HSA8000 (IXC2) User Guide
Digital Power Processor

Configurable Analog Front End™

EnSilica 32-bit RISC processor

Contents

Contents 2
Description 3
Functional Block Diagram 4
Functional Description 6
Analog Interface 6
Analog inputs ANx_CMPx (x = 0 to 3) 7
Analog inputs CMPx (x = 4, 5 and 7) 8
Analog inputs AN4, AN5, AN6, and AN7_DACVR 10
Analog inputs AN10P/N and AN11P/N 11
Analog inputs AN12_ACP and AN13_ACN 12
Analog to digital converters ADC0, ADC1 and ADC2 13
Analog DACs 21
Temperature sensor 21
Digital Interface 22
CPU 22
Interrupts 23
ROM and RAM 25
Memory map 25
Switching Engine 26
I/O Block 72
Capture (Timing Measurements) block 49
Compare (Timing Generator) 51
AC PLL 52
Serial Peripheral Interface 60
UART interface 61
I^2C serial interface 65
Watchdog 66
Math Accelerators Block 67
Timer 69
Digital DAC 72
DEBUG 76
Documentation Support 80

Description

The HSA8000 is a digital power processor. One IC offers high-speed analog peripherals, digital accelerators, event control, and digital processing. Its flexibility and performance enable designers to meet demanding compliance standards. Industrial, automotive, and renewable energy applications can benefit significantly from the enhanced performance and reduced component count it offers. The solution addresses many power conversion applications. It easily fits into advanced topologies for AC-DC inverters, battery chargers, and isolated DC-DC converters.

The HSA8000 has an advanced mixed-signal architecture. The core is a 32-bit RISC 50 MHz micro-processor. A rich set of high-performance digital power peripherals supports the core. Communications, data memory, and general-purpose inputs and outputs (GPIO) are also provided.

The HSA8000 is a fully software-programmable platform. Programming enables control, monitoring and optimization. Those features allow a design solution to meet aggressive requirements. It also allows for easily differentiated products for competitive markets. Huada bundles a software development environment with application-specific evaluation hardware to enable customers to achieve faster time-to-market.

The HSA8000 arose from extensive experience in power systems design. Those systems frequently require optimization of multiple voltages through control loops with adaptive dead-time control of multiple power trains and phases. For example, Totem-Pole PFC, LLC, and Interleaved PFC need such complex control schemes. Here, the core architecture optimizes processor usage by using high speed analog peripherals, digital accelerators and a high performance PLL. The peripherals control high speed loop functions leaving the digital core processor free to maintain low-speed functions. Those functions include slower control loops, protection, optimization and housekeeping. The HSA8000's peripheral set is the industry's most complete single-chip offering. It includes configurable high-speed voltage/current sensing, and high-speed comparators (10nS), a variety of high-performance ADCs, DACs, programmable filtering, a multi-event interrupt-based timing engine, a high performance digital PLL, and PWM control for up to 8 power devices are integrated into a single digital power processor.

Functional Block Diagram

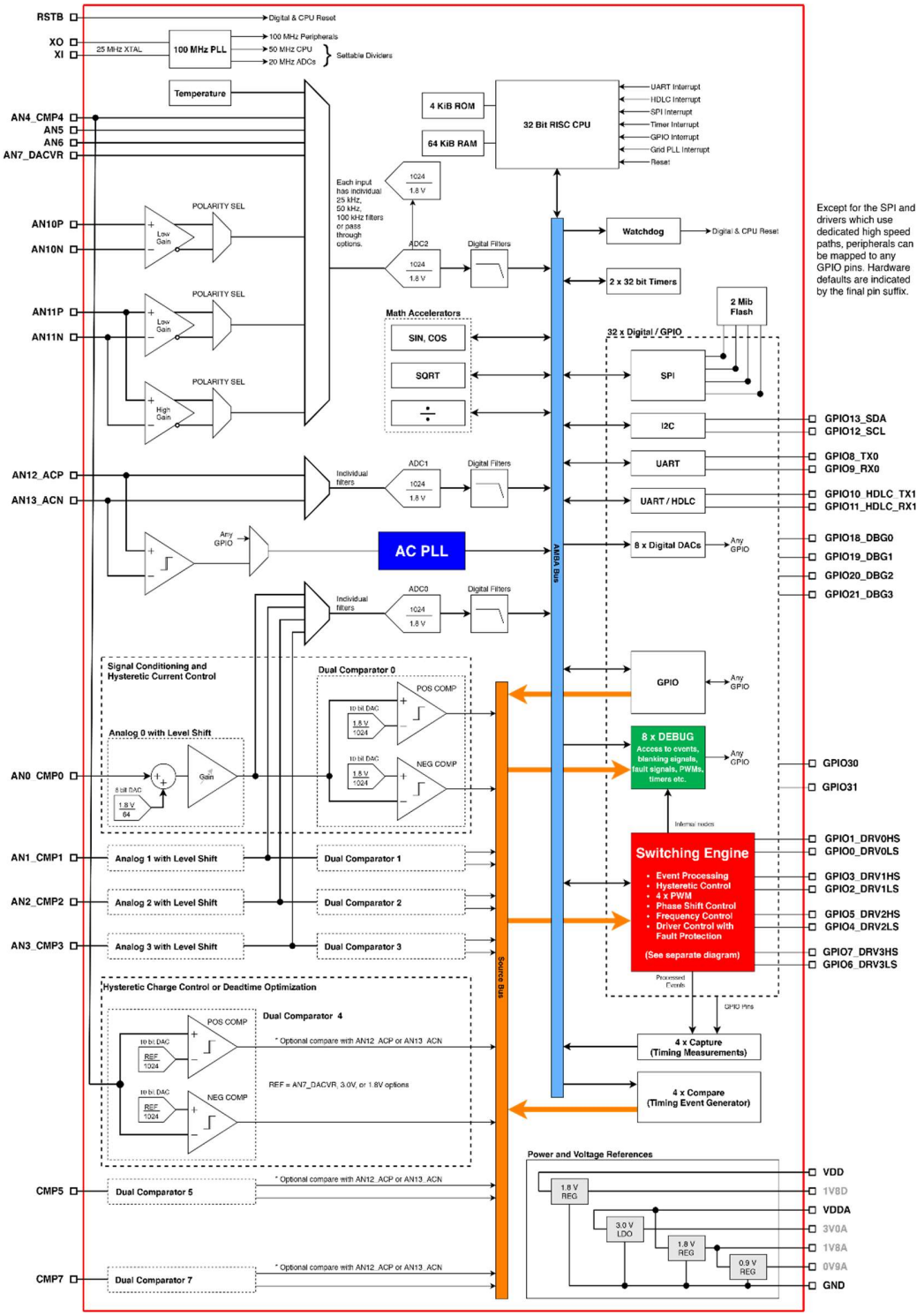


Figure 1 Functional block diagram

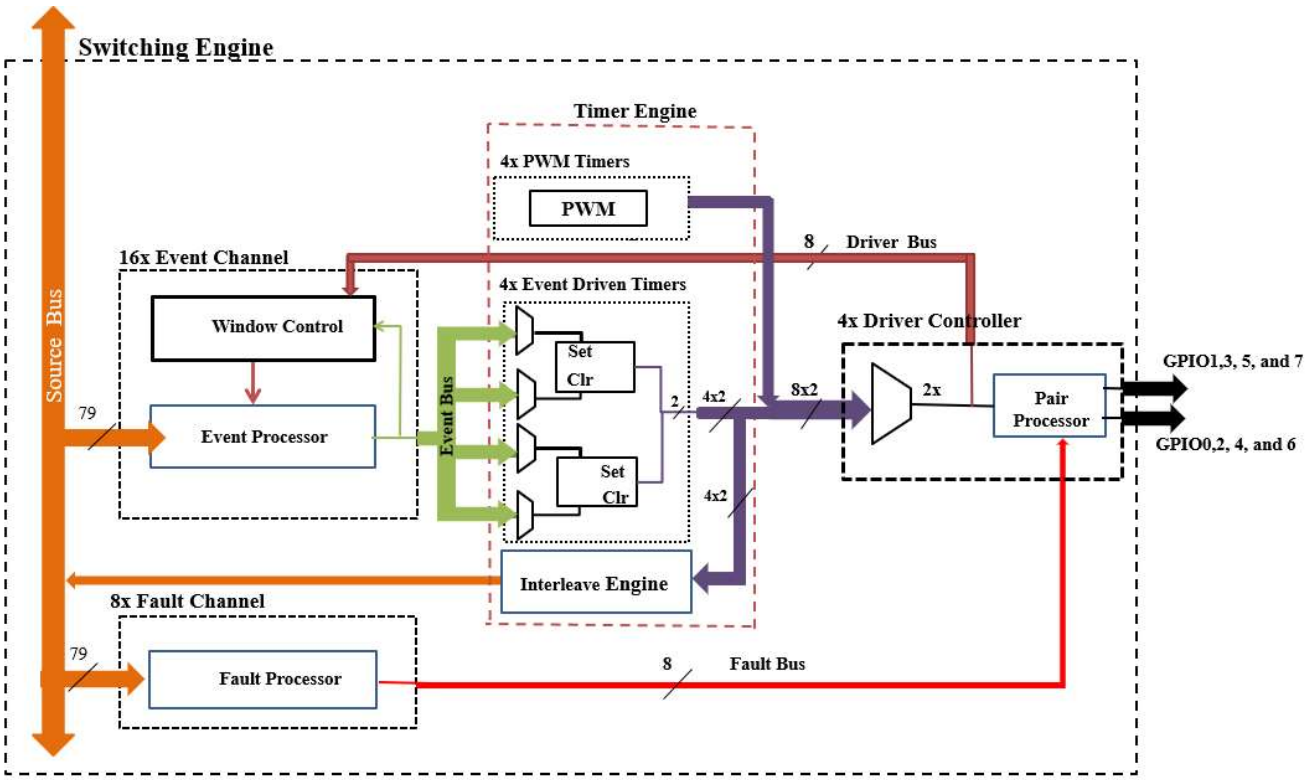


Figure 2 -High level block diagram of HSA8000 Switching Engine

Functional Description

The HSA8000 is a mixed-signal integrated circuit optimized for power conversion using digital control methods. The analog inputs can create timing events using high speed comparators while monitoring voltages and currents using three independent ADCs. The digital peripherals of which the Switching Engine is the heart can drive the gate signals of complex power applications. The functional diagram, in Figure 1, gives an overview of the HSA8000. The 32-bit RISC core oversees the configuration and monitoring of the peripherals while implementing control algorithms and state machines not possible in purely analog circuits.

Analog Interface

The HSA8000 is a highly integrated IC with rich power control-centric analog features:

- 18 analog input pins
- 1 sixteen-channel, 10-bit, 1.4 MS/s ADC with digital filters
- 2 four-channel, 10-bit, 1.4 MS/s ADCs with digital filters
- 17 10 ns fast comparators for event generation and fault detection
- 16 10-bit analog DACs for comparators internal references
- 1 10-bit analog DAC for debagging
- 4 6-bit analog DACs for level shifting
- 2 differential high-speed current sensing amplifier interfaces
- 1 differential high gain amplifier
- 14 programmable 4^{th} order anti-alias low-pass filters
- 1 25 kHz low pass filter
- Internal temperature sensor

All comparator outputs are connected to the Event Bus, whereas all ADCs and DACs are connected to the AMBA bus. HSA8000 has 18 analog inputs that can be used for sensing the controls signal for different power applications.

Analog inputs ANx_CMPx (x = 0 to 3)

These are dual-function analog inputs feeding ADC0 and two high speed comparators as shown in Figure 3. The level shift and gain stage allow small signals to be scaled to the 0 - 1.8V input range of both the ADC0 and comparators. The shifting is done by the level shift DAC. The amplifier has four gains: 1, 2, 4 or 8. To reduce the impact of the switching ripple common for the power supply signals, the ADC0 input can be optionally filtered by a 4th order low pass filter 25 kHz, 50 kHz, 100 kHz or by passed.

For performing hysteretic current or voltage control or fault detection, the amplified signal is also sent to the dual comparator block. The block has two comparators. Each comparator has its own DAC reference which can be set or controlled through software to implement a dynamic behavior and advanced control methods. The comparators' outputs are sent to the Source Bus and to SWE_DEBUG.

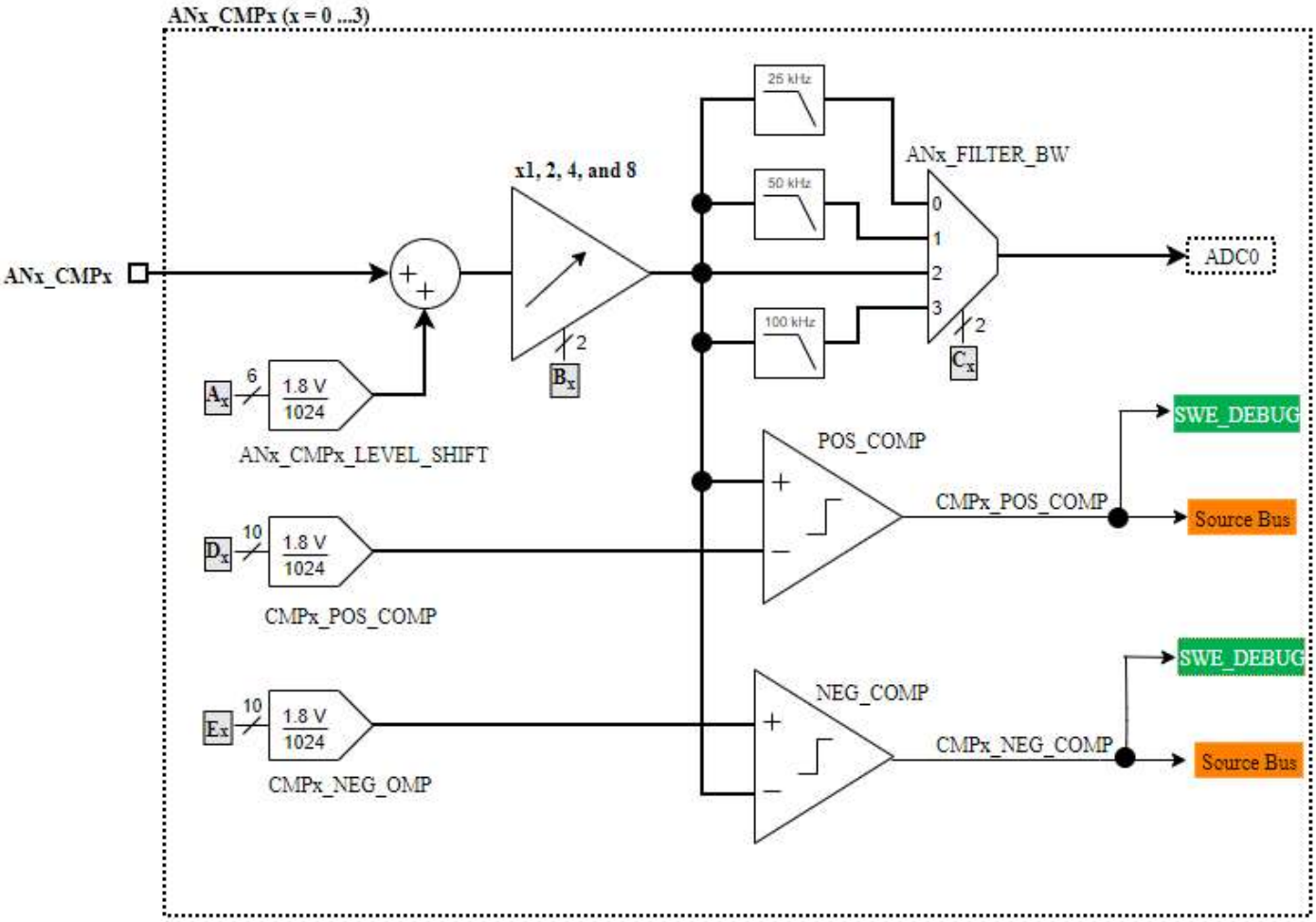


Figure 3 - Blocks associated with analog inputs ANx_CMPx (x = 0 …3)

The IC_CONFIG registers are used to setup the input signal gain and bandwidth as well as the comparator hysteresis (**x** = 0 to 3).

	PERIPHERAL.FIELD_NAME	Default
B_x	IC_CONFIG->ANx_CMPx_GAIN__U2[1]	00b
C_x	IC_CONFIG->ANx_FILTER_BW__U2[2]	00b

[1]Amplifier gains: 00b = ×1, 01b = ×2, 10b = ×4, and 11b = ×8.
[2]Low pass filter bandwidth settings: 00b = 25 kHz, 01b = 50 kHz, 10b = bypass, and 11b =100 kHz

The DAC registers provide the level shifting of the input voltage and comparator references (x=0 to 3).

	PERIPHERAL.FIELD_NAME	Default
A_x	ANALOG_DAC->ANx_CMPX_LEVEL_SHIFT__U6	0
D_x	ANALOG_DAC->CMPx_POS_COMP__U10	0
E_x	ANALOG_DAC->CMPx_NEG_COMP__U10	0

To enable the comparator used the following bids:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> CMPx_POS_COMP_ENABLE__U1[1]	0
SOURCE_BUS-> CMPx_NEG_COMP_ENABLE__U1[1]	0

[1]for x= 0-3.

The output of the comparators could be set to high by following bits:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> CMPx_POS_COMP_FORCE_HIGH__U1[1]	0
SOURCE_BUS-> CMPx_NEG_COMP_FORCE_HIGH__U1[1]	0

[1]for x= 0-3.

Analog inputs CMPx (x = 4, 5 and 7)

These analog inputs are connected to three high speed comparators as shown in Figure 4. The comparators can create events for controlling the event driven timers that are switching the powertrain switches.
For performing hysteretic charge control or dead time optimization, the input signal is sent to two comparators POS_COMP and NEG_COMP. The references to the comparators are set by their corresponding DACs. To implement the dynamic behavior and perform advanced control methods, the comparators' references can be set or controlled through the software. The DACs range can be chosen to be 1.8 V, 3V or AN7_DACVR_BUF. The CMPx block has also a zero-crossing comparator, ZC_COMP that compares the input signal with signals applied to AREF pin, AN12_BUF, AN13_ BUF or 0 V.
The comparator output CMPx _NEG_COMP is sent to the Source Bus while only one output of the comparators POS_COMP and ZC_COMP is selected to the Source Bus and SWE_DEBUG. By default, the comparator POS_COMP is selected to the Source Bus and SWE DEBUG.

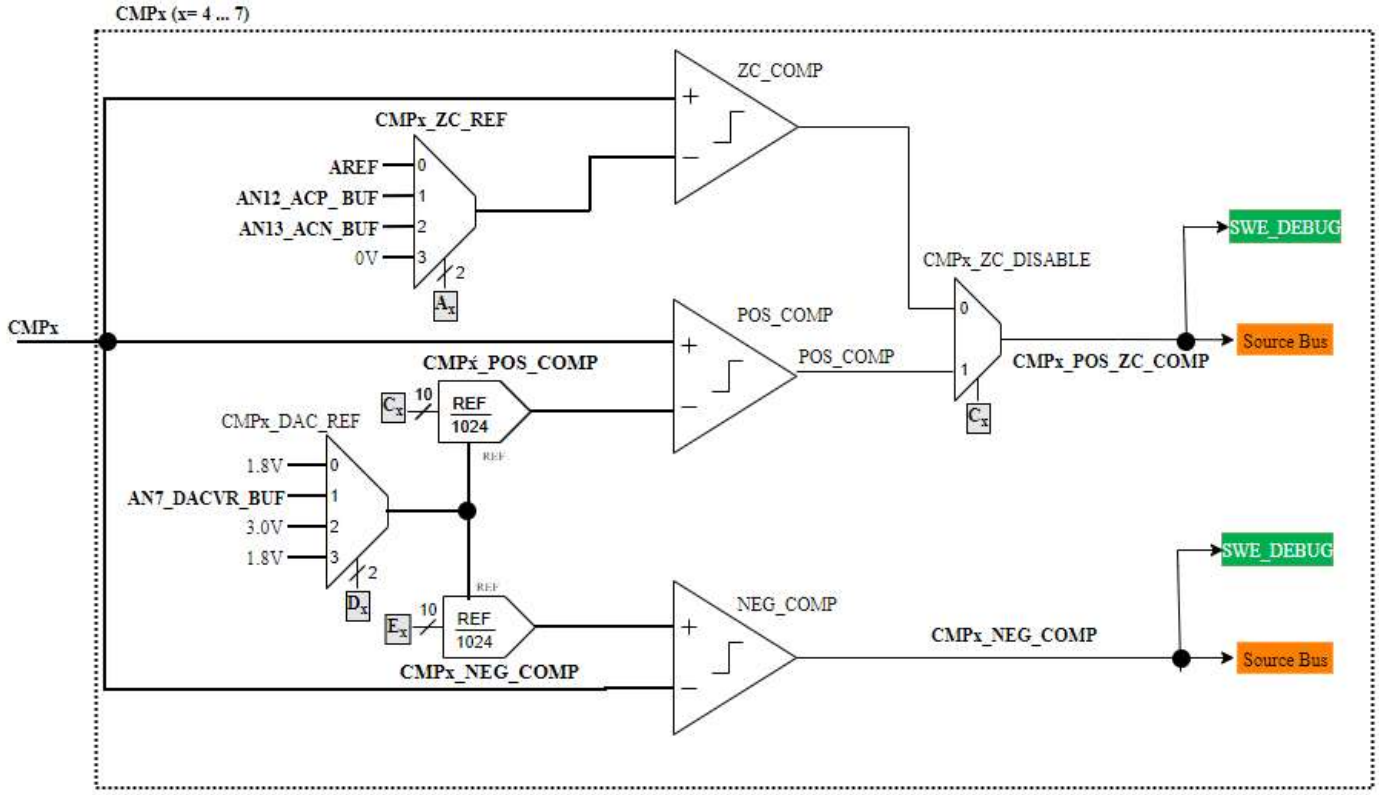


Figure 4- Blocks associated with analog inputs CMPx (x = 4, 5 and 7)

The control of the blocks is done by the CMPx registers fields.

	PERIPHERAL.FIELD_NAME	Default
A_x	IC_CONFIG-> CMPx_ZC_REF__U2[1]	00b
B_x	IC_CONFIG->CMPx_ZC_DESABLE__U2[2]	01b
D_x	IC_CONFIG->CMPx_DAC_REF__U2[3]	01b

[1]ZC comparator reference: 00b = AREF, 01b = AN12_BUF, 10b = AN13_BUF, and 11b = 0 V.
[2]Enabls the ZC comparator output to be connected to Source Bus. 0b = ZC comparator, 1b = POS comparator.
[3] DACs' references: 00b = 1.8 V; 01b = AN7_DACVR_BUF; 10b = 3 V, and 11b =1.8 V.

The DACs in the CMPx blocks are set by the following fields:

	PERIPHERAL.FIELD_NAME	Default
C_x	ANALOG_DAC-> CMPx_POS_COMP__U10	540
E_x	ANALOG_DAC-> CMPx_NEG_COMP__U10	540

To enable the comparators used the following bids:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> CMPx_POS_COMP_ENABLE__U1[1]	0
SOURCE_BUS-> CMPx_NEG_COMP_ENABLE__U1[1]	0

[1]for x= 4, 5 and 7.

The output of the comparators could be set to high and low by following bits:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> CMPx_POS_COMP_FORCE_HIGH__U1[1]	0
SOURCE_BUS-> CMPx_NEG_COMP_FORCE_HIGH__U1[1]	0
SOURCE_BUS-> CMPx_POS_COMP_FORCE_LOW__U1[1]	0
SOURCE_BUS-> CMPx_NEG_COMP_FORCE_LOW__U1[1]	0

[1]for x= 4-7.

Analog inputs AN4, AN5, AN6, and AN7_DACVR

The inputs are intended for sensing of low frequency signals by ADC2 which data is controlling the powertrains. The blocks associated with them are shown in Figure 5. The signals applied to the blocks can be optionally subtracted by a signal applied to AREF input. To reduce the impact of the switching ripple common for the power supply signals, the ADC2 input can be optionally filtered by a 4th order low pass filter 25 kHz, 50 kHz, or 100 kHz.

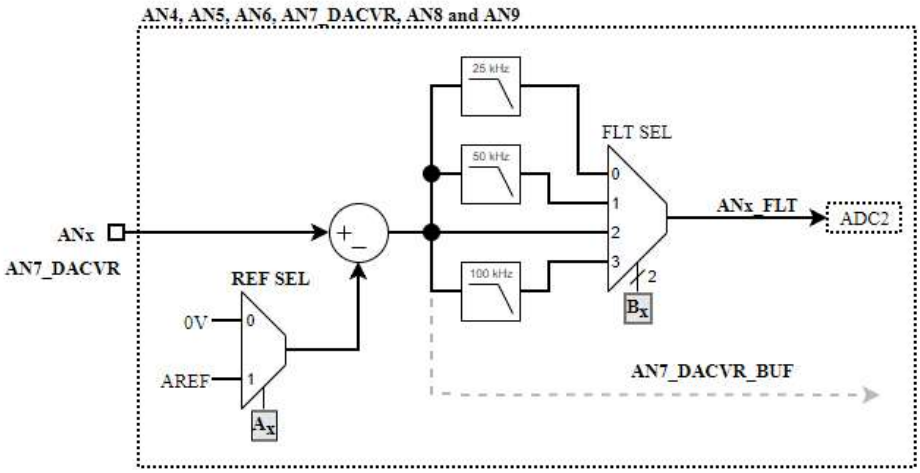


Figure 5 - Blocks associated with analog inputs AN4, AN5, AN6, and AN7_DACVR, inputs.

The control of the blocks is done by the fields of the AN4, AN5, AN6, and AN7_DACVR registers.

