EZR32WG

ACMP - Analog Comparator

ACMP - Analog Comparator

Analog comparator (ACMP) Peripheral API.

The Analog Comparator is used to compare voltage of two analog inputs with a digital output indicating which input voltage is higher. Inputs can either be one of the selectable internal references or from external pins. Response time and current consumption can be configured by altering the current supply to the comparator.

ACMP is available down to EM3 and is able to wake up the system when input signals pass a certain threshold. Use ACMP_IntEnable() to enable an edge interrupt to use this functionality.

This example shows how to use the em_acmp.h API for comparing an input pin to an internal 2.5 V reference voltage.

```
/* The ACMP is a high-frequency peripheral so we need to enable
 * both the HFPER clock and the ACMP0 clock. */
CMU_ClockEnable(cmuClock_HFPER, true);
CMU_ClockEnable(cmuClock_ACMP0, true);

/* Initialize with default settings. */
ACMP_Init_TypeDef init = ACMP_INIT_DEFAULT;
init.enable = false;
ACMP_Init(ACMP0, &init);

/* Setup the two channels to compare. The first argument is the negative
 * channel and the second argument is the positive channel.
 *
 * Here we compare channel 0 with the 2.5 V internal reference. When
 * channel 0 is lower than 2.5 V then the ACMP output is 0 and when
 * channel 0 voltage is higher than 2.5 V then the ACMP output is 1. */
ACMP_ChannelSet(ACMP0, acmpChannel2V5, acmpChannel0);

/* To be able to probe the output we can send the ACMP output to a pin.
 * The second argument to this function is the pin location which is
 * device dependent. */
ACMP_GPIOSetup(ACMP0, 0, true, false);

/* Finally we enable the ACMP. */
ACMP_Enable(ACMP0);
```

Note

- ACMP can also be used to compare two separate input pins.

ACMP also contains specialized hardware for capacitive sensing. This module contains the ACMP_CapsenseInit() function to initialize ACMP for capacitive sensing and the ACMP_CapsenseChannelSet() function to select the current capsense channel.

For applications that require capacitive sensing it is recommended to use a library, such as cslib, which is provided by Silicon Labs.

Modules

ACMP_CapsenseInit_TypeDef

ACMP_Init_TypeDef

Enumerations

enum ACMP_CapsenseResistor_TypeDef {

acmpResistor0 = _ACMP_INPUTSEL_CSRESSEL_RES0
acmpResistor1 = _ACMP_INPUTSEL_CSRESSEL_RES1
acmpResistor2 = _ACMP_INPUTSEL_CSRESSEL_RES2
acmpResistor3 = _ACMP_INPUTSEL_CSRESSEL_RES3

}
Resistor values used for the internal capacitive sense resistor.

enum ACMP_HysteresisLevel_TypeDef {

acmpHysteresisLevel0 = _ACMP_CTRL_HYSTSEL_HYST0
acmpHysteresisLevel1 = _ACMP_CTRL_HYSTSEL_HYST1
acmpHysteresisLevel2 = _ACMP_CTRL_HYSTSEL_HYST2
acmpHysteresisLevel3 = _ACMP_CTRL_HYSTSEL_HYST3
acmpHysteresisLevel4 = _ACMP_CTRL_HYSTSEL_HYST4
acmpHysteresisLevel5 = _ACMP_CTRL_HYSTSEL_HYST5
acmpHysteresisLevel6 = _ACMP_CTRL_HYSTSEL_HYST6
acmpHysteresisLevel7 = _ACMP_CTRL_HYSTSEL_HYST7

}
Hysteresis level.

enum ACMP_WarmTime_TypeDef {

acmpWarmTime4 = _ACMP_CTRL_WARMTIME_4CYCLES
acmpWarmTime8 = _ACMP_CTRL_WARMTIME_8CYCLES
acmpWarmTime16 = _ACMP_CTRL_WARMTIME_16CYCLES
acmpWarmTime32 = _ACMP_CTRL_WARMTIME_32CYCLES
acmpWarmTime64 = _ACMP_CTRL_WARMTIME_64CYCLES
acmpWarmTime128 = _ACMP_CTRL_WARMTIME_128CYCLES
acmpWarmTime256 = _ACMP_CTRL_WARMTIME_256CYCLES
acmpWarmTime512 = _ACMP_CTRL_WARMTIME_512CYCLES

}
ACMP warmup time.

enum ACMP_Channel_TypeDef {

acmpChannel0 = _ACMP_INPUTSEL_NEGSEL_CH0
acmpChannel1 = _ACMP_INPUTSEL_NEGSEL_CH1
acmpChannel2 = _ACMP_INPUTSEL_NEGSEL_CH2
acmpChannel3 = _ACMP_INPUTSEL_NEGSEL_CH3
acmpChannel4 = _ACMP_INPUTSEL_NEGSEL_CH4
acmpChannel5 = _ACMP_INPUTSEL_NEGSEL_CH5
acmpChannel6 = _ACMP_INPUTSEL_NEGSEL_CH6
acmpChannel7 = _ACMP_INPUTSEL_NEGSEL_CH7
acmpChannel1V25 = _ACMP_INPUTSEL_NEGSEL_1V25
acmpChannel2V5 = _ACMP_INPUTSEL_NEGSEL_2V5
acmpChannelVDD = _ACMP_INPUTSEL_NEGSEL_VDD
acmpChannelDAC0Ch0 = _ACMP_INPUTSEL_NEGSEL_DAC0CH0
acmpChannelDAC0Ch1 = _ACMP_INPUTSEL_NEGSEL_DAC0CH1
acmpChannelCapSense = _ACMP_INPUTSEL_NEGSEL_CAPSENSE

}
ACMP inputs.

Functions

void ACMP_CapsenseInit(ACMP_TypeDef *acmp, const ACMP_CapsenseInit_TypeDef *init)
Set up ACMP for use in capacitive sense applications.

void ACMP_CapsenseChannelSet(ACMP_TypeDef *acmp, ACMP_Channel_TypeDef channel)
Set the ACMP channel used for capacitive sensing.

void ACMP_ChannelSet(ACMP_TypeDef *acmp, ACMP_Channel_TypeDef negSel, ACMP_Channel_TypeDef posSel)
Set which channels should be used in ACMP comparisons.

void ACMP_Disable(ACMP_TypeDef *acmp)
Disable ACMP.

void ACMP_Enable(ACMP_TypeDef *acmp)
Enable ACMP.

void ACMP_GPIOSetup(ACMP_TypeDef *acmp, uint32_t location, bool enable, bool invert)
Set up GPIO output from ACMP.

void ACMP_Init(ACMP_TypeDef *acmp, const ACMP_Init_TypeDef *init)
Initialize ACMP.

void ACMP_Reset(ACMP_TypeDef *acmp)
Reset ACMP to the same state that it was in after a hardware reset.

void ACMP_IntClear(ACMP_TypeDef *acmp, uint32_t flags)
Clear one or more pending ACMP interrupts.

void ACMP_IntDisable(ACMP_TypeDef *acmp, uint32_t flags)
Disable one or more ACMP interrupts.

void ACMP_IntEnable(ACMP_TypeDef *acmp, uint32_t flags)
Enable one or more ACMP interrupts.

uint32_t ACMP_IntGet(ACMP_TypeDef *acmp)
Get pending ACMP interrupt flags.

uint32_t ACMP_IntGetEnabled(ACMP_TypeDef *acmp)
Get enabled and pending ACMP interrupt flags.

void ACMP_IntSet(ACMP_TypeDef *acmp, uint32_t flags)
Set one or more pending ACMP interrupts from software.

Macros

#define ACMP_CAPSENSE_INIT_DEFAULT undefined
A default configuration for capacitive sense mode initialization.

#define ACMP_INIT_DEFAULT undefined
Default configuration for ACMP regular initialization.

Enumeration Documentation

ACMP_CapsenseResistor_TypeDef

> ACMP_CapsenseResistor_TypeDef

Resistor values used for the internal capacitive sense resistor.

See data sheet for your device for details on each resistor value.

Enumerator	
acmpResistor0	Resistor value 0.
acmpResistor1	Resistor value 1.
acmpResistor2	Resistor value 2.
acmpResistor3	Resistor value 3.

ACMP_HysteresisLevel_TypeDef

> ACMP_HysteresisLevel_TypeDef

Hysteresis level.

See data sheet for your device for details on each level.

Enumerator	
acmpHysteresisLevel0	Hysteresis level 0.
acmpHysteresisLevel1	Hysteresis level 1.
acmpHysteresisLevel2	Hysteresis level 2.
acmpHysteresisLevel3	Hysteresis level 3.
acmpHysteresisLevel4	Hysteresis level 4.
acmpHysteresisLevel5	Hysteresis level 5.
acmpHysteresisLevel6	Hysteresis level 6.
acmpHysteresisLevel7	Hysteresis level 7.

ACMP_WarmTime_TypeDef

> ACMP_WarmTime_TypeDef

ACMP warmup time.

The delay is measured in HFPERCLK / HFPERCCLK cycles and should be at least 10 us.

Enumerator	
acmpWarmTime4	4 cycles warmup
acmpWarmTime8	8 cycles warmup
acmpWarmTime16	16 cycles warmup
acmpWarmTime32	32 cycles warmup
acmpWarmTime64	64 cycles warmup
acmpWarmTime128	128 cycles warmup
acmpWarmTime256	256 cycles warmup
acmpWarmTime512	512 cycles warmup

ACMP_Channel_TypeDef

> ACMP_Channel_TypeDef

ACMP inputs.

Note that scaled VDD and bandgap references can only be used as negative inputs.

Enumerator	
acmpChannel0	Channel 0.
acmpChannel1	Channel 1.
acmpChannel2	Channel 2.
acmpChannel3	Channel 3.
acmpChannel4	Channel 4.
acmpChannel5	Channel 5.
acmpChannel6	Channel 6.
acmpChannel7	Channel 7.
acmpChannel1V25	1.25 V internal reference

acmpChannel2V5	2.5 V internal reference
acmpChannelVDD	Scaled VDD reference.
acmpChannelDAC0Ch0	DAC0 channel 0.
acmpChannelDAC0Ch1	DAC0 channel 1.
acmpChannelCapSense	Capacitive sense mode.

Function Documentation

ACMP_CapsenseInit

```
void ACMP_CapsenseInit (ACMP_TypeDef * acmp, const ACMP_CapsenseInit_TypeDef * init)
```

Set up ACMP for use in capacitive sense applications.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
const ACMP_CapsenseInit_TypeDef *	[in]	init	A pointer to the initialization structure used to configure ACMP for capacitive sensing operation.

This function sets up ACMP for use in capacitive sense applications. To use the capacitive sense functionality in the ACMP, use the PRS output of the ACMP module to count the number of oscillations in the capacitive sense circuit (possibly using a TIMER).

Note

- A basic example of capacitive sensing can be found in the STK BSP (capsense demo).

ACMP_CapsenseChannelSet

```
void ACMP_CapsenseChannelSet (ACMP_TypeDef * acmp, ACMP_Channel_TypeDef channel)
```

Set the ACMP channel used for capacitive sensing.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
ACMP_Channel_TypeDef	[in]	channel	The ACMP channel to use for capacitive sensing (Possel).

Note

- A basic example of capacitive sensing can be found in the STK BSP (capsense demo).

ACMP_ChannelSet

```
void ACMP_ChannelSet (ACMP_TypeDef * acmp, ACMP_Channel_TypeDef negSel, ACMP_Channel_TypeDef posSel)
```

Set which channels should be used in ACMP comparisons.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
ACMP_Channel_TypeDef	N/A	negSel	A channel to use on the negative input to the ACMP.
ACMP_Channel_TypeDef	N/A	posSel	A channel to use on the positive input to the ACMP.

ACMP_Disable

```
void ACMP_Disable (ACMP_TypeDef * acmp)
```

Disable ACMP.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.

ACMP_Enable

```
void ACMP_Enable (ACMP_TypeDef * acmp)
```

Enable ACMP.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.

ACMP_GPIOSetup

```
void ACMP_GPIOSetup (ACMP_TypeDef * acmp, uint32_t location, bool enable, bool invert)
```

Set up GPIO output from ACMP.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
uint32_t	N/A	location	The pin location to use. See the data sheet for location to pin mappings.
bool	N/A	enable	Enable or disable pin output.
bool	N/A	invert	Invert output.

Note

- GPIO must be enabled in the CMU before this function call, i.e.,

```
CMU_ClockEnable(cmuClock_GPIO, true);
```

ACMP_Init

> void ACMP_Init (ACMP_TypeDef * acmp, const ACMP_Init_TypeDef * init)

Initialize ACMP.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
const ACMP_Init_TypeDef *	[in]	init	A pointer to the initialization structure used to configure ACMP.

ACMP_Reset

> void ACMP_Reset (ACMP_TypeDef * acmp)

Reset ACMP to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.

Note

- The GPIO ACMP ROUTE register is NOT reset by this function to allow for centralized setup of this feature.
- The peripheral may be enabled and disabled during reset.

ACMP_IntClear

> void ACMP_IntClear (ACMP_TypeDef * acmp, uint32_t flags)

Clear one or more pending ACMP interrupts.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
uint32_t	[in]	flags	Pending ACMP interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the ACMP module. The flags can be, for instance, ACMP_IFC_EDGE or ACMP_IFC_WARMUP.

ACMP_IntDisable

> void ACMP_IntDisable (ACMP_TypeDef * acmp, uint32_t flags)

Disable one or more ACMP interrupts.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
uint32_t	[in]	flags	ACMP interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the ACMP module. The flags can be, for instance, ACMP_IEN_EDGE or ACMP_IEN_WARMUP.

ACMP_IntEnable

```
void ACMP_IntEnable (ACMP_TypeDef * acmp, uint32_t flags)
```

Enable one or more ACMP interrupts.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
uint32_t	[in]	flags	ACMP interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the ACMP module. The flags can be, for instance, ACMP_IEN_EDGE or ACMP_IEN_WARMUP.

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. Consider using ACMP_IntClear() prior to enabling if a pending interrupt should be ignored.

ACMP_IntGet

```
uint32_t ACMP_IntGet (ACMP_TypeDef * acmp)
```

Get pending ACMP interrupt flags.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.

Note

- This function does not clear event bits.

Returns

- Pending ACMP interrupt sources. A bitwise logic OR combination of valid interrupt flags for the ACMP module. The pending interrupt sources can be, for instance, ACMP_IF_EDGE or ACMP_IF_WARMUP.

ACMP_IntGetEnabled

```
uint32_t ACMP_IntGetEnabled (ACMP_TypeDef * acmp)
```

Get enabled and pending ACMP interrupt flags.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- This function does not clear interrupt flags.

Returns

- Pending and enabled ACMP interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in ACMPx_IEN_nnn register (ACMPx_IEN_nnn) and
  - the OR combination of valid interrupt flags of the ACMP module (ACMPx_IF_nnn).

ACMP_IntSet

```
void ACMP_IntSet (ACMP_TypeDef * acmp, uint32_t flags)
```

Set one or more pending ACMP interrupts from software.

Parameters

Type	Direction	Argument Name	Description
ACMP_TypeDef *	[in]	acmp	A pointer to the ACMP peripheral register block.
uint32_t	[in]	flags	ACMP interrupt sources to set as pending. Use a bitwise logic OR combination of valid interrupt flags for the ACMP module. The flags can be, for instance, ACMP_IFS_EDGE or ACMP_IFS_WARMUP.

Macro Definition Documentation

ACMP_CAPSENSE_INIT_DEFAULT

```
#define ACMP_CAPSENSE_INIT_DEFAULT
```

Value:

```
{                                                   \
  false,            /* fullBias */                  \
  false,            /* halfBias */                  \
  0x7,              /* biasProg */                  \
  acmpWarmTime512,  /* 512 cycle warmup to be safe */ \
  acmpHysteresisLevel5,                             \
  acmpResistor3,                                    \
  false,            /* low power reference */       \
  0x3D,             /* VDD level */                 \
  true              /* Enable after init. */        \
}
```

A default configuration for capacitive sense mode initialization.

ACMP_INIT_DEFAULT

```
#define ACMP_INIT_DEFAULT
```

Value:

```
{                                                   \
```

```
    false,            /* fullBias */                                    \
    false,            /* halfBias */                                    \
    0x7,              /* biasProg */                                    \
    false,            /* No interrupt on falling edge. */               \
    false,            /* No interrupt on rising edge. */                \
    acmpWarmTime512,  /* 512 cycle warmup to be safe */                 \
    acmpHysteresisLevel5,                                               \
    false,            /* Disabled emitting inactive value during warmup. */ \
    false,            /* low power reference */                         \
    0x3D,             /* VDD level */                                   \
    true              /* Enable after init. */                          \
  }
```

Default configuration for ACMP regular initialization.

ACMP_CapsenseInit_TypeDef

Capsense initialization structure.

Public Attributes

bool	fullBias Full-bias current.
bool	halfBias Half-bias current.
uint32_t	biasProg Bias current.
ACMP_WarmTime_TypeDef	warmTime Warmup time, which is measured in HFPERCLK / HFPERCCLK cycles and should be about 10 us in wall clock time.
ACMP_HysteresisLevel_TypeDef	hysteresisLevel Hysteresis level.
ACMP_CapsenseResistor_TypeDef	resistor A resistor used in the capacitive sensing circuit.
bool	lowPowerReferenceEnabled Low-power reference enabled.
uint32_t	vddLevel VDD reference value.
bool	enable If true, ACMP is enabled after configuration.

Public Attribute Documentation

fullBias

```
bool ACMP_CapsenseInit_TypeDef::fullBias
```

Full-bias current.

See the ACMP chapter about bias and response time in the reference manual for details.

halfBias

```
bool ACMP_CapsenseInit_TypeDef::halfBias
```

Half-bias current.

See the ACMP chapter about bias and response time in the reference manual for details.

biasProg

```
uint32_t ACMP_CapsenseInit_TypeDef::biasProg
```

Bias current.

See the ACMP chapter about bias and response time in the reference manual for details.

warmTime

```
ACMP_WarmTime_TypeDef ACMP_CapsenseInit_TypeDef::warmTime
```

Warmup time, which is measured in HFPERCLK / HFPERCCLK cycles and should be about 10 us in wall clock time.

hysteresisLevel

```
ACMP_HysteresisLevel_TypeDef ACMP_CapsenseInit_TypeDef::hysteresisLevel
```

Hysteresis level.

resistor

```
ACMP_CapsenseResistor_TypeDef ACMP_CapsenseInit_TypeDef::resistor
```

A resistor used in the capacitive sensing circuit.

For values see the device data sheet.

lowPowerReferenceEnabled

```
bool ACMP_CapsenseInit_TypeDef::lowPowerReferenceEnabled
```

Low-power reference enabled.

This setting, if enabled, reduces the power used by VDD and bandgap references.

vddLevel

```
uint32_t ACMP_CapsenseInit_TypeDef::vddLevel
```

VDD reference value.

VDD_SCALED = (VDD * VDDLEVEL) / 63. Valid values are in the 0-63 range.

enable

```
bool ACMP_CapsenseInit_TypeDef::enable
```

If true, ACMP is enabled after configuration.

ACMP_Init_TypeDef

ACMP initialization structure.

Public Attributes

bool	fullBias Full-bias current.
bool	halfBias Half-bias current.
uint32_t	biasProg Bias current.
bool	interruptOnFallingEdge Enable setting the interrupt flag on the falling edge.
bool	interruptOnRisingEdge Enable setting the interrupt flag on the rising edge.
ACMP_WarmTime_TypeDef	warmTime Warmup time, which is measured in HFPERCLK / HFPERCCLK cycles and should be about 10 us in wall clock time.
ACMP_HysteresisLevel_TypeDef	hysteresisLevel Hysteresis level.
bool	inactiveValue Inactive value emitted by ACMP during warmup.
bool	lowPowerReferenceEnabled Low power reference enabled.
uint32_t	vddLevel VDD reference value.
bool	enable If true, ACMP is enabled after configuration.

Public Attribute Documentation

fullBias

```
bool ACMP_Init_TypeDef::fullBias
```

Full-bias current.

See the ACMP chapter about bias and response time in the reference manual for details.

halfBias

```
bool ACMP_Init_TypeDef::halfBias
```

Half-bias current.

See the ACMP chapter about bias and response time in the reference manual for details.

```
uint32_t ACMP_Init_TypeDef::biasProg
```

Bias current.

See the ACMP chapter about bias and response time in the reference manual for details. Valid values are in the range 0-7.

interruptOnFallingEdge

```
bool ACMP_Init_TypeDef::interruptOnFallingEdge
```

Enable setting the interrupt flag on the falling edge.

interruptOnRisingEdge

```
bool ACMP_Init_TypeDef::interruptOnRisingEdge
```

Enable setting the interrupt flag on the rising edge.

warmTime

```
ACMP_WarmTime_TypeDef ACMP_Init_TypeDef::warmTime
```

Warmup time, which is measured in HFPERCLK / HFPERCCLK cycles and should be about 10 us in wall clock time.

hysteresisLevel

```
ACMP_HysteresisLevel_TypeDef ACMP_Init_TypeDef::hysteresisLevel
```

Hysteresis level.

inactiveValue

```
bool ACMP_Init_TypeDef::inactiveValue
```

Inactive value emitted by ACMP during warmup.

lowPowerReferenceEnabled

```
bool ACMP_Init_TypeDef::lowPowerReferenceEnabled
```

Low power reference enabled.

This setting, if enabled, reduces the power used by the VDD and bandgap references.

vddLevel

```
uint32_t ACMP_Init_TypeDef::vddLevel
```

VDD reference value.

VDD_SCALED = VDD * VDDLEVEL * 50 mV/3.8 V. Valid values are in the 0-63 range.

enable

```
bool ACMP_Init_TypeDef::enable
```

If true, ACMP is enabled after configuration.

ADC - Analog to Digital Converter

ADC - Analog to Digital Converter

Analog to Digital Converter (ADC) Peripheral API.

This module contains functions to control the ADC peripheral of Silicon Labs 32-bit MCUs and SoCs. The ADC is used to convert analog signals into a digital representation.

Modules

ADC_Init_TypeDef

ADC_InitScanInput_TypeDef

ADC_InitScan_TypeDef

ADC_InitSingle_TypeDef

Enumerations

enum ADC_AcqTime_TypeDef {

```
adcAcqTime1 = _ADC_SINGLECTRL_AT_1CYCLE
adcAcqTime2 = _ADC_SINGLECTRL_AT_2CYCLES
adcAcqTime4 = _ADC_SINGLECTRL_AT_4CYCLES
adcAcqTime8 = _ADC_SINGLECTRL_AT_8CYCLES
adcAcqTime16 = _ADC_SINGLECTRL_AT_16CYCLES
adcAcqTime32 = _ADC_SINGLECTRL_AT_32CYCLES
adcAcqTime64 = _ADC_SINGLECTRL_AT_64CYCLES
adcAcqTime128 = _ADC_SINGLECTRL_AT_128CYCLES
adcAcqTime256 = _ADC_SINGLECTRL_AT_256CYCLES
```

}

Acquisition time (in ADC clock cycles).

enum ADC_LPFilter_TypeDef {

```
adcLPFilterBypass = _ADC_CTRL_LPFMODE_BYPASS
adcLPFilterRC = _ADC_CTRL_LPFMODE_RCFILT
adcLPFilterDeCap = _ADC_CTRL_LPFMODE_DECAP
```

}

Lowpass filter mode.

enum ADC_OvsRateSel_TypeDef {

```
adcOvsRateSel2 = _ADC_CTRL_OVSRSEL_X2
adcOvsRateSel4 = _ADC_CTRL_OVSRSEL_X4
adcOvsRateSel8 = _ADC_CTRL_OVSRSEL_X8
adcOvsRateSel16 = _ADC_CTRL_OVSRSEL_X16
adcOvsRateSel32 = _ADC_CTRL_OVSRSEL_X32
adcOvsRateSel64 = _ADC_CTRL_OVSRSEL_X64
adcOvsRateSel128 = _ADC_CTRL_OVSRSEL_X128
adcOvsRateSel256 = _ADC_CTRL_OVSRSEL_X256
adcOvsRateSel512 = _ADC_CTRL_OVSRSEL_X512
adcOvsRateSel1024 = _ADC_CTRL_OVSRSEL_X1024
adcOvsRateSel2048 = _ADC_CTRL_OVSRSEL_X2048
adcOvsRateSel4096 = _ADC_CTRL_OVSRSEL_X4096
```

}

Oversample rate select.

enum ADC_PRSSEL_TypeDef {

adcPRSSELCh0 = _ADC_SINGLECTRL_PRSSEL_PRSCH0
adcPRSSELCh1 = _ADC_SINGLECTRL_PRSSEL_PRSCH1
adcPRSSELCh2 = _ADC_SINGLECTRL_PRSSEL_PRSCH2
adcPRSSELCh3 = _ADC_SINGLECTRL_PRSSEL_PRSCH3
adcPRSSELCh4 = _ADC_SINGLECTRL_PRSSEL_PRSCH4
adcPRSSELCh5 = _ADC_SINGLECTRL_PRSSEL_PRSCH5
adcPRSSELCh6 = _ADC_SINGLECTRL_PRSSEL_PRSCH6
adcPRSSELCh7 = _ADC_SINGLECTRL_PRSSEL_PRSCH7
adcPRSSELCh8 = _ADC_SINGLECTRL_PRSSEL_PRSCH8
adcPRSSELCh9 = _ADC_SINGLECTRL_PRSSEL_PRSCH9
adcPRSSELCh10 = _ADC_SINGLECTRL_PRSSEL_PRSCH10
adcPRSSELCh11 = _ADC_SINGLECTRL_PRSSEL_PRSCH11

}
Peripheral Reflex System signal used to trigger a single sample.

enum ADC_Ref_TypeDef {

adcRef1V25 = _ADC_SINGLECTRL_REF_1V25
adcRef2V5 = _ADC_SINGLECTRL_REF_2V5
adcRefVDD = _ADC_SINGLECTRL_REF_VDD
adcRef5VDIFF = _ADC_SINGLECTRL_REF_5VDIFF
adcRefExtSingle = _ADC_SINGLECTRL_REF_EXTSINGLE
adcRef2xExtDiff = _ADC_SINGLECTRL_REF_2XEXTDIFF
adcRef2xVDD = _ADC_SINGLECTRL_REF_2XVDD

}
Single and scan mode voltage references.

enum ADC_Res_TypeDef {

adcRes12Bit = _ADC_SINGLECTRL_RES_12BIT
adcRes8Bit = _ADC_SINGLECTRL_RES_8BIT
adcRes6Bit = _ADC_SINGLECTRL_RES_6BIT
adcResOVS = _ADC_SINGLECTRL_RES_OVS

}
Sample resolution.

enum ADC_SingleInput_TypeDef {

adcSingleInputCh0 = _ADC_SINGLECTRL_INPUTSEL_CH0
adcSingleInputCh1 = _ADC_SINGLECTRL_INPUTSEL_CH1
adcSingleInputCh2 = _ADC_SINGLECTRL_INPUTSEL_CH2
adcSingleInputCh3 = _ADC_SINGLECTRL_INPUTSEL_CH3
adcSingleInputCh4 = _ADC_SINGLECTRL_INPUTSEL_CH4
adcSingleInputCh5 = _ADC_SINGLECTRL_INPUTSEL_CH5
adcSingleInputCh6 = _ADC_SINGLECTRL_INPUTSEL_CH6
adcSingleInputCh7 = _ADC_SINGLECTRL_INPUTSEL_CH7
adcSingleInputTemp = _ADC_SINGLECTRL_INPUTSEL_TEMP
adcSingleInputVDDDiv3 = _ADC_SINGLECTRL_INPUTSEL_VDDDIV3
adcSingleInputVDD = _ADC_SINGLECTRL_INPUTSEL_VDD
adcSingleInputVSS = _ADC_SINGLECTRL_INPUTSEL_VSS
adcSingleInputVrefDiv2 = _ADC_SINGLECTRL_INPUTSEL_VREFDIV2
adcSingleInputDACOut0 = _ADC_SINGLECTRL_INPUTSEL_DAC0OUT0
adcSingleInputDACOut1 = _ADC_SINGLECTRL_INPUTSEL_DAC0OUT1
adcSingleInputATEST = 15
adcSingleInputCh0Ch1 = _ADC_SINGLECTRL_INPUTSEL_CH0CH1
adcSingleInputCh2Ch3 = _ADC_SINGLECTRL_INPUTSEL_CH2CH3
adcSingleInputCh4Ch5 = _ADC_SINGLECTRL_INPUTSEL_CH4CH5
adcSingleInputCh6Ch7 = _ADC_SINGLECTRL_INPUTSEL_CH6CH7
adcSingleInputDiff0 = 4

}
Single sample input selection.

enum ADC_Start_TypeDef {

adcStartSingle = ADC_CMD_SINGLESTART
adcStartScan = ADC_CMD_SCANSTART
adcStartScanAndSingle = ADC_CMD_SCANSTART | ADC_CMD_SINGLESTART

}
ADC start command.

enum ADC_Warmup_TypeDef {

adcWarmupNormal = _ADC_CTRL_WARMUPMODE_NORMAL
adcWarmupFastBG = _ADC_CTRL_WARMUPMODE_FASTBG
adcWarmupKeepScanRefWarm = _ADC_CTRL_WARMUPMODE_KEEPSCANREFWARM
adcWarmupKeepADCWarm = _ADC_CTRL_WARMUPMODE_KEEPADCWARM

}
Warm-up mode.

Functions

void ADC_Init(ADC_TypeDef *adc, const ADC_Init_TypeDef *init)
Initialize ADC.

void ADC_InitScan(ADC_TypeDef *adc, const ADC_InitScan_TypeDef *init)
Initialize the ADC scan sequence.

void ADC_InitSingle(ADC_TypeDef *adc, const ADC_InitSingle_TypeDef *init)
Initialize the single ADC sample conversion.

uint8_t ADC_PrescaleCalc(uint32_t adcFreq, uint32_t hfperFreq)
Calculate the prescaler value used to determine the ADC clock.

void ADC_Reset(ADC_TypeDef *adc)
Reset ADC to a state that it was in after a hardware reset.

uint8_t ADC_TimebaseCalc(uint32_t hfperFreq)
Calculate a timebase value to get a timebase providing at least 1 us.

uint32_t ADC_DataSingleGet(ADC_TypeDef *adc)
Get a single conversion result.

uint32_t ADC_DataSinglePeek(ADC_TypeDef *adc)
Peek single conversion result.

uint32_t ADC_DataScanGet(ADC_TypeDef *adc)
Get a scan result.

uint32_t ADC_DataScanPeek(ADC_TypeDef *adc)
Peek scan result.

void ADC_IntClear(ADC_TypeDef *adc, uint32_t flags)
Clear one or more pending ADC interrupts.

void ADC_IntDisable(ADC_TypeDef *adc, uint32_t flags)
Disable one or more ADC interrupts.

void ADC_IntEnable(ADC_TypeDef *adc, uint32_t flags)
Enable one or more ADC interrupts.

uint32_t ADC_IntGet(ADC_TypeDef *adc)
Get pending ADC interrupt flags.

uint32_t ADC_IntGetEnabled(ADC_TypeDef *adc)
Get enabled and pending ADC interrupt flags.

void ADC_IntSet(ADC_TypeDef *adc, uint32_t flags)
Set one or more pending ADC interrupts from software.

void ADC_Start(ADC_TypeDef *adc, ADC_Start_TypeDef cmd)
Start scan sequence and/or single conversion.

Macros

#define ADC_INIT_DEFAULT undefined
Default configuration for ADC initialization structure.

#define ADC_INITSCAN_DEFAULT undefined
Default configuration for ADC scan initialization structure.

#define ADC_INITSINGLE_DEFAULT undefined
Default configuration for ADC single conversion initialization structure.

Enumeration Documentation

ADC_AcqTime_TypeDef

ADC_AcqTime_TypeDef

Acquisition time (in ADC clock cycles).

Enumerator	
adcAcqTime1	1 clock cycle.
adcAcqTime2	2 clock cycles.
adcAcqTime4	4 clock cycles.
adcAcqTime8	8 clock cycles.
adcAcqTime16	16 clock cycles.
adcAcqTime32	32 clock cycles.
adcAcqTime64	64 clock cycles.
adcAcqTime128	128 clock cycles.
adcAcqTime256	256 clock cycles.

ADC_LPFilter_TypeDef

ADC_LPFilter_TypeDef

Lowpass filter mode.

Enumerator	
adcLPFilterBypass	No filter or decoupling capacitor.
adcLPFilterRC	On-chip RC filter.
adcLPFilterDeCap	On-chip decoupling capacitor.

ADC_OvsRateSel_TypeDef

ADC_OvsRateSel_TypeDef

Oversample rate select.

Enumerator	
adcOvsRateSel2	2 samples per conversion result.
adcOvsRateSel4	4 samples per conversion result.
adcOvsRateSel8	8 samples per conversion result.
adcOvsRateSel16	16 samples per conversion result.
adcOvsRateSel32	32 samples per conversion result.

adcOvsRateSel64	64 samples per conversion result.
adcOvsRateSel128	128 samples per conversion result.
adcOvsRateSel256	256 samples per conversion result.
adcOvsRateSel512	512 samples per conversion result.
adcOvsRateSel1024	1024 samples per conversion result.
adcOvsRateSel2048	2048 samples per conversion result.
adcOvsRateSel4096	4096 samples per conversion result.

ADC_PRSSEL_TypeDef

> ADC_PRSSEL_TypeDef

Peripheral Reflex System signal used to trigger a single sample.

Enumerator	
adcPRSSELCh0	PRS channel 0.
adcPRSSELCh1	PRS channel 1.
adcPRSSELCh2	PRS channel 2.
adcPRSSELCh3	PRS channel 3.
adcPRSSELCh4	PRS channel 4.
adcPRSSELCh5	PRS channel 5.
adcPRSSELCh6	PRS channel 6.
adcPRSSELCh7	PRS channel 7.
adcPRSSELCh8	PRS channel 8.
adcPRSSELCh9	PRS channel 9.
adcPRSSELCh10	PRS channel 10.
adcPRSSELCh11	PRS channel 11.

ADC_Ref_TypeDef

> ADC_Ref_TypeDef

Single and scan mode voltage references.

Using unshifted enumerations and or in ADC_CTRLX_VREFSEL_REG to select the extension register CTRLX_VREFSEL. ADC Reference

Enumerator	
adcRef1V25	Internal 1.25 V reference.
adcRef2V5	Internal 2.5 V reference.
adcRefVDD	Buffered VDD.
adcRef5VDIFF	Internal differential 5 V reference.
adcRefExtSingle	Single-ended external reference from pin 6.
adcRef2xExtDiff	Differential external reference from pin 6 and 7.
adcRef2xVDD	Unbuffered 2xVDD.

ADC_Res_TypeDef

ADC_Res_TypeDef

Sample resolution.

Enumerator	
adcRes12Bit	12 bit sampling.
adcRes8Bit	8 bit sampling.
adcRes6Bit	6 bit sampling.
adcResOVS	Oversampling.

ADC_SingleInput_TypeDef

ADC_SingleInput_TypeDef

Single sample input selection.

Enumerator	
adcSingleInputCh0	Channel 0.
adcSingleInputCh1	Channel 1.
adcSingleInputCh2	Channel 2.
adcSingleInputCh3	Channel 3.
adcSingleInputCh4	Channel 4.
adcSingleInputCh5	Channel 5.
adcSingleInputCh6	Channel 6.
adcSingleInputCh7	Channel 7.
adcSingleInputTemp	Temperature reference.
adcSingleInputVDDDiv3	VDD divided by 3.
adcSingleInputVDD	VDD.
adcSingleInputVSS	VSS.
adcSingleInputVrefDiv2	Vref divided by 2.
adcSingleInputDACOut0	DAC output 0.
adcSingleInputDACOut1	DAC output 1.
adcSingleInputATEST	ATEST.
adcSingleInputCh0Ch1	Positive Ch0, negative Ch1.
adcSingleInputCh2Ch3	Positive Ch2, negative Ch3.
adcSingleInputCh4Ch5	Positive Ch4, negative Ch5.
adcSingleInputCh6Ch7	Positive Ch6, negative Ch7.
adcSingleInputDiff0	Differential 0.

ADC_Start_TypeDef

ADC_Start_TypeDef

ADC start command.

Enumerator	
adcStartSingle	Start a single conversion.
adcStartScan	Start a scan sequence.
adcStartScanAndSingle	Start a scan sequence and single conversion, typically used when tailgating a single conversion after a scan sequence.

ADC_Warmup_TypeDef

```
ADC_Warmup_TypeDef
```

Warm-up mode.

Enumerator	
adcWarmupNormal	ADC shutdown after each conversion.
adcWarmupFastBG	Do not warm up bandgap references.
adcWarmupKeepScanRefWarm	Reference selected for scan mode kept warm.
adcWarmupKeepADCWarm	ADC and reference selected for scan mode kept at warmup allowing continuous conversion.

Function Documentation

ADC_Init

```
void ADC_Init (ADC_TypeDef * adc, const ADC_Init_TypeDef * init)
```

Initialize ADC.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
const ADC_Init_TypeDef *	[in]	init	A pointer to the ADC initialization structure.

Initializes common parts for both single conversion and scan sequence. In addition, single and/or scan control configuration must be done. See ADC_InitSingle() and ADC_InitScan() respectively. For ADC architectures with the ADCn->SCANINPUTSEL register, use ADC_ScanSingleEndedInputAdd() to configure single-ended scan inputs or ADC_ScanDifferentialInputAdd() to configure differential scan inputs. ADC_ScanInputClear() is also provided for applications that need to update the input configuration.

Note

- This function will stop any ongoing conversion.

ADC_InitScan

```
void ADC_InitScan (ADC_TypeDef * adc, const ADC_InitScan_TypeDef * init)
```

Initialize the ADC scan sequence.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
const ADC_InitScan_TypeDef *	[in]	init	A pointer to the ADC initialization structure.

See ADC_Start() for starting a scan sequence.

When selecting an external reference, the gain and offset calibration must be set explicitly (CAL register). For other references, the calibration is updated with values defined during manufacturing. For ADC architectures with the ADCn->SCANINPUTSEL register, use ADC_ScanSingleEndedInputAdd() to configure single-ended scan inputs or ADC_ScanDifferentialInputAdd() to configure differential scan inputs. ADC_ScanInputClear() is also provided for applications that need to update the input configuration.

Note

- This function will stop any ongoing scan sequence.

ADC_InitSingle

```
void ADC_InitSingle (ADC_TypeDef * adc, const ADC_InitSingle_TypeDef * init)
```

Initialize the single ADC sample conversion.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
const ADC_InitSingle_TypeDef *	[in]	init	A pointer to the ADC initialization structure.

See ADC_Start() for starting a single conversion.

When selecting an external reference, the gain and offset calibration must be set explicitly (CAL register). For other references, the calibration is updated with values defined during manufacturing.

Note

- This function will stop any ongoing single conversion.

ADC_PrescaleCalc

```
uint8_t ADC_PrescaleCalc (uint32_t adcFreq, uint32_t hfperFreq)
```

Calculate the prescaler value used to determine the ADC clock.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	adcFreq	ADC frequency wanted. The frequency will automatically be adjusted to a valid range according to the reference manual.
uint32_t	[in]	hfperFreq	Frequency in Hz of reference HFPER/HFPERC clock. Set to 0 to use currently defined HFPER/HFPERC clock setting.

The ADC clock is given by: (HFPERCLK or HFPERCCLK) / (prescale + 1).

Note

- The return value is clamped to the maximum prescaler value that the hardware supports.

Returns

- A prescaler value to use for ADC in order to achieve a clock value <= `adcFreq`.

ADC_Reset

```
void ADC_Reset (ADC_TypeDef * adc)
```

Reset ADC to a state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to ADC peripheral register block.

Note

- The ROUTE register is NOT reset by this function to allow a centralized setup of this feature.

ADC_TimebaseCalc

```
uint8_t ADC_TimebaseCalc (uint32_t hfperFreq)
```

Calculate a timebase value to get a timebase providing at least 1 us.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	hfperFreq	Frequency in Hz of the reference HFPER/HFPERC clock. Set to 0 to use currently defined HFPER/HFPERC clock setting.

Returns

- A timebase value to use for ADC to achieve at least 1 us.

ADC_DataSingleGet

```
uint32_t ADC_DataSingleGet (ADC_TypeDef * adc)
```

Get a single conversion result.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Note

- Check data valid flag before calling this function.

Returns

- Single conversion data.

ADC_DataSinglePeek

```
uint32_t ADC_DataSinglePeek (ADC_TypeDef * adc)
```

Peek single conversion result.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Note

- Check data valid flag before calling this function.

Returns

- Single conversion data.

ADC_DataScanGet

```
uint32_t ADC_DataScanGet (ADC_TypeDef * adc)
```

Get a scan result.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Note

- Check data valid flag before calling this function.

Returns

- Scan conversion data.

ADC_DataScanPeek

```
uint32_t ADC_DataScanPeek (ADC_TypeDef * adc)
```

Peek scan result.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Note

- Check data valid flag before calling this function.

Returns

- Scan conversion data.

ADC_IntClear

```
void ADC_IntClear (ADC_TypeDef * adc, uint32_t flags)
```

Clear one or more pending ADC interrupts.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
uint32_t	[in]	flags	Pending ADC interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the ADC module (ADC_IF_nnn).

ADC_IntDisable

```
void ADC_IntDisable (ADC_TypeDef * adc, uint32_t flags)
```

Disable one or more ADC interrupts.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
uint32_t	[in]	flags	ADC interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the ADC module (ADC_IF_nnn).

ADC_IntEnable

```
void ADC_IntEnable (ADC_TypeDef * adc, uint32_t flags)
```

Enable one or more ADC interrupts.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
uint32_t	[in]	flags	ADC interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the ADC module (ADC_IF_nnn).

Note

- Depending on use, a pending interrupt may already be set prior to enabling the interrupt. Consider using ADC_IntClear() prior to enabling if the pending interrupt should be ignored.

ADC_IntGet

```
uint32_t ADC_IntGet (ADC_TypeDef * adc)
```

Get pending ADC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Note

- This function does not clear event bits.

Returns

- ADC interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the ADC module (ADC_IF_nnn).

ADC_IntGetEnabled

```
uint32_t ADC_IntGetEnabled (ADC_TypeDef * adc)
```

Get enabled and pending ADC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- This function does not clear interrupt flags.

Returns

- Pending and enabled ADC interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in ADCx_IEN_nnn register (ADCx_IEN_nnn) and
  - the OR combination of valid interrupt flags of the ADC module (ADCx_IF_nnn).

ADC_IntSet

```
void ADC_IntSet (ADC_TypeDef * adc, uint32_t flags)
```

Set one or more pending ADC interrupts from software.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
uint32_t	[in]	flags	ADC interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for the ADC module (ADC_IF_nnn).

ADC_Start

```
void ADC_Start (ADC_TypeDef * adc, ADC_Start_TypeDef cmd)
```

Start scan sequence and/or single conversion.

Parameters

Type	Direction	Argument Name	Description
ADC_TypeDef *	[in]	adc	A pointer to the ADC peripheral register block.
ADC_Start_TypeDef	[in]	cmd	A command indicating which type of sampling to start.

Macro Definition Documentation

ADC_INIT_DEFAULT

```
#define ADC_INIT_DEFAULT
```

Value:

```
{                                                                   \
  adcOvsRateSel2,              /* 2x oversampling (if enabled). */       \
  adcLPFilterBypass,           /* No input filter selected. */           \
  adcWarmupNormal,             /* ADC shutdown after each conversion. */ \
  _ADC_CTRL_TIMEBASE_DEFAULT,  /* Use hardware default value. */         \
  _ADC_CTRL_PRESC_DEFAULT,     /* Use hardware default value. */         \
  false                        /* Do not use tailgate. */                \
}
```

Default configuration for ADC initialization structure.

ADC_INITSCAN_DEFAULT

```
#define ADC_INITSCAN_DEFAULT
```

Value:

```
{                                                                       \
  adcPRSSELCh0,            /* PRS ch0 (if enabled). */                  \
  adcAcqTime1,             /* 1 ADC_CLK cycle acquisition time. */      \
  adcRef1V25,              /* 1.25 V internal reference. */             \
  adcRes12Bit,             /* 12 bit resolution. */                     \
  0,                       /* No input selected. */                     \
  false,                   /* Single-ended input. */                    \
  false,                   /* PRS disabled. */                          \
  false,                   /* Right adjust. */                          \
  false,                   /* Deactivate conversion after one scan sequence. */ \
}
```

Default configuration for ADC scan initialization structure.

ADC_INITSINGLE_DEFAULT

```
#define ADC_INITSINGLE_DEFAULT
```

Value:

```
{                                                                       \
  adcPRSSELCh0,            /* PRS ch0 (if enabled). */                  \
  adcAcqTime1,             /* 1 ADC_CLK cycle acquisition time. */      \
  adcRef1V25,              /* 1.25 V internal reference. */             \
  adcRes12Bit,             /* 12 bit resolution. */                     \
  adcSingleInpCh0,         /* CH0 input selected. */                    \
  false,                   /* Single-ended input. */                    \
  false,                   /* PRS disabled. */                          \
  false,                   /* Right adjust. */                          \
  false                    /* Deactivate conversion after one scan sequence. */ \
}
```

Default configuration for ADC single conversion initialization structure.

ADC_Init_TypeDef

ADC initialization structure, common for single conversion and scan sequence.

Public Attributes

ADC_OvsRateSel_TypeDef	**ovsRateSel** Oversampling rate select.
ADC_LPFilter_TypeDef	**lpfMode** Lowpass or decoupling capacitor filter.
ADC_Warmup_TypeDef	**warmUpMode** ADC Warm-up mode.
uint8_t	**timebase** Timebase for ADC warm up.
uint8_t	**prescale** Clock division factor N, ADC clock = (HFPERCLK or HFPERCCLK) / (N + 1).
bool	**tailgate** Enable/disable conversion tailgating.

Public Attribute Documentation

ovsRateSel

```
ADC_OvsRateSel_TypeDef ADC_Init_TypeDef::ovsRateSel
```

Oversampling rate select.

To have any effect, oversampling must be enabled for single/scan mode.

lpfMode

```
ADC_LPFilter_TypeDef ADC_Init_TypeDef::lpfMode
```

Lowpass or decoupling capacitor filter.

warmUpMode

```
ADC_Warmup_TypeDef ADC_Init_TypeDef::warmUpMode
```

ADC Warm-up mode.

timebase

```
uint8_t ADC_Init_TypeDef::timebase
```

Timebase for ADC warm up.

Select N to give (N+1) HFPERCLK / HFPERCCLK cycles. (Additional delay is added for bandgap references. See the reference manual for more information.) Normally, N should be selected so that the timebase is at least 1 us. See ADC_TimebaseCalc() to obtain a suggested timebase of, at least, 1 us.

prescale

```
uint8_t ADC_Init_TypeDef::prescale
```

Clock division factor N, ADC clock = (HFPERCLK or HFPERCCLK) / (N + 1).

tailgate

```
bool ADC_Init_TypeDef::tailgate
```

Enable/disable conversion tailgating.

ADC_InitScanInput_TypeDef

Scan input configuration.

Public Attributes

uint32_t	scanInputSel Input range select to be applied to ADC_SCANINPUTSEL.
uint32_t	scanInputEn Input enable mask.
uint32_t	scanNegSel Alternative negative input.

Public Attribute Documentation

scanInputSel

```
uint32_t ADC_InitScanInput_TypeDef::scanInputSel
```

Input range select to be applied to ADC_SCANINPUTSEL.

scanInputEn

```
uint32_t ADC_InitScanInput_TypeDef::scanInputEn
```

Input enable mask.

scanNegSel

```
uint32_t ADC_InitScanInput_TypeDef::scanNegSel
```

Alternative negative input.

ADC_InitScan_TypeDef

Scan sequence initialization structure.

Public Attributes

ADC_PRSSEL_TypeDef	**prsSel** Peripheral reflex system trigger selection.
ADC_AcqTime_TypeDef	**acqTime** Acquisition time (in ADC clock cycles).
ADC_Ref_TypeDef	**reference** Sample reference selection.
ADC_Res_TypeDef	**resolution** Sample resolution.
uint32_t	**input** Scan input selection.
bool	**diff** Select if single-ended or differential input.
bool	**prsEnable** Peripheral reflex system trigger enable.
bool	**leftAdjust** Select if left adjustment should be done.
bool	**rep** Select if continuous conversion until explicit stop.

Public Attribute Documentation

prsSel

```
ADC_PRSSEL_TypeDef ADC_InitScan_TypeDef::prsSel
```

Peripheral reflex system trigger selection.

Only applicable if `prsEnable` is enabled.

acqTime

```
ADC_AcqTime_TypeDef ADC_InitScan_TypeDef::acqTime
```

Acquisition time (in ADC clock cycles).

reference

```
ADC_Ref_TypeDef ADC_InitScan_TypeDef::reference
```

Sample reference selection.

Note that, for external references, the ADC calibration register must be set explicitly.

resolution

```
ADC_Res_TypeDef ADC_InitScan_TypeDef::resolution
```

Sample resolution.

input

```
uint32_t ADC_InitScan_TypeDef::input
```

Scan input selection.

If single ended (`diff` is false), use logical combination of ADC_SCANCTRL_INPUTMASK_CHx defines. If differential input (`diff` is true), use logical combination of ADC_SCANCTRL_INPUTMASK_CHxCHy defines. (Notice underscore prefix for defines used.)

diff

```
bool ADC_InitScan_TypeDef::diff
```

Select if single-ended or differential input.

prsEnable

```
bool ADC_InitScan_TypeDef::prsEnable
```

Peripheral reflex system trigger enable.

leftAdjust

```
bool ADC_InitScan_TypeDef::leftAdjust
```

Select if left adjustment should be done.

rep

```
bool ADC_InitScan_TypeDef::rep
```

Select if continuous conversion until explicit stop.

ADC_InitSingle_TypeDef

Single conversion initialization structure.

Public Attributes

ADC_PRSSEL_TypeDef	**prsSel** Peripheral reflex system trigger selection.
ADC_AcqTime_TypeDef	**acqTime** Acquisition time (in ADC clock cycles).
ADC_Ref_TypeDef	**reference** Sample reference selection.
ADC_Res_TypeDef	**resolution** Sample resolution.
ADC_SingleInput_TypeDef	**input** Sample input selection, use single-ended or differential input according to setting of `diff`.
bool	**diff** Select if single-ended or differential input.
bool	**prsEnable** Peripheral reflex system trigger enable.
bool	**leftAdjust** Select if left adjustment should be done.
bool	**rep** Select if continuous conversion until explicit stop.

Public Attribute Documentation

prsSel

> ADC_PRSSEL_TypeDef ADC_InitSingle_TypeDef::prsSel

Peripheral reflex system trigger selection.

Only applicable if `prsEnable` is enabled.

acqTime

> ADC_AcqTime_TypeDef ADC_InitSingle_TypeDef::acqTime

Acquisition time (in ADC clock cycles).

reference

> ADC_Ref_TypeDef ADC_InitSingle_TypeDef::reference

Sample reference selection.

Note that, for external references, the ADC calibration register must be set explicitly.

resolution

```
ADC_Res_TypeDef ADC_InitSingle_TypeDef::resolution
```

Sample resolution.

input

```
ADC_SingleInput_TypeDef ADC_InitSingle_TypeDef::input
```

Sample input selection, use single-ended or differential input according to setting of `diff`.

diff

```
bool ADC_InitSingle_TypeDef::diff
```

Select if single-ended or differential input.

prsEnable

```
bool ADC_InitSingle_TypeDef::prsEnable
```

Peripheral reflex system trigger enable.

leftAdjust

```
bool ADC_InitSingle_TypeDef::leftAdjust
```

Select if left adjustment should be done.

rep

```
bool ADC_InitSingle_TypeDef::rep
```

Select if continuous conversion until explicit stop.

AES - AES Accelerator

AES - AES Accelerator

Advanced Encryption Standard Accelerator (AES) Peripheral API.

This module contains functions to control the AES peripheral of Silicon Labs 32-bit MCUs and SoCs.

The AES peripheral supports AES block cipher encryption and decryption with 128 bit and 256 bit keys. The following block cipher modes are supported:

- CBC - Cipher Block Chaining mode
- CFB - Cipher Feedback mode
- CTR - Counter mode
- ECB - Electronic Code Book mode
- OFB - Output Feedback mode

The following input/output notations should be noted:

- Input/output data (plaintext, ciphertext, key, and so on) are treated as byte arrays, starting with the most significant byte, i.e., 32 bytes of plaintext (B0...B31) is located in memory in the same order, with B0 at the lower address and B31 at the higher address.
- Byte arrays must always be a multiple of AES block size, i.e., a multiple of 16. Padding, if required, is done at the end of the byte array.
- Byte arrays should be word (32 bit) aligned for performance considerations, since the array is accessed with a 32 bit access type. Cortex-M supports unaligned accesses with a performance penalty.
- It is possible to specify the same output buffer as an input buffer as long as they point to the same address. In that case, the provided input buffer is replaced with the encrypted/decrypted output. Notice that buffers must be exactly overlapping. If partly overlapping, the behavior is undefined.

Use a cipher mode according to its requirements to avoid breaking security. See a specific cipher mode theory for details.

References:

- Wikipedia - Cipher modes, http://en.wikipedia.org/wiki/Cipher_modes
- Recommendation for Block Cipher Modes of Operation, NIST Special Publication 800-38A, 2001 Edition, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

The following example shows how to perform an AES-128 CBC encryption:

Enable clocks:

```
/* AES is a HFCORECLK peripheral */
CMU_ClockEnable(cmuClock_AES, true);
```

Execute AES-128 CBC encryption:

```
/* Encrypt a plaintext message (64 bytes) using the AES CBC block cipher mode with a 128 bits key and initial vector
(iv) of 16 bytes.  */
const uint8_t plaintext[64] = { 0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
                                0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A,
                                0xAE, 0x2D, 0x8A, 0x57, 0x1E, 0x03, 0xAC, 0x9C,
                                0x9E, 0xB7, 0x6F, 0xAC, 0x45, 0xAF, 0x8E, 0x51,
                                0x30, 0xC8, 0x1C, 0x46, 0xA3, 0x5C, 0xE4, 0x11,
                                0xE5, 0xFB, 0xC1, 0x19, 0x1A, 0x0A, 0x52, 0xEF,
                                0xF6, 0x9F, 0x24, 0x45, 0xDF, 0x4F, 0x9B, 0x17,
                                0xAD, 0x2B, 0x41, 0x7B, 0xE6, 0x6C, 0x37, 0x10 };
const uint8_t key[16] = { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
                          0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C };
const uint8_t iv[16] = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                         0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F };
uint8_t ciphertext[64]; /* Output buffer for encrypted data (ciphertext). */
AES_CBC128(ciphertext, plaintext, 64, key, iv, true);  /* true means encrypt. */
```

Typedefs

typedef void(* AES_CtrFuncPtr_TypeDef)(uint8_t *ctr)
An AES counter modification function pointer.

Functions

void AES_CBC128(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv, bool encrypt)
Cipher-block chaining (CBC) cipher mode encryption/decryption, 128 bit key.

void AES_CBC256(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv, bool encrypt)
Cipher-block chaining (CBC) cipher mode encryption/decryption, 256 bit key.

void AES_CFB128(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv, bool encrypt)
Cipher feedback (CFB) cipher mode encryption/decryption, 128 bit key.

void AES_CFB256(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv, bool encrypt)
Cipher feedback (CFB) cipher mode encryption/decryption, 256 bit key.

void AES_CTR128(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, uint8_t *ctr, AES_CtrFuncPtr_TypeDef ctrFunc)
Counter (CTR) cipher mode encryption/decryption, 128 bit key.

void AES_CTR256(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, uint8_t *ctr, AES_CtrFuncPtr_TypeDef ctrFunc)
Counter (CTR) cipher mode encryption/decryption, 256 bit key.

void AES_CTRUpdate32Bit(uint8_t *ctr)
Update last 32 bits of 128 bit counter by incrementing with 1.

void AES_DecryptKey128(uint8_t *out, const uint8_t *in)
Generate a 128 bit decryption key from the 128 bit encryption key.

void AES_DecryptKey256(uint8_t *out, const uint8_t *in)
Generate a 256 bit decryption key from the 256 bit encryption key.

void AES_ECB128(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, bool encrypt)
Electronic Codebook (ECB) cipher mode encryption/decryption, 128 bit key.

void AES_ECB256(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, bool encrypt)
Electronic Codebook (ECB) cipher mode encryption/decryption, 256 bit key.

void	AES_OFB128(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv) Output feedback (OFB) cipher mode encryption/decryption, 128 bit key.
void	AES_OFB256(uint8_t *out, const uint8_t *in, unsigned int len, const uint8_t *key, const uint8_t *iv) Output feedback (OFB) cipher mode encryption/decryption, 256 bit key.
void	AES_IntClear(uint32_t flags) Clear one or more pending AES interrupts.
void	AES_IntDisable(uint32_t flags) Disable one or more AES interrupts.
void	AES_IntEnable(uint32_t flags) Enable one or more AES interrupts.
uint32_t	AES_IntGet(void) Get pending AES interrupt flags.
uint32_t	AES_IntGetEnabled(void) Get enabled and pending AES interrupt flags.
void	AES_IntSet(uint32_t flags) Set one or more pending AES interrupts from software.

Typedef Documentation

AES_CtrFuncPtr_TypeDef

```
typedef void(* AES_CtrFuncPtr_TypeDef) (uint8_t *ctr) )(uint8_t *ctr)
```

An AES counter modification function pointer.

Parameters:

- ctr - Ptr to byte array (16 bytes) holding a counter to be modified.

Function Documentation

AES_CBC128

```
void AES_CBC128 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv,
bool encrypt)
```

Cipher-block chaining (CBC) cipher mode encryption/decryption, 128 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.

Type	Direction	Argument Name	Description
const uint8_t *	[in]	key	When encrypting, this is the 128 bit encryption key. When decrypting, this is the 128 bit decryption key. The decryption key may be generated from the encryption key with AES_DecryptKey128(). On devices supporting key buffering, this argument can be null. If so, the key will not be loaded as it is assumed the key has been loaded into KEYHA previously.
const uint8_t *	[in]	iv	128 bit initialization vector.
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

Encryption:

```
*           Plaintext                  Plaintext
*               |                          |
*               V                          V
* InitVector ->XOR        +--------------->XOR
*               |         |                |
*               V         |                V
*       +---------------+ |        +---------------+
* Key ->| Block cipher  | |  Key ->| Block cipher  |
*       |  encryption   | |        |  encryption   |
*       +---------------+ |        +---------------+
*               |---------+                |
*               V                          V
*           Ciphertext                 Ciphertext
*
```

Decryption:

```
*         Ciphertext                 Ciphertext
*              |----------+                |
*              V          |                V
*       +---------------+ |        +---------------+
* Key ->| Block cipher  | |  Key ->| Block cipher  |
*       |  decryption   | |        |  decryption   |
*       +---------------+ |        +---------------+
*               |         |                |
*               V         |                V
* InitVector ->XOR        +--------------->XOR
*               |                          |
*               V                          V
*           Plaintext                  Plaintext
*
```

See general comments on layout and byte ordering of parameters.

AES_CBC256

> void AES_CBC256 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv, bool encrypt)

Cipher-block chaining (CBC) cipher mode encryption/decryption, 256 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	When encrypting, this is the 256 bit encryption key. When decrypting, this is the 256 bit decryption key. The decryption key may be generated from the encryption key with AES_DecryptKey256().
const uint8_t *	[in]	iv	128 bit initialization vector to use.
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

See AES_CBC128() for the CBC figure.

See general comments on layout and byte ordering of parameters.

AES_CFB128

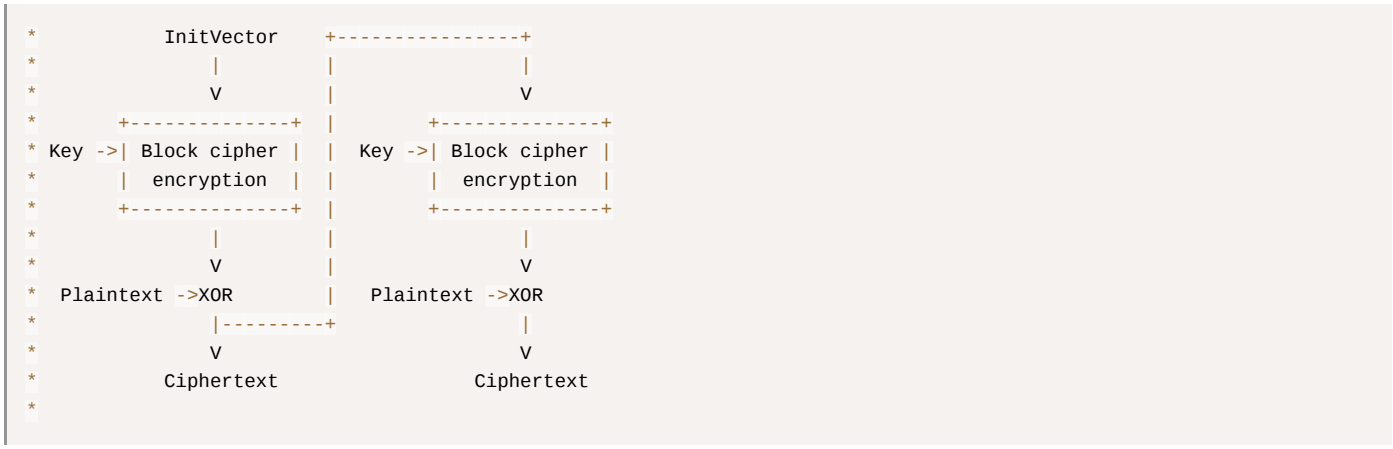
> void AES_CFB128 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv, bool encrypt)

Cipher feedback (CFB) cipher mode encryption/decryption, 128 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	128 bit encryption key is used for both encryption and decryption modes.
const uint8_t *	[in]	iv	128 bit initialization vector to use.
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

Encryption:



Decryption:

```
*          InitVector     +----------------+
*               |         |                |
*               V         |                V
*       +---------------+ |        +---------------+
* Key ->| Block cipher  | |  Key ->| Block cipher  |
*       |  encryption   | |        |  encryption   |
*       +---------------+ |        +---------------+
*               |         |                |
*               V         |                V
*              XOR<- Ciphertext           XOR<- Ciphertext
*               |                          |
*               V                          V
*           Plaintext                  Plaintext
*
```

See general comments on layout and byte ordering of parameters.

AES_CFB256

void AES_CFB256 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv, bool encrypt)

Cipher feedback (CFB) cipher mode encryption/decryption, 256 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	256 bit encryption key is used for both encryption and decryption modes.
const uint8_t *	[in]	iv	128 bit initialization vector to use.
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

See AES_CFB128() for the CFB figure.

See general comments on layout and byte ordering of parameters.

AES_CTR128

void AES_CTR128 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, uint8_t * ctr, AES_CtrFuncPtr_TypeDef ctrFunc)

Counter (CTR) cipher mode encryption/decryption, 128 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.

Type	Direction	Argument Name	Description
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	128 bit encryption key. On devices supporting key buffering this argument can be null. If so, the key will not be loaded, as it is assumed the key has been loaded into KEYHA previously.
uint8_t *	[inout]	ctr	128 bit initial counter value. The counter is updated after each AES block encoding through use of `ctrFunc`.
AES_CtrFuncPtr_TypeDef	[in]	ctrFunc	A function used to update the counter value.

Encryption:

```
*           Counter                    Counter
*             |                          |
*             V                          V
*       +--------------+           +--------------+
* Key ->| Block cipher |     Key ->| Block cipher |
*       |  encryption  |           |  encryption  |
*       +--------------+           +--------------+
*             |                          |
* Plaintext ->XOR            Plaintext ->XOR
*             |                          |
*             V                          V
*         Ciphertext                 Ciphertext
*
```

Decryption:

```
*           Counter                    Counter
*             |                          |
*             V                          V
*       +--------------+           +--------------+
* Key ->| Block cipher |     Key ->| Block cipher |
*       |  encryption  |           |  encryption  |
*       +--------------+           +--------------+
*             |                          |
* Ciphertext ->XOR            Ciphertext ->XOR
*             |                          |
*             V                          V
*          Plaintext                  Plaintext
*
```

See general comments on layout and byte ordering of parameters.

AES_CTR256

void AES_CTR256 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, uint8_t * ctr, AES_CtrFuncPtr_TypeDef ctrFunc)

Counter (CTR) cipher mode encryption/decryption, 256 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	256 bit encryption key.
uint8_t *	[inout]	ctr	128 bit initial counter value. The counter is updated after each AES block encoding through use of `ctrFunc`.
AES_CtrFuncPtr_TypeDef	[in]	ctrFunc	Function used to update counter value.

See AES_CTR128() for CTR figure.

See general comments on layout and byte ordering of parameters.

AES_CTRUpdate32Bit

```
void AES_CTRUpdate32Bit (uint8_t * ctr)
```

Update last 32 bits of 128 bit counter by incrementing with 1.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[inout]	ctr	A buffer holding 128 bit counter to be updated.

Notice that no special consideration is given to possible wrap around. If 32 least significant bits are 0xFFFFFFFF, they will be updated to 0x00000000, ignoring overflow.

See general comments on layout and byte ordering of parameters.

AES_DecryptKey128

```
void AES_DecryptKey128 (uint8_t * out, const uint8_t * in)
```

Generate a 128 bit decryption key from the 128 bit encryption key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place 128 bit decryption key. Must be at least 16 bytes long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding 128 bit encryption key. Must be at least 16 bytes long.

The decryption key is used for some cipher modes when decrypting.

See general comments on layout and byte ordering of parameters.

AES_DecryptKey256

> void AES_DecryptKey256 (uint8_t * out, const uint8_t * in)

Generate a 256 bit decryption key from the 256 bit encryption key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place 256 bit decryption key. Must be at least 32 bytes long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding 256 bit encryption key. Must be at least 32 bytes long.

The decryption key is used for some cipher modes when decrypting.

See general comments on layout and byte ordering of parameters.

AES_ECB128

> void AES_ECB128 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, bool encrypt)

Electronic Codebook (ECB) cipher mode encryption/decryption, 128 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	When encrypting, this is the 128 bit encryption key. When decrypting, this is the 128 bit decryption key. The decryption key may be generated from the encryption key with AES_DecryptKey128().
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

Encryption:

```
*          Plaintext                  Plaintext
*              |                          |
*              V                          V
*       +--------------+           +--------------+
* Key ->| Block cipher |     Key ->| Block cipher |
*       |  encryption  |           |  encryption  |
*       +--------------+           +--------------+
*              |                          |
*              V                          V
*         Ciphertext                 Ciphertext
*
```

Decryption:

```
*         Ciphertext                  Ciphertext
*              |                           |
*              V                           V
*       +--------------+            +--------------+
* Key ->| Block cipher |     Key ->| Block cipher |
*       |  decryption  |            |  decryption  |
*       +--------------+            +--------------+
*              |                           |
*              V                           V
*          Plaintext                   Plaintext
*
```

See general comments on layout and byte ordering of parameters.

AES_ECB256

void AES_ECB256 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, bool encrypt)

Electronic Codebook (ECB) cipher mode encryption/decryption, 256 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	When encrypting, this is the 256 bit encryption key. When decrypting, this is the 256 bit decryption key. The decryption key may be generated from the encryption key with AES_DecryptKey256().
bool	[in]	encrypt	Set to true to encrypt, false to decrypt.

See AES_ECB128() for the ECB figure.

See general comments on layout and byte ordering of parameters.

AES_OFB128

void AES_OFB128 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv)

Output feedback (OFB) cipher mode encryption/decryption, 128 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	128 bit encryption key.

Type	Direction	Argument Name	Description
const uint8_t *	[in]	iv	128 bit initialization vector to use.

Encryption:

```
*          InitVector    +-----------------+
*              |         |                 |
*              V         |                 V
*       +--------------+ |        +--------------+
* Key ->| Block cipher | |  Key ->| Block cipher |
*       |  encryption  | |        |  encryption  |
*       +--------------+ |        +--------------+
*              |         |                 |
*              |---------+                 |
*              V                           V
* Plaintext ->XOR            Plaintext ->XOR
*              |                           |
*              V                           V
*         Ciphertext                  Ciphertext
*
```

Decryption:

```
*          InitVector    +-----------------+
*              |         |                 |
*              V         |                 V
*       +--------------+ |        +--------------+
* Key ->| Block cipher | |  Key ->| Block cipher |
*       |  encryption  | |        |  encryption  |
*       +--------------+ |        +--------------+
*              |         |                 |
*              |---------+                 |
*              V                           V
* Ciphertext ->XOR           Ciphertext ->XOR
*              |                           |
*              V                           V
*          Plaintext                   Plaintext
*
```

See general comments on layout and byte ordering of parameters.

AES_OFB256

> void AES_OFB256 (uint8_t * out, const uint8_t * in, unsigned int len, const uint8_t * key, const uint8_t * iv)

Output feedback (OFB) cipher mode encryption/decryption, 256 bit key.

Parameters

Type	Direction	Argument Name	Description
uint8_t *	[out]	out	A buffer to place encrypted/decrypted data. Must be at least `len` long. It may be set equal to `in`, in which case the input buffer is overwritten.
const uint8_t *	[in]	in	A buffer holding data to encrypt/decrypt. Must be at least `len` long.
unsigned int	[in]	len	A number of bytes to encrypt/decrypt. Must be a multiple of 16.
const uint8_t *	[in]	key	256 bit encryption key.
const uint8_t *	[in]	iv	128 bit initialization vector to use.

See AES_OFB128() for OFB figure.

See general comments on layout and byte ordering of parameters.

AES_IntClear

```
void AES_IntClear (uint32_t flags)
```

Clear one or more pending AES interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	A pending AES interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the AES module (AES_IF_nnn).

AES_IntDisable

```
void AES_IntDisable (uint32_t flags)
```

Disable one or more AES interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	An AES interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the AES module (AES_IF_nnn).

AES_IntEnable

```
void AES_IntEnable (uint32_t flags)
```

Enable one or more AES interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	AES interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the AES module (AES_IF_nnn).

Note

- Depending on use, a pending interrupt may already be set prior to enabling the interrupt. Consider using AES_IntClear() prior to enabling if a pending interrupt should be ignored.

AES_IntGet

```
uint32_t AES_IntGet (void )
```

Get pending AES interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function does not clear event bits.

Returns

- AES interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the AES module (AES_IF_nnn).

AES_IntGetEnabled

```
uint32_t AES_IntGetEnabled (void )
```

Get enabled and pending AES interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- This function does not clear interrupt flags.

Returns

- Pending and enabled AES interrupt sources. The return value is the bitwise AND of
  - the enabled interrupt sources in AES_IEN and
  - the pending interrupt flags AES_IF

AES_IntSet

```
void AES_IntSet (uint32_t flags)
```

Set one or more pending AES interrupts from software.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	AES interrupt sources to set as pending. Use a bitwise logic OR combination of valid interrupt flags for the AES module (AES_IF_nnn).

BURTC - Backup RTC

BURTC - Backup RTC

Backup Real Time Counter (BURTC) Peripheral API.

This module contains functions to control the BURTC peripheral of Silicon Labs 32-bit MCUs. The Backup Real Time Counter allows timekeeping in all energy modes. The Backup RTC is also available when the system is in backup mode.

Modules

BURTC_Init_TypeDef

Enumerations

enum BURTC_ClkSel_TypeDef {
  burtcClkSelULFRCO = BURTC_CTRL_CLKSEL_ULFRCO
  burtcClkSelLFRCO = BURTC_CTRL_CLKSEL_LFRCO
  burtcClkSelLFXO = BURTC_CTRL_CLKSEL_LFXO
}
BURTC clock selection.

enum BURTC_Mode_TypeDef {
  burtcModeDisable = BURTC_CTRL_MODE_DISABLE
  burtcModeEM2 = BURTC_CTRL_MODE_EM2EN
  burtcModeEM3 = BURTC_CTRL_MODE_EM3EN
  burtcModeEM4 = BURTC_CTRL_MODE_EM4EN
}
BURTC mode of operation.

enum BURTC_LP_TypeDef {
  burtcLPDisable = BURTC_LPMODE_LPMODE_DISABLE
  burtcLPEnable = BURTC_LPMODE_LPMODE_ENABLE
  burtcLPBU = BURTC_LPMODE_LPMODE_BUEN
}
BURTC low power mode.

Functions

void BURTC_Init(const BURTC_Init_TypeDef *burtcInit)
Initialize BURTC.

void BURTC_Enable(bool enable)
Enable or Disable BURTC peripheral reset and start counter.

void BURTC_CompareSet(unsigned int comp, uint32_t value)
Set BURTC compare channel.

uint32_t BURTC_CompareGet(unsigned int comp)
Get the BURTC compare value.

void BURTC_CounterReset(void)
Reset counter.

void BURTC_Reset(void)
Restore BURTC to reset state.

uint32_t BURTC_ClockFreqGet(void)
Get the clock frequency of the BURTC.

void BURTC_IntClear(uint32_t flags)
Clear one or more pending BURTC interrupts.

void BURTC_IntDisable(uint32_t flags)
Disable one or more BURTC interrupts.

void BURTC_IntEnable(uint32_t flags)
Enable one or more BURTC interrupts.

uint32_t BURTC_IntGet(void)
Get pending BURTC interrupt flags.

uint32_t BURTC_IntGetEnabled(void)
Get enabled and pending BURTC interrupt flags.

void BURTC_IntSet(uint32_t flags)
Set one or more pending BURTC interrupts from SW.

uint32_t BURTC_Status(void)
Status of BURTC RAM, timestamp and LP Mode.

void BURTC_StatusClear(void)
Clear and reset BURTC status register.

void BURTC_SyncWait(void)
Wait for the BURTC to complete all synchronization of register changes and commands.

uint32_t BURTC_CounterGet(void)
Get BURTC counter.

uint32_t BURTC_TimestampGet(void)
Get BURTC timestamp for entering BU.

void BURTC_FreezeEnable(bool enable)
Freeze register updates until enabled.

void BURTC_Powerdown(bool enable)
Shut down power to retention register bank.

void BURTC_RetRegSet(uint32_t num, uint32_t data)
Set a value in one of the retention registers.

uint32_t BURTC_RetRegGet(uint32_t num)
Read a value from one of the retention registers.

void BURTC_Lock(void)
Lock BURTC registers, which will protect from writing new config settings.

void BURTC_Unlock(void)
Unlock BURTC registers, which will enable write access to change configuration.

Macros

#define burtcClkDiv_1 1
BURTC clock divisors.

#define burtcClkDiv_2 2
Divide clock by 2.

#define burtcClkDiv_4 4
Divide clock by 4.

#define burtcClkDiv_8 8
Divide clock by 8.

#define burtcClkDiv_16 16
Divide clock by 16.

#define burtcClkDiv_32 32
Divide clock by 32.

#define burtcClkDiv_64 64
Divide clock by 64.

#define burtcClkDiv_128 128
Divide clock by 128.

#define BURTC_INIT_DEFAULT undefined
Default configuration for BURTC initialization structure.

Enumeration Documentation

BURTC_ClkSel_TypeDef

> BURTC_ClkSel_TypeDef

BURTC clock selection.

Enumerator	
burtcClkSelULFRCO	Ultra low frequency (1 kHz) clock.
burtcClkSelLFRCO	Low frequency RC oscillator.
burtcClkSelLFXO	Low frequency crystal oscillator.

BURTC_Mode_TypeDef

> BURTC_Mode_TypeDef

BURTC mode of operation.

Enumerator	
burtcModeDisable	Disable BURTC.
burtcModeEM2	Enable and start BURTC counter in EM0 to EM2.
burtcModeEM3	Enable and start BURTC counter in EM0 to EM3.
burtcModeEM4	Enable and start BURTC counter in EM0 to EM4.

BURTC_LP_TypeDef

> BURTC_LP_TypeDef

BURTC low power mode.

Enumerator	
burtcLPDisable	Low Power Mode is disabled.
burtcLPEnable	Low Power Mode is always enabled.
burtcLPBU	Low Power Mode when system enters backup mode.

Function Documentation

BURTC_Init

> void BURTC_Init (const BURTC_Init_TypeDef * burtcInit)

Initialize BURTC.

Parameters

Type	Direction	Argument Name	Description
const BURTC_Init_TypeDef *	[in]	burtcInit	A pointer to the BURTC initialization structure.

Configures the BURTC peripheral.

Note

- Before initialization, BURTC module must first be enabled by clearing the reset bit in the RMU, i.e.,

```
*   RMU_ResetControl(rmuResetBU, rmuResetModeClear);
*
```

  Compare channel 0 must be configured outside this function, before initialization if enable is set to true. The counter will always be reset.

BURTC_Enable

> void BURTC_Enable (bool enable)

Enable or Disable BURTC peripheral reset and start counter.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true; asserts reset to BURTC, halts counter, if false; deassert reset

BURTC_CompareSet

> void BURTC_CompareSet (unsigned int comp, uint32_t value)

Set BURTC compare channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	comp	Compare the channel index, must be 0 for current devices.
uint32_t	[in]	value	New compare value.

BURTC_CompareGet

> uint32_t BURTC_CompareGet (unsigned int comp)

Get the BURTC compare value.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	comp	Compare the channel index value, must be 0 for Giant/Leopard Gecko.

Returns

- The currently configured value for this compare channel.

BURTC_CounterReset

```
void BURTC_CounterReset (void )
```

Reset counter.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

BURTC_Reset

```
void BURTC_Reset (void )
```

Restore BURTC to reset state.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Before accessing the BURTC, BURSTEN in RMU->CTRL must be cleared. LOCK will not be reset to default value, as this will disable access to core BURTC registers.

BURTC_ClockFreqGet

```
uint32_t BURTC_ClockFreqGet (void )
```

Get the clock frequency of the BURTC.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The current frequency in Hz.

BURTC_IntClear

```
void BURTC_IntClear (uint32_t flags)
```

Clear one or more pending BURTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	BURTC interrupt sources to clear. Use a set of interrupt flags OR-ed together to clear multiple interrupt sources for the BURTC module (BURTC_IFS_nnn).

BURTC_IntDisable

```
void BURTC_IntDisable (uint32_t flags)
```

Disable one or more BURTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	BURTC interrupt sources to disable. Use a set of interrupt flags OR-ed together to disable multiple interrupt sources for the BURTC module (BURTC_IFS_nnn).

BURTC_IntEnable

```
void BURTC_IntEnable (uint32_t flags)
```

Enable one or more BURTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	BURTC interrupt sources to enable. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the BURTC module (BURTC_IFS_nnn).

Note

- Depending on use, a pending interrupt may already be set prior to enabling the interrupt. Consider using BURTC_IntClear() prior to enabling if a pending interrupt should be ignored.

BURTC_IntGet

```
uint32_t BURTC_IntGet (void )
```

Get pending BURTC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

-

This function does not clear the event bits.

Returns

- Pending BURTC interrupt sources. Returns a set of interrupt flags OR-ed together for multiple interrupt sources in the BURTC module (BURTC_IFS_nnn).

BURTC_IntGetEnabled

```
uint32_t BURTC_IntGetEnabled (void )
```

Get enabled and pending BURTC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- The event bits are not cleared by the use of this function.

Returns

- Pending BURTC interrupt sources that is also enabled. Returns a set of interrupt flags OR-ed together for multiple interrupt sources in the BURTC module (BURTC_IFS_nnn).

BURTC_IntSet

```
void BURTC_IntSet (uint32_t flags)
```

Set one or more pending BURTC interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	BURTC interrupt sources to set to pending. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the BURTC module (BURTC_IFS_nnn).

BURTC_Status

```
uint32_t BURTC_Status (void )
```

Status of BURTC RAM, timestamp and LP Mode.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- A mask logically OR-ed status bits

BURTC_StatusClear

> void BURTC_StatusClear (void)

Clear and reset BURTC status register.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

BURTC_SyncWait

> void BURTC_SyncWait (void)

Wait for the BURTC to complete all synchronization of register changes and commands.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

BURTC_CounterGet

> uint32_t BURTC_CounterGet (void)

Get BURTC counter.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- BURTC counter value

BURTC_TimestampGet

> uint32_t BURTC_TimestampGet (void)

Get BURTC timestamp for entering BU.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- BURTC Time Stamp value

BURTC_FreezeEnable

> void BURTC_FreezeEnable (bool enable)

Freeze register updates until enabled.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, registers are not updated until enabled again.

BURTC_Powerdown

```
void BURTC_Powerdown (bool enable)
```

Shut down power to retention register bank.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, shuts off power to retention registers.

Note

- When power retention is disabled, it can't be enabled again (until reset).

BURTC_RetRegSet

```
void BURTC_RetRegSet (uint32_t num, uint32_t data)
```

Set a value in one of the retention registers.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	num	Register to set
uint32_t	[in]	data	Value to put into register

BURTC_RetRegGet

```
uint32_t BURTC_RetRegGet (uint32_t num)
```

Read a value from one of the retention registers.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	num	Retention Register to read

Returns

- Value of the retention register

BURTC_Lock

```
void BURTC_Lock (void )
```

Lock BURTC registers, which will protect from writing new config settings.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

BURTC_Unlock

```
void BURTC_Unlock (void )
```

Unlock BURTC registers, which will enable write access to change configuration.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Macro Definition Documentation

burtcClkDiv_1

```
#define burtcClkDiv_1
```

Value:

```
1
```

BURTC clock divisors.

These values are valid for the BURTC prescaler. Divide clock by 1.

burtcClkDiv_2

```
#define burtcClkDiv_2
```

Value:

```
2
```

Divide clock by 2.

burtcClkDiv_4

```
#define burtcClkDiv_4
```

Value:

```
4
```

Divide clock by 4.

burtcClkDiv_8

```
#define burtcClkDiv_8
```

Value:

```
8
```

Divide clock by 8.

burtcClkDiv_16

```
#define burtcClkDiv_16
```

Value:

```
16
```

Divide clock by 16.

burtcClkDiv_32

```
#define burtcClkDiv_32
```

Value:

```
32
```

Divide clock by 32.

burtcClkDiv_64

```
#define burtcClkDiv_64
```

Value:

```
64
```

Divide clock by 64.

burtcClkDiv_128

```
#define burtcClkDiv_128
```

Value:

```
128
```

Divide clock by 128.

BURTC_INIT_DEFAULT

```
#define BURTC_INIT_DEFAULT
```

Value:

```
{                       \
  true,                 \
  burtcModeEM2,         \
  false,                \
  burtcClkSelULFRCO,    \
  burtcClkDiv_1,        \
  0,                    \
  true,                 \
  false,                \
  burtcLPDisable,       \
}
```

Default configuration for BURTC initialization structure.

BURTC_Init_TypeDef

BURTC initialization structure for Series 0 devices.

Public Attributes

bool	enable Enable BURTC after initialization (starts counter).
BURTC_Mode_TypeDef	mode Configure energy mode operation.
bool	debugRun If true, counter will keep running under debug halt.
BURTC_ClkSel_TypeDef	clkSel Select clock source.
uint32_t	clkDiv Clock divider; for ULFRCO 1Khz or 2kHz operation.
uint32_t	lowPowerComp Number of least significant clock bits to ignore in low power mode.
bool	timeStamp Enable time stamp on entering backup power domain.
bool	compare0Top Set if Compare Value 0 is also top value (counter restart).
BURTC_LP_TypeDef	lowPowerMode Low power operation mode, requires LFXO or LFRCO.

Public Attribute Documentation

enable

```
bool BURTC_Init_TypeDef::enable
```

Enable BURTC after initialization (starts counter).

mode

```
BURTC_Mode_TypeDef BURTC_Init_TypeDef::mode
```

Configure energy mode operation.

debugRun

```
bool BURTC_Init_TypeDef::debugRun
```

If true, counter will keep running under debug halt.

clkSel

```
BURTC_ClkSel_TypeDef BURTC_Init_TypeDef::clkSel
```

Select clock source.

clkDiv

```
uint32_t BURTC_Init_TypeDef::clkDiv
```

Clock divider; for ULFRCO 1Khz or 2kHz operation.

lowPowerComp

```
uint32_t BURTC_Init_TypeDef::lowPowerComp
```

Number of least significant clock bits to ignore in low power mode.

timeStamp

```
bool BURTC_Init_TypeDef::timeStamp
```

Enable time stamp on entering backup power domain.

compare0Top

```
bool BURTC_Init_TypeDef::compare0Top
```

Set if Compare Value 0 is also top value (counter restart).

lowPowerMode

```
BURTC_LP_TypeDef BURTC_Init_TypeDef::lowPowerMode
```

Low power operation mode, requires LFXO or LFRCO.

BUS - Bitfield Read/Write

BUS - Bitfield Read/Write

BUS register and RAM bit/field read/write API.

API to perform bit-band and field set/clear access to RAM and peripherals.

Functions

void	BUS_RamBitWrite(volatile uint32_t *addr, unsigned int bit, unsigned int val) Perform a single-bit write operation on a 32-bit word in RAM.
unsigned int	BUS_RamBitRead(volatile const uint32_t *addr, unsigned int bit) Perform a single-bit read operation on a 32-bit word in RAM.
void	BUS_RegBitWrite(volatile uint32_t *addr, unsigned int bit, unsigned int val) Perform a single-bit write operation on a peripheral register.
unsigned int	BUS_RegBitRead(volatile const uint32_t *addr, unsigned int bit) Perform a single-bit read operation on a peripheral register.
void	BUS_RegMaskedSet(volatile uint32_t *addr, uint32_t mask) Perform a masked set operation on a peripheral register address.
void	BUS_RegMaskedClear(volatile uint32_t *addr, uint32_t mask) Perform a masked clear operation on the peripheral register address.
void	BUS_RegMaskedWrite(volatile uint32_t *addr, uint32_t mask, uint32_t val) Perform peripheral register masked write.
uint32_t	BUS_RegMaskedRead(volatile const uint32_t *addr, uint32_t mask) Perform a peripheral register masked read.

Function Documentation

BUS_RamBitWrite

> void BUS_RamBitWrite (volatile uint32_t * addr, unsigned int bit, unsigned int val)

Perform a single-bit write operation on a 32-bit word in RAM.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	addr	An ddress of a 32-bit word in RAM.
unsigned int	[in]	bit	A bit position to write, 0-31.
unsigned int	[in]	val	A value to set bit to, 0 or 1.

This function uses Cortex-M bit-banding hardware to perform an atomic read-modify-write operation on a single bit write on a 32-bit word in RAM. See the reference manual for more details about bit-banding.

Note

- This function is atomic on Cortex-M cores with bit-banding support. Bit- banding is a multi cycle read-modify-write bus operation. RAM bit-banding is performed using the memory alias region at BITBAND_RAM_BASE.

BUS_RamBitRead

```
unsigned int BUS_RamBitRead (volatile const uint32_t * addr, unsigned int bit)
```

Perform a single-bit read operation on a 32-bit word in RAM.

Parameters

Type	Direction	Argument Name	Description
volatile const uint32_t *	[in]	addr	RAM address.
unsigned int	[in]	bit	A bit position to read, 0-31.

This function uses Cortex-M bit-banding hardware to perform an atomic read operation on a single register bit. See the reference manual for more details about bit-banding.

Note

- This function is atomic on Cortex-M cores with bit-banding support. RAM bit-banding is performed using the memory alias region at BITBAND_RAM_BASE.

Returns

- The requested bit shifted to bit position 0 in the return value.

BUS_RegBitWrite

```
void BUS_RegBitWrite (volatile uint32_t * addr, unsigned int bit, unsigned int val)
```

Perform a single-bit write operation on a peripheral register.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	addr	A peripheral register address.
unsigned int	[in]	bit	A bit position to write, 0-31.
unsigned int	[in]	val	A value to set bit to, 0 or 1.

This function uses Cortex-M bit-banding hardware to perform an atomic read-modify-write operation on a single register bit. See the reference manual for more details about bit-banding.

Note

- This function is atomic on Cortex-M cores with bit-banding support. Bit- banding is a multi cycle read-modify-write bus operation. Peripheral register bit-banding is performed using the memory alias region at BITBAND_PER_BASE.

BUS_RegBitRead

```
unsigned int BUS_RegBitRead (volatile const uint32_t * addr, unsigned int bit)
```

Perform a single-bit read operation on a peripheral register.

Parameters

Type	Direction	Argument Name	Description
volatile const uint32_t *	[in]	addr	A peripheral register address.

Type	Direction	Argument Name	Description
unsigned int	[in]	bit	A bit position to read, 0-31.

This function uses Cortex-M bit-banding hardware to perform an atomic read operation on a single register bit. See the reference manual for more details about bit-banding.

Note

- This function is atomic on Cortex-M cores with bit-banding support. Peripheral register bit-banding is performed using the memory alias region at BITBAND_PER_BASE.

Returns

- The requested bit shifted to bit position 0 in the return value.

BUS_RegMaskedSet

```
void BUS_RegMaskedSet (volatile uint32_t * addr, uint32_t mask)
```

Perform a masked set operation on a peripheral register address.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	addr	A peripheral register address.
uint32_t	[in]	mask	A mask to set.

A peripheral register masked set provides a single-cycle and atomic set operation of a bit-mask in a peripheral register. All 1s in the mask are set to 1 in the register. All 0s in the mask are not changed in the register. RAMs and special peripherals are not supported. See the reference manual for more details about the peripheral register field set.

Note

- This function is single-cycle and atomic on cores with peripheral bit set and clear support. It uses the memory alias region at PER_BITSET_MEM_BASE.

BUS_RegMaskedClear

```
void BUS_RegMaskedClear (volatile uint32_t * addr, uint32_t mask)
```

Perform a masked clear operation on the peripheral register address.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	addr	A peripheral register address.
uint32_t	[in]	mask	A mask to clear.

A peripheral register masked clear provides a single-cycle and atomic clear operation of a bit-mask in a peripheral register. All 1s in the mask are set to 0 in the register. All 0s in the mask are not changed in the register. RAMs and special peripherals are not supported. See the reference manual for more details about the peripheral register field clear.

Note

- This function is single-cycle and atomic on cores with peripheral bit set and clear support. It uses the memory alias region at PER_BITCLR_MEM_BASE.

BUS_RegMaskedWrite

```
void BUS_RegMaskedWrite (volatile uint32_t * addr, uint32_t mask, uint32_t val)
```

Perform peripheral register masked write.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	addr	A peripheral register address.
uint32_t	[in]	mask	A peripheral register mask.
uint32_t	[in]	val	A peripheral register value. The value must be shifted to the correct bit position in the register corresponding to the field defined by the mask parameter. The register value must be contained in the field defined by the mask parameter. The register value is masked to prevent involuntary spillage.

This function first reads the peripheral register and updates only bits that are set in the mask with content of val. Typically, the mask is a bit-field in the register and the value val is within the mask.

Note

- The read-modify-write operation is executed in a critical section to guarantee atomicity. Note that atomicity can only be guaranteed if register is modified only by the core, and not by other peripherals (like DMA).

BUS_RegMaskedRead

```
uint32_t BUS_RegMaskedRead (volatile const uint32_t * addr, uint32_t mask)
```

Perform a peripheral register masked read.

Parameters

Type	Direction	Argument Name	Description
volatile const uint32_t *	[in]	addr	A peripheral register address.
uint32_t	[in]	mask	A peripheral register mask.

Read an unshifted and masked value from a peripheral register.

Note

- This operation is not hardware accelerated.

Returns

- An unshifted and masked register value.

CHIP - Chip Errata Workarounds

CHIP - Chip Errata Workarounds

Chip errata workaround APIs.

API to apply chip errata workarounds at initialization and reset.

Functions

void CHIP_Init(void)
Chip initialization routine for revision errata workarounds.

void CHIP_Reset(void)
Chip reset routine with errata workarounds.

Function Documentation

CHIP_Init

```
void CHIP_Init (void )
```

Chip initialization routine for revision errata workarounds.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function must be called immediately in main().

This initialization function configures the device to a state as similar to later revisions as possible to improve software compatibility with newer parts. See the device-specific errata for details.

CHIP_Reset

```
void CHIP_Reset (void )
```

Chip reset routine with errata workarounds.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function should be called to reset the chip. It does not return.

This function applies any errata workarounds needed to cleanly reset the device and then performs a system reset. See the device-specific errata for details.

CMU - Clock Management Unit

CMU - Clock Management Unit

Clock management unit (CMU) Peripheral API.

This module contains functions for the CMU peripheral of Silicon Labs 32-bit MCUs and SoCs. The CMU module controls oscillators, clocks gates, clock multiplexers, pre-scalers, calibration modules and wait-states.

Modules

CMU_LFXOInit_TypeDef

CMU_HFXOInit_TypeDef

Enumerations

enum CMU_HFRCOBand_TypeDef {

cmuHFRCOBand_1MHz = _CMU_HFRCOCTRL_BAND_1MHZ
cmuHFRCOBand_7MHz = _CMU_HFRCOCTRL_BAND_7MHZ
cmuHFRCOBand_11MHz = _CMU_HFRCOCTRL_BAND_11MHZ
cmuHFRCOBand_14MHz = _CMU_HFRCOCTRL_BAND_14MHZ
cmuHFRCOBand_21MHz = _CMU_HFRCOCTRL_BAND_21MHZ
cmuHFRCOBand_28MHz = _CMU_HFRCOCTRL_BAND_28MHZ

}

High-frequency system RCO bands.

enum CMU_AUXHFRCOBand_TypeDef {

cmuAUXHFRCOBand_1MHz = _CMU_AUXHFRCOCTRL_BAND_1MHZ
cmuAUXHFRCOBand_7MHz = _CMU_AUXHFRCOCTRL_BAND_7MHZ
cmuAUXHFRCOBand_11MHz = _CMU_AUXHFRCOCTRL_BAND_11MHZ
cmuAUXHFRCOBand_14MHz = _CMU_AUXHFRCOCTRL_BAND_14MHZ
cmuAUXHFRCOBand_21MHz = _CMU_AUXHFRCOCTRL_BAND_21MHZ
cmuAUXHFRCOBand_28MHz = _CMU_AUXHFRCOCTRL_BAND_28MHZ

}

AUX high-frequency RCO bands.

enum CMU_Clock_TypeDef {

cmuClock_HF = (CMU_HFCLKDIV_REG << CMU_DIV_REG_POS) | (CMU_HFCLKSEL_REG << CMU_SEL_REG_POS) | (CMU_NO_EN_REG << CMU_EN_REG_POS) | (0 << CMU_EN_BIT_POS) | (CMU_HF_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_DBG = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_DBGCLKSEL_REG << CMU_SEL_REG_POS) | (CMU_NO_EN_REG << CMU_EN_REG_POS) | (0 << CMU_EN_BIT_POS) | (CMU_DBG_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_AUX = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_NO_EN_REG << CMU_EN_REG_POS) | (0 << CMU_EN_BIT_POS) | (CMU_AUX_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_HFPER = (CMU_HFPERCLKDIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKDIV_EN_REG << CMU_EN_REG_POS) | (_CMU_HFPERCLKDIV_HFPERCLKEN_SHIFT << CMU_EN_BIT_POS) | (CMU_HFPER_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_USARTRF0 = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKEN0_EN_REG << CMU_EN_REG_POS) | (_CMU_HFPERCLKEN0_USARTRF0_SHIFT << CMU_EN_BIT_POS) | (CMU_HFPER_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_USART1 = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKEN0_EN_REG << CMU_EN_REG_POS) | (_CMU_HFPERCLKEN0_USART1_SHIFT << CMU_EN_BIT_POS) | (CMU_HFPER_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_USART2 = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKEN0_EN_REG << CMU_EN_REG_POS) | (_CMU_HFPERCLKEN0_USART2_SHIFT << CMU_EN_BIT_POS) | (CMU_HFPER_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_UART0 = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKEN0_EN_REG << CMU_EN_REG_POS) | (_CMU_HFPERCLKEN0_UART0_SHIFT << CMU_EN_BIT_POS) | (CMU_HFPER_CLK_BRANCH << CMU_CLK_BRANCH_POS)
cmuClock_UART1 = (CMU_NODIV_REG << CMU_DIV_REG_POS) | (CMU_NOSEL_REG << CMU_SEL_REG_POS) | (CMU_HFPERCLKEN0_EN_REG << CMU_EN_REG_POS) |

```
(_CMU_HFPERCLKEN0_UART1_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_TIMER0 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_TIMER0_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_TIMER1 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_TIMER1_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_TIMER2 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_TIMER2_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_TIMER3 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_TIMER3_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_ACMP0 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_ACMP0_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_ACMP1 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_ACMP1_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_PRS = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_PRS_SHIFT <<
CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_DAC0 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_DAC0_SHIFT
<< CMU_EN_BIT_POS) |
```

```
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_GPIO =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_GPIO_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_VCMP =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_VCMP_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_ADC0 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_ADC0_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_I2C0 =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_I2C0_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_I2C1 = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFPERCLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFPERCLKEN0_I2C1_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFPER_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_CORE =
(CMU_HFCORECLKDIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_NO_EN_REG <<
CMU_EN_REG_POS) | (0 <<
CMU_EN_BIT_POS) |
(CMU_HFCORE_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_AES = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFCORECLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFCORECLKEN0_AES_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFCORE_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_DMA =
(CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_HFCORECLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_HFCORECLKEN0_DMA_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_HFCORE_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_HFLE =
(CMU_HFCORECLKLEDIV_REG <<
```

```
(_CMU_HFCORECLKEN0_LE_SHIFT <<
CMU_EN_BIT_POS) |
(CMU_HFLE_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_LFA = (CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_LFACLKSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_NO_EN_REG <<
CMU_EN_REG_POS) | (0 <<
CMU_EN_BIT_POS) |
(CMU_LFA_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_RTC =
(CMU_LFAPRESC0_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_LFACLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_LFACLKEN0_RTC_SHIFT <<
CMU_EN_BIT_POS) |
(CMU_RTC_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_LETIMER0 =
(CMU_LFAPRESC0_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_LFACLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_LFACLKEN0_LETIMER0_SHIFT <<
CMU_EN_BIT_POS) |
(CMU_LETIMER0_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_PCNT0 = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_PCNT_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_PCNTCTRL_PCNT0CLKEN_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_LFA_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_PCNT1 = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_PCNT_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_PCNTCTRL_PCNT1CLKEN_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_LFA_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_PCNT2 = (CMU_NODIV_REG
<< CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_PCNT_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_PCNTCTRL_PCNT2CLKEN_SHIFT
<< CMU_EN_BIT_POS) |
(CMU_LFA_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_LESENSE =
(CMU_LFAPRESC0_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_LFACLKEN0_EN_REG <<
CMU_EN_REG_POS) |
(_CMU_LFACLKEN0_LESENSE_SHIFT <<
CMU_EN_BIT_POS) |
(CMU_LESENSE_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_LFB = (CMU_NODIV_REG <<
CMU_DIV_REG_POS) |
(CMU_LFBCLKSEL_REG <<
CMU_SEL_REG_POS) |
(CMU_NO_EN_REG <<
CMU_EN_REG_POS) | (0 <<
CMU_EN_BIT_POS) |
(CMU_LFB_CLK_BRANCH <<
CMU_CLK_BRANCH_POS)
cmuClock_LEUART0 =
(CMU_LFBPRESC0_REG <<
CMU_DIV_REG_POS) |
(CMU_NOSEL_REG <<
```

```
  CMU_SEL_REG_POS) |
  (CMU_LFBCLKEN0_EN_REG <<
  CMU_EN_REG_POS) |
  (_CMU_LFBCLKEN0_LEUART0_SHIFT
  << CMU_EN_BIT_POS) |
  (CMU_LEUART0_CLK_BRANCH <<
  CMU_CLK_BRANCH_POS)
  cmuClock_LEUART1 =
  (CMU_LFBPRESC0_REG <<
  CMU_DIV_REG_POS) |
  (CMU_NOSEL_REG <<
  CMU_SEL_REG_POS) |
  (CMU_LFBCLKEN0_EN_REG <<
  CMU_EN_REG_POS) |
  (_CMU_LFBCLKEN0_LEUART1_SHIFT
  << CMU_EN_BIT_POS) |
  (CMU_LEUART1_CLK_BRANCH <<
  CMU_CLK_BRANCH_POS)
}
```

Clock points in CMU.

enum CMU_Osc_TypeDef {

```
  cmuOsc_LFXO
  cmuOsc_LFRCO
  cmuOsc_HFXO
  cmuOsc_HFRCO
  cmuOsc_AUXHFRCO
  cmuOsc_ULFRCO
}
```

Oscillator types.

enum CMU_OscMode_TypeDef {

```
  cmuOscMode_Crystal
  cmuOscMode_AcCoupled
  cmuOscMode_External
}
```

Oscillator modes.

enum CMU_Select_TypeDef {

```
  cmuSelect_Error
  cmuSelect_Disabled
  cmuSelect_LFXO
  cmuSelect_LFRCO
  cmuSelect_HFXO
  cmuSelect_HFRCO
  cmuSelect_HFCLKLE
  cmuSelect_AUXHFRCO
  cmuSelect_HFSRCCLK
  cmuSelect_HFCLK
  cmuSelect_ULFRCO
}
```

Selectable clock sources.

enum CMU_LFXOBoost_TypeDef {

```
  cmuLfxoBoost70 = 0x0
  cmuLfxoBoost100 = 0x2
}
```

LFXO Boost values.

Typedefs

typedef uint32_t CMU_ClkDiv_TypeDef

Clock divider configuration.

Functions

CMU_AUXHFRCOBand_TypeDef CMU_AUXHFRCOBandGet(void)

Get the AUXHFRCO band in use.

void CMU_AUXHFRCOBandSet(CMU_AUXHFRCOBand_TypeDef band)

Set the AUXHFRCO band and the tuning value based on the value in the calibration table made during production.

uint32_t	CMU_Calibrate(uint32_t HFCycles, CMU_Osc_TypeDef reference) Calibrate the clock.
void	CMU_CalibrateConfig(uint32_t downCycles, CMU_Osc_TypeDef downSel, CMU_Osc_TypeDef upSel) Configure the clock calibration.
uint32_t	CMU_CalibrateCountGet(void) Get the calibration count register.
void	CMU_ClockEnable(CMU_Clock_TypeDef clock, bool enable) Enable/disable a clock.
CMU_ClkDiv_TypeDef	CMU_ClockDivGet(CMU_Clock_TypeDef clock) Get the clock divisor/prescaler.
void	CMU_ClockDivSet(CMU_Clock_TypeDef clock, CMU_ClkDiv_TypeDef div) Set the clock divisor/prescaler.
uint32_t	CMU_ClockFreqGet(CMU_Clock_TypeDef clock) Get the clock frequency for a clock point.
void	CMU_ClockSelectSet(CMU_Clock_TypeDef clock, CMU_Select_TypeDef ref) Select the reference clock/oscillator used for a clock branch.
CMU_Select_TypeDef	CMU_ClockSelectGet(CMU_Clock_TypeDef clock) Get the currently selected reference clock used for a clock branch.
uint16_t	CMU_LF_ClockPrecisionGet(CMU_Clock_TypeDef clock) Gets the precision (in PPM) of the specified low frequency clock branch.
uint16_t	CMU_HF_ClockPrecisionGet(CMU_Clock_TypeDef clock) Gets the precision (in PPM) of the specified high frequency clock branch.
void	CMU_FreezeEnable(bool enable) CMU low frequency register synchronization freeze control.
CMU_HFRCOBand_TypeDef	CMU_HFRCOBandGet(void) Get HFRCO band in use.
void	CMU_HFRCOBandSet(CMU_HFRCOBand_TypeDef band) Set HFRCO band and the tuning value based on the value in the calibration table made during production.
uint32_t	CMU_HFRCOStartupDelayGet(void) Get the HFRCO startup delay.
void	CMU_HFRCOStartupDelaySet(uint32_t delay) Set the HFRCO startup delay.
void	CMU_HFXOInit(const CMU_HFXOInit_TypeDef *hfxoInit) Set HFXO control registers.
uint32_t	CMU_LCDClkFDIVGet(void) Get the LCD framerate divisor (FDIV) setting.
void	CMU_LCDClkFDIVSet(uint32_t div) Set the LCD framerate divisor (FDIV) setting.
void	CMU_LFXOInit(const CMU_LFXOInit_TypeDef *lfxoInit) Set LFXO control registers.
void	CMU_LFXOPrecisionSet(uint16_t precision)

Sets LFXO's crystal precision, in PPM.

uint16_t	CMU_LFXOPrecisionGet(void) Gets LFXO's crystal precision, in PPM.
void	CMU_HFXOPrecisionSet(uint16_t precision) Sets HFXO's crystal precision, in PPM.
uint16_t	CMU_HFXOPrecisionGet(void) Gets HFXO's crystal precision, in PPM.
void	CMU_OscillatorEnable(CMU_Osc_TypeDef osc, bool enable, bool wait) Enable/disable oscillator.
uint32_t	CMU_OscillatorTuningGet(CMU_Osc_TypeDef osc) Get the oscillator frequency tuning setting.
void	CMU_OscillatorTuningSet(CMU_Osc_TypeDef osc, uint32_t val) Set the oscillator frequency tuning control.
void	CMU_PCNTClockExternalSet(unsigned int instance, bool external) Select the PCNTn clock.
bool	CMU_PCNTClockExternalGet(unsigned int instance) Determine if the currently selected PCNTn clock used is external or LFBCLK.
void	CMU_UpdateWaitStates(uint32_t freq, int vscale) Configure various wait states to switch to a certain frequency and a certain voltage scale.
void	CMU_CalibrateCont(bool enable) Configure continuous calibration mode.
void	CMU_CalibrateStart(void) Start calibration.
void	CMU_CalibrateStop(void) Stop the calibration counters.
uint32_t	CMU_DivToLog2(CMU_ClkDiv_TypeDef div) Convert divider to logarithmic value.
void	CMU_IntClear(uint32_t flags) Clear one or more pending CMU interrupts.
void	CMU_IntDisable(uint32_t flags) Disable one or more CMU interrupts.
void	CMU_IntEnable(uint32_t flags) Enable one or more CMU interrupts.
uint32_t	CMU_IntGet(void) Get pending CMU interrupts.
uint32_t	CMU_IntGetEnabled(void) Get enabled and pending CMU interrupt flags.
void	CMU_IntSet(uint32_t flags) Set one or more pending CMU interrupts.
void	CMU_Lock(void) Lock the CMU to protect some of its registers against unintended modification.
void	CMU_Unlock(void) Unlock the CMU so that writing to locked registers again is possible.
uint32_t	auxClkGet(void) Get the AUX clock frequency.
uint32_t	dbgClkGet(void)

Get the Debug Trace clock frequency.

void flashWaitStateControl(uint32_t coreFreq, int vscale)
Configure flash access wait states to support the given core clock frequency.

void flashWaitStateMax(void)
Configure flash access wait states to the most conservative setting for this target.

uint32_t lfClkGet(CMU_Clock_TypeDef lfClkBranch)
Get the LFnCLK frequency based on the current configuration.

void syncReg(uint32_t mask)
Wait for an ongoing sync of register(s) to low-frequency domain to complete.

void hfperClkSafePrescaler(void)
Set HFPER clock tree prescalers to safe values.

void hfperClkOptimizedPrescaler(void)
Set HFPER clock tree prescalers to give highest possible clock node frequency while still beeing within spec.

Macros

#define CMU_CLOCK_SELECT_SET (clock, sel)
Macro to set clock sources in the clock tree.

#define cmuClkDiv_1 1
Clock divisors.

#define cmuClkDiv_2 2
Divide clock by 2.

#define cmuClkDiv_4 4
Divide clock by 4.

#define cmuClkDiv_8 8
Divide clock by 8.

#define cmuClkDiv_16 16
Divide clock by 16.

#define cmuClkDiv_32 32
Divide clock by 32.

#define cmuClkDiv_64 64
Divide clock by 64.

#define cmuClkDiv_128 128
Divide clock by 128.

#define cmuClkDiv_256 256
Divide clock by 256.

#define cmuClkDiv_512 512
Divide clock by 512.

#define cmuClkDiv_1024 1024
Divide clock by 1024.

#define cmuClkDiv_2048 2048
Divide clock by 2048.

#define cmuClkDiv_4096 4096
Divide clock by 4096.

#define cmuClkDiv_8192 8192
Divide clock by 8192.

#define cmuClkDiv_16384 16384

Divide clock by 16384.

#define cmuClkDiv_32768 32768
Divide clock by 32768.

#define CMU_LFXOINIT_DEFAULT undefined
Default LFXO initialization values.

#define CMU_LFXOINIT_EXTERNAL_CLOCK undefined
Default LFXO initialization for external clock.

#define CMU_HFXOINIT_DEFAULT undefined
Default HFXO initialization values for Platform 1 devices.

#define CMU_HFXOINIT_EXTERNAL_CLOCK undefined
Default HFXO initialization for external clock.

Enumeration Documentation

CMU_HFRCOBand_TypeDef

> CMU_HFRCOBand_TypeDef

High-frequency system RCO bands.

Enumerator	
cmuHFRCOBand_1MHz	1 MHz HFRCO band
cmuHFRCOBand_7MHz	7 MHz HFRCO band
cmuHFRCOBand_11MHz	11 MHz HFRCO band
cmuHFRCOBand_14MHz	14 MHz HFRCO band
cmuHFRCOBand_21MHz	21 MHz HFRCO band
cmuHFRCOBand_28MHz	28 MHz HFRCO band

CMU_AUXHFRCOBand_TypeDef

> CMU_AUXHFRCOBand_TypeDef

AUX high-frequency RCO bands.

Enumerator	
cmuAUXHFRCOBand_1MHz	1 MHz RC band
cmuAUXHFRCOBand_7MHz	7 MHz RC band
cmuAUXHFRCOBand_11MHz	11 MHz RC band
cmuAUXHFRCOBand_14MHz	14 MHz RC band
cmuAUXHFRCOBand_21MHz	21 MHz RC band
cmuAUXHFRCOBand_28MHz	28 MHz RC band

CMU_Clock_TypeDef

> CMU_Clock_TypeDef

Clock points in CMU.

See CMU overview in the reference manual.

Enumerator	
cmuClock_HF	High-frequency clock.
cmuClock_DBG	Debug clock.
cmuClock_AUX	AUX clock.
cmuClock_HFPER	High-frequency peripheral clock.
cmuClock_USARTRF0	Universal sync/async receiver/transmitter 0 clock.
cmuClock_USART1	Universal sync/async receiver/transmitter 1 clock.
cmuClock_USART2	Universal sync/async receiver/transmitter 2 clock.
cmuClock_UART0	Universal async receiver/transmitter 0 clock.
cmuClock_UART1	Universal async receiver/transmitter 1 clock.
cmuClock_TIMER0	Timer 0 clock.
cmuClock_TIMER1	Timer 1 clock.
cmuClock_TIMER2	Timer 2 clock.
cmuClock_TIMER3	Timer 3 clock.
cmuClock_ACMP0	Analog comparator 0 clock.
cmuClock_ACMP1	Analog comparator 1 clock.
cmuClock_PRS	Peripheral-reflex system clock.
cmuClock_DAC0	Digital-to-analog converter 0 clock.
cmuClock_GPIO	General-purpose input/output clock.
cmuClock_VCMP	Voltage comparator clock.
cmuClock_ADC0	Analog-to-digital converter 0 clock.
cmuClock_I2C0	I2C 0 clock.
cmuClock_I2C1	I2C 1 clock.
cmuClock_CORE	Core clock.
cmuClock_AES	Advanced encryption standard accelerator clock.
cmuClock_DMA	Direct memory access controller clock.
cmuClock_HFLE	Low-energy clock divided down from HFCORECLK.
cmuClock_LFA	Low-frequency A clock.
cmuClock_RTC	Real time counter clock.
cmuClock_LETIMER0	Low-energy timer 0 clock.
cmuClock_PCNT0	Pulse counter 0 clock.
cmuClock_PCNT1	Pulse counter 1 clock.
cmuClock_PCNT2	Pulse counter 2 clock.
cmuClock_LESENSE	LESENSE clock.
cmuClock_LFB	Low-frequency B clock.
cmuClock_LEUART0	Low-energy universal asynchronous receiver/transmitter 0 clock.
cmuClock_LEUART1	Low-energy universal asynchronous receiver/transmitter 1 clock.

CMU_Osc_TypeDef

> CMU_Osc_TypeDef

Oscillator types.

Enumerator	
cmuOsc_LFXO	Low-frequency crystal oscillator.
cmuOsc_LFRCO	Low-frequency RC oscillator.
cmuOsc_HFXO	High-frequency crystal oscillator.
cmuOsc_HFRCO	High-frequency RC oscillator.
cmuOsc_AUXHFRCO	Auxiliary high-frequency RC oscillator.
cmuOsc_ULFRCO	Ultra low-frequency RC oscillator.

CMU_OscMode_TypeDef

```
CMU_OscMode_TypeDef
```

Oscillator modes.

Enumerator	
cmuOscMode_Crystal	Crystal oscillator.
cmuOscMode_AcCoupled	AC-coupled buffer.
cmuOscMode_External	External digital clock.

CMU_Select_TypeDef

```
CMU_Select_TypeDef
```

Selectable clock sources.

Enumerator	
cmuSelect_Error	Usage error.
cmuSelect_Disabled	Clock selector disabled.
cmuSelect_LFXO	Low-frequency crystal oscillator.
cmuSelect_LFRCO	Low-frequency RC oscillator.
cmuSelect_HFXO	High-frequency crystal oscillator.
cmuSelect_HFRCO	High-frequency RC oscillator.
cmuSelect_HFCLKLE	High-frequency LE clock divided by 2 or 4.
cmuSelect_AUXHFRCO	Auxiliary clock source can be used for debug clock.
cmuSelect_HFSRCCLK	High-frequency source clock.
cmuSelect_HFCLK	Divided HFCLK on Giant for debug clock, undivided on Tiny Gecko and for USBC (not used on Gecko).
cmuSelect_ULFRCO	Ultra low-frequency RC oscillator.

CMU_LFXOBoost_TypeDef

```
CMU_LFXOBoost_TypeDef
```

LFXO Boost values.

Enumerator

cmuLfxoBoost70	
cmuLfxoBoost100	

Typedef Documentation

CMU_ClkDiv_TypeDef

```
typedef uint32_t CMU_ClkDiv_TypeDef
```

Clock divider configuration.

Function Documentation

CMU_AUXHFRCOBandGet

```
CMU_AUXHFRCOBand_TypeDef CMU_AUXHFRCOBandGet (void )
```

Get the AUXHFRCO band in use.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- AUXHFRCO band in use.

CMU_AUXHFRCOBandSet

```
void CMU_AUXHFRCOBandSet (CMU_AUXHFRCOBand_TypeDef band)
```

Set the AUXHFRCO band and the tuning value based on the value in the calibration table made during production.

Parameters

Type	Direction	Argument Name	Description
CMU_AUXHFRCOBand_TypeDef	[in]	band	AUXHFRCO band to activate.

CMU_Calibrate

```
uint32_t CMU_Calibrate (uint32_t HFCycles, CMU_Osc_TypeDef reference)
```

Calibrate the clock.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	HFCycles	The number of HFCLK cycles to run the calibration. Increasing this number increases precision but the calibration will take more time.

Type	Direction	Argument Name	Description
CMU_Osc_TypeDef	[in]	reference	The reference clock used to compare HFCLK.

Run a calibration for HFCLK against a selectable reference clock. See the reference manual, CMU chapter, for more details.

Note

- This function will not return until the calibration measurement is completed.

Returns

- The number of ticks the reference clock after HFCycles ticks on the HF clock.

CMU_CalibrateConfig

> void CMU_CalibrateConfig (uint32_t downCycles, CMU_Osc_TypeDef downSel, CMU_Osc_TypeDef upSel)

Configure the clock calibration.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	downCycles	The number of downSel clock cycles to run the calibration. Increasing this number increases precision but the calibration will take more time.
CMU_Osc_TypeDef	[in]	downSel	The clock, which will be counted down downCycles.
CMU_Osc_TypeDef	[in]	upSel	The reference clock; the number of cycles generated by this clock will be counted and added up and the result can be given with the CMU_CalibrateCountGet() function call.

Configure a calibration for a selectable clock source against another selectable reference clock. See the reference manual, CMU chapter, for more details.

Note

- After configuration, a call to CMU_CalibrateStart() is required and the resulting calibration value can be read out with the CMU_CalibrateCountGet() function call.

CMU_CalibrateCountGet

> uint32_t CMU_CalibrateCountGet (void)

Get the calibration count register.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- If continuous calibration mode is active, calibration busy will almost always be off and only the value needs to be read. In a normal case, this function call is triggered by the CALRDY interrupt flag.

Returns

-

The calibration count, the number of UPSEL clocks in the period of DOWNSEL oscillator clock cycles configured by a previous write operation to CMU->CALCNT.

CMU_ClockEnable

> void CMU_ClockEnable (CMU_Clock_TypeDef clock, bool enable)

Enable/disable a clock.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	The clock to enable/disable. Notice that not all defined clock points have separate enable/disable control. See the CMU overview in the reference manual.
bool	[in]	enable	• true - enable specified clock. • false - disable specified clock.

In general, module clocking is disabled after a reset. If a module clock is disabled, the registers of that module are not accessible and reading from such registers may return undefined values. Writing to registers of clock-disabled modules has no effect. Avoid accessing module registers of a module with a disabled clock.

Note

- If enabling/disabling an LF clock, synchronization into the low-frequency domain is required. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed. See CMU_FreezeEnable() for a suggestion on how to reduce the stalling time in some use cases.

HFCLKLE prescaler is automatically modified when peripherals with clock domain HFBUSCLK is chosen based on the maximum HFLE frequency allowed.

CMU_ClockDivGet

> CMU_ClkDiv_TypeDef CMU_ClockDivGet (CMU_Clock_TypeDef clock)

Get the clock divisor/prescaler.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	A clock point to get the divisor/prescaler for. Notice that not all clock points have a divisor/prescaler. See the CMU overview in the reference manual.

Returns

- The current clock point divisor/prescaler. 1 is returned if `clock` specifies a clock point without a divisor/prescaler.

CMU_ClockDivSet

> void CMU_ClockDivSet (CMU_Clock_TypeDef clock, CMU_ClkDiv_TypeDef div)

Set the clock divisor/prescaler.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	Clock point to set divisor/prescaler for. Notice that not all clock points have a divisor/prescaler. See the CMU overview in the reference manual.
CMU_ClkDiv_TypeDef	[in]	div	The clock divisor to use (<= cmuClkDiv_512).

Note

- If setting an LF clock prescaler, synchronization into the low-frequency domain is required. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed. See CMU_FreezeEnable() for a suggestion on how to reduce the stalling time in some use cases.

HFCLKLE prescaler is automatically modified when peripherals with clock domain HFBUSCLK is chosen based on the maximum HFLE frequency allowed.

CMU_ClockFreqGet

```
uint32_t CMU_ClockFreqGet (CMU_Clock_TypeDef clock)
```

Get the clock frequency for a clock point.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	A clock point to fetch the frequency for.

Returns

- The current frequency in Hz.

CMU_ClockSelectSet

```
void CMU_ClockSelectSet (CMU_Clock_TypeDef clock, CMU_Select_TypeDef ref)
```

Select the reference clock/oscillator used for a clock branch.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	A clock branch to select reference clock for. One of: • cmuClock_HF • cmuClock_LFA • cmuClock_LFB • cmuClock_DBG

Type	Direction	Argument Name	Description
CMU_Select_TypeDef	[in]	ref	A reference selected for clocking. See the reference manual for details about references available for a specific clock branch. • cmuSelect_HFRCO • cmuSelect_LFRCO • cmuSelect_HFXO • cmuSelect_LFXO • cmuSelect_HFCLKLE • cmuSelect_AUXHFRCO • cmuSelect_HFCLK • cmuSelect_ULFRCO

Notice that if a selected reference is not enabled prior to selecting its use, it will be enabled and this function will wait for the selected oscillator to be stable. It will however NOT be disabled if another reference clock is selected later.

This feature is particularly important if selecting a new reference clock for the clock branch clocking the core. Otherwise, the system may halt.

Note

- HFCLKLE prescaler is automatically modified when peripherals with clock domain HFBUSCLK is chosen based on the maximum HFLE frequency allowed.

CMU_ClockSelectGet

> CMU_Select_TypeDef CMU_ClockSelectGet (CMU_Clock_TypeDef clock)

Get the currently selected reference clock used for a clock branch.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	Clock branch to fetch selected ref. clock for. One of: • cmuClock_HF • cmuClock_LFA • cmuClock_LFB • cmuClock_DBG

Returns

- The reference clock used for clocking the selected branch, cmuSelect_Error if invalid `clock` provided.

CMU_LF_ClockPrecisionGet

> uint16_t CMU_LF_ClockPrecisionGet (CMU_Clock_TypeDef clock)

Gets the precision (in PPM) of the specified low frequency clock branch.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	Clock branch.

Returns

- Precision, in PPM, of the specified clock branch.

Note

- This function is only for internal usage.
- The current implementation of this function is used to determine if the clock has a precision <= 500 ppm or not (which is the minimum required for BLE). Future version of this function should provide more accurate precision numbers to allow for further optimizations from the stacks.

CMU_HF_ClockPrecisionGet

> uint16_t CMU_HF_ClockPrecisionGet (CMU_Clock_TypeDef clock)

Gets the precision (in PPM) of the specified high frequency clock branch.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	clock	Clock branch.

Returns

- Precision, in PPM, of the specified clock branch.

Note

- This function is only for internal usage.
- The current implementation of this function is used to determine if the clock has a precision <= 500 ppm or not (which is the minimum required for BLE). Future version of this function should provide more accurate precision numbers to allow for further optimizations from the stacks.

CMU_FreezeEnable

> void CMU_FreezeEnable (bool enable)

CMU low frequency register synchronization freeze control.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• true - enable freeze, modified registers are not propagated to the LF domain • false - disable freeze, modified registers are propagated to the LF domain

Some CMU registers require synchronization into the low-frequency (LF) domain. The freeze feature allows for several such registers to be modified before passing them to the LF domain simultaneously (which takes place when the freeze mode is disabled).

Another use case for this feature is using an API (such as the CMU API) for modifying several bit fields consecutively in the same register. If freeze mode is enabled during this sequence, stalling can be avoided.

Note

-

When enabling freeze mode, this function will wait for all current ongoing CMU synchronization to LF domain to complete (normally synchronization will not be in progress.) However, for this reason, when using freeze mode, modifications of registers requiring LF synchronization should be done within one freeze enable/disable block to avoid unnecessary stalling.

CMU_HFRCOBandGet

```
CMU_HFRCOBand_TypeDef CMU_HFRCOBandGet (void )
```

Get HFRCO band in use.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- HFRCO band in use.

CMU_HFRCOBandSet

```
void CMU_HFRCOBandSet (CMU_HFRCOBand_TypeDef band)
```

Set HFRCO band and the tuning value based on the value in the calibration table made during production.

Parameters

Type	Direction	Argument Name	Description
CMU_HFRCOBand_TypeDef	[in]	band	HFRCO band to activate.

Note

- HFCLKLE prescaler is automatically modified based on the maximum HFLE frequency allowed.

CMU_HFRCOStartupDelayGet

```
uint32_t CMU_HFRCOStartupDelayGet (void )
```

Get the HFRCO startup delay.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

See the reference manual for more details.

Returns

- The startup delay in use.

CMU_HFRCOStartupDelaySet

```
void CMU_HFRCOStartupDelaySet (uint32_t delay)
```

Set the HFRCO startup delay.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	delay	The startup delay to set (<= 31).

See the reference manual for more details.

CMU_HFXOInit

```
void CMU_HFXOInit (const CMU_HFXOInit_TypeDef * hfxoInit)
```

Set HFXO control registers.

Parameters

Type	Direction	Argument Name	Description
const CMU_HFXOInit_TypeDef *	[in]	hfxoInit	HFXO setup parameters.

Note

- HFXO configuration should be obtained from a configuration tool, app note, or crystal data sheet. This function disables the HFXO to ensure a valid state before update.

CMU_LCDClkFDIVGet

```
uint32_t CMU_LCDClkFDIVGet (void )
```

Get the LCD framerate divisor (FDIV) setting.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The LCD framerate divisor.

CMU_LCDClkFDIVSet

```
void CMU_LCDClkFDIVSet (uint32_t div)
```

Set the LCD framerate divisor (FDIV) setting.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	div	The FDIV setting to use.

Note

- The FDIV field (CMU LCDCTRL register) should only be modified while the LCD module is clock disabled (CMU LFACLKEN0.LCD bit is 0). This function will NOT modify FDIV if the LCD module clock is enabled. See CMU_ClockEnable() for disabling/enabling LCD clock.

CMU_LFXOInit

```
void CMU_LFXOInit (const CMU_LFXOInit_TypeDef * lfxoInit)
```

Set LFXO control registers.

Parameters

Type	Direction	Argument Name	Description
const CMU_LFXOInit_TypeDef *	[in]	lfxoInit	LFXO setup parameters.

Note

- LFXO configuration should be obtained from a configuration tool, app note, or crystal data sheet. This function disables the LFXO when necessary to ensure a valid state before update.

CMU_LFXOPrecisionSet

```
void CMU_LFXOPrecisionSet (uint16_t precision)
```

Sets LFXO's crystal precision, in PPM.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	precision	LFXO's crystal precision, in PPM.

Note

- LFXO precision should be obtained from a crystal datasheet.

CMU_LFXOPrecisionGet

```
uint16_t CMU_LFXOPrecisionGet (void )
```

Gets LFXO's crystal precision, in PPM.

Parameters

Type	Direction	Argument Name	Description
void	[in]		LFXO's crystal precision, in PPM.

CMU_HFXOPrecisionSet

```
void CMU_HFXOPrecisionSet (uint16_t precision)
```

Sets HFXO's crystal precision, in PPM.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	precision	HFXO's crystal precision, in PPM.

Note

HFXO precision should be obtained from a crystal datasheet.

CMU_HFXOPrecisionGet

```
uint16_t CMU_HFXOPrecisionGet (void )
```

Gets HFXO's crystal precision, in PPM.

Parameters

Type	Direction	Argument Name	Description
void	[in]		HFXO's crystal precision, in PPM.

CMU_OscillatorEnable

```
void CMU_OscillatorEnable (CMU_Osc_TypeDef osc, bool enable, bool wait)
```

Enable/disable oscillator.

Parameters

Type	Direction	Argument Name	Description
CMU_Osc_TypeDef	[in]	osc	The oscillator to enable/disable.
bool	[in]	enable	• true - enable specified oscillator. • false - disable specified oscillator.
bool	[in]	wait	Only used if `enable` is true. • true - wait for oscillator start-up time to timeout before returning. • false - do not wait for oscillator start-up time to timeout before returning.

Note

- WARNING: When this function is called to disable either cmuOsc_LFXO or cmuOsc_HFXO, the LFXOMODE or HFXOMODE fields of the CMU_CTRL register are reset to the reset value. In other words, if external clock sources are selected in either LFXOMODE or HFXOMODE fields, the configuration will be cleared and needs to be reconfigured if needed later.

CMU_OscillatorTuningGet

```
uint32_t CMU_OscillatorTuningGet (CMU_Osc_TypeDef osc)
```

Get the oscillator frequency tuning setting.

Parameters

Type	Direction	Argument Name	Description
CMU_Osc_TypeDef	[in]	osc	An oscillator to get tuning value for, one of the following: • cmuOsc_LFRCO • cmuOsc_HFRCO • cmuOsc_AUXHFRCO • cmuOsc_HFXO if CMU_HFXOCTRL_PEAKDETSHUNTOPTMODE is defined

Returns

- The oscillator frequency tuning setting in use.

CMU_OscillatorTuningSet

```
void CMU_OscillatorTuningSet (CMU_Osc_TypeDef osc, uint32_t val)
```

Set the oscillator frequency tuning control.

Parameters

Type	Direction	Argument Name	Description
CMU_Osc_TypeDef	[in]	osc	An oscillator to set tuning value for, one of the following: • cmuOsc_LFRCO • cmuOsc_HFRCO • cmuOsc_AUXHFRCO • cmuOsc_HFXO if PEAKDETSHUNTOPTMODE is available. Note that CMD mode is set.
uint32_t	[in]	val	The oscillator frequency tuning setting to use.

Note

- Oscillator tuning is done during production and the tuning value is automatically loaded after reset. Changing the tuning value from the calibrated value is for more advanced use. Certain oscillators also have build-in tuning optimization.

CMU_PCNTClockExternalSet

```
void CMU_PCNTClockExternalSet (unsigned int instance, bool external)
```

Select the PCNTn clock.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	instance	PCNT instance number to set selected clock source for.
bool	[in]	external	Set to true to select the external clock, false to select LFBCLK.

CMU_PCNTClockExternalGet

```
bool CMU_PCNTClockExternalGet (unsigned int instance)
```

Determine if the currently selected PCNTn clock used is external or LFBCLK.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	instance	PCNT instance number to get currently selected clock source for.

Returns

- ◦ true - selected clock is external clock.
  ◦ false - selected clock is LFBCLK.

CMU_UpdateWaitStates

```
void CMU_UpdateWaitStates (uint32_t freq, int vscale)
```

Configure various wait states to switch to a certain frequency and a certain voltage scale.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	freq	The core clock frequency to configure wait-states.
int	[in]	vscale	The voltage scale to configure wait-states. Expected values are 0 or 2, higher number is lower voltage.

This function will set up the necessary flash, bus, and RAM wait states. Updating the wait state configuration must be done before increasing the clock frequency and it must be done after decreasing the clock frequency. Updating the wait state configuration must be done before core voltage is decreased and it must be done after a core voltage is increased.

- 0 = 1.2 V (VSCALE2)
- 2 = 1.0 V (VSCALE0)

CMU_CalibrateCont

```
void CMU_CalibrateCont (bool enable)
```

Configure continuous calibration mode.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, enables continuous calibration, if false disables continuous calibration.

CMU_CalibrateStart

```
void CMU_CalibrateStart (void )
```

Start calibration.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This call is usually invoked after CMU_CalibrateConfig() and possibly CMU_CalibrateCont().

CMU_CalibrateStop

```
void CMU_CalibrateStop (void )
```

Stop the calibration counters.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

CMU_DivToLog2

```
uint32_t CMU_DivToLog2 (CMU_ClkDiv_TypeDef div)
```

Convert divider to logarithmic value.

Parameters

Type	Direction	Argument Name	Description
CMU_ClkDiv_TypeDef	[in]	div	An unscaled divider.

It only works for even numbers equal to 2^n.

Returns

- Logarithm base 2 (binary) value, i.e. exponent as used by fixed 2^n prescalers.

CMU_IntClear

```
void CMU_IntClear (uint32_t flags)
```

Clear one or more pending CMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	CMU interrupt sources to clear.

CMU_IntDisable

```
void CMU_IntDisable (uint32_t flags)
```

Disable one or more CMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	CMU interrupt sources to disable.

CMU_IntEnable

```
void CMU_IntEnable (uint32_t flags)
```

Enable one or more CMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	CMU interrupt sources to enable.

Note

- Depending on use case, a pending interrupt may already be set prior to enabling the interrupt. Consider using CMU_IntClear() prior to enabling if the pending interrupt should be ignored.

CMU_IntGet

```
uint32_t CMU_IntGet (void )
```

Get pending CMU interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- CMU interrupt sources pending.

CMU_IntGetEnabled

```
uint32_t CMU_IntGetEnabled (void )
```

Get enabled and pending CMU interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- This function does not clear event bits.

Returns

- Pending and enabled CMU interrupt sources. The return value is the bitwise AND of
  - the enabled interrupt sources in CMU_IEN and
  - the pending interrupt flags CMU_IF

CMU_IntSet

```
void CMU_IntSet (uint32_t flags)
```

Set one or more pending CMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	CMU interrupt sources to set to pending.

CMU_Lock

```
void CMU_Lock (void )
```

Lock the CMU to protect some of its registers against unintended modification.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

See the reference manual for CMU registers that will be locked.

Note

- If locking the CMU registers, they must be unlocked prior to using any CMU API functions modifying CMU registers protected by the lock.

CMU_Unlock

```
void CMU_Unlock (void )
```

Unlock the CMU so that writing to locked registers again is possible.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

auxClkGet

```
static uint32_t auxClkGet (void )
```

Get the AUX clock frequency.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Used by MSC flash programming and LESENSE, by default also as a debug clock.

Returns

- AUX Frequency in Hz.

dbgClkGet

```
static uint32_t dbgClkGet (void )
```

Get the Debug Trace clock frequency.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Debug Trace frequency in Hz.

flashWaitStateControl

```
static void flashWaitStateControl (uint32_t coreFreq, int vscale)
```

Configure flash access wait states to support the given core clock frequency.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	coreFreq	The core clock frequency to configure flash wait-states.
int	[in]	vscale	Voltage Scale level. Supported levels are 0 and 2 where 0 is the default.

flashWaitStateMax

```
static void flashWaitStateMax (void )
```

Configure flash access wait states to the most conservative setting for this target.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Retain SCBTP (Suppressed Conditional Branch Target Prefetch) setting.

lfClkGet

```
static uint32_t lfClkGet (CMU_Clock_TypeDef lfClkBranch)
```

Get the LFnCLK frequency based on the current configuration.

Parameters

Type	Direction	Argument Name	Description
CMU_Clock_TypeDef	[in]	lfClkBranch	Selected LF branch.

Returns

The LFnCLK frequency in Hz. If no LFnCLK is selected (disabled), 0 is returned.

syncReg

```
void syncReg (uint32_t mask)
```

Wait for an ongoing sync of register(s) to low-frequency domain to complete.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	mask	A bitmask corresponding to SYNCBUSY register defined bits, indicating registers that must complete any ongoing synchronization.

hfperClkSafePrescaler

```
static void hfperClkSafePrescaler (void )
```

Set HFPER clock tree prescalers to safe values.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function applies to EFM32GG11B. There are 3 HFPER clock trees with these frequency limits: HFPERCLK (A-tree): 20MHz in VSCALE0 mode, 50MHz in VSCALE2 mode. HFPERBCLK (B-tree): 20MHz in VSCALE0 mode, 72MHz in VSCALE2 mode. HFPERCCLK (C-tree): 20MHz in VSCALE0 mode, 50MHz in VSCALE2 mode.

hfperClkOptimizedPrescaler

```
static void hfperClkOptimizedPrescaler (void )
```

Set HFPER clock tree prescalers to give highest possible clock node frequency while still beeing within spec.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function applies to EFM32GG11B. There are 3 HFPER clock trees with these frequency limits: HFPERCLK (A-tree): 20MHz in VSCALE0 mode, 50MHz in VSCALE2 mode. HFPERBCLK (B-tree): 20MHz in VSCALE0 mode, 72MHz in VSCALE2 mode. HFPERCCLK (C-tree): 20MHz in VSCALE0 mode, 50MHz in VSCALE2 mode.

Macro Definition Documentation

CMU_CLOCK_SELECT_SET

```
#define CMU_CLOCK_SELECT_SET
```

Value:

```
(clock, sel)
```

Macro to set clock sources in the clock tree.

cmuClkDiv_1

```
#define cmuClkDiv_1
```

Value:

```
1
```

Clock divisors.

These values are valid for prescalers. Divide clock by 1.

cmuClkDiv_2

```
#define cmuClkDiv_2
```

Value:

```
2
```

Divide clock by 2.

cmuClkDiv_4

```
#define cmuClkDiv_4
```

Value:

```
4
```

Divide clock by 4.

cmuClkDiv_8

```
#define cmuClkDiv_8
```

Value:

```
8
```

Divide clock by 8.

cmuClkDiv_16

```
#define cmuClkDiv_16
```

Value:

```
16
```

Divide clock by 16.

cmuClkDiv_32

```
#define cmuClkDiv_32
```

Value:

```
32
```

Divide clock by 32.

cmuClkDiv_64

```
#define cmuClkDiv_64
```

Value:

```
64
```

Divide clock by 64.

cmuClkDiv_128

```
#define cmuClkDiv_128
```

Value:

```
128
```

Divide clock by 128.

cmuClkDiv_256

```
#define cmuClkDiv_256
```

Value:

```
256
```

Divide clock by 256.

cmuClkDiv_512

```
#define cmuClkDiv_512
```

Value:

```
512
```

Divide clock by 512.

cmuClkDiv_1024

```
#define cmuClkDiv_1024
```

Value:

```
1024
```

Divide clock by 1024.

cmuClkDiv_2048

```
#define cmuClkDiv_2048
```

Value:

```
2048
```

Divide clock by 2048.

cmuClkDiv_4096

```
#define cmuClkDiv_4096
```

Value:

```
4096
```

Divide clock by 4096.

cmuClkDiv_8192

```
#define cmuClkDiv_8192
```

Value:

```
8192
```

Divide clock by 8192.

cmuClkDiv_16384

```
#define cmuClkDiv_16384
```

Value:

```
16384
```

Divide clock by 16384.

cmuClkDiv_32768

#define cmuClkDiv_32768

Value:

```
32768
```

Divide clock by 32768.

CMU_LFXOINIT_DEFAULT

#define CMU_LFXOINIT_DEFAULT

Value:

```
{                                \
  cmuLfxoBoost70,                \
  _CMU_CTRL_LFXOTIMEOUT_DEFAULT, \
  cmuOscMode_Crystal,            \
}
```

Default LFXO initialization values.

CMU_LFXOINIT_EXTERNAL_CLOCK

#define CMU_LFXOINIT_EXTERNAL_CLOCK

Value:

```
{                                \
  cmuLfxoBoost70,                \
  _CMU_CTRL_LFXOTIMEOUT_8CYCLES, \
  cmuOscMode_External,           \
}
```

Default LFXO initialization for external clock.

CMU_HFXOINIT_DEFAULT

#define CMU_HFXOINIT_DEFAULT

Value:

```
{                                                    \
  _CMU_CTRL_HFXOBOOST_DEFAULT, /* 100% HFXO boost */   \
  _CMU_CTRL_HFXOTIMEOUT_DEFAULT, /* 16 K startup delay */ \
  false,                       /* Disable glitch detector */ \
  cmuOscMode_Crystal,          /* Crystal oscillator */ \
}
```

Default HFXO initialization values for Platform 1 devices.

CMU_HFXOINIT_EXTERNAL_CLOCK

#define CMU_HFXOINIT_EXTERNAL_CLOCK

Value:

```
0   {                                                                   \
0     0,                            /* Minimal HFXO boost, 50% */       \
0     _CMU_CTRL_HFXOTIMEOUT_8CYCLES, /* Minimal startup delay, 8 cycles */ \
0     false,                        /* Disable glitch detector */       \
0     cmuOscMode_External,          /* External digital clock */        \
0   }
```

Default HFXO initialization for external clock.

CMU_LFXOInit_TypeDef

LFXO initialization structure.

Initialization values should be obtained from a configuration tool, application note or crystal data sheet.

Public Attributes

CMU_LFXOBoost_TypeDef	**boost** LFXO boost.
uint8_t	**timeout** Startup delay.
CMU_OscMode_TypeDef	**mode** Oscillator mode.

Public Attribute Documentation

boost

```
CMU_LFXOBoost_TypeDef CMU_LFXOInit_TypeDef::boost
```

LFXO boost.

timeout

```
uint8_t CMU_LFXOInit_TypeDef::timeout
```

Startup delay.

mode

```
CMU_OscMode_TypeDef CMU_LFXOInit_TypeDef::mode
```

Oscillator mode.

CMU_HFXOInit_TypeDef

HFXO initialization structure.

Initialization values should be obtained from a configuration tool, application note or crystal data sheet.

Public Attributes

uint8_t	**boost** HFXO Boost, 0=50% 1=70%, 2=80%, 3=100%.
uint8_t	**timeout** Startup delay.
bool	**glitchDetector** Enable/disable glitch detector.
CMU_OscMode_TypeDef	**mode** Oscillator mode.

Public Attribute Documentation

boost

```
uint8_t CMU_HFXOInit_TypeDef::boost
```

HFXO Boost, 0=50% 1=70%, 2=80%, 3=100%.

timeout

```
uint8_t CMU_HFXOInit_TypeDef::timeout
```

Startup delay.

glitchDetector

```
bool CMU_HFXOInit_TypeDef::glitchDetector
```

Enable/disable glitch detector.

mode

```
CMU_OscMode_TypeDef CMU_HFXOInit_TypeDef::mode
```

Oscillator mode.

CORE - Core Interrupt

CORE - Core Interrupt

Introduction

CORE interrupt API provides a simple and safe means to disable and enable interrupts to protect sections of code.

This is often referred to as "critical sections". This module provides support for three types of critical sections, each with different interrupt blocking capabilities.

- CRITICAL section: Inside a critical section, all interrupts are disabled (except for fault handlers). The PRIMASK register is always used for interrupt disable/enable.
- ATOMIC section: This type of section is configurable and the default method is to use PRIMASK. With BASEPRI configuration, interrupts with priority equal to or lower than a given configurable level are disabled. The interrupt disable priority level is defined at compile time. The BASEPRI register is not available for all architectures.
- NVIC mask section: Disable NVIC (external interrupts) on an individual manner.

em_core also has an API for manipulating RAM-based interrupt vector tables.

Compile-time Configuration

The following #defines are used to configure em_core:

```
// The interrupt priority level used inside ATOMIC sections.
#define CORE_ATOMIC_BASE_PRIORITY_LEVEL    3

// A method used for interrupt disable/enable within ATOMIC sections.
#define CORE_ATOMIC_METHOD                 CORE_ATOMIC_METHOD_PRIMASK
```

If the default values do not support your needs, they can be overridden by supplying -D compiler flags on the compiler command line or by collecting all macro redefinitions in a file named emlib_config.h and then supplying -DEMLIB_USER_CONFIG on a compiler command line.

Note

- The default emlib configuration for ATOMIC section interrupt disable method is using PRIMASK, i.e., ATOMIC sections are implemented as CRITICAL sections.
- Due to architectural limitations Cortex-M0+ devices do not support ATOMIC type critical sections using the BASEPRI register. On M0+ devices ATOMIC section helper macros are available but they are implemented as CRITICAL sections using PRIMASK register.

Macro API

The primary em_core API is the macro API. Macro API will map to correct CORE functions according to the selected CORE_ATOMIC_METHOD and similar configurations (the full CORE API is of course also available). The most useful macros are as follows:

CORE_DECLARE_IRQ_STATE
CORE_ENTER_ATOMIC()

@n Used together to implement an ATOMIC section.

```
{
  CORE_DECLARE_IRQ_STATE;           // Storage for saving IRQ state prior to
                                    // atomic section entry.

  CORE_ENTER_ATOMIC();              // Enter atomic section.

  ...
  ... your code goes here ...
  ...

  CORE_EXIT_ATOMIC();               // Exit atomic section, IRQ state is restored.
}
```

@n A concatenation of all three macros above.

```
{
  CORE_ATOMIC_SECTION(
    ...
    ... your code goes here ...
    ...
  )
}
```

CORE_DECLARE_IRQ_STATE
CORE_ENTER_CRITICAL()
CORE_EXIT_CRITICAL()
@n These macros implement CRITICAL sections in a similar fashion as described above for ATOMIC sections.

CORE_DECLARE_NVIC_STATE
CORE_ENTER_NVIC()
CORE_EXIT_NVIC()
@n These macros implement NVIC mask sections in a similar fashion as described above for ATOMIC sections. See Examples for an example.

Refer to Macros or Macro Definition Documentation below for a full list of macros.

API reimplementation

Most of the functions in the API are implemented as weak functions. This means that it is easy to reimplement when special needs arise. Shown below is a reimplementation of CRITICAL sections suitable if FreeRTOS OS is used:

```
CORE_irqState_t CORE_EnterCritical(void)
{
  vPortEnterCritical();
  return 0;
}

void CORE_ExitCritical(CORE_irqState_t irqState)
{
  (void)irqState;
  vPortExitCritical();
}
```

Also note that CORE_Enter/ExitCritical() are not implemented as inline functions. As a result, reimplementations will be possible even when original implementations are inside a linked library.

Some RTOSes must be notified on interrupt handler entry and exit. Macros CORE_INTERRUPT_ENTRY() and CORE_INTERRUPT_EXIT() are suitable placeholders for inserting such code. Insert these macros in all your interrupt handlers and then override the default macro implementations. This is an example if uC/OS is used:

```
// In emlib_config.h:

#define CORE_INTERRUPT_ENTRY()   OSIntEnter()
#define CORE_INTERRUPT_EXIT()    OSIntExit()
```

Interrupt vector tables

When using RAM based interrupt vector tables it is the user's responsibility to allocate the table space correctly. The tables must be aligned as specified in the CPU reference manual.

@n Initialize a RAM based vector table by copying table entries from a source vector table to a target table. VTOR is set to the address of the target vector table.

CORE_GetNvicRamTableHandler()
@n Use these functions to get or set the interrupt handler for a specific IRQn. They both use the interrupt vector table defined by the current VTOR register value.

Maximum Interrupt Disabled Time

The maximum time spent (in cycles) in critical and atomic sections can be measured for performance and interrupt latency analysis. To enable the timings, use the SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING configuration option. When enabled, the functions
CORE_get_max_time_critical_section()
CORE_get_max_time_atomic_section()
can be used to get the max timings since startup.

Examples

Implement an NVIC critical section:

```
{
  CORE_DECLARE_NVIC_ZEROMASK(mask); // A zero initialized NVIC disable mask

  // Set mask bits for IRQs to block in the NVIC critical section.
  // In many cases, you can create the disable mask once upon application
  // startup and use the mask globally throughout the application lifetime.
  CORE_NvicMaskSetIRQ(LEUART0_IRQn, &mask);
  CORE_NvicMaskSetIRQ(VCMP_IRQn,    &mask);

  // Enter NVIC critical section with the disable mask
  CORE_NVIC_SECTION(&mask,
    ...
    ... your code goes here ...
    ...
  )
}
```

Porting from em_int

Existing code using INT_Enable() and INT_Disable() must be ported to the em_core API. While em_int used, a global counter to store the interrupt state, em_core uses a local variable. Any usage of INT_Disable(), therefore, needs to be replaced with a declaration of the interrupt state variable before entering the critical section.

Since the state variable is in local scope, the critical section exit needs to occur within the scope of the variable. If multiple nested critical sections are used, each needs to have its own state variable in its own scope.

In many cases, completely disabling all interrupts using CRITICAL sections might be more heavy-handed than needed. When porting, consider whether other types of sections, such as ATOMIC or NVIC mask, can be used to only disable a subset of the interrupts.

Replacing em_int calls with em_core function calls:

```
void func(void)
{
  // INT_Disable();
  CORE_DECLARE_IRQ_STATE;
  CORE_ENTER_ATOMIC();
    .
    .
    .
  // INT_Enable();
  CORE_EXIT_ATOMIC();
}
```

Modules

dwt_cycle_counter_handle_t

CORE_nvicMask_t

Typedefs

typedef uint32_t — CORE_irqState_t
Storage for PRIMASK or BASEPRI value.

typedef uint32_t — CORE_irqState_t
Storage for PRIMASK or BASEPRI value.

Functions

void — CORE_CriticalDisableIrq(void)
Disable interrupts.

void — CORE_CriticalEnableIrq(void)
Enable interrupts.

CORE_irqState_t — CORE_EnterCritical(void)
Enter a CRITICAL section.

void — CORE_ExitCritical(CORE_irqState_t irqState)
Exit a CRITICAL section.

void — CORE_YieldCritical(void)
Brief interrupt enable/disable sequence to allow handling of pending interrupts.

void — CORE_AtomicDisableIrq(void)
Disable interrupts.

void — CORE_AtomicEnableIrq(void)
Enable interrupts.

CORE_irqState_t	CORE_EnterAtomic(void) Enter an ATOMIC section.
void	CORE_ExitAtomic(CORE_irqState_t irqState) Exit an ATOMIC section.
void	CORE_YieldAtomic(void) Brief interrupt enable/disable sequence to allow handling of pending interrupts.
void	CORE_EnterNvicMask(CORE_nvicMask_t *nvicState, const CORE_nvicMask_t *disable) Enter a NVIC mask section.
void	CORE_NvicDisableMask(const CORE_nvicMask_t *disable) Disable NVIC interrupts.
void	CORE_NvicEnableMask(const CORE_nvicMask_t *enable) Set current NVIC interrupt enable mask.
void	CORE_YieldNvicMask(const CORE_nvicMask_t *enable) Brief NVIC interrupt enable/disable sequence to allow handling of pending interrupts.
void	CORE_NvicMaskSetIRQ(IRQn_Type irqN, CORE_nvicMask_t *mask) Utility function to set an IRQn bit in a NVIC enable/disable mask.
void	CORE_NvicMaskClearIRQ(IRQn_Type irqN, CORE_nvicMask_t *mask) Utility function to clear an IRQn bit in a NVIC enable/disable mask.
bool	CORE_InIrqContext(void) Check whether the current CPU operation mode is handler mode.
bool	CORE_IrqIsBlocked(IRQn_Type irqN) Check if a specific interrupt is disabled or blocked.
bool	CORE_IrqIsDisabled(void) Check if interrupts are disabled.
void	CORE_GetNvicEnabledMask(CORE_nvicMask_t *mask) Get the current NVIC enable mask state.
bool	CORE_GetNvicMaskDisableState(const CORE_nvicMask_t *mask) Get NVIC disable state for a given mask.
bool	CORE_NvicIRQDisabled(IRQn_Type irqN) Check if an NVIC interrupt is disabled.
void *	CORE_GetNvicRamTableHandler(IRQn_Type irqN) Utility function to get the handler for a specific interrupt.
void	CORE_SetNvicRamTableHandler(IRQn_Type irqN, void *handler) Utility function to set the handler for a specific interrupt.
void	CORE_InitNvicVectorTable(uint32_t *sourceTable, uint32_t sourceSize, uint32_t *targetTable, uint32_t targetSize, void *defaultHandler, bool overwriteActive) Initialize an interrupt vector table by copying table entries from a source to a target table.
void	cycle_counter_start(dwt_cycle_counter_handle_t *handle) Start a recording.
void	cycle_counter_stop(dwt_cycle_counter_handle_t *handle) Stop a recording.
uint32_t	CORE_get_max_time_critical_section(void) Returns the max time spent in critical section.
uint32_t	CORE_get_max_time_atomic_section(void) Returns the max time spent in atomic section.

void CORE_clear_max_time_critical_section(void)
Clears the max time spent in atomic section.

void CORE_clear_max_time_atomic_section(void)
Clears the max time spent in atomic section.

Macros

#define CORE_INTERRUPT_ENTRY ()
Placeholder for optional interrupt handler entry code.

#define CORE_INTERRUPT_EXIT ()
Placeholder for optional interrupt handler exit code.

#define CORE_ATOMIC_METHOD_PRIMASK 0
Use PRIMASK register to disable interrupts in ATOMIC sections.

#define CORE_ATOMIC_METHOD_BASEPRI 1
Use BASEPRI register to disable interrupts in ATOMIC sections.

#define CORE_ATOMIC_BASE_PRIORITY_LEVEL 3
The interrupt priority level disabled within ATOMIC regions.

#define CORE_DECLARE_IRQ_STATE CORE_irqState_t irqState
Allocate storage for PRIMASK or BASEPRI value for use by CORE_ENTER/EXIT_ATOMIC() and CORE_ENTER/EXIT_CRITICAL() macros.

#define CORE_CRITICAL_IRQ_DISABLE ()
CRITICAL style interrupt disable.

#define CORE_CRITICAL_IRQ_ENABLE ()
CRITICAL style interrupt enable.

#define CORE_CRITICAL_SECTION (yourcode)
Convenience macro for implementing a CRITICAL section.

#define CORE_ENTER_CRITICAL ()
Enter CRITICAL section.

#define CORE_EXIT_CRITICAL ()
Exit CRITICAL section.

#define CORE_YIELD_CRITICAL ()
CRITICAL style yield.

#define CORE_ATOMIC_IRQ_DISABLE ()
ATOMIC style interrupt disable.

#define CORE_ATOMIC_IRQ_ENABLE ()
ATOMIC style interrupt enable.

#define CORE_ATOMIC_SECTION (yourcode)
Convenience macro for implementing an ATOMIC section.

#define CORE_ENTER_ATOMIC ()
Enter ATOMIC section.

#define CORE_EXIT_ATOMIC ()
Exit ATOMIC section.

#define CORE_YIELD_ATOMIC ()
ATOMIC style yield.

#define CORE_IRQ_DISABLED ()
Check if IRQ is disabled.

#define CORE_IN_IRQ_CONTEXT ()
Check if inside an IRQ handler.

#define CORE_DECLARE_IRQ_STATE
Allocate storage for PRIMASK or BASEPRI value for use by CORE_ENTER/EXIT_ATOMIC() and CORE_ENTER/EXIT_CRITICAL() macros.

#define CORE_CRITICAL_IRQ_DISABLE ()
CRITICAL style interrupt disable.

#define CORE_CRITICAL_IRQ_ENABLE ()
CRITICAL style interrupt enable.

#define CORE_CRITICAL_SECTION (yourcode)
Convenience macro for implementing a CRITICAL section.

#define CORE_ENTER_CRITICAL ()
Enter CRITICAL section.

#define CORE_EXIT_CRITICAL ()
Exit CRITICAL section.

#define CORE_YIELD_CRITICAL ()
CRITICAL style yield.

#define CORE_ATOMIC_IRQ_DISABLE ()
ATOMIC style interrupt disable.

#define CORE_ATOMIC_IRQ_ENABLE ()
ATOMIC style interrupt enable.

#define CORE_ATOMIC_SECTION (yourcode)
Convenience macro for implementing an ATOMIC section.

#define CORE_ENTER_ATOMIC ()
Enter ATOMIC section.

#define CORE_EXIT_ATOMIC ()
Exit ATOMIC section.

#define CORE_YIELD_ATOMIC ()
ATOMIC style yield.

#define CORE_NVIC_REG_WORDS ((EXT_IRQ_COUNT + 31) / 32)
Number of words in a NVIC mask set.

#define CORE_DEFAULT_VECTOR_TABLE_ENTRIES (EXT_IRQ_COUNT + 16)
Number of entries in a default interrupt vector table.

#define CORE_INTERRUPT_HIGHEST_PRIORITY 0
Highest priority for core interrupt.

#define CORE_INTERRUPT_DEFAULT_PRIORITY 5
Default priority for core interrupt.

#define CORE_INTERRUPT_LOWEST_PRIORITY 7
Lowest priority for core interrupt.

#define CORE_ATOMIC_METHOD_DEFAULT CORE_ATOMIC_METHOD_BASEPRI
Default method to disable interrupts in ATOMIC sections.

#define CORE_ATOMIC_METHOD CORE_ATOMIC_METHOD_PRIMASK
Specify which method to use when implementing ATOMIC sections.

#define CORE_DECLARE_NVIC_STATE CORE_nvicMask_t nvicState
Allocate storage for NVIC interrupt masks for use by CORE_ENTER/EXIT_NVIC() macros.

#define CORE_DECLARE_NVIC_MASK (x)
Allocate storage for NVIC interrupt masks.

#define CORE_DECLARE_NVIC_ZEROMASK (x)
Allocate storage for and zero initialize NVIC interrupt mask.

#define CORE_NVIC_DISABLE (mask)
NVIC mask style interrupt disable.

#define CORE_NVIC_ENABLE (mask)
NVIC mask style interrupt enable.

#define CORE_NVIC_SECTION (mask, yourcode)
Convenience macro for implementing a NVIC mask section.

#define CORE_ENTER_NVIC (disable)
Enter NVIC mask section.

#define CORE_EXIT_NVIC ()
Exit NVIC mask section.

#define CORE_YIELD_NVIC (enable)
NVIC maks style yield.

Typedef Documentation

CORE_irqState_t

```
typedef uint32_t CORE_irqState_t
```

Storage for PRIMASK or BASEPRI value.

CORE_irqState_t

```
typedef uint32_t CORE_irqState_t
```

Storage for PRIMASK or BASEPRI value.

Function Documentation

CORE_CriticalDisableIrq

```
void CORE_CriticalDisableIrq (void )
```

Disable interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Disable all interrupts by setting PRIMASK. (Fault exception handlers will still be enabled).

CORE_CriticalEnableIrq

```
void CORE_CriticalEnableIrq (void )
```

Enable interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Enable interrupts by clearing PRIMASK.

CORE_EnterCritical

```
CORE_irqState_t CORE_EnterCritical (void )
```

Enter a CRITICAL section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

When a CRITICAL section is entered, all interrupts (except fault handlers) are disabled.

Returns

- The value of PRIMASK register prior to the CRITICAL section entry.

CORE_ExitCritical

```
void CORE_ExitCritical (CORE_irqState_t irqState)
```

Exit a CRITICAL section.

Parameters

Type	Direction	Argument Name	Description
CORE_irqState_t	[in]	irqState	The interrupt priority blocking level to restore to PRIMASK when exiting the CRITICAL section. This value is usually the one returned by a prior call to CORE_EnterCritical().

CORE_YieldCritical

```
void CORE_YieldCritical (void )
```

Brief interrupt enable/disable sequence to allow handling of pending interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Usually used within a CRITICAL section.

CORE_AtomicDisableIrq

```
void CORE_AtomicDisableIrq (void )
```

Disable interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Disable interrupts with a priority lower or equal to CORE_ATOMIC_BASE_PRIORITY_LEVEL. Sets core BASEPRI register to CORE_ATOMIC_BASE_PRIORITY_LEVEL.

Note

- If CORE_ATOMIC_METHOD is CORE_ATOMIC_METHOD_PRIMASK, this function is identical to CORE_CriticalDisableIrq().

CORE_AtomicEnableIrq

```
void CORE_AtomicEnableIrq (void )
```

Enable interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Enable interrupts by setting core BASEPRI register to 0.

Note

- If CORE_ATOMIC_METHOD is CORE_ATOMIC_METHOD_BASEPRI and PRIMASK is set (CPU is inside a CRITICAL section), interrupts will still be disabled after calling this function.
- If CORE_ATOMIC_METHOD is CORE_ATOMIC_METHOD_PRIMASK, this function is identical to CORE_CriticalEnableIrq().

CORE_EnterAtomic

```
CORE_irqState_t CORE_EnterAtomic (void )
```

Enter an ATOMIC section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

When an ATOMIC section is entered, interrupts with priority lower or equal to CORE_ATOMIC_BASE_PRIORITY_LEVEL are disabled.

Note

- If CORE_ATOMIC_METHOD is CORE_ATOMIC_METHOD_PRIMASK, this function is identical to CORE_EnterCritical().

Returns

- The value of BASEPRI register prior to ATOMIC section entry.

CORE_ExitAtomic

> void CORE_ExitAtomic (CORE_irqState_t irqState)

Exit an ATOMIC section.

Parameters

Type	Direction	Argument Name	Description
CORE_irqState_t	[in]	irqState	The interrupt priority blocking level to restore to BASEPRI when exiting the ATOMIC section. This value is usually the one returned by a prior call to CORE_EnterAtomic().

Note

- If CORE_ATOMIC_METHOD is set to CORE_ATOMIC_METHOD_PRIMASK, this function is identical to CORE_ExitCritical().

CORE_YieldAtomic

> void CORE_YieldAtomic (void)

Brief interrupt enable/disable sequence to allow handling of pending interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Usully used within an ATOMIC section.
- If CORE_ATOMIC_METHOD is CORE_ATOMIC_METHOD_PRIMASK, this function is identical to CORE_YieldCritical().

CORE_EnterNvicMask

> void CORE_EnterNvicMask (CORE_nvicMask_t * nvicState, const CORE_nvicMask_t * disable)

Enter a NVIC mask section.

Parameters

Type	Direction	Argument Name	Description
CORE_nvicMask_t *	[out]	nvicState	Return NVIC interrupts enable mask prior to section entry.
const CORE_nvicMask_t *	[in]	disable	A mask specifying which NVIC interrupts to disable within the section.

When a NVIC mask section is entered, specified NVIC interrupts are disabled.

CORE_NvicDisableMask

> void CORE_NvicDisableMask (const CORE_nvicMask_t * disable)

Disable NVIC interrupts.

Parameters

Type	Direction	Argument Name	Description
const CORE_nvicMask_t *	[in]	disable	A mask specifying which NVIC interrupts to disable.

CORE_NvicEnableMask

```
void CORE_NvicEnableMask (const CORE_nvicMask_t * enable)
```

Set current NVIC interrupt enable mask.

Parameters

Type	Direction	Argument Name	Description
const CORE_nvicMask_t *	[out]	enable	A mask specifying which NVIC interrupts are currently enabled.

CORE_YieldNvicMask

```
void CORE_YieldNvicMask (const CORE_nvicMask_t * enable)
```

Brief NVIC interrupt enable/disable sequence to allow handling of pending interrupts.

Parameters

Type	Direction	Argument Name	Description
const CORE_nvicMask_t *	[in]	enable	A mask specifying which NVIC interrupts to briefly enable.

Note

- Usually used within an NVIC mask section.

CORE_NvicMaskSetIRQ

```
void CORE_NvicMaskSetIRQ (IRQn_Type irqN, CORE_nvicMask_t * mask)
```

Utility function to set an IRQn bit in a NVIC enable/disable mask.

Parameters

Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt.
CORE_nvicMask_t *	[inout]	mask	The mask to set the interrupt bit in.

CORE_NvicMaskClearIRQ

```
void CORE_NvicMaskClearIRQ (IRQn_Type irqN, CORE_nvicMask_t * mask)
```

Utility function to clear an IRQn bit in a NVIC enable/disable mask.

Parameters

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt.
CORE_nvicMask_t *	[inout]	mask	The mask to clear the interrupt bit in.

CORE_InIrqContext

```
bool CORE_InIrqContext (void )
```

Check whether the current CPU operation mode is handler mode.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if the CPU is in handler mode (currently executing an interrupt handler).
  False if the CPU is in thread mode.

CORE_IrqIsBlocked

```
bool CORE_IrqIsBlocked (IRQn_Type irqN)
```

Check if a specific interrupt is disabled or blocked.

Parameters

Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt to check.

Returns

- True if the interrupt is disabled or blocked.

CORE_IrqIsDisabled

```
bool CORE_IrqIsDisabled (void )
```

Check if interrupts are disabled.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if interrupts are disabled.

CORE_GetNvicEnabledMask

void CORE_GetNvicEnabledMask (CORE_nvicMask_t * mask)

Get the current NVIC enable mask state.

Parameters

Type	Direction	Argument Name	Description
CORE_nvicMask_t *	[out]	mask	The current NVIC enable mask.

CORE_GetNvicMaskDisableState

bool CORE_GetNvicMaskDisableState (const CORE_nvicMask_t * mask)

Get NVIC disable state for a given mask.

Parameters

Type	Direction	Argument Name	Description
const CORE_nvicMask_t *	[in]	mask	An NVIC mask to check.

Returns

- True if all NVIC interrupt mask bits are clear.

CORE_NvicIRQDisabled

bool CORE_NvicIRQDisabled (IRQn_Type irqN)

Check if an NVIC interrupt is disabled.

Parameters

Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt to check.

Returns

- True if the interrupt is disabled.

CORE_GetNvicRamTableHandler

void * CORE_GetNvicRamTableHandler (IRQn_Type irqN)

Utility function to get the handler for a specific interrupt.

Parameters

Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt.

Returns

- The handler address.

Note

- Uses the interrupt vector table defined by the current VTOR register value.

CORE_SetNvicRamTableHandler

```
void CORE_SetNvicRamTableHandler (IRQn_Type irqN, void * handler)
```

Utility function to set the handler for a specific interrupt.

Parameters

Type	Direction	Argument Name	Description
IRQn_Type	[in]	irqN	The IRQn_Type enumerator for the interrupt.
void *	[in]	handler	The handler address.

Note

- Uses the interrupt vector table defined by the current VTOR register value.

CORE_InitNvicVectorTable

```
void CORE_InitNvicVectorTable (uint32_t * sourceTable, uint32_t sourceSize, uint32_t * targetTable, uint32_t targetSize, void * defaultHandler, bool overwriteActive)
```

Initialize an interrupt vector table by copying table entries from a source to a target table.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[in]	sourceTable	The address of the source vector table.
uint32_t	[in]	sourceSize	A number of entries in the source vector table.
uint32_t *	[in]	targetTable	The address of the target (new) vector table.
uint32_t	[in]	targetSize	A number of entries in the target vector table.
void *	[in]	defaultHandler	An address of the interrupt handler used for target entries for which where there is no corresponding source entry (i.e., the target table is larger than the source table).
bool	[in]	overwriteActive	When true, a target table entry is always overwritten with the corresponding source entry. If false, a target table entry is only overwritten if it is zero. This makes it possible for an application to partly initialize a target table before passing it to this function.

Note

- This function will set a new VTOR register value.

cycle_counter_start

```
static void cycle_counter_start (dwt_cycle_counter_handle_t * handle)
```

Start a recording.

Parameters

Type	Direction	Argument Name	Description
dwt_cycle_counter_handle_t *	[in]	handle	Pointer to initialized counter handle.

Note

- SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

cycle_counter_stop

```
static void cycle_counter_stop (dwt_cycle_counter_handle_t * handle)
```

Stop a recording.

Parameters

Type	Direction	Argument Name	Description
dwt_cycle_counter_handle_t *	[in]	handle	Pointer to initialized counter handle.

Note

- SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

CORE_get_max_time_critical_section

```
uint32_t CORE_get_max_time_critical_section (void )
```

Returns the max time spent in critical section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The max time spent in critical section.

Note

- SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

CORE_get_max_time_atomic_section

```
uint32_t CORE_get_max_time_atomic_section (void )
```

Returns the max time spent in atomic section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The max time spent in atomic section.

Note

SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

CORE_clear_max_time_critical_section

```
void CORE_clear_max_time_critical_section (void )
```

Clears the max time spent in atomic section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

CORE_clear_max_time_atomic_section

```
void CORE_clear_max_time_atomic_section (void )
```

Clears the max time spent in atomic section.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- SL_EMLIB_CORE_ENABLE_INTERRUPT_DISABLED_TIMING must be enabled.

Macro Definition Documentation

CORE_INTERRUPT_ENTRY

```
#define CORE_INTERRUPT_ENTRY
```

Placeholder for optional interrupt handler entry code.

This might be needed when working with an RTOS.

CORE_INTERRUPT_EXIT

```
#define CORE_INTERRUPT_EXIT
```

Placeholder for optional interrupt handler exit code.

This might be needed when working with an RTOS.

CORE_ATOMIC_METHOD_PRIMASK

```
#define CORE_ATOMIC_METHOD_PRIMASK
```

Value:

```
0
```

Use PRIMASK register to disable interrupts in ATOMIC sections.

CORE_ATOMIC_METHOD_BASEPRI

```
#define CORE_ATOMIC_METHOD_BASEPRI
```

Value:

```
1
```

Use BASEPRI register to disable interrupts in ATOMIC sections.

CORE_ATOMIC_BASE_PRIORITY_LEVEL

```
#define CORE_ATOMIC_BASE_PRIORITY_LEVEL
```

Value:

```
3
```

The interrupt priority level disabled within ATOMIC regions.

Interrupts with priority level equal to or lower than this definition will be disabled within ATOMIC regions.

CORE_DECLARE_IRQ_STATE

```
#define CORE_DECLARE_IRQ_STATE
```

Value:

```
CORE_irqState_t irqState
```

Allocate storage for PRIMASK or BASEPRI value for use by CORE_ENTER/EXIT_ATOMIC() and CORE_ENTER/EXIT_CRITICAL() macros.

CORE_CRITICAL_IRQ_DISABLE

```
#define CORE_CRITICAL_IRQ_DISABLE
```

Value:

```
()
```

CRITICAL style interrupt disable.

CORE_CRITICAL_IRQ_ENABLE

```
#define CORE_CRITICAL_IRQ_ENABLE
```

Value:

```
()
```

CRITICAL style interrupt enable.

CORE_CRITICAL_SECTION

```
#define CORE_CRITICAL_SECTION
```

Value:

```
{                                   \
  CORE_DECLARE_IRQ_STATE;           \
  CORE_ENTER_CRITICAL();            \
  {                                 \
    yourcode                        \
  }                                 \
  CORE_EXIT_CRITICAL();             \
}
```

Convenience macro for implementing a CRITICAL section.

CORE_ENTER_CRITICAL

```
#define CORE_ENTER_CRITICAL
```

Value:

```
()
```

Enter CRITICAL section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_EXIT_CRITICAL

```
#define CORE_EXIT_CRITICAL
```

Value:

```
()
```

Exit CRITICAL section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_YIELD_CRITICAL

```
#define CORE_YIELD_CRITICAL
```

Value:

```
()
```

CRITICAL style yield.

CORE_ATOMIC_IRQ_DISABLE

```
#define CORE_ATOMIC_IRQ_DISABLE
```

Value:

```
()
```

ATOMIC style interrupt disable.

CORE_ATOMIC_IRQ_ENABLE

```
#define CORE_ATOMIC_IRQ_ENABLE
```

Value:

```
()
```

ATOMIC style interrupt enable.

CORE_ATOMIC_SECTION

```
#define CORE_ATOMIC_SECTION
```

Value:

```
{                                   \
  CORE_DECLARE_IRQ_STATE;           \
  CORE_ENTER_ATOMIC();              \
  {                                 \
    yourcode                        \
  }                                 \
  CORE_EXIT_ATOMIC();               \
}
```

Convenience macro for implementing an ATOMIC section.

CORE_ENTER_ATOMIC

```
#define CORE_ENTER_ATOMIC
```

Value:

```
()
```

Enter ATOMIC section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_EXIT_ATOMIC

```
#define CORE_EXIT_ATOMIC
```

Value:

```
()
```

Exit ATOMIC section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_YIELD_ATOMIC

```
#define CORE_YIELD_ATOMIC
```

Value:

```
()
```

ATOMIC style yield.

CORE_IRQ_DISABLED

```
#define CORE_IRQ_DISABLED
```

Value:

```
()
```

Check if IRQ is disabled.

CORE_IN_IRQ_CONTEXT

```
#define CORE_IN_IRQ_CONTEXT
```

Value:

```
()
```

Check if inside an IRQ handler.

CORE_DECLARE_IRQ_STATE

```
#define CORE_DECLARE_IRQ_STATE
```

Allocate storage for PRIMASK or BASEPRI value for use by CORE_ENTER/EXIT_ATOMIC() and CORE_ENTER/EXIT_CRITICAL() macros.

CORE_CRITICAL_IRQ_DISABLE

```
#define CORE_CRITICAL_IRQ_DISABLE
```

CRITICAL style interrupt disable.

CORE_CRITICAL_IRQ_ENABLE

```
#define CORE_CRITICAL_IRQ_ENABLE
```

CRITICAL style interrupt enable.

CORE_CRITICAL_SECTION

```
#define CORE_CRITICAL_SECTION
```

Value:

```
{                                   \
  CORE_DECLARE_IRQ_STATE;           \
  CORE_ENTER_CRITICAL();            \
  {                                 \
    yourcode                        \
  }                                 \
  CORE_EXIT_CRITICAL();             \
}
```

Convenience macro for implementing a CRITICAL section.

CORE_ENTER_CRITICAL

```
#define CORE_ENTER_CRITICAL
```

Enter CRITICAL section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_EXIT_CRITICAL

```
#define CORE_EXIT_CRITICAL
```

Exit CRITICAL section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_YIELD_CRITICAL

```
#define CORE_YIELD_CRITICAL
```

CRITICAL style yield.

CORE_ATOMIC_IRQ_DISABLE

```
#define CORE_ATOMIC_IRQ_DISABLE
```

ATOMIC style interrupt disable.

CORE_ATOMIC_IRQ_ENABLE

```
#define CORE_ATOMIC_IRQ_ENABLE
```

ATOMIC style interrupt enable.

CORE_ATOMIC_SECTION

```
#define CORE_ATOMIC_SECTION
```

Value:

```
{                                   \
  CORE_DECLARE_IRQ_STATE;           \
  CORE_ENTER_ATOMIC();              \
  {                                 \
    yourcode                        \
  }                                 \
  CORE_EXIT_ATOMIC();               \
}
```

Convenience macro for implementing an ATOMIC section.

CORE_ENTER_ATOMIC

```
#define CORE_ENTER_ATOMIC
```

Enter ATOMIC section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_EXIT_ATOMIC

```
#define CORE_EXIT_ATOMIC
```

Exit ATOMIC section.

Assumes that a CORE_DECLARE_IRQ_STATE exist in scope.

CORE_YIELD_ATOMIC

```
#define CORE_YIELD_ATOMIC
```

ATOMIC style yield.

CORE_NVIC_REG_WORDS

```
#define CORE_NVIC_REG_WORDS
```

Value:

```
((EXT_IRQ_COUNT + 31) / 32)
```

Number of words in a NVIC mask set.

CORE_DEFAULT_VECTOR_TABLE_ENTRIES

```
#define CORE_DEFAULT_VECTOR_TABLE_ENTRIES
```

Value:

```
(EXT_IRQ_COUNT + 16)
```

Number of entries in a default interrupt vector table.

CORE_INTERRUPT_HIGHEST_PRIORITY

```
#define CORE_INTERRUPT_HIGHEST_PRIORITY
```

Value:

```
0
```

Highest priority for core interrupt.

CORE_INTERRUPT_DEFAULT_PRIORITY

```
#define CORE_INTERRUPT_DEFAULT_PRIORITY
```

Value:

```
5
```

Default priority for core interrupt.

CORE_INTERRUPT_LOWEST_PRIORITY

```
#define CORE_INTERRUPT_LOWEST_PRIORITY
```

Value:

```
7
```

Lowest priority for core interrupt.

CORE_ATOMIC_METHOD_DEFAULT

```
#define CORE_ATOMIC_METHOD_DEFAULT
```

Value:

```
CORE_ATOMIC_METHOD_BASEPRI
```

Default method to disable interrupts in ATOMIC sections.

CORE_ATOMIC_METHOD

```
#define CORE_ATOMIC_METHOD
```

Value:

```
CORE_ATOMIC_METHOD_PRIMASK
```

Specify which method to use when implementing ATOMIC sections.

You can select between BASEPRI or PRIMASK method. Note

- On Cortex-M0+ devices only PRIMASK can be used.

CORE_DECLARE_NVIC_STATE

```
#define CORE_DECLARE_NVIC_STATE
```

Value:

```
CORE_nvicMask_t nvicState
```

Allocate storage for NVIC interrupt masks for use by CORE_ENTER/EXIT_NVIC() macros.

CORE_DECLARE_NVIC_MASK

```
#define CORE_DECLARE_NVIC_MASK
```

Value:

```
(x)
```

Allocate storage for NVIC interrupt masks.

CORE_DECLARE_NVIC_ZEROMASK

```
#define CORE_DECLARE_NVIC_ZEROMASK
```

Value:

```
(x)
```

Allocate storage for and zero initialize NVIC interrupt mask.

CORE_NVIC_DISABLE

```
#define CORE_NVIC_DISABLE
```

Value:

```
(mask)
```

NVIC mask style interrupt disable.

CORE_NVIC_ENABLE

```
#define CORE_NVIC_ENABLE
```

Value:

```
(mask)
```

NVIC mask style interrupt enable.

CORE_NVIC_SECTION

```
#define CORE_NVIC_SECTION
```

Value:

```
{                                   \
  CORE_DECLARE_NVIC_STATE;          \
  CORE_ENTER_NVIC(mask);            \
  {                                 \
    yourcode                        \
  }                                 \
  CORE_EXIT_NVIC();                 \
}
```

Convenience macro for implementing a NVIC mask section.

CORE_ENTER_NVIC

```
#define CORE_ENTER_NVIC
```

Value:

```
(disable)
```

Enter NVIC mask section.

Assumes that a CORE_DECLARE_NVIC_STATE exist in scope.

CORE_EXIT_NVIC

```
#define CORE_EXIT_NVIC
```

Value:

```
()
```

Exit NVIC mask section.

Assumes that a CORE_DECLARE_NVIC_STATE exist in scope.

CORE_YIELD_NVIC

```
#define CORE_YIELD_NVIC
```

Value:

```
(enable)
```

NVIC maks style yield.

dwt_cycle_counter_handle_t

A Cycle Counter Instance.

Public Attributes

uint32_t	start Cycle counter at start of recording.
uint32_t	cycles Cycles elapsed in last recording.
uint32_t	max Max recorded cycles since last reset or init.

Public Attribute Documentation

start

```
uint32_t dwt_cycle_counter_handle_t::start
```

Cycle counter at start of recording.

cycles

```
uint32_t dwt_cycle_counter_handle_t::cycles
```

Cycles elapsed in last recording.

max

```
uint32_t dwt_cycle_counter_handle_t::max
```

Max recorded cycles since last reset or init.

CORE_nvicMask_t

Storage for NVIC interrupt masks.

Public Attributes

uint32_t	a Array of NVIC mask words.

Public Attribute Documentation

a

```
uint32_t CORE_nvicMask_t::a[CORE_NVIC_REG_WORDS]
```

Array of NVIC mask words.

DAC - Digital to Analog Converter

DAC - Digital to Analog Converter

Digital to Analog Converter (DAC) Peripheral API.

This module contains functions to control the DAC peripheral of Silicon Labs 32-bit MCUs and SoCs. The DAC converts digital values to analog signals at up to 500 ksps with 12-bit accuracy. The DAC is designed for low-energy consumption and can also provide very good performance.

Modules

DAC_Init_TypeDef

DAC_InitChannel_TypeDef

Enumerations

enum DAC_ConvMode_TypeDef {

dacConvModeContinuous = _DAC_CTRL_CONVMODE_CONTINUOUS
dacConvModeSampleHold = _DAC_CTRL_CONVMODE_SAMPLEHOLD
dacConvModeSampleOff = _DAC_CTRL_CONVMODE_SAMPLEOFF

}

Conversion mode.

enum DAC_Output_TypeDef {

dacOutputDisable = _DAC_CTRL_OUTMODE_DISABLE
dacOutputPin = _DAC_CTRL_OUTMODE_PIN
dacOutputADC = _DAC_CTRL_OUTMODE_ADC
dacOutputPinADC = _DAC_CTRL_OUTMODE_PINADC

}

Output mode.

enum DAC_PRSSEL_TypeDef {

dacPRSSELCh0 = _DAC_CH0CTRL_PRSSEL_PRSCH0
dacPRSSELCh1 = _DAC_CH0CTRL_PRSSEL_PRSCH1
dacPRSSELCh2 = _DAC_CH0CTRL_PRSSEL_PRSCH2
dacPRSSELCh3 = _DAC_CH0CTRL_PRSSEL_PRSCH3
dacPRSSELCh4 = _DAC_CH0CTRL_PRSSEL_PRSCH4
dacPRSSELCh5 = _DAC_CH0CTRL_PRSSEL_PRSCH5
dacPRSSELCh6 = _DAC_CH0CTRL_PRSSEL_PRSCH6
dacPRSSELCh7 = _DAC_CH0CTRL_PRSSEL_PRSCH7
dacPRSSELCh8 = _DAC_CH0CTRL_PRSSEL_PRSCH8
dacPRSSELCh9 = _DAC_CH0CTRL_PRSSEL_PRSCH9
dacPRSSELCh10 = _DAC_CH0CTRL_PRSSEL_PRSCH10
dacPRSSELCh11 = _DAC_CH0CTRL_PRSSEL_PRSCH11

}

Peripheral Reflex System signal used to trigger single sample.

enum DAC_Ref_TypeDef {

dacRef1V25 = _DAC_CTRL_REFSEL_1V25
dacRef2V5 = _DAC_CTRL_REFSEL_2V5
dacRefVDD = _DAC_CTRL_REFSEL_VDD

}

Reference voltage for DAC.

enum DAC_Refresh_TypeDef {

dacRefresh8 = _DAC_CTRL_REFRSEL_8CYCLES
dacRefresh16 = _DAC_CTRL_REFRSEL_16CYCLES
dacRefresh32 = _DAC_CTRL_REFRSEL_32CYCLES
dacRefresh64 = _DAC_CTRL_REFRSEL_64CYCLES

}
Refresh interval.

Functions

void DAC_Enable(DAC_TypeDef *dac, unsigned int ch, bool enable)
Enable/disable the DAC channel.

void DAC_Init(DAC_TypeDef *dac, const DAC_Init_TypeDef *init)
Initialize DAC.

void DAC_InitChannel(DAC_TypeDef *dac, const DAC_InitChannel_TypeDef *init, unsigned int ch)
Initialize DAC channel.

void DAC_ChannelOutputSet(DAC_TypeDef *dac, unsigned int channel, uint32_t value)
Set the output signal of a DAC channel to a given value.

uint8_t DAC_PrescaleCalc(uint32_t dacFreq, uint32_t hfperFreq)
Calculate prescaler value used to determine the DAC clock.

void DAC_Reset(DAC_TypeDef *dac)
Reset DAC to the same state that it was in after a hardware reset.

void DAC_Channel0OutputSet(DAC_TypeDef *dac, uint32_t value)
Set the output signal of DAC channel 0 to a given value.

void DAC_Channel1OutputSet(DAC_TypeDef *dac, uint32_t value)
Set the output signal of DAC channel 1 to a given value.

void DAC_IntClear(DAC_TypeDef *dac, uint32_t flags)
Clear one or more pending DAC interrupts.

void DAC_IntDisable(DAC_TypeDef *dac, uint32_t flags)
Disable one or more DAC interrupts.

void DAC_IntEnable(DAC_TypeDef *dac, uint32_t flags)
Enable one or more DAC interrupts.

uint32_t DAC_IntGet(DAC_TypeDef *dac)
Get pending DAC interrupt flags.

uint32_t DAC_IntGetEnabled(DAC_TypeDef *dac)
Get enabled and pending DAC interrupt flags.

void DAC_IntSet(DAC_TypeDef *dac, uint32_t flags)
Set one or more pending DAC interrupts from SW.

Macros

#define DAC_INIT_DEFAULT undefined
Default configuration for DAC initialization structure.

#define DAC_INITCHANNEL_DEFAULT undefined
Default configuration for DAC channel initialization structure.

Enumeration Documentation

DAC_ConvMode_TypeDef

DAC_ConvMode_TypeDef

Conversion mode.

Enumerator	
dacConvModeContinuous	Continuous mode.
dacConvModeSampleHold	Sample/hold mode.
dacConvModeSampleOff	Sample/shut off mode.

DAC_Output_TypeDef

DAC_Output_TypeDef

Output mode.

Enumerator	
dacOutputDisable	Output to pin and ADC disabled.
dacOutputPin	Output to pin only.
dacOutputADC	Output to ADC only.
dacOutputPinADC	Output to pin and ADC.

DAC_PRSSEL_TypeDef

DAC_PRSSEL_TypeDef

Peripheral Reflex System signal used to trigger single sample.

Enumerator	
dacPRSSELCh0	PRS channel 0.
dacPRSSELCh1	PRS channel 1.
dacPRSSELCh2	PRS channel 2.
dacPRSSELCh3	PRS channel 3.
dacPRSSELCh4	PRS channel 4.
dacPRSSELCh5	PRS channel 5.
dacPRSSELCh6	PRS channel 6.
dacPRSSELCh7	PRS channel 7.
dacPRSSELCh8	PRS channel 8.
dacPRSSELCh9	PRS channel 9.
dacPRSSELCh10	PRS channel 10.
dacPRSSELCh11	PRS channel 11.

DAC_Ref_TypeDef

DAC_Ref_TypeDef

Reference voltage for DAC.

Enumerator	
dacRef1V25	Internal 1.25V bandgap reference.
dacRef2V5	Internal 2.5V bandgap reference.
dacRefVDD	VDD reference.

DAC_Refresh_TypeDef

> DAC_Refresh_TypeDef

Refresh interval.

Enumerator	
dacRefresh8	Refresh every 8 prescaled cycles.
dacRefresh16	Refresh every 16 prescaled cycles.
dacRefresh32	Refresh every 32 prescaled cycles.
dacRefresh64	Refresh every 64 prescaled cycles.

Function Documentation

DAC_Enable

> void DAC_Enable (DAC_TypeDef * dac, unsigned int ch, bool enable)

Enable/disable the DAC channel.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
unsigned int	[in]	ch	A channel to enable/disable.
bool	[in]	enable	true to enable DAC channel, false to disable.

DAC_Init

> void DAC_Init (DAC_TypeDef * dac, const DAC_Init_TypeDef * init)

Initialize DAC.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
const DAC_Init_TypeDef *	[in]	init	A pointer to the DAC initialization structure.

Initializes common parts for both channels. In addition, channel control configuration must be done. See DAC_InitChannel().

Note

- This function will disable both channels prior to configuration.

DAC_InitChannel

```
void DAC_InitChannel (DAC_TypeDef * dac, const DAC_InitChannel_TypeDef * init, unsigned int ch)
```

Initialize DAC channel.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
const DAC_InitChannel_TypeDef *	[in]	init	A pointer to the DAC initialization structure.
unsigned int	[in]	ch	A channel number to initialize.

DAC_ChannelOutputSet

```
void DAC_ChannelOutputSet (DAC_TypeDef * dac, unsigned int channel, uint32_t value)
```

Set the output signal of a DAC channel to a given value.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
unsigned int	[in]	channel	A channel number to set the output.
uint32_t	[in]	value	A value to write to the channel output register CHnDATA.

This function sets the output signal of a DAC channel by writing `value` to the corresponding CHnDATA register.

DAC_PrescaleCalc

```
uint8_t DAC_PrescaleCalc (uint32_t dacFreq, uint32_t hfperFreq)
```

Calculate prescaler value used to determine the DAC clock.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	dacFreq	DAC frequency wanted. The frequency will automatically be adjusted to be below the maximum allowed DAC clock.
uint32_t	[in]	hfperFreq	Frequency in Hz of the reference HFPER clock. Set to 0 to use the currently defined HFPER clock setting.

The DAC clock is given by: HFPERCLK / (prescale ^ 2). If the requested DAC frequency is low and the maximum prescaler value can't adjust the actual DAC frequency lower than the requested DAC frequency, the maximum prescaler value is returned resulting in a higher DAC frequency than requested.

Returns

- Prescaler value to use for DAC to achieve a clock value <= `dacFreq`.

DAC_Reset

```
void DAC_Reset (DAC_TypeDef * dac)
```

Reset DAC to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the ADC peripheral register block.

DAC_Channel0OutputSet

```
void DAC_Channel0OutputSet (DAC_TypeDef * dac, uint32_t value)
```

Set the output signal of DAC channel 0 to a given value.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	value	Value to write to the channel 0 output register CH0DATA.

This function sets the output signal of DAC channel 0 by writing `value` to the CH0DATA register.

DAC_Channel1OutputSet

```
void DAC_Channel1OutputSet (DAC_TypeDef * dac, uint32_t value)
```

Set the output signal of DAC channel 1 to a given value.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	value	Value to write to the channel 1 output register CH1DATA.

Sets the output signal of DAC channel 1 by writing `value` to the CH1DATA register.

DAC_IntClear

```
void DAC_IntClear (DAC_TypeDef * dac, uint32_t flags)
```

Clear one or more pending DAC interrupts.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	flags	Pending DAC interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the DAC module (DAC_IF_nnn).

DAC_IntDisable

```
void DAC_IntDisable (DAC_TypeDef * dac, uint32_t flags)
```

Disable one or more DAC interrupts.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	flags	DAC interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the DAC module (DAC_IF_nnn).

DAC_IntEnable

```
void DAC_IntEnable (DAC_TypeDef * dac, uint32_t flags)
```

Enable one or more DAC interrupts.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	flags	DAC interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the DAC module (DAC_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using DAC_IntClear() prior to enabling the interrupt.

DAC_IntGet

```
uint32_t DAC_IntGet (DAC_TypeDef * dac)
```

Get pending DAC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.

Note

- Event bits are not cleared by the use of this function.

Returns

- DAC interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the DAC module (DAC_IF_nnn).

DAC_IntGetEnabled

uint32_t DAC_IntGetEnabled (DAC_TypeDef * dac)

Get enabled and pending DAC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled DAC interrupt sources. Return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in DACx_IEN_nnn register (DACx_IEN_nnn) and
  - the OR combination of valid interrupt flags of the DAC module (DACx_IF_nnn).

DAC_IntSet

void DAC_IntSet (DAC_TypeDef * dac, uint32_t flags)

Set one or more pending DAC interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	Pointer to DAC peripheral register block.
uint32_t	[in]	flags	DAC interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for the DAC module (DAC_IF_nnn).

Macro Definition Documentation

DAC_INIT_DEFAULT

#define DAC_INIT_DEFAULT

Value:

```
{                                                       \
  dacRefresh8,            /* Refresh every 8 prescaled cycles. */    \
  dacRef1V25,             /* 1.25V internal reference. */            \
  dacOutputPin,           /* Output to pin only. */                  \
  dacConvModeContinuous,  /* Continuous mode. */                     \
  0,                      /* No prescaling. */                       \
  false,                  /* Do not enable low pass filter. */       \
  false,                  /* Do not reset prescaler on ch0 start. */ \
  false,                  /* DAC output enable always on. */         \
  false,                  /* Disable sine mode. */                   \
  false                   /* Single ended mode. */                   \
}
```

Default configuration for DAC initialization structure.

DAC_INITCHANNEL_DEFAULT

#define DAC_INITCHANNEL_DEFAULT

Value:

```
0   {                                                                   \
0     false,            /* Leave channel disabled when initialization done. */ \
0     false,            /* Disable PRS triggering. */                   \
0     false,            /* Channel not refreshed automatically. */      \
0     dacPRSSELCh0      /* Select PRS ch0 (if PRS triggering enabled). */     \
0   }
```

Default configuration for DAC channel initialization structure.

DAC_Init_TypeDef

DAC initialization structure, common for both channels.

Public Attributes

DAC_Refresh_TypeDef	**refresh** Refresh interval.
DAC_Ref_TypeDef	**reference** Reference voltage to use.
DAC_Output_TypeDef	**outMode** Output mode.
DAC_ConvMode_TypeDef	**convMode** Conversion mode.
uint8_t	**prescale** Prescaler used to get DAC clock.
bool	**lpEnable** Enable/disable use of low pass filter on output.
bool	**ch0ResetPre** Enable/disable reset of prescaler on ch0 start.
bool	**outEnablePRS** Enable/disable output enable control by CH1 PRS signal.
bool	**sineEnable** Enable/disable sine mode.
bool	**diff** Select if single ended or differential mode.

Public Attribute Documentation

refresh

```
DAC_Refresh_TypeDef DAC_Init_TypeDef::refresh
```

Refresh interval.

Only used if REFREN bit set for a DAC channel.

reference

```
DAC_Ref_TypeDef DAC_Init_TypeDef::reference
```

Reference voltage to use.

outMode

```
DAC_Output_TypeDef DAC_Init_TypeDef::outMode
```

Output mode.

convMode

```
DAC_ConvMode_TypeDef DAC_Init_TypeDef::convMode
```

Conversion mode.

prescale

```
uint8_t DAC_Init_TypeDef::prescale
```

Prescaler used to get DAC clock.

Derived as follows: DACclk=HFPERclk/(2^prescale). The DAC clock should be <= 1MHz.

lpEnable

```
bool DAC_Init_TypeDef::lpEnable
```

Enable/disable use of low pass filter on output.

ch0ResetPre

```
bool DAC_Init_TypeDef::ch0ResetPre
```

Enable/disable reset of prescaler on ch0 start.

outEnablePRS

```
bool DAC_Init_TypeDef::outEnablePRS
```

Enable/disable output enable control by CH1 PRS signal.

sineEnable

```
bool DAC_Init_TypeDef::sineEnable
```

Enable/disable sine mode.

diff

```
bool DAC_Init_TypeDef::diff
```

Select if single ended or differential mode.

DAC_InitChannel_TypeDef

DAC channel initialization structure.

Public Attributes

bool	enable Enable channel.
bool	prsEnable Peripheral reflex system trigger enable.
bool	refreshEnable Enable/disable automatic refresh of channel.
DAC_PRSSEL_TypeDef	prsSel Peripheral reflex system trigger selection.

Public Attribute Documentation

enable

```
bool DAC_InitChannel_TypeDef::enable
```

Enable channel.

prsEnable

```
bool DAC_InitChannel_TypeDef::prsEnable
```

Peripheral reflex system trigger enable.

If false, channel is triggered by writing to CHnDATA.

refreshEnable

```
bool DAC_InitChannel_TypeDef::refreshEnable
```

Enable/disable automatic refresh of channel.

Refresh interval must be defined in common control initialization, see DAC_Init() for more information.

prsSel

```
DAC_PRSSEL_TypeDef DAC_InitChannel_TypeDef::prsSel
```

Peripheral reflex system trigger selection.

Only applicable if `prsEnable` is enabled.

DBG - Debug

DBG - Debug

Debug (DBG) Peripheral API.

This module contains functions to control the DBG peripheral of Silicon Labs 32-bit MCUs and SoCs. The Debug Interface is used to program and debug Silicon Labs devices.

Enumerations

enum DBG_LockMode_TypeDef {
  dbgLockModeAllowErase = 1UL
}
Lock modes.

Functions

bool DBG_Connected(void)
Check if a debugger is connected (and debug session activated).

void DBG_SWOEnable(unsigned int location)
Enable Serial Wire Output (SWO) pin.

void DBG_DisableDebugAccess(DBG_LockMode_TypeDef lockMode)
Disable debug access.

Enumeration Documentation

DBG_LockMode_TypeDef

> DBG_LockMode_TypeDef

Lock modes.

Enumerator	
dbgLockModeAllowErase	Lock debug access.

Function Documentation

DBG_Connected

> bool DBG_Connected (void)

Check if a debugger is connected (and debug session activated).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Used to make run-time decisions depending on whether or not a debug session has been active since last reset, i.e., using a debug probe or similar. In some cases, special handling is required in that scenario.

Returns

- True if a debug session is active since last reset, otherwise false.

DBG_SWOEnable

> void DBG_SWOEnable (unsigned int location)

Enable Serial Wire Output (SWO) pin.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	location	A pin location used for SWO pin on the application in use.

The SWO pin (sometimes denoted SWV, serial wire viewer) allows for miscellaneous output to be passed from the Cortex-M3 debug trace module to an external debug probe. By default, the debug trace module and pin output may be disabled.

Since the SWO pin is only useful when using a debugger, a suggested use of this function during startup may be:

```
* if (DBG_Connected())
* {
* DBG_SWOEnable(1);
* }
*
```

By checking if the debugger is attached, a setup leading to a higher energy consumption when the debugger is attached can be avoided when not using a debugger.

Another alternative may be to set the debugger tool chain to configure the required setup (similar to the content of this function) by some sort of toolchain scripting during its attach/reset procedure. In that case, the above suggested code for enabling the SWO pin is not required in the application.

DBG_DisableDebugAccess

> void DBG_DisableDebugAccess (DBG_LockMode_TypeDef lockMode)

Disable debug access.

Parameters

Type	Direction	Argument Name	Description
DBG_LockMode_TypeDef	[in]	lockMode	Debug lock mode to be used.

DMA - Direct Memory Access

DMA - Direct Memory Access

Direct Memory Access (DMA) Peripheral API.

DMA access functions provide basic support for the following types of DMA cycles:

- Basic, used for transferring data between memory and peripherals.
- Auto-request, used for transferring data between memory locations.
- Ping-pong, used for for continuous transfer of data between memory and peripherals, automatically toggling between primary and alternate descriptors.
- Memoryscatter-gather, used for transferring a number of buffers between memory locations.
- Peripheralscatter-gather, used for transferring a number of buffers between memory and peripherals.

A basic understanding of the DMA controller is assumed. See the reference manual for more details.

The term 'descriptor' is synonymous to the 'channel control data structure' term.

To use the DMA controller, the initialization function must have been executed once (normally during the system initialization):

```
* DMA_Init();
*
```

Normally, a DMA channel is configured:

```
* DMA_CfgChannel();
*
```

The channel configuration only has to be done once if reusing the channel for the same purpose later.

To set up a DMA cycle, the primary and/or alternate descriptor has to be set up as indicated below.

For basic or auto-request cycles, use once on either primary or alternate descriptor:

```
* DMA_CfgDescr();
*
```

For ping-pong cycles, configure both primary or alternate descriptors:

```
* DMA_CfgDescr(); // Primary descriptor config
* DMA_CfgDescr(); // Alternate descriptor config
*
```

For scatter-gather cycles, program the alternate descriptor array:

```
* // 'n' is the number of scattered buffers
* // 'descr' points to the start of the alternate descriptor array
*
* // Fill in 'cfg'
* DMA_CfgDescrScatterGather(descr, 0, cfg);
* // Fill in 'cfg'
* DMA_CfgDescrScatterGather(descr, 1, cfg);
* :
* // Fill in 'cfg'
* DMA_CfgDescrScatterGather(descr, n - 1, cfg);
*
```

In many cases, the descriptor configuration only has to be done once if re-using the channel for the same type of DMA cycles later.

To activate the DMA cycle, use the respective DMA_Activate...() function.

For ping-pong DMA cycles, use DMA_RefreshPingPong() from the callback to prepare the completed descriptor for reuse. Notice that the refresh must be done prior to the other active descriptor completes, otherwise the ping-pong DMA cycle will halt.

Modules

DMA_CB_TypeDef

DMA_CfgChannel_TypeDef

DMA_CfgDescr_TypeDef

DMA_CfgLoop_TypeDef

DMA_CfgRect_TypeDef

DMA_CfgDescrSGAlt_TypeDef

DMA_Init_TypeDef

Enumerations

enum DMA_DataInc_TypeDef {

dmaDataInc1 = _DMA_CTRL_SRC_INC_BYTE
dmaDataInc2 = _DMA_CTRL_SRC_INC_HALFWORD
dmaDataInc4 = _DMA_CTRL_SRC_INC_WORD
dmaDataIncNone = _DMA_CTRL_SRC_INC_NONE

}

Amount source/destination address should be incremented for each data transfer.

enum DMA_DataSize_TypeDef {

dmaDataSize1 = _DMA_CTRL_SRC_SIZE_BYTE
dmaDataSize2 = _DMA_CTRL_SRC_SIZE_HALFWORD
dmaDataSize4 = _DMA_CTRL_SRC_SIZE_WORD

}

Data sizes (in number of bytes) to be read/written by DMA transfer.

enum DMA_CycleCtrl_TypeDef {

dmaCycleCtrlBasic = _DMA_CTRL_CYCLE_CTRL_BASIC
dmaCycleCtrlAuto = _DMA_CTRL_CYCLE_CTRL_AUTO
dmaCycleCtrlPingPong = _DMA_CTRL_CYCLE_CTRL_PINGPONG
dmaCycleCtrlMemScatterGather = _DMA_CTRL_CYCLE_CTRL_MEM_SCATTER_GATHER
dmaCycleCtrlPerScatterGather = _DMA_CTRL_CYCLE_CTRL_PER_SCATTER_GATHER

}

Types of DMA transfer.

enum DMA_ArbiterConfig_TypeDef {

dmaArbitrate1 = _DMA_CTRL_R_POWER_1
dmaArbitrate2 = _DMA_CTRL_R_POWER_2
dmaArbitrate4 = _DMA_CTRL_R_POWER_4
dmaArbitrate8 = _DMA_CTRL_R_POWER_8
dmaArbitrate16 = _DMA_CTRL_R_POWER_16
dmaArbitrate32 = _DMA_CTRL_R_POWER_32
dmaArbitrate64 = _DMA_CTRL_R_POWER_64
dmaArbitrate128 = _DMA_CTRL_R_POWER_128
dmaArbitrate256 = _DMA_CTRL_R_POWER_256
dmaArbitrate512 = _DMA_CTRL_R_POWER_512
dmaArbitrate1024 = _DMA_CTRL_R_POWER_1024

}

Number of transfers before controller does new arbitration.

Typedefs

typedef void(* DMA_FuncPtr_TypeDef)(unsigned int channel, bool primary, void *user)

DMA interrupt callback function pointer.

Functions

void DMA_ActivateAuto(unsigned int channel, bool primary, void *dst, const void *src, unsigned int nMinus1)

Activate the DMA auto-request cycle (used for memory-memory transfers).

void DMA_ActivateBasic(unsigned int channel, bool primary, bool useBurst, void *dst, const void *src, unsigned int nMinus1)

Activate the DMA basic cycle (used for memory-peripheral transfers).

void DMA_ActivatePingPong(unsigned int channel, bool useBurst, void *primDst, const void *primSrc, unsigned int primNMinus1, void *altDst, const void *altSrc, unsigned int altNMinus1)

Activate a DMA ping-pong cycle (used for memory-peripheral transfers).

void DMA_ActivateScatterGather(unsigned int channel, bool useBurst, DMA_DESCRIPTOR_TypeDef *altDescr, unsigned int count)

Activate the DMA scatter-gather cycle (used for either memory-peripheral or memory-memory transfers).

void DMA_CfgChannel(unsigned int channel, DMA_CfgChannel_TypeDef *cfg)

Configure a DMA channel.

void DMA_CfgDescr(unsigned int channel, bool primary, DMA_CfgDescr_TypeDef *cfg)

Configure the DMA descriptor for auto-request, basic, or ping-pong DMA cycles.

void DMA_CfgLoop(unsigned int channel, DMA_CfgLoop_TypeDef *cfg)

Configure the DMA channel for Loop mode or 2D transfer.

void DMA_CfgRect(unsigned int channel, DMA_CfgRect_TypeDef *cfg)

Configure the DMA channel 2D transfer properties.

void DMA_ResetLoop(unsigned int channel)

Clear Loop configuration for channel.

void DMA_ResetRect(unsigned int channel)

Clear Rect/2D DMA configuration for channel.

void DMA_CfgDescrScatterGather(DMA_DESCRIPTOR_TypeDef *descr, unsigned int indx, DMA_CfgDescrSGAlt_TypeDef *cfg)

Configure an alternate DMA descriptor for use with scatter-gather DMA cycles.

void DMA_ChannelEnable(unsigned int channel, bool enable)

Enable or disable a DMA channel.

bool DMA_ChannelEnabled(unsigned int channel)

Check if the DMA channel is enabled.

void	DMA_ChannelRequestEnable(unsigned int channel, bool enable) Enable or disable a DMA channel request.
void	DMA_Init(DMA_Init_TypeDef *init) Initialize the DMA controller.
void	DMA_IRQHandler(void) Interrupt handler for the DMA cycle completion handling.
void	DMA_RefreshPingPong(unsigned int channel, bool primary, bool useBurst, void *dst, const void *src, unsigned int nMinus1, bool stop) Refresh a descriptor used in a DMA ping-pong cycle.
void	DMA_Reset(void) Reset the DMA controller.
void	DMA_IntClear(uint32_t flags) Clear one or more pending DMA interrupts.
void	DMA_IntDisable(uint32_t flags) Disable one or more DMA interrupts.
void	DMA_IntEnable(uint32_t flags) Enable one or more DMA interrupts.
uint32_t	DMA_IntGet(void) Get pending DMA interrupt flags.
uint32_t	DMA_IntGetEnabled(void) Get enabled and pending DMA interrupt flags.
void	DMA_IntSet(uint32_t flags) Set one or more pending DMA interrupts.

Enumeration Documentation

DMA_DataInc_TypeDef

> DMA_DataInc_TypeDef

Amount source/destination address should be incremented for each data transfer.

Enumerator	
dmaDataInc1	Increment address 1 byte.
dmaDataInc2	Increment address 2 bytes.
dmaDataInc4	Increment address 4 bytes.
dmaDataIncNone	Do not increment address.

DMA_DataSize_TypeDef

> DMA_DataSize_TypeDef

Data sizes (in number of bytes) to be read/written by DMA transfer.

Enumerator	
dmaDataSize1	1 byte DMA transfer size.
dmaDataSize2	2 byte DMA transfer size.
dmaDataSize4	4 byte DMA transfer size.

DMA_CycleCtrl_TypeDef

DMA_CycleCtrl_TypeDef

Types of DMA transfer.

Enumerator	
dmaCycleCtrlBasic	Basic DMA cycle.
dmaCycleCtrlAuto	Auto-request DMA cycle.
dmaCycleCtrlPingPong	Ping-pong DMA cycle.
dmaCycleCtrlMemScatterGather	Memory scatter-gather DMA cycle.
dmaCycleCtrlPerScatterGather	Peripheral scatter-gather DMA cycle.

DMA_ArbiterConfig_TypeDef

DMA_ArbiterConfig_TypeDef

Number of transfers before controller does new arbitration.

Enumerator	
dmaArbitrate1	Arbitrate after 1 DMA transfer.
dmaArbitrate2	Arbitrate after 2 DMA transfers.
dmaArbitrate4	Arbitrate after 4 DMA transfers.
dmaArbitrate8	Arbitrate after 8 DMA transfers.
dmaArbitrate16	Arbitrate after 16 DMA transfers.
dmaArbitrate32	Arbitrate after 32 DMA transfers.
dmaArbitrate64	Arbitrate after 64 DMA transfers.
dmaArbitrate128	Arbitrate after 128 DMA transfers.
dmaArbitrate256	Arbitrate after 256 DMA transfers.
dmaArbitrate512	Arbitrate after 512 DMA transfers.
dmaArbitrate1024	Arbitrate after 1024 DMA transfers.

Typedef Documentation

DMA_FuncPtr_TypeDef

typedef void(* DMA_FuncPtr_TypeDef) (unsigned int channel, bool primary, void *user))(unsigned int channel, bool primary, void *user)

DMA interrupt callback function pointer.

Parameters:

- channel - DMA channel the callback function is invoked for.
- primary - Indicates if callback is invoked for completion of primary (true) or alternate (false) descriptor. Mainly useful for ping-pong DMA cycles, to know which descriptor to refresh.
- user - User definable reference that may be used to pass information to be used by the callback handler. If used, referenced data must be valid at the point when the interrupt handler invokes callback. If callback changes any data in the provided user structure, remember that those changes are done in interrupt context and proper protection of data may be required.

Function Documentation

DMA_ActivateAuto

```
void DMA_ActivateAuto (unsigned int channel, bool primary, void * dst, const void * src, unsigned int nMinus1)
```

Activate the DMA auto-request cycle (used for memory-memory transfers).

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to activate the DMA cycle for.
bool	[in]	primary	• true - activate using the primary descriptor • false - activate using an alternate descriptor
void *	[in]	dst	An ddress to a start location to transfer data to. If NULL, leave setting in descriptor as is from a previous activation.
const void *	[in]	src	An address to a start location to transfer data from. If NULL, leave setting in descriptor as is from a previous activation.
unsigned int	[in]	nMinus1	A number of DMA transfer elements (minus 1) to transfer (<= 1023). The size of the DMA transfer element (1, 2 or 4 bytes) is configured with DMA_CfgDescr().

Prior to activating the DMA cycle, the channel and descriptor to be used must have been properly configured.

Note

- If using this function on a channel already activated and in use by the DMA controller, the behavior is undefined.

DMA_ActivateBasic

```
void DMA_ActivateBasic (unsigned int channel, bool primary, bool useBurst, void * dst, const void * src, unsigned int nMinus1)
```

Activate the DMA basic cycle (used for memory-peripheral transfers).

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to activate the DMA cycle for.
bool	[in]	primary	• true - activate using the primary descriptor • false - activate using an alternate descriptor
bool	[in]	useBurst	The burst feature is only used on peripherals supporting DMA bursts. Bursts must not be used if the total length (as given by nMinus1) is less than the arbitration rate configured for the descriptor. See the reference manual for more details on burst usage.
void *	[in]	dst	An address to a start location to transfer data to. If NULL, leave setting in descriptor as is from a previous activation.
const void *	[in]	src	An address to a start location to transfer data from. If NULL, leave setting in descriptor as is from a previous activation.

Type	Direction	Argument Name	Description
unsigned int	[in]	nMinus1	A number of DMA transfer elements (minus 1) to transfer (<= 1023). The size of the DMA transfer element (1, 2 or 4 bytes) is configured with DMA_CfgDescr().

Prior to activating the DMA cycle, the channel and descriptor to be used must have been properly configured.

Note

- If using this function on a channel already activated and in use by the DMA controller, the behavior is undefined.

DMA_ActivatePingPong

```
void DMA_ActivatePingPong (unsigned int channel, bool useBurst, void * primDst, const void * primSrc,
unsigned int primNMinus1, void * altDst, const void * altSrc, unsigned int altNMinus1)
```

Activate a DMA ping-pong cycle (used for memory-peripheral transfers).

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to activate DMA cycle for.
bool	[in]	useBurst	The burst feature is only used on peripherals supporting DMA bursts. Bursts must not be used if the total length (as given by nMinus1) is less than the arbitration rate configured for the descriptors. See the reference manual for more details on burst usage. Notice that this setting is used for both the primary and alternate descriptors.
void *	[in]	primDst	An address to a start location to transfer data to, for the primary descriptor. If NULL, leave setting in descriptor as is from a previous activation.
const void *	[in]	primSrc	An address to a start location to transfer data from, for the primary descriptor. If NULL, leave setting in the descriptor as is from a previous activation.
unsigned int	[in]	primNMinus1	A number of DMA transfer elements (minus 1) to transfer (<= 1023), for primary descriptor. The size of the DMA transfer element (1, 2 or 4 bytes) is configured with DMA_CfgDescr().
void *	[in]	altDst	An address to a start location to transfer data to, for an alternate descriptor. If NULL, leave setting in descriptor as is from a previous activation.
const void *	[in]	altSrc	An address to a start location to transfer data from, for an alternate descriptor. If NULL, leave setting in descriptor as is from a previous activation.
unsigned int	[in]	altNMinus1	A number of DMA transfer elements (minus 1) to transfer (<= 1023), for an alternate descriptor. The size of the DMA transfer element (1, 2 or 4 bytes) is configured with DMA_CfgDescr().

Prior to activating the DMA cycle, the channel and both descriptors must have been properly configured. The primary descriptor is always the first descriptor to be used by the DMA controller.

Note

- If using this function on a channel already activated and in use by the DMA controller, the behavior is undefined.

DMA_ActivateScatterGather

void DMA_ActivateScatterGather (unsigned int channel, bool useBurst, DMA_DESCRIPTOR_TypeDef * altDescr, unsigned int count)

Activate the DMA scatter-gather cycle (used for either memory-peripheral or memory-memory transfers).

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to activate DMA cycle for.
bool	[in]	useBurst	The burst feature is only used on peripherals supporting DMA bursts (this parameter is ignored for memory scatter-gather cycles). This parameter determines if bursts should be enabled during DMA transfers using the alternate descriptors. Bursts must not be used if the total length (as given by nMinus1 for the alternate descriptor) is less than the arbitration rate configured for the descriptor. See the reference manual for more details on burst usage.
DMA_DESCRIPTOR_TypeDef *	[inout]	altDescr	A pointer to a start of an array with prepared alternate descriptors. The last descriptor will have its cycle control type reprogrammed to a basic type.
unsigned int	[in]	count	A number of alternate descriptors in `altDescr` array. The maximum number of alternate descriptors is 256.

Prior to activating the DMA cycle, the array with alternate descriptors must have been properly configured. This function can be reused without reconfiguring the alternate descriptors, as long as `count` is the same.

Note

- If using this function on a channel already activated and in use by the DMA controller, the behavior is undefined.

DMA_CfgChannel

void DMA_CfgChannel (unsigned int channel, DMA_CfgChannel_TypeDef * cfg)

Configure a DMA channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to configure.
DMA_CfgChannel_TypeDef *	[in]	cfg	The configuration to use.

Configure miscellaneous issues for a DMA channel. This function is typically used once to set up a channel for a certain type of use.

Note

- If using this function on a channel already in use by the DMA controller, the behavior is undefined.

DMA_CfgDescr

> void DMA_CfgDescr (unsigned int channel, bool primary, DMA_CfgDescr_TypeDef * cfg)

Configure the DMA descriptor for auto-request, basic, or ping-pong DMA cycles.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to configure for.
bool	[in]	primary	• true - configure the primary descriptor • false - configure an alternate descriptor
DMA_CfgDescr_TypeDef *	[in]	cfg	The configuration to use.

This function is used to configure a descriptor for the following DMA cycle types:

- auto-request - used for a memory/memory transfer
- basic - used for a peripheral/memory transfer
- ping-pong - used for a ping-pong-based peripheral/memory transfer style providing time to refresh one descriptor while the other is in use.

The DMA cycle is not activated. See DMA_ActivateAuto(), DMA_ActivateBasic(), or DMA_ActivatePingPong() to activate the DMA cycle. In many cases, the configuration only has to be done once, and all subsequent cycles may be activated with the activate function.

For ping-pong DMA cycles, this function must be used both on the primary and the alternate descriptor prior to activating the DMA cycle.

Notice that the DMA channel must also be configured. See DMA_CfgChannel().

Note

- If using this function on a descriptor already activated and in use by the DMA controller, the behavior is undefined.

DMA_CfgLoop

> void DMA_CfgLoop (unsigned int channel, DMA_CfgLoop_TypeDef * cfg)

Configure the DMA channel for Loop mode or 2D transfer.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to configure for.
DMA_CfgLoop_TypeDef *	[in]	cfg	The configuration to use.

For 2D transfer, set cfg->enable to "false" and only configure nMinus1 to the same width as the channel descriptor.

DMA_CfgRect

> void DMA_CfgRect (unsigned int channel, DMA_CfgRect_TypeDef * cfg)

Configure the DMA channel 2D transfer properties.

Parameters

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to configure for.
DMA_CfgRect_TypeDef *	[in]	cfg	The configuration to use.

DMA_ResetLoop

```
void DMA_ResetLoop (unsigned int channel)
```

Clear Loop configuration for channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	Channel to reset loop configuration for.

DMA_ResetRect

```
void DMA_ResetRect (unsigned int channel)
```

Clear Rect/2D DMA configuration for channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	Channel to reset loop configuration for.

DMA_CfgDescrScatterGather

```
void DMA_CfgDescrScatterGather (DMA_DESCRIPTOR_TypeDef * descr, unsigned int indx,
DMA_CfgDescrSGAlt_TypeDef * cfg)
```

Configure an alternate DMA descriptor for use with scatter-gather DMA cycles.

Parameters

Type	Direction	Argument Name	Description
DMA_DESCRIPTOR_TypeDef *	[in]	descr	Points to the start of a memory area holding the alternate descriptors.
unsigned int	[in]	indx	An alternate descriptor index number to configure (numbered from 0).
DMA_CfgDescrSGAlt_TypeDef *	[in]	cfg	The configuration to use.

In scatter-gather mode, the alternate descriptors are located in one contiguous memory area. Each of the alternate descriptors must be fully configured prior to starting the scatter-gather DMA cycle.

The DMA cycle is not activated by this function. See DMA_ActivateScatterGather() to activate the DMA cycle. In some cases, the alternate configuration only has to be done once and all subsequent transfers may be activated with the activate function.

Notice that the DMA channel must also be configured, see DMA_CfgChannel().

DMA_ChannelEnable

> void DMA_ChannelEnable (unsigned int channel, bool enable)

Enable or disable a DMA channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to enable or disable.
bool	[in]	enable	If 'true', the channel will be enabled. If 'false', the channel will be disabled.

Use this function to explicitly enable or disable a DMA channel. A DMA channel is automatically disabled when the DMA controller has finished a transaction.

DMA_ChannelEnabled

> bool DMA_ChannelEnabled (unsigned int channel)

Check if the DMA channel is enabled.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to check.

The DMA channel is disabled when the DMA controller has finished a DMA cycle.

Returns

- True if the channel is enabled, false if not.

DMA_ChannelRequestEnable

> void DMA_ChannelRequestEnable (unsigned int channel, bool enable)

Enable or disable a DMA channel request.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to enable or disable the request on.
bool	[in]	enable	If 'true', the request will be enabled. If 'false', the request will be disabled.

Use this function to enable or disable a DMA channel request. This will prevent the DMA from proceeding after its current transaction if disabled.

DMA_Init

> void DMA_Init (DMA_Init_TypeDef * init)

Initialize the DMA controller.

Parameters

Type	Direction	Argument Name	Description
DMA_Init_TypeDef *	[in]	init	A pointer to a structure containing the DMA initialization information.

This function resets and prepares the DMA controller for use. Although it may be used several times, it is normally only used during system initialization. If reused during a normal operation, any ongoing DMA transfers will be aborted. When complete, the DMA controller is in an enabled state.

Note

- Must be invoked before using the DMA controller.

DMA_IRQHandler

```
void DMA_IRQHandler (void )
```

Interrupt handler for the DMA cycle completion handling.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Clears any pending flags and calls registered callback (if any).

If using the default interrupt vector table setup provided, this function is automatically placed in the IRQ table due to weak linking. If taking control over the interrupt vector table in some other way, this interrupt handler must be installed to support callback actions.

For the user to implement a custom IRQ handler or run without a DMA IRQ handler, define EXCLUDE_DEFAULT_DMA_IRQ_HANDLER with a #define statement or with the compiler option -D.

DMA_RefreshPingPong

```
void DMA_RefreshPingPong (unsigned int channel, bool primary, bool useBurst, void * dst, const void * src, unsigned int nMinus1, bool stop)
```

Refresh a descriptor used in a DMA ping-pong cycle.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	channel	The DMA channel to refresh the ping-pong descriptor for.
bool	[in]	primary	• true - refresh the primary descriptor • false - refresh an alternate descriptor
bool	[in]	useBurst	The burst feature is only used on peripherals supporting DMA bursts. Bursts must not be used if the total length (as given by nMinus1) is less than the arbitration rate configured for the descriptor. See the reference manual for more details on burst usage.

Type	Direction	Argument Name	Description
void *	[in]	dst	An address to a start location to transfer data to. If NULL, leave setting in descriptor as is.
const void *	[in]	src	An address to a start location to transfer data from. If NULL, leave setting in descriptor as is.
unsigned int	[in]	nMinus1	A number of DMA transfer elements (minus 1) to transfer (<= 1023). The size of the DMA transfer element (1, 2 or 4 bytes) is configured with DMA_CfgDescr().
bool	[in]	stop	Indicate that the DMA ping-pong cycle stops when done using this descriptor.

During a ping-pong DMA cycle, the DMA controller automatically alternates between the primary and alternate descriptors, when completing use of a descriptor. While the other descriptor is in use by the DMA controller, the software should refresh the completed descriptor. This is typically done from the callback defined for the ping-pong cycle.

DMA_Reset

```
void DMA_Reset (void )
```

Reset the DMA controller.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This functions will disable the DMA controller and set it to a reset state.

Note

- Note that any ongoing transfers will be aborted.

DMA_IntClear

```
void DMA_IntClear (uint32_t flags)
```

Clear one or more pending DMA interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending DMA interrupt sources to clear. Use one or more valid interrupt flags for the DMA module (DMA_IFC_nnn).

DMA_IntDisable

```
void DMA_IntDisable (uint32_t flags)
```

Disable one or more DMA interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	DMA interrupt sources to disable. Use one or more valid interrupt flags for the DMA module (DMA_IEN_nnn).

DMA_IntEnable

```
void DMA_IntEnable (uint32_t flags)
```

Enable one or more DMA interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	DMA interrupt sources to enable. Use one or more valid interrupt flags for the DMA module (DMA_IEN_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using DMA_IntClear() prior to enabling the interrupt.

DMA_IntGet

```
uint32_t DMA_IntGet (void )
```

Get pending DMA interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Event bits are not cleared by the use of this function.

Returns

- DMA interrupt sources pending. Returns one or more valid interrupt flags for the DMA module (DMA_IF_nnn).

DMA_IntGetEnabled

```
uint32_t DMA_IntGetEnabled (void )
```

Get enabled and pending DMA interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled DMA interrupt sources Return value is the bitwise AND of
  - the enabled interrupt sources in DMA_IEN and
  - the pending interrupt flags DMA_IF.

DMA_IntSet

```
void DMA_IntSet (uint32_t flags)
```

Set one or more pending DMA interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	DMA interrupt sources to set to pending. Use one or more valid interrupt flags for the DMA module (DMA_IFS_nnn).

DMA_CB_TypeDef

Callback structure that can be used to define DMA complete actions.

A reference to this structure is only stored in the primary descriptor for a channel (if using callback feature). If callback is required for both primary and alternate descriptor completion, this must be handled by one common callback, using the provided 'primary' parameter with the callback function.

Public Attributes

DMA_FuncPtr_TypeDef	**cbFunc** Pointer to callback function to invoke when DMA transfer cycle is done.
void *	**userPtr** User defined pointer to provide with callback function.
uint8_t	**primary** For internal use only: Indicates if next callback applies to primary or alternate descriptor completion.

Public Attribute Documentation

cbFunc

```
DMA_FuncPtr_TypeDef DMA_CB_TypeDef::cbFunc
```

Pointer to callback function to invoke when DMA transfer cycle is done.

Notice that this function is invoked in interrupt context, and therefore should be short and non-blocking.

userPtr

```
void* DMA_CB_TypeDef::userPtr
```

User defined pointer to provide with callback function.

primary

```
uint8_t DMA_CB_TypeDef::primary
```

For internal use only: Indicates if next callback applies to primary or alternate descriptor completion.

Mainly useful for ping-pong DMA cycles. Set this value to 0 prior to configuring callback handling.

DMA_CfgChannel_TypeDef

Configuration structure for a channel.

Public Attributes

bool	highPri Select if channel priority is in the high or default priority group with respect to arbitration.
bool	enableInt Select if interrupt will be enabled for channel (triggering interrupt handler when dma_done signal is asserted).
uint32_t	select Channel control specifying the source of DMA signals.
DMA_CB_TypeDef *	cb User definable callback handling configuration.

Public Attribute Documentation

highPri

```
bool DMA_CfgChannel_TypeDef::highPri
```

Select if channel priority is in the high or default priority group with respect to arbitration.

Within a priority group, lower numbered channels have higher priority than higher numbered channels.

enableInt

```
bool DMA_CfgChannel_TypeDef::enableInt
```

Select if interrupt will be enabled for channel (triggering interrupt handler when dma_done signal is asserted).

It should normally be enabled if using the callback feature for a channel, and disabled if not using the callback feature.

select

```
uint32_t DMA_CfgChannel_TypeDef::select
```

Channel control specifying the source of DMA signals.

If accessing peripherals, use one of the DMAREQ_nnn defines available for the peripheral. Set to 0 for memory-to-memory DMA cycles.

cb

```
DMA_CB_TypeDef* DMA_CfgChannel_TypeDef::cb
```

User definable callback handling configuration.

Refer to structure definition for details. The callback is invoked when specified DMA cycle is complete (when dma_done signal asserted). Callback is invoked in interrupt context, and should be efficient and non-blocking. Set to NULL to not use the callback feature. Note

- Referenced structure is used by the interrupt handler, and must be available until no longer used. Thus, in most cases it should not be located on the stack.

DMA_CfgDescr_TypeDef

Configuration structure for primary or alternate descriptor (not used for scatter-gather DMA cycles).

Public Attributes

DMA_DataInc_TypeDef	**dstInc** Destination increment size for each DMA transfer.
DMA_DataInc_TypeDef	**srcInc** Source increment size for each DMA transfer.
DMA_DataSize_TypeDef	**size** DMA transfer unit size.
DMA_ArbiterConfig_TypeDef	**arbRate** Arbitration rate, i.e., number of DMA transfers done before re-arbitration takes place.
uint8_t	**hprot** HPROT signal state, refer to reference manual, DMA chapter for further details.

Public Attribute Documentation

dstInc

```
DMA_DataInc_TypeDef DMA_CfgDescr_TypeDef::dstInc
```

Destination increment size for each DMA transfer.

srcInc

```
DMA_DataInc_TypeDef DMA_CfgDescr_TypeDef::srcInc
```

Source increment size for each DMA transfer.

size

```
DMA_DataSize_TypeDef DMA_CfgDescr_TypeDef::size
```

DMA transfer unit size.

arbRate

```
DMA_ArbiterConfig_TypeDef DMA_CfgDescr_TypeDef::arbRate
```

Arbitration rate, i.e., number of DMA transfers done before re-arbitration takes place.

hprot

> uint8_t DMA_CfgDescr_TypeDef::hprot

HPROT signal state, refer to reference manual, DMA chapter for further details.

Normally set to 0 if protection is not an issue. The following bits are available:

- bit 0 - HPROT[1] control for source read accesses, privileged/non-privileged access.
- bit 3 - HPROT[1] control for destination write accesses, privileged/non-privileged access.

DMA_CfgLoop_TypeDef

Configuration structure for loop mode.

Public Attributes

bool	enable Enable repeated loop.
uint16_t	nMinus1 Width of transfer, reload value for nMinus1.

Public Attribute Documentation

enable

```
bool DMA_CfgLoop_TypeDef::enable
```

Enable repeated loop.

nMinus1

```
uint16_t DMA_CfgLoop_TypeDef::nMinus1
```

Width of transfer, reload value for nMinus1.

DMA_CfgRect_TypeDef

Configuration structure for rectangular copy.

Public Attributes

uint16_t	dstStride DMA channel destination stride (width of destination image, distance between lines).
uint16_t	srcStride DMA channel source stride (width of source image, distance between lines).
uint16_t	height 2D copy height.

Public Attribute Documentation

dstStride

```
uint16_t DMA_CfgRect_TypeDef::dstStride
```

DMA channel destination stride (width of destination image, distance between lines).

srcStride

```
uint16_t DMA_CfgRect_TypeDef::srcStride
```

DMA channel source stride (width of source image, distance between lines).

height

```
uint16_t DMA_CfgRect_TypeDef::height
```

2D copy height.

DMA_CfgDescrSGAlt_TypeDef

Configuration structure for alternate scatter-gather descriptor.

Public Attributes

void *	**src** Pointer to location to transfer data from.
void *	**dst** Pointer to location to transfer data to.
DMA_DataInc_TypeDef	**dstInc** Destination increment size for each DMA transfer.
DMA_DataInc_TypeDef	**srcInc** Source increment size for each DMA transfer.
DMA_DataSize_TypeDef	**size** DMA transfer unit size.
DMA_ArbiterConfig_TypeDef	**arbRate** Arbitration rate, i.e., number of DMA transfers done before re-arbitration takes place.
uint16_t	**nMinus1** Number of DMA transfers minus 1 to do.
uint8_t	**hprot** HPROT signal state, refer to reference manual, DMA chapter for further details.
bool	**peripheral** Specify if a memory or peripheral scatter-gather DMA cycle.

Public Attribute Documentation

src

```
void* DMA_CfgDescrSGAlt_TypeDef::src
```

Pointer to location to transfer data from.

dst

```
void* DMA_CfgDescrSGAlt_TypeDef::dst
```

Pointer to location to transfer data to.

dstInc

```
DMA_DataInc_TypeDef DMA_CfgDescrSGAlt_TypeDef::dstInc
```

Destination increment size for each DMA transfer.

srcInc

DMA_DataInc_TypeDef DMA_CfgDescrSGAlt_TypeDef::srcInc

Source increment size for each DMA transfer.

size

DMA_DataSize_TypeDef DMA_CfgDescrSGAlt_TypeDef::size

DMA transfer unit size.

arbRate

DMA_ArbiterConfig_TypeDef DMA_CfgDescrSGAlt_TypeDef::arbRate

Arbitration rate, i.e., number of DMA transfers done before re-arbitration takes place.

nMinus1

uint16_t DMA_CfgDescrSGAlt_TypeDef::nMinus1

Number of DMA transfers minus 1 to do.

Must be <= 1023.

hprot

uint8_t DMA_CfgDescrSGAlt_TypeDef::hprot

HPROT signal state, refer to reference manual, DMA chapter for further details.

Normally set to 0 if protection is not an issue. The following bits are available:

- bit 0 - HPROT[1] control for source read accesses, privileged/non-privileged access.
- bit 3 - HPROT[1] control for destination write accesses, privileged/non-privileged access.

peripheral

bool DMA_CfgDescrSGAlt_TypeDef::peripheral

Specify if a memory or peripheral scatter-gather DMA cycle.

Notice that this parameter should be the same for all alternate descriptors.

- true - this is a peripheral scatter-gather cycle.
- false - this is a memory scatter-gather cycle.

DMA_Init_TypeDef

DMA initialization structure.

Public Attributes

uint8_t	hprot HPROT signal state when accessing the primary/alternate descriptors.
DMA_DESCRIPTOR_TypeDef *	controlBlock Pointer to the control block in memory holding descriptors (channel control data structures).

Public Attribute Documentation

hprot

```
uint8_t DMA_Init_TypeDef::hprot
```

HPROT signal state when accessing the primary/alternate descriptors.

Normally set to 0 if protection is not an issue. The following bits are available:

- bit 0 - HPROT[1] control for descriptor accesses (i.e., when the DMA controller accesses the channel control block itself), privileged/non-privileged access.

controlBlock

```
DMA_DESCRIPTOR_TypeDef* DMA_Init_TypeDef::controlBlock
```

Pointer to the control block in memory holding descriptors (channel control data structures).

This memory must be properly aligned at a 256 bytes, i.e., the 8 least significant bits must be zero.

Refer to the reference manual, DMA chapter for more details.

It is possible to provide a smaller memory block, only covering those channels actually used, if not all available channels are used. For instance, if only using 4 channels (0-3), both primary and alternate structures, then only 16*2*4 = 128 bytes must be provided. However, this implementation has no check if later exceeding such a limit by configuring for instance channel 4, in which case memory overwrite of some other data will occur.

EMU - Energy Management Unit

EMU - Energy Management Unit

Energy Management Unit (EMU) Peripheral API.

This module contains functions to control the EMU peripheral of Silicon Labs 32-bit MCUs and SoCs. The EMU handles the different low energy modes in Silicon Labs microcontrollers.

Modules

EMU_EM23Init_TypeDef

EMU_EM4Init_TypeDef

EMU_BUPDInit_TypeDef

Enumerations

enum EMU_EM4Osc_TypeDef {
  emuEM4Osc_ULFRCO = EMU_EM4CONF_OSC_ULFRCO
  emuEM4Osc_LFXO = EMU_EM4CONF_OSC_LFXO
  emuEM4Osc_LFRCO = EMU_EM4CONF_OSC_LFRCO
}
EM4 duty oscillator.

enum EMU_Probe_TypeDef {
  emuProbe_Disable = EMU_BUCTRL_PROBE_DISABLE
  emuProbe_VDDDReg = EMU_BUCTRL_PROBE_VDDDREG
  emuProbe_BUIN = EMU_BUCTRL_PROBE_BUIN
  emuProbe_BUOUT = EMU_BUCTRL_PROBE_BUOUT
}
Backup Power Voltage Probe types.

enum EMU_Resistor_TypeDef {
  emuRes_Res0 = EMU_PWRCONF_PWRRES_RES0
  emuRes_Res1 = EMU_PWRCONF_PWRRES_RES1
  emuRes_Res2 = EMU_PWRCONF_PWRRES_RES2
  emuRes_Res3 = EMU_PWRCONF_PWRRES_RES3
}
Backup Power Domain resistor selection.

enum EMU_Power_TypeDef {
  emuPower_None = EMU_BUINACT_PWRCON_NONE
  emuPower_BUMain = EMU_BUINACT_PWRCON_BUMAIN
  emuPower_MainBU = EMU_BUINACT_PWRCON_MAINBU
  emuPower_NoDiode = EMU_BUINACT_PWRCON_NODIODE
}
Backup Power Domain power connection.

enum EMU_BODMode_TypeDef {
  emuBODMode_Active
  emuBODMode_Inactive
}
BOD threshold setting selector, active or inactive mode.

enum EMU_EM4State_TypeDef {
  emuEM4Shutoff = 0
  emuEM4Hibernate = 1
}
EM4 modes.

enum EMU_PowerConfig_TypeDef {
  emuPowerConfig_DcdcToDvdd
}
Power configurations.

Functions

void EMU_EM23Init(const EMU_EM23Init_TypeDef *em23Init)
Update the EMU module with Energy Mode 2 and 3 configuration.

void EMU_EM23PresleepHook(void)
Energy mode 2/3 pre-sleep hook function.

void EMU_EM23PostsleepHook(void)
Energy mode 2/3 post-sleep hook function.

void EMU_EFPEM23PresleepHook(void)
EFP's Energy mode 2/3 pre-sleep hook function.

void EMU_EFPEM23PostsleepHook(void)
EFP's Energy mode 2/3 post-sleep hook function.

void EMU_EnterEM2(bool restore)
Enter energy mode 2 (EM2).

void EMU_EnterEM3(bool restore)
Enter energy mode 3 (EM3).

void EMU_Save(void)
Save the CMU HF clock select state, oscillator enable, and voltage scaling (if available) before EMU_EnterEM2() or EMU_EnterEM3() are called with the restore parameter set to false.

void EMU_Restore(void)
Restore CMU HF clock select state, oscillator enable, and voltage scaling (if available) after EMU_EnterEM2() or EMU_EnterEM3() are called with the restore parameter set to false.

void EMU_EM4Init(const EMU_EM4Init_TypeDef *em4Init)
Update the EMU module with Energy Mode 4 configuration.

void EMU_EM4PresleepHook(void)
Energy mode 4 pre-sleep hook function.

void EMU_EFPEM4PresleepHook(void)
EFP's Energy mode 4 pre-sleep hook function.

void EMU_EnterEM4(void)
Enter energy mode 4 (EM4).

void EMU_EnterEM4Wait(void)
Enter energy mode 4 (EM4).

void EMU_MemPwrDown(uint32_t blocks)
Power down memory block.

void EMU_RamPowerDown(uint32_t start, uint32_t end)
Power down RAM memory blocks.

void EMU_RamPowerUp(void)
Power up all available RAM memory blocks.

void EMU_UpdateOscConfig(void)
Update EMU module with CMU oscillator selection/enable status.

void EMU_BUPDInit(const EMU_BUPDInit_TypeDef *bupdInit)
Configure Backup Power Domain settings.

void EMU_BUThresholdSet(EMU_BODMode_TypeDef mode, uint32_t value)
Configure the Backup Power Domain BOD Threshold value.

void EMU_BUThresRangeSet(EMU_BODMode_TypeDef mode, uint32_t value)
Configure the Backup Power Domain BOD Threshold Range.

void EMU_BUStatEnSet(bool enable)
Enable backup mode status export.

void EMU_BUEnableSet(bool enable)
Enable backup mode.

void EMU_EnterEM1(void)
Enter energy mode 1 (EM1).

void EMU_IntClear(uint32_t flags)
Clear one or more pending EMU interrupts.

void EMU_IntDisable(uint32_t flags)
Disable one or more EMU interrupts.

void EMU_IntEnable(uint32_t flags)
Enable one or more EMU interrupts.

uint32_t EMU_IntGet(void)
Get pending EMU interrupt flags.

uint32_t EMU_IntGetEnabled(void)
Get enabled and pending EMU interrupt flags.

void EMU_IntSet(uint32_t flags)
Set one or more pending EMU interrupts.

void EMU_EM4Lock(bool enable)
Enable or disable EM4 lock configuration.

void EMU_BUReady(void)
Halts until backup power functionality is ready.

void EMU_BUPinEnable(bool enable)
Disable BU_VIN support.

void EMU_Lock(void)
Lock EMU registers in order to protect them against unintended modification.

void EMU_Unlock(void)
Unlock the EMU so that writing to locked registers again is possible.

void EMU_EM2Block(void)
Block entering EM2 or higher number energy modes.

void EMU_EM2UnBlock(void)
Unblock entering EM2 or higher number energy modes.

Macros

#define EMU_EM23INIT_DEFAULT undefined
Default initialization of EM2 and 3 configuration.

#define EMU_EM4INIT_DEFAULT undefined
Default initialization of EM4 configuration (Series 0).

#define EMU_BUPDINIT_DEFAULT undefined
Default Backup Power Domain configuration.

Enumeration Documentation

EMU_EM4Osc_TypeDef

> EMU_EM4Osc_TypeDef

EM4 duty oscillator.

Enumerator	
emuEM4Osc_ULFRCO	Select ULFRCO as duty oscillator in EM4.
emuEM4Osc_LFXO	Select LFXO as duty oscillator in EM4.
emuEM4Osc_LFRCO	Select LFRCO as duty oscillator in EM4.

EMU_Probe_TypeDef

> EMU_Probe_TypeDef

Backup Power Voltage Probe types.

Enumerator	
emuProbe_Disable	Disable voltage probe.
emuProbe_VDDDReg	Connect probe to VDD_DREG.
emuProbe_BUIN	Connect probe to BU_IN.
emuProbe_BUOUT	Connect probe to BU_OUT.

EMU_Resistor_TypeDef

> EMU_Resistor_TypeDef

Backup Power Domain resistor selection.

Enumerator	
emuRes_Res0	Main power and backup power connected with RES0 series resistance.
emuRes_Res1	Main power and backup power connected with RES1 series resistance.
emuRes_Res2	Main power and backup power connected with RES2 series resistance.
emuRes_Res3	Main power and backup power connected with RES3 series resistance.

EMU_Power_TypeDef

> EMU_Power_TypeDef

Backup Power Domain power connection.

Enumerator	
emuPower_None	No connection between main and backup power.
emuPower_BUMain	Main power and backup power connected through diode, allowing current from backup to main only.
emuPower_MainBU	Main power and backup power connected through diode, allowing current from main to backup only.

emuPower_NoDiode	Main power and backup power connected without diode.

EMU_BODMode_TypeDef

EMU_BODMode_TypeDef

BOD threshold setting selector, active or inactive mode.

Enumerator	
emuBODMode_Active	Configure BOD threshold for active mode.
emuBODMode_Inactive	Configure BOD threshold for inactive mode.

EMU_EM4State_TypeDef

EMU_EM4State_TypeDef

EM4 modes.

Enumerator	
emuEM4Shutoff	EM4 Shutoff.
emuEM4Hibernate	EM4 Hibernate.

EMU_PowerConfig_TypeDef

EMU_PowerConfig_TypeDef

Power configurations.

DCDC-to-DVDD is currently the only supported mode.

Enumerator	
emuPowerConfig_DcdcToDvdd	DCDC is connected to DVDD.

Function Documentation

EMU_EM23Init

void EMU_EM23Init (const EMU_EM23Init_TypeDef * em23Init)

Update the EMU module with Energy Mode 2 and 3 configuration.

Parameters

Type	Direction	Argument Name	Description
const EMU_EM23Init_TypeDef *	[in]	em23Init	Energy Mode 2 and 3 configuration structure.

EMU_EM23PresleepHook

void EMU_EM23PresleepHook (void)

Energy mode 2/3 pre-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is called by EMU_EnterEM2() and EMU_EnterEM3() functions just prior to execution of the WFI instruction. The function implementation does not perform anything, but it is SL_WEAK so that it can be re-implemented in application code if actions are needed.

EMU_EM23PostsleepHook

```
void EMU_EM23PostsleepHook (void )
```

Energy mode 2/3 post-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is called by EMU_EnterEM2() and EMU_EnterEM3() functions just after wakeup from the WFI instruction. The function implementation does not perform anything, but it is SL_WEAK so that it can be re-implemented in application code if actions are needed.

EMU_EFPEM23PresleepHook

```
void EMU_EFPEM23PresleepHook (void )
```

EFP's Energy mode 2/3 pre-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is similar to EMU_EM23PresleepHook() but is reserved for EFP usage.

Note

- The function is primarily meant to be used in systems with EFP circuitry. (EFP = Energy Friendly Pmic (PMIC = Power Management IC)). In such systems there is a need to drive certain signals to EFP pins to notify about energy mode transitions.

EMU_EFPEM23PostsleepHook

```
void EMU_EFPEM23PostsleepHook (void )
```

EFP's Energy mode 2/3 post-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is similar to EMU_EM23PostsleepHook() but is reserved for EFP usage.

Note

- The function is primarily meant to be used in systems with EFP circuitry. (EFP = Energy Friendly Pmic (PMIC = Power Management IC)). In such systems there is a need to drive certain signals to EFP pins to notify about energy mode transitions.

EMU_EnterEM2

```
void EMU_EnterEM2 (bool restore)
```

Enter energy mode 2 (EM2).

Parameters

Type	Direction	Argument Name	Description
bool	[in]	restore	• true - save and restore oscillators, clocks and voltage scaling, see function details. • false - do not save and restore oscillators and clocks, see function details.

When entering EM2, high-frequency clocks are disabled, i.e., HFXO, HFRCO and AUXHFRCO (for AUXHFRCO, see exception note below). When re-entering EM0, HFRCO is re-enabled and the core will be clocked by the configured HFRCO band. This ensures a quick wakeup from EM2.

However, prior to entering EM2, the core may have been using another oscillator than HFRCO. The `restore` parameter gives the user the option to restore all HF oscillators according to state prior to entering EM2, as well as the clock used to clock the core. This restore procedure is handled by SW. However, since handled by SW, it will not be restored before completing the interrupt function(s) waking up the core!

Note

- If restoring core clock to use the HFXO oscillator, which has been disabled during EM2 mode, this function will stall until the oscillator has stabilized. Stalling time can be reduced by adding interrupt support detecting stable oscillator, and an asynchronous switch to the original oscillator. See CMU documentation. Such a feature is however outside the scope of the implementation in this function.
- If ERRATA_FIX_EMU_E110_ENABLE is active, the core's SLEEPONEXIT feature can not be used.
- This function is incompatible with the Power Manager module. When the Power Manager module is present, it must be the one deciding at which EM level the device sleeps to ensure the application properly works. Using both at the same time could lead to undefined behavior in the application.
- If HFXO is re-enabled by this function, and NOT used to clock the core, this function will not wait for HFXO to stabilize. This must be considered by the application if trying to use features relying on that oscillator upon return.
- If a debugger is attached, the AUXHFRCO will not be disabled if enabled upon entering EM2. It will thus remain enabled when returning to EM0 regardless of the `restore` parameter.
- If HFXO autostart and select is enabled by using CMU_HFXOAutostartEnable(), the automatic starting and selecting of the core clocks will be done, regardless of the `restore` parameter, when waking up on the wakeup sources corresponding to the autostart and select setting.
- If voltage scaling is supported, the restore parameter is true and the EM0 voltage scaling level is set higher than the EM2 level, then the EM0 level is also restored.
- On Series 2 Config 2 devices (EFRxG22), this function will also relock the DPLL if the DPLL is used and `restore` is true.

Note that the hardware will automatically update the HFRCO frequency in the case where voltage scaling is used in EM2/EM3 and not in EM0/EM1. When the restore argument to this function is true then software will restore the original HFRCO frequency after EM2/EM3 wake up. If the restore argument is false then the HFRCO frequency is 19 MHz when coming out of EM2/EM3 and all wait states are at a safe value.

The `restore` option should only be used if all clock control is done via the CMU API.

EMU_EnterEM3

```
void EMU_EnterEM3 (bool restore)
```

Enter energy mode 3 (EM3).

Parameters

Type	Direction	Argument Name	Description
bool	[in]	restore	• true - save and restore oscillators, clocks and voltage scaling, see function details. • false - do not save and restore oscillators and clocks, see function details.

When entering EM3, the high-frequency clocks are disabled by hardware, i.e., HFXO, HFRCO, and AUXHFRCO (for AUXHFRCO, see exception note below). In addition, the low-frequency clocks, i.e., LFXO and LFRCO are disabled by software. When re-entering EM0, HFRCO is re-enabled and the core will be clocked by the configured HFRCO band. This ensures a quick wakeup from EM3.

However, prior to entering EM3, the core may have been using an oscillator other than HFRCO. The `restore` parameter gives the user the option to restore all HF/LF oscillators according to state prior to entering EM3, as well as the clock used to clock the core. This restore procedure is handled by software. However, since it is handled by software, it will not be restored before completing the interrupt function(s) waking up the core!

Note

- If restoring core clock to use an oscillator other than HFRCO, this function will stall until the oscillator has stabilized. Stalling time can be reduced by adding interrupt support detecting stable oscillator, and an asynchronous switch to the original oscillator. See CMU documentation. This feature is, however, outside the scope of the implementation in this function.
- If ERRATA_FIX_EMU_E110_ENABLE is active, the core's SLEEPONEXIT feature can't be used.
- This function is incompatible with the Power Manager module. When the Power Manager module is present, it must be the one deciding at which EM level the device sleeps to ensure the application properly works. Using both at the same time could lead to undefined behavior in the application.
- If HFXO/LFXO/LFRCO are re-enabled by this function, and NOT used to clock the core, this function will not wait for those oscillators to stabilize. This must be considered by the application if trying to use features relying on those oscillators upon return.
- If a debugger is attached, the AUXHFRCO will not be disabled if enabled upon entering EM3. It will, therefore, remain enabled when returning to EM0 regardless of the `restore` parameter.
- If voltage scaling is supported, the restore parameter is true and the EM0 voltage scaling level is set higher than the EM3 level, then the EM0 level is also restored.
- On Series 2 Config 2 devices (EFRxG22), this function will also relock the DPLL if the DPLL is used and `restore` is true.
- The `restore` option should only be used if all clock control is done via the CMU API.

EMU_Save

```
void EMU_Save (void )
```

Save the CMU HF clock select state, oscillator enable, and voltage scaling (if available) before EMU_EnterEM2() or EMU_EnterEM3() are called with the restore parameter set to false.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Calling this function is equivalent to calling EMU_EnterEM2() or EMU_EnterEM3() with the restore parameter set to true, but it allows the state to be saved without going to sleep. The state can be restored manually by calling EMU_Restore().

EMU_Restore

```
void EMU_Restore (void )
```

Restore CMU HF clock select state, oscillator enable, and voltage scaling (if available) after EMU_EnterEM2() or EMU_EnterEM3() are called with the restore parameter set to false.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Calling this function is equivalent to calling EMU_EnterEM2() or EMU_EnterEM3() with the restore parameter set to true, but it allows the application to evaluate the wakeup reason before restoring state.

EMU_EM4Init

```
void EMU_EM4Init (const EMU_EM4Init_TypeDef * em4Init)
```

Update the EMU module with Energy Mode 4 configuration.

Parameters

Type	Direction	Argument Name	Description
const EMU_EM4Init_TypeDef *	[in]	em4Init	Energy Mode 4 configuration structure.

EMU_EM4PresleepHook

```
void EMU_EM4PresleepHook (void )
```

Energy mode 4 pre-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is called by EMU_EnterEM4() just prior to the sequence of writes to put the device in EM4. The function implementation does not perform anything, but it is SL_WEAK so that it can be re-implemented in application code if actions are needed.

EMU_EFPEM4PresleepHook

```
void EMU_EFPEM4PresleepHook (void )
```

EFP's Energy mode 4 pre-sleep hook function.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function is similar to EMU_EM4PresleepHook() but is reserved for EFP usage.

Note

- The function is primarily meant to be used in systems with EFP circuitry. (EFP = Energy Friendly Pmic (PMIC = Power Management IC)). In such systems there is a need to drive certain signals to EFP pins to notify about energy mode transitions.

EMU_EnterEM4

```
__NO_RETURN void EMU_EnterEM4 (void )
```

Enter energy mode 4 (EM4).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function never returns. It waits after the EM4 entry request to make sure the CPU is properly shutdown by calling WFI.

Note

- Only a power on reset or external reset pin can wake the device from EM4.

EMU_EnterEM4Wait

```
void EMU_EnterEM4Wait (void )
```

Enter energy mode 4 (EM4).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function never returns. It waits after the EM4 entry request to make sure the CPU is properly shutdown by calling WFI.

Note

- Only a power on reset or external reset pin can wake the device from EM4.

EMU_MemPwrDown

```
void EMU_MemPwrDown (uint32_t blocks)
```

Power down memory block.

Parameters

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
uint32_t	[in]	blocks	Specifies a logical OR of bits indicating memory blocks to power down. Bit 0 selects block 1, bit 1 selects block 2, and so on. Memory block 0 cannot be disabled. See the reference manual for available memory blocks for a device.

Note

- Only a POR reset can power up the specified memory block(s) after power down.

EMU_RamPowerDown

```
void EMU_RamPowerDown (uint32_t start, uint32_t end)
```

Power down RAM memory blocks.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	start	The start address of the RAM region to power down. This address is inclusive.
uint32_t	[in]	end	The end address of the RAM region to power down. This address is exclusive. If this parameter is 0, all RAM blocks contained in the region from start to the upper RAM address will be powered down.

This function will power down all the RAM blocks that are within a given range. The RAM block layout is different between device families, so this function can be used in a generic way to power down a RAM memory region which is known to be unused.

This function will only power down blocks which are completely enclosed by the memory range given by [start, end).

This is an example to power down all RAM blocks except the first one. The first RAM block is special in that it cannot be powered down by the hardware. The size of the first RAM block is device-specific. See the reference manual to find the RAM block sizes.

```
EMU_RamPowerDown(SRAM_BASE, SRAM_BASE + SRAM_SIZE);
```

Note

- Only a reset can power up the specified memory block(s) after power down on a series 0 device. The specified memory block(s) will stay off until a call to EMU_RamPowerUp() is done on series 1/2.

EMU_RamPowerUp

```
void EMU_RamPowerUp (void )
```

Power up all available RAM memory blocks.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function will power up all the RAM blocks on a device, this means that the RAM blocks are retained in EM2/EM3. Note that this functionality is not supported on Series 0 devices. Only a reset will power up the RAM blocks on a series 0 device.

EMU_UpdateOscConfig

```
void EMU_UpdateOscConfig (void )
```

Update EMU module with CMU oscillator selection/enable status.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

EMU_BUPDInit

```
void EMU_BUPDInit (const EMU_BUPDInit_TypeDef * bupdInit)
```

Configure Backup Power Domain settings.

Parameters

Type	Direction	Argument Name	Description
const EMU_BUPDInit_TypeDef *	[in]	bupdInit	Backup power domain initialization structure.

EMU_BUThresholdSet

```
void EMU_BUThresholdSet (EMU_BODMode_TypeDef mode, uint32_t value)
```

Configure the Backup Power Domain BOD Threshold value.

Parameters

Type	Direction	Argument Name	Description
EMU_BODMode_TypeDef	[in]	mode	Active or Inactive mode
uint32_t	[in]	value	

Note

- These values are precalibrated.

EMU_BUThresRangeSet

```
void EMU_BUThresRangeSet (EMU_BODMode_TypeDef mode, uint32_t value)
```

Configure the Backup Power Domain BOD Threshold Range.

Parameters

Type	Direction	Argument Name	Description
EMU_BODMode_TypeDef	[in]	mode	Active or Inactive mode
uint32_t	[in]	value	

Note

- These values are precalibrated.

EMU_BUStatEnSet

```
void EMU_BUStatEnSet (bool enable)
```

Enable backup mode status export.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	True to enable status export. False to disable.

EMU_BUEnableSet

```
void EMU_BUEnableSet (bool enable)
```

Enable backup mode.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	True to enable backup mode. False to disable.

EMU_EnterEM1

```
void EMU_EnterEM1 (void )
```

Enter energy mode 1 (EM1).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function is incompatible with the Power Manager module. When the Power Manager module is present, it must be the one deciding at which EM level the device sleeps to ensure the application properly works. Using both at the same time could lead to undefined behavior in the application.

EMU_IntClear

```
void EMU_IntClear (uint32_t flags)
```

Clear one or more pending EMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending EMU interrupt sources to clear. Use one or more valid interrupt flags for the EMU module (EMU_IFC_nnn or EMU_IF_nnn).

EMU_IntDisable

```
void EMU_IntDisable (uint32_t flags)
```

Disable one or more EMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	EMU interrupt sources to disable. Use one or more valid interrupt flags for the EMU module (EMU_IEN_nnn).

EMU_IntEnable

```
void EMU_IntEnable (uint32_t flags)
```

Enable one or more EMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	EMU interrupt sources to enable. Use one or more valid interrupt flags for the EMU module (EMU_IEN_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using EMU_IntClear() prior to enabling the interrupt.

EMU_IntGet

```
uint32_t EMU_IntGet (void )
```

Get pending EMU interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Event bits are not cleared by the use of this function.

Returns

- EMU interrupt sources pending. Returns one or more valid interrupt flags for the EMU module (EMU_IF_nnn).

EMU_IntGetEnabled

> uint32_t EMU_IntGetEnabled (void)

Get enabled and pending EMU interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled EMU interrupt sources Return value is the bitwise AND of
  - the enabled interrupt sources in EMU_IEN and
  - the pending interrupt flags EMU_IF.

EMU_IntSet

> void EMU_IntSet (uint32_t flags)

Set one or more pending EMU interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	EMU interrupt sources to set to pending. Use one or more valid interrupt flags for the EMU module (EMU_IFS_nnn).

EMU_EM4Lock

> void EMU_EM4Lock (bool enable)

Enable or disable EM4 lock configuration.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, locks down EM4 configuration.

EMU_BUReady

> void EMU_BUReady (void)

Halts until backup power functionality is ready.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

EMU_BUPinEnable

```
void EMU_BUPinEnable (bool enable)
```

Disable BU_VIN support.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, enables BU_VIN input pin support, if false disables it.

EMU_Lock

```
void EMU_Lock (void )
```

Lock EMU registers in order to protect them against unintended modification.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- If locking EMU registers, they must be unlocked prior to using any EMU API functions modifying EMU registers, excluding interrupt control and regulator control if the architecture has a EMU_PWRCTRL register. An exception to this is the energy mode entering API (EMU_EnterEMn()), which can be used when the EMU registers are locked.

EMU_Unlock

```
void EMU_Unlock (void )
```

Unlock the EMU so that writing to locked registers again is possible.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

EMU_EM2Block

```
void EMU_EM2Block (void )
```

Block entering EM2 or higher number energy modes.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

EMU_EM2UnBlock

```
void EMU_EM2UnBlock (void )
```

Unblock entering EM2 or higher number energy modes.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Macro Definition Documentation

EMU_EM23INIT_DEFAULT

```
#define EMU_EM23INIT_DEFAULT
```

Value:

```
{                                                                                \
  false,                                        /* Reduced voltage regulator drive strength in EM2/3.*/ \
}
```

Default initialization of EM2 and 3 configuration.

EMU_EM4INIT_DEFAULT

```
#define EMU_EM4INIT_DEFAULT
```

Value:

```
{                                                                \
  false,                          /* Do not lock configuration after it's been set. */ \
  false,                          /* No reset will be asserted due to BOD in EM4. */   \
  emuEM4Osc_ULFRCO,               /* Use default ULFRCO oscillator.  */                \
  true,                           /* Wake up on EM4 BURTC interrupt. */                \
  true,                           /* Enable VREG. */                                   \
}
```

Default initialization of EM4 configuration (Series 0).

EMU_BUPDINIT_DEFAULT

```
#define EMU_BUPDINIT_DEFAULT
```

Value:

```
{                                                                                \
  emuProbe_Disable, /* Do not enable voltage probe. */                           \
  false,          /* Disable BOD calibration mode. */                            \
  false,          /* Disable BU_STAT pin for backup mode indication. */          \
                                                                                 \
  emuRes_Res0,    /* RES0 series resistance between main and backup power. */    \
  false,          /* Do not enable strong switch. */                             \
  false,          /* Do not enable medium switch. */                             \
  false,          /* Do not enable weak switch. */                               \
                                                                                 \
  emuPower_None,  /* No connection between main and backup power. (inactive mode) */ \
  emuPower_None,  /* No connection between main and backup power. (active mode) */   \
  true            /* Enable BUPD enter on BOD, enable BU_VIN pin, release BU reset.  */ \
}
```

Default Backup Power Domain configuration.

EMU_EM23Init_TypeDef

EM2 and 3 initialization structure.

Public Attributes

bool	em23VregFullEn Enable full VREG drive strength in EM2/3.

Public Attribute Documentation

em23VregFullEn

```
bool EMU_EM23Init_TypeDef::em23VregFullEn
```

Enable full VREG drive strength in EM2/3.

EMU_EM4Init_TypeDef

EM4 initialization structure.

Public Attributes

bool	lockConfig Lock configuration of regulator, BOD and oscillator.
bool	buBodRstDis When set, no reset will be asserted due to Brownout when in EM4.
EMU_EM4Osc_TypeDef	osc EM4 duty oscillator.
bool	buRtcWakeup Wake up on EM4 BURTC interrupt.
bool	vreg Enable EM4 voltage regulator.

Public Attribute Documentation

lockConfig

```
bool EMU_EM4Init_TypeDef::lockConfig
```

Lock configuration of regulator, BOD and oscillator.

buBodRstDis

```
bool EMU_EM4Init_TypeDef::buBodRstDis
```

When set, no reset will be asserted due to Brownout when in EM4.

osc

```
EMU_EM4Osc_TypeDef EMU_EM4Init_TypeDef::osc
```

EM4 duty oscillator.

buRtcWakeup

```
bool EMU_EM4Init_TypeDef::buRtcWakeup
```

Wake up on EM4 BURTC interrupt.

vreg

bool EMU_EM4Init_TypeDef::vreg

Enable EM4 voltage regulator.

EMU_BUPDInit_TypeDef

Backup Power Domain Initialization structure.

Public Attributes

EMU_Probe_TypeDef	**probe** Voltage probe select, selects ADC voltage.
bool	**bodCal** Enable BOD calibration mode.
bool	**statusPinEnable** Enable BU_STAT status pin for active BU mode.
EMU_Resistor_TypeDef	**resistor** Power domain resistor.
bool	**voutStrong** BU_VOUT strong enable.
bool	**voutMed** BU_VOUT medium enable.
bool	**voutWeak** BU_VOUT weak enable.
EMU_Power_TypeDef	**inactivePower** Power connection, when not in Backup Mode.
EMU_Power_TypeDef	**activePower** Power connection, when in Backup Mode.
bool	**enable** Enable backup power domain, and release reset, enable BU_VIN pin.

Public Attribute Documentation

probe

```
EMU_Probe_TypeDef EMU_BUPDInit_TypeDef::probe
```

Voltage probe select, selects ADC voltage.

bodCal

```
bool EMU_BUPDInit_TypeDef::bodCal
```

Enable BOD calibration mode.

statusPinEnable

```
bool EMU_BUPDInit_TypeDef::statusPinEnable
```

Enable BU_STAT status pin for active BU mode.

resistor

```
EMU_Resistor_TypeDef EMU_BUPDInit_TypeDef::resistor
```

Power domain resistor.

voutStrong

```
bool EMU_BUPDInit_TypeDef::voutStrong
```

BU_VOUT strong enable.

voutMed

```
bool EMU_BUPDInit_TypeDef::voutMed
```

BU_VOUT medium enable.

voutWeak

```
bool EMU_BUPDInit_TypeDef::voutWeak
```

BU_VOUT weak enable.

inactivePower

```
EMU_Power_TypeDef EMU_BUPDInit_TypeDef::inactivePower
```

Power connection, when not in Backup Mode.

activePower

```
EMU_Power_TypeDef EMU_BUPDInit_TypeDef::activePower
```

Power connection, when in Backup Mode.

enable

```
bool EMU_BUPDInit_TypeDef::enable
```

Enable backup power domain, and release reset, enable BU_VIN pin.

GPIO - General Purpose Input/Output

GPIO - General Purpose Input/Output

General Purpose Input/Output (GPIO) API.

This module contains functions to control the GPIO peripheral of Silicon Labs 32-bit MCUs and SoCs. The GPIO peripheral is used for pin configuration and direct pin manipulation and sensing as well as routing for peripheral pin connections.

Enumerations

enum GPIO_Port_TypeDef {

gpioPortA = 0
gpioPortB = 1
gpioPortC = 2
gpioPortD = 3
gpioPortE = 4
gpioPortF = 5

}

GPIO ports IDs.

enum GPIO_DriveMode_TypeDef {

gpioDriveModeStandard = GPIO_P_CTRL_DRIVEMODE_STANDARD
gpioDriveModeLowest = GPIO_P_CTRL_DRIVEMODE_LOWEST
gpioDriveModeHigh = GPIO_P_CTRL_DRIVEMODE_HIGH
gpioDriveModeLow = GPIO_P_CTRL_DRIVEMODE_LOW

}

GPIO drive mode.

enum GPIO_Mode_TypeDef {

gpioModeDisabled = _GPIO_P_MODEL_MODE0_DISABLED
gpioModeInput = _GPIO_P_MODEL_MODE0_INPUT
gpioModeInputPull = _GPIO_P_MODEL_MODE0_INPUTPULL
gpioModeInputPullFilter = _GPIO_P_MODEL_MODE0_INPUTPULLFILTER
gpioModePushPull = _GPIO_P_MODEL_MODE0_PUSHPULL
gpioModePushPullDrive = _GPIO_P_MODEL_MODE0_PUSHPULLDRIVE
gpioModeWiredOr = _GPIO_P_MODEL_MODE0_WIREDOR
gpioModeWiredOrPullDown = _GPIO_P_MODEL_MODE0_WIREDORPULLDOWN
gpioModeWiredAnd = _GPIO_P_MODEL_MODE0_WIREDAND
gpioModeWiredAndFilter = _GPIO_P_MODEL_MODE0_WIREDANDFILTER
gpioModeWiredAndPullUp = _GPIO_P_MODEL_MODE0_WIREDANDPULLUP
gpioModeWiredAndPullUpFilter = _GPIO_P_MODEL_MODE0_WIREDANDPULLUPFILTER
gpioModeWiