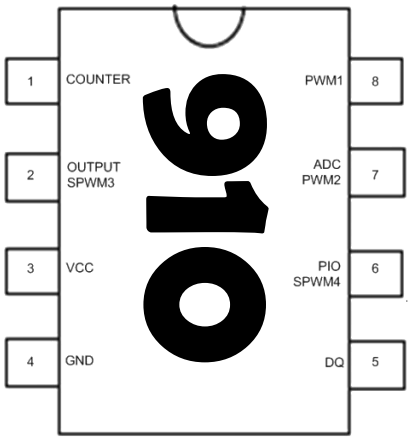
BAE0910 Multi-function 1-wire slave device

i/o, adc, pwm, rtc, counter, EEPROM, Automation Engine

Microcontroller based 1-wire slave for flexible solutions.

Main features

Pin and connections



BAE0910 SOIC-8

PIN	NAME	DESCRIPTION
1	COUNTER	Selectable rising or falling edge counter. Up to 10KHz.
2	OUTPUT/SPWM3	Simple output or SPWM3 (software Pulse Width Modulation) Capable of sinking up to 20mA
3	VCC	Power supply 2,7V to 5,5V 8mA typical consumption @ 5V
4	GND	Ground
5	DQ	1-Wire interface open drain
6	PIO/SPWM4	PIO or SPWM4 (software Pulse Width Modulation) Input mode: Configurable internal pullup/pulldown 37KΩ resistor Output mode: Capable of sourcing and sinking up to 20mA
7	ADC/PWM2/DO	Analog to Digital Converter 8 bit : 0V to 5V operating range. PWM2: True hardware Pulse Width Modulation Ddigital output
8	PWM1/DO/DI	PWM1: True hardware Pulse Width Modulation Digital input/output

Five I/O pins allows various functions:

- One to four 16 bit PWM: two hardware (PWM1 & 2) + two software(PWM3 & 4)
- One 8 to 10 bit ADC input 0 to 5V
- One 32 bit counter
- Up to 4 digital outputs (sink up to 20mA)
- Up to 3 digital inputs

Compatibility with 1-wire protocol:

- Standard speed operation: protocol implemented with low latency interrupts in background.
- Support every standard ROM commands: read rom, match rom, search rom, skip rom, resume rom, conditional search
- Unique serial number
- Family code 0xFC: future devices will use this family code. The *read_type & read_version* commands expose the model/revision.
- Control functions: see FUNCTION COMMANDS page

Physical characteristics:

- Single chip microcontroller based solution in an 8-pin SOIC.
- 5.0V supply voltage, 8mA typical consumption
- powerful 32MHz operation
- Fully functional without additional external components

Additional features

- **Firmware upgradable via 1-wire bus.**
  The chip firmware is contained in FLASH and can be upgraded directly from 1-wire bus.
- **Automation Engine: embed your programs in the device.**
  This new feature allows to define powerful autonomous behavior of the chip
- 32 bit RTC clock incrementing each second.
- PIO has configurable internal pull-up / pull-down resistor
- Counter is configurable on rising/falling edge
- 8192 bit of EEPROM storage (2x512bytes)
- 32 bytes user RAM
- PWM : Pulse Width Modulation
  - Versatile clock allows range from 2Hz to 8MHz
  - polarity selectable
  - 0% and 100% duty cycle possible
- ADC
  - works in continuous conversion mode (no start conversion command)
  - calculates average over one second (configurable from 2 to 4000 samples/sec)
  - sums every one second averages in 32 bit accumulators
  - registers maintain min and max measurements
  - configurable alarms for conditional search
  - 8 bit samples can be handled as unsigned or signed (middle offset)
  - Signed samples can accumulate on distinct counters

Typical usage

Extend the functionalities normally available on 1-wire networks by embedding local logic in the slave device. This includes but not limits to:

PWM

- very precise servo controller for toys, webcams, …
- DC Motor controller; requires power drivers like mosfet or H-drive
- Buzzer control with programmable volume and tone
- From LED dimming down to LED blinking (2Hz)
- RGB LEDs: precise control of intensity for each component of RGB LEDs
- DAC: digital to analog conversion

ADC

- Watt-meter with adc accumulators (see example)
- Solar energy meter

Counter

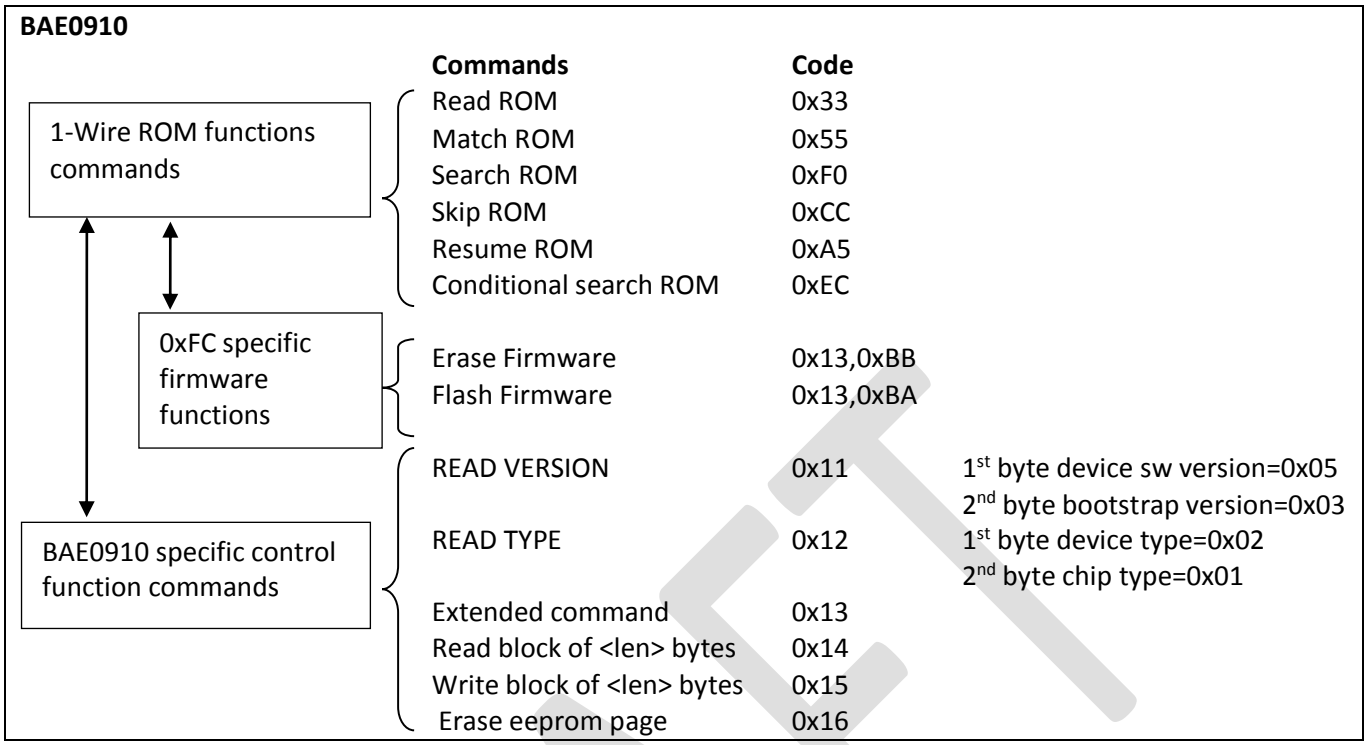
- Water-meter
- Anemometer
- RPM counter for motors
- Passage count

Automation Engine

- Closed loop control applications: motor speed , temperature, liquid level, flows …
- Alarms/security applications, access control
- HVAC applications
- Home automation projects
- Custom electronic interfaces for toys, robotics, gadgets, etc…

Table of Contents

Hierarchical structure



Absolute Maximum Ratings

Stress beyond the limits specified below may affect device reliability or cause permanent damage to the device.

Description	Value	Unit
Supply voltage Vcc	–0.3 to +5.8	V
Maximum current into Vcc	120	mA
Digital input voltage	–0.3 to Vcc + 0.3	V
Instantaneous maximum current on a single pin	± 25	mA
Operating temperature range	–40 to 85	°C

Function commands

0x11: READ VERSION

Master Mode	DATA	Description
TX	0x11	READ VERSION function
RX	<SW_VER>	1st byte device software version (currently 0x07), This is the software version of the upgradable part of the firmware. A chip with corrupted firmware or without firmware installed or forced in safe mode will answer with a SW_VER=0x00. Even number are stable releases version, while odd numvers are intermediate beta versions.
RX	<BOOTSTRAP_VER>	2nd byte bootstrap version (factory fixed, current chips are 0x03) Version of the fixed part of the firmware. (The layer that communicates on the 1-wire bus and handles firmware upgrade functions.)
RX	<CRC16>	calculated CRC16 including command byte and version bytes

0x12: READ TYPE

Master Mode	DATA	Description
TX	0x12	READ TYPE function
RX	<DEVICE_TYPE>	1st byte device type. (factory fixed) The type refers to the functionality. Type 2 for BAE0910. Type 3 for BAE0911 (planned). When firmware erased/corrupted/in safe mode, this field report the “expected” device_type or 0xff for chips that allow firmware upgrade to “change personality”
RX	<CHIP_TYPE>	2nd byte chip type (factory fixed), This is the microcontroller model/package. Chip type= 0x01 for the MC9S08SH8, 8 pin package soic8
RX	<CRC16>	calculated CRC16 including command byte, and type bytes

0x13: EXTENDED COMMAND

Master Mode	DATA	Description
TX	0x13	Extended COMMAND function
TX	<LEN>	Length of payload
TX	<ECMD>	Extended command byte
TX	<payload>	data buffer accompanying the ecmd (<LEN> bytes)
RX	<CRC16>	calculated CRC16 including command byte, ecmd, len, data bytes
TX	0xBC	If CRC16 correct , master TX 0xBC, and device execute the command else master TX RESET.

0x14: READ MEMORY

Master Mode	DATA	Description
TX	0x14	READ MEMORY function
TX	<TA1>	starting byte location low address
TX	<TA2>	starting byte location hi address
TX	<LEN>	Length of data to read (max 32 bytes)
RX	<data bytes>	<LEN> Bytes beginning at specified address
RX	<CRC16>	calculated CRC16 including command byte, address, data bytes

Read memory command allows reading at either register locations (TA =0x0000 to 0x007F) or EEPROM data (TA=0xE000 to 0xE3FF)

0x15: WRITE MEMORY

Master Mode	DATA	Description
TX	0x15	WRITE MEMORY function
TX	<TA1>	starting byte location low address
TX	<TA2>	starting byte location hi address
TX	<LEN>	Length of data to write (max 32 bytes)
TX	<data bytes>	Bytes beginning at specified address up to <LEN>
RX	<CRC16>	calculated CRC16 including command byte, address, len, data bytes
TX	0xBC	If CRC16 correct , ->then master TX 0xBC, device store data to destination registers ->else master TX RESET.

Write memory command allows writing either register locations (TA =0x0000 to 0x007F)or EEPROM data (TA=0xE000 to 0xE3FF).
Normally EEPROM need to be erased prior to writing. The erasing operation set every bytes to 0xFF (all bits are set to 1). Eeprom writing operation is in fact only capable clearing bits. It is possible thus to write any bytes that are still at 0xFF value. The only way to revert a cleared bit to 1 is to erase the whole page.
Writing 32 bytes on EEPROM requires 1,5ms.

0x16: ERASE EEPROM PAGE

This command erases a page of 512 bytes at specified starting location (aligned on 512 bytes boundary)
The resulting content of the page becomes filled with 0xff bytes.

Master Mode	DATA	Description
TX	0x16	ERASE EEPROM PAGE command function
TX	<TA1>	Starting byte location low address (always 0x00)
TX	<TA2>	Starting byte location high address Least significant bit must be zero to respect 512 bytes boundary.
RX	<CRC16>	Calculated CRC16 including command byte, TA1 and TA2 bytes.
TX	0xBC	If CRC is correct, master TX 0xBC to confirm erase command. The erase cycle requires 25ms to finish; other bus activity is allowed during this time.

On BAE0910 chip, there are 2 pages available:
0xE000 to 0xE1FF →TA1=0x00, TA2=0xE0
0xE200 to 0xE3FF →TA1=0x00, TA2=0xE2

Extended commands

0xBB: INITIATE FLASH FIRMWARE

This command is mandatory before issuing a flash firmware command. A consistency check is performed on the chip before accepting a new firmware. When the initiate flash firmware is accepted and confirmed by the master, an erase operation will be performed.
Only the flash memory related to the device functionality is affected; Bootstrap, Rom functions and upgrade code are always preserved.

Master Mode	DATA	Description
TX	0x13	Extended command function
TX	0x20	Length of data bytes being sent (32 bytes)
TX	0xBB	ECMD for INITIATE FLASH FIRMWARE
TX	<validation data>	First 32 bytes from firmware file.
RX	<CRC16>	calculated CRC16 including command byte, ecmd, len, data bytes or bad CRC16 if validation data is not valid
TX	0xBC	If CRC is correct, master TX 0xBC to confirms erase command. The erase cycle requires 180ms to finish; other bus activity is allowed during this time.

Structure of <Validation data>:

```
byte DeviceVersion ;      //SW_VER of device functionality
byte BootstrapVersion ;   //BOOTSTRAP_VER of bootstrap part
byte DeviceType ;         //Identification of device functionality
byte ChipType;            //Identification of hardware chip (0x01= SOIC8@32MHz)
word firmware_start_addr;//Starting location of firmware - currently 0xE400
word firmware_end_addr;   //Ending location of firmware - currently 0xF3FF
byte targetSN[8];         //FF is a wildcard of the corresponding byte in ROM_SN
...                       //begin of firmware data
```

Fields in **bold** have to match with internal chip info to allow upgrade validation

0xBA: FLASH FIRMWARE

This flash firmware is only possible directly after a valid erase command.
The firmware is divided in two parts:
Bootstrap: including 1wire protocol stack, rom commands and firmware function.
Functional code: servicing functions commands and controlling I/O pins.
The flash operation installs new functional code in the BAE0910 device. This allows upgrading or changing the personality of the device.
The bootstrap section cannot be upgraded via the 1-wire bus.
If the upgrade operation fails, the device will stay in firmware upgrade mode and will accept re-flashing again with a correct firmware.

Master Mode	DATA	Description
TX	0x13	Extended command function
TX	0x20	Length of data bytes being sent (32 bytes)
TX	0xBA	ECMD for FLASH FIRMWARE
TX	<data bytes>	Sequential data from firmware file. (32 bytes blocks)
RX	<CRC16>	calculated CRC16 including command byte, ecmd, len, data bytes
TX	0XBC	If CRC16 correct , ->then master TX 0xBC and program cycle is busy during the next 2ms. ->else master TX RESET and restart upgrade from the last failed bloc.
…	…	Repeat sequence (from TX 0x13) until end of firmware file (typically 4096 bytes)

If the new firmware is corrupt or buggy, the device could become unresponsive after upgrade.
In that case, refer to Safe mode firmware restore section.

Register configuration

Register address map Part I

The behavior of the device is controlled by writing registers.

Address	Size	Read /Write	content type	register	description	b7 128	b6 64	b5 32	b4 16	b3 8	b2 4	b1 2	b0 1
0x000	2	RESERVED				MSB							
0x001													LSB
0x002	1	R/W	Stable	ADCC	configuration bits for ADC				ADCEN	10BIT	STP	GRP	OFS
0x003	1	R/W	Stable	CNTC	counter configuration bits						LOW	STP	POL
0x004	1	R/W	Stable	OUTC	OUTPUT configuration bits				OUTEN	DS			
0x005	1	R/W	Stable	PIOC	PIO configuration bits				PIOEN	DS	PD	PE	DD
0x006	1	R/W	Stable	ALARMC	alarm mask configuration bits	US2	US1	RTC	PIO	AN	AP	CPS	CT
0x007	1	R/W	Stable	RTCC	Real Time Clock configuration bits				TRM3	TRM2	TRM1	TRM0	STP
0x008	1	R/W	Stable	TPM1C	TPM1 configuration bits (PWM1 &	POL			PWMDI		PS2	PS1	PS0
0x009	1	R/W	Stable	TPM2C	TPM2 configuration bits (PWM2/ADC)	POL		IN_ENA	PWMDI		PS2	PS1	PS0
0x00A	2	R/W	Stable	PERIOD1	Period for PWM1 & 3	MSB							
0x00B													LSB
0x00C	2	R/W	Stable	PERIOD2	Period for PWM2, PWM4 & ADC	MSB							
0x00D													LSB
0x00E	2	R/W	Stable	DUTY1	PWM1 duty cycle	MSB							
0x00F													LSB
0x010	2	R/W	Stable	DUTY2	PWM2 duty cycle	MSB							
0x011													LSB
0x012	2	R/W	Stable	DUTY3	PWM3 duty cycle	MSB							
0x013													LSB
0x014	2	R/W	Stable	DUTY4	PWM4 duty cycle	MSB							
0x015													LSB
0x016	2	R	Volatile	ADCAP	ADC positive avg of the last 1second period (sum of last second samples	MSB							
0x017													LSB
0x018	2	R	Volatile	ADCAN	ADC negative avg of the last 1second period (unsigned short)	MSB							
0x019													LSB
0x01A	2	R/W	Volatile	MAXAP	maximum positive value encountered from ADCAP	MSB							
0x01B													LSB
0x01C	2	R/W	Volatile	MAXAN	maximum negative value encountered from ADCAN	MSB							
0x01D													LSB
0x01E	2	R	Volatile	CPS	Count Per Second	MSB							
0x01F													LSB
0x020	4	R/W	Volatile	ADCTOTP	ADC TOTAL for positive values	MSB							
0x021													
0x022													
0x023													LSB
0x024	4	R/W	Volatile	ADCTOTN	ADC TOTAL for negative values	MSB							
0x025													
0x026													
0x027													LSB
0x028	4	R/W	Volatile	RTC	32 bit RTC	MSB							
0x029													
0x02A													
0x02B													LSB
0x02C	4	R/W	Volatile	COUNT	32 bit Counter	MSB							
0x02D													
0x02E													
0x02F													LSB
0x030	1	R/W	Stable	OUT	set/clear to change OUTPUT pin								D
0x031	1	R/W	Volatile	PIO	set/clear/read PIO pin								D
0x032	1	R	Volatile	ADC	current ADC conversion (8bits)	MSB							LSB
0x033	1	R	Volatile	CNT	read COUNTER pin state								D
0x034	1	R/W	Volatile	ALARM	Alarmed bits	US2	US1	RTC	PIO	AN	AP	CPS	CT
0x035													
0x036	2	R/W	Volatile	PC0	Program counter of script instance 0	MSB							
0x037													LSB
0x038	2	R/W	Volatile	PC1	Program counter of script instance 1	MSB							
0x039													LSB
0x03A	2	R/W	Volatile	PC2	Program counter of script instance 2	MSB							
0x03B													LSB
0x03C	2	R/W	Volatile	PC3	Program counter of script instance 3	MSB							
0x03D													LSB
0x03E	2	R	Volatile	ADC10	10 bit ADC sample	MSB							
0x03F													LSB

MSB: most significant bit, LSB least significant bit

The registers in the range 0 to 63 are “triggerable” registers. Writing to them triggers automatically an appropriate action in the chip. If the user attempts to write an invalid value in the registers, the register content is constrained to the acceptable range.

On multi-byte registers, the trigger is activated when last byte is written (byte at higher address).

Register address map Part II

Address	Size	Read Write	Content Type	Register	Description	b7 128	b6 64	b5 32	b4 16	b3 8	b2 4	b1 2	b0 1
0x040	2	R/W	Stable	ALAP	Alarm level for positive ADC	MSB							
0x041													LSB
0x042	2	R/W	Stable	ALAN	Alarm level for negative ADC	MSB							
0x043													LSB
0x044	2	R/W	Stable	ALCPS	Alarm when Counter Rate (CPS) above this value	MSB							
0x045													LSB
0x046	4	R/W	Stable	ALCT	Alarm when Counter (Total) above this value	MSB							
0x047													
0x048													
0x049													LSB
0x04A	4	R/W	Stable	ALRT	Alarm when RTC above this value	MSB							
0x04B													
0x04C													
0x04D													LSB
0x04E													
0x04F													
0x050													
0x051													
0x052													
0x053													
0x054													
0x055													
0x056	2	R/W	Volatile	RESETCNT	Number of reset seen on 1wire bus	MSB							
0x057													LSB
0x058	2	R/W	Volatile	SELECTCNT	Number of device select	MSB							
0x059													LSB
0x05A	2	R/W	Volatile	STALLEDCNT	1wire bus stalled (DQ low more than 3ms)	MSB							
0x05B													LSB
0x05C	2	R/W	Volatile	OVRUNCNT	Overrun error count, when the device is too busy to respect 1-wire timing	MSB							
0x05D													LSB
0x05E	2	R/W	Volatile	MAXCPS	maximum CPS encountered	MSB							
0x05F													LSB
0x060	1	R/W	Volatile	USERA	8bit User variable	MSB							LSB
0x061	1	R/W	Volatile	USERB	8bit User variable	MSB							LSB
0x062	1	R/W	Volatile	USERC	8bit User variable	MSB							LSB
0x063	1	R/W	Volatile	USERD	8bit User variable	MSB							LSB
0x064	1	R/W	Volatile	USERE	8bit User variable	MSB							LSB
0x065	1	R/W	Volatile	USERF	8bit User variable	MSB							LSB
0x066	1	R/W	Volatile	USERG	8bit User variable	MSB							LSB
0x067	1	R/W	Volatile	USERH	8bit User variable	MSB							LSB
0x068	2	R/W	Volatile	USERI	16bit User variable	MSB							
0x069													LSB
0x06A	2	R/W	Volatile	USERJ	16bit User variable	MSB							
0x06B													LSB
0x06C	2	R/W	Volatile	USERK	16bit User variable	MSB							
0x06D													LSB
0x06E	2	R/W	Volatile	USERL	16bit User variable	MSB							
0x06F													LSB
0x070	4	R/W	Volatile	USERM	32bit User variable	MSB							
0x071													
0x072													
0x073													LSB
0x074	4	R/W	Volatile	USERN	32bit User variable	MSB							
0x075													
0x076													
0x077													LSB
0x078	4	R/W	Volatile	USERO	32bit User variable	MSB							
0x079													
0x07A													
0x07B													LSB
0x07C	4	R/W	Volatile	USERP	32bit User variable	MSB							
0x07D													
0x07E													
0x07F													LSB

MSB: most significant bit, LSB least significant bit

The registers are stored in big endian byte order (High byte at lower location).

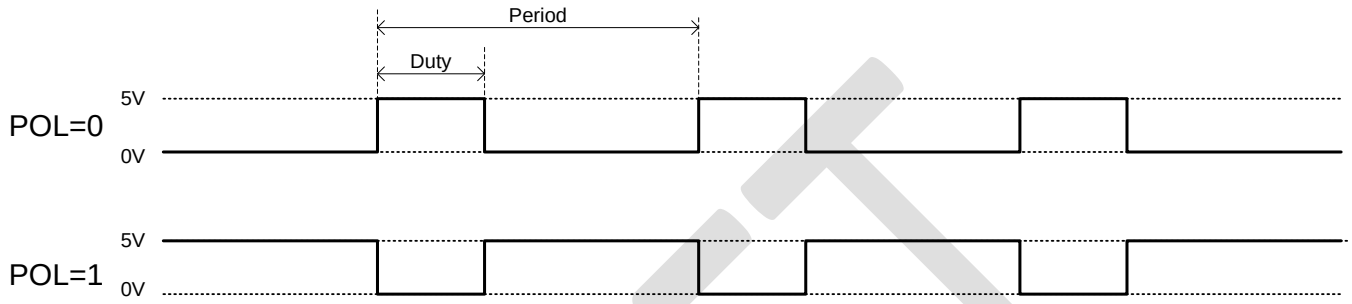
Writing to a Read only register has no effect.

PWM features

Pulse Width Modulation function is an efficient way to deliver proportional power to electrical devices by controlling the 'ON' duration of the cyclic signal. It is used to control light intensity, motor rpm, sound tone and volume. PWM signal are also used to control RC/servo motors with high precision.

The 3 main aspects when defining a PWM signal are:

- The Period: controlled by both PS2:PS1:PS0 prescaler bits and PERIODx registers
- The Duty cycle: controlled by DUTYn 16bits registers
- The polarity: controlled by POL bit from TPMxC register



This chip has two independent Timer Pulse Modulator (TPM) subsystems, each of them control two PWM channels (one hardware and one software).
TPM1C and PERIOD1 controls the frequency of PWM1 & PWM3*.
TPM2C and PERIOD2 controls the frequency of PWM2 & PWM4*.

* *PWM3 and PWM4 are software based PWMs.*

Software PWM3 share the same pin of OUT.
Software PWM4 share the same pin of PIO.
To enable PWM2, PWM3 or PMW4, disable respectively ADC, OUT or PIO functions.

PWM related registers

REGISTERS	Description
TPM1C, TPM2C	Clock and polarity configuration bits.
PERIOD1 PERIOD2	Modulo counter for corresponding TPM This register defines the period between pulses.
DUTY1 DUTY2 DUTY3 DUTY4	Duration of the pulse for corresponding PWM. if DUTY == 0, this will not produce any pulse (0%) if DUTY >= PERIODx, this produce a constant level (100%) any value in between produce a repetitive pulse of specified width.
ADCC:ADCEN	When this bit is set, ADC function is enabled, PWM2 is disabled
OUTC:OUTEN	When this bit is set, OUT function is enabled, PWM3 is disabled
PIO:PIOEN	When this bit is set, PIO function is enabled, PWM4 is disabled

PWM configuration registers

TPMxC	Clock / polarity configuration bits for PWM
POL	Polarity bit for PWM1 and PWM2 0 : Pulse signal is high (5v), idle is low (0v) 1 : Pulse signal is low (0v), idle is high (5v) When PWMDIS is set, POL control the output pin:0=low, 1 =high
PWMDIS	PWM disable: This bit allow to use the corresponding PWM pin as digital output (or digital input see in_enable) controlled by the POL bit.
IN_ENABLE	On PWM1, pin8, the IN_ENABLE bit allow to use the corresponding pin as digital input. The POL bit represent the level present on the pin. PWMDIS bit must also be set.
PS2:PS1:PS0	Prescaler factor setting. This 3-bit field selects one of 8 division factors for the PWM clock input. See table below

Prescaler table

PS2:PS1:PS0 prescaler settings	Divisor	TPM Clock	resolution		max period (16bits) PERIODx=65535 (resolution x 65535)		Frequency range		
0	1	16MHz	0,0625	µs	4,10	ms	245Hz	–	8MHz
1	2	8MHz	0,125	µs	8,19	ms	123Hz	–	4MHz
2	4	4MHz	0,25	µs	16,38	ms	62Hz	–	2MHz
3	8	2MHz	0,5	µs	32,77	ms	31Hz	–	1MHz
4	16	1MHz	1	µs	65,54	ms	16Hz	–	500KHz
5	32	500KHz	2	µs	131,07	ms	8Hz	–	250KHz
6	64	250KHz	4	µs	262,14	ms	4Hz	–	125KHz
7	128	125KHz	8	µs	524,29	ms	2Hz	–	62,5KHz

When software PWM is in use, the maximum frequency for that TPM is limited to 10Khz.

Using PMW 1 & PWM2 as digital output

It is possible to use PWM as digital output by setting duty to 0 to produce constant low output, and duty>=period to produce constant high output.
However, this method may introduce uncontrollable delays before the state is actually changed. The change will be effective after the end of the current pwm cycle.
When you require direct control of a PWM pin with low latency, it is possible to disable normal pwm operation by setting the PWMDIS bit.

When PWM is disabled and IN_ENABLE is not set, the level of the pin is directly controlled by the POL bit.

You may find an example of controlling a stepper motor this way BAE0910-prototyping-board-usermanual.pdf

Using PMW 1 as digital input

It is possible to use PWM1 pin as digital input by disabling PWM with the PWMDIS bit and setting IN_ENABLE bit
When PWM1 pin is configured as input, the POL bit represent the state of the level of the pin.

ADC configuration

The analog signal provided to the ADC pin should be in the range 0 to 5V. The 8 bit ADC works in continuous mode.

Working with simple ADC conversions.

As the ADC register is updated continuously, there is no start conversion command needed, just read the ADC register to obtain latest sampled voltage present on ADC pin. Reading ADC register is immediate and has not to wait for a conversion completion.
The value read on ADC always reflects the voltage present on the pin 7 of the chip.
To read an externally applied voltage, ADC has to be enabled by setting ADCC=16. This bit ADCEN bit in fact disable PWM2 function and put the ADC pin in high impedance mode. If ADCEN not set, PWM2 function drive the pin and reading ADC in such condition return the voltage level imposed by PWM2.

ADC Registers.

REGISTER	Description
ADCC	ADC configuration bits, see below
ADC	Instantaneous 8 bit adc sample (read only)
ADCAP ADCAN	read only 16bit registers containing the last second averages for **P**ositive and **N**egative samples: ADCA=ADC x freq / 64
ADCTOTP ADCTOTN	32 bit Accumulators for every second's averages (Read/Write registers) ADCTOT**P** accumulate positives measures, ADCTOT**N** for negatives

ADCC	ADC configuration bits
ADCEN	ADC hi impedance enable (PWM2 disable) 0 : PWM2 is enabled, ADC input works but return voltage from driven by PWM. 1 : PWM2 is disabled. ADC input pin is in high impedance and follow voltage externally applied.
10BIT	10 bit mode 0 : 10 bit disable, (work in 8 bit mode) 1 : 10 bit ensable, ADC10 contain the sample When 10 bit is enabled, averaging ant totalisation registers are not significative.
OFS	Offset control bit 0 : conversions are considered 8bit unsigned and added into ADCAP 1 : conversion values above 128 are positive, below are negative. (subtract 128) Positives values are averaged on ADCAP Negatives values are averaged on ADCAN
GRP	Group result control bit. 0 : totalize ADCTOTx with corresponding ADCAx 1 : totalize ADCTOTP with both ADCAP and ADCAN
STP	0 : totalization enabled (ADCTOTx+=ADCAx). 1 : Stop accumulation of adc samples in ADCTOTx (ADC and ADCAx are still updated)

Reading instantaneous ADC sample does not require configuring the totalizer function described below.

Working with advanced ADC.

The advanced ADC functions implemented are the **averager** and the **totalizer**.
The averager produce an average per second by summing adc samples at a configurable frequency to an internal 24 bit register. After each second, this value is divided by 64 and stored on ADCA[P/N]
The totalizer add theses every second averages into the ADCTOT[P/N] registers for metering applications.

The averaged values could also help to compensate relatively low 8bit sampling resolution to a higher interpolated resolution up to 14bit (at the cost of 1 second time resolution).

ADC samples with offset.

Some analog sources need to represent positive and negative values. With a range of 0 to 5V, this is accomplished with adding a 2.5v offset (the middle). In such case, reading 2.5V represents 0, upper values are positives, and lower are negatives.
To cope with such representations, the OFS bit on register ADCC control the behavior of average calculation, while the GRP bit control the destination registers used for summations.

ADC averages: ADCAx

Based on PS2:PS1:PS0 prescaler bits and PERIOD2 value (see PWM section), you can control time between samples summation. Fine control on the number of samples summed every second allows precise calibration of the averaged value.
The 8bit ADC value is summed at a maximum rate of 4096 Hz. After one second you get an ADC measure that is multiplied by 4096 or 2^{12}. This produces a 20 bits value (2^8 x 2^{12}). The 14 most significant bits are then stored on ADCAx registers.

Use TPM2 and PERIOD2 to control the summation frequency.
With TPM Clock = 16MHz, 1Hz steps are possible in the 244Hz to 4000Hz frequency range.

$$\text{Summation frequency} = \frac{\text{TPM Clock}}{\text{PERIOD2}}$$

Use DUTY2 to control the sample instant at which the sample is read and accumulated. Summation is disabled if DUTY2 set to zero or above PERIOD2.

When ADCEN is set, the maximum frequency allowed is 4096 accumulation per second. The AE engine will process AE user code more slowly when high rate is selected.

PIO configuration

The PIO pin could be used for digital input and digital output. When used as input, an internal pullup or pulldown resistor could be enabled.

Address	Size	Read /Write	content type	register	description	b7 128	b6 64	b5 32	b4 16	b3 8	b2 4	b1 2	b0 1
0x005	1	R/W	Stable	PIOC	PIO configuration bits				PIOEN	DS	PD	PE	DD
0x031	1	R/W	Volatile	PIO	set/clear/read PIO pin								D

PIO pin is configurable as INPUT or OUTPUT via PIOC register:

PIOC	PIO configuration bits
DD	Data direction: select INPUT / OUTPUT mode 0 : Input (output driver disabled) 1 : Output driver enabled
PE	Internal Pull Enable - determines if the internal pull-up or pull-down device is enabled If PIO configured as output, this bit has no effect and the internal pull device is disabled. 0 : Internal pull-up/pull-down device disabled 1 : Internal pull-up/pull-down device enabled
PD	Pull Down Enable - select a pull-up or pull-down device if enabled. 0 : A pull-up device is connected 1 : A pull-down device is connected
DS	Output Drive Strength selects between low and high output drive. When configured as input, this bit has no effect. 0 : Low output drive strength selected. 1 : High output drive strength selected.
PIOEN	PIO/PWM4 selection bit 0 : normal PIO not active, PWM4 is enabled 1 : normal PIO mode enabled

PIO register read/control PIO pin level.

Mode	PIO	PIO data bit (READ/WRITE)
Input	D	Read the level present on the PIO pin 0 : less than 0.35 x Vcc volts is present to input pin 1 : more than 0.7 x Vcc is present to input pin
Output	D	Output Data bit 0 : output pin is driven low to GND (sink up to 20mA) 1 : output pin is driven high to Vcc (source up to 20mA)

Writing to an input pin has no effect.
Reading an output pin reports the last data written.

RTC configuration

The RTC is a 32 bit read/write register that increments once every second. It starts at 0 at power-up.
The RTC value could be set by user to represent timestamp.

RTC is configurable via RTCC.

RTCC	RTC configuration bits
STP	RTC stop bit 0 : RTC is incrementing every second. (default) 1 : RTC is stopped.
TRM0-3	Clock trimming bits These 4 bits allows to increase clock speed. Set RTCC register to a even value in the range 0 to 30 (ie 0,2, 4, 6...30) to increase the clock from 32MHz to 32,615 MHz

The RTC clock is factory calibrated. The precision is ±0.5%

COUNTER configuration

CNT is basically an input pin that increment a 32bit COUNTER on falling/rising edges of signals presented to the pin 1 of the chip.
However, it is possible to use this pin as an output by simply forcing the pin low via the LOW bit in CNTC register.

CNTC	COUNTER configuration bits
POL	Edge select bit 0 : falling edge count. 1 : rising edge count.
STP	Stop Counter bit 0 : counter is incrementing on edge events. 1 : counter stops counting, pin state remains available.
LOW	open drain output operation 0 : normal counter mode pulled up by external resistor (CNT pin is an input) 1 : CNT pin is forced low (acting as an output pin)

CNT: The value of this register reflect the current level present on the pin, writing has no effect.
COUNTER: This 32 bit register increments on edge events on the pin, writing this register is allowed.
CPS: this 16 bit register reflect the current count per second.
CPSMAX: this 16 bit register reflect the maximum count per second seen since last reset.

The CNT pin has no internal pullup/pulldown resistors, external resistor should be added if needed.

Warning: the CNT pin does not contain a clamp diode to Vcc, installing an external Schottky diode is recommended to protect the device from transients.

OUTPUT configuration

OUTPUT pin is configurable via OUTC register to select between OUTPUT/PWM3 mode.

OUTC	OUTPUT configuration bits
OUTEN	OUTPUT/PWM3 selection bit 0 : OUTPUT not active, PWM3 is enabled 1 : normal OUTPUT mode enabled (default)
DS	Output Drive Strength selects between low and high output drive. 0 : Low output drive strength selected. (4mA) 1 : High output drive strength selected. (20mA)

OUT register control the output pin level.

OUT	OUTPUT data bit (READ/WRITE)
D	Output Data bit 0 : output pin is driven low to GND (sink up to 20mA) 1 : output pin is driven high to Vcc (source up to 20mA)

Important notice: *on power up, OUT may not be held low by an external device. Forcing OUT to low level during poweron will disable the chip. A disabled chip does not operate until power cycled again.*

ALARM configuration

The alarm registers controls how the device responds to conditional search 1-Wire command.

The state of various alarm conditions is maintained in ALARM register where each bit represents a specific condition.
The ALARMC register enable corresponding alarm bits to participate to conditional search.
Clearing the bit neutralize participation to conditional search.

Address	Size	Read Write	Content Type	Register	Description	b7 128	b6 64	b5 32	b4 16	b3 8	b2 4	b1 2	b0 1
6	1	R/W	Volatile	ALARMC	alarm mask configuration bits	US2	US1	RTC	PIO	AN	AP	CPS	CT
52	1	R/W	Volatile	ALARM	Alarmed bits	US2	US1	RTC	PIO	AN	AP	CPS	CT

REGISTER	DESCRIPTION
ALARM	ALARM bits. See ALARM table for details
ALARMC	Alarm configuration bits. Bits set to 1 will enable response to conditional search ROM command
ALAP	Positive alarming level average compared with ADCAP
ALAN	Negative alarming level average compared with ADCAN
ALCPS	Alarm level for CPS (count per second) exceeded
ALCT	Alarm level when COUNTER reached this value
ALRT	Alarm level when RTC reached this value

ALARM/ ALARMC	ALARM/ ALARMC configuration bits
CT	COUNTER reached a certain value (ALCT) 0 : COUNTER < ALCT 1 : COUNTER >= ALCT This bit is only cleared when the condition is no longer met.
CPS	COUNTER rate per second exceeded a certain value (ALCPS) 0 : rate not exceeded 1 : rate exceed This bit remains set until manually cleared.
AP	ADCAP exceeded a certain value (ALAP) 0 : ADCAP not exceeded 1 : ADCAP exceed This bit remains set until manually cleared.
AN	ADCAN exceeded a certain value (ALAN) 0 : ADCAN not exceeded 1 : ADCAN exceed This bit remains set until manually cleared.
PIO	PIO pin state changed 0 : PIO unchanged 1 : PIO change was detected This bit remains set until manually cleared.
RTC	RTC reached a certain value (ALRT) 0 : RTC < ALRT 1 : RTC >= ALRT This bit is only cleared when the condition is no longer met.
US1	Bit available for user alarming via AE.
US2	Bit available for user alarming via AE

Automation Engine - AE

Description

The BAE chips embed an Automation Engine that allows creating automatic (re)actions directly within the chip.
This is implemented as a virtual CPU that process up to four flow of instruction in parallel.
The AE programs are stored in non volatile chip memory (eeprom) which is remotely programmable under master control.

Utilization

The perspectives allowed by a programmable device are numerous and not restricted to the examples proposed here:

- Thermostat – control against user defined setpoints
- Motor speed control with closed loop feedback
- Push button momentary switch with timed relay output
- Programmed sequential cycles with servo and end of course detection
- Liquid flow measurement with alarm on volume/duration/flow
- Night/day current consumption counter
- twilight switch
- Aquarium monitoring – temperature, pumps, lights, food…
- Intrusion detection with embedded conditions to trigger alarms, tamper detection,…
- Status reporting with different blink patterns, tone or melody
- Front door bell with Morse code sesame.
- …

Features

- 1Kbyte eeprom program storage
- Four independent processes
- Semi-interpreted byte code - open source assembler provided
- Execution speed: up to 10K instructions per second
- Full R/W access to BAE registers
- 32 byte dedicated stack space per process
- Four 32bit shared user registers
- Four 16bit shared user registers
- Eight 8bit shared user registers
- Arithmetic operations on 8, 16, 32 bit operands
- autoexec feature at chip power on
- Simplified direct control from 1-wire master
- Stability enforced by running processes as separate virtual cpu instances

How AE is working?

As the BAE chips implement the various functions via registers exposed to 1-wire, controlling operation of the device is a matter of reading/writing to the appropriate register when condition(s) is(are) met.

The device registers are a key aspect of the BAE chips. They not only control electrical devices connected, but registers are also the communication windows between the master and the embedded processes.
Operations possible with AE are:

- Register affectations
- Timing operations
- Arithmetic operations
- Compare operations
- Stack operations
- Sub routine branch-return
- Conditional flow
- Parallel task control

1-wire input-output operation is as simple as reading-writing dedicated user registers
Automation require no more than reading-writing device registers.

Performance

AE is implemented as a virtual CPU that handles four contexts. The opcodes are fetched from the four instruction flows and are executed in round robin sequence to allow a pseudo parallelism. Blocking instruction 'WAIT' put the corresponding flow on hold giving more slices to active processes.

The virtual cpu has an average frequency of 50KHz. This may seems a quite low compared to MHz class today cpu's, but this is largely sufficient to reach millisecond level real-time processing. Moreover, time critical operations like 1-wire communication, pwm, counter inputs, adc, … are handled in background by the hardware. AE is only the glue logic used to build state machine, or closed loop control.

On the performance perspective, instructions could be divided in 4 categories

- 1 cpu cycles: Flow control instructions, excluding 16bits CALL,RET,JMP
- 2 cpu cycles: 8 bits operations
- 3 cpu cycles: 16 bits operations, including CALL,RET,JMP
- 4 cpu cycles: 32 bits operations

When only one process is running, around 10K instr/s are processed,
For 2 concurrent processes, 7,5K instr/s each (total 15K instr/s)
For 3 concurrent processes, 5,8K instr/s each (total 17,5K instr/s)
For 4 concurrent processes, 4,7K instr/s each (total 19K instr/s)

Instruction set

mnemonic	object code	operands on stack	type	
ADD	1	2	u/s	#define m_ADD1 4#define m_ADD2 5#define m_ADD4 6
SUB	2	2	u/s	#define m_SUB1 8#define m_SUB2 9#define m_SUB4 10
MUL	3	2	u/s	#define m_MUL1 12#define m_MUL2 13#define m_MUL4 14
DIV	5	2	u	#define m_DIV1 20#define m_DIV2 21#define m_DIV4 22
SDIV	6	2	s	#define m_SDIV1 24#define m_SDIV2 25#define m_SDIV4 26
MOD	7	2	u	#define m_MOD1 28#define m_MOD2 29#define m_MOD4 30
SMOD	8	2	s	#define m_SMOD1 32#define m_SMOD2 33#define m_SMOD4 34
AND	9	2	u/s	#define m_AND1 36#define m_AND2 37#define m_AND4 38
OR	10	2	u/s	#define m_OR1 40#define m_OR2 41#define m_OR4 42
XOR	11	2	u/s	#define m_XOR1 44#define m_XOR2 45#define m_XOR4 46
NOT	12	1	u/s	#define m_NOT1 48#define m_NOT2 49#define m_NOT4 50
NEG	13	1	s	#define m_NEG1 52#define m_NEG2 53#define m_NEG4 54
XTU	14	1	u	#define m_XTU1 56#define m_XTU2 57#define m_XTU4 58
XTS	15	1	s	#define m_XTS1 60#define m_XTS2 61#define m_XTS4 62
RDU	16	1	u/s	#define m_RDU1 64#define m_RDU2 65#define m_RDU4 66
JMP	17	1	u	#define m_JMP1 68#define m_JMP2 69#define m_JMP4 70
CALL	18	1	u	#define m_CALL1 72#define m_CALL2 73#define m_CALL4 74
RET	19	0		#define m_RET1 76#define m_RET2 77#define m_RET4 78

Params:

opr8	immediate byte
opr16	immediate word
opr32	immediate unsigned long (32bit)
opr8a	immediate address to register / stack (prefixed with @)
opr16a	immediate address to the code code (prefixed with @)
rel	relative adress (signed byte)
(stack)	implicit parameter for stack operation

Flags:

C	carry flag
Z	zero bit flag
N	negative bit flag
V	overflow bit
S	supend

Each mnemonic operating on opr8a and stack has three variations: .B, .W, .L for byte, word & long respectively.
Example:

```
INC.B   @96  // will increment the byte at location USERA
PUSH.W  @30  // push to stack the content of word at location CPS
CLR.L   @40  // will clear the 32bit RTC timer
```

An include file 'bae0910.inc' is available for a more user friendly programming. Theses produce exactly the same code but are more readable.

```
INC.B   B_USERA   //by convention, byte register names are prefixed by B_
PUSH.W  W_CPS     //by convention, word register names are prefixed by W_
```

```
CLR.L   L_RTC     //by convention, long register names are prefixed by L_
```

USR opcode

An opcode has been added to call custom functions for specific user requested developments. Maximum 8 such functions can be implemented.
Currently following functions have been added:

```
USR 0 // 1wire reset signal on PIO, allows to access 1W sensors connected on PIO
USR 1 // 1wire send/receive on PIO: to receive, send $FF via stack
```

Example: reading temperature on a DS18B20 connected on PIO

```
read:
      PUSH.B 0    //prepare one byte on stack for USR commands
      USR 0       //RESET
      BIT.B @-1,1 //verify presence pulse
      BNE nodevice // ds18B20 did not reply with presence pulse
      SET @-1,$CC //skip rom
      USR 1       //send rom command on PIO
      SET @-1,$44 //convert command for DS18B20
      USR 1       //send command
      WAIT 16     // wait on second before reading conversion
      USR 0       //RESET
      SET @-1,$CC //skip rom
      USR 1       //send rom command on PIO
      SET @-1,$BE //read scratchpad command for DS18B20
      USR 1       //send command
      SET @-1,$FF //$FF for read 1st byte of scratchpad c
      USR 1       //read
      PUSH.B $FF  // second byte
      USR 1       //read
      POP.W W_USERI //temperature result stored on USERI
      BRA read    //infinite loop on convert
:nodevice
      CLR.W W_USERI // clear USERI (no temp avail)
      BRA read          //infinit loop on convert
```

AE stack

There is a distinct stack for each four processes. The stack is 32 byte deep and is used for:

- CALL/RET branching,
- parameter passing
- local variable allocation
- arithmetic operations

It is important to note that the stack grows upward. I.e. The stack pointer begins at zero and is incremented to the next available byte on each push.

Subroutine

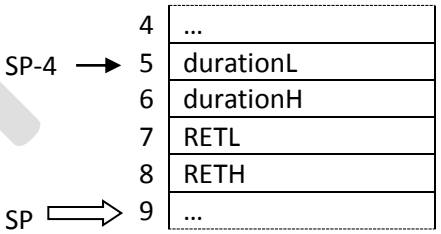
Each CALL consumes 2 bytes to store return address that are POPed on RETurn. Before leaving a subroutine, ensure that SP (stack pointer) is pointing to the same position as at the entry. See below for subroutine branching.

Parameter passing

If the caller needs to pass a parameter to the subroutine, the caller push the value in the stack before CALL operation. The called subroutine access the parameter via opr8a negative offset.
After return, the caller has responsibility to free the stack from the parameter

Example: caller call a longwait subroutine where the word param is duration in seconds.

```
     ...
     PUSH.W   600       // store param to wait 10 minutes
     CALL     longwait  // call subroutine
     AIS      -2        // free the two bytes (word) from the stack
     ...
longwait:
     WAIT     16        // wait 1 second
     DEC.W    @-4       // decrement the word at stack position -4
     BNE      longwait  // loop until param reached zero
     RET                // return to caller
```



Local variables

Allocating local variable is a simple matter of pushing content in the stack. To access this content, use the negative offset in the opr8a operand. Local variable is preferred to global user registers to avoid inconsistency between distinct processes/functions.
Before leaving the subroutine, the called sub has the responsibility to free allocated stack space.

Arithmetic operations

Stack is also used for arithmetic operations. This is a very efficient way to solve even complex calculations in the reverse Polish notation. I.e. the operands are pushed in the stack and the operator is then applied on implicit data: USERA= 5 + (1 + 2) can be written down like this:

```
PUSH.B   5         // push byte operands
PUSH.B   1
PUSH.B   2
ADD.B              // add 1+2 : pop 2, pop 1, add them and push 3 on the stack
ADD.B              // add 5+3  = pop 3, pop 5, add them and push 8 on the stack
POP.B    @B_USERA  // pop the result ant store it on USERA register
```

AE addressing

The registers are addressed via **opr8a** operands. The format of the operand is an 8bit value preceded by a '@' symbol to help compiler to distinguish between addresses and immediate values.
As the register memory space is 0 to 127, only the seven lsb bits are significant.
The msb has a special meaning: when set, it is a negative value that represents a relative stack address.

AE branching

Within a process, AE programs are executed sequentially. The PC (program counter) points to the instruction to be executed, when the instruction at PC is finished, PC is incremented to the next instruction that has to be executed at his turn then PC is incremented again. AE continues this sequence until an END instruction is reached. The branching instructions allow altering this normal sequence by modifying PC to branch to other part of the code.

Absolute branching: JMP

The parameter given to the branch instruction is the new value to set in the program counter (PC). Doing so instruct AE to continue execution at the new location provided. The parameter needed by the branch instruction is a word value capable to refer to any absolute position within the code. To avoid the hassle to calculate this new PC value, the compiler resolve this word value based on a simple label defined in the program.

```
JMP  label // this allows to jump to any location of AE code with no restriction on the distance
...

label:
...
```

An absolute branching require three bytes of code memory (1 byte for instruction+ 2 bytes for address).

Relative branching: BRA, BEQ, BGR, BLO, …

Instruction for relative branching allows conditionally skipping small portion of code or looping to near location. They are called relative because the program counter is incremented by the signed value provided as parameter of the branching instruction. The parameter is coded on a single signed byte. The displacement range is -128 to 127 allow branching both forward and backward relative to current location. To avoid the hassle to calculate this displacement value, the compiler calculates this based a simple label defined in the program.
See instruction list table for details on branching conditions.
If the label is unreachable due to distance between branch instruction and the label, the compiler stops with the following error:

```
BRANCH outside relative range -128..127! on line 128 : labelname
```

To solve such issue use the JMP instruction where BRA is not possible.
For conditionals branch's, as no conditional JMP exists, use an opposite test that skips an unconditional JMP.

A relative branching instruction require only two bytes of code memory (1 byte for instruction+ 1 bytes for displacement).

Branching to subroutine CALL, RET

When designing sequential programs, some portions of code need to be repeated to execute the same action at different places in your program. Such code portions could be coded as a subroutine.
A subroutine is a normal code that terminate by a RET instruction meaning RETurn to caller. The sub routine is placed outside the main execution loop and is called when need from any place within your program logic.
The CALL instruction is used to branch to a subroutine by providing the label of this subroutine. It is similar to a JMP instruction but also push in the stack the position of the instruction (PC is a word = two byte) that follows the CALL to allow RETurning to the original sequence when the subroutine has terminated his work.
To avoid overflowing/underflowing the stack, is important that the CALL - RET flow instructions are correctly disposed (for each CALL instruction executed a corresponding RET instruction is required). It is however possible to have a subroutine that CALL another subroutine up the stack depth (32bytes/2 = maximum 16 level) as long as long as each subroutine terminate with their own RET.
See below for CALL example.

A CALL instruction consume three bytes of code memory (1 byte for instruction+ 2 bytes for subroutine address).
A RET instruction consume only one byte of code memory (1 byte for instruction) as the returning address is stored in stack at runtime.

Controlling the processes

AE Programs stored in the chip are easily controlled by the master: pc0 to pc3 represents the program counter of the corresponding process. Writing 0 to this register stops the process. Writing a value (re)start the process at the location instructed.

REGISTER	DESCRIPTION
PC0	Program Counter of process 0
PC1	Program Counter of process 1
PC2	Program Counter of process 2
PC3	Program Counter of process 3

Autoexec of the process 0 on power on

One special feature is to have the device to start automatically the process zero on power on. This is achieved putting a NOP on location zero (i.e. at the very first byte of the page.0)
This feature is useful to have the device initialized with some user settings at startup, but also to have an autonomous behavior initiated without requiring master attention.

Writing code for BAE devices

Building ae-asm commandline assembler

The BAE0910 is capable of executing AE compiled code saved in embedded eeprom. You can find in the brain4home.eu download area the ae-asm.tgz archive containing source code to build on a Linux platform. The archive contains a README file with needed instructions.

Defining eeprom page to use in source code

A compilation directive #eeprom start,end instruct compiler to generate code from start page to end page

Defining location of the code in the page(s)

A compilation directive #org position instruct compiler to generate code following this directive at the location specified

Register definitions in source code

AE programs rely heavily on device registers. To avoid hard coding of register addresses within source code, an include file is provided with definition of BAE0910 registers. This is realized by including the registers definition file.
Regigisters names are prefixed with their type (B_ for byte, W_ for word, L_ for long) which remember you to use the adequate mnemonic suffix.
Example:

```
#include "/opt/ae_asm/include/bae0910.inc"  //include register definition
#eeprom  0,1          // start page, end page
                      // 0,1 will generate a binary file of 2x512 byte from page.0 to page.1
#org $0               // start code at location 0x00
         NOP
         CALL  sub
         END
#org $100             // put the subroutine at location 0x100
sub:     CLR.L L_RTC // to clear RTC time counter
         RET
```

Sample AE programs

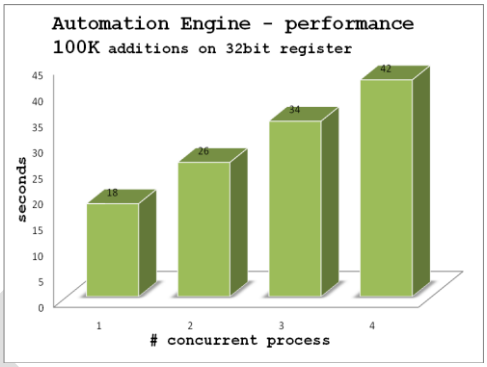
The following listing was used to measure performance:

```
// heavy.asm
// load loop measurement program for BAE0910
// Dec. 2009 Pascal Baerten

#include  "bae0910.inc"                 // register definition file
#eeprom   0,0                           // start_page, end_page
#org      $00                           // code start at location 00

main:
          NOP
four:                                   //entry point for 4 process measurement
          START     1,process
three:                                  //entry point for 3 process measurement
          START     2,process
two:                                    //entry point for 2 process measurement
          START     3,process
one:                                    //entry point for 1 process measurement
          CLR.L     L_USERO
          SET.L     L_USERN,100000
          SET.L     L_USERM,L_RTC       //save current time in USERM
bcl:
          DEC.L     L_USERN
          BNE       bcl
          PUSH.L    L_RTC               // put end time in stack
          PUSH.L    L_USERM             // put start time in stack
          SUB.L                         // diff end-start (seconds elapsed)
          POP.L     L_USERM             // store diff in USERM
          SUSPEND 1                     //
          SUSPEND 2                     // stop other processes
          SUSPEND 3                     //
          END                           // all  done.

process:
          INC.L     L_USERO
          BRA               process
          END
```




```
$ ae_asm heavy.asm -o heavy.bin
processing file heavy.asm  - pass 1,
processing file heavy.asm  - pass 2,
Code start at :0x0000
Code end   at :0x002f
Size reserved :0x0200
Symbol Table  :
            main LABEL    0x0000
            four LABEL    0x0001
         process LABEL    0x002a
           three LABEL    0x0005
             two LABEL    0x0009
             one LABEL    0x000d
             bcl LABEL    0x0018
ouptut to heavy.bin
$ echo 1 > /tmp/ow/FC.000000000003/eeprom/erase.0
$ cp /tmp/heavy.bin /tmp/ow/FC.000000000003/eeprom/page.0
$ echo 12 > /tmp/ow/FC.000000000003/910/pc0
$ cat /tmp/ow/FC.000000000003/910/userm
          18
$ echo 9 > /tmp/ow/FC.000000000003/910/pc0
$ cat /tmp/ow/FC.000000000003/910/userm
          26
$ echo 5 > /tmp/ow/FC.000000000003/910/pc0
$ cat /tmp/ow/FC.000000000003/910/userm
          34
$ echo 1 > /tmp/ow/FC.000000000003/910/pc0
$ cat /tmp/ow/FC.000000000003/910/userm
          42
```

Generate the binary file

Erase eeprom & store program

Start process0 at label “one”

Start process0 at label “two”

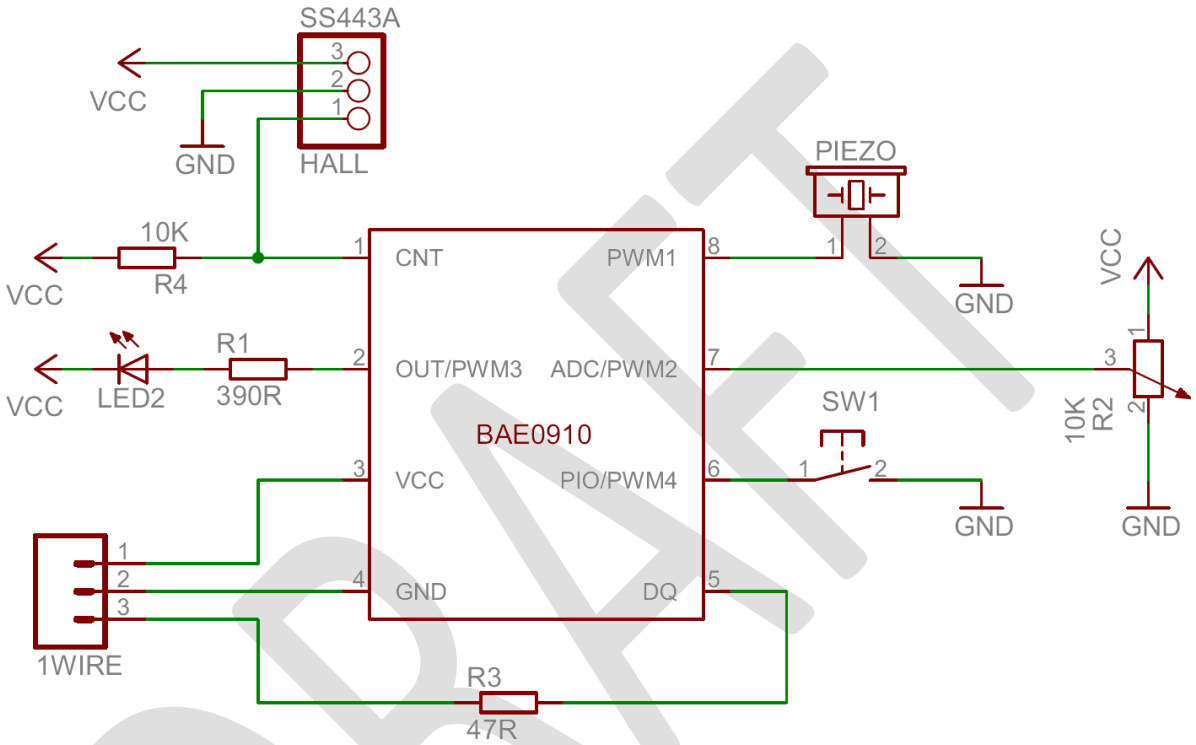
Start process0 at label “three”

Start process0 at label “four”

Schematics & examples

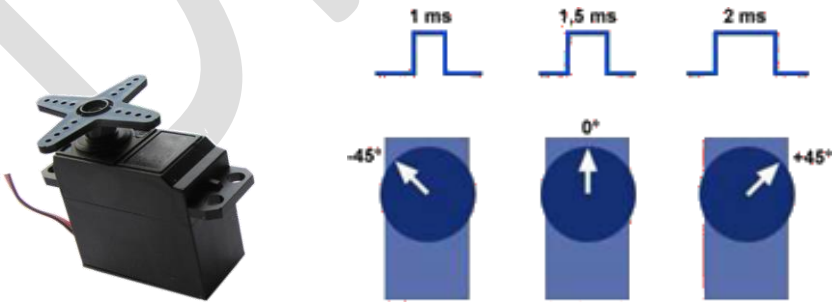
This example proposes to use the chip with following functions:

- COUNTER gets pulsed signal from an HALL effect sensor and count rotations of a magnet.
- PWM1 output a tone alarm
- ADC gets consign information from a potentiometer
- PIO gets input from a tactile switch
- OUT drive a LED with on/off or in dimmer mode with PWM3 configuration.



Controlling servo motors

A particular application of PWM is the control of servo motors used in robotics and RC models.

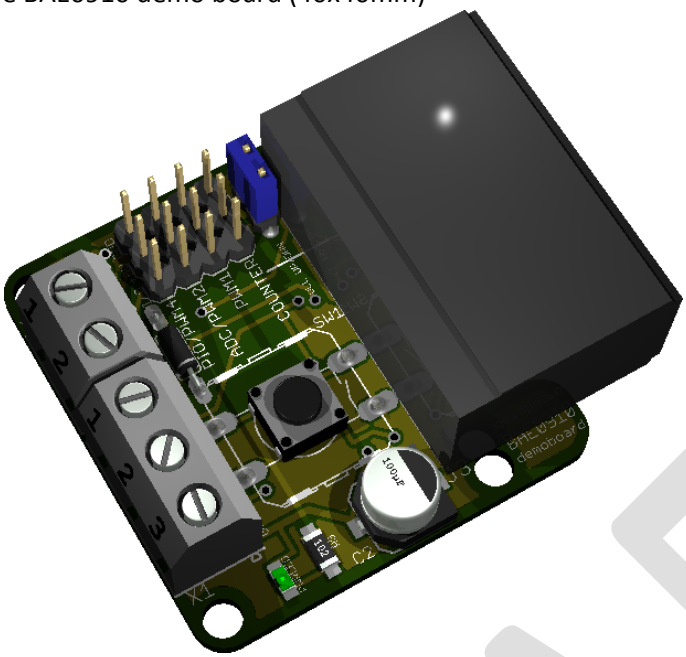


A servo motor requires a 5Vdc supply and a control signal in the form of a repeated pulse every 20ms (50Hz). The width of the pulse defines the shaft position. Such signal is easily produced by PWM function.

This behavior can easily be implemented with PWM, no specific power drivers are needed allowing you to connect the servos directly to the TTL pins of BAE0910.
Example for a servo connected on PWM1:
Set TPM1c=4 → prescaler PS2:PS1:PS0 to 1:0:0 (1µs resolution) , Clear POL bit,
Set PERIOD1=20000 → period of 20000µs = 50Hz,
Set DUTY1 a value in the range of 1000 to 2000 to control precise servo position (1000 steps).

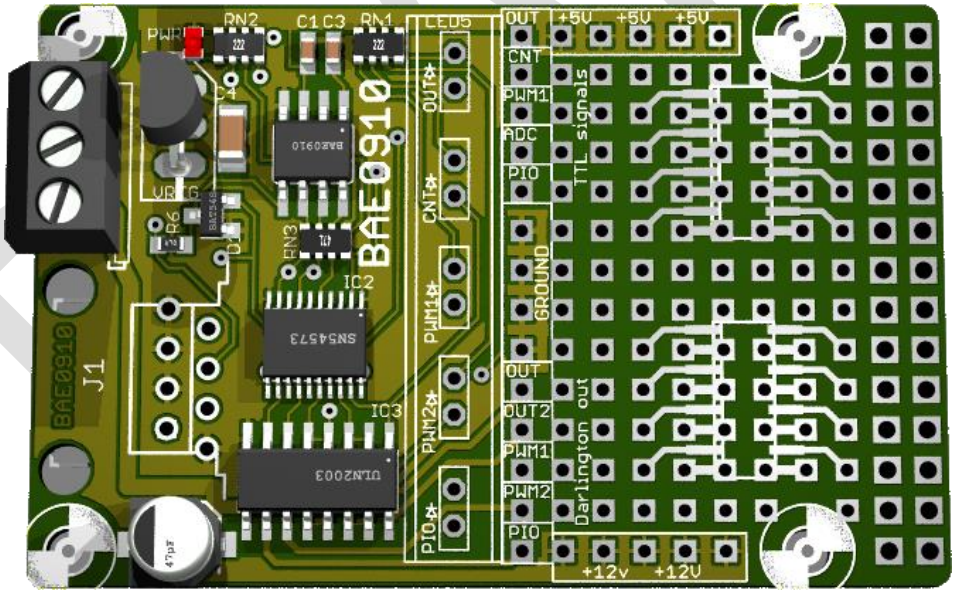
Some board examples:

The BAE0910 demo board (40x40mm)



The BAE0910 Tiny board 13x36 mm

The BAE0910 Prototyping board 40x60 mm



Recovering a failing chip

The BAE chips are microcontroller based. They are factory programmed to present specific 1-wire functionalities. The embedded software is composed of a bootstrap and a firmware. The bootstrap part is static and cannot be altered without special hardware. This part contains 1-wire communication stack and the firmware update code. The second part, the firmware, is upgradable and contains the functionalities advertised by the device.
There are some situations that could alter the device behavior.

Question & Answer

Q. Firmware upgrades failure.
The device functionality is assured by the upgradable part of the firmware a failure during upgrade process will alter normal behavior of the chip.
A. In case of failure during upgrade, the chip automatically restarts in firmware update mode waiting for a new firmware to be re-installed, just restart the firmware procedure to restore full functionalities

Q. Firmware bug.
Even if we carefully test released firmwares, the risk to encounter a software bug in cannot be totally excluded. Also, less stable beta versions will be proposed to testers.
A. If such a problem is suspected, re-install a known stable firmware version (downgrade is always possible). If the device even refuses to respond to firmware commands, see below to force firmware update mode.

Q. Device is not responding to firmware command.
A. The device could be forced to firmware update mode with the following manipulation: maintain the DQ pin low, then power up the chip, then releases DQ. This will force the device to start in firmware upgrade mode. You dispose of 40 seconds to start a new firmware upload.

Q. Out of range settings.
The settings related to frequencies are the more subject to put CPU under heavy stress, particularly software PWM and ADC sampling frequency. Limit checks would normally prevent to put the device in an unresponsive state. But using multiple resources at the limits in unfavorable situations like weak network reliability could put the chip on the knees.
A. Power cycling the device will solve the situation. This could become a concern if the offending settings are automatically set via autoexec feature from user EEPROM. For such situations, a special reset firmware file is available in the download section. Install this firmware with recovery procedure. This reset firmware erases all user data and firmware stored in the chip. Once erased, re-flash the virgin chip with a valid firmware.

Q. Out of frequency external signal.
BAE0910 device has a COUNTER input pin that trigger software action in edge (level transition) of the external signal connected to the corresponding pin. A frequency out of range could consume every available CPU cycle. Such situation could prevent to communicate normally with the chip.
A. Remove the offending signal to revert to a stable situation.

Q. Inactive mode.
If the chip is powered on with the OUT pin forced low, the device will be put in inactive mode with all pins in high impedance mode. In that mode, the is totally disabled. If OUT pin is used, ensure that the signal is either in high impedance or pulled high at power on.
A. Disconnect the circuit that force the OUT pin low and cycle power on the device.

Q. Out of range electrical signal.
A. Such situation should be avoided with adequate isolation circuit and protection components. Exceeding maximum electrical characteristics could cause permanent damage to the device.

Q. Breaking the secured flash code.
It is not possible to access the internal code even with adequate microcontroller programming tools.
A. Attempting this will definitively erase the firmware and void the device.

Support

Online support is available via the forum on www.brain4home.eu and via the discussion list.
To subscribe, list-subscribe@brain4home.eu

Availability

Chips and boards can be ordered online on www.brain4home.eu

Condition of use

The BAE chips are intended for hobbyist usage and are not approved for use where it constitute or may constitute a danger to human life or health.

Terms of license

The software embedded in the chips is protected by copyright laws. Customer is not allowed to reverse engineer, decompile, or disassemble the embedded software.

About the author

Pascal Baerten is primarily an IT consultant with technical background in automation. He followed A2 technical studies until 1985 where he played with CNC machines and pneumatic automates. Graduated in Computer Sciences from the Robert Shuman High school in Belgium in 1989, his thesis was titled “A terminal emulator” where he mastered serial communication and networking programming.
His first computer was a Sinclair ZX81, where he learned the basics of exploiting very constrained computing resources in assembler. Later, a Commodore 64 opened the way to interfacing computers with electronic toys.
Since 1990 he developed network based resource sharing solutions in assembler and C.: Telex server, Minitel server, mainframe front end, mail server, print server, text2speech telephone server, database gateway, IM server …
As skilled networking/server architect, he is working as IT consultant for large financial companies since 1997.
In parallel, developments in home automation have contributed to accumulate some experience with microcontrollers and embedded computing.

Credits

Special thanks to Paul Alfille for integrating support of this device in OWFS project and for the constant feedback he provided during development of the device.

Revision history

Revision #	Date	Description
0.1	Oct 15, 2009	Initial draft
0.5	Nov 22, 2009	Added eeprom functionalities,
0.61	Dec 1, 2009	Proofreading, typo corrections.
0.62	Dec 29, 2009	Added AutomationEngine section.
0.63	Feb 15, 2010	Correction sink only on OUT
0.64	Apr 5, 2010	Added new feature for RTCC & OUTC, Added more information on AE operation
0.65	Sep 9, 2010	Updated for consistency with firmware v8 Added feature on PWM1 & PWM2 to act as digital OUTPUT Required for instantaneous control of the pins (during stepper motor cycle for example).
0.66	Jun 26, 2011	Updated chip illustration to match real chip label
0.70	May 1, 2012	New USR command documented Added support for 10bit ADC
0.71	Aug 30, 2014	PWM1 (pin8) now support digital input since fw version 0x0A