	PERIPHERAL.FIELD_NAME	Default
A_i	IC_CONFIG-> AN4_REF__U1[1]	0b
B_i	IC_CONFIG-> AN4_FILTER_BW__U2[2]	00b

[1]Selector input select: 0b = 0 V and 1b = AREF

[2] Low pass filter bandwidth settings: 00b = 25 kHz, 01b = 50 kHz, 10b = bypass, and 11b =100 kHz

The subtractor output of the AN7_DACVR_BUF, can be selected to set the DACs range of the CMPx (x= 4…7) blocks (see Figure 4 for the AN7_DACVR_BUF signal).

Analog inputs AN10P/N and AN11P/N

Figure 6 shows three blocks, two external current sense interfaces, XCSI0 and XCSI1, and high gain amplifier, HGA, associated with analog inputs AN10P/N and AN11P/N. The XCSI0 and XCSI1 blocks are designed to work with fully-differential isolation amplifier AMC1100 (used for current measurements). The blocks contain a differential to single ended programmable low gain amplifier. To reduce the impact of the switching ripple common for the power supply signals, the ADC2 input can be optionally filtered by a 4th order low pass filter 25 kHz, 50 kHz, or 100 kHz. The AN11P/N inputs are also connected to the HGA block. The block has a programmable high gain amplifier and a 25 kHz low pass filter.

The low gain amplifier settings are 0.45, 0.4, or 0.3. The input range voltage of XCSI0 and XCSI1 blocks is -2 V to +2 V differentially applied with a common mode of 1.2 V. When the differential signal is zero, the inputs ANxP and ANxN have the same voltage level equal to 1.2 V. When the differential signal is 2 V the voltage on the input ANxP referenced to GND is: 1.2 V+1 V = 2.2 V and the voltage on input ANxN is: 1.2 V-1 V= 0.2 V. Thus, when the differential input range voltage is -2 V to +2 V with 1.2 V common mode and the amplifier gain set to 0.45, the amplifier multiplies this 0.2 V to 2.2 V by 0.45 and remove the common mode component. As a result, the amplifier output voltage is somewhere between -0.9 V and +0.9 V. Since, the ADC input range is 0 V to 1.8 V, 0.9 V is added to shift the amplifier output voltage to be in ADC2 range (see Figure 6).

The HGA block gain settings are 30, 60, or 120. The HGA allows small differential signals to be amplified, shifted with 0.9 V for 0 V to 1.8 V ADC2 range. The HGA input range depends on the gain settings.

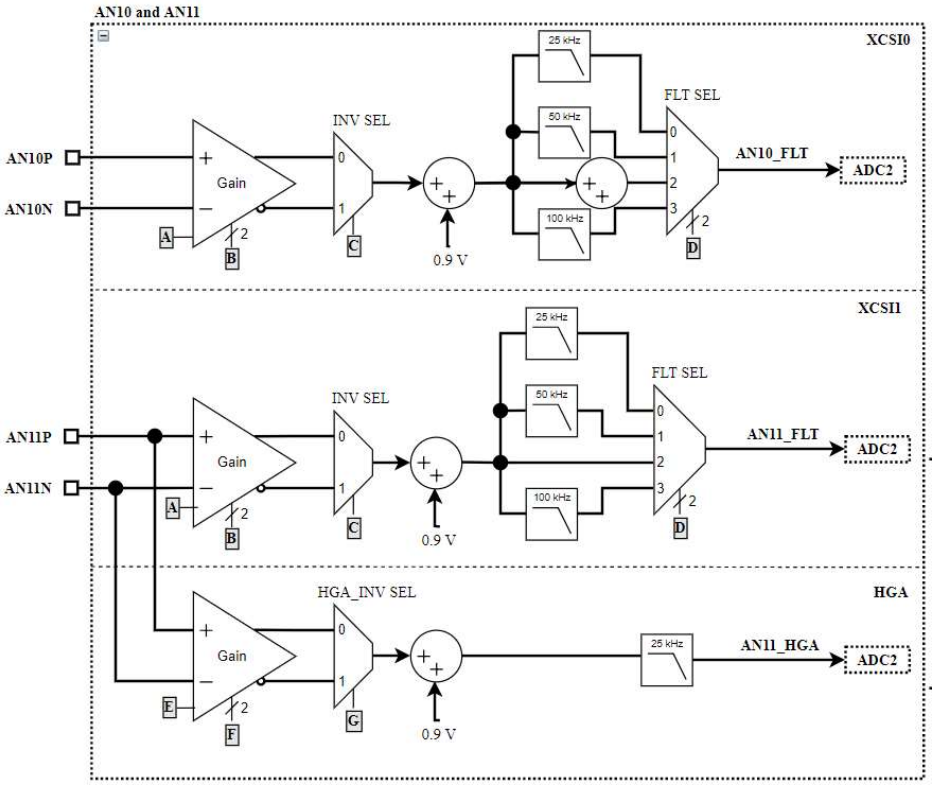


Figure 6 - Blocks associated with analog inputs AN10P/N and AN11P/N inputs

By default, the amplifiers of the XCSI0 and XCSI0 blocks are enabled and the HGA block is disabled. If the blocks are not used, to save energy they can be also disabled.
To prevent noise from the HGA to XCSI1 blocks and vice versa only one of them should be enabled. The XCSI0 and XCSI1 blocks are enabled or disabled by the ANx register fields (x=10 and 11).

	PERIPHERAL.FIELD_NAME	Default
A	IC_CONFIG-> ANx_OPAMP_DISABLE__U2[1]	00b

[1] Enable XCSIx amplifier: 0b = enable; 1b= disable.

The HGA block is enabled by:

	PERIPHERAL.FIELD_NAME	Default
E	IC_CONFIG-> AN11_HGA_ENABLE__U2[1]	00b

[1] Enable HGA block: 1b = enable; 0b= disable.

The XCSI0 and XCSI1 blocks control is done by the ANx register fields (x = 10 and 11).

	PERIPHERAL.FIELD_NAME	Default
B	IC_CONFIG->ANx_GAIN__U2[1]	00b
D	IC_CONFIG-> AN11_FILTER_BW__U2[2]	00b

[1] Chopper amplifier gains: 00b = ×0.45, 01b = ×0.4, 10b = ×0.3, and 11b =×0.3.
[2] Low pass filter bandwidth settings: 00b = 25 kHz, 01b = 50 kHz, 10b = bypass, and 11b =100 kHz

The HGA gain is set by the following register:

	PERIPHERAL.FIELD_NAME	Default
F	IC_CONFIG-> AN11_HGA_GAIN[1]	00b

[1] HGA settings: 00b = ×120, 01b = ×60, 10b = ×60, and 11b = ×30.

The input signal of the XCSI and HGA blocks can be inverted by the following fields:

	PERIPHERAL.FIELD_NAME	Default
C	IC_CONFIG->AN10_AN11_INVERT	0b
G	IC_CONFIG->AN11_HGA_INVERT	0b

Analog inputs AN12_ACP and AN13_ACN

The inputs are intended for sensing of low frequency signals specially AC grid voltages. The blocks associated with them are shown in Figure 7. The signals applied to the blocks can be optionally subtracted by a signal applied to AREF input. To reduce the impact of the switching ripple common for the power supply signals, the ADC1 input can be optionally filtered by a 4th order low pass filter 25 kHz, 50 kHz, or 100 kHz. The unfiltered signals are also connected to the zero-crossing comparator (ZCC) which output signal is used by the AC PLL block.

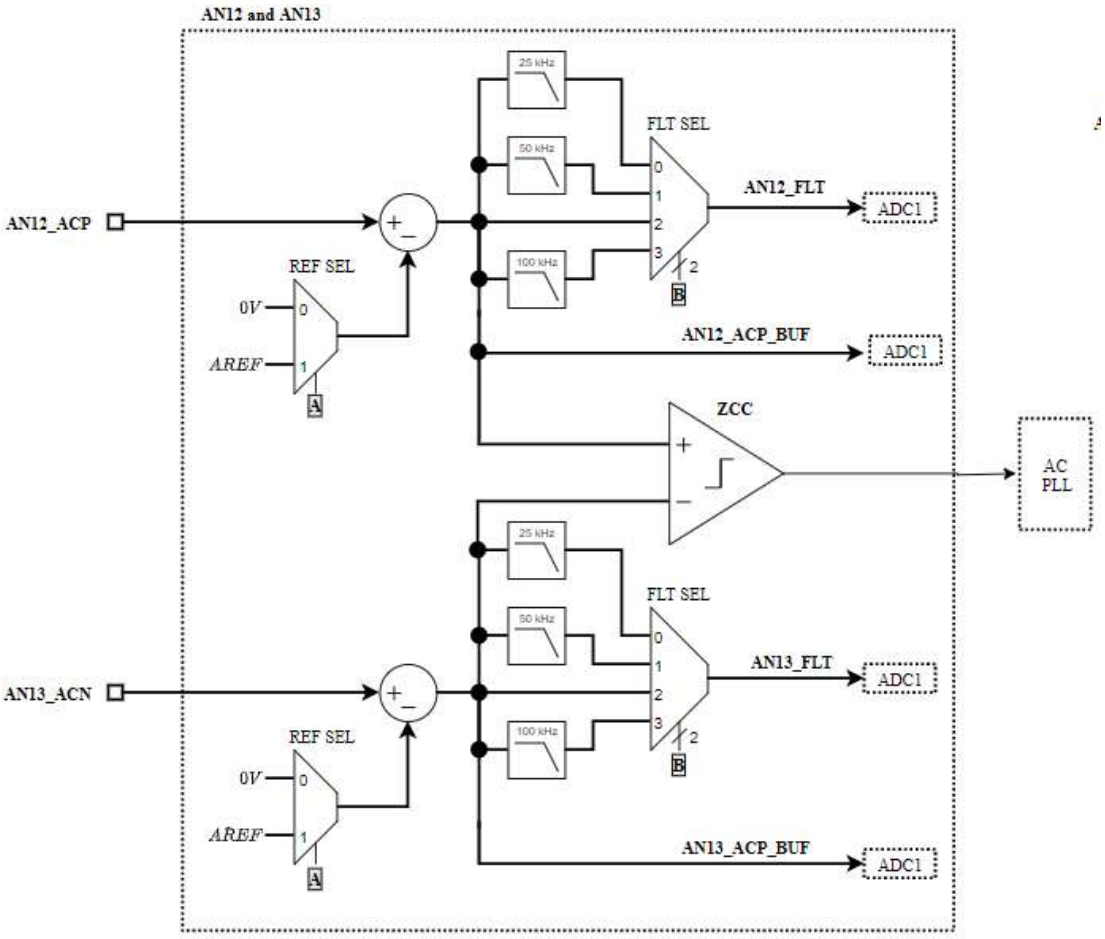


Figure 7 - Blocks associated with analog inputs AN12_ACP and AN13_ACN.

The control of the blocks is done by the AN**x** register fields (**x** = 12 and 13).

	PERIPHERAL.FIELD_NAME	Default
A_x	IC_CONFIG-> ANx_ACP_REF__U1	0b
B_x	IC_CONFIG-> AN12_FILTER_DISABLE__U2[1]	00b

[1] Low pass filter settings: 00b = 25 kHz, 01b = 50 kHz, 10b = bypass, and 11b =100 kHz

Analog to digital converters ADC0, ADC1 and ADC2

HSA8000 has three 10-bit analog to digital converters that convert the analog signals into digital. The ADC0 and ADC1 have four input and four output channels, while ADC2 has sixteen input and eight output channels. By default, the ADC clock is 20 MHz, but can be reduced to save power. It takes 14 clock cycles to convert, which results in a maximum conversion rate f_{s_max} = 1.429 MHz. When n channels are sampled, then the sampling frequency is reduced by the same factor:

$$f_s = \frac{f_{s\_max}}{n}$$

Each ADC channel output includes a digital filter. The raw data from the ADC occupies bits 15 … 6, and the bits 5 … 0 are 0. If the digital filtering is enabled, then all the bits are used. The filter adds more resolution hence the bits 5 … 0 will be populated, but the MSB stays bit 15. The filter transfer function is similar to a first order RC low pass filter.

The 3-dB bandwidth, *B*, is calculated by the following equation:

$$B = -\frac{f_s}{2\pi}\ln(1 - \frac{\alpha}{4096})$$

where 0 < α < 255 value set in the corresponding ADC register:

- For ADC0 the fields are:

PERIPHERAL.FIELD_NAME	Default
ADC->ADC0_ANx_ALPHA__U8[1]	255

[1]where x =0 …3 the fields are:

- For ADC1

PERIPHERAL.FIELD_NAME	Default
ADC->ADC1_ANx_FLT_ALPHA__U8[1]	255
ADC->ADC1_ANx_BUF _ALPHA__U8[1]	255

[1]where x =0 …3

- For ADC2

PERIPHERAL.FIELD_NAME	Default
ADC->ADC2_CHANALx_ALPHA__U8[1]	255

[1]where x channel number.

When α =255 the filter is bypassed, while when α = 0 the filter will latch the current value indefinitely. For a given bandwidth and sampling frequency, the value of α that should be set in the register is calculated by the equation:

$$\alpha = \left(1 - e^{-\frac{2\pi B}{f_s}}\right) 4096$$

For example, let assume that the desired filter 3-dB bandwidth is 10 kHz and the sampling frequency is f_s = 1.429 MHz, then substituting in the equation above, the value that should be written in the filter register is 176.

The unfiltered output of every ADC channel is connected to two digital comparators (See Figure 8 and Figure 9). The signal is compared between the maximum and minimum values. When the signal is larger than the maximum or smaller than the minimum values, a bit of the corresponding channel is set to 1 and exported to the Source Bus. This bit is sticky and is cleared manually. The status for the minimum and maximum values of the ADC0 and ADC1 comparators are OR and shown in field C_x (Figure 8). For ADC2, the comparators' status for the minimum and maximum values are and shown in C_x and D_x fields (Figure 9).

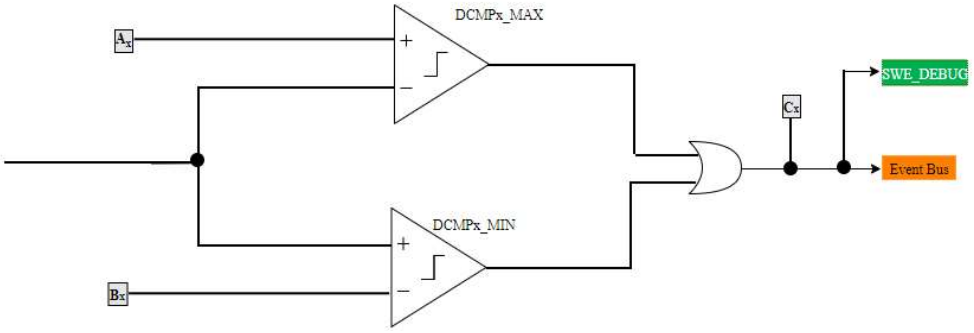


Figure 8 – ADC0 and ADC1 Output Status circuit

The fields for setting the maximum and minimum value, report and register used to set and clear the ADC output status are shown in the table below:

- For ADC0 the fields are:

	PERIPHERAL.FIELD_NAME	Default
A_x	ADC -> ADC0_ANx_COMP_MAX__U16	65472
B_x	ADC -> ADC0_ANx_COMP_MIN__U16	0
C_x	ADC -> ADC0_ANx_COMP_MIN_MAN_STATUS__U1	0b
	ADC -> ADC0_ANx_COMP_STATUS_CLEAR__U1	0b

where x = 0 … 3.

- For ADC1 the fields are:

	PERIPHERAL.FIELD_NAME	Default
A_x	ADC -> ADC1_ANx_FLT_COMP_MAX__U16	65472
	ADC -> ADC1_ANx_BUF_COMP_MAX__U16	65472
B_x	ADC -> ADC1_ANx_FLT_COMP_MIN__U16	0
	ADC -> ADC1_ANx_FLT_COMP_MIN__U16	
C_x	ADC -> ADC1_ANx_FLT_COMP_MIN_MAX_STATUS__U1	0b
	ADC -> ADC1_ANx_BUF_COMP_MIN_MAX_STATUS__U1	
	ADC -> ADC1_ANx_FLT_COMP_STATUS_CLEAR__U1	0b
	ADC -> ADC1_ANx_BUF_COMP_STATUS_CLEAR__U1	

where x= 12 and 13.

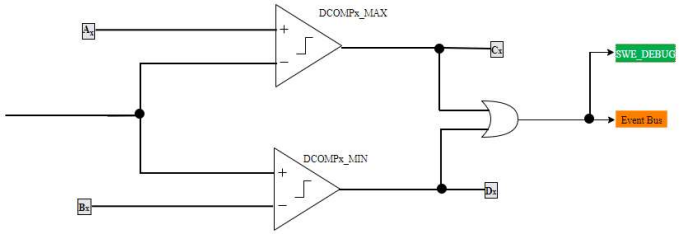


Figure 9 -ADC2 Output status circuit

- For ADC2 the fields are:

	PERIPHERAL.FIELD_NAME	Default
A_x	ADC -> ADC2_CHANNELx_COMP_MAX	65472
B_x	ADC ->ADC2_CHANNELx_COMP_MIN	0
C_x	ADC -> ADC2_CHANNELx_COMP_MAX_STATUS__U1	0b
D_x	ADC -> ADC2_CHANNELx_COMP_MIN_STATUS__U1	0b
	ADC -> ADC2_CHANNELx_COMP_STATUS_CLEAR__U1	0b

where x = 0 … 7.

The digital comparators of ADC0, ADC1, and ADC2 can be forced high and low with the following registers:

- For ADC0 the fields are:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS->ADC0_ANx_COMP_STATUS_FORCE_HIGH__U1	0
SOURCE_BUS->ADC0_ANx_COMP_STATUS_FORCE_HIGH__U1	0

where x = 0 …3.

- For ADC1 the fields are:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> ADC1_ANx_FLT_COMP_STATUS_FORCE_HIGH__U1	0
SOURCE_BUS-> ADC1_ANx_BUF_COMP_STATUS_FORCE_HIGH__U1	0
SOURCE_BUS-> ADC1_ANx_FLT_COMP_STATUS_FORCE_LOW__U1	0
SOURCE_BUS-> ADC1_ANx_BUF_COMP_STATUS_FORCE_LOW__U1	0

xhere x = 12 and 13.

- For ADC2 the fields are:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS-> ADC2_CHANNEL0_COMP_STATUS_FORCE_HIGH__U1	0
SOURCE_BUS-> ADC2_CHANNEL0_COMP_STATUS_FORCE_LOw__U1	0

where x = 0 …7.

To read any ADC, a read request should be sent by the following register:

PERIPHERAL.FIELD_NAME	Default
ADC ->READ_REQUEST__U1	0b

After the request is sent and the data is ready to be read, the register value is set to zero automatically.

The ADC clocks and busy can be exported on a GIPO pin by the DEBUG selector.

PERIPHERAL.FIELD_NAME	Selections
DEBUG ->SIGNALx	5 = ADC0_CLK 6 = ADC1_CLK 7 = ADC2_CLK 8 = ADC0_BUSY 9 = ADC1_BUSY 10 = ADC2_BUSY

x= 0 to 7 (see section DEBUG).

ADC0

The ADC0 digitizes signals coming from the ANx_CMPx (x = 0 to 3) inputs. Figure 10 shows the ADC0 block diagram. ADC0 has four sequences. For initializing the ADC0, first the number of time slots should be specified:

00b: all four of the time slots are read (0, 1, 2, 3, 0, 1, 2, 3, 0, ...)
01b: only first of the time slots is read (0, 0, 0, 0, 0, 0, ...)
10b: only the first two time slots are read (0, 1, 0, 1, 0, 1, ...)
11b: only the first three time slots are read (0, 1, 2, 0, 1, 2, …)

The second step is to select the channels for the time slots:

00: AN0_FLT
01: AN1_FLT
10: AN2_FLT
11: AN3_FLT

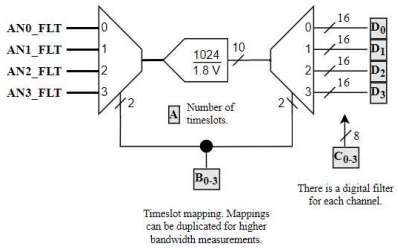


Figure 10 - ADC0 block diagram.

The registers' fields for ADC0 are shown in the table below:

	PERIPHERAL.FIELD_NAME	Default
A	ADC ->ADC0_TIME_SLOTS__U2	00b
B_{0-3}	ADC -> ADC0_TIME_SLOT0 _ SELECTION __U2	00b
	ADC -> ADC0_ TIME_SLOT1 _SELECTION __U2	01b
	ADC -> ADC0_ TIME_SLOT2 _SELECTION __U2	10b
	ADC -> ADC0_ TIME_SLOT3 _SELECTION __U2	11b
C_{0-3}	ADC->ADC0_CHx_ALPHA__U8	255
D_{0-7}	ADC ->ADC0_DATA_CHx__U16	R/O

Example 1: Let assume that the signals from inputs AN0_CMP0 and AN3_CMP3 should be read by ADC0. Only two time slots are to be used (i.e. time slots 0 and 1). AN0_CMP0 will be assigned to time slot 0 and AN3_CMP3 will be assigned to time slot 1. Also apply 1kHz bandwidth digital filters to the selected channels.

Solution: The registers settings are as follow:

The time slots number is 2.
ADC0_TIME_SLOTS__U2 = 2
Assign channel 0 (AN0_FLT) to time slot 0:
ADC0_ TIME_SLOT0 _SELECTION __U2 = 0
Assign channel 3 (AN3_FLT) to time slot 1:
ADC0_ TIME_SLOT1 _SELECTION__U2 = 3
Assign α for each channel:
ADC0_CH0_ALPHA __U8 = 0x24
ADC0_CH3_ALPHA __U8 = 0x24

Example 2: Let assume that the signals from inputs AN0_CMP0, AN2_CMP2, and AN3_CMP3 should be read by ADC0. Let's assume that it is desired to read AN0_CMP0 at twice the rate versus the other two signals. All time slots are used. AN0_CMP0 will be assigned to time slots 0 and 2, AN2_CMP2 will be assigned to time slot 1 and AN3_CMP3 will be assigned to time slot 3. Also apply 1kHz bandwidth digital filters to the selected channels.

Solution: The registers settings are as follow:

The time slots number is 4, hence write 0 into the designated register.
ADC0_TIME_SLOTS__U2 = 0
Assign channel 0 (AN0_FLT) to time slot 0 and time slot 2, to get even sampling at twice the rate:
ADC0_TIME_SLOT0 _SELECTION __U2 = 0
ADC0_TIME_SLOT2 _SELECTION __U2 = 0
Assign channel 2 (AN2_FLT) to time slot 1:
ADC0_TIME_SLOT1 _SELECTION __U2 = 2
Assign input 3 (AN3_FLT) to time slot 3:
ADC0_TIME_SLOT3 _SELECTION __U2 = 3
Assign α for each channel:
ADC0_CH0_ALPHA__U8 = 0x47/2 – it is sampled at twice the rate vs the other two channels, hence alpha should be halved to get the same bandwidth
– irrelevant, channel 1 is not used hereADC0_CH2_ALPHA = 0x47
ADC0_CH3_ALPHA__U8 = 0x47

ADC1

The ADC1 digitizes signals coming on the AN12_ACP and AN13_ACN inputs. Figure 11 shows the ADC1 block diagram. ADC1 has four time slots for measuring the input filtered and unfiltered signals. For initializing of ADC1, first the number of time slot should be specified:

00b: all four of the time slots are read (0, 1, 2, 3, 0, 1, 2, 3, 0, ...)
01b: only first of the time slots is read (0, 0, 0, 0, 0, 0, ...)
10b: only the first two time slots are read (0, 1, 0, 1, 0, 1, ...)
11b: only the first three time slots are read (0, 1, 2, 0, 1, 2, …)

The second step is to select the channel for the time slot:
00b: AN12_FLT
01b: AN13_FLT
10b: AN12_ACP_BUF
11b: AN13_ACN_BUF

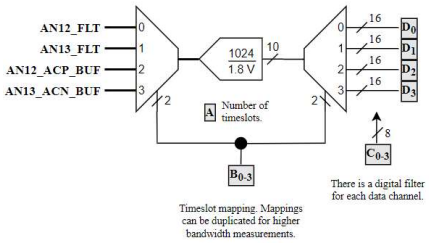


Figure 11 - ADC1 block diagram.

The registers' fields for ADC1 are shown in the table below:

	PERIPHERAL.FIELD_NAME	Default
A	ADC ->ADC1_TIME_SLOTS__U2	00b
B_{0-3}	ADC -> ADC1_TIME_SLOT0 _SELECTION __U2	00b
	ADC -> ADC1_TIME_SLOT1 _SELECTION __U2	01b
	ADC -> ADC1_TIME_SLOT2 _SELECTION __U2	10b
	ADC -> ADC1_TIME_SLOT3 _SELECTION __U2	11b
C_{0-3}	ADC-> ADC1_CHx_ALPHA __U8	255
D_{0-7}	ADC ->ADC1_DATA_CHx__U16	R/O

ADC2

Figure 12 shows the ADC2 block diagram. ADC2 has eight time slots. The initialization of ADC2 is done in three steps. First the number of time slots is selected:

000b - eight of the time slots are read (0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 0, ...)
001b - only the first of the time slots is read (0, 0, 0, 0, 0, 0, ...)
010b - only the first two time slots are read (0, 1, 0, 1, 0, 1, ...)
011b - only the first three time slots are read (0, 1, 2, 0, 1, 2, 0, ...)
100b - only the first four time slots are read (0, 1, 2, 3, 0, 1, 2, 3, 0, ...)
101b - only the first five time slots are read (0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...)
110b - only the first six time slots will be read (0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 0, ...)
111b - only the first seven time slots are read (0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, ...)

Then the channels associated for each time slot are selected.

000b – CH0
001b – CH1
010b – CH2
011b – CH3
100b – CH4
101b – CH5
110b – CH6
111b – CH7

The third step is to select the input for the corresponding channel:

0000b - Internal Temperature
0001b - AN4_FLT
0010b - AN5_FLT

0011b - AN6_FLT
0100b - AN7_FLT
0101b - High Impedance (no connect)
0110b - 1V8A
0111b – AN10_FLT
1000b – AN11_FLT
1001b – Reserved
1010b – Reserved
1011b - GND
1100b - AN11_HGA
1101b - CAL_DAC
1110b - Reserved
1111b – Reserved

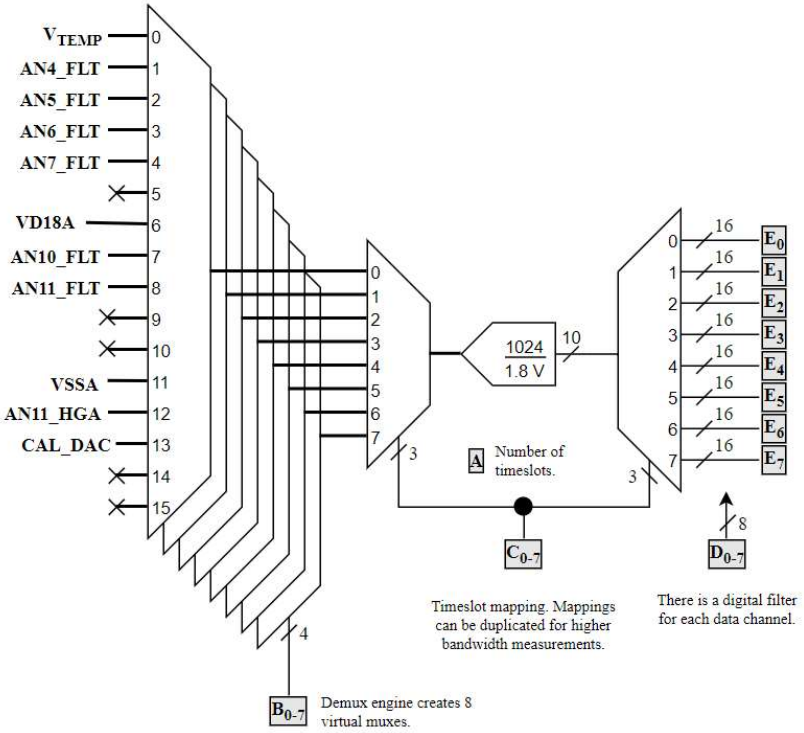


Figure 12 - ADC2 block diagram.

The registers' fields for ADC2 are shown in the table below:

	PERIPHERAL.FIELD_NAME	Default
A	ADC ->ADC2_TIME_SLOTS__U3	000b
B_{0-3}	ADC -> ADC2_INPUT_CH0__U4	0000b
	ADC -> ADC2_INPUT_CH1__U4	0001b
	ADC -> ADC2_INPUT_CH2__U4	0010b
	ADC -> ADC2_INPUT_CH3__U4	0011b
	ADC -> ADC2_INPUT_CH4__U4	0100b
	ADC -> ADC2_INPUT_CH5__U4	0101b
	ADC -> ADC2_INPUT_CH6__U4	0110b
	ADC -> ADC2_INPUT_CH7__U4	0111b
C_{0-3}	ADC -> ADC2_TIME_SLOT0 _SELECTION__U3	000b
	ADC -> ADC2_TIME_SLOT1 _SELECTION__U3	001b
	ADC -> ADC2_TIME_SLOT2 _SELECTION__U3	010b
	ADC -> ADC2_TIME_SLOT3 _SELECTION__U3	011b
	ADC -> ADC2_TIME_SLOT4 _SELECTION__U3	100b
	ADC -> ADC2_TIME_SLOT5 _SELECTION__U3	101b
	ADC -> ADC2_TIME_SLOT6 _SELECTION__U3	110b
	ADC -> ADC2_TIME_SLOT7 _SELECTION__U3	111b
D_{0-7}	ADC->ADC2_CHx_ALPHA__U8	255
$F_{0\ 7}$	ADC ->ADC2_DATA_CHx__U16	R/O

Analog DACs

The IXC 2 has the following analog DACs:

- 16 10-bit DACs for comparators internal references
- 1 10-bit DAC for debugging
- 4 6-bit DACs for level shifting

The DACs values for the comparator references and debugging, DAC is calculated by:

$$DAC = \frac{1024 * V_{ref}}{1.8}$$

where V_{ref} is the desire voltage reference.

The DAC value for level is calculated by:

$$DAC = \frac{64 * V_{shift}}{1.8}$$

where V_{shift} is the level shifting voltage.

Temperature sensor

The HSA8000 temperature is measured by a temperature sensor. The sensor is designed to change its output voltage based upon HSA8000 temperature. The output voltage is measured by ADC2. The relationship between the temperature and the voltage is linear. The slope is approximately a constant while the offset is process dependent. Therefore, the temperature sensor requires one-point calibration.

Digital Interface

The digital interface processes the analog signals sensed by the analog interface and sends signals to control the power train switches. The digital interface includes the following blocks:

- CPU
- ROM and RAM
- Switch Engine
- I/O Block
- Grid PLL
- COMMs
- Serial Structure interface
- Watchdog system
- Math Accelerators (sin, cos, sqrt, divider).
- RTC
- ADCs
- DACs.
- DEBUG infrastructure

As it is shown in Figure 1, there are two buses for communication between the different HSA8000 blocks:

- AMBA Bus (32 bit/ 50 MHz)
- EVENT SOURCE Bus.

The AMBA BUS connects all HSA8000 blocks to the CPU. The EVENT SOURCE BUS connects the analog interface and the I/O block to the switching engine. The EVENT SOURCE BUS is carrying data from the comparators, GPIOs, and other digital AUX sources to the timing engine and is synchronized with 100 MHz clock. In the following section the functionality of the digital interface blocks is presented.

CPU

The 32-bit RISC micro-controller operates at 50 MHz with 64 kiB internal RAM, 4 kiB ROM (boot loader) and provides the execution of code for customizable control algorithms.
The CPU core is an EnSilica 32-bit RISC processor. For more information, refer to the EnSilica web site at ***http://www.ensilica.com***.

The specific configuration used is outlined in the following table. The interface to the various functional blocks is through the AMBA bus.

Configuration type	**Configuration options used**
Architectural	BITS:32, REGISTERS:16, von Neumann
Clock speed	50 MHz maximum
Memory	ENDIAN (Little), 32-bit byte-addressable

Interrupts

HSA8000 has 6 maskable interrupts which are defined in Solantro_interrupt_map.h. The sources of the interrupts are as follow:

- UART
- SPI
- GPIO
- UART_HDLC
- TIMER
- AC_PLL

There is only one level of interrupts, but each interrupt source can have its own handler function. Each interrupt has an interrupt service routine (ISR) defined somewhere in the code memory. The microcontroller makes use of interrupt vector table to find the starting address of ISRs. The interrupt vector table defines where the code of a particular interrupt routine is located in microcontroller memory. When an event occurs, the controller generates a hardware interrupt. The interrupt forces the controller's program counter to jump to a specific address in the program memory. At this memory location is installed a special function known as an interrupt service routine (ISR) or an interrupt handler.

Thus, upon generating a hardware interrupt, the program execution jumps to the interrupt handler and executes the code in that handler. When the handler is done, then program control returns the controller to the original program and continue with its execution. Therefore, a hardware interrupt allows the controller to interrupt an existing program and react to some external hardware event and then continue with the program.

The vectors to the functions are stored in the exception table which is defined in exception.h. The interrupt.h file defines functions for interrupts' enabling and masking. The following code ncludes the exception and interrupt headers.

```
#include <esirisc/esirisc.h>
```

Interrupt Initialization Function

Interrupt initialization function is called once from the MAIN INIT section. It creates a local temporary pointer to hold the base address of the exception table.

Handler Function

The handler function prototype is:

```
ESI_EXCEPTION_HANDLER void Interrupt_Handler(void);
```

The function should clear any hardware flags in the peripheral which initiated the interrupt, handle the interrupt and finish by setting an acknowledge mask. The timer peripheral mask is used below as an example.

```
esi_interrupt_acknowledge_mask(PERIPHERAL_INTERRUPT__MASK__TIMER);
```

Vector Table

The vector table is an array of pointers for various exceptions including interrupts. When a handler function has been defined for an interrupt then the pointer can be updated with the following code.

```
esi_exception_table_t *vectors;

vectors = esi_exception_get_exception_table();

vectors->interrupt[PERIPHERAL_INTERRUPT__MASK__TIMER] = Interrupt_Handler;
```

Interrupt Mask

The interrupt mask allows the hardware signal to generate an exception for the CPU to handle. The following code is an example of enabling the mask for the timer peripheral.

```
esi_interrupt_set_mask(esi_interrupt_get_mask() | PERIPHERAL_INTERRUPT__MASK__TIMER);
```

Enable

Once the functions have been defined, the vectors updated, the peripherals initialized and enabled then the interrupts can be enabled with the following command:

```
esi_interrupt_enable();
```

Example

Below is an example of AC_PLL interrupt and handling:

```
#include "iso646.h"                // Used for C++ operator synonyms.
#include "stdbool.h"               // Used for true / false.

#include <esirisc/esirisc.h>       // enSilca library used for interrupt control

#include "solantro_interrupt_map.h"
#include "solantro_peripheral_map.h"

ESI_EXCEPTION_HANDLER void PLL_Interrupt_Handler(void);

void interrupt_init(void) {
    esi_exception_table_t *vectors;

    vectors = esi_exception_get_exception_table();
    vectors->interrupt[PERIPHERAL_INTERRUPT__INDEX__AC_PLL] =
PLL_Interrupt_Handler;

    // The PLL has multiple sources which can trigger an interrupt.
    // Enabling just the two below.
    PERIPHERAL.AC_PLL->RECONSTRUCTED_AC_SAMPLES_IRQ_ENABLE__U1 = 1;
    PERIPHERAL.AC_PLL->RECONSTRUCTED_AC_CROSSING_IRQ_ENABLE__U1 = 1;

 esi_interrupt_set_mask(esi_interrupt_get_mask() |
PERIPHERAL_INTERRUPT__MASK__AC_PLL);
}

ESI_EXCEPTION_HANDLER void PLL_Interrupt_Handler(void) {
    PERIPHERAL.AC_PLL->STATUS__REG = 0x1F;
```

```
        Controller_Interrupt();
      esi_interrupt_acknowledge_mask(PERIPHERAL_INTERRUPT__MASK__AC_PLL);
}
```

ROM and RAM

The memory-model is 32 bits wide and is byte-addressable using appropriate byte-access load/store instructions. There are 24 addressing bits, but for 32-bit word accesses the 2 least significant address bits are ignored.

Memory map

The HSA8000 memory map is listed in the following table below:

Address	Description
0x000000 - 0x000FFF	ROM (4 kB)
0x020000 - 0x027FFF	RAM0 (32 kB)
0x028000 - 0x02FFFF	RAM1 (32 kB)
0x800000 - 0xFFFFFF	AMBA

Internal ROM

The HSA8000 internal ROM is 4 kB of on-chip ROM, organized as 1k 32-bit words. The internal ROM starts at address 0x000000. When the processor resets, it starts executing instructions from this ROM. The ROM-based routines include:

- initialization - stack pointer, exception vectors, watchdogs
- self-checks - ROM checksum, RAM test, flash code checksum
- flash-boot-load instructions into the RAM from external flash memory connected to the SPI, check the checksum, and if OK jump to the first instruction.

Internal SRAM

The HSA8000 internal SRAM is organized into two adjacent 32 kB blocks (64 kB total), in von-Neumann mode.

AMBA Structure access

Memory addresses from 0x800000 to 0xFFFFFF map to the embedded AMBA Structure bus. HSA8000 complies with the AMBA 4 standard. Accesses to AMBA-space must be 32-bit word aligned (the two least-significant-bits of the address must be zero). An exception is generated when unaligned accesses are attempted. Details for each of the set of registers within each Structure is provided in the ***HSA8000_Register Map*** document.

Flash memory

HSA8000 contains 256 kB flash memory. The ***Huada Test and Control Tool User's Guide*** **and the** ***Huada Test and Control Tool Programmer's Guide*** details the compilation and debug environment for building loads that

are downloaded into the HSA8000 flash memory for auto-booting. The flash memory is accessed serially through the SPI interface, which is connected to the CPU through the AMBA bus, like any other peripheral. At reset, the bootloader (located in ROM) will copy the executable image of the code from the flash into the RAM and will launch it in execution.

Switching Engine

The Switching Engine is one of the most important HSA8000 blocks. It is used to control the power switches of different applications such as DC/DC convertors, DC-AC invertors, battery chargers, and others. The high-level block diagram of the HSA8000 Switching Engine block is shown in Figure 13. The Switching Engine block has four main blocks: Event Control, Fault Processor, Timing Engine, and Driver Control.

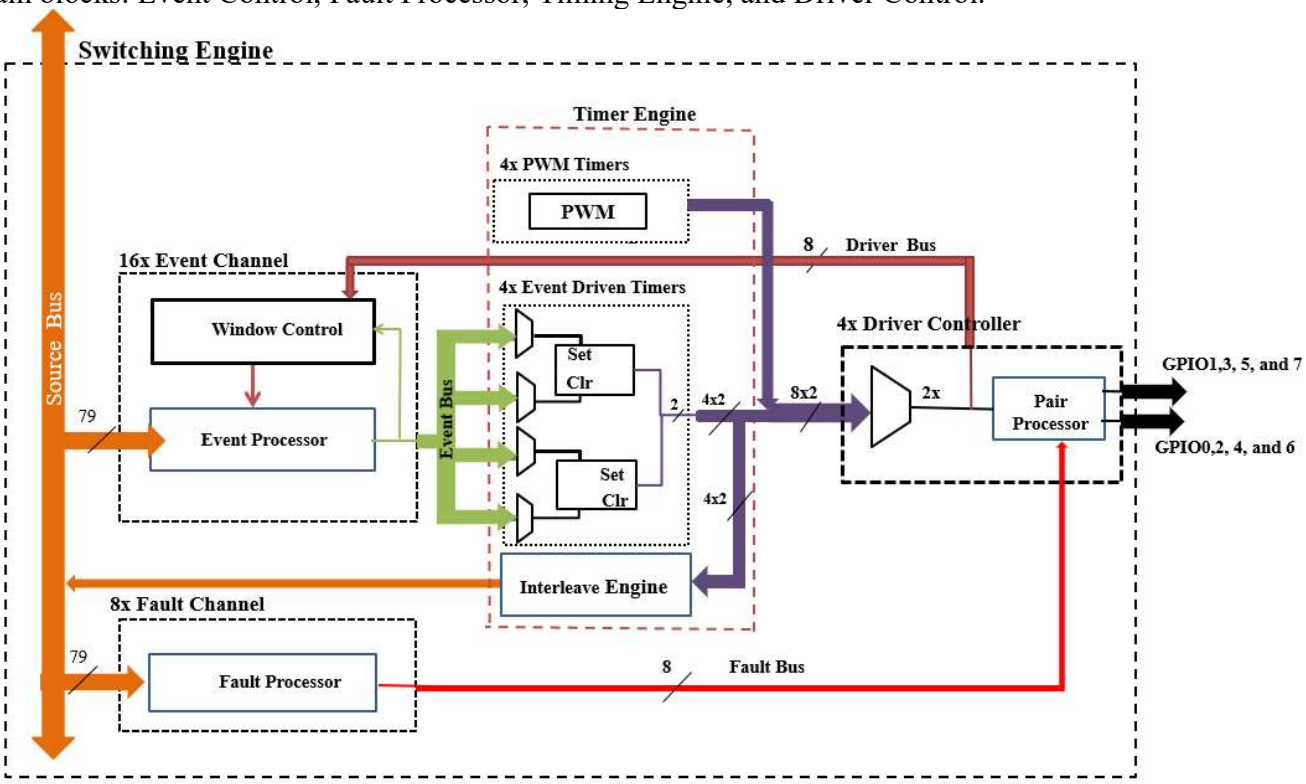


Figure 13 - High level block diagram of HSA8000 Switching Engine

Event Control block

The Event Control block consists of 16 Event Channels. Every channel has an Event Processor and a Window Control block. The function of Event Control block is to select up to 16 signals from the Source Bus, process them and output up to 16 events. The Source Bus includes signals from Analog Sensing blocks, Digital blocks (ADC, timing engine) and any GPIO.

Event Processor

The Event Processor[x] block diagram (**x** is from 0 to 15 - the channel number) is shown in Figure 14.

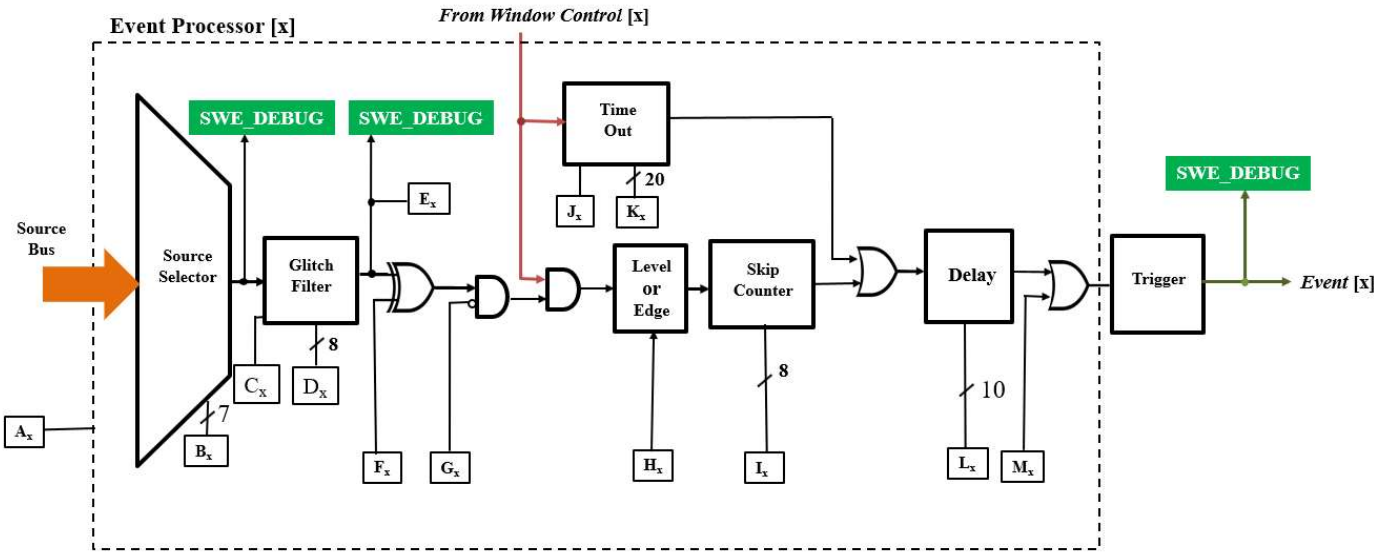


Figure 14 - An Even Processor block diagram

	PERIPHERAL.FIELD_NAME	Default
A_x	SWE_ECFG->EVENTx_ENABLE__U1	0b
B_x	SWE_ECFG->EVENTx_INPUT__U7	0b
C_x	SWE_ECFG->EVENTx_GLITCH_MODE__U1	0b
D_x	SWE_ECFG->EVENTx_GLITCH_WIDTH__U8	0b
E_x	SWE_ECFG->EVENTx_GLITCH_FILTER_STATUS__U1	r/o
F_x	SWE_ECFG->EVENTx_INVERT__U1	0b
G_x	SWE_ECFG->EVENTx_DISABLE_SOURCE__U1	0b
H_x	SWE_ECFG->EVENTx_TRIGGER_TYPE__U1	0b
I_x	SWE_ECFG->EVENTx_SKIP_COUNT__U7	0b
J_x	SWE_ECFG->EVENTx_TIMEOUT_MODE__U1	0b
K_x	SWE_ECFG-> EVENTx_TIMEOUT_U20	0b
L_x	SWE_ECFG-> EVENTx_DELAY_U10	0b
M_x	SWE_ECFG-> EVENTx_TRIGGER__U1	0b

The Event processor has a Source Selector that selects any source from the Source Bus. The selection is done by B_x. In the table below are the given sources that can be selected.

0 = CMP0_POS	35 = ADC0_OVR_CH3	53 = BKBT_PHASE_2	80 = GPIO_16
1 = CMP0_NEG	36 = ADC1_OVR_CH0	54 = BKBT_PHASE_3	81 = GPIO_17
2 = CMP1_POS	37 = ADC1_OVR_CH1	55=0	82 = GPIO_18
3 = CMP1_NEG	38 = ADC1_OVR_CH2	64 = GPIO_0	83 = GPIO_19
4 = CMP2_POS	39 = ADC1_OVR_CH3	65 = GPIO_1	84 = GPIO_20
5 = CMP2_NEG	40 = ADC2_OVR_CH0	66 = GPIO_2	85 = GPIO_21
6 = CMP3_POS	41 = ADC2_OVR_CH1	67 = GPIO_3	86 = GPIO_22
7 = CMP3_NEG	42 = ADC2_OVR_CH2	68 = GPIO_4	87 = GPIO_23
16 = CMP4_POS	43 = ADC2_OVR_CH3	69 = GPIO_5	88 = GPIO_24
17 = CMP4_NEG	44 = ADC2_OVR_CH4	70 = GPIO_6	89 = GPIO_25
18 = CMP5_POS	45 = ADC2_OVR_CH5	71 = GPIO_7	90 = GPIO_26
19 = CMP5_NEG	46 = ADC2_OVR_CH6	72 = GPIO_8	91 = GPIO_27
20 =	47 = ADC2_OVR_CH7	73 = GPIO_9	92 = GPIO_28
21 =	48 = TGEN_EVENT_0	74 = GPIO_10	93 = GPIO_29
22 = CMP7_POS	49 = TGEN_EVENT_1	75 = GPIO_11	94 = GPIO_30
23 = CMP7_NEG	50 = TGEN_EVENT_2	76 = GPIO_12	95 = GPIO_31
32 = ADC0_OVR_CH0	51 = TGEN_EVENT_3	77 = GPIO_13	
33 = ADC0_OVR_CH1	52 = BKBT_PHASE_1	78 = GPIO_14	
34 = ADC0_OVR_CH2		79 = GPIO_15	

The selected signal can be optionally glitch-filtered by the Glitch Filter block. Two modes of filtering can be selected (C_x): integrating and consecutive.
For consecutive mode, the filter output toggles to either "1" or "0" when the input signal stays at "1" or "0" for a specified consecutive number of clock ticks (10ns). The number of consecutive clock ticks is set in the EVENTx_GLITCH_WIDTH (ns) (D_x). If the number is not reached, the filter stays in the same state. For integrating mode, when the input is in “1” an accumulator is incremented until it reaches the number set in EVENTx_GLITCH_WIDTH(ns)(Dx), and when the input is in “0” the (same) accumulator is decremented until it reaches 0. The output toggles to “1” when the accumulator reaches the number set in EVENTx_GLITCH_WIDTH(ns)(Dx), and toggles to”0” when the accumulator reaches 0.
An event can be triggered by the rising or the falling edge of the input signal (F_x). The source signal can also be disabled by using an AND gate (G_x). The selected event source is considered only after a window is opened by the Window Control block (Figure 13).
There are two modes of generating an event - level or edge (H_x). When the field EVENTx_TRIGGER_TYPE is set to 1 the edge mode is selected. Figure 15 and Figure 16 show the timing diagrams for event generation in edge mode on the falling and rising edge respectively.

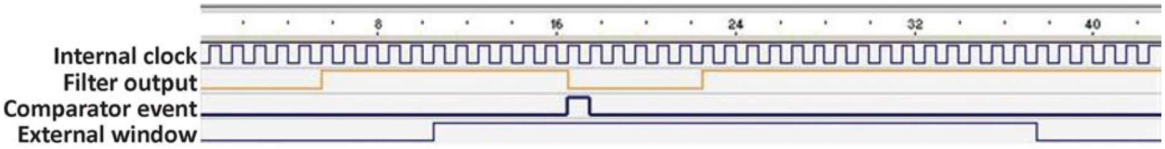


Figure 15 - Timing Diagram for Generating an Event on the Falling Edge

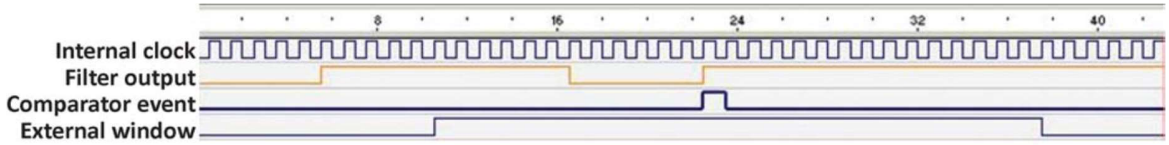


Figure 16 - Timing Diagram for Generating an Event on the Rising Edge

When the EVENTx_TRIGGER_TYPE field is set to “0”, level mode is selected. In this case, an event is generated if the filter output is already in "1" when the window opens. Figure 16 and Figure 17 show two examples for the same input, but two different modes are used: one with edge mode (Figure 16) and the other with the level mode (Figure 17).

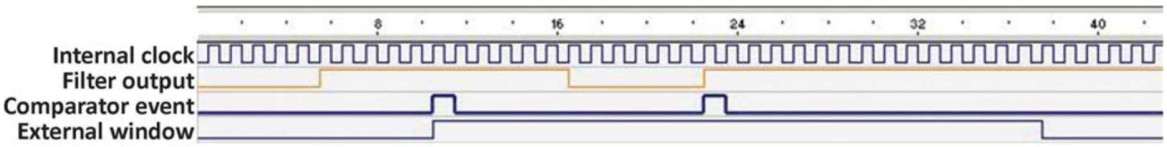


Figure 17 - Timing Diagram for Generating of an Event on the Level Mode

The window for generating an event is created in the Window Control block (Figure 13).
The first pulses after the window starts can be skipped. The skipped pulses number is given in the EVENTx_SKIP_COUNT (I_x).
A timeout event can be scheduled if the timeout mode is set (J_x) and a value in the EVENTx_TIMEOUT register (K_x) is set. The timeout event is relative to the moment when the external window is turned ON. The timeout is intended to operate for safety if the source event does not occur.
After an event is registered, it can be reported after a programmable delay, if desired. The delay is specified in the EVENTx_DELAY register (L_x). An event can be also triggered by setting EVENTx_TRIGGER field to 1.
The selected signal from the Source Bus, the filtered and the output signal are connected to SWE_DEBUG for testing purpose.

Window Control block

Each of the 16 Event Control channels have its associated Window Channel[x], where x is the channel number from 0 to 15. An Event processor only looks for the selected event source when its associated window is turned ON. Basically, a window is turned ON based on a rising or falling edge of a driver output or a combination of those. The **Window Triger Bus** coming from the eight driver output signals feeds all 16 window control channels (see Figure 13). The window is always turned OFF when the associated event occurs (either based on the hardware source or on timeout). The output of the windows is connected also to SWE_DEBUG. A Window Channel is shown in Figure 18.

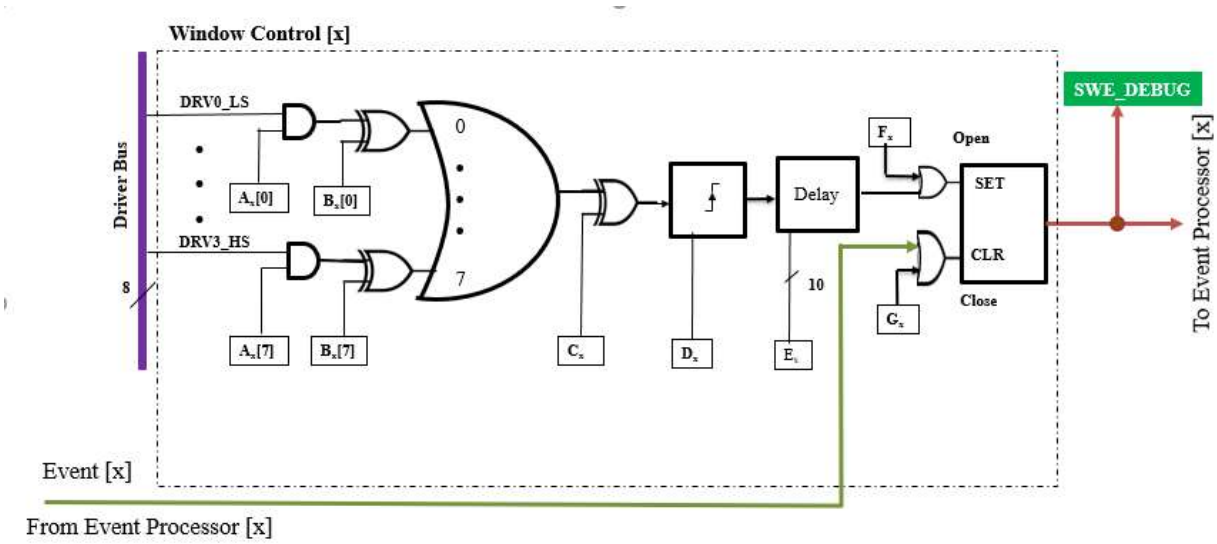


Figure 18 - A Window Channel of the HSA8000 Window Control block

	PERIPHERAL.FIELD_NAME	Default
$A_x[0]$	SWE_WCFG->WINDOWx_DRV0_LS_TRIGEGER__U1	0b
$A_x[1]$	SWE_WCFG->WINDOWx_DRV0_HS_TRIGEGER __U1	0b
…	. . .	0b
$A_x[7]$	SWE_WCFG->WINDOWx_DRV3_HS_TRIGEGER __U1	0b
$B_x[0]$	SWE_WCFG->WINDOWx_DRV0_LS_INVERT__U1	0b
$B_x[1]$	SWE_WCFG->WINDOWx_DRV0_HS_INVERT__U1	0b
…	. . .	0b
$B_x[7]$	SWE_WCFG-> WINDOWx_DRV0_HS_INVERT__U1	0b
C_x	SWE_WCFG->WINDOWx_MODE__U1	0b
D_x	SWE_WCFG->WINDOWx_MASK_1CLK_AFTER_STR_U1	0b
E_x	SWE_WCFG->WINDOWx_DELAY_U10	0b
F_x	SWE_WCFG->WINDOWx_FORCE_OPEN__U1	0b
G_x	SWE_WCFG-> WINDOWxFORCE_CLOSE__U1	0b

A Window Channel has 8 inputs (DRV0_LS, DRV0_HS … DRV3_HS) coming from the Driver Controller block through the Driver Bus. The inputs opening a window can be selected by the fields (A_X). The inputs can be also inverted before **OR**-ing (B_X). If, for example, the window must be opened when either of two gates are turned ON (e.g. DRV0_LS or DRV1_LS are ON) then the user should enable the inputs (DRV0_LS and DRV1_LS) and then choose rising edge to trigger the window. Another example, when the window must be turned ON after two gates are both turned ON (e.g. DRV0_LS or DRV1_LS are ON); the user should enable the inputs, then invert them, then select the falling edge to trigger the window. According to De Morgan's law, by complementing the inputs and the output of an OR-gate, the OR-gate will be practically transformed into an AND-gate.
When changing an inverting register bit, a false event can be created. To prevent from generating a false event, the rising/falling block can be inhibited for the first clock cycle after the inversion bit has been changed (D_x).

When an edge is detected, the control window starts after a delay set in the WINDOWxDELAY. The purpose of the delay is to blank the source for the event after a driver was switched, to prevent generating false events due to the noise created when the driver was toggled.
The beginning and end of a window can be also forced ON or OFF with the software by setting "1" to the WINDOWx_FORCE_OPEN and WINDOWx_FORCE_CLOSE fields.

Timing Engine block

The time engine block consists of Event Driven Timer and Pulse Width Modulation (PWM) blocks.

Event Driven Timer

The events from the Event Processor block are sent to the Event Driven Timer block. These events are used by the Event Driven Timer block to create signals for the Driver Controller block (Figure 13). The Event Driven Timer block has 4 timers (Event Driven Timer [**x**], where x is equal from 0 to 3). Every Event Driven Timer has 16 inputs and one output pair. The signals of the four pairs are sent to the Driver Controller block and also to the Interleave Engine (Figure 13).
The Event Driven Timer[x] block contains two identical blocks EDT**x**_LS and EDT**x**_HS (Figure 19). The blocks have two selectors and a Flip Flop. The selectors select the turn ON and OFF events for the driver. If necessary, the driver can be forced ON and OFF by the controller. The timer can also be disabled after the first event.
The EDT**x**_LS and EDT**x**_HS are also connected to the SWE_DEBUG.

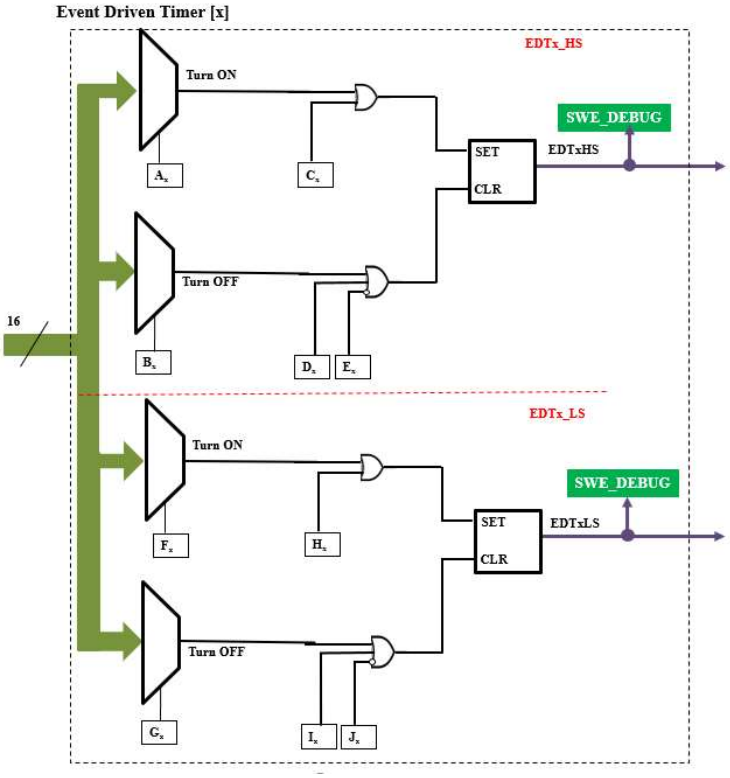


Figure 19 – An Event Timer block diagram

	PERIPHERAL.FIELD_NAME	Default
A_x	SWE_EDT->EDTx_HS_TURN_ON_EVENT__U5	0b
B_x	SWE_EDT->EDTx_HS_TURN_OFF_EVENT__U5	0b
C_x	SWE_EDT->EDTx_HS_FORCE_ON__U1	0b
D_x	SWE_EDT->EDTx_HS_FORCE_OFF__U1	0b
E_x	SWE_EDT->EDTx_HS_ENABLE__U1	0b
F_x	SWE_EDT->EDTx_LS_TURN_ON_EVENT__U5	0b
G_x	SWE_EDT->EDTx_LS_TURN_OFF_EVENT__U5	0b
H_x	SWE_EDT->EDTx_LS_FORCE_ON__U1	0b
I_x	SWE_EDT->EDTx_LS_FORCE_OFF__U1	0b
J_x	SWE_EDT->EDTx_LS_ENABLE__U1	0b

Interleave Engine

The Interleave Engine is part of the Event Driven Timer block (
Figure 20). It has a selector selecting a master phase from the inputs of the four pairs coming from the Event Driven Timers.

	PERIPHERAL.FIELD_NAME	Default
A	SWE_EDT->EDT_INTERLEAVE_ENABLE__U1	0b
B	SWE_EDT->EDT_INTERLEAVE_MODE__U1	0b
C	SWE_EDT->EDT_INTERLEAVE_MASTER__U1	0b
D	SWE_EDT->EDT_SLAVE_MAX_ON_TIME__U2	0b

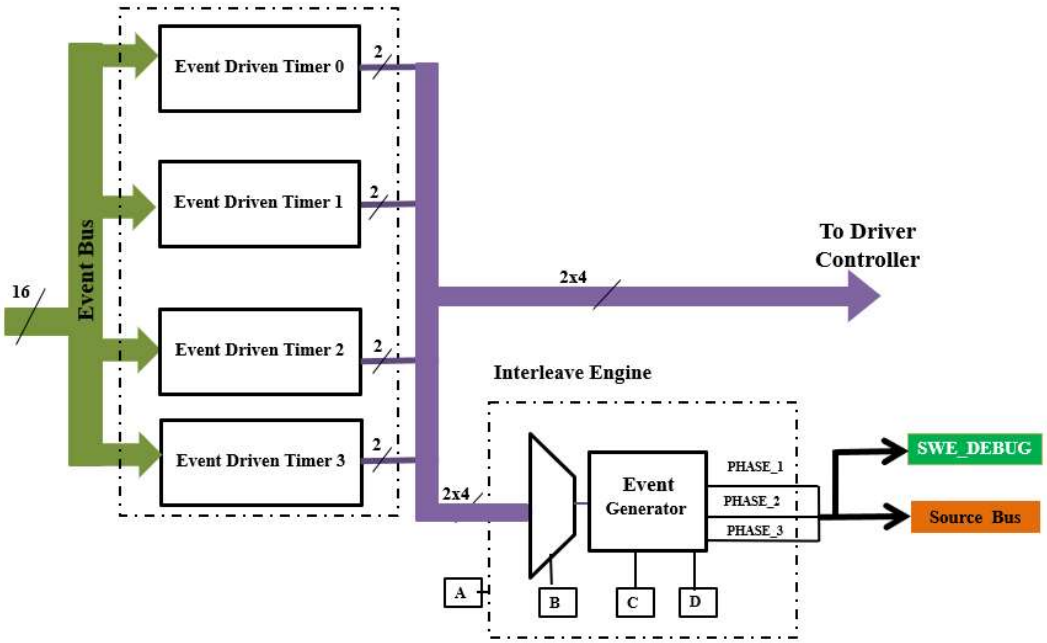


Figure 20 - Event Driven Timer block diagram

	PERIPHERAL.FIELD_NAME	Default
A	SWE_EDT->EDT_INTERLEAVE_ENABLE__U1	0b
B	SWE_EDT->EDT_INTERLEAVE_MODE__U1	0b
C	SWE_EDT->EDT_INTERLEAVE_MASTER__U1	0b
D	SWE_EDT->EDT_SLAVE_MAX_ON_TIME__U2	0b

The operating mode of the Interleave Engine is as follow:
00 – reserved (do not use)
01 - two phases (master phase and phase 2 at 180º, and phases 1 and 3 is OFF)
10 - three phases (master phase, phase 1 at 120º, phase 2 at 240º, and phase 3 is OFF)
11 – four phases (master, phase 1 at 90º, phase 2 at 180º, and phase 3 at 270º).
Every face can be forced high and low by the following registers:

PERIPHERAL.FIELD_NAME	Default
SOURCE_BUS->EDT_INTERLEAVE_PHASEx_FORCE_HIGH__U1	0b
SOURCE_BUS->EDT_INTERLEAVE_PHASEx_FORCE_LOW__U1	0b

Figure 21 provides an example of one phase (non-interleaved) mode. In this case, no signal is sent to the event source bus. The timers of the pair, chosen as the master, control the switching; the Interleave Engine should be disabled.

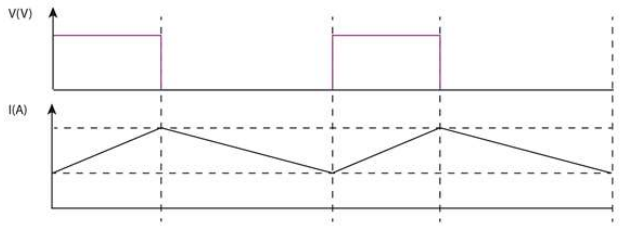


Figure 21 - One Phase (non-interleaved) Mode Waveforms

Figure 22 provides an example of two-phase interleaved mode. The timers of the pair chosen as the master control the switching of the first phase. The engine measures the switching period of the master phase and creates a control signal that is sent to the Event Processor (Figure 13) through the Source Bus. The controlling signal represents pulses with a period equal to the master switching period but 180º out of phase with respect to the master. This controlling signal should be used as the turn ON event of the second phase.

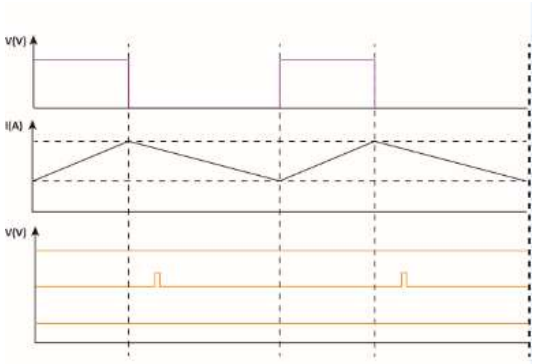


Figure 22 - Two Phases Interleave Mode Waveforms

Figure 23 provides an example of three-phase interleaved mode. The timers of the pair chosen as a master controls the switching of the first phase. The engine measures the switching period of the master phase and creates two controlling signals that are sent to the Event Processor (Figure 13) through the Source Bus. The first controlling signal represents pulses with a period equal to the master switching period. The first signal starts at one third of the period. This controlling signal should be used as the turn ON event of the second phase. The second controlling signal represents pulses with a period equal to the master switching period. The second signal starts at two thirds of the period. This controlling signal should be used as a turn ON event of the third phase.

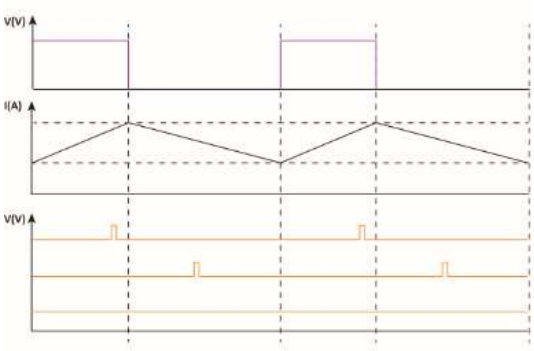


Figure 23 - Three Phases Interleave Mode Waveforms

An example of three-phase interleaved mode is provided in Figure 24. The timers of the pair chosen as a master controls the switching of the first phase. The engine measures the switching period of the master phase and creates three controlling signals that are sent to the Event Processor (Figure 13) through the Source Bus to control the other three phases. The first controlling signal represents pulses with period equal to the master switching period.

It starts at one fourth of the period. This controlling signal should be used as the turn ON event of the second phase. The second controlling signal represents pulses with a period equal to the master switching period. It starts at the half of the period. This controlling signal should be used as the turn ON event of the third phase. The third controlling signal represents pulses with a period equal to the master switching period. The third signal starts at three fourths of the period. This controlling signal should be used as the turn ON event of the fourth phase.

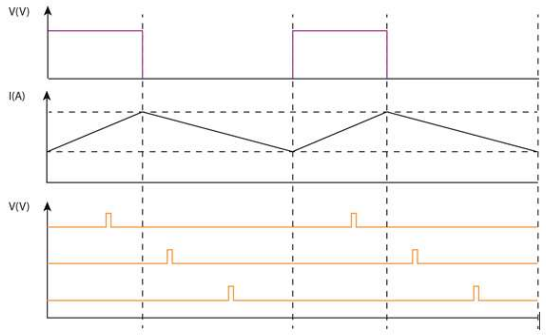


Figure 24 - Four phases interleave mode waveforms

For an example, a master phase can be set by the events coming from the CMPx_POS and CMPx_NEG comparators of one of the ANx_CMPx blocks. The CMPx_POS comparator is used as the source for the turn OFF event of the switch, while the CMPx_NEG comparator is used as the source for the turn ON of the switch. The slave phases are turned ON based on the signals coming from the Interleave Engine and turned OFF based on the CMPx_POS comparator.

Figure 25 to Figure 31 provide different simulation examples for a two-phase interleaved mode. In a real situation master and slave phases are shifted 180º, but in these examples, they are shown as being in phase (to be easier to visualize how the stability of the slave phase is ensured). It is well known that when using peak control only for the interleaved slave phase, it will naturally converge for duty cycles less than 50%, but it will naturally diverge for duty cycles greater than 50%. The interleave machine has a special feature to force the slave phase to converge for any duty cycle.

Figure 25 provides waveforms of the master and slave switching waveforms for a two-phase interleaved mode. The inductance of the inductor of the master phase, L_m, is equal to the inductance of the slave inductor, L_S. The master phase switches at a duty cycle smaller than 50%. The master and the slave phase are purposely set out of phase at the beginning, to show that after some cycles they are naturally converging and become in phase.

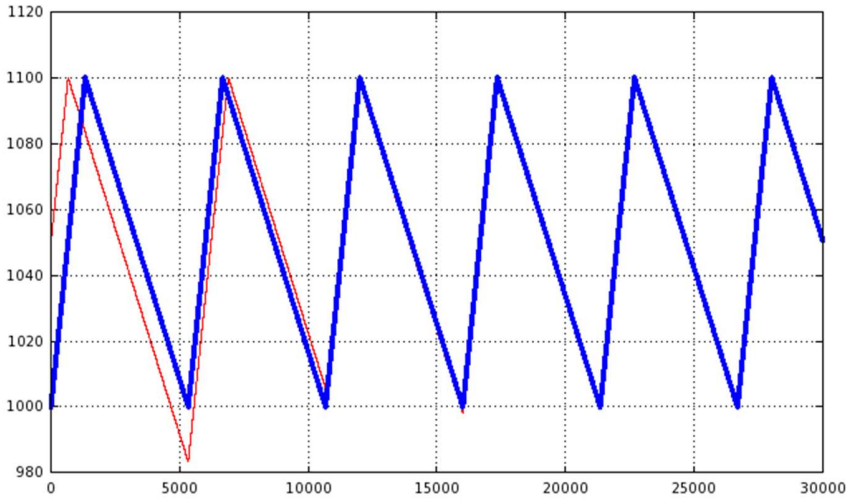


Figure 25 - Waveforms of a master (blue) and slave (red) switch forms with D< 50% and $L_m = L_S$

The inductors from the master phase and slave phase paths are supposed to be equal, but in reality, that will never occur; slight differences will be present. Figure 26 provides the waveforms of master and slave switching forms for a two phase interleave mode where the inductance of the master phase inductor, L_m, is slightly smaller than the inductance of the slave inductor, L_S. The master phase switches at a duty cycle smaller than 50%. The master and the slave phases are set out of phase at the beginning, but after some cycles they converge and become in phase. The two currents will not be exactly equal, but they will be close enough (depending on how close the two inductors are).

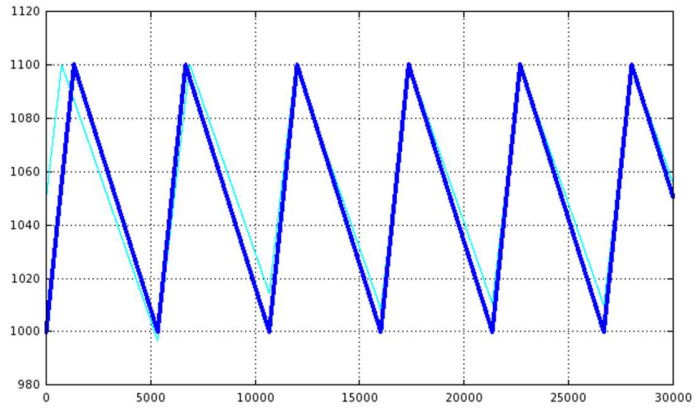


Figure 26 - Waveforms of a master (blue) and slave (light blue) switch forms with D< 50% and $L_m > L_S$

Figure 27 shows the waveforms of the master and slave switching forms for a two phase interleave mode where the inductance of the inductor of the master phase, L_m, is larger than the inductance of the slave inductor, L_S. The master phase switches at a duty cycle smaller than 50 %. The master and the slave phases are set out of phase from the beginning, but after some cycles they converge and become in phase.

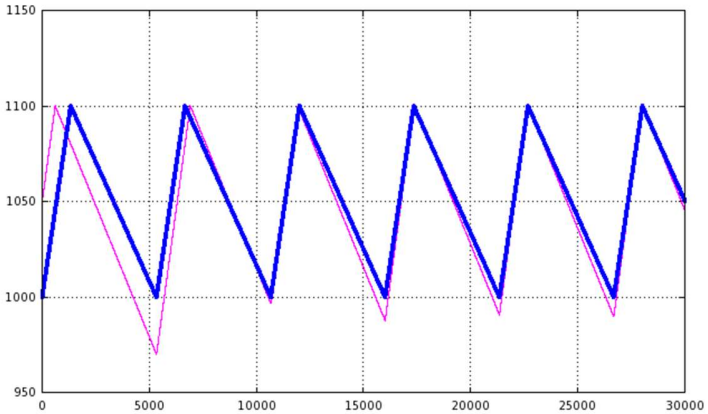


Figure 27 - Waveforms of a master (blue) and slave (pink) switch forms with D < 50% and $L_m < L_S$

Figure 28 shows the waveforms of the master and slave switching forms for a two phase interleave mode. The master phase switches exactly at 50 % duty cycle. The master and the slave phases are set out of phase from the beginning and never naturally converge. In this case, the current waveform finds a quasi-stable solution, which is not desired.

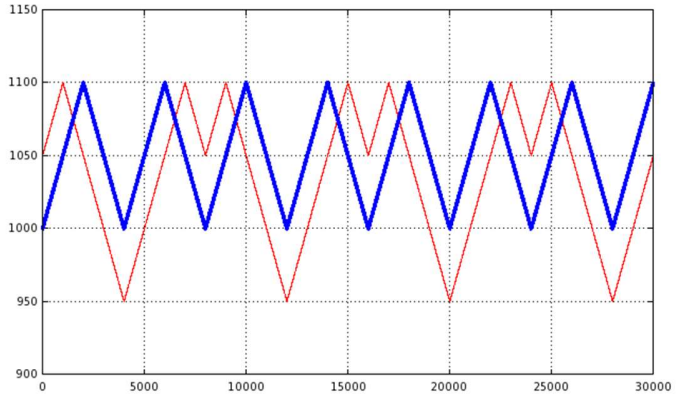


Figure 28 - Waveforms of a master (blue) and slave (rad) switch forms with D = 50%

Figure 29 shows the waveforms of the master and slave switching forms for a two phase interleave mode. The master phase switches at duty cycle higher than 50 %. The master and the slave phases are set in phase at the beginning and they naturally diverge.

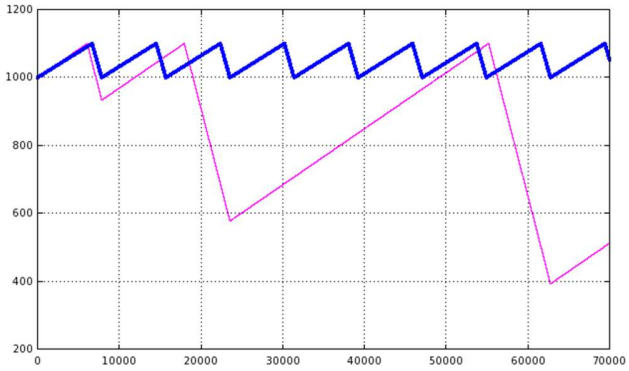


Figure 29 - Waveforms of a master (blue) and slave (pink) switch forms with D = 50%

To make the slave phase converge to the master for a duty cycle equal and larger than 50%, the dynamic maximum ON time (dynamic timeout) mode should be set for the slave. The EDT_SLAVE_MAX_ON_TIME field is used to set the slave maximum ON time.
00 – ON time DIV64 (max ON time is equal to $t_{ON} + t_{ON}/64$)
01 – ON time DIV 32 (max ON time is equal to $t_{ON} + t_{ON}/32$)
10 – ON time DIV 16 (max ON time is equal to $t_{ON} + t_{ON}/16$)
11 – ON time DIV 8 (max ON time is equal to $t_{ON} + t_{ON}/8$).
The bigger the slave max ON time, the bigger the current ripple will be, but the convergence will be faster. It is up to the designer to select the max ON time for his system.
Figure 30 shows the waveforms of the master and slave switching forms for a two phase interleave mode. The master phase switches at 50 % duty cycle and a dynamic maximum ON time is used. The master and the slave phase are unbalanced from the beginning and they converge after some switching cycles.

Figure 31 shows the waveforms of the master and slave switching forms for a two phase interleave mode. The master phase switches at duty cycle higher than 50 %. The master and the slave phases are set to be out of phase at the beginning and they converge after some switching cycles when dynamic maximum ON time is used.

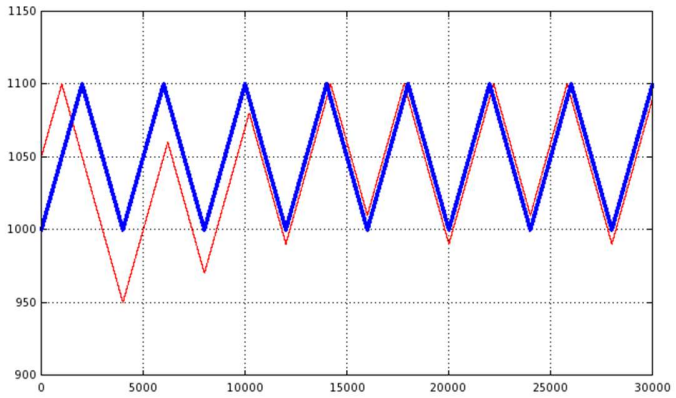


Figure 30 - Waveforms of a master (blue) and slave (read) switch forms with D = 50% and using dynamic maximum ON time

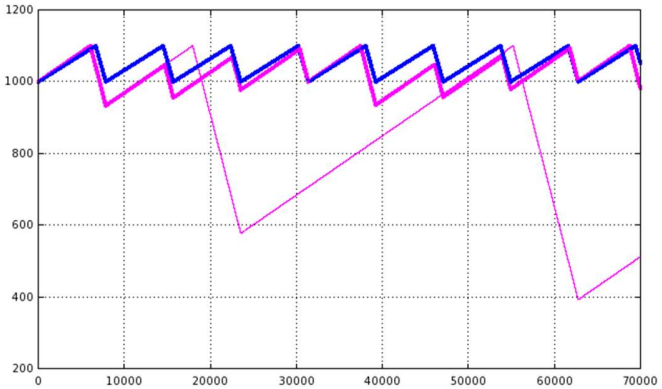


Figure 31 - Waveforms of a master (blue) and slave (read) switch forms with D> 50% for two cases:

- **Dynamic maximum ON time and not dynamic maximum ON time**

PWM Timer

The PWM timer block is a part of the HSA8000 Timing Engine (Figure 13). The PWM timing is only controlled by the software. It has four channels (CH. x for x equal from 0 to 3); every channel has direct and complementary outputs. The PWM timer outputs are connected to the Driver Controller block of the HSA8000 Switching Engine. The registers configuring PWM timers are listed below.

	PERIPHERAL.FIELD_NAME	Default
A_x	SWE_PWM->PWMx_ENABLE__U1	0b
B_x	SWE_PWM->PWMx_LOAD__U1	0b
C	SWE_PWM->PWM_SYNC__U3	0b
D_x	SWE_PWM->PWMx_STATUS__U1	0b
E_x	SWE_PWM->PWMx_ON_TIME__U16	0b
F_x	SWE_PWM->PWMx_PERIOD__U16	0b
G_x	SWE_PWM->PWMx_DEADTIME_HS__U16	0b
H_x	SWE_PWM->PWMx_DEADTIME_LS__U16	0b
I_x	SWE_PWM->PWMx_INIT_DELAY__U16	0b
J_x	SWE_PWM->PWMx_PHASE_STEP__U16	0b
K_x	SWE_PWM->PWMx_ON_TIME_PHASE_STEP__U16	0b
L_x	SWE_PWM->PWMx_MODE__U1	0b

PWM_ENABLE, PWM_LOAD, and PWM_STATUS registers

The PWM_ENABLE register enables the channels of the PWM timing engine. The register has four bits, one bit for each channel (A_x). Grouping all four bits into one single register allows the user to enable two or more channels simultaneously to ensure a defined phase relation between the channels at startup if necessary.

The PWM_LOAD register loads all PWM parametric registers at once while the timer is operating. The register has four bits for the PWM four-output channels (B_x). Similar to the PWM_ENABLE register, grouping the four bits into one single register allows the user to enable the loading of two or more channels simultaneously if necessary.

The PWM_STATUS register provides the status of the four PWM channels (C_x).

The **PWM_SYNC register** chooses the operation mode:

000 - All four channels are independent and cannot be synchronized (except at startup)
001 - CH. 0 and CH.1 are synchronized as a pair, CH.2 and CH.3 are independent
010 - CH. 0 and CH. 1 are synchronized as a pair, CH. 2 and CH. 3 are synchronized as a pair
011 - CH. 0, CH. 1 and CH.2 are synchronized, CH. 3 is independent.
100 - All four channels are synchronized.

All channels (synchronized or not) can be synchronized at startup. The synchronized channels have some extra features which allow them to be kept in synchronous mode while they are running and their (common) frequency and/or their relative phase are changed.

PWM_MODE register

The PWM timer has two modes of operation:

- Variable duty;
- 50 % duty.

For every channel, the PWM timer has 8 PWM parametric registers defining the pulse parameters. The table below shows the register names and their respective functionality (**x** is the channel number from 0 to 3).

Register Name	Functionality
PWMx_ON_TIME	Sets the pulse ON duration.
PWMx_PERIOD	Sets the pulse period.
PWMx_DEADTIME_HS	Sets the dead time between LS and HS (t_1 in Figure 32 Figure 32)
PWMx_DEADTIME_LS	Sets the dead time for LS rising (t_2 in Figure 32)
PWMx_INIT_DELAY	Sets the initial delay t_3 (Figure 33)
PWMx_PHASE_STEP	Shifts the phase step compared to the original t_4. (Figure 34)
PWMx_ON_TIME_PHASE_STEP	Duty cycle for one period when the phase step is changed (Figure 34)

The PWMx_ON_TIME register represents the ON duration of the x channel in ns. The register has 15 bits for the integer part of the ON duration time and 4 bits for the fraction part. It is the only register having a 4-bit fractional part; all other PWM registers represent integer numbers (programming resolution is 10 ns).
Example 1:
0x640 --> 100.0 clock ticks
The integer part is 100 and the fractional part is 0. The ON duration will be 100 clocks.
Example 2:
0x648 --> 100.5 clock ticks
The fraction part and the fractional part is 0.5. The ON duration will be 100 clocks for one switching cycle and 101 clocks for the next, so the average ON duration is 100.5.
Example 3:
0x641 --> 100.0625 clock ticks
The integer part is 100 and the fractional part is 1/16 = 0.0625. The ON duration for 15 switching cycles is 100 clock ticks and the ON duration for the next switching cycle is 101 clocks, so the average ON duration is 100 + 1/16 = 100.0625 clock ticks. The integer resolution of the ON time duration is 10 ns at the nominal clock frequency of 100 MHz. The resolution is 10 ns/16 = 625 ps when factional mode is used. The fractional mode is automatically activated when the last nibble of the PWMx_ON_TIME register is non-zero.
For designers using Helios, the numbers are automatically calculated.

The PWM0_PERIOD register represents the duration of the PWM period in ns; it is an integer value between 0 and 15 bits.

Figure 32 shows a time diagram of a PWM channel showing dead times between the complementary and direct outputs, t_1, and direct and complementary outputs, t_2.

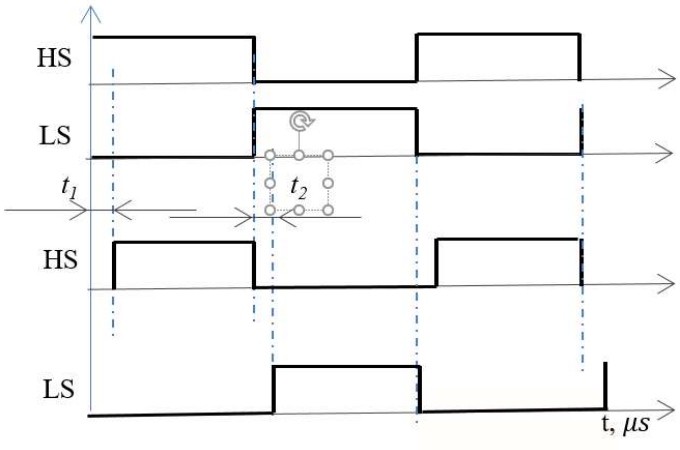


Figure 32 - Time diagram of a PWM channel showing dead times between the complementary and direct outputs

A channel can have an initial delay, t_3, which is the time between moment the enable signal is set and time the channel starts pulsing (Figure 33).

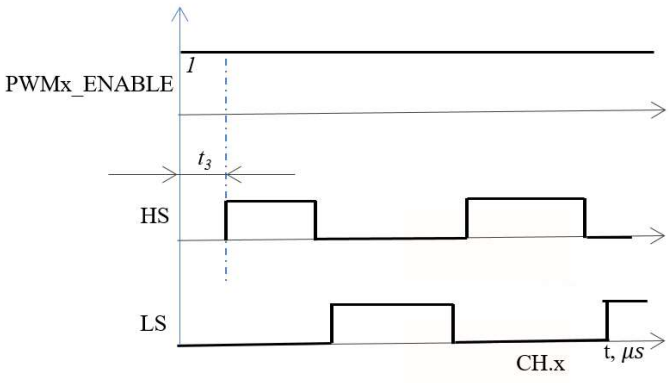


Figure 33 - Time diagram of a PWM HS channel showing an output with initial delay

Figure 34 shows a time diagram of a PWM timer Ch.x output signal before and after phase shifting. The period of the PWM signal is assumed to be T. When a phase shift (t_4) occurs, the period of the signal is adjusted for one cycle to be T+ t_4. After this cycle, the period of PWM timer Ch.x output signal returns to T. If the duty cycle is set to 50% (by setting PWMx_MODE register to "1"), then during the switching cycle which creates the phase shift the duty cycle is automatically preserved to 50%. Otherwise, for the phase shift cycle, the value of the ON time is adjusted with the (signed) value from the PWMx_PHASE_STEP field.

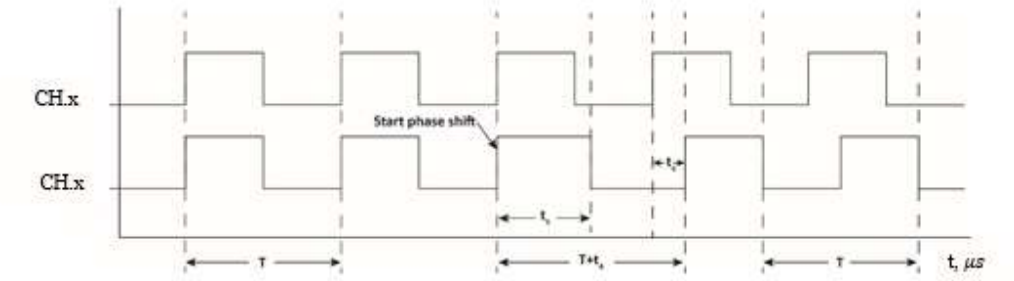


Figure 34 - Time diagram of PWM timer CH.x before and after phase shifting

The parametric registers which are used by the PWM engine can be loaded when the ENABLE register is set to "0" and the STATUS register is also cleared. When the ENABLE register is turned high, the PWM engine starts with the loaded parameters. In order to coherently modify the parametric registers while the PWM machine is running, the following sequence is recommended:

- Clear PWM_LOADS register.
- Modify the desired registers. The new values are stored in temporary buffers and do not affect the PWM timing.
- When done, set the PWM_LOADS register.
- The STATUS bit (i.e. load pending) is automatically set by the hardware.
- The new values stored in the temporary buffers are transferred all at once into the inner PWM registers just before the PWM is about to start a new switching cycle.
- When the new values have been loaded into the inner registers, the STATUS bit will be automatically cleared. (see Figure 35)

Clearing the PWM_LOADS register while PWMx_STATUS bit is high will cancel the pending load and will clear the status.

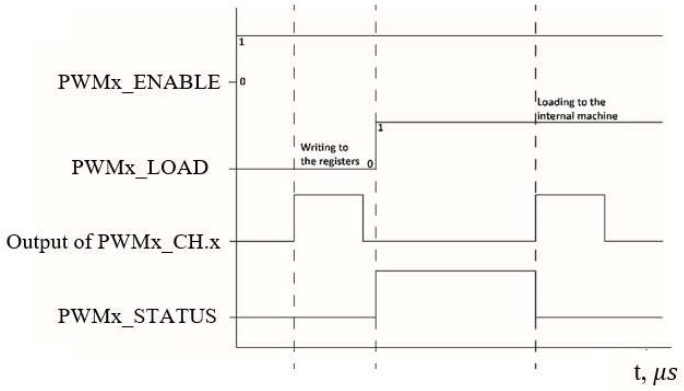


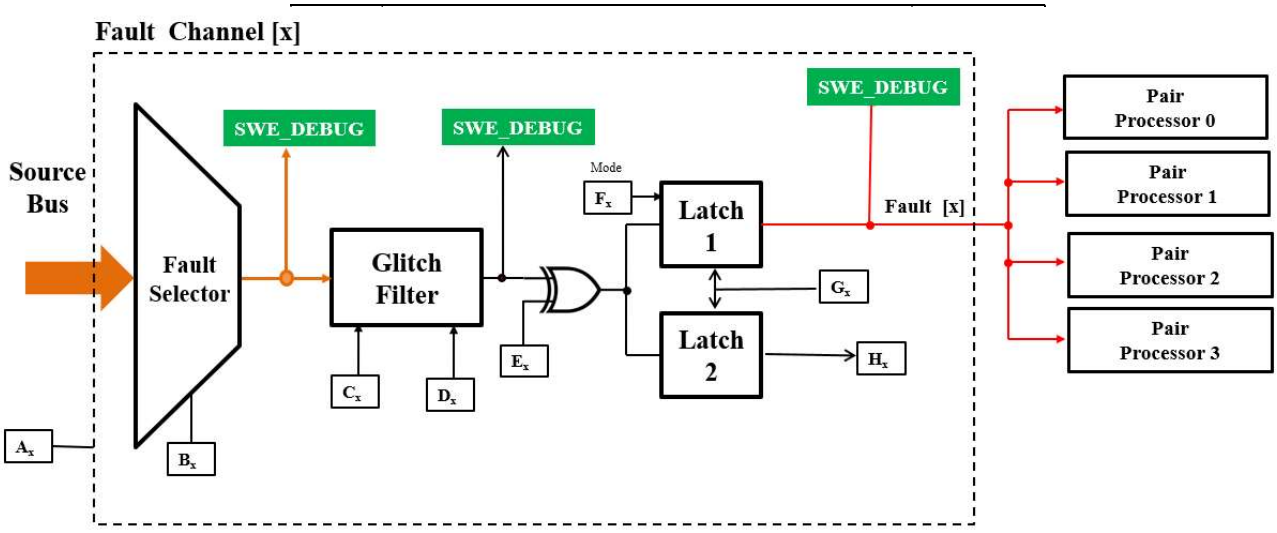
Figure 35 - Time diagram for using PWM_REGS_READY register for loading the data from the parametric registers

For debugging purpose, the LS and HS outputs of the PWM timers are connected to SWE_DEBUG. References that do not include the delays are also connected to SWE_DEBUG.

Fault Processing block

Fault Processing block selects and processes up to 8 faults from the Source Bus (sources from the analog interface and I/O block). It has 8 channels. Every channel sends fault signals to the 4 Pair Processors from (Figure 13). Figure 36 shows a Fault Channel[x] diagram (x is equal from 0 to 7). Every channel has a fault selector selecting one source signal from the Source Bus. A channel is enabled with (A_x) register.

Figure 36 - A Fault Channel [x] block diagram



The selected fault from the Source Bus by selector (B_x) can be optionally glitch-filtered (C_x) and inverted (E_x). The signal is then sent to the Latch1 and Latch2 blocks after the XOR gate. Latch1 generates the actual hardware

signal to force OFF the selected pair in real time in case of a fault. This latch can operate in two modes (F_x). The mode is set by FAULTx_LATCH_MODE bit:

- 0- transparent;
- 1 - stiky.

Latch2 feeds a status register readable through the AMBA bus and it is always a sticky bit. When Latch1 is programmed to sticky mode, Latch2 (the readable status bit) will mimic the logic state of Latch1 (actual hardware fault signal). However, if Latch1 is programmed in transparent mode, the status will be set along with Latch1 in case of fault. The status will remain set even after the fault condition has disappeared to inform the software that an actual fault condition occurred and disappeared. Clear of Latch1 and Latch2 can be done by FAULTx_CLEAR field (H_x).

The selected signal from the Source Bus, the filtered and the output signals are connected to SWE_DEBUG for debugging.

Driver Controller block

As shown in Figure 13, 8 pairs - 4 from Event Driven Timer block and 4 from PWM block are inputs to the Driver Controller. The block diagram of the Driver Controller block is shown in Figure 37.

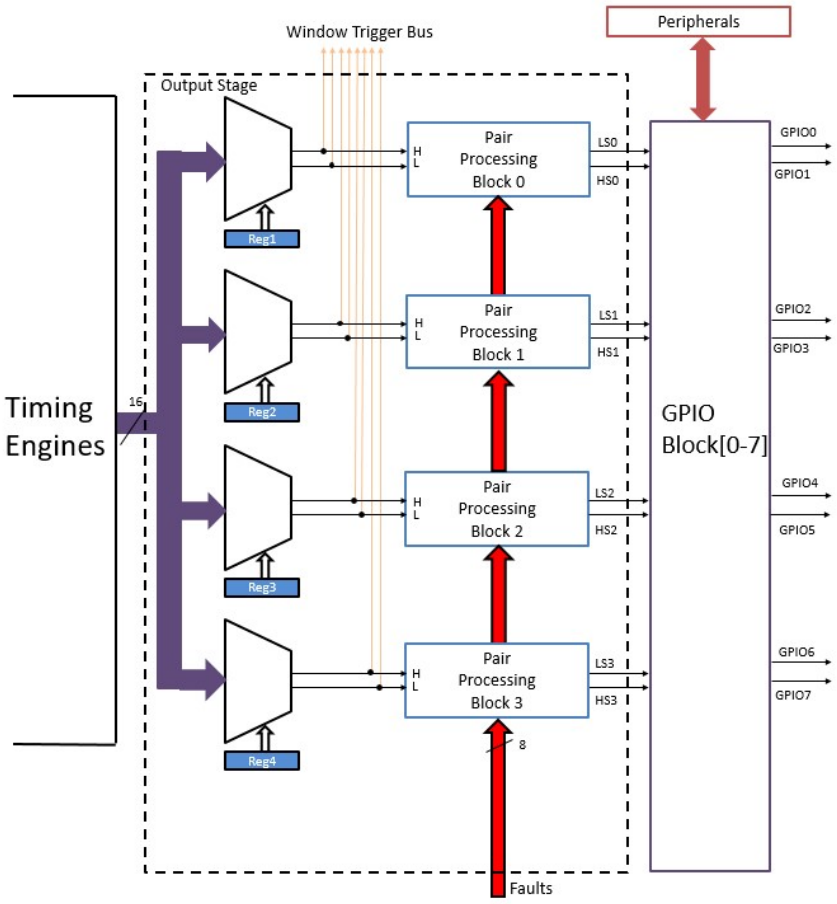


Figure 37 - The block diagram of the Gate Control block

The Driver Controller has 4 selectors that select 4 pairs to be exported to the driver pins. The selection is done through the DRIVER**x**_INPUT register (**x** is from 0 to 3). The selected timer and their corresponding bits are shown in the table below:

	Timer
0	PWM0
1	PWM1
2	PWM2
3	PWM3
4	EDT0
5	EDT1
6	EDT2
7	EDT3

The selected pair is sent to a Pair Processor[x] block whose diagram is shown in Figure 38.

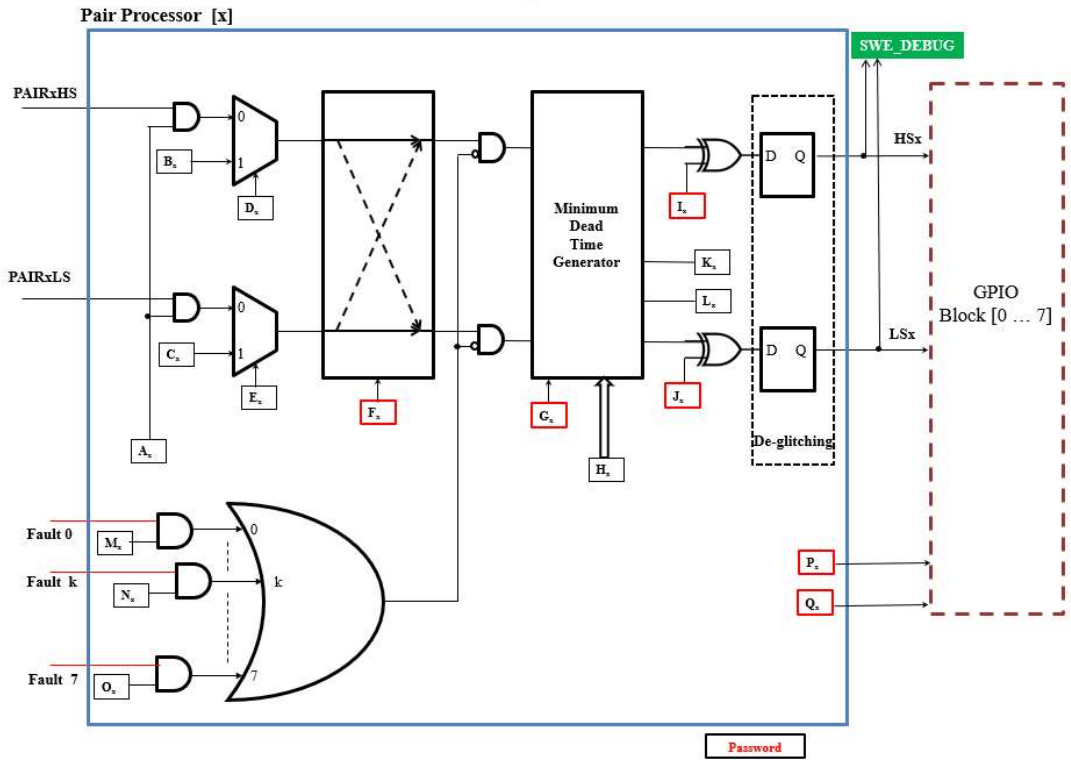


Figure 38 - The block diagram of the Pair Processor block

	PERIPHERAL.FIELD_NAME	Default
A_x	SWE_DRV->DRIVERx_ENABLE__U1	0b
B_x	SWE_DRV->DRIVERx_HS_FORCE__U1	0b
C_x	SWE_DRV->DRIVERx_LS_FORCE__U1	0b
D_x	SWE_DRV->DRIVERx_HS_FORCE_ENABLE__U1	0b
E_x	SWE_DRV->DRIVERx_LS_FORCE_ENABLE__U1	0b
F_x	SWE_DRV-> DRIVERx_PIN_SWAP__U1	0b
G_x	SWE_DRV->DRIVERx_OVERLAP_PROTECTION__U1	0b
H_x	SWE_DRV->DRIVERx_DEADTIME__U8	0b
I_x	SWE_DRV->DRIVERx_HS_INVERT__U1	0b
J_x	SWE_DRV->DRIVERx_LS_INVERT__U1	0b
K_x	SWE_DRV->DRIVERx_OVERLAP_STATUS__U1	0b
L_x	SWE_DRV-> DRIVERx_OVERLAP_STATUS_CLEAR __U1	0b
M_x	SWE_DRV->DRIVERx_FAULT0_ENABLE__U1	0b
N_x	SWE_DRV->DRIVERx_FAULTk_ENABLE__U1	0b
O_x	SWE_DRV->DRIVERx_FAULT7_ENABLE__U1	0b
P_x	SWE_DRV->DRIVERx_HS_MODE__U1	0b
Q_x	SWE_DRV->DRIVERx_LS_MODE__U1	0b
R	SWE_DRV->DRIVER_PASSWORD__U32	0b

To enable the pair processor block, A_x should be set to 1. The selected pair inputs can be overridden by setting the force mode bit (D_x for HS and E_x for LS). If the force mode is set to "1" for high side, the value that is set in B_x (for HS) and will be forced into the pair processor block output. Similarly, C_x is set for low side.
The HS and LS output signals can be swapped if necessary (F_x). The LS and HS outputs will be turned OFF if there is a fault on one of the fault condition inputs coming from the Fault Processing block. For this purpose, the fault inputs should be enabled (M_x, N_x, …, O_x).
To prevent the LS and HS switches from simultaneously turning ON, a hardware programmable minimum dead time between the two outputs (H_x) is present, ensured by the pair processing block itself.
In a case when two switches are not used as a HS and LS pair, the overlap protection feature can be disabled (G_x). The overlap status is shown in K_x and can be cleared with L_x.
The output signals, HS and LS, can be inverted (I_x and J_x respectively). They are also sent to SWE_DEBUG.
The output signals are passed through de-glitching flops before they reach the output pins in order to remove any combinatorial glitches. When the overlap protection is disabled, swapping HS/LS and inverting the outputs can have disastrous consequences if accidentally programmed erroneously. To avoid this, for every Pair Processing block the bits (in red color Figure **45)** are grouped into one register, DRIVERx_PROTECTED_SETTINGS which is password protected. Therefore, in order to change these bits, the value of the DRIVER_PASSWORD register, common for the Pair Processor blocks, must be set to 0x13579BDF before the settings register can be modified. It is highly recommended to clear the PASSWORD register once the procedure is finished. Attempting to modify any of the protected settings without having the correct password written into the PASSWORD register, will have no effect.

Capture (Timing Measurements) block

The Capture block measures the actual time interval between two processed events, GPIO signals, or any combination of those. The block has four channels: Measure Channel x (x = 0 to 3). Each of the channels contains three independent registers – current measurement and min/max values – which are continuously refreshed in the background. When a read request is issued for a particular channel, the three data registers from that channel will be transferred into the AMBA read registers (Figure 39). The AMBA read registers are common for all four channels, hence only one channel can be read at the time. The time intervals (measurements) are expressed in clock ticks (10 ns).

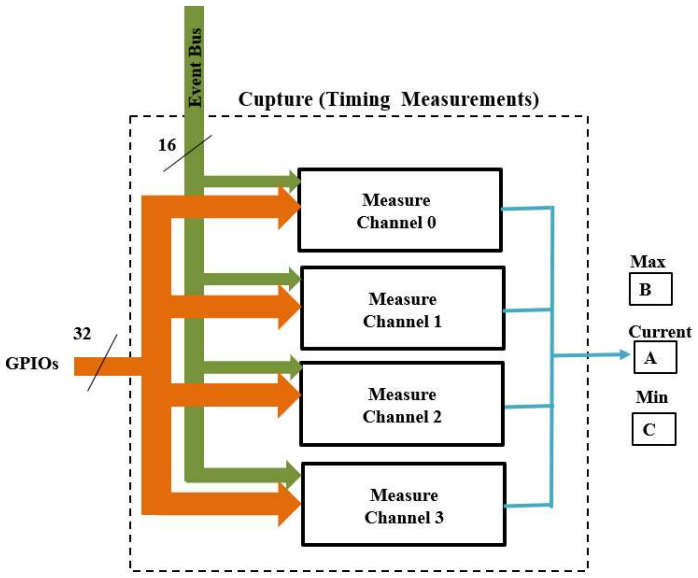


Figure 39 – Capture (Timing Measurements) block diagram

	PERIPHERAL.FIELD_NAME	Default
A	CAPTURE->TIME__U32	0b
B	CAPTURE->MAX_TIME__U32	0b
C	CAPTURE->MIN_TIME__U32	0b

A channel of the Capture block is shown in Figure 40. The channel has two selectors for selecting a start and a stop signal for the measurement. The selection is done by STARTx_INPUT and STOPx_INPUT. The measurement can be configured to start and stop on either (falling/rising) edge of the selected start/stop signals by STARTx_EDGE and STOPx_EDGE.

After specifying the signal and the rising/falling edge, the next step is to enable the measurement channel ENABLEx. When the measurement conditions are met, and the first measurement is available, the CAPTUREDx flag is set by the hardware. A read request (READ_REQUEST) for that channel should be issued, which will

transfer the most recent measurement into the read buffers and will clear the CAPTUREDx flag. After reading, the CAPTUREDx flag will be set again automatically when a new measurement is performed.

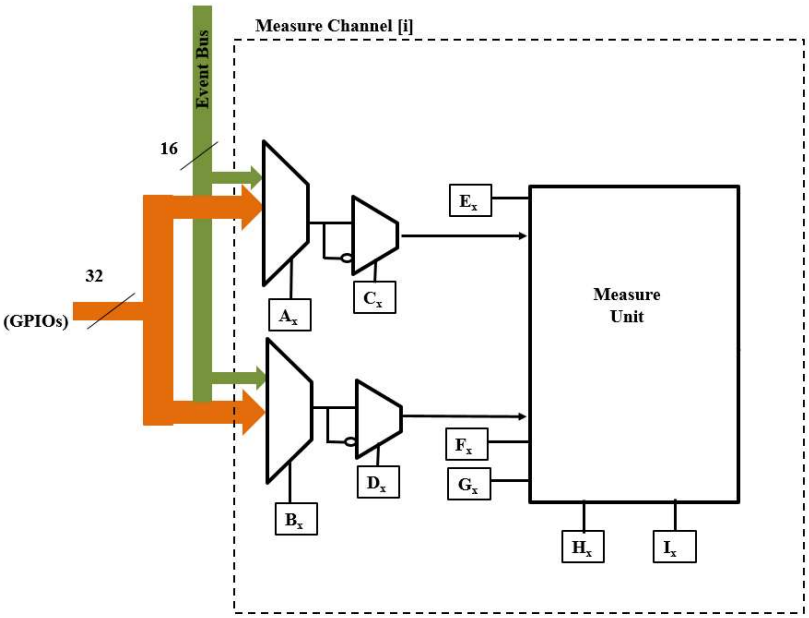


Figure 40 - A channel of the Capture (Timing Measurements) block

	PERIPHERAL.FIELD_NAME	Default
A_x	CAPTURE->STARTx__U6	0b
B_x	CAPTURE->STOPx__U1	0b
C_x	CAPTURE->STARTx_EDGE__U1	0b
D_x	CAPTURE->STOPx_EDGE__U1	0b
E_x	CAPTURE->ENABLEx__U1	0b
F_x	CAPTURE->ENABLEx_MAX__U1	0b
G_x	CAPTURE-> ENABLEx_MIN__U1	0b
H_x	CAPTURE->CAPTUREDx__U1	0b
I_x	CAPTURE->READx_REQUEST__U1	0b

If the minimum and/or maximum functions are enabled, the minimum and/or maximum values of the currently measured data will be recorded in the dedicated min/max registers.
The maximum and minimum values can be reinitialized by disabling and then re-enabling the maximum and minimum functions respectively.

Compare (Timing Generator)

The Timing Generator can create four signals with the same period but different phases. The period of the pulses in ns, is given by the PERIOD register. A tick is equal to 10 ns. The start of every pulse depends of the value written in the TIMEx register (**x** is the output signal number from 0 to 3). The output signals are sent to Source Bus and SWE_DEBUG.

PERIPHERAL.FIELD_NAME	Default
COMPARE->ENABLE__U1	0b
COMPARE-> LOAD__U1	0b
COMPARE-> TIMEx_COMP__U16	0b
COMPARE-> PERIOD__U16	0b

The LOAD resister is used to start the sequences. Figure 41 provides an example showing the four outputs when the period of the input signals is set to 1 µs.
TIME0 is set to 200 ns.
TIME1 is set to 400 ns.
TIME2 is set to 600 ns.
TIME3 is set to 800 ns.

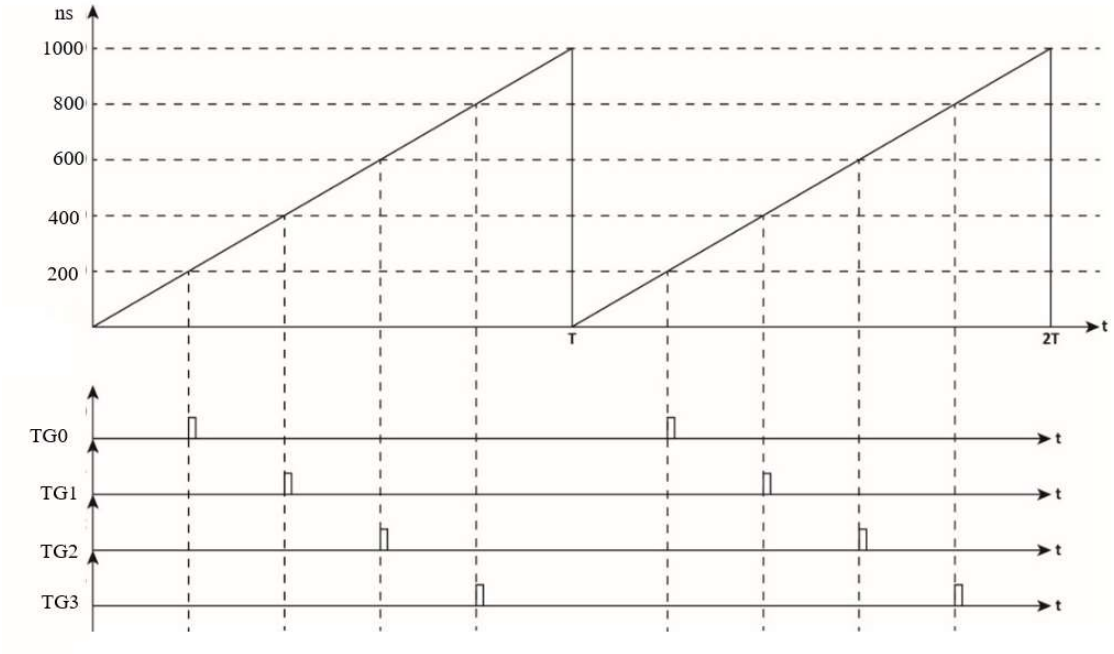


Figure 41 - Example for Output Signals of Timing Generator

AC PLL

AC PLL block synchronizes the HSA8000 controller with the grid frequency. The basic functions of AC PLL block are:

- lock on the grid frequency (50/60Hz) based on zero-crossings;
- track the drift of the grid frequency up to 0.1 Hz/s;
- generate an integer multiple of the grid frequency used for processing in the CPU program.

AC PLL can be used for DC to AC inversion and ensures efficient grid current injection operation or in PFC applications to generate grid current reference. The grid voltages are scaled down by voltage dividers and then applied to the subtractors inputs (pins AN12_ACP and AN13_ACN).

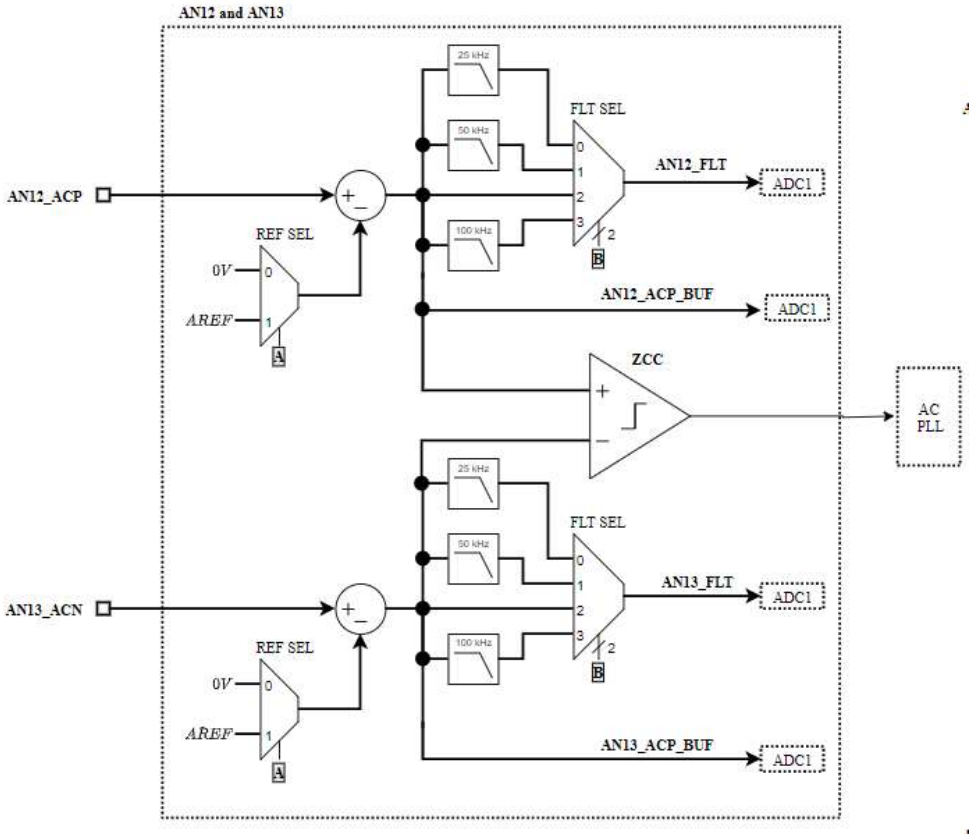


Figure 42 - Blocks associated with analog inputs AN12_ACP and AN13_ACN.

The subtractors outputs are inputs to the zero-crossing comparator, ZCC, as shown in Figure 42.
The block diagram in Figure 43 illustrates the AC PLL main blocks:

- Glitch filter;
- Grid frequency measurements;
- Grid reconstruction;
- Period divider:
- Interrupt request (IRQ) generation;
- Configuration and report registers.

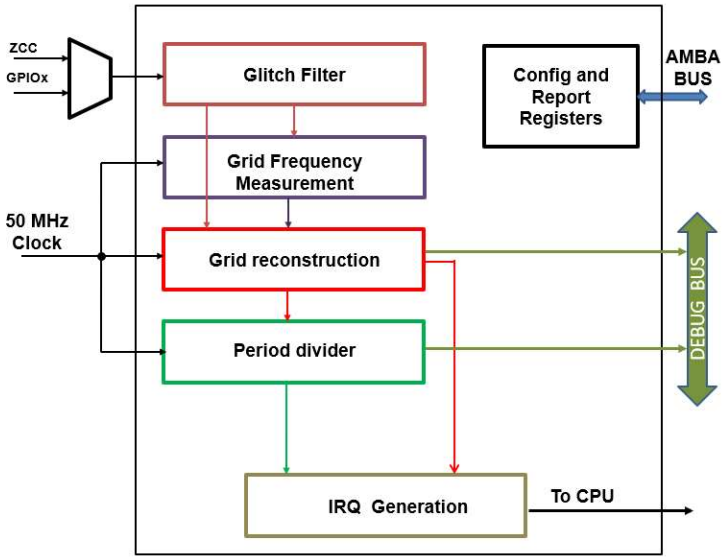


Figure 43 - AC PLL block diagram

To estimate the grid frequency, AC PLL should be set to acquisition mode. During acquisition mode AC PLL is not locked. The acquisition mode is performed during HSA8000 start. It also can be forced by the HSA8000 CPU.

Glitch filter

For the glitch filter input, ZCC output or any of the GPIOs can be selected. When ZCC is selected, the comparator should be enabled. The AC PLL input configuration is done by field of the INPUT_CONFIG register.

FIELD_NAME	**Default**	
AC_PLL->ZC_COMP_ENABLE__U1	0x1	Enables ZCC
AC_PLL->INPUT_SELECT__U1	0x0	0 - selects ZC comparator 1 - selects a GPIO
AC_PLL->INPUT_INVERT__U1	0x0	This bit inverts the strobe going to the comparator.
AC_PLL->GPIO_PIN_FOR_INPUT__U5	0x0	Select GPIO

The filter can be set to consecutive or integration mode. The glitch filter is controlled by the following fields:

PERIPHERAL.FIELD_NAME	Default
AC_PLL->GLITCH_WIDTH__20ns_U15	0x7FFF
AC_PLL-> GLITCH_MODE__U1	1b

Grid frequency measurements

AC PLL can be synchronized with the grid frequency in different programed limits. The period limits are provided for 50 Hz and 60 Hz through programmable registers. For each frequency, there are four limits.

Figure 44 shows the Grid frequency measurement block diagram. The block measures the grid period and locks the PLL if the frequency is in the range. For this purpose, the signal from the glitch filer is applied to the "Capture grid rising and falling edges" block. The outputs of the block are the negative and positive grid durations. From the two durations, period and asymmetry are calculated in the "ADD/Subtract" block and their values are stored in the report registers. When the period is in the wide limits for time, t_{lock}, PLL enables smaller limits locking mode. The wide period limits and the lock time are set by the used in the config registers. The durations, period and asymmetry values are reported by the report registers.

PERIPHERAL.FIELD_NAME	Default
AC_PLL->MAX_50HZ_WIDE_PERIOD__REG	0x103B95
AC_PLL->MIN_50HZ_WIDE_PERIOD__REG	0xE6524
AC_PLL->MAX_60HZ_WIDE_PERIOD__REG	0xD6288
AC_PLL->MIN_60HZ_WIDE_PERIOD__REG	0xC1C32
AC_PLL-> LOCK_DELAY__REG	0x20
AC_PLL-> REPORT_HALF_PERIOD_POSITIVE_WIDTH__REG	0x0
AC_PLL-> REPORT_HALF_PERIOD_NEGATIVE_WIDTH__REG	0x41
AC_PLL->REPORT_PERIOD_WIDTH__REG	0x41
AC_PLL->REPORT_PERIOD_ASYMETRY__REG	0x3FFFBF

The period limits for 50 Hz and 60 Hz can be programed by a user. For example, if the user wants the wide range frequency f to be 47 Hz, then the period is:

$$T = \frac{1}{f} = \frac{1}{47} = 21{,}276{,}595\ ns.$$

Since 1 clock is 20 ns, the value that should be written in the register will be:

$$T/20 = 1063829 = 0x103B95.$$

When the "smaller limits locking mode" is enabled, the calculated period and asymmetry are filtered by 1st order low pass filter and then checked with the smaller range limits given in the config registers. The period limits and the low pass filter alpha parameter can be programed by the user. The filtered period and asymmetry are reported by the report registers.

PERIPHERAL.FIELD_NAME	Default
AC_PLL-> MAX_50HZ_PERIOD__REG	0x103B95
AC_PLL-> MIN_50HZ_PERIOD__REG	0xEF5A8
AC_PLL-> MAX_60HZ_PERIOD__REG	0xCEE61
AC_PLL-> MIN_60HZ_PERIOD__REG	0xC81D8
AC_PLL-> PERIOD_ASYMETRY_ALPHA__REG	0x4000
AC_PLL-> PERIOD_FILTER_ALPHA__REG	0x4000
AC_PLL-> REPORT_FILTERED_PERIOD_WIDTH__REG	0
AC_PLL-> REPORT_FILTERED_PERIOD_ASYMETRY__REG	0

The output of the Grid frequency measurement block is the filtered frequency.

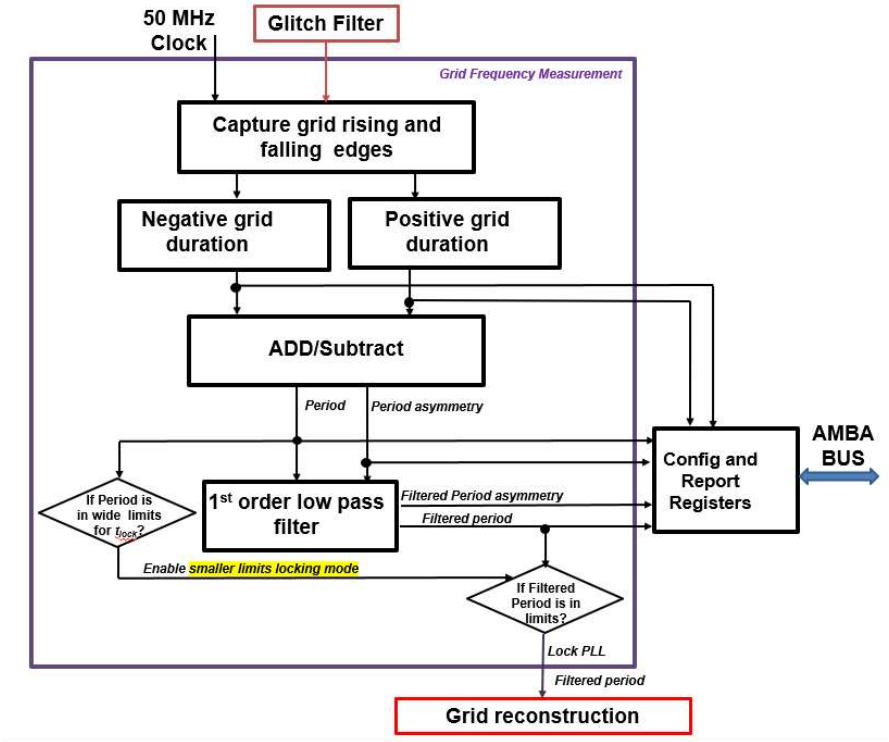


Figure 44 - Grid frequency measurement block diagram

Grid regenerations

The grid regeneration block diagram is shown in Figure 45.

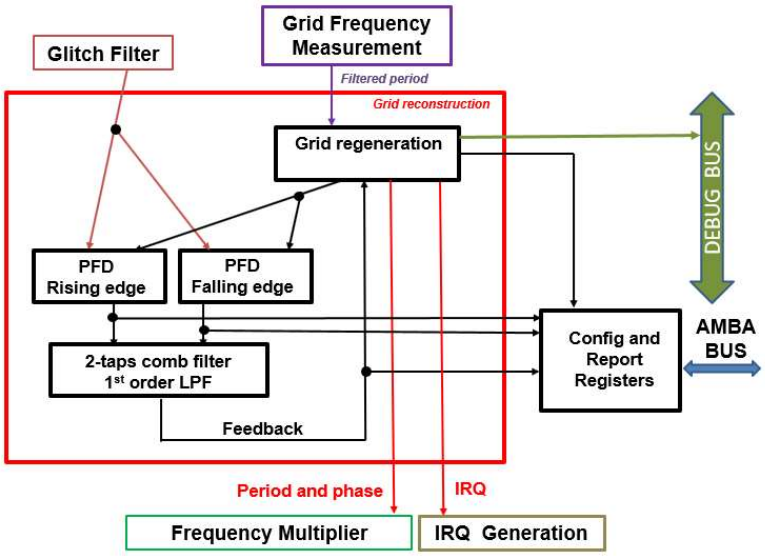


Figure 45 - Grid regeneration block diagram

The filtered period from the "Grid Frequency Measurement" block is sent to the "Grid regeneration" block for grid regeneration. The grid and the regenerated grid are dual phase/frequency detectors (PFD) for compensating any asymmetries due to imperfect rising edge/falling edge timing matching of ZCC and not exactly 0 threshold.

The detected phase errors by PFDs are filtered by a 2-taps comb filter to completely cancel the asymmetries due to the grid comp offset and due to the differences between rising and falling edges and a 1st order low-pass filter after the comb which guarantees stability.
Grid regeneration counter is:

- real-time frequency and phase programmability;
- when measured frequency and phase are steady for a certain period of time, the counter is preloaded with the best estimate of the frequency, initial phase, and the PFDs are enabled;
- it generates interrupts at the rising edge of the grid signal;
- the reconstructed grid can be observed through DEBUG BUS.

Once the PLL is locked:

- it tracks the frequency as long as the Grid frequency measurement block is in “smaller limits locking mode”;
- short term grid glitches are reported to the software immediately, but are not fed to the filters nor PFDs, therefore the grid regenerator keeps running with the previous frequency and phase;
- software decides what short-term means and forces a new acquisition or not.

Period divider

Period divider block diagram is shown in Figure 46. The regenerated grid period is used as reference for the period divider.

- the regenerated grid has 50% duty (or very close) of filtered period;
- the period division is recalculated at every half-grid cycle;
- rising edge of the divided signal is detected and sent to create an interrupt;
- the divider output can be seen through DEBUG BUS.

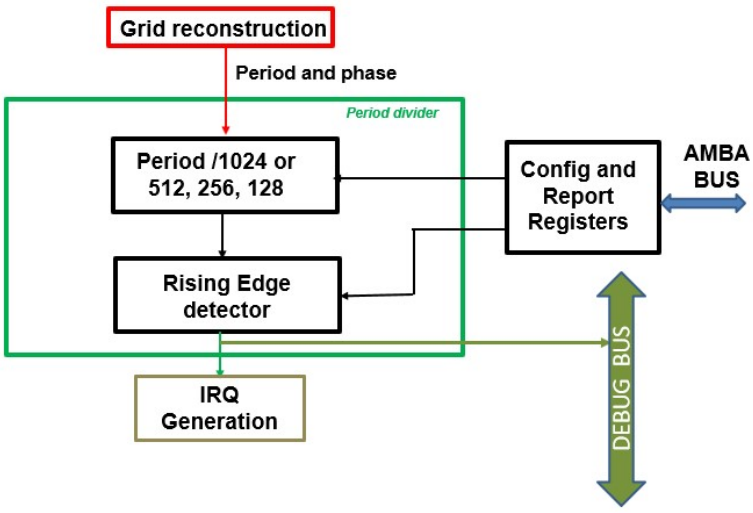


Figure 46 – Period divider block diagram

The period is divided by a number, N, set by SAMPLES_PER_GRID_PERIOD. The number can be 32, 64, 128, 256, 512, and 1024. The divider output is a sequence of N pulses (samples). By default, the samples per grid

period is 256. Every sample sets IRQ to the CPU. During operation, the sample number can be seen by SAMPLE_NUMBER register.

For debugging, DEBUG_SAMPLE_NUMBER_OUTPUT register is used. The register sets the sample number that will be output on DEBUG.

PERIPHERAL.FIELD_NAME	Default
AC_PLL-> SAMPLES_PER_GRID_PERIOD__REG	x3
AC_PLL-> SAMPLE_NUMBER__REG	
AC_PLL-> DEBUG_SAMPLE_NUMBER_OUTPUT__REG	0

IRQs generation

PLL generates interrupts by signals coming from the Rising Edge detector of the Period divider block (Figure 45) and Grid generation of the Grid reconstruction block (Figure 46). Enabling IRQs generation is done by the following fields:

PERIPHERAL.FIELD_NAME	Default
AC_PLL-> RECONSTRUCTED_AC_CROSSING_IRQ_ENABLE__U1	1
AC_PLL-> RECONSTRUCTED_AC_SAMPLES_IRQ_ENABLE__U1	1

Anti-islanding

During islanding, the divided period is artificially set slightly smaller within each half grid period. As a result, the phase error accumulates during the current grid half period. If the grid is driving the PLL, then the phase error is reset at the beginning of a new half grid while if the PLL is driving the grid, then the resonant component of the (missing) grid drives the next comparator edge sooner and sooner and the frequency eventually goes out of limits.

AC PLL status

The PLL status is checked by the CPU by reading the STATUS register. When a condition to generate an interrupt request is met, the corresponding interrupt flag in the register is set. The flag is set regardless of the state of the interrupt enable bit from register IRQ_ENABLES, but the hardware IRQ (interrupt request) is generated only if the interrupt enable bit is set. The interrupt service routine reads the IRQ flags to detect which event generated the IRQ, and then it must
clear the corresponding flag by writing a 1 into the corresponding control bit. The STATUS register fields and their descriptions are given below:

FIELD_NAME	**Default**	
AC_CROSSING_IRQ__U1	0x0	IRQ capture grid flag status bit. Write 1 to clear.
RECONSTRUCTED_AC_CROSSING_IRQ__U1	0x0	IRQ capture reconstructed grid status bit. Write 1 to clear
RECONSTRUCTED_AC_SAMPLES_IRQ	0x0	IRQ compare frequency multiplier flag status bit. Write 1 to clear.
NO_GRID	0x0	Grid not detected status bit.
WIDE_PERIOD_IN_RANGE	0x0	Grid measurement period fast in range status bit.
FILTERED_PERIOD_IN_RANGE	0x0	Grid measurement period filtered in range status bit.
WIDE_PERIOD_TRACKING	0x0	Grid measurement period fast mode status bit.

NOT_LOCKED	0x0	Grid reconstruct not locked status bit.
GRID_COMP_FILTERED	0x0	Grid comparator filtered status bit.
FREQUENCY_MODE	0x0	Grid measurement period 50 Hz detected status bit. 0 = 60 Hz 1 = 50 Hz
AC_PHASE	0x0	Grid reconstruct positive status bit. 0 = NEGATIVE 1 = POSITIVE

Serial Peripheral Interface

The HSA8000 Serial Peripheral Interface (SPI) implements full-duplex, synchronous, serial communications, and has the following capabilities:

- master operation
- programmable word size (8 or 16-bits), bit ordering (MSB first / LSB first), clock polarity and phase, and bit rate

Figure 47 shows a block diagram of the HSA8000 SPI.

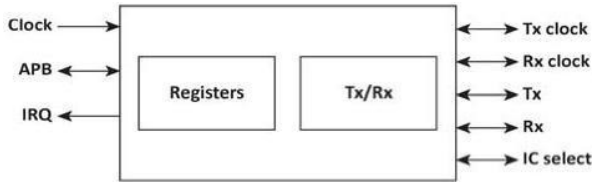


Figure 47 - SPI block diagram

Data to be transmitted over the serial interface should be written to the TX_DATA register. The register should not be written to while the TX_FULL bit in the STATUS register is set,
otherwise data loss may occur. When the BIT_SIZE field in the CONTROL register is set to 0, only the lower 8-bits of data written to the register are transmitted, when the field is set to 1, 16-bits of the data are transferred.
Data that is received over the serial interface can be read in the RX_DATA register. When the BIT_SIZE field in the CONTROL register is set to 0, only the lower 8-bits of the register contain valid data.
The STATUS register contains a selection of flags that indicate the current status of the SPI. To clear a bit in the register, write a 1 to it. Writing 0 will leave it unchanged. In table below is given names, decryptions and values its fields:

Fail Name	Description	Values
TX_EMPTY	Transmits buffer empty.	0 - Not empty 1 - Empty
TX_FULL	Transmits buffer full	0 - Not full 1 - Full
TX_OVERFLOW	Transmits buffer overflow	0 - No overflow 1– Overflow
TX_UNDERRUN	Transmits buffer underrun	0 - No underrun 1 - Underrun
RX_EMPTY	Receives buffer empty	0 - Not empty 1 - Empty
RX_FULL	Receives buffer full	0 - Not full 1 - Full
RX_OVERFLOW	Receive buffer overflow	0 - No overflow 1– Overflow

The CONTROL register contains a selection of flags that control the operation of the SPI. In table below is given names, decryptions and values of its fields:

Fail Name	Description	Values
ENABLE	Enables the SPI. When disabled, data will not be received or transmitted.	0 - Disabled 1 - Enabled
MODE	Operating mode	0 - Master 1 - Slave
WORD_SIZE	Word size	0 - 8-bits 1 - 16-bits
BIT_ORDER	Bit ordering	0 - MSB first 1 - LSB first
CLOCK_EDGE	Clock polarity	0 - Idle low 1 - Idle high
IDLE_STATE	Transmit interrupt enable	0 - Disabled 1 - Enabled
TX_IRQ_ENABLE	Transmit interrupt enable	0 - Disabled 1 - Enabled
RX_IRQ_ENABLE	Receive interrupt enable	0 - Disabled 1 - Enabled

BIT_WIDTH shows how many ns is each UART bit. The bit duration is reciprocal of the baudrate. Since the baudrate is 115200, the value of the BIT_WIDTH register in ns is:

$$\text{BIT\_WIDTH register} = \frac{1}{115200\frac{\text{bit}}{\text{s}}} = 8680\ ns$$

The BIT_WIDTH

The value of the BIT_WIDTH register, depends on the CPU clock frequency, CPU_{Clk}, and the baudrate, BR. Since, the CPU clock frequency is 50 MHz and the BIT_WIDTH register in clock cycle is:

$$\text{BIT\_WIDTH register } = 8680 * 10^{-9}\ s * 50\ 000\ 000\ \frac{\text{clock}}{\text{s}} = = 434 \text{ clocks /bit} = 1B2(HEX)$$

DEVICE_SELECT is a single bit used to control the GPIO14_SEN output (for bootloader portability), otherwise the user can use any available GPIO to generate SEN function.

UART interface

There are two UARTs in the HSA8000. UART0, connected by default as alternate function to pins GPIO8_TX0 and GPIO9_RX0, is a general purpose one. UART1 is connected by default as alternate function to pins GPIO10_HDLC_TX1 and GPIO11_HDLC_RX1. It can be used as a general purpose one, or it can be configured to work with the hardware HDLC interface.

The UARTs have the following capabilities:

- 7 or 8 data bits
- 1 or 2 stop bits
- parity bit (None / Even / Odd / Mark / Space)
- programmable bit rate

Figure 48 shows a block diagram of the HSA8000 UARTs.

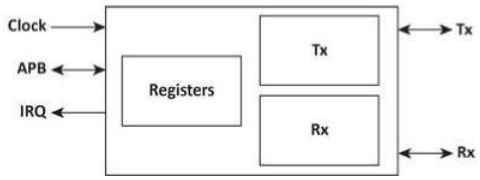


Figure 48 - UART interface block diagram

UART0

Data to be transmitted over the serial interface should be written to the lower 8 bits of the TX_DATA register. The register should not be written to while the TX_FULL bit in the STATUS register is set, otherwise data loss may occur.

Data that is received over the serial interface can be read in the lower 8 bits of the RX_DATA register.

The STATUS register contains a selection of flags that indicate the current status of the UART. To clear a bit in the register, write a 1 to it. Writing 0 will leave it unchanged. In table below is given names, decryptions and values its fields:

Fail Name	Description	Values
TX_EMPTY	Transmits buffer empty.	0 - Not empty 1 - Empty
TX_FULL	Transmits buffer full	0 - Not full 1 - Full
TX_OVERFLOW	Transmits buffer overflow	0 - No overflow 1– Overflow
RX_EMPTY	Receives buffer empty	0 - Not empty 1 - Empty
RX_FULL	Receives buffer full	0 - Not full 1 - Full
RX_OVERFLOW	Receive buffer overflow	0 - No overflow 1– Overflow
RX_PARITY	Parity error	0 - No error 1 - Parity error
RETRY_LIMIT_REACHED	Retry limited reached.	0 – Not reached 1 – Reached

The CONTROL register contains a selection of flags that control the operation of the UART. In table below is given names, decryptions and values of its fields:

Fail Name	Description	Values
ENABLE	Enables the UART. When disabled, data will not be received or transmitted.	0 - Disabled 1 - Enabled
STOP_BITS	Number of stop bits	0 - 1 stop bit 1 - 2 stop bits
DATA_BITS	Number of data bits	0 - 8 data bits 1 - 7 data bits
PARITY_MODE	Parity bit control	0 - No parity bit 1 - Even parity 2 - Odd parity 3 - Mark 4 - Space
FLOW_CONTROL	Flow control	0 - No flow control 1 - RTS/CTS
BREAK	Transmit break. When set, the transmit data line will be held low, signaling a break	0 - Do not send break 1 - Send break
TX_IRQ_ENABLE	Transmit interrupt enable	0 - Disabled 1 - Enabled
RX_IRQ_ENABLE	Receive interrupt enable	0 - Disabled 1 - Enabled
RX_BREAK_IRQ_ENABLE	Receive break interrupt enable	0 - Disabled 1 - Enabled
LIMIT_REACHED_IRQ_ENABLE	Retry limited reached interrupt enabled	0 - Disabled 1 - Enabled
DUPLEX	Duplex	0 – Full duplex 1 – Half duplex

BIT_WIDTH is a 16-bit register that specifies how many cycles of the CPU clock, each bit is transmitted for. Use of a 16-bit register provides support for a wide range of clock frequencies and baud rates.

UART1 HDLC

TX_CONTROL is the HDLC control field to be transmitted.
TX_ADDRESS shows the HDLC address field to be transmitted.
TX_DATA shows the HDLC data field to be transmitted. Writing to this field will trigger a transmission of an HDLC packet that consists the current contents of the control (TX_CONTROL), address (TX_ADDRESS) and data (TX_DATA) fields.
RX_CONTROL shows the HDLC control field of the last received HDLC packet.
RX_ADDRESS is the HDLC address of the last received HDLC packet.
RX_DATA - HDLC data field of the last received HDLC packet. Reading this field will send an acknowledgement to the RX framer thereby enabling the receipt of the next packet. Until this field is read, the RX framer will hold off all new incoming packets. Pending packets will be stored in the RX FIFO.
RX_CRC - HDLC cyclic redundancy check of the last received HDLC packet.

The fields of the CONTROL register control UART1 HDLC. In the table below are given names, decryptions and values of its fields:

Fail Name	Description	Values
CLEAR_COUNTERS	Clear both TX_PACKET_COUNT and RX_PACKET_COUNT registers. This bit is self clearing therefore will always read back as '0'.	0 – Read back 1 - Clear
PARITY_MODE	Parity control	0 - Even parity 1 - Odd parity
PARITY_ENABLE	Enables parity insertion on the UART byte transmission and parity checking on the UART byte reception/	0 - Disabled 1 - Enabled
HDLC_ENABLE	Enable HDLC framers. Both fields: UART_ENABLE and HDLC_ENABLE need to be set for proper HDLC operation.	0 - Disabled 1 - Enabled
UART_ENABLE	Enable the UART interface	0 - Disabled 1 - Enabled

Status register has two fields that provide flow control information:
TX_BUSY – is 1 when the TX framer is busy transferring the HDLC packet to the UART interface (via the TX FIFO). If the TX FIFO is full, the TX framer will keep TX_BUSY until the complete HDLC packet is transferred successfully.
RX_READY is 1 when data is ready to be read.

- During non-HDLC operation, RX_READY register signals that RX_DATA register is ready to be read.
- During HDLC operation, RX_READY indicates that RX_CONTROL, RX_ADDRESS, RX_DATA and RX_CRC are ready to be read. RX_READY will remain in 1 until RX_DATA is read thereby acknowledging to the RX framer that data was received.

I²C serial interface

The HSA8000 has an I²C-compatible master interface which implements a subset of the I²C protocol with the following restrictions:

- Master only (no slave supported)
- 7-bit address space only
- No clock stretching
- Only 1 or 2 bytes of data (programmable through a register, default 2)

Read/write messages are sent in the following format:

- 1 byte device address
- 1 byte register address
- 1 or 2 bytes of data (programmable,).

Data to be transmitted over the I²C interface should be written to the WRITE register.
The Write register should not be written to while the BUSY bit in the DATA_AND_STATUS register is set, otherwise data loss may occur.
WRITE register consist of three fields:
- WRITE_DATA – the size of the transmitted data depends on setting of the DATA_MODE field. The data bits can be either 16 bits, when DATA_MODE is set to 1, or 8 bits when is set to 0.
- WRITE_REG_ADDRESS – shows the 8 bits register address within the device the data should be transmitted.
- WRITE_DEV_ADDRESS – shows 7 bits device address to which the data should be transmitted.

Data that is received over the I2C interface can be read in the READ register. The register has two fields:
READ_REG_ADDRESS shows 8 bits register address within the device the data should be read from.
READ_DEV_ADDRESS shows 7 bits device address from which the data should be read.
The DATA_AND_STATUS register contains of the following flags:
- DATA – shows the data read from an I2C read cycle, when transfer is completed i.e. the BUSY flag goes back to 0.
- CLOCK_STATUS – shows the status of the I2C clock. When the flag shows 1 the clock is disabled, when 0, the clock is enabled.
- BUSY – the flag stays 1 while a I2C transfer is in progress, goes to 0 when the transfer is done.
- ERROR - the flag is set/cleared after each I2C transfer, depending if there were any I2C errors during the transfer.

CLOCK_ DISABLE - writing a '1' in the filed disables the I2C clock; writing a '0' enables the clock (default is 0).
PRESCALLER register shows the I2C clock division ratio with respect to the CPU clock.

Watchdog

The watchdog's main purpose is to watch over the user program and make sure that it is executing properly. It will produce a signal which resets all digital circuitry including the CPU if it is not fed within the set timeout period. Feeding the watchdog is achieved by writing specific values to the "bone" registers. Typically, the user program writes one of the bones in the main routine and the other bone in the interrupt routine. If either routine is not running due to a stalled loop or the lack of CPU cycles, then a reset will be initiated which may allow the system to recover. Figure 49 shows HSA8000 watchdog system a block diagram.

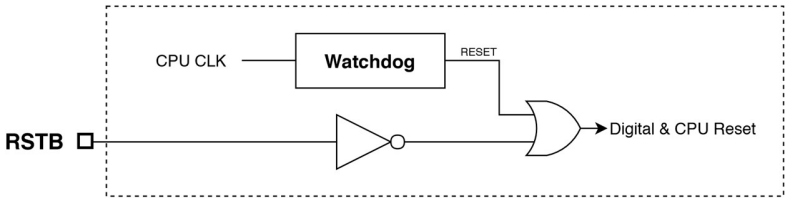


Figure 49- HSA8000 Watchdog system block diagram.

Watchdog Registers

The watchdog registers are defined in watchdog.h and visible in the Helios GUI application under the WATCHDOG structure.

PERIPHERAL->FIELD	Type	POR Value
WATCHDOG->ENABLE__U1	WO	OFF
WATCHDOG->POR_INDICATOR_CLEAR__U1	WO	OFF
WATCHDOG->WD_INDICATOR_CLEAR__U1	WO	OFF
WATCHDOG->PWRON_WATCHDOG_DISABLE__U1	WO	OFF
WATCHDOG->ENABLED__U1	RO	OFF
WATCHDOG->POR_INDICATOR__U1	RO	ON
WATCHDOG->WD_INDICATOR__U1	RO	OFF
WATCHDOG->PWRON_WATCHDOG_ENABLED__U1	RO	ON
WATCHDOG->BONE0_FAIL__U1	RO	OFF
WATCHDOG->BONE1_FAIL__U1	RO	OFF
WATCHDOG->TIMEOUT__U32	RW	0xFFFFFFFF
WATCHDOG->BONE0__U32	WO	0
WATCHDOG->BONE1__U32	WO	0
WATCHDOG->PRESERVED__U32	RW	0

Watchdog Functions and Features

Power-on Watchdog

The watchdog guards against improper booting by starting in a special mode. The user software must disable the power-on watchdog after booting by setting PWRON_WATCHDOG_DISABLE. If this is not set, then a watchdog reset will occur after 85 seconds.

Watchdog

The watchdog cannot be disabled once it is enabled. BONE0 must be written with 0x1234ABCD and BONE1 must be written with 0x5678CDEF within the timeout period to prevent a reset. The timeout value which counts CPU cycles is reloaded only after both bones have been written so changing this value does not immediately

change the actual timeout in progress. Whichever bone(s) have not been written since the counter was reloaded, and therefore caused the reset, will be indicated by the bone fail values.

Preserved Value

The PRESERVED register will not be cleared by a watchdog reset. It however will be cleared during a power-on reset. This register could be used as a counter register to track the number of watchdog resets which have occurred since the last power-on reset or store a specific value to be recalled when the software reboots. It would be up to the software to increment the value or write a specific value to this register to be preserved.

Power-on Reset Indicator

POR_INDICATOR is set during a power on reset and only cleared by setting the POR_INDICATOR_CLEAR. A subsequent watchdog reset will not clear this value. An external device which communicates with the software could clear this value. If it is later found to be set, then a power-on reset has occurred, and appropriate actions could be taken.

Watchdog Indicator

WD_INDICATOR will be set when the hardware reset was the result of a watchdog reset as opposed to a power-on reset. A subsequent power-on reset will clear this value. This can be cleared by software using WD_INDICATOR_CLEAR as a way of indicating that the watchdog reset has been serviced. Clearing this value also clears the bone fail flags.

Math Accelerators Block

The Math Accelerator Block includes three blocks that perform mathematical function necessary for calculating. The blocks functionality is explained below:

MATH_SQRT

ARGUMENT register

The number to be square rooted is written to the ARGUMENT register. By writing to it, a BUSY flag is set, and the SQRT state machine is started automatically. The operation takes up to 16 clock ticks at the Peripheral clock rate 100 MHz to complete the calculation. The BUSY flag is cleared automatically after the calculation is done. The operation duration depends on the operand effective size:

- up to 16 bits for 8 clock ticks
- 17 to 24 bits for 12 clock ticks
- 25 to 32 bits for 16 clock ticks.

The machine automatically detects the effective size of the operand and decides the cycles number necessary to be completed.

RESULT_TRUNC and RESULT_ROUND registers

After the operation is completed, the result truncated and rounded values are written in the RESULT_TRUNC and RESULT_ROUND registers.

MATH_DIVIDER

NUMERATOR and DENOMINATOR registers

The numerator and denominator are written in the NUMERATOR and DENOMINATOR registers respectively. Both numbers are 32 bits unsigned. The result is 64 bits, with 32 bits integer part and 32 bits fractional part. If the denominator is "0" then the result will be filled with all ones. By writing to any of the NUMERATOR OR DENOMINATOR registers, a BUSY flag is set, and the divider state machine is started automatically. Writing to any of these operand registers before the calculation is completed will abort the current calculation and restart a new one with the new data. The operation takes about 22 clock cycles at 100 MHz Peripheral clock rate. The BUSY flag is cleared automatically when the calculation is done.

TRUNC_INTEGER and TRUNC_FRACTIONAL register

The result of the integer and fractional parts of the division result is written in the TRUNC_INTEGER and TRUNC_FRACTIONAL correspondingly. The TRUNC_FRACTIONAL result in this register is truncated with respect to the last binary decimal.

MATH_SIN_COS

ARGUMENT register

The register represents the angle of the desired sin and cos, which is quantized on 10 bits. A value of "0" represents 0^o, a value of "512" represents 180^o, and the max value, "1023", represents 359.6484375^o. A value of 1024 would represent 360^o, but in this implementation, it will wrap down to 0. Writing to the ARGUMENT register triggers the state machine and the result is available in the very next AMBA cycle so a busy flag is not necessary.

RESULT_SIN and RESULT_COS registers

The registers show the results of sin and cos of the written in the ARGUMENT register.

The SIN (COS) register contains the truncated sin (cos) value of the argument. The result is in 16-bit two's complement format.

Example:

sin (256) produces 32767

sin (512+256) produces -32767

Timer

There are two 32-bit counters in the timer block. The main purpose is to generate real-time interrupts, but they can be used to generate flexible periodic signals too. The counters are clocked at the same speed as the CPU which is typically 50 MHz. Timer block diagram is shown in Figure 50.

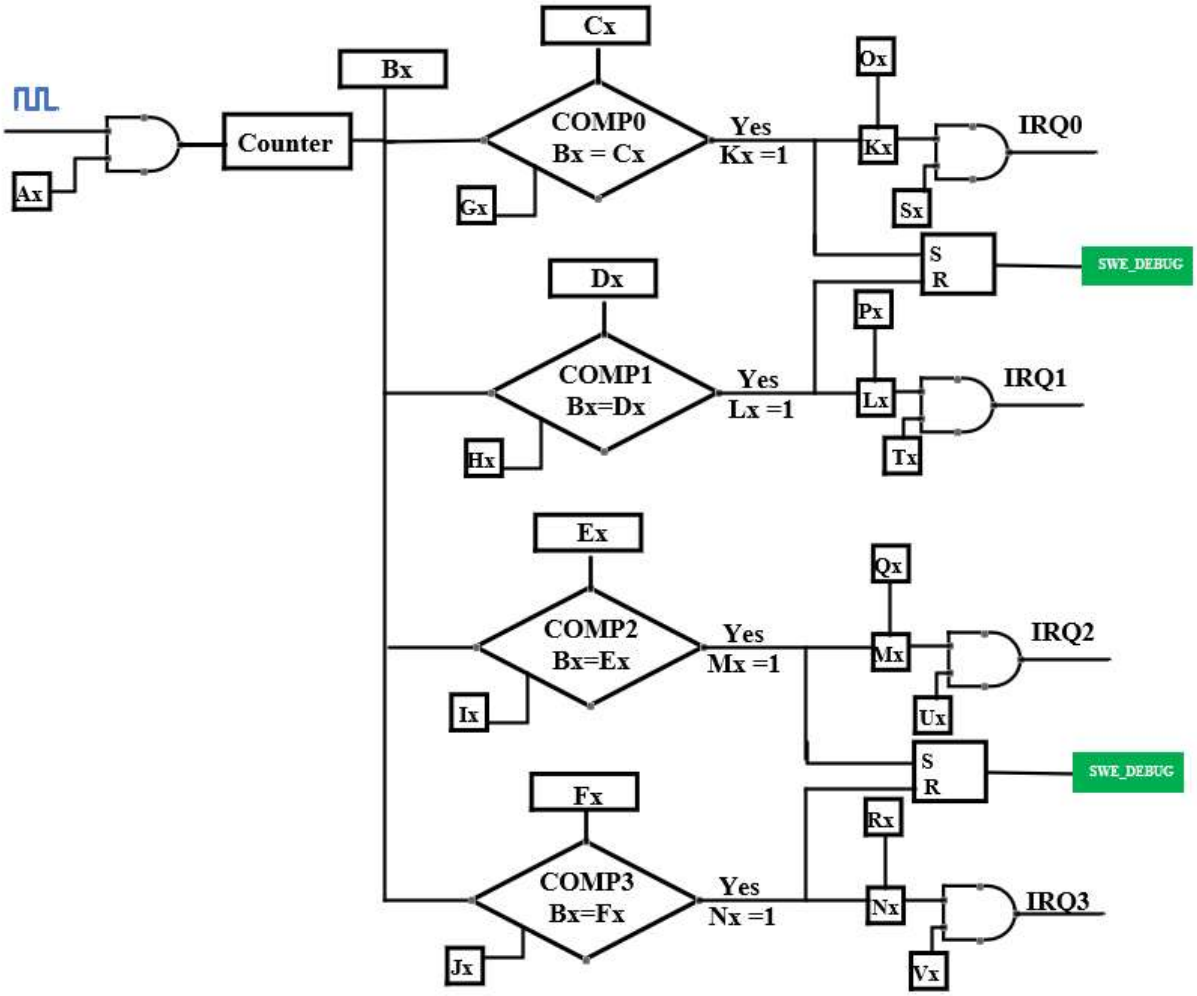


Figure 50 - Timer block diagram

Timer Registers

The timer block registers are shown in the table below:

	PERIPHERAL->FIELD	Type	POR Value
Ax	TIMER-> COUNTx_ENABLE__U1	WR	OFF
Bx	TIMER-> COUNTx__REG	RO	
Cx	TIMER_>COUNTx_COMP0__REG	WO	
Dx	TIMER_>COUNTx_COMP1__REG	WO	
Ex	TIMER_>COUNTx_COMP2__REG	WO	
Fx	TIMER_> COUNTx_COMP3_PERIOD__REG	WO	
Gx	TIMER-> COUNTx_COMP0_EQUAL_ENABLE__U1	WR	OFF
Hx	TIMER-> COUNTx_COMP1_EQUAL_ENABLE__U1	WR	OFF
Ix	TIMER-> COUNTx_COMP2_EQUAL_ENABLE__U1	WR	OFF
Jx	TIMER-> COUNTx_COMP3_EQUAL_ENABLE__U1	WR	OFF
Kx	TIMER->COUNTx_COMP0_EQUAL	RO	OFF
Lx	TIMER->COUNTx_COMP1_EQUAL	RO	OFF
Mx	TIMER->COUNTx_COMP2_EQUAL	RO	OFF
Nx	TIMER->COUNTx_COMP3_EQUAL	RO	OFF
Ox	TIMER->COUNTx_COMP0_EQUAL_CLEAR__U1	WO	OFF
Px	TIMER->COUNTx_COMP1_EQUAL_CLEAR__U1	WO	OFF
Qx	TIMER->COUNTx_COMP2_EQUAL_CLEAR__U1	WO	OFF
Rx	TIMER->COUNTx_COMP3_EQUAL_CLEAR__U1	WO	OFF
Sx	TIMER->COUNTx_COMP0_EQUAL_IRQ_ENABLE__U1	WR	ON
Tx	TIMER->COUNTx_COMP1_EQUAL_IRQ_ENABLE__U1	WR	OFF
Ux	TIMER->COUNTx_COMP2_EQUAL_IRQ_ENABLE__U1	WR	ON
Vx	TIMER->COUNTx_COMP3_EQUAL_IRQ_ENABLE__U1	WR	OFF

x = 0 and 1.

The counters are enabled by the corresponding Ax bit. Each counter goes from 0 up to the value set in Fx register minus 1 before reloading. The value of the counter can be seen in Bx register. There are another three registers (Cx, Dx, and Ex) providing counts for comparison. To enable comparison, corresponding bit of the COMP block (Gx, Hx, Ix and Jx) should be set to 1. When an equal event occurs, the corresponding bit (Kx, Lx, Mx, and Nx) is set to 1. This bit can be cleared only in the software by writing 0 to corresponding bit (Ox, Px, Qx and Rx). Every equal event can create an interrupt request when the corresponding bit is enabled (Sx, Tx, Ux, and Vx).

Timer Functions and Features

Compare values

Four compare values create an EQUAL signal when they match[1] the count. If the EQUAL_ENABLE is set, then the EQUAL flag will be set in the register for the software to read and clear.

COUNTx_COMP3_PERIOD matches when the value in COUNTx register is equal to COUNTx_COMP3_PERIOD – 1. after that, the counter is reset.

Interrupt Requests

If the IRQ_ENABLE is set, then the processor will be interrupted when the EQUAL flag is set provided that the interrupt mask has also been set. The CPU interrupt mask for the timer can be set with the following command:

```
esi_interrupt_set_mask(esi_interrupt_get_mask() | PERIPHERAL_INTERRUPT__INDEX__TIMER);
```

Debug Signals

For every counter, two debug signals are derived from the COMP_EQUAL flags:

- TIMER_COUNTx_0_1_EQUAL is set when COMP0 equals the count and clears when COMP1 equals the count.
- TIMER_COUNTx_2_3_EQUAL is set when COMP2 equals the count and gets cleared when the counter reaches its final count.

The actual hardware signal when observed is delayed by one clock period.

The EQUAL signals can be forced high or low with the following registers:

PERIPHERAL->FIELD	Type	POR Value
SOURCE_BUS -> COMPARE_TIMEx_EQUAL_FORCE_HIGH__U1	WO	OFF
SOURCE_BUS -> COMPARE_TIMEx_EQUAL_FORCE_LOW__U1	WO	OFF

where x = 0 -3.

Timer Example:

Let's enable the counters, and IRQ bits and set the following registers:

```
TIMER->COUNT0_COMP0__REG = 20;
TIMER->COUNT0_COMP1__REG = 30;
TIMER->COUNT0_COMP2__REG = 40;
TIMER-> COUNT0_COMP3_PERIOD__REG = 100;
```

Counter 0 will count from 0 to 99 and then will start over from 0.
There will be an event when the counter reaches 20, then 30, 40 and finally when it reaches 99.
The register TIMER->COUNTx_COMP0_EQUAL will be automatically set to 1, when the counter is equal to 20.
It can be cleared by setting 0 to TIMER->COUNTx_COMP0_EQUAL_CLEAR__U1.
For the example above, TIMER_COUNTx_0_1_EQUAL will be set high set when the counter reaches 20 and it is cleared when the counter reaches 30. TIMER_COUNTx_2_3_EQUAL will be set high when the counter reaches 40 and it is cleared when the counter reaches 99.
See section DEBUG to observe the signals out on the debug pins.

I/O Block

The I/O block controls the functionality of the digital input/output pins: The GPIO pins can be configured as:

- Push-pull outputs
- Open-drain outputs
- Inputs with optional pull-up or pull-down.

The HSA8000 has 32 GPIOs. In addition to the basic IO functions, to every GPIO can be assigned an alternate function, i.e. to be connected to a hardware peripheral (either an input or an output).

GPIO0 to GPIO7 are slightly different than GPIO8 to GPIO31, since they have faster dedicated paths to export the driver signals. GPIOx Block (x = 0 to 7) is shown in Figure 51. GPIO0 to GPIO7 can export the driver signals coming from the Driver Controller block when P_x and Q_x fields of the Pair Processor block (Figure 38) are set to 1. By default, the drivers are enabled (P_x and Q_x fields are set 1). The fast driver paths are illustrated in Figure 51.

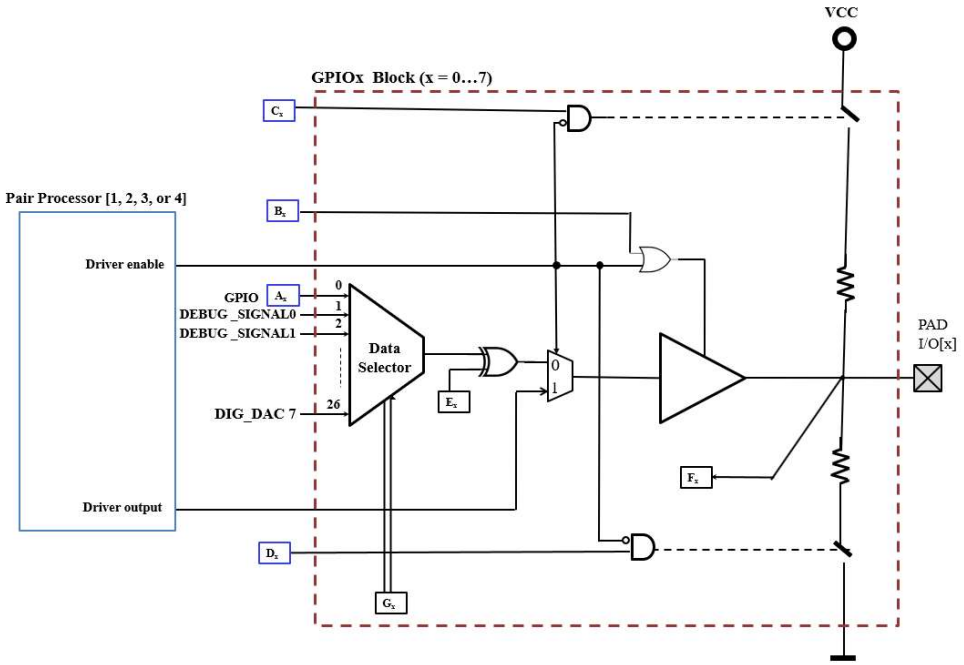


Figure 51 - The block diagram of GPIOx Block (x = 0…7)

GPIOx Block (x = 8 to 31) is shown in **Figure** 52. The blocks do not have special driver paths and therefore, the gate drivers cannot be exported on these GPIOs.

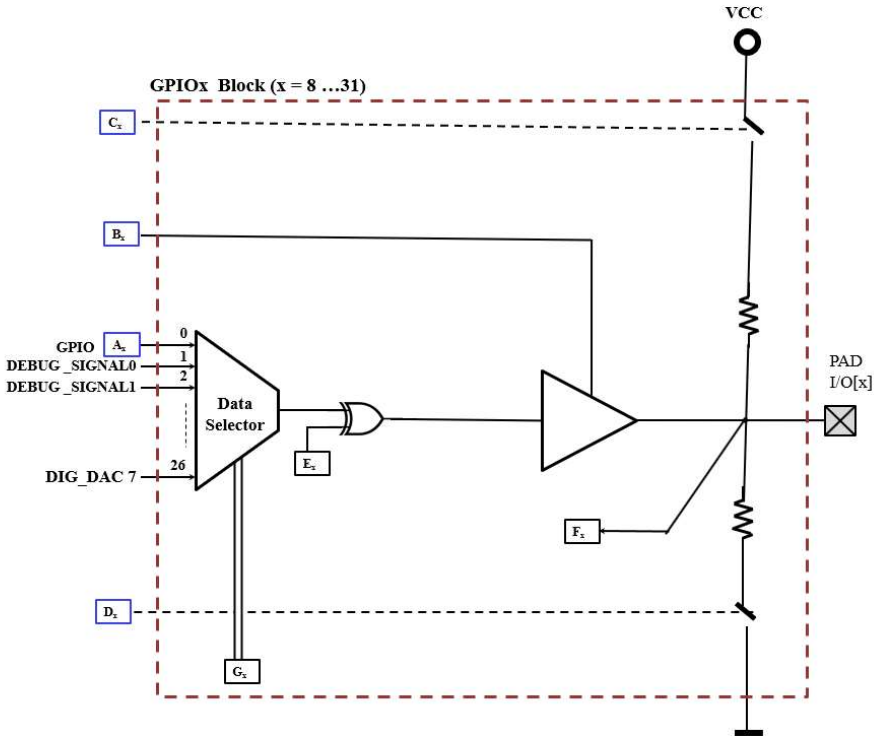


Figure 52 - The block diagram of GPIOx Block (x = 8 … 31)

	PERIPHERAL.FIELD_NAME	Default
A_x	GPIO->OUTPUTx__U1	0b
A_x	GPIO-> OUTPUTx_SET__U1	0b
A_x	GPIO->OUTPUTx_CLR__U1	0b
A_x	GPIO-> OUTPUTx_TOGGLE__U1	0b
B_x	GPIO-> DIRECTIONx__U1	0b
B_x	GPIO->DIRECTIONx_OUTPUT__U1	0b
B_x	GPIO->DIRECTIONx_INPUT__U1	0b
C_x	GPIO->PULL_UPx__U1	0b
D_x	GPIO->PULL_DOWNx__U1	0b
E_x	GPIO->OUTPUTx_INVERT__U1	0b
F_x	GPIO->INPUTx__U1	0b
G_x	GPIO->ALT_FUNCx__U5	xb
H_x	GPIO->RISING_EDGEx_IRQ_ENABLE__U1	0b
I_x	GPIO->FALLING_EDGEx_IRQ_ENABLE__U1	0b
J_x	GPIO->RISING_EDGEx_IRQ__U1	0b
K_x	GPIO->RISING_EDGEx_IRQ_CLEAR__U1	0b
L_x	GPIO->FALLING_EDGEx_IRQ__U1	0b
L_x	GPIO->FALLING_EDGEx_IRQ_CLEAR__U1	0b
M_x	GPIO->JTAG_ENABLE__U1	0b

Every GPIO block has an Alternate Function selector. The selector has 27 inputs. The signals from different hardware blocks are connected to the selectors' inputs. The following table shows the alternate function names.

Selector Number	Alternative Function Name	Selector Number	Alternative Function Name	Selector Number	Alternative Function Name
0	GPIO	9	UART TXD	18	UART HDLC RX1
1	DEBUG_SIGNAL0	10	UART RXD	19	DIG DAC 0
2	DEBUG_SIGNAL1	11	SPI SEN	20	DIG DAC 1
3	DEBUG_SIGNAL2	12	SPI SCK	21	DIG DAC 2
4	DEBUG_SIGNAL3	13	SPI SDO	22	DIG DAC 3
5	DEBUG_SIFNAL4	14	SPI SDI	23	DIG DAC 4
6	DEBUG_SIGNAL5	15	I2C SDA	24	DIG DAC 5
7	DEBUG_SIGNAL6	16	I2C SCK	25	DIG DAC 6
8	DEBUG_SIGNAL7	17	UART HDLC TX1	26	DIG DAC 7

When ALT_FUNCx is set to 0 (i.e. plain GPIO), then the direction (input/output) and the pull-up or pull-down controls of the selected pin can be programmed through the associated registers from the GPIO block. When another alternative function is selected, the controls (I/O, pull-up/down) are determined by the specific alternate function which is selected and cannot be programmed manually.

ALT_FUNCx register (x = 0 to 31) is associated with each GPIO.
OUTPUT register holds the programmed state of the GPIO pins configured as outputs. For instance, writing the value 0x00000005, sets GPIOs 0 and 2, and clears the rest (except the GPIOs which are assigned an Alternate Function). Reading back the register returns the programmed state of the GPIOs, not the actual state of the GPIO pin. For example, if GPIO2 was shorted to ground in the hardware, reading the OUTPUT register will return the value "1" for GPIO2 (the programmed value) and not "0" which is the true state of the GPIO2 in the hardware.
OUTPUT_SETS register sets the GPIOs. For example, writing the value 0x00000010 sets GPIO4 and leaves all the other GPIOs unchanged.
OUTPUT_CLEARS register clears GPIOs. For example, writing the value 0x00000014, clears GPIO4 and GPIO2, and leaves all the other GPIOs unchanged.
OUTPUT_TOGGLES toggle the GPIOs with ones and leaves the others unchanged.
OUTPUT_INVERTS register inverts the signal coming from the Data selector.
To Output a signal to the pin the following registers are used:
DIRECTIONS register enables the outputs. When the bit is set to "1" the GPIO is configured as an output, "0" the GPIO is configured as an input. For example, writing the value 0x000000F0 into the register configure GPIOs 4…7 as outputs and the rest as inputs.
DIRECTION_OUTPUTS register set GPIOs enable. Writing, for example, 0x00000001 into it configures GPIO0 as an output and leaves the others with their previous configuration (direction).
DIRECTION_INPUTS register clears the GPIOs direction. Writing 0x00000002 into the register configures GPIO1 as an input and leaves all other GPIOs with their previous configuration.
For GPIOs configured as inputs, optional pull-up or pull-down internal resistors can be connected if desired, in accordance with the following two registers: PULL_UP and PULL_DOWN.

INPUTS register reads the actual state of the GPIOs. The main purpose of this register is to read the state of the GPIOs configured as inputs. However, it can be used to read the actual state of the outputs as well and check, if desired, if the programmed state of the output matches the actual hardware value from the board.

Digital DAC

HSA8000 has 8 digital DACs. The digital DACs can function in either PWM or Sigma-Delta modes. A register controls the DACs clock frequency. When the DAC is set in PWM mode, a register can set the duty cycle. The maximum value is when the register is set to 0x400 which responds to 100%. Setting the register to 0x200, the duty cycle is 50% and setting to 0x00 the duty cycle is 0%. Every digital DAC can be exported on a GPIOx pin (x = 0-31) by setting its ALT_FUNCx.
In the table below are the field names for the digital DACs (x = 0 to 7).

PERIPHERAL.FIELD_NAME	Default
DIGITAL_DAC->DACx_ENABLE__U1	0b
DIGITAL_DAC->DACx_MODE__U2[1]	0b
DIGITAL_DAC->DACx_PRESCALLER__U1[2]	0b
DIGITAL_DAC->DACx_DUTY__U10[3]	0b
GPIO->ALT_FUNCx__U5	xb

[1]Mode: 0b = SIGMA DELTA, 1b = PWM.
[2]Setting the DAC clock frequency: 00= Sets to 100 MHz; 01= Divide by 4 (25 MHz); 10 = Divide by 8 (12.5 MHz); 11 = Divided by 16 (6.25 MHz)
[3]Setting the DAC duty cycle: 0 to 1024. Example: 0 = 0%; 256 = 0x100 = 25%; 526 = 0x200 = 50%, 768=0x300=75%; 1024 =x0x400 =100%.

In table below is shown the numbers that should be selected in the ALT_FUNCx for the corresponding digital DAC to be exported.

Selector Number	Alternative Function Name
19	DIG DAC 0
20	DIG DAC 1
21	DIG DAC 2
22	DIG DAC 3
23	DIG DAC 4
24	DIG DAC 5
25	DIG DAC 6
26	DIG DAC 7

DEBUG

HSA8000 is designed to export signals in real-time. Figure 53 shows three selectors used for signals debugging.

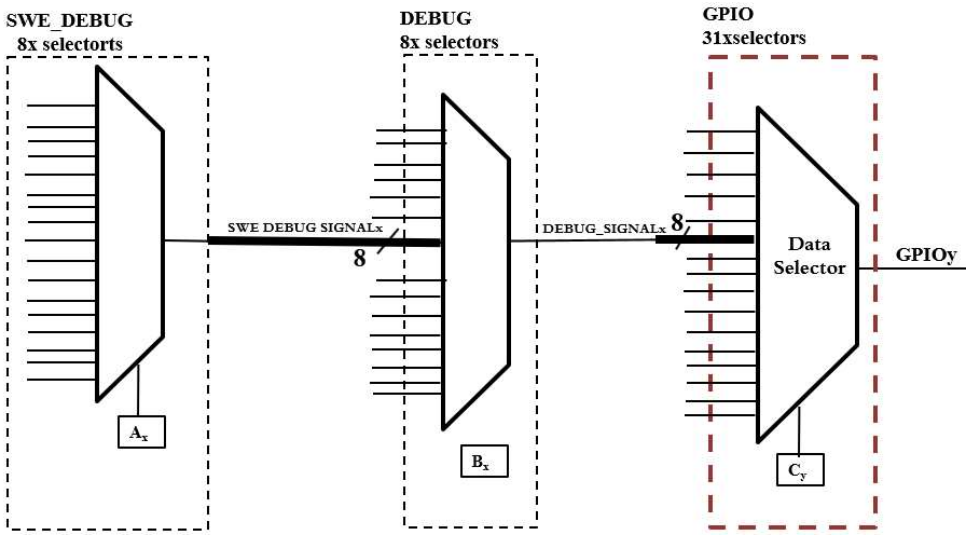


Figure 53 HSA8000 Debug block

	PERIPHERAL.FIELD_NAME	Default
A_x	SWE_DEBUG->DEBUG_SIGNALx__U8	0b
B_x	DEBUG->SIGNALx__U8	0b
C_y	GPIO->ALT_FUNCy__U5	0b

In the table above x = 0 to 7 and y =0 to 31.

The SWE_DEBUG block has eight selectors A_x that select signals from the Switching Engine Block.

0 = CMP0_POS_COMP
1 = CMP0_NEG_COMP
2 = CMP1_POS_COMP
3 = CMP1_NEG_COMP
4 = CMP2_POS_COMP
5 = CMP2_NEG_COMP
6 = CMP3_POS_COMP
7 = CMP3_NEG_COMP
16 = CMP4_POS_COMP
17 = CMP4_NEG_COMP
18 = CMP5_POS_COMP
19 = CMP5_NEG_COMP
20 =
21 =
22 = CMP7_POS_COMP
23 = CMP7_NEG_COMP
24 = ADC0_AN0_COMP_STATUS
25 = ADC0_AN1_COMP_STATUS
26 = ADC0_AN2_DCOMP_STATUS
27 = ADC0_AN3_COMP_STATUS
28 = ADC1_AN12_FLT_COMP_STATUS

29 = ADC1_AN13_FLT_COMP_STATUS
30 = ADC1_AN12_BUF_COMP_STATUS
31 = ADC1_AN13_BUF_COMP_STATUS
32 = ADC2_CHANNEL0_COMP_STATUS
33 = ADC2_CHANNEL1_COMP_STATUS
34 = ADC2_CHANNEL2_COMP_STATUS
35 = ADC2_CHANNEL3_COMP_STATUS
36 = ADC2_CHANNEL4_COMP_STATUS
37 = ADC2_CHANNEL5_COMP_STATUS
38 = ADC2_CHANNEL6_COMP_STATUS
39 = ADC2_CHANNEL7_COMP_STATUS
40 = COMPARE_TIME0_EQUAL
41 = COMPARE_TIME1_EQUAL
42 = COMPARE_TIME2_EQUAL
43 = COMPARE_TIME3_EQUAL
44 = SWE_EDT_INTERLEAVE_PHASE1
45 = SWE_EDT_INTERLEAVE_PHASE2
46 = SWE_EDT_INTERLEAVE_PHASE3
64 = SWE_DRIVER0_LS
65 = SWE_DRIVER0_HS
66 = SWE_DRIVER1_LS
67 = SWE_DRIVER1_HS
68 = SWE_DRIVER2_LS
69 = SWE_DRIVER2_HS
70 = SWE_DRIVER3_LS
71 = SWE_DRIVER3_HS
72 = SWE_EDT0_HS
73 = SWE_EDT0_LS
74 = SWE_EDT1_HS
75 = SWE_EDT1_LS
76 = SWE_EDT2_HS
77 = SWE_EDT2_LS
78 = SWE_EDT3_HS
79 = SWE_EDT3_LS
80 = SWE_PWM0_LS
81 = SWE_PWM0_HS
82 = SWE_PWM1_LS
83 = SWE_PWM1_HS
84 = SWE_PWM2_LS
85 = SWE_PWM2_HS
86 = SWE_PWM3_LS
87 = SWE_PWM3_HS
88 = SWE_PWM0_REFERENCE
89 = SWE_PWM1_REFERENCE
90 = SWE_PWM2_REFERENCE
91 = SWE_PWM0_REFERENCE
92 = SWE_FAULT0
93 = SWE_FAULT1
94 = SWE_FAULT2
95 = SWE_FAULT3
96 = SWE_FAULT4
97 = SWE_FAULT5
98 = SWE_FAULT6
99 = SWE_FAULT7
104 = SWE_FAULT0_SELECTED_SOURCE_RAW
105 = SWE_FAULT1_SELECTED_SOURCE_RAW
106 = SWE_FAULT2_SELECTED_SOURCE_RAW
107 = SWE_FAULT3_SELECTED_SOURCE_RAW
108 = SWE_FAULT4_SELECTED_SOURCE_RAW
109 = SWE_FAULT5_SELECTED_SOURCE_RAW
110 = SWE_FAULT6_SELECTED_SOURCE_RAW
111 = SWE_FAULT7_SELECTED_SOURCE_RAW

112 = SWE_FAULT0_SELECTED_SOURCE_FLT
113 = SWE_FAULT1_SELECTED_SOURCE_FLT
114 = SWE_FAULT2_SELECTED_SOURCE_FLT
115 = SWE_FAULT3_SELECTED_SOURCE_FLT
116 = SWE_FAULT4_SELECTED_SOURCE_FLT
117 = SWE_FAULT5_SELECTED_SOURCE_FLT
118 = SWE_FAULT6_SELECTED_SOURCE_FLT
119 = SWE_FAULT7_SELECTED_SOURCE_FLT
128 = SWE_WINDOW0
129 = SWE_WINDOW1
130 = SWE_WINDOW2
131 = SWE_WINDOW3
132 = SWE_WINDOW4
133 = SWE_WINDOW5
134 = SWE_WINDOW6
135 = SWE_WINDOW7
136 = SWE_WINDOW8
137 = SWE_WINDOW9
138 = SWE_WINDOW10
139 = SWE_WINDOW11
140 = SWE_WINDOW12
141 = SWE_WINDOW13
142 = SWE_WINDOW14
143 = SWE_WINDOW15
144 = SWE_EVENT0
145 = SWE_EVENT1
146 = SWE_EVENT2
147 = SWE_EVENT3
148 = SWE_EVENT4
149 = SWE_EVENT5
150 = SWE_EVENT6
151 = SWE_EVENT7
152 = SWE_EVENT8
153 = SWE_EVENT9
154 = SWE_EVENT10
155 = SWE_EVENT11
156 = SWE_EVENT12
157 = SWE_EVENT13
158 = SWE_EVENT14
159 = SWE_EVENT15
160 = SWE_EVENT0_INPUT
161 = SWE_EVENT1_INPUT
162 = SWE_EVENT2_INPUT
163 = SWE_EVENT3_INPUT
164 = SWE_EVENT4_INPUT
165 = SWE_EVENT5_INPUT
166 = SWE_EVENT6_INPUT
167 = SWE_EVENT7_INPUT
168 = SWE_EVENT8_INPUT
169 = SWE_EVENT9_INPUT
170 = SWE_EVENT10_INPUT
171 = SWE_EVENT11_INPUT
172 = SWE_EVENT12_INPUT
173 = SWE_EVENT13_INPUT
174 = SWE_EVENT14_INPUT
175 = SWE_EVENT15_INPUT
176 = SWE_EVENT0_FILTERED_INPUT
177 = SWE_EVENT1_FILTERED_INPUT
178 = SWE_EVENT2_FILTERED_INPUT
179 = SWE_EVENT3_FILTERED_INPUT
180 = SWE_EVENT4_FILTERED_INPUT
181 = SWE_EVENT5_FILTERED_INPUT

182 = SWE_EVENT6_FILTERED_INPUT
183 = SWE_EVENT7_FILTERED_INPUT
184 = SWE_EVENT8_FILTERED_INPUT
185 = SWE_EVENT9_FILTERED_INPUT

The DEBUG selector can select the following signals:

0 = LOW
1 = HIGH
2 = 50 MHz
3 = RX0
4 = 100 MHz
5 = ADC0_CLK
6 = ADC1_CLK
7 = ADC2_CLK
8 = ADC0_BUSY
9 = ADC1_BUSY
10 = ADC2_BUSY
11 = ADC2_SUMPLE_HOLD
12 = TIMER_COUNTER0_0_1_EQUAL
13 = TIMER_COUNTER0_2_3_EQUAL
14 = TIMER_COUNTER1_0_1_EQUAL
15 = TIMER_COUNTER1_2_3_EQUAL
16 = SWE_DBG_DEBUG_SIGNAL0
17 = SWE_DBG_DEBUG_SIGNAL1
18 = SWE_DBG_DEBUG_SIGNAL2
19 = SWE_DBG_DEBUG_SIGNAL3
20 = SWE_DBG_DEBUG_SIGNAL4
21 = SWE_DBG_DEBUG_SIGNAL5
22 = SWE_DBG_DEBUG_SIGNAL6
23 = SWE_DBG_DEBUG_SIGNAL7
25 = RX1
26 = TX1
42 = AC_PLL_sync_grid_comp
43 = AC_PLL_comp_filtered
44 = AC_PLL_out_of_range
45 = AC_PLL_compare_0_pulse
46 = AC_PLL_compare_1_pulse
47 = AC_PLL_phdet_re_up_pulse
48 = AC_PLL_phdet_re_dn_pulse
49 = AC_PLL_phdet_fe_up_pulse
50 = AC_PLL_phdet_fe_dn_pulse
51 = AC_PLL_reconstruct_not_locked
52 = AC_PLL_reconstruct_positive
53 = AC_PLL_reconstruct_be
54 = AC_PLL_fmul_pulse
55 = AC_PLL_sample_pulse
56 = AC_PLL_sample_N

Documentation Support

DPD1126-4_HSA8000_IXC2_HW_and_Register_Users_Guide

DPD1144-2_HSA8000_IXC2_Register_Map