><td>CD</td><td>Configuration data</td><td>RW</td></tr>
<tr><td>0x17B</td><td>1</td><td>BLD_RLS</td><td>Bootloader release code</td><td>RW</td></tr>
<tr><td>0x17C</td><td>1</td><td></td><td colspan="2">FWD1 Checksum</td><td>RW</td></tr>
<tr><td>0x17D</td><td>1</td><td></td><td colspan="2">FWD2 Checksum</td><td>RW</td></tr>
<tr><td>0x17E</td><td>1</td><td></td><td colspan="2">FWU Checksum</td><td>RW</td></tr>
<tr><td>0x17F</td><td>1</td><td>FW_CS</td><td colspan="2">FWA Checksum</td><td>RW</td></tr>
<tr><td></td><td>0x180–0x1FF</td><td>128</td><td>NU</td><td colspan="2">Not used</td><td>-</td></tr>
</table>

RI = Remote Interface

3.1 RAM

The front end data buffer (**FDB**) is the area that contains the measurement data. The content alternates for ToF and temperature measurements:

Table 3-1 FDB after TOF measurements

RAA Address	Name	Description
0x080	FDB_US_TOF_ADD_ALL_U	Ultrasonic TOF Add All Value Up
0x081	FDB_US_PW_U	Ultrasonic Pulse Width Ratio Up
0x082	FDB_US_AM_U	Ultrasonic Amplitude Value Up
0x083	FDB_US_AMC_VH	Ultrasonic Amplitude Calibrate Value High
0x084	FDB_US_TOF_ADD_ALL_D	Ultrasonic TOF Add All Value Down
0x085	FDB_US_PW_D	Ultrasonic Pulse Width Ratio Down
0x086	FDB_US_AM_D	Ultrasonic Amplitude Value Down
0x087	FDB_US_AMC_VL	Ultrasonic Amplitude Calibrate Value Low
0x088 to 0x08F	FDB_US_TOF_0_U	Ultrasonic TOF Up Value 0 to Value 7
0x090 to 0x097	FDB_US_TOF_0_D	Ultrasonic TOF Down Value 0 to Value 7

Table 3-2 FDB after temperature measurements

RAA Address	Name	Description
0x080	FDB_TM_PP_M1	Offset Delay Compensation Value
0x081	FDB_TM_PTR_RAB_M1	PT Ref: Impedance Value
0x082	FDB_TM_PTC_CAB_M1	PT Cold: Impedance Value
0x083	FDB_TM_PTH_HAB_M1	PT Hot: Impedance Value
0x084	FDB_TM_PTR_RA_M1	PT Ref: 1st Offset resistance Value
0x085	FDB_TM_PP_M2	Offset Delay Compensation Value
0x086	FDB_TM_PTR_RAB_M2	PT Ref: Impedance Value
0x087	FDB_TM_PTC_CAB_M2	PT Cold: Impedance Value
0x088	FDB_TM_PTH_HAB_M2	PT Hot: Impedance Value
0x089	FDB_TM_PTR_RA_M2	PT Ref: 1st Offset resistance Value
0x08A	FDB_TM_PTR_4W_RB_M1	PT Ref: 2nd Offset resistance Value
0x08B to 0x08E	FDB_TM_PTC_4W_CA_M1	PT Cold: 1st to 4th Offset resistance Value
0x08F to 0x092	FDB_TM_PTH_4W_HA_M1	PT Hot: 1st to 4th Offset resistance Value
0x093	FDB_TM_PTR_4W_RB_M2	PT Ref: 2nd Offset resistance Value
0x094 to 0x097	FDB_TM_PTC_4W_CA_M2	PT Cold: 1st to 4th Offset resistance Value
0x098 to 0x09B	FDB_TM_PTH_4W_HA_M2	PT Hot: 1st to 4th Offset resistance Value

For details about the **FDB** please refer to volume 1.

The firmware variables area (**FWV**) can be used by the firmware for temporarily data storage.

3.2 Direct mapped register

This section contains the configuration registers **CR** that define the operation of the chip. After a system reset, the content is copied from configuration data (**CD**) in the NVRAM into this RAM cells.

Further, in this section there are the system handling registers as well as the status and result registers.

For debugging purposes also the ALU registers and flags can be read there (read only).

For details about the RAM and the please refer to volume 1.

3.3 NVRAM

This section is a combination of a volatile SRAM and a non-volatile FLASH memory.

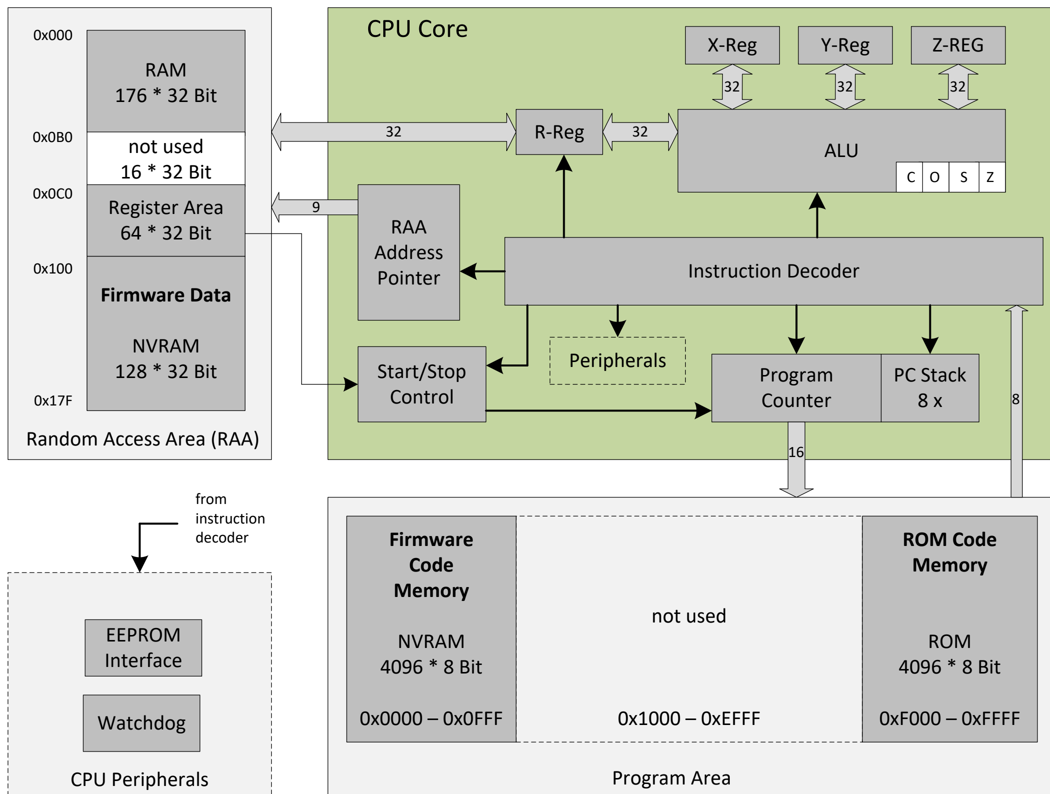
After a system reset, the configuration data (**CD**) in the NVRAM is copied into the configuration register (**CR**) by the bootloader if the bootloader release code (**BLD_RLS** = 0xABCD_7654) is set.

Finally, there are four checksums. Those registers contain the nominal values for the user firmware, the acam firmware and the two firmware data sections. A checksum execution will generate the actual GP30 checksums and compare them with those in **FWD1**, **FWD2**, **FWU** and **FWA**. The checksum execution is done typically after the bootloader, by the checksum timer (register **CR_MRG_TS** : **TS_CST**) or by a remote command. In case of discrepancies error flags will be set for each section (register **CR_IEH**).

4 CPU

Figure 4-1 shows in detail how the CPU is structured and implemented. It has access to the RAM, including the result registers and the status registers.

Figure 4-1



4.1 Registers and Accumulators

The 32 bit-CPU operates on three internal registers, the X, Y, and Z-accumulators, and on one register or RAM cell, addressed by the CPU's RAM address pointer. The latter register is denoted with R, it can be any accessible cell within the **RAA** address range. R is handled in the same way as an accumulator by most commands. One specialty of R is the byte coding and decoding by the bytesel and bytedir command, which only acts on R in read direction (details see below). This function is built in for simplified and accelerated byte operations.

4.2 CPU Flags

The CPU uses four flags to classify the results of operations: Carry (C), equal Zero (Z), Sign (S) and Overflow (O). Zero and Sign flags are set with each CPU write access to any register, RAM or accumulator. Additionally, the Carry and Overflow flags are set in case of a calculation, shift or rotation. Flags which are not actively changed by an operation remain in their former state. It is possible to query each flag in a jump or skip instruction.

4.2.1 Carry (C)

Shows the carry over in an addition or subtraction. Note that the carry flag is calculated assuming unsigned binary numbers, in contrast to the overflow flag. Thus it may produce confusing results, refer to the detail description of instructions for usage. With shift operations (shiftL, shiftR, rotL, shiftR.), the carry flag is set to the (last) bit that has been shifted out.

4.2.2 Overflow (O)

Indicates an overflow during an addition or subtraction of two numbers in two's complement representation. This is strictly an overflow for positive numbers, underflow in case of negative numbers is not indicated. If the eventuality of a negative underflow can't be avoided, additional calculations to indicate the underflow are required.

4.2.3 Zero (Z)

The zero flag indicates if the last number written into a register (by add, sub, move, swap, etc.) was zero or not equal to zero.

4.2.4 Sign (S)

The sign flag indicates if the last number written into a register (by add, sub, move, swap, etc.) has the highest bit (MSB) set to 1 or to 0. It thus indicates the sign of this number, with zero indicated positive. The sign flag assumes a two's complement number representation.

4.3 Arithmetic Operations

An arithmetic command processes two of the registers X, Y, Z or R, and writes back the result into the first mentioned register (or, for commands with 64 bit results, into both). These operations also affect flags of the CPU. In particular, the carry (C) and overflow (O) flags should be checked to ensure correctness of the last operation.

All arithmetic operations process a 32 bit wide input, (mostly) based on the common two's complement operations. This means that the MSB (the most significant bit of the binary word, here bit 31) defines the sign of the binary number, with negative signs having MSB=1. Number values of positive numbers are as usual, while the value of a negative number A follows the rule |A| = NOT(A) +1 , in words: negative numbers are converted into positives by bitwise inversion, and then adding 1 (see the instructions "compl" and "invert")

4.4 Branch Instructions

There are 4 principles of jumping within the code:

Goto: Jumps with relative or absolute addressing. Within an address vicinity of
–128 to +127, the assembler automatically uses relative addressing ("Branch"). For wide distances, absolute addressing within the whole address space of 64 kB is automatically used ("Jump"). The latter is more flexible, but needs one code byte more.

Jsub: Absolute or relative jump, used to call a subroutine. The difference to goto is that the code returns to the calling address at jsubret (for example at the end of the subroutine). The CPU keeps up to 8 calling addresses in its address stack. When no return to the calling address is desired, it is better (and of course possible) to use goto instead of jsub.

Skip: Suppress the execution of the next 1, 2 or 3 instructions. Note that the skipped instructions are in fact processed, but they produce no result or further activity. Thus skip does not save processing time of the skipped instructions, in contrast to goto or jsub. However, the skip command itself is only one byte short, and in addition it is highly suitable for structured programming.

Goto and skip come in different flavors, as unconditional command as well as controlled by some bit or CPU flag. Refer to the detail instruction list below for details.

4.5 Instruction Set

The complete instruction set of the GP30 consists of 70 core instructions that have unique op-codes decoded by the CPU. It is widely identical to the instruction set of acam's PSØ9 chip. Table 4-1 gives an overview of all available expressions, details are given further below.

Table 4-1: Instruction set overview

Logic	**Simple Arithmetic**	**Complex Arithmetic**	**Shift & Rotate**	**Bitwise**
and	abs	div	rotl	bitclr
eor	add	divmod	rotR	bitinv
eorn	compare	mult	shiftL	bitset
invert	compl		shiftR	
nand	decr		**Flags**	**Registerwise**
nor	incr		clrC	clear
or	sign		getflag	move
	sub		setC	swap

RAM access	**Jsub**	**Jump**	**Skip**	**Miscellaneous**
bytedir	jsub	goto	skip	clkmode
bytesel	jsubret	gotoBitC	skipBitC	clrwdt
decramadr		gotoBitS	skipBitS	equal
getramadr		gotoCarC	skipCarC	equal1
incramadr		gotoCarS	skipCarS	i2cclk
ramadr		gotoEQ	skipEQ	i2creq
		gotoNE	skipNE	i2crw
		gotoNeg	skipNeg	mcten
		gotoOvrC	skipOvrC	nop
		gotoOvrS	skipOvrS	revfwa
		gotoPos	skipPos	revfwu
				stop

4.6 Detailed Description of Commands

The following description lists every GP30 instruction which is recognized by the assembler. Most of them directly correspond to an op-code, which is a sequence of bytes in an executable code for GP30, as it is produced by the assembler. The tabular lines have the following meaning and usage:

Command	Short Description
Syntax:	*Command name, followed by parameters p1, p2, p3…*
Parameters:	*Description of parameters. They may be registers REG [x, y, z, r] or numbers in a given range.*
Calculus:	*Mathematical operation in Verilog notation (uncommon syntax is explained in case). This line also defines the result output, which is most of the time simply p1. Note that this means that the content of p1 is changed by the operation.*
Flags affected:	*Some or all of the flags C (carry), Z (Zero), S (sign) and O (Overflow) are affected by the described operation, according to the result*
Bytes:	*Length of the complete op-code, including parameter designation*
Cycles:	*Number of calculation cycles needed by the CPU*
Description:	*Literal description an remarks on the operation*
Category:	*One of the categories in the overview*

There are some more expressions used in the list:

- PC: The program counter; this is actually the code address where the next CPU op-code is read.
- JUMPLABEL: Label for a jump destination, which becomes an actual code address after code assembly. In assembler code, this is usually a placeholder for a position within the code, it may also be a fixed number (not recommended). To define a jump destination by a jump label in assembler code, write the label followed by a colon. Labels must be followed by an instruction. Add one nop if the label would be followed directly by ORG.
- LSB: Least significant bit, the rightmost bit of a binary number
- MSB: Most significant bit, the leftmost bit of a binary number. In the common two's complement representation, the MSB is used to indicate the sign of a number; MSB = 1 defines a negative number.
- “>>” or “<<”: right shift and left shift, e.g. “1<<p2”: a 1 shifted left by p2 bit positions

4.6.1 List of instructions

In the following there is a list of all CPU instructions in alphabetic order.

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

add	**Addition**
Syntax:	add p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 + p2
Flags affected:	C O Z S
Bytes:	1 (p2 = REG) 5 (p2 = number)
Cycles:	1 (p2 = REG) 5 (p2 = number)
Description:	Addition of two registers or addition of a constant to a register
Category:	Simple arithmetic

and	**Logic AND**
Syntax:	and p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 AND p2 *in the resulting bit sequence in p1, a bit is 1 when the corresponding bits of P1 and P2 are both equal to 1*
Flags affected:	Z S
Bytes:	2 (p2 = REG) 6 (p2 = number)
Cycles:	3 (p2 = REG) 7 (p2 = number)
Description:	Bitwise logic AND of 2 registers or Logic AND of register and constant
Category:	Logic

bitclr	**Clear single bit**
Syntax:	bitclr p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = number 0 to 31
Calculus:	p1 = p1 and not (1<<p2) *“1<<p2”: a “1” shifted left by p2 bit positions*
Flags affected:	Z S
Bytes:	2
Cycles:	2
Description:	Clear the single bit on position p2 in the destination register p1, other bits remain unchanged Note: Don’t use on register R in combination with bytesel ≠ 0
Category:	Bitwise

bitinv	**Invert single bit**
Syntax:	bitinv p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = number 0 to 31
Calculus:	p1 = p1 XOR (1<<p2) *“1<<p2”: a “1” shifted left by p2 bit positions*
Flags affected:	Z S
Bytes:	2
Cycles:	2
Description:	Invert the single bit on position 1<<p2 in the destination register p1, other bits remain unchanged Note: Don’t use on register R in combination with bytesel ≠ 0
Category:	Bitwise

bitset	**Set single bit**
Syntax:	bitset p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = number 0 to 31
Calculus:	p1 = p1 OR (1<<p2) *“1<<p2”: a “1” shifted left by p2 bit positions*
Flags affected:	Z S
Bytes:	2
Cycles:	2
Description:	Set the single bit on position p2 in the destination register p1, other bits remain unchanged Note: Don’t use on register R in combination with bytesel ≠ 0
Category:	Bitwise

bytedir	**Define configuration for bytesel**
Syntax:	bytedir p1
Parameters:	p1 = number 0 or 1
Calculus:	-
Flags affected:	-
Bytes:	1

Cycles:	1
Description:	Basic definition for the configuration of bytesel (see description of bytesel) p1 = 0 : align read data at LSB p1 = 1 : shift read byte(s) to various positions **Important remarks**: ▪ Bytedir permanently sets the read configuration until it is changed. ▪ Bytedir is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.
Category:	RAM access

bytesel	**Define RAM reading mode**
Syntax:	bytesel p1
Parameters:	p1 = number 0 to 7
Calculus:	-
Flags affected:	-
Bytes:	1
Cycles:	1
Description:	Read from addressed register R using a byte-oriented shift operation in various configurations. The bytesel command is implemented to simplify bytewise operations, for example read and write from an external EEPROM, or internal selection of coefficients which are shorter than 32 Bit. It actually provides a means for fast bytewise shifting. Denoting the four bytes in the 32-Bit word in R by B3/B2/B1/B0, the following content of R is actually read, depending on the last bytedir setting: After "bytedir 0" (or without using bytedir): p1 = 0 : R is read as B3/B2/B1/B0 (default setting, no shifts) p1 = 1 : R is read as 00/00/B2/B1 p1 = 2 : R is read as 00/00/B1/B0 p1 = 3 : R is read as 00/00/B3/B2 p1 = 4 : R is read as 00/00/00/B0 p1 = 5 : R is read as 00/00/00/B1 p1 = 6 : R is read as 00/00/00/B2 p1 = 7 : R is read as 00/00/00/B3 After "bytedir 1": p1 = 0 : R is read as B3/B2/B1/B0 (default setting, no shifts) p1 = 1 : R is read as 00/B1/B0/00 p1 = 2 : R is read as 00/00/B1/B0 p1 = 3 : R is read as B1/B0/00/00 p1 = 4 : R is read as 00/00/00/B0 p1 = 5 : R is read as 00/00/B0/00 p1 = 6 : R is read as 00/B0/00/00 p1 = 7 : R is read as B0/00/00/00 **Important remarks**: ▪ Bytesel affects the read direction for any register addressed as R. Any read access to R is affected, so the content of R for any operation is configured according to the list above. ▪ Bytesel has no effect in write direction. ▪ Bytesel permanently sets the read configuration until it is changed.

	▪ Bytesel is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not recommended. ▪ Note that the commands bitset, bitclr or bitinv and shiftL, shiftR, rotL and rotR include read access. Set bytesel = 0 before applying one of these commands to R, to avoid undefined results.
Category:	RAM access

clear	**Clear register**
Syntax:	clear p1
Parameters:	p1 = REG [x, y, z, r]
Calculus:	p1 = 0
Flags affected:	Z S
Bytes:	1
Cycles:	1
Description:	Clear addressed register to 0
Category:	Registerwise

clkmode	**Clock mode**
Syntax:	clkmode p1
Parameters:	p1 = number 0 or 1
Calculus:	-
Flags affected:	-
Bytes:	2
Cycles:	2
Description:	p1 = 0 : CPU clock is the internal oscillator p1 = 1 : CPU clock is 2 MHz, derived from the high speed clock (HSC) **Remark**: ▪ clkmode sets the clock mode permanently until the next change or until stop. After stop or after power up, clkmode is 0. ▪ Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.
Category:	Miscellaneous

clrC	**Clear flags**
Syntax:	clrC
Parameters:	-
Calculus:	-
Flags affected:	C O
Bytes:	2
Cycles:	2
Description:	Clear Carry and Overflow flags
Category:	Flags

clrwdt	**Clear watchdog**
Syntax:	clrwdt

Parameters:	-
Calculus:	-
Flags affected:	-
Bytes:	2
Cycles:	
Description:	Clear watchdog. This instruction is used to restart the watchdog timer at the end of a program run. Apply clrwdt right before ‚stop' to avoid a reset by the watchdog, if enabled.
Category:	Miscellaneous

compare	**Compare two values**
Syntax:	compare p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	no register change; only the flags are set to the result of the operation p2 - p1
Flags affected:	C O Z S
Bytes:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Description:	Comparison of the two inputs by subtraction. The flags are changed according to the subtraction result, but not the register contents themselves.
Category:	Simple arithmetic

compl	**Complement**
Syntax:	compl p1
Parameters:	p1 = REG [x, y, z, r]
Calculus:	p1 = - p1 = (NOT p1) + 1
Flags affected:	Z S
Bytes:	2
Cycles:	2
Description:	two's complement of register
Category:	Simple arithmetic

decr	**Decrement**
Syntax:	decr p1
Parameters:	p1 = REG [x, y, z, r]
Calculus:	p1 = p1 – 1
Flags affected:	C O Z S
Bytes:	1
Cycles:	1
Description:	Decrement register by 1
Category:	Simple arithmetic

decramadr	**Decrement RAM address pointer**
Syntax:	decramadr
Parameters:	-
Calculus:	-
Flags affected:	-
Bytes:	1
Cycles:	1
Description:	Decrement RAM address pointer by one
Category:	RAM access

div	**Signed division 32 Bit**
Syntax:	div p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r]
Calculus:	p1 = (p1 << 32) / p2 *"p1<<32": p1 shifted left by 32 bit positions* *or* p1 = p1*232/ p2 *in standard notation* condition for correct calculation: $\lvert p1 \rvert < \lvert 2*p2 \rvert$ In consequence, the result integers in p1 are between $-0.5*2^{32}$ and $0.5*2^{32}$
Flags affected:	Z and S according to the result in p1
Bytes:	2
Cycles:	38
Description:	Signed division of 2 registers: 32 fractional bits of the division of 2 registers are assigned to p1; p2 remains unchanged
Category:	Complex arithmetic

divmod	**Signed modulo division**
Syntax:	divmod p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r]
Calculus:	p1 = integer (p1 / p2) p2 = p1 % p2 *"%" is the modulo operation*
Flags affected:	Z and S according to the result in p1
Bytes:	2
Cycles:	Similar to div
Description:	Signed modulo division of 2 registers, 32 higher bits of the integer division of 2 registers, result is assigned to p1; the remainder is assigned to p2
Category:	Complex arithmetic

eor	**Exclusive OR**
Syntax:	eor p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 XOR p2 *in the resulting bit sequence in p1, a bit is 0 when the corresponding bits of P1 and P2 are equal, or 1 otherwise*
Flags affected:	Z S
Bytes:	2 (p1 = REG, p2 = REG)

	6 (p1 = REG, p2 = number)
Cycles:	3 (p1 = REG, p2 = REG) 7 (p1 = REG, p2 = number)
Description:	Bitwise Logic exclusive OR (antivalence) of the two given parameters
Category:	Logic

eorn	**Exclusive NOR**
Syntax:	eorn p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 XNOR p2 *in the resulting bit sequence in p1, a bit is 1 when the corresponding bits of P1 and P2 are equal, or 0 otherwise*
Flags affected:	Z S
Bytes:	2 (p1 = REG, p2 = REG) 6 (p1 = REG, p2 = number)
Cycles:	3 (p1 = REG, p2 = REG) 7 (p1 = REG, p2 = number)
Description:	Bitwise Logic, exclusive not OR (equivalence) of the two given parameters
Category:	Logic

equal	**Write 3 given Bytes to the executable code**
Syntax:	equal p1
Parameters:	p1 = 3-Byte string
Calculus:	-
Flags affected:	-
Bytes:	3
Cycles:	3 (or more if an executable command was written)
Description:	This instruction is recognized by the assembler. It writes exactly the three bytes given in p1 to the executable code. This can be used to add customized information like version numbers. Handle with care, since the bytes will be interpreted as code when the PC points to them. **Important Note:** Has unpredictable effects under the influence of any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.
Category:	Miscellaneous

equal1	**Write 1 given Bytes to the executable code**
Syntax:	equal1 p1
Parameters:	p1 = Byte string
Calculus:	-
Flags affected:	-
Bytes:	1
Cycles:	1 (or more if an executable command was written)
Description:	This instruction is recognized only by the assembler. It writes exactly the one byte given in p1 to the executable code. This can be used to add customized

	information like version numbers. Handle with care, since the byte will be interpreted as code when the PC points to it. **Important Note:** Has unpredictable effects under the influence of any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.
Category:	Miscellaneous

getflag	**Set S and Z flags**
Syntax:	getflag p1
Parameters:	p1 = REG [x, y, z, r]
Calculus:	Signum flag S is set if p1 < 0 Zero flag Z indicates Zero if p1 = 0
Flags affected:	Z S
Bytes:	1
Cycles:	1
Description:	Set the signum and zero flag according to the addressed register, content of the register is not affected
Category:	Simple arithmetic

getramadr	**Set RAM address pointer to the value in Z**
Syntax:	getramadr
Parameters:	- *The input address is always taken from Z*
Calculus:	RAM address pointer = Z
Flags affected:	-
Bytes:	1
Cycles:	1
Description:	Set the RAM address pointer to the value given in Z
Category:	RAM access

goto	**jump without condition**
Syntax:	goto p1
Parameters:	p1 = JUMPLABEL
Calculus:	PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section “branch instructions”* 3 (absolute jump)
Cycles:	3 (relative jump) *see section “branch instructions”* 4 (absolute jump)
Description:	Jump without condition. Program counter (PC) is set to target address. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoBitC	Jump on bit clear
Syntax:	gotoBitC p1, p2, p3
Parameters:	p1 = REG [x, y, z, r] p2 = number [0...31] p3 = JUMPLABEL or number
Calculus:	if (bit p2 of register p1 == 0) PC = p3 *if ((1<<p2 and p1) == 0)*
Flags affected:	-
Bytes:	2 (relative jump) 3 (absolute jump) *see section "branch instructions"*
Cycles:	3 (relative jump) 4 (absolute jump) *see section "branch instructions"*
Description:	Jump on bit clear. Program counter (PC) is set to target address if selected bit p2 in register p1 is clear. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoBitS	Jump on bit set
Syntax:	gotoBitS p1, p2, p3
Parameters:	p1 = REG [x, y, z, r] p2 = number [0..31] p3 = JUMPLABEL or number
Calculus:	if (bit p2 of register p1 == 1) PC = p3 *if ((2^{p2} AND p1) == 1) …*
Flags affected:	-
Bytes:	2 (relative jump) 3 (absolute jump) *see section "branch instructions"*
Cycles:	3 (relative jump) 4 (absolute jump) *see section "branch instructions"*
Description:	Jump on bit set. Program counter (PC) is set to target address if selected bit p2 in register p1 is set. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoCarC	Jump on carry clear
Syntax:	gotoCarC p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (carry is clear) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) 3 (absolute jump) *see section "branch instructions"*
Cycles:	3 (relative jump) 4 (absolute jump) *see section "branch instructions"*
Description:	Jump on carry clear. Program counter (PC) is set to target address if the last operation that affected the carry (C) flag left it clear.

	The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoCarS	**Jump on carry set**
Syntax:	gotoCarS p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (carry is set) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on carry set. Program counter (PC) is set to target address if the last operation that affected the carry (C) flag left it set. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoEQ	**Jump on equal zero**
Syntax:	gotoEQ p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (Z indicates zero) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on equal zero. Program counter (PC) is set to target address if the last operation that affected the zero (Z) flag indicated a zero result. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoNE	**Jump on not equal zero**
Syntax:	gotoNE p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (Z indicates not-equal zero) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on not-equal zero. Program counter (PC) is set to target address if the last operation that affected the zero (Z) flag indicated a not-equal zero result.

	The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoNeg	**Jump on negative**
Syntax:	gotoNeg p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (S indicates negative) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on negative. Program counter (PC) is set to target address if the last operation that affected the sign (S) flag indicated a result below 0. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoOvrC	**Jump on overflow clear**
Syntax:	gotoOvrC p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (O is clear) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on overflow clear. Program counter (PC) is set to target address if the last operation that affected the overflow (O) flag indicated no overflow. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoOvrS	**Jump on overflow set**
Syntax:	gotoOvrS p1
Parameters:	p1 = JUMPLABEL
Calculus:	if (O is set) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on overflow set. Program counter (PC) is set to target address if the last operation that affected the overflow (O) flag indicated an overflow.

	The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

gotoPos	**Jump on positive**
Syntax:	gotoPos p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	if (S indicates positive) PC = p1
Flags affected:	-
Bytes:	2 (relative jump) *see section "branch instructions"* 3 (absolute jump)
Cycles:	3 (relative jump) *see section "branch instructions"* 4 (absolute jump)
Description:	Jump on positive. Program counter (PC) is set to target address if the last operation that affected the sign (S) flag indicated a result equal or above 0. The target address is given by using a jump label or by an absolute number. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jump

The following three I2C instructions are very basic and listed for the sake of completeness only. The user is asked to access the ready-made ROM routines as described in section 5.1.1.

i2cclk	**I2C clock**
Syntax:	i2cclk
Description:	Triggers the simplified I2C-interface Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.

i2creq	**I2C request**
Syntax:	i2creq p1
Parameters:	p1 = number 0 or 1
Description:	p1 = 0 : clears the I2C request p1 = 1 : sets the I2C request Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.

i2crw	**I2C read / write**
Syntax:	i2crw p1
Parameters:	p1 = number 0 or 1
Description:	p1 = 0 : I2C write p1 = 1 : I2C read Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.

incr	**Increment**
Syntax:	incr p1
Parameters:	p1 = REG [x, y, z, r]

Calculus:	p1 = p1 + 1
Flags affected:	C O Z S
Bytes:	1
Cycles:	1
Description:	Increment register by one
Category:	Simple arithmetic

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

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

jsub	**Unconditional jump to a subroutine**
Syntax:	jsub p1
Parameters:	p1 = JUMPLABEL or number
Calculus:	PC = p1
Flags affected:	-
Bytes:	2 (relative jump) 3 (absolute jump) *see section "branch instructions"*
Cycles:	3 (relative jump) 4 (absolute jump) *see section "branch instructions"*
Description:	Jump to subroutine without condition. The program counter is loaded by the address given through the parameter. The subroutine is processed until the keyword ‚jsubret' occurs. Then a jump back is performed and the next command after the jsub instruction is executed. Jsub needs temporarily a place in the program counter (PC) stack to remember the return address. The PC stack has a depth of 8, so jsub works for up to 8 nested calls. Jump range: 0 to 4095 (Firmware code) and 61440 to 65535 (ROM code)
Category:	Jsub

jsubret	Return from subroutine
Syntax:	jsubret
Parameters:	-
Calculus:	PC = PC after last jsub operation
Flags affected:	-
Bytes:	1
Cycles:	3
Description:	Return from subroutine. A subroutine called via ‚jsub‘ has to be exited by using jsubret. The program is continued at the next command following the calling jsub instruction. The address for continuing is stored in the program counter (PC) stack, which has a depth of 8. This means, the combination jsub-jsubret can be used for up to 8 nested calls.
Category:	Jsub

mcten	Enable / disable measure cycle timer
Syntax:	mcten p1
Parameters:	p1 = number 0 or 1
Calculus:	-
Flags affected:	-
Bytes:	2
Cycles:	2
Description:	p1 = 0 : Measure cycle timer disabled p1 = 1 : Measure cycle timer enabled **Remark:** ▪ Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.
Category:	Miscellaneous

Move	Move
Syntax:	move p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-bit number
Calculus:	p1 = p2
Flags affected:	Z S
Bytes:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Description:	Move content of p2 to p1 (p1 = REG, p2 = REG) Move constant to p1 (p1 = REG, p2 = number)
Category:	Registerwise

mult	Signed 32-Bit multiplication
Syntax:	mult p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r]

Calculus:	p1, p2 = p1 * p2 *the 32-bit numbers p1 and p2 are multiplied to a 64-bit result witch is stored in p1 (upper 32 bits and sign) and p2 (lower 32 bits)*
Flags affected:	Z and S according to p1
Bytes:	2
Cycles:	38
Description:	Signed multiplication of two registers. Higher 32 bits of the multiplication result are placed to p1; lower 32 bits of the multiplication result are placed to p2. Note that the sign of the whole number is defined through the MSB of p1, while the MSB of p2 is just bit 31 of the result (p2 is unsigned). This can lead to misinterpretation by subsequent operations which assume signed numbers.
Category:	Complex arithmetic

nand	**Logic NAND**
Syntax:	nand p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 NAND p2 *not (p1 AND p2); in the resulting bit sequence in p1, a bit is 0 when the corresponding bits of P1 and P2 are both equal to 1, otherwise the bit is 1*
Flags affected:	Z S
Bytes:	2 (p1 = REG, p2 = REG) 6 (p1 = REG, p2 = number)
Cycles:	3 (p1 = REG, p2 = REG) 7 (p1 = REG, p2 = number)
Description:	Bitwise logic NAND (negated AND) of the two input parameters
Category:	Logic

nop	**No operation**
Syntax:	nop
Parameters:	-
Calculus:	-
Flags affected:	-
Bytes:	1
Cycles:	1
Description:	Placeholder code or timing adjust, no operation. May be needed sometimes to separate two code bytes to prevent an assembler error message.
Category:	Miscellaneous

nor	**Logic NOR**
Syntax:	nor p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 NOR p2 *p1 = not (p1 OR p2); in the resulting bit sequence in p1, a bit is 1 when the corresponding bits of P1 and P2 are both equal to 0, otherwise the bit is 0*
Flags affected:	Z S

Bytes:	2 (p1 = REG, p2 = REG) 6 (p1 = REG, p2 = number)
Cycles:	3 (p1 = REG, p2 = REG) 7 (p1 = REG, p2 = number)
Description:	Bitwise logic NOR (negated OR) of the two input parameters
Category:	Logic

or	**Logic OR**
Syntax:	or p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p1 OR p2 *in the resulting bit sequence in p1, a bit is 0 when the corresponding bits of P1 and P2 are both equal to 0, otherwise 1*
Flags affected:	Z S
Bytes:	2 (p1 = REG, p2 = REG) 6 (p1 = REG, p2 = number)
Cycles:	3 (p1 = REG, p2 = REG) 7 (p1 = REG, p2 = number)
Description:	Bitwise logic OR of the two input parameters
Category:	Logic

ramadr	**Set RAM address pointer**
Syntax:	ramadr p1
Parameters:	p1 = RAM cell name or 9-Bit number
Calculus:	-
Flags affected:	-
Bytes:	1 (address range 0x000 to 0x03F) 2 (any address > 0x03F)
Cycles:	1 (address range 0x000 to 0x03F) 2 (any address > 0x03F)
Description:	Set pointer to RAM address (range: 0...511)
Category:	RAM access

The following two routines are low-level and listed for the sake of completeness only. The user is asked to access the ready-made ROM routines as described in section 5.1.1

revfwa	**Store acam firmware revision number**
Syntax:	revfwa
Description:	Stores content of X-Register to SRR_FWA_REV Used by FW ROM Code when bootloader is executed Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.

revfwu	**Store user firmware revision number**
Syntax:	revfwu
Description:	Stores content of X-Register to SRR_FWU_REV Used by FW ROM Code when bootloader is executed

	Is not affected by any conditional or unconditional skip command. Usage within the range of any skip command is not permitted.

rotL	**Rotate left**
Syntax:	rotL p1(, p2)
Parameters:	p1 = REG [x, y, z, r] p2 = no entry or number 2..15
Calculus:	case rotL p1, without p2: p1 = (p1 << 1) + carry ; carry = MSB(p1) p1 = 2*p1 + carry ; carry = MSB(p1) case rotL p1, p2: p1 = repeat (p2 times) rotL p1 *Adding carry finally lets the bits of p1 circulate left over 1 or p2 positions.*
Flag affected:	C O (resulting from the last rot step), Z S (according to the final result in p1)
Bytes:	1 (p1 = REG, p2 = none) 2 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = none) 1 + p2 (p1 = REG, p2 = number)
Description:	Without p2 or p2 = 1 : Rotate p1 left by one bit position over carry. This means in detail, shift p1 register to the left, fill LSB with present carry, then move the former MSB to carry. With 2 ≤ p2 ≤ 15 : Rotate p1 left by p2 bit positions over carry. This means in detail, shift p1 register p2 times to the left, in each step fill LSB with the present carry and then move the former MSB to carry. Note: Don't use on register R in combination with bytesel ≠ 0
Category:	Shift and rotate

rotR	**Rotate right**
Syntax:	rotR p1(, p2)
Parameters:	p1 = REG [x, y, z, r] p2 = no entry or number 2..15
Calculus:	case rotR p1, without p2: p1 = (p1 >> 1) + (carry << 31) ; carry = LSB(p1) → Carry is shifted left to position 31, or p1 = integer (p1 / 2) + (carry*2^{31}) ; carry = LSB(p1) case rotR p1, p2: p1 = repeat (p2 times) rotL p1 *Placing carry at MSB lets the bits of p1 circulate right over 1 or p2 positions.*
Flags affected:	C O (resulting from the last rot step), Z S (according to the final result in p1)
Bytes:	1 (p1 = REG, p2 = none) 2 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = none) 1 + p2 (p1 = REG, p2 = number)
Description:	Without p2 or p2 = 1 : Rotate p1 right by one bit position over carry. This means in detail, shift p1 register to the right, fill MSB with present carry, then move the former LSB to carry. With 2 ≤ p2 ≤ 15 : Rotate p1 right by p2 bit positions over carry. This means in detail, shift p1 register p2 times to the right, in each step fill MSB with the present carry and then move the former LSB to carry. Note: Don't use on register R in combination with bytesel ≠ 0

Category:	Shift and rotate

setC	**Set carry flag**
Syntax:	setC
Parameters:	-
Calculus:	-
Flags affected:	C O
Bytes:	2
Cycles:	2
Description:	Set carry flag and clear overflow flag
Category:	Flags

shiftL	**Shift Left**
Syntax:	shiftL p1(, p2)
Parameters:	p1 = REG [x, y, z, r] p2 = no entry or number 2..15
Calculus:	case shiftL p1, without p2: p1 = (p1 << 1); carry = MSB(p1) *“p1 << 1”: p1 shifted left by 1 bit, actually means p1 multiplied by 2* *in standard notation: p1 = 2 *p1 as long as MSB remains unchanged* case shiftL p1, p2: p1 = repeat (p2 times) shiftL p1 *in standard notation: p1 = p1*2p2 as long as MSB remains unchanged*
Flags affected:	C O (resulting from the last shift step), Z S (according to the final result in p1)
Bytes:	1 (p1 = REG, p2 = none) 2 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = none) 1 + p2 (p1 = REG, p2 = number)
Description:	Without p2 or p2 = 1 : Unsigned Shift p1 left by one bit position, LSB set to zero, MSB shifted out to carry. Note that this can cause fake sign changes. With 2 ≤ p2 ≤ 15 : Unsigned Shift p1 left by p2 bit positions, b2 lower bits set to zero, MSB of last step shifted out to carry. Note that this operation can cause fake sign changes. Check by OVL flag Note: Don’t use on register R in combination with bytesel ≠ 0
Category:	Shift and rotate

shiftR	**Shift right**
Syntax:	shiftR p1(, p2)
Parameters:	p1 = REG [x, y, z, r] p2 = no entry or number 2..15
Calculus:	case shiftR p1, without p2: p1 = (p1 >> 1); carry = LSB(p1) *“p1 >> 1”: p1 shifted right by 1 bit, actually means p1 divided by 2* *in standard notation:* p1 = p1 /2 with a truncation error if LSB(p1)=1 case shiftR p1, p2: p1 = repeat (p2 times) shiftR p1 *in standard notation:* p1 = p1/2p2 *with some truncation error due to lost lower bits*
Flags affected:	C O (resulting from the last shift step), Z S (according to the final result in p1)
Bytes:	1 (p1 = REG, p2 = none)

	2 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = none) 1 + p2 (p1 = REG, p2 = number)
Description:	Without p2 or p2 = 1 : Signed Shift p1 right by one bit position, MSB duplicated to keep sign unchanged, LSB shifted out to carry. The latter can be used to correct a possible truncation error. With 2 ≤ p2 ≤ 15 : Signed Shift p1 right by p2 bit positions, p2 leading bits set to initial MSB to keep sign unchanged. Carry is set to the last LSB shifted out, which can be used to reduce a possible truncation error. Note: Don't use on register R in combination with bytesel ≠ 0
Category:	Shift and rotate

sign	**Sign**
Syntax:	sign p1
Parameters:	p1 = REG [x, y, z, r]
Calculus:	p1 = 1 = 0x00000001 if p1 >= 0 p1 = -1 = 0xFFFFFFFF if p1 < 0
Flags affected:	Z S
Bytes:	2
Cycles:	2
Description:	Sign of addressed register in complement of two notations. A positive value returns 1, a negative value returns -1 Zero is assumed to be positive
Category:	Simple arithmetic

skip	**Skip**
Syntax:	skip p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 instructions without conditions. The one, two or three active instructions following the skip command produce no result, except some instructions that may not be skipped (see below). Note that the skipped instructions are processed, but they produce no result or further activity. Use the skip commands (conditional or unconditional) for structured programming or to ignore very short code sequences – for long sequences goto is more effective. **Note**: The following instructions may not be skipped: bitclr, bitinv, bitset, bytedir, bytesel, equal, equal1, i2cclk, i2creq, i2crw, revfwa, revfwu, clkmode, mcten
Category:	Skip

skipBitC	**Skip on bit clear**
Syntax:	skipBitC p1, p2,p3
Parameters:	p1 = REG [x, y, z, r]

	p2 = number [0..31] p3 = number [1, 2, 3]
Calculus:	if (bit p2 of register p1 == 0) PC = PC + code bytes of next p3 instructions
Flags affected:	-
Bytes:	2
Cycles:	2 + cycles of the skipped commands
Description:	Skip p3 commands if bit p2 of register p1 is clear. See “skip” for more details.
Category:	Skip

skipBitS	**Skip on bit set**
Syntax:	skipBitS p1, p2,p3
Parameters:	p1 = REG [x, y, z, r] p2 = number[0..31] p3 = number[1, 2, 3]
Calculus:	if (bit p2 of register p1 == 1) PC = PC + code bytes of next p3 instructions
Flags affected:	-
Bytes:	2
Cycles:	2 + cycles of the skipped commands
Description:	Skip p3 commands if bit p2 of register p1 is set. See “skip” for more details.
Category:	Skip

skipCarC	**Skip carry clear**
Syntax:	skipCarC p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (carry == 0) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if carry clear. See “skip” for more details.
Category:	Skip

skipCarS	**Skip carry set**
Syntax:	skipCarS p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (carry == 1) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if carry set. See “skip” for more details.
Category:	Skip

skipEQ	Skip on zero
Syntax:	skipEQ p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (Z indicates zero) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if result of previous operation is equal to zero. See “skip” for more details.
Category:	Skip

skipNE	Skip on non-zero
Syntax:	skipNE p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (Z indicates not-equal zero) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if result of previous operation is not equal to zero. See “skip” for more details.
Category:	Skip

skipNeg	Skip on negative
Syntax:	skipNeg p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (S indicates negative) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if result of previous operation was smaller than 0. See “skip” for more details.
Category:	Skip

skipOvrC	Skip on overflow clear
Syntax:	skipOvrC p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (O is clear) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if overflow is clear. See “skip” for more details.

Category:	Skip

skipOvrS	**Skip on overflow set**
Syntax:	skipOvrS p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (O is set) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if overflow is set. See “skip” for more details.
Category:	Skip

skipPos	**Skip on positive**
Syntax:	skipPos p1
Parameters:	p1 = number [1, 2, 3]
Calculus:	if (S indicates positive) PC = PC + code bytes of next p1 instructions
Flags affected:	-
Bytes:	1
Cycles:	1 + cycles of the skipped commands
Description:	Skip p1 commands if result of previous operation was greater or equal to 0. See “skip” for more details.
Category:	Skip

stop	**Stop**
Syntax:	stop
Parameters:	-
Calculus:	-
Flags affected:	-
Bytes:	1
Cycles:	1
Description:	The CPU and the CPU clock are stopped. Usually this instruction is the last command in the assembler listing, it ends any CPU activity. New activity starts by request of the task sequencer or over external communication. Note that the request flag that started the CPU activity must be cleared by the CPU before stop, to indicate that this request was processed.
Category:	Miscellaneous

sub	**Substraction**
Syntax:	sub p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r] or 32-Bit number
Calculus:	p1 = p2 – p1
Flags affected:	C O Z S

Bytes:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Cycles:	1 (p1 = REG, p2 = REG) 5 (p1 = REG, p2 = number)
Description:	Subtraction of the two parameters
Category:	Simple arithmetic

swap	**Swap**
Syntax:	swap p1, p2
Parameters:	p1 = REG [x, y, z, r] p2 = REG [x, y, z, r]
Calculus:	p1 = p2 and p2 = p1
Flags affected:	-
Bytes:	1
Cycles:	3
Description:	Swap of 2 registers. The value of two registers is exchanged between each other.
Category:	Registerwise

5 Libraries and pre-defined routines

GP30Y comes with a number of predefined routines in its ROM. Some of them are ready-to-use and freely available. The ROM routines are organized in a library, defined by a so called header file which relates routine and variable names to their call addresses and memory addresses, respectively:

- common.h General purpose routines

File “common.h” that comes with the assembler must be included in codes that use any of these routines (use the “include” statement in the main *.asm file). The routines are called using their ROM routine name after jsub or any goto statement. The ROM routine name is a synonym of the call address, as defined in the header file. The call address may be used alternatively.

Some routines come in different alternative versions or with alternative start addresses. To some extent, this allows the user to select different RAM cells for data storage. A typical example would be a routine which needs some cells of usual RAM, and an alternative version where cells in the firmware data (FWD) range are used instead – this second one frees up RAM space and could make use of automated non-volatile storage, at the cost of firmware data space. Another reason for alternative start addresses is to skip a part of the routine if some part of the preparation work is not needed or undesired (for example when some numbers calculated at the start of the routine are already known). The differences between the versions are explained for each routine in detail in the subsequent sections.

Number format: As usual in fixed decimal-point arithmetic, care has to be taken to set values in the right format. Unless differently noted, all numbers are in **two’s complement** (MSB determines sign). The binary representation B_bin of a fractional number is defined with a fixed number N of fractional binary digits, such that the corresponding decimal number B_dec is calculated as: B_dec = Bin_into_decimal(B_bin)/2^N. Throughout this document, such a format will be labeled “**fd N” – N fractional digits**. A typical value format is fd 16, covering a fractional number range from about -32768.0 to 32768.0 (when using 32 bit RAM cells).
The second factor to be considered in calculations is the unit, which in many cases comes with a fixed factor, for example whenever values are related to a particular physical value. A typical example is measured TOF time, which is always given as fd 16 in HSC periods (250 ns for 4 MHz operation). This means, the measured Time-of-flight value in time units TOF relates to the measured number TOF_bin as: TOF = (Bin_into_decimal(TOF_bin)/2^16) * 250 ns. Another example is the first hit level FHL, which is given as an integer binary number FHL_bin with an LSB of about 0.88mV: FHL = Bin_into_decimal(FHL_bin) * 0.88 mV.

Due to the internal calculation processes, the range of values which generate correct results in some calculation is limited and depends on the format definition. For example, a multiplication of two 32 bit numbers always generates a correct 2*32 bit result (in two words, Y and X register). But if this result is formatted into one single word in fd 16 format (for example using ROM_FORMAT_64_to_32BIT)

for further calculations, the result can only be right when the leading 16 bit of the original result where 0 (and of course, some accuracy is lost by cutting the lowest 16 bit, too). Such effects have to be considered in any routine that deals with actual calculations. Wherever applicable, number formats and additional range limitations are given in the subsequent routine descriptions.

5.1 common.h

The general purpose routines defined in common.h are listed in the following. Note that not all of them can be used in the same code, depending on memory allocation. Some routines are included in alternative versions, to enable optimized memory usage. In the following sections, the ROM routines defined in common.h are grouped according to their usage. The table gives an overview:

Table 5-1 ROM-routines for common usage

Name	Description	Remarks
Filtering		
ROM_INIT_FILTER	Routine to initialize the RAM cells for any filter (rolling average or median) with a given value	
ROM_ROLL_AVG	Routine to filter the ***FILTER_IN*** values using a rolling average filter	Filter length can be configured
ROM_ROLLAVG_OUTLIER	Routine to filter the ***FILTER_IN*** values using a rolling average filter. One value which deviates most is always ignored.	Filter length can be configured
ROM_MEDIAN	Routine to filter the ***FILTER_IN*** values using a median filter	Filter length can be configured
ROM_FILTER_FLOW	Routine to filter flow values used by acam firmware	Filter length is fixed N=16
Error detection and handling		
ROM_EH	This routine checks all error flags and suppresses processing of wrong results.	many RAM cells fixed
ROM_PP_AM_MON	Monitor the amplitude values and check limits to identify bad measurements	alternative calls exist
ROM_PP_AM_CALIB	This routine gets the Amplitude Calibration values (H & L) and evaluates the gradient and offset that can be used for calculating the actual amplitude.	alternative calls exist
Pulse interface and flow volume		
ROM_CFG_PULSE_IF	This routine configures the pulse interface with the parameters calculated from the given configuration.	
ROM_PI_UPD	Pulse Interface Update Routine	
ROM_PP_PI_UPD	Pulse Interface Update Routine with input from RAM	
ROM_RECFG_PULSEIF_FOROMR_ERROR1	Reconfiguring the pulse interface outputs GPIO0 and GPIO1 as normal GPIOs to signal an error	
ROM_SIGNAL_ERROR_ON_PULSEIF1	Signaling error on the pulse interface GPIO0 and GPIO1	
ROM_RECFG_PULSEIF_FOR_PULSE1	Configuring GPIO0 and GPIO1 as pulse interface outputs	

ROM_SAVE_FLOW_VOLUME	This routine is used to store the converted flow (in LPH), cumulatively to flow volume in cubic meter.	alternative versions exist
Sensor temperature measurement		
ROM_TEMP_POLYNOM	Calculates the temperature of a PT sensor using a polynomial approximation	
ROM_TEMP_LINEAR_FN	This routine is used to calculate the temperature of any sensor as a linear function of sensor resistance using the nominal resistance and sensor slope.	
ROM_TM_SUM_RESULT	Sums up the results of double temperature measurements. The double measurements are performed to eliminate the 50/60 Hz disturbance.	
Interface communication		
ROM_I2C_ST	I2C Start Byte Transfer	Low-level routines, covered bythe ones following
ROM_I2C_BT	I2C Byte Transfer	
ROM_I2C_LT	I2C Last Byte Transfer	
ROM_I2C_DWORD_WR	Write 4 bytes of data to a specified address through the I2C interface	
ROM_I2C_BYTE_WR	Write a single byte of data to a specified address through the I2C interface	
ROM_I2C_DWORD_RD	Sequentially read 4 data bytes from the I2C interface	
ROM_I2C_BYTE_RD	Sequentially read a single data byte from the I2C interface	
ROM_COPY_UART_PRB_DATA	This routine is used to copy the relevant data for the UART Master into the Probe data area. The data is then sent over UART interface to the master.	alternative calls for optimized memory
Housekeeping		
ROM_CPU_CHK	Check kind of CPU request: This routine is called by hardware design after any Post Processing (PP) request, it is the starting point of any CPU activity, including the firmware call at **MK_CPU_REQ**.	automati-cally started
ROM_USER_RAM_INIT	Initialize the entire user RAM with 0	
High speed oscillator		
ROM_HSO_WAIT_SETTL_TIME	This routine is used to switch on the High speed oscillator clock and wait out its settling time (122 us)	
ROM_HSC_CALIB	This routine evaluates the high speed clock scaling factor for the 4MHz / 8 MHz clock	
ROM_SCALE_WITH_HSC	Routine to scale the input parameter with the HS Clock Calibration factor	
Configuration		
ROM_RESTORE_TOF_RATE	Routine to reconfigure TOF_RATE generator from a lower rate after ZERO FLOW ends	alternative version for free configuration
ROM_RECFG_TOF_RATE	Routine to reconfigure TOF_RATE generator to a lower rate, depending on the parameter N	
Mathematics		

ROM_FORMAT_64_TO_32BIT	Routine to format a 64-bit value (in Y and X) into a 32 bit result with 16 integer + 16 fractional bits. Useful for formatting 64 bit multiplication results with 32 integer + 32 fractional bits	alternative version: faster, but needs temporary RAM
ROM_DIV_BY_SHIFT	Perform the division of a value Y by X, where X=2^N is an integer power of two	
ROM_SQRT	Evaluate the square root accurately for values in the range (196 <= X <= 5476)	
ROM_LINEAR_CORRECTION	Linear interpolation of a coefficient between two sampling points	alternative version is fixed to interpolation over THETA
ROM_FIND_SLOPE	Used to find the slope between two points, given the coefficient values and parameter values at the two points.	

5.1.1 ROM Routines in Detail

Data Filtering:

ROM routine name	***ROM_INIT_FILTER / ROM_INIT_FILTER1***
Description	Routine to initialize the RAM cells for any block of RAM cells of size N and starting at a given address with a given value (use to initialize rolling average or median filter). The routine has an alternative start address ***ROM_INIT_FILTER1***, where ***RAM_R_V32_FILTER*** remains unchanged, but Y returns undefined.
Prerequisite	-
Input parameters / register values	**X**: contains value to be initialised in the RAM cells (any format) **Y**: Number of RAM cells to be initialised N (integer) **Z**: Starting RAM Address of the filter
Output/Return value	NVRAM cells starting at the address in Z are initialised with the value in X
Temporary RAM	***RAM_R_V32_FILTER*** (remains unchanged when using ***ROM_INIT_FILTER1***)
Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Z**; (**Y** unchanged when using ***ROM_INIT_FILTER***)

ROM routine name	***ROM_ROLL_AVG***

Description	This routine averages a list of the last N ***FILTER_IN*** values using a rolling average filter. With every call it removes the oldest value from the list end and adds the new one at the beginning. Then it determines the new average by calculating the arithmetic mean from the N list values. The final output value must not exceed the maximal representable number/N in the chosen format (else overflow occurs).
Prerequisite	For correct results, all N RAM cells of the filter must contain valid values. Use ROM_INIT_FILTER once before first routine call for proper initialization.
Input parameters / register values	**X**: new value to be added to the filter list (***FILTER_IN***) (any format) **Y**: filter length N (integer) **Z**: Starting address of the rolling average filter RAM cell block of length N
Output/Return value	**X**: new average (same format as X input)
Temporary RAM	RAM_R_V32_FILTER
Permanent RAM	-
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_ROLLAVG_OUTLIER***
Description	This routine averages a list of the last N FILTER_IN values using a rolling average filter. With every call it removes the oldest value from the list end and adds the new one at the beginning. Then it determines the new average by calculating the arithmetic mean from the N list values – except the one value that deviates the most from the last average (the OUTLIER). This one value is replaced by the previous one (ONLY for the calculation, the original value remains in the list), in order to filter out single error points. The final output value must not exceed the maximal representable number/N in the chosen format (else overflow occurs).
Prerequisite	For correct results, all N RAM cells of the filter must contain valid values. Use ***ROM_INIT_FILTER*** once before first routine call for proper initialization.
Input parameters / register values	**X**: new value to be added to the filter list (***FILTER_IN***) (any format) **Y**: filter length N (integer) **Z**: Starting address of the rolling average filter RAM cell block of length N ***RAM_R_V30_PREV_AVG*** with the previous averaged result
Output/Return value	**X**: new average (same format as **X** input) **Z**: last valid value that replaced the OUTLIER (same format as **X** input) Bit **BNR_NEW_VAL_IS_OUTLIER** in ***RAM_R_FW_STATUS*** is set if the current new value is then OUTLIER – to be recognized for later error handling, for example to replace the new value by Z in other calculations..
Temporary RAM	***RAM_R_V32_FILTER, RAM_R_V31_FILTER_2, RAM_R_V33_FILTER_SUM***
Permanent RAM	***RAM_R_FW_STATUS, RAM_R_V30_PREV_AVG***
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_MEDIAN***
Description	This routine calculates the median of a list of the last N ***FILTER_IN*** values. With every call it removes the most different value from the ordered list and sorts the new one. Then it calculates the new median.
Prerequisite	For correct results, all N RAM cells of the filter must contain valid values. Use ***ROM_INIT_FILTER*** once before first routine call for proper initialization.
Input parameters / register values	**X**: new value to be added to the filter list (FILTER_IN) (any format) **Y**: filter length N (integer) **Z**: Starting address of median filter RAM cell block of length N
Output/Return value	**X**: new median (same format as **X** input)
Temporary RAM	***RAM_R_V32_FILTER, RAM_R_V31_FILTER_2***
Permanent RAM	-
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_FILTER_FLOW***
Description	Routine to filter the flow value with a rolling average filter of length 16. The routine initializes the filter at its first call with its input value, and calculates with each new call a new averaged flow value with the oldest value replaced by the new input from RAM_R_VA2_FLOW_LPH_TO_FLT. The final output value must not exceed the maximal representable number/16 in the chosen format (else overflow occurs). Used by ams firmware only.
Prerequisite	All 16 filter cells in RAM, starting at ***RAM_R_ROLAVG_1***, must still contain the former values.
Input parameters / register values	***RAM_R_VA2_FLOW_LPH_TO_FLT*** (any format)
Output/Return value	**X**: averaged flow value (same format as input) Values in 16 filter cells in RAM, starting at ***RAM_R_ROLAVG_1***, are shifted by one cell, dropping the oldest value at the end and storing the new input value in ***RAM_R_ROLAVG_1*** First call: all 16 filter cells get initialized to ***RAM_R_VA2_FLOW_LPH_TO_FLT***, and Bit **BNR_FLOW_FILT_INIT_DONE** in ***RAM_R_FW_STATUS*** is set.
Temporary RAM	-
Permanent RAM	***RAM_R_FW_STATUS, RAM_R_VA2_FLOW_LPH_TO_FLT, RAM_R_ROLAVG_1, RAM_ ROLAVG_2 .. RAM_ROLAVG_16***
Routines used	***ROM_INIT_FILTER, ROM_ROLL_AVG***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

Error detection and handling:

ROM routine name	***ROM_EH***
Description	Error Handling: This routine checks all error flags and suppresses processing of wrong results.
Prerequisite	-
Input parameters / register values	error flags in ***SRR_ERR_FLAG***
Output/Return value	error counters and flags are updated.
Temporary RAM	-
Permanent RAM	***RAM_R_TM_ERR_CTR, RAM_R_AM_ERR_CTR, RAM_R_USM_ERR_CTR, RAM_R_FW_STATUS, RAM_R_PT_INT_TEMPERATURE, RAM_R_PTC_TEMPERATURE,***

	RAM_R_PTC_TEMPERATURE, RAM_R_V2D_PT_INT_TEMPERATURE_OLDVAL, RAM_R_V2B_PTC_TEMPERATURE_OLDVAL, RAM_R_V2C_PTH_TEMPERATURE_OLDVAL, RAM_R_FHL_ERR_CTR, RAM_R_FLOW_LPH, RAM_R_THETA, RAM_R_V29_FLOW_LPH_OLDVAL, RAM_R_V2A_FLOW_THETA_OLDVAL
Routines used	***ROM_REPLACE_WITH_OLD_TOFS***
Unchanged registers	**Z**

ROM routine names	***ROM_PP_AM_MON / ROM_PP1_AM_MON***
Description	AM monitoring: This routine reads the raw amplitude values from the front end data buffer and checks if they are above the user given limit in ***FWD_R_AM_MIN***. This is done by direct comparison of ***FDB_US_AM_U*** and ***FDB_US_AM_D*** with ***RAM_R_AM_MIN_RAW*** (provided by ***ROM_PP_AM_CALIB*** or ***ROM_PP1_AM_CALIB***). The routine sets the flag **BNR_AMP_VAL_TOO_LOW** bit in RAM_R_FW_ERR_FLAGS register, if any of the amplitudes is too low, or clears it in the opposite case (sufficient signal amplitudes). The alternative call ***ROM_PP1_AM_MON*** does not need the RAM cell ***RAM_R_AM_MIN_RAW***, it gets the same value from Z.
Prerequisite	An AM measurement must be done before, with valid results in **FDB_US_AM_U** and **FDB_US_AM_D**.
Input parameters / register values	RAM_R_AM_MIN_RAW: Min. amplitude raw value in HSC periods (fd 16) equivalent to the user's minimum amplitude limit **FWD_R_AM_MIN** in mV. or alternatively, with ***ROM_PP1_AM_MON***, use Z instead as input: **Z**: Minimal. amplitude raw value in HSC periods (as above) (fd 16)
Output/Return value	**BNR_AMP_VAL_TOO_LOW** in ***RAM_R_FW_ERR_FLAGS*** register
Temporary RAM	-
Permanent RAM	***RAM_R_FW_ERR_FLAGS*** only ***ROM_PP_AM_MON***: ***RAM_R_AM_MIN_RAW***
Routines used	-
Unchanged registers	**X**, **Y**

ROM routine name	***ROM_PP_AM_CALIB / ROM_PP1_AM_CALIB***
Description	These routines are used after an amplitude calibration measurement. Using the new amplitude calibration values (**FDB_US_AMC_VH** and **FDB_US_AMC_VL**), they calculate gradient ***RAM_R_AMC_GRADIENT*** and offset ***RAM_R_AMC_OFFSET*** that are needed for calculating actual amplitudes (this is not done here, see manual for equations). In addition, they scale the amplitude limit **FWD_R_AM_MIN** into an equivalent raw value ***RAM_R_AM_MIN_RAW***, which can be directly compared to measured time values. This avoids frequent multiplications or divisions. Note that all calculated values change slowly over time, so they need to be updated rarely, but regularly. The routine ***ROM_PP_AM_CALIB*** uses a hard-coded typical AM amplitude value of 350mV as reference. The alternative call ***ROM_PP1_AM_CALIB*** has an input cell from firmware data (***FWD_R_VCAL_TYP***) instead, such that the voltage reference can be adapted if necessary. Applied formulae: ***RAM_R_AMC_GRADIENT*** = ***VCAL***/ (**FDB_US_AMC_VH - FDB_US_AMC_VL**) ***RAM_R_AMC_OFFSET*** = (2 * **FDB_US_AMC_VL - FDB_US_AMC_VH**)

	* ***RAM_R_AMC_GRADIENT*** (note: for subsequent amplitude calculations, ***RAM_R_AMC_OFFSET*** must be further corrected outside the routine: offset = ***RAM_R_AMC_OFFSET*** +(***SHR_ZCD_LVL*** - 796)*0.9V/1024) ***RAM_R_AM_MIN_RAW*** = (**FWD_R_AM_MIN + AMC_OFFSET** + **AM_CORR_FACTOR**) / **AMC_GRADIENT** **AM_CORR_FACTOR**= (**SHR_ZCD_LVL** - 796) * 0.9V /1024 **SHR_ZCD_LVL** is the current zero cross detection level (LSB ≈0.88mV).
Prerequisite	AM calibration measurements must be done before, with valid results in **FDB_US_AMC_VH** and **FDB_US_AMC_VL**. The zero cross detection level **SHR_ZCD_LVL** must be adjusted. All these parameters are assumed to change slowly enough to be considered constant between two AM calibrations.
Input parameters / register values	**FWD_R_AM_MIN**: User given lower amplitude limit in mV (fd 16) only ***ROM_PP1_AM_CALIB***: **FWD_R_VCAL_TYP**: reference amplitude in mV (integer)
Output/Return value	**X** = ***RAM_R_AM_MIN_RAW*** in HSC periods: (fd 16) raw lower amplitude limit for direct measurement comparison **Y** = ***RAM_R_AMC_GRADIENT*** in mV/HSC period, (fd 16) ***RAM_R_AMC_OFFSET*** in mV: (fd 16) parameters for calculation of amplitudes in mV from **FDB_US_AM_U** and **FDB_US_AM_D** (use raw values in HSC periods)
Temporary RAM	***RAM_R_VA9_AMC_DIFF***
Permanent RAM	**FWD_R_AM_MIN*****, RAM_R_AM_MIN_RAW, RAM_R_AMC_GRADIENT, RAM_R_AMC_OFFSET***; only ***ROM_PP1_AM_CALIB: FWD_R_VCAL_TYP***
Routines used	***ROM_FORMAT1_64_TO_32BIT***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

Pulse interface and flow volume:

ROM routine name	***ROM_CFG_PULSE_IF***
Description	This routine configures the pulse interface with the parameters calculated from the given configuration. The routine thus prepares the use of ***ROM_PI_UPD*** for flow output over the pulse interface. The implemented configuration aims at generating not more than one pulse per auto update, to prevent multiple pulses.
Prerequisite	-
Input parameters / register values	**X** : Number of pulses per liter (***PULSE_PER_LITER***) (integer) **Y** : Maximum flow in liter per hour (MAX_FLOW) (integer) **Z** : ***MEAS_RATE_INV*** = ((**TS_CM** + 1) * **TS_CT** * **TOF_RATE**) / (1000 + (**LP_MODE***24)) (fd 16) - **TS_CM** : Cycle mode (Task sequencer) - **TS_CT**: Cycle time (Task sequencer) - **TOF_RATE**: Time of Flight Rate - **LP_MODE**: Low Power Mode (accounts for change in task sequencer period in low power mode from 1ms to 1000/1024ms) **LP_MODE** - Low Power Mode
Output/Return value	Pulse interface registers **SHR_PI_AU_NMB, SHR_PI_AU_TIME, SHR_PI_TPA** and the **PI_TPW** bits in **CR_PI** are configured. **X**: ***FLOW_SCALE_FACT*** = ***PULSE_PER_LITER**** ***MEAS_RATE_INV*** (fd 16) The ***FLOW_SCALE_FACT*** is used to be multiplied with the actual flow (l/h) for updating the pulse interface. It is typically applied by moving it to ***RAM_R_FLOW_SCALE_FACT*** before calling ***ROM_PI_UPD***.
Temporary RAM	***RAM_R_VA4_FLOWVAR_2, RAM_R_VA5_FLOWVAR_1***
Permanent RAM	-
Routines used	***ROM_DIV_BY_SHIFT***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_PI_UPD***
Description	Pulse Interface Update routine This routine calculates the number of pulses equivalent to a given flow (in l/h), based on the configuration settings, and initializes it in the **SHR_PI_NPULSE** register.
Prerequisite	***ROM_CFG_PULSE_IF*** must be called ONCE before using this routine, and the resulting ***FLOW_SCALE_FACTOR*** must be moved to ***RAM_R_FLOW_SCALE_FACT***.
Input parameters / register values	**X** : Flow in liter per Hour (fd 16) ***RAM_R_FLOW_SCALE_FACT*** must contain ***FLOW_SCALE_FACT*** (fd 16) (using ***ROM_CFG_PULSE_IF*** routine, see above)
Output/Return value	**SHR_PI_NPULSE** register is updated with the Number of Pulses equivalent to the current flow in l/h.
Temporary RAM	***RAM_R_VA5_FLOWVAR_1***

Permanent RAM	***RAM_R_FLOW_SCALE_FACT***
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_PP_PI_UPD***
Description	This routine organizes the update of the pulse interface by calling ***ROM_PI_UPD*** with ***RAM_R_FLOW_LPH*** as input argument.
Prerequisite	-
Input parameters / register values	***RAM_R_FLOW_LPH***: current flow in l/h (fd 16)
Output/Return value	**SHR_PI_NPULSE** register is updated with the Number of Pulses equivalent to the current flow in l/h.
Temporary RAM	-
Permanent RAM	***RAM_R_FLOW_LPH***
Routines used	***ROM_PI_UPD***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_RECFG_PULSEIF_FOR_ERROR1***
Description	Reconfigure the pulse interface outputs GPIO0 and GPIO1 as normal GPIOs to signal an error. Use this routine once whenever the pulse interface has to signal an error, then repeat calling (e.g. once in every measurement cycle) ***ROM_SIGNAL_ERROR_ON_PULSEIF***.
Prerequisite	-
Input parameters / register values	-
Output/Return value	Register **CR_GPC** is configured for using GPIO 0 and GPIO 1 as normal GPIO output, such that the CPU has direct control over the signals.
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Y**, **Z**

ROM routine name	***ROM_SIGNAL_ERROR_ON_PULSEIF1***
Description	Signaling error on the “pulse interface” outputs GPIO0 and GPIO1. GPIO0: Output =1 GPIO1 : Output - toggling signal (toggles after every routine call). Prepare signaling an error over the pulse interface by calling ***ROM_RECFG_PULSEIF_FOR_ERROR*** (to hand over signal control to the CPU). Then repeat calling ***ROM_SIGNAL_ERROR_ON_PULSEIF*** (e.g. once in every measurement cycle) as long as the error condition remains.
Prerequisite	Call ***ROM_RECFG_PULSEIF_FOR_ERROR*** once before
Input parameters / register values	The last state of GPIOs is stored in **SHR_GPO**
Output/Return value	GPIO0: Output =1 GPIO1 : Output - toggling signal (toggles after every routine call).
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**Y**, **Z**

ROM routine name	***ROM_RECFG_PULSEIF_FOR_PULSE1***
Description	Configures GPIO0 and GPIO1 as pulse interface outputs in direction mode. Use this routine to initialize the pulse interface on GPIO0 and GPIO1, or to end an error state after ***ROM_RECFG_PULSEIF_FOR_ERROR*** and ***ROM_SIGNAL_ERROR_ON_PULSEIF*** were applied.
Prerequisite	-
Input parameters / register values	-
Output/Return value	Register **CR_GPC** is configured for using GPIO 0 and GPIO 1 as pulse interface outputs in direction mode.
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Y**, **Z**

ROM routine names	***ROM_SAVE_FLOW_VOLUME / ROM_SAVE01_FLOW_VOLUME*** ***ROM_SAVE1_FLOW_VOLUME / ROM_SAVE11_FLOW_VOLUME*** ***ROM_SAVE2_FLOW_VOLUME / ROM_SAVE21_FLOW_VOLUME*** ***ROM_SAVE3_FLOW_VOLUME / ROM_SAVE31_FLOW_VOLUME***
Description	These routines are used to calculate the flow volume of one measurement cycle from the present flow (usually in l/h), and store it cumulatively to flow volume (usually in cubic meter). Operation***: FLOW_LPH*** * ***VOLUME_FACTOR*** (V.F.) -> flow (e.g. in cubic meters) per cycle -> accumulate (add to/subtract from) the flow volume (integer and fractional part) Calculation steps ***for VOLUME_FACTOR*** for liter -> l/h: a) ***FLOW_LPH***/3600 -> ***FLOW_LPS*** b) ***FLOW_LPS*** * ***MEAS_RATE_INV*** -> ***FLOW*** in liter per meas. cycle c) Flow in liter per meas. cycle/1000 -> Flow in cubic meter per cycle d) Flow in cubic meter per cycle -> add it to the Flow Volume (integer and fractional part) Actual calculation of the ***VOLUME_FACTOR*** from configuration data: ***VOLUME_FACTOR*** = [(**TS_CM** + 1) * **TS_CT** * **TOF_RATE**] / [(1000 + (**LP_MODE***24))*3600*1000] Here the following configuration parameters are used: - **TS_CM** : Cycle mode (Task sequencer) - **TS_CT**: Cycle time (Task sequencer) - **TOF_RATE**: Time of Flight Rate - **LP_MODE**: Low Power Mode (accounts for change in task sequencer period in low power mode from 1ms to 1000/1024ms) The actual decision on units is done by the user through defining the appropriate input scaling (l/h or something else) and VOLUME_FACTOR.

Difference between the routine calls: The routines operate either on usual RAM or on firmware data (FWD) RAM, which is useful for regular per-manent storage. Their input comes from X, Y

ROM routine name	RAM region:	input from:	parameterm eaning:
[1]***ROM_SAVE_FLOW_VOLUME***	RAM	RAM	Flow & V.F.
[2]***ROM_SAVE1_FLOW_VOLUME***	RAM	X,Y	Flow & V.F.
[3]***ROM_SAVE2_FLOW_VOLUME***	FWD	X,Y	Flow & V.F.
[4]***ROM_SAVE3_FLOW_VOLUME***	FWD	FWD-RAM	Flow & V.F.
[5]***ROM_SAVE01_FLOW_VOLUM E,***	RAM	X,Y	Volume

or RAM, and can have different meaning (see below).	***[6]ROM_SAVE11_FLOW_VOLUME***			
	[7]ROM_SAVE21_FLOW_VOLUME, ***[8]ROM_SAVE31_FLOW_VOLUME***	FWD	X,Y	Volume
Prerequisite depending on actual call, see above	The RAM cells ***RAM_R_FLOW_VOLUME_INT*** and ***RAM_R_FLOW_VOLUME_FRACTION*** must contain the flow volume of previous measurements. [1,4]RAM cells used for input must contain the right parameter (see above)			
Input parameters / register values: The meaning of input depends on the actual call, see above.	[2,3]**X** or ***[4]FWD_R_FLOW_LPH*** or ***[1]RAM_R_FLOW_LPH***: the present flow value (fd 16, usually in l/h) or [5,6,7,8]**X**: the additional flow volume (fd 32, usually cubic meters) [2,3]**Y** or ***[4]FWD_R_VOLUME_FACTOR*** or ***[1]FWD_R_VOLUME_FACTOR***: ***VOLUME_FACTOR*** (fd 44; note that this is usually a very small number, so the upper 12 fractional digits are zero and "above" the actual data word) or [5,6,7,8]**Y** = **X**			
Output/Return value Output may be in usual RAM or FWD, depending on actual call, see above.	64-bit Volume Flow result in RAM Addresses [1,2,5,6]***RAM_R_FLOW_VOLUME_INT*** (integer, usually in cubic meters) ***RAM_R_FLOW_VOLUME_FRACTION*** (fd 32, usually in cubic meters) or [3,4,7,8]***FWD_R_FLOW_VOLUME_INT*** (integer, usually in cubic meters) ***FWD_R_FLOW_VOLUME_FRACTION*** (fd 32, usually in cubic meters)			
Temporary RAM	-			
Permanent RAM depending on actual call, see above	[1,2,5,6]***RAM_R_FLOW_VOLUME_INT***, ***RAM_R_FLOW_VOLUME_FRACTION*** or [3,4,7,8]***FWD_R_FLOW_VOLUME_INT, FWD_R_FLOW_VOLUME_FRACTION***; [1]***RAM_R_FLOW_LPH, RAM_R_VOLUME_FACTOR*** or [4]***RAM_R_FLOW_LPH, RAM_R_VOLUME_FACTOR***			
Routines used	-			
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)			

Sensor temperature measurement:

ROM routine name	***ROM_TEMP_POLYNOM***
Description	This routine calculates the temperature of a PT sensor in °C using the polynomial approximation Temperature T = {[(***PT_COEFF2*** * *PT_RATIO*) + ***PT_COEFF1***] * ***PT_RATIO*** } + ***PT_COEFF0*** where ***PT_RATIO*** = **PT_RES** / R0 of the PT sensor ***PT_COEFF2*** = 10.115 (fd 16) ***PT_COEFF1*** = 235.57 (fd 16) ***PT_COEFF0*** = -245.683 (fd 16) This polynomial resembles the inverted R(T)-polynomial for PT (according to IEC 60751:2008) within 3mK accuracy between 0°C and 100°C.
Prerequisite	-

Input parameters / register values	**X**: PT_RATIO (fd 16)
Output/Return value	**X**: Temperature in °C (fd 16)
Temporary RAM	-
Permanent RAM	-
Routines used	***ROM_FORMAT1_64_TO_32BIT***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	*ROM_TEMP_LINEAR_FN*
Description	This routine is used to calculate the temperature of any sensor as a linear function of sensor resistance using the nominal resistance and sensor slope. Applied formula: Temperature T = (Sensor Resistance at T[°C] – Nominal resistance) / ***RAM_R_VAF_REF_RES_VAL*** * Sensor slope
Prerequisite	-
Input parameters / register values	**X**: Nominal resistance (fd 16) **Y**: Sensor slope (fd 16) **Z**: Sensor resistance (fd 16) ***RAM_R_VAF_REF_RES_VAL***: Reference Resistance (fd 16)
Output/Return value	**X**: Temperature (fd 16)
Temporary RAM	-
Permanent RAM	***RAM_R_VAF_REF_RES_VAL***
Routines used	***ROM_FORMAT1_64_TO_32BIT***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	*ROM_TM_SUM_RESULT*
Description	In sensor temperature measurement, each single time measurement is repeated after some fixed delay time. Averaging these results eliminates a possible 50/60 Hz disturbance. This routine sums up all duplicate measurements (from the frontend data buffer in cells for measurement 1 and 2) and stores it in the frontend data buffer (in the cells of measurement 1). The routine works for all 2-wire or 4-wire temperature measurement results, it reads the configuration from **CR_TM**.
Prerequisite	The routine should be called directly after a temperature measurement.
Input parameters / register values	All frontend data buffer (**FDB**) cells (addresses 0x80 – 0x9B)
Output/Return value	The added results overwrite the original measurements in the first measurement **FDB** cells (addresses 0x80- 0x84 and 0x8A - 0x92)
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

Interface communication

ROM routine name	*ROM_I2C_ST*
Description	I2C Start Byte Transfer: Initiate an I2C read or write operation, depending on preceding i2crw-command (1=read, 0=write).
Prerequisite	I2C slave device address must be defined in **CR_PI_I2C**. Read or write direction must be defined by command i2crw (1=read, 0=write).
Input parameters / register values	-
Output/Return value	**SRR_MSC_STF** contains a flag to indicate I2C acknowledge (Bit **I2C_ACK**).
Temporary RAM	-

Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Y**, **Z**, **R**

ROM routine name	***ROM_I2C_BT***
Description	I2C Byte Transfer: Read or write one byte of data over I2C, depending on preceding i2crw-command (1=read, 0=write).
Prerequisite	I2C slave device address must be defined in **CR_PI_I2C**. Read or write direction must be defined by command i2crw (1=read, 0=write).In write case, R must point to the desired input RAM cell, and usually the bytedir and bytesel command must be used to select the desired byte part of the 4Byte-word in the RAM cell (use bytedir 0 and bytesel 4, 5, 6 or 7).
Input parameters / register values	R is not changed, but used as pointer to the data register by the chip hardware in write case (see “Prerequisite”).
Output/Return value	**SRR_MSC_STF** contains a flag to indicate I2C acknowledge (Bit **I2C_ACK**). In read case, **SRR_E2P_RD** contains the transferred byte. For storing the received byte in a 4Byte RAM cell, use the bytedir and bytesel command to select the desired byte position (use bytedir 1 and bytesel 4, 5, 6 or 7, and the or command to add a new byte to a partly filled 4Byte-word).
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Y**, **Z**, **R**

ROM routine name	***ROM_I2C_LT***
Description	I2C Byte Transfer: Read or write the last transmitted byte of data over I2C, depending on preceding i2crw-command (1=read, 0=write). The routine sends the stop signal at the end of transmission.
Prerequisite	I2C slave device address must be defined in **CR_PI_I2C**. Read or write direction must be defined by command i2crw (1=read, 0=write).In write case, R must point to the desired input RAM cell, and usually the bytedir and bytesel command must be used to select the desired byte part of the 4Byte-word in the RAM cell (use bytedir 0 and bytesel 4, 5, 6 or 7).
Input parameters / register values	R is not changed, but used as pointer to the data register by the chip hardware in write case (see “Prerequisite” below).
Output/Return value	**SRR_MSC_ST**F contains a flag to indicate I2C acknowledge (Bit **I2C_ACK**). In read case, **SRR_E2P_RD** contains the transferred byte. For storing the received byte in a 4Byte RAM cell, use the bytedir and bytesel command to select the desired byte position (use bytedir 1 and bytesel 4, 5, 6 or 7, and the or command to add a new byte to a partly filled 4Byte-word).
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**X**, **Y**, **Z**, **R**

ROM routine name	***ROM_I2C_DWORD_WR***
Description	This routine is used to write a 4 byte-word of data to the I2C slave device, starting at a given memory address.The device address for the I2C device is taken automatically from I2C slave address of **CR_PI_I2C**. The routine starts with switching on the HSC and switches it off after transmission. It thus causes a high current consumption and has a long runtime. In power critical applications, its use should thus be restricted. Timing properties of the I2C slave should also be considered (e.g. long storing times after some data transmission).
Prerequisite	-
Input parameters / register values	**X**: 16-bit memory address (start) where the data has to be written **Y**: 4 bytes of data
Output/Return value	The four bytes from **Y** are written to the I2C slave, starting at the address given in X. In case of an error, the transmission was not acknowledged by the I2C slave device and bit **BNR_I2C_ABORT** of ***RAM_R_FW_STATUS*** is set.
Temporary RAM	-
Permanent RAM	***RAM_R_VA1_I2CADDR, RAM_R_VA2_I2CDATA, RAM_R_FW_STATUS***
Routines used	***ROM_I2C_ST, ROM_I2C_BT, ROM_I2C_LT***
Unchanged registers	**Z**

ROM routine name	***ROM_I2C_BYTE_WR***
Description	This routine is used to write a single of data to the I2C slave device to given memory address. The device address for the I2C device is taken automatically from I2C slave address of **CR_PI_I2C**. The routine starts with switching on the HSC and switches it off after transmission. It thus causes a high current consumption and has a long runtime. In power critical applications, its use should thus be restricted. Timing properties of the I2C slave should also be considered (e.g. long storing times after some data transmission).
Prerequisite	-
Input parameters / register values	**X**: 16-bit address where the data has to be written **Y**: 1 byte of data (B0 of the 32 bit-word in Y is transferred)
Output/Return value	Byte B0 from Y is written to the I2C slave to the address given in X. In case of an error, the transmission was not acknowledged by the I2C slave device and bit **BNR_I2C_ABORT** of ***RAM_R_FW_STATUS*** is set.
Temporary RAM	-
Permanent RAM	***RAM_R_VA1_I2CADDR, RAM_R_VA2_I2CDATA, RAM_R_FW_STATUS***
Routines used	***ROM_I2C_ST, ROM_I2C_BT, ROM_I2C_LT***
Unchanged registers	**Z**

ROM routine name	***ROM_I2C_DWORD_RD***
Description	This routine is used to sequentially read 4 data bytes from the I2C slave device, starting at a given memory address. The device address for the I2C device is taken automatically from I2C slave address of **CR_PI_I2C**. The routine starts with switching on the HSC and switches it off after transmission. It thus causes a high current consumption and has a long runtime. In power critical applications, its use should thus be restricted. Timing properties of the I2C slave should also be considered.
Prerequisite	-
Input parameters / register values	**X**: 16-bit memory address (start) where the data is read.

Output/Return value	**X**: 4 bytes of read data In case of an error, the transmission was not acknowledged by the I2C slave device and bit **BNR_I2C_ABORT** of ***RAM_R_FW_STATUS*** is set.
Temporary RAM	-
Permanent RAM	***RAM_R_VA1_I2CADDR, RAM_R_FW_STATUS***
Routines used	***ROM_I2C_ST, ROM_I2C_BT, ROM_I2C_LT***
Unchanged registers	**Z**

ROM routine name	***ROM_I2C_BYTE_RD***
Description	This routine is used to sequentially read one single byte from the I2C slave device from a given memory address. The device address for the I2C device is taken automatically from I2C slave address of **CR_PI_I2C**. The routine starts with switching on the HSC and switches it off after transmission. It thus causes a high current consumption and has a long runtime. In power critical applications, its use should thus be restricted. Timing properties of the I2C slave should also be considered.
Prerequisite	-
Input parameters / register values	**X**: 16-bit memory address where the data is read.
Output/Return value	**X**: Single data byte, stored in byte B0 of X In case of an error, the transmission was not acknowledged by the I2C slave device and bit **BNR_I2C_ABORT** of ***RAM_R_FW_STATUS*** is set.
Temporary RAM	-
Permanent RAM	***RAM_R_VA1_I2CADDR, RAM_R_FW_STATUS***
Routines used	***ROM_I2C_ST, ROM_I2C_BT, ROM_I2C_LT***
Unchanged registers	Z

ROM routine names	***ROM_COPY_UART_PRB_DATA / ROM_COPY1_UART_PRB_DATA***
Description	This routine copies the relevant data flow, temperature and flow volume for the UART Master into the probe data area. When the system is setup for UART communication, the data is then sent over UART interface to the master after the firmware execution ended (see manual). Probe Data Area 0xA0 - 0xA3 The alternative call ***ROM_COPY1_UART_PRB_DATA*** takes input data from firmware data (**FWD**) cells instead of usual RAM cells, which in some cases is preferable for permanent storage or memory optimization.
Prerequisite	The probe data area has to be configured in CR_UART (***UART_DATA_MSG_ADR***=0xA0 and ***UART_DATA_MSG_LEN***=4)
Input parameters / register values	***RAM_R_FLOW_LPH, RAM_R_THETA, RAM_R_FLOW_VOLUME_INT, RAM_R_FLOW_VOLUME_FRACTION*** or, using ***ROM_COPY1_UART_PRB_DATA*** ***FWD_R_FLOW_LPH, FWD_R_THETA, FWD_R_FLOW_VOLUME_INT, FWD_R_FLOW_VOLUME_FRACTION***
Output/Return value	using ***ROM_COPY_UART_PRB_DATA*** or ***ROM_COPY1_UART_PRB_DATA***: ***RAM_R_VA0_UART_PRB_DATA1=RAM***.../ ***FWD_R_FLOW_LPH***, ***RAM_R_VA1_UART_PRB_DATA2=RAM***.../ ***FWD_R_THETA***, ***RAM_R_VA2_UART_PRB_DATA3=RAM***.../ ***FWD_R_FLOW_VOLUME_INT***, ***RAM_R_VA3_UART_PRB_DATA4*** = ***RAM***.../ ***FWD_R_FLOW_VOLUME_FRACTION***
Temporary RAM	-
Permanent RAM	***RAM_R_FLOW_LPH, RAM_R_THETA, RAM_R_FLOW_VOLUME_INT, RAM_R_FLOW_VOLUME_FRACTION***

	or, using ***ROM_COPY1_UART_PRB_DATA*** ***FWD_R_FLOW_LPH, FWD_R_THETA, FWD_R_FLOW_VOLUME_INT, FWD_R_FLOW_VOLUME_FRACTION; RAM_R_VA0_UART_PRB_DATA1, RAM_R_VA1_UART_PRB_DATA2, RAM_R_VA2_UART_PRB_DATA3, RAM_R_VA3_UART_PRB_DATA4***
Routines used	-
Unchanged registers	**Y**, **Z**

Housekeeping:

ROM routine name	***ROM_CPU_CHK***
Description	Check kind of CPU request: This routine is called by hardware design after any Post Processing (PP) request. It checks the system handling register **SHR_CPU_REQ** and calls the requested routines.
Prerequisite	-
Input parameters / register values	Flag settings in **SHR_CPU_REQ**
Output/Return value	The routine directly calls firmware (**MK_CPU_REQ**), boot loader (***ROM_BLD***) or checksum generation (***ROM_CSM***) using goto. These routines generally return, after execution, to the start of ***ROM_CPU_CHK*** to see if **SHR_CPU_REQ** has changed in the meantime.
Temporary RAM	(depending on called routines)
Permanent RAM	(depending on called routines)
Routines used	**MK_CPU_REQ**, ***ROM_BLD, ROM_CSM***
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)
Call Address	61440 / 0xF000

ROM routine name	***ROM_USER_RAM_INIT***
Description	This ROM routine is used to initialize the entire user RAM with 0 as default value.
Prerequisite	-
Input parameters / register values	-
Output/Return value	All 176 user RAM cells (addresses 0x00 - 0xAF) are initialised to 0
Temporary RAM	All 176 user RAM cells
Permanent RAM	-
Routines used	-
Unchanged registers	**Y**, **Z**

High speed oscillator:

ROM routine name	***ROM_HSO_WAIT_SETTL_TIME***
Description	This routine is used to switch on the HSO clock and wait for a fixed settling time to get a stable oscillation. A settling time of 122 us (4 clock periods of the low speed clock) is waited, using the low speed clock count value **SRR_LSC_CV**.
Prerequisite	-
Input parameters / register values	-
Output/Return value	-
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**Z**

ROM routine name	***ROM_HSC_CALIB***
Description	High speed clock (HSC) calibration is based on a measurement of 4 periods of the 32 kHz clock, which is done by the task sequencer as configured. The result is stored in SRR_HCC_VAL as number of high speed clock periods (fd 16, ideally 488.28125 or 976.5625 periods). This routine evaluates the high speed clock scaling factor for the 4 MHz or 8 MHz clock using the formula 4 MHz : ***HSC_SCALE_FACT*** = 4 MHz/(32.768kHz/4) / **SRR_HCC_VAL** 8 MHz : ***HSC_SCALE_FACT*** = 8 MHz/(32.768kHz/4) / **SRR_HCC_VAL** It additionally checks for the deviation of measured clock period from its ideal value. If the deviation is greater than the input value in Z, then no scaling is done and bit **BNR_HS_CALIB_FAIL** in the ***RAM_R_FW_ERR_FLAGS*** register is set.
Prerequisite	The measurement of **SRR_HCC_VAL** must be done (initiated by the task sequencer)
Input parameters / register values	**Z**: Maximum allowed clock deviation in periods of the HSC (fd 16) (usually from the firmware register value **FWD_HSC_DEV** before the call).
Output/Return value	Calibration factor in ***RAM_R_HSC_SCALE_FACT*** (fd 24)
	Bit **BNR_HS_CALIB_FAIL** (Bit 0) in ***RAM_R_FW_ERR_FLAGS*** register is set if the clock deviation is greater than allowed.
Temporary RAM	-
Permanent RAM	***RAM_R_HSC_SCALE_FACT, RAM_R_FW_ERR_FLAGS***
Routines used	-
Unchanged registers	**Z**

ROM routine name	***ROM_SCALE_WITH_HSC***
Description	Routine to scale the input parameter with the HS Clock Calibration factor (***RAM_R_HSC_SCALE_FACT***) Scaled output = Input / ***RAM_R_HSC_SCALE_FACT***
Prerequisite	***RAM_R_HSC_SCALE_FACT*** must have the valid HS Clock Calibration factor (Use ***ROM_HSC_CALIB*** routine)
Inputs	**X** - Parameter to be scaled (any format, integer value < 2^{30})
Output	**X** - Scaled parameter (same format as input X)
Temporary RAM	-
Permanent RAM	***RAM_R_HSC_SCALE_FACT***
Routines used	-
Unchanged registers	**Z**

Configuration:

ROM routine name	***ROM_RESTORE_TOF_RATE*** / ***ROM_RESTORE1_TOF_RATE***
Description	Routine to reconfigure the **TOF_RATE** generator to the original setting, stored in ***RAM_R_CFG_TOF_RATE***. The alternative call ***ROM_RESTORE1_TOF_RATE*** does not use *RAM_R_CFG_TOF_RATE*, but Y as input for the TOF rate and thus permits a differently chosen value. The routine can only be used for TOF rates up to 31.
Prerequisite	-
Input parameters / register values	***RAM_R_CFG_TOF_RATE*** (using ***ROM_RESTORE_TOF_RATE***) or **Y** (using *ROM_RESTORE1_TOF_RATE*): new TOF rate (< 32) (integer)

Output/Return value	The TOF rate is reconfigured in **SHR_TOF_RATE** according to the input, and bit **BNR_TOF_RATE_REDUCED** in ***RAM_R_FW_STATUS*** is cleared.
Temporary RAM	-
Permanent RAM	***RAM_R_CFG_TOF_RATE***, ***RAM_R_FW_STATUS***
Routines used	-
Unchanged registers	**X**, **Z**

ROM routine name	***ROM_RECFG_TOF_RATE***
Description	Routine to reconfigure **TOF_RATE** generator for less measurements, depending on the parameter N: New **TOF_RATE** = Original **TOF_RATE** * N The actual measurement is done only every Nth time. The routine manipulates **SHR_TOF_RATE** for this adjustment. It is only usable for **TOF_RATES** up to 31.
Prerequisite	-
Input parameters / register values	**X** - Factor N for lowering the **TOF_RATE**
Output/Return value	**Z** - Original **TOF_RATE** value from **SHR_TOF_RATE** Register **TOF_RATE** bits in **SHR_TOF_RATE** Register are changed (see description). Bit **BNR_TOF_RATE_REDUCED** is set in ***RAM_R_FW_STATUS*** register to indicate that the **TOF_RATE** was reconfigured.
Temporary RAM	-
Permanent RAM	**RAM_R_FW_STATUS**
Routines used	-
Unchanged registers	(all registers **X**, **Y** ,**Z** and **R** are in use)

Mathematics:

ROM routine name	***ROM_FORMAT_64_TO_32BIT***
Description	Routine to format a 64-bit value (in **Y** and **X**) into a 32 bit result with 16 integer + 16 fractional bits This can be used to format 64 bit multiplication results with 32 integer + 32 fractional bits into a usual fd 16 word. The MSB of the integer part in **Y** defines the sign, as if **Y** and **X** would be a single 64 bit word. This routine has the same function as ***ROM_FORMAT1_64_TO_32BIT***, but does not need RAM. It is, however, essentially slower.
Prerequisite	-
Input parameters / register values	**Y**: Higher 32 bits of the value (integer part with maximum 16 significant bits, signed !) **X**: Lower 32 bits of value (fractional part, unsigned !)
Output/Return value	**X**: 32-bit result with 16 Integer + 16 fractional bits (fd 16)
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**Z**, **R**

ROM routine name	***ROM_FORMAT1_64_TO_32BIT***
escription	Routine to format a 64-bit value (in **Y** and **X**) into a 32 bit result with 16 integer + 16 fractional bits This can be used to format 64 bit multiplication results with 32 integer + 32 fractional bits into a usual fd 16 word. The MSB of the integer part in **Y** defines the sign, as if **Y** and **X** would be a single 64 bit word.

	This routine has the same function as ***ROM_FORMAT_64_TO_32BIT***, but is essentially faster at the cost of one temporary RAM cell.
Prerequisite	-
Input parameters / register values	**Y**: Higher 32 bits of the value (integer part with maximum 16 significant bits, signed !) **X**: Lower 32 bits of value (fractional part, unsigned !)
Output/Return value	**X**: 32-bit result with 16 Integer + 16 fractional bits (fd 16)
Temporary RAM	***RAM_R_V1F_SHIFT***
Permanent RAM	-
Routines used	-
Unchanged registers	**Z**

ROM routine name	***ROM_DIV_BY_SHIFT***
Description	Routine to perform the division of a value **Y** by **X**, where **X**=2^N is an integer power of two. Result = **Y**/**X** = **Y**/2^N
Prerequisite	-
Input parameters / register values	**X** - Divisor (denominator) = 2^N value (integer) **Y** - Dividend (numerator) (any format)
Output/Return value	**Y** - Result of division (same format as input **Y**)
Temporary RAM	-
Permanent RAM	-
Routines used	-
Unchanged registers	**Z**, **R**

ROM routine name	***ROM_SQRT***
Description	This routine is used to evaluate the square root using the Newton method accurately for values in the range (196 <= X <= 5476). sqrt(x) is calculated by iterating the following steps 1. Choose GUESS = 32; Iteration counter = 3 2. Find x/GUESS (1st division by shift and normal division for next 2 iterations) 3. Average of GUESS and x/GUESS 4. GUESS <--- Average ; Decrement iteration counter 5. Repeat steps 2-4 till counter = 0 6. SquareRoot = Last GUESS value When used in flow temperature calculation, values of X between 196 = (14^2) and 5476 = (74^2) result in temperature errors < 1°C. This X range is equivalent to a temperature range of 60°C to 0°C.
Prerequisite	-
Input parameters / register values	**X** : Radicand (fd 16, 196 <= **X** <= 5476)
Output/Return value	: Square root of input **X** (fd 16)
Temporary RAM	***RAM_R_V2F_SQRT_X, RAM_R_V2E_SQRT_Y***
Permanent RAM	-
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

ROM routine name	***ROM_LINEAR_CORRECTION / ROM_LINEAR1_CORRECTION***
Description	Linear interpolation of a coefficient over any parameter (here: temperature), knowing the coefficient value at two points and given the current value of the parameter (stored in ***RAM_R_VA3_CURRENT_THETA***) Applied formula: Result = slope * (***RAM_R_VA3_CURRENT_THETA*** - Parameter@Point1) + offset = **X** * (**RAM_R_VA3_CURRENT_THETA** - **Y**) + **Z** When the coefficient value is known at two parameter points, slope and offset can be calculated as slope = (Coefficent@Point2 - Coefficent@Point1) / (Parameter@Point2 - Parameter@Point1) offset = Coefficent@Point1 The routine has an alternative call address ***ROM_LINEAR1_CORRECTION***, where the RAM cell of the current parameter value can be freely chosen.
Prerequisite	-
Input parameters / register values	**X** : Slope between the two points (fd 16) **Y**: Parameter@Point1 (fd 16) **Z**: Offset (fd 16) ***RAM_R_VA3_CURRENT_THETA*** current parameter (temperature) (fd 16) With alternative call ***ROM_LINEAR1_CORRECTION***: **R**: Pointer to RAM cell with current parameter value
Output/Return value	**X**: Coefficient corrected linearly over temperature
Temporary RAM	-
Permanent RAM	***RAM_R_VA3_CURRENT_THETA*** (none when ***ROM_LINEAR1_CORRECTION*** is used)
Routines used	***ROM_FORMAT1_64_TO_32BIT***
Unchanged registers	(all registers **X**,**Y**,**Z** and **R** are in use)

ROM routine name	***ROM_FIND_SLOPE***
Description	This routine is used to find the slope between two points, given the coefficient and corresponding parameter values at the two points (for example two correction factors over two temperatures). The routine is used as preparation for any linear interpolation. Basically, all input- and output-values have the same format. Due to internal calculations, the format must be chosen such that the four leading bits of the parameters are zero, and at least 12 leading bits of the resulting slope are zero, too. Otherwise the result will be wrong.
Prerequisite	Parameter interval (***RAM_R_VA5_FLOWVAR***_1 – **Z**) must be numerically larger than (**Y** - **X**)/23, to avoid overflow in an internal division.
Input parameters / register values	**X** : Coefficient at point 1 (any format, typically 16 fd) **Y**: Coefficient at point 2 (same format as **X**) **Z**: Parameter at point 1 (same format as **X**, 4 leading bits must be 0) ***RAM_R_VA5_FLOWVAR_1***: Parameter at point 2 (same format as **Z**)
Output/Return value	**X** : Slope (same format as input **X**; 12 leading bits are always 0)
Temporary RAM	-
Permanent RAM	**RAM_R_VA5_FLOWVAR_1**
Routines used	-
Unchanged registers	(all registers **X**, **Y**, **Z** and **R** are in use)

6 CPU Handling

6.1 CPU Handling

As soon as one of the bits in the system handling register **SHR_CPU_REQ** is set, the CPU starts with the request handling. All bits are typically triggered by the task sequencer, the error handling, a general purpose pin or the remote control. For test or debugging purposes it is also possible to write directly to these registers.

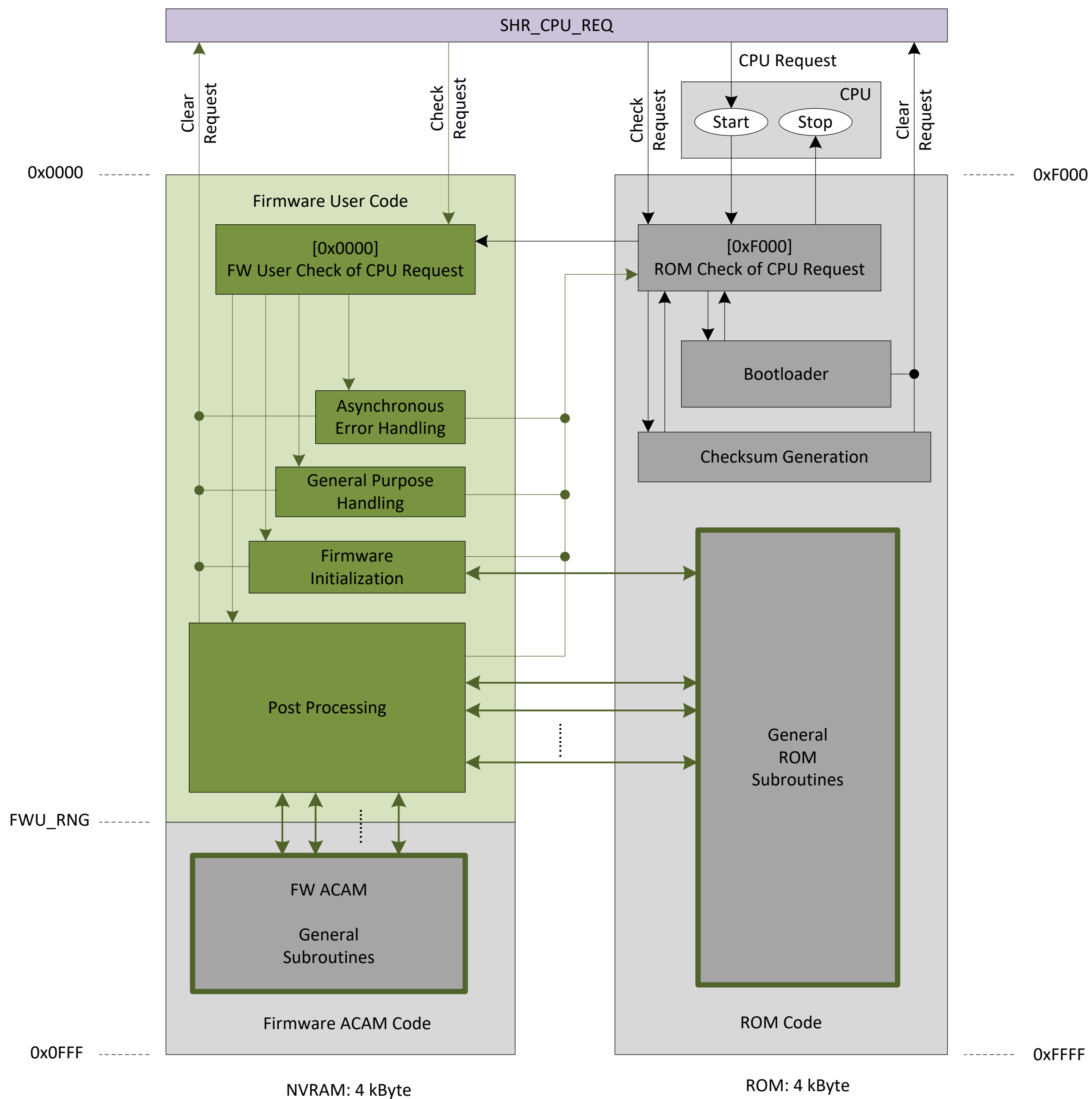
Post processing is activated having set bits **CPU_REQ_EN_PP** in register **CR_IEH** and **PP_EN** in **CR_MRG_TS**. There, following requests are possible to start execution of program code in CPU:

- **CPU_REQ_BLD_EXC:** Bootloader
- **CPU_REQ_CHKSUM:** Checksum Generation
- **CPU_REQ_PP:** Post Processing triggered by task sequencer
- **CPU_REQ_GPH:** General Purpose Handling triggered by general purpose timer
- **CPU_REQ_FW_INIT:** Firmware Initialization

Program Code in green colour has to be defined and programmed by customer, whereby public subroutines in Firmware ACAM Code or ROM Code can also be used by customer.

The request bits have to be cleared by the system program code or the user program code.

Figure 6-1 CPU request handling



6.1.1 Check of CPU Request

In case that any of the request bits is set in **SHR_CPU_REQ** the CPU starts at first with code in the ROM that checks the type of request.

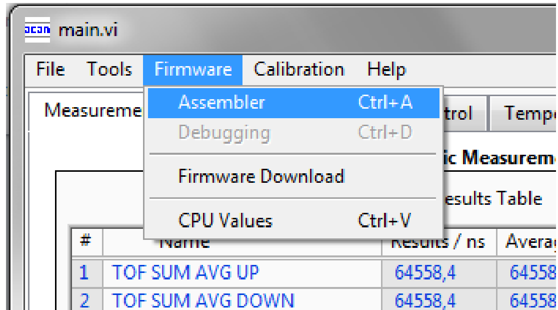
In case of a post processing request, a general purpose request or a firmware initialization request the CPU jumps into firmware code. This means that the user has to implement in his firmware also a CPU request check.

- Post Processing (FW code): This will be the most common request, namely for data post processing like flow or temperature calculation.
- General purpose handling (FW code):

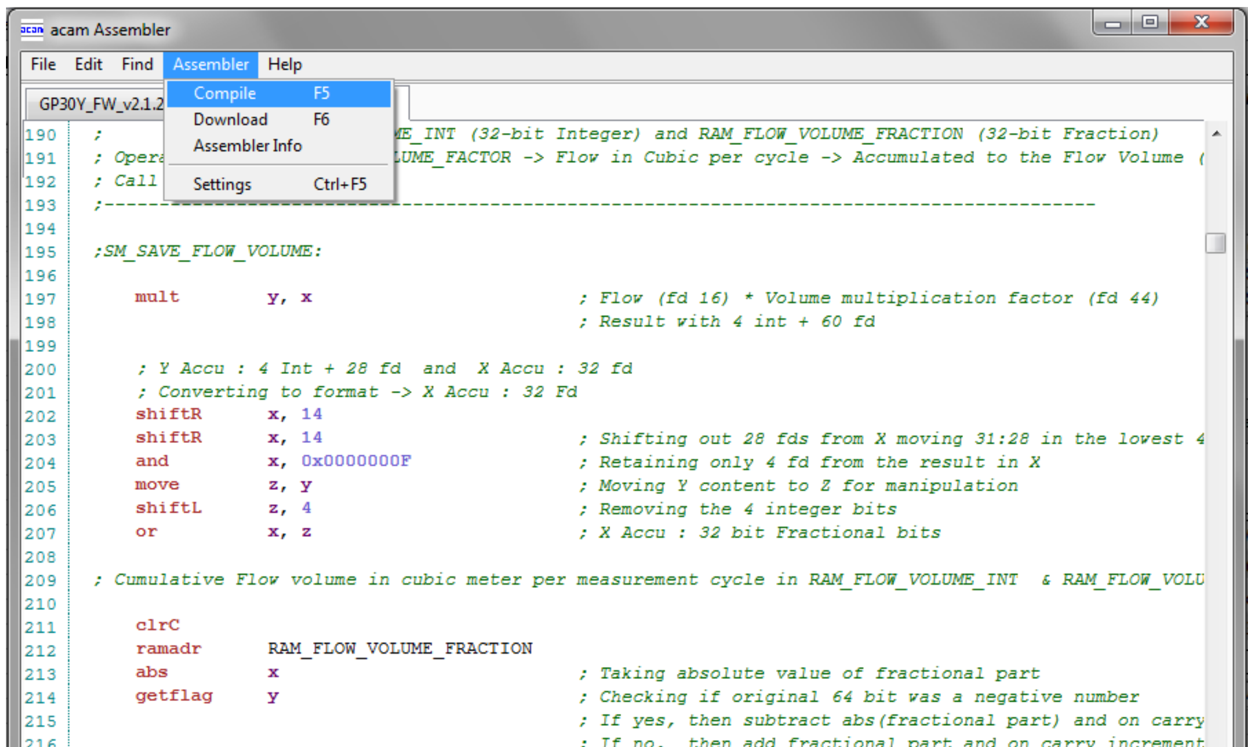
- Bootloader (ROM code): The bootloader is always requested after a system reset has been occurred. The bootloader loads the configuration data into the register area, gets the firmware revision, sets the high speed clock, activates the task sequencer, optionally activates the CRC mode for UART and finally sets the checksum generation request. However, bootloader actions are only performed if the bootloader release code is set (register **BLD_RLS**: 0xABCD_7654). In a final initialization, the bootloader also sets a CPU request for “FW Init” and, if configured, a request for “Checksum Generation”. Finally, the bootloader clears its request in **SHR_CPU_REQ** and jumps back to ROM Check of CPU Request.
- Checksum generation (ROM code): Checksum generation can be requested by remote command RC_FW_CHKSUM, by the bootloader or the checksum timer. The checksums of all four FW areas will be generated and compared to the checksums stored at as **FWD1**, **FWD2**, **FWU** and **FWA** in the **RAA**. At last the checksum generation clears its request in **SHR_CPU_REQ** and jumps back to ROM Check of CPU Request.
- Firmware initialization (FW code): Besides the configuration done by the boatloader some additional configurations can be performed, which typically are some initializations of the SHR register.

7 Assembler Software

The TDC-GP30 assembler is integrated into the GP30 evaluation software. It is opened in the Firmware menu of the main program:



The following window comes up:



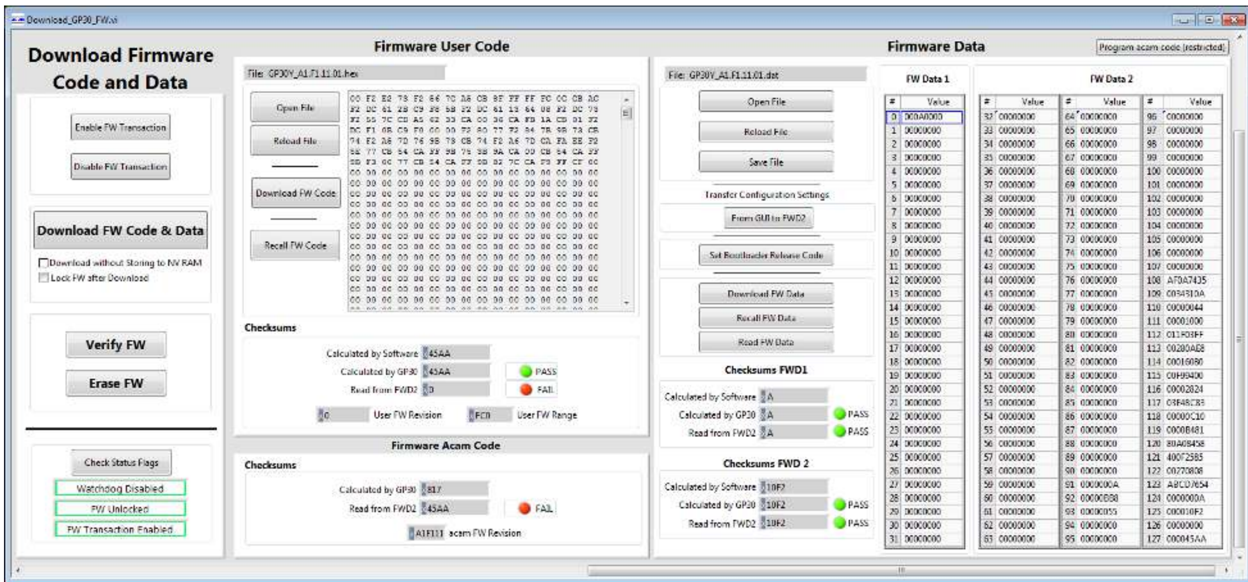
This is a comfortable editor with syntax highlighting, search and replace, copy and paste functions.

Under menu item “Assembler” the user finds the compile and download options.

Whether the call of these functions was successful or not is indicated by the messages at the bottom of the assembler window.

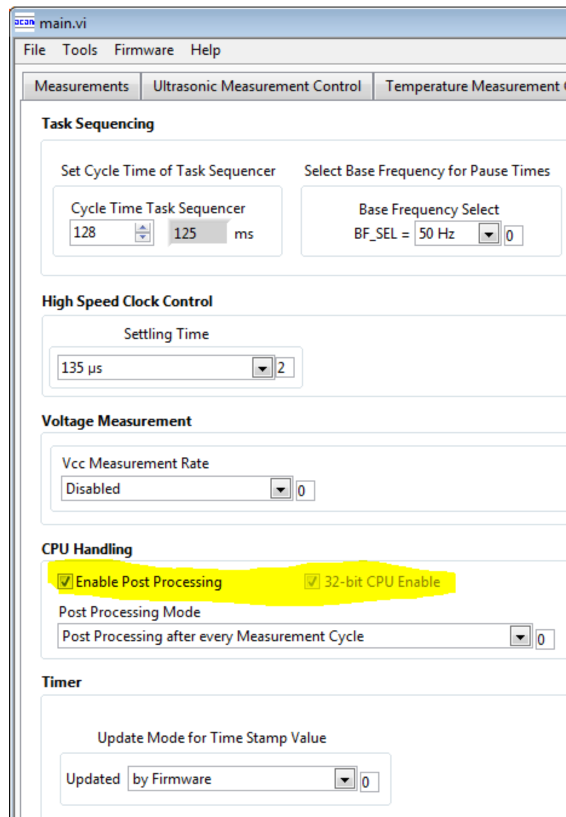
Pressing “Download” here or in the main program’s “Firmware” menu item has the same effect. The “Download Firmware and Data” Window is opened:

Figure 7-1 Firmware download window



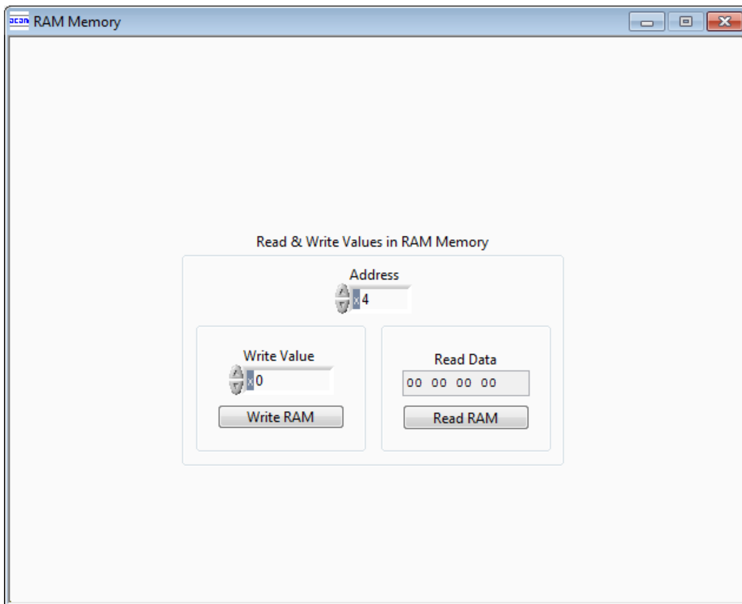
In this window the user can load firmware code (programs) and firmware data (e.g. calibration data). By pressing the “Verify” button the checksums of the two files is compared to the checksum calculated in the GP30.

After the firmware was downloaded, it is necessary to activate the CPU post-processing. This is done in the main program under tabulator “Clock control & Power & Task Sequencer”.

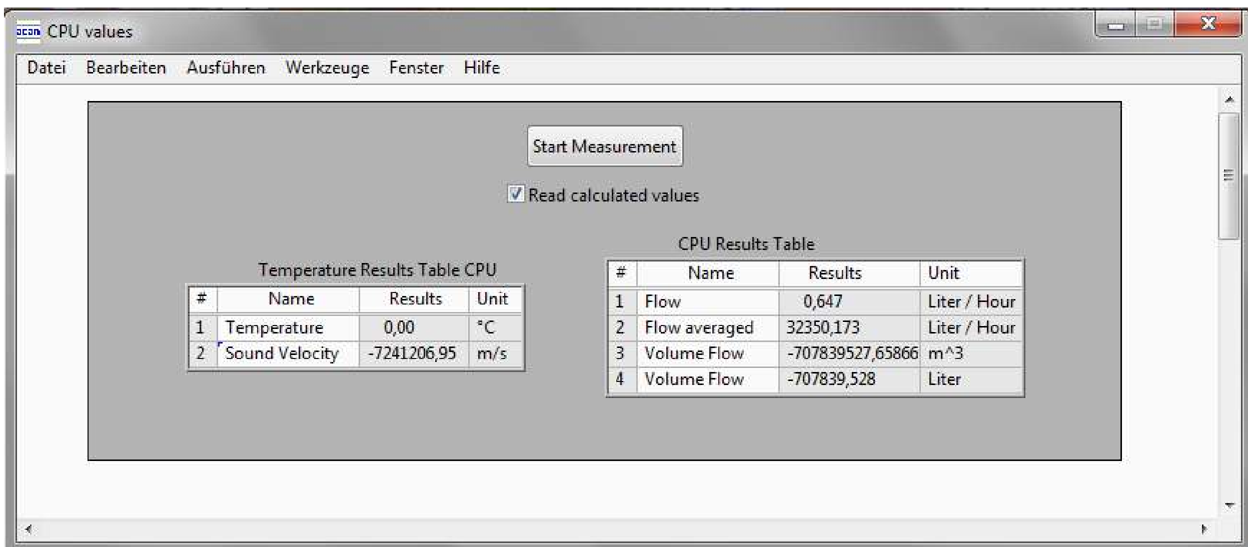


Download the new configuration before starting the next measurement.

For debugging purposes use the RAM Memory window where you can read the RAM content:



The main program has a menu item “Firmware/CPU values”. This one is firmware specific and actually for acam internal purpose only.



7.1 Assembly Programs

The TDC-GP30 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 + 2)/3)
```

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 REV_ADDRESS 3964` `CONST FW_VER + 2`

`LABEL:`	Label for target address of jump instructions. Labels end with a colon. All rules that apply to symbol names also apply to labels. Labels must be followed by an instruction. Add one nop if the label would be followed directly by ORG.	`jsub BLD_CFG;`
		`BLD_CFG:`
		`move y,16;`

`;`	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.	`; Call Address: XXX`
`org`	Sets a new origin in program memory for subsequent statements.	`org 0`
`equal`	Insert three bytes of user defined data in program memory, starting at the address as defined by `org`.	`equal 0xcfcf01`
`#include`	Include the header or library file named in the quotation marks "". The code will be added at the line of the include command. In quotation marks there might be just the file name in case it is in the same folder as the program, but also the complete path.	`#include "GP30Y_FW_v2.h"`

7.2 Basic Structure

The following code shows the basic structure of a GP30 firmware.

7.3 Example 1: Simple TOF Difference via Pulse Interface

```
;---- File     : GP30Y_A1.F1.11.01.asm

;----------- include files --------------------
#include "GP30Y_A1.F1.11.01.h"       ; header file containing memory address, constants def.
#include "GP30Y_ROM_A1.common.h"     ; Definition of ROM routine start addresses

;------------ Version  number definition ---------------------------
CONST FW_ROMVERSION_REV       0xA1
CONST FW_VERSION_NUM          0xF11101
CONST ROM_FWI                 0xf85b  ; 63579 Start address for acam's firmware init, used here
                                      ; merely for pulse interface and first hit initialisation

org        0                          ; File start

;========================== Check of CPU Requests - Main loop ============================
; This routine checks the status flags for activity requests and calls the according routines
; Inputs       :      Status flags in SRR_FEP_STF and SHR_CPU_REQ
; Output       :      all processing done and status flags cleared
; (total)unused: (all used)
; Temporary RAM : -
; Permanent RAM : RAM_R_FW_STATUS, RAM_FEP_STF
; Routines used : ROM_FWI, MK_PP, ROM_CPU_CHK
;=========================================================================================

MK_CPU_REQ:
      nop      ;--
                              ; save front end status flags in RAM_FEP_STF for remote
                              ; communication and RAM_R_FW_STATUS for processing control

      ramadr   SRR_FEP_STF    ; Front end Status Flags
      move     x, r           ; Saving the FEP Status flags for further processing
      ramadr   RAM_FEP_STF
      move     r,x            ; save a copy of SRR_FEP_STF as flags for remote com.
      ramadr   RAM_R_FW_STATUS
      and      r, 0xFFFFFC0C  ; Clearing only the bits in the RAM_R_FW_STATUS pertaining to
                              ; the SRR_FEP_STS register
      or       r, x           ; The status flags from the SRR_FEP_STS is copied to the
                              ; RAM_R_FW_STATUS Check for CPU requests: Here only Firmware init
                              ; and Post processing (others ignored)

      ramadr   SHR_CPU_REQ    ; Set RAM Address to SHR_CPU_REQ
      skipBitC r, BNR_FWI, 1  ; Check Firmware Init Flag
      goto     ROM_FWI        ; Jump to Firmware Init; return to start over ROM routine
                              ; ROM_CPU_CHK

      ramadr   SHR_CPU_REQ    ; Set RAM Address to SHR_CPU_REQ
      skipBitC r, BNR_PP, 1   ; Check User Memory Post Processing Flag
      goto     MK_PP          ; Jump to User Post Processing; return to start over ROM
                              ; routine ROM_CPU_CHK

; if the process ever gets here... (usually ROM_CPU_CHK ends execution when all CPU requests are
; cleared)then some request was missed; indicate error by saving the current SHR_CPU_REQ
      ramadr   SHR_CPU_REQ
      move     x,r
      ramadr   RAM_CPU_REQ_ERROR ; save a copy of the current CPU request register here
      move     r,x

      stop                    ; end execution without looking back, to avoid infinite loops

;############################# End of Main Program MK_CPU_REQ #########################

;============================== Post Processing - simple PI output version ==============
MK_PP:
      ramadr   RAM_R_FW_STATUS        ; Firmware Status
      skipBitC r, BNR_FLOW_CALC_REQ, 2 ; Check if a new measurement requests for calculation...
       jsub    MK_PP_DIFTOF_SCALE     ; Jump to PP-Subroutine Flow Calculation
```

```
       jsub     ROM_PP_PI_UPD          ; Jump to PP-Subroutine Pulse Interface Update

MK_PP_END:
      clrwdt                  ; Clearing watchdog
      ramadr   SHR_CPU_REQ    ; SHR_CPU_REQ
    bitclr     r, BNR_PP      ; Clear UPM Post Processing Flag directly in CPU Request Register
      goto     ROM_CPU_CHK    ; Jump Back to CPU Request Check in System Memory

;##################################### End of MK_PP

;=============== Flow Calculation - simple scaling of DIFTOF to flow version ======

MK_PP_DIFTOF_SCALE:
    nop
;----------------------- Reading and Unifying TOF raw values-----------------------

      ramadr   FDB_US_TOF_ADD_ALL_U   ; TOF Sum Up of all the configured hits
      move     y, r
      ramadr   FDB_US_TOF_ADD_ALL_D   ; TOF Sum Down of all the configured hits
      move     z, r

      ramadr   RAM_R_TOF_HIT_NO
      move     x, r
      divmod   y, x
      ramadr   RAM_R_VA5_TOF_TO_FLT_U ; TOF sum up scaled to single TOF
      move     r, y
      move     y, z                   ; Z Accu : Down measurement TOF_ADD_ALL
      ramadr   RAM_R_TOF_HIT_NO
      move     x, r
      divmod   y, x
      ramadr   RAM_R_VA6_TOF_TO_FLT_D ; TOF sum down scaled to single TOF
      move     r, y

;----------- Calculate DIFTOF: ROM_CALC_TOF_DIFF directly uses the RAM addresses from above ---

      jsub     ROM_CALC_TOF_DIFF      ; Calculate the DIFF TOF = TOF_UP - TOF_DOWN
                                      ; scaling DIFTOF to s with fd32

      ramadr   RAM_R_TDC_PERIOD       ; 125 or 250s * 10^-9 * 2^32
      move     y,r

      mult     y,x
      jsub     ROM_FORMAT1_64_TO_32BIT ; Result in x with 16 integer + 16 fractional digits
                                      ; x contains now DIFTOF in s fd 32
      move     y,0x3B9ACA00           ;scale to ns by multiplying 10^9/2^16 fd 16
      mult     y,x
      jsub     ROM_FORMAT1_64_TO_32BIT ; Result in x with 16 integer + 16 fractional digits
                                      ; apply scaling factor; DIFTOF in ns fd 16 is still in x
      ramadr          FWD_SIMPLE_SCALE ; read scaling factor from firmware data fd16
      move     y,r
      mult     y,x
      jsub     ROM_FORMAT1_64_TO_32BIT ; Result in x with 16 integer + 16 fractional digits
      ramadr          RAM_R_FLOW_LPH
      move     r, x                   ; Flow result in Lph with 16 fd
; Accumulating the measured Flow cumulatively to get Flow Volume in Cubic meter

      jsub     ROM_SAVE_FLOW_VOLUME   ; Cumulative volume result is in RAM_R_FLOW_VOLUME_INT
                                      ; and _FRACTION
MK_PP_DIFTOF_SCALE_END:
      jsubret

;############################ End of MK_PP_DIFTOF_SCALE ############################

;===================== Write Version number at the end of the writable hex code =======

      org      REV_ADDRESS
      equal1   FW_ROMVERSION_REV
      equal    FW_VERSION             ; Defined at the beginning of this file

;======================== Fill file with zeros for locked memory part ================
      org      4094
      nop
```

```
;================================== MK_END_OF_FW ======================================
;############################# End of MK_END_OF_FW ####################################
```

8 Miscellaneous

8.1 Bug Report

8.2 Last Changes

19.09.2019	Labels must be followed by an instruction . Add nop if followed by org
27.06.2019 Version 0.1	ROM Routine ROM_FILTER_FLOW with fixed length Instruction help clean up (RAMADR, NAND, skip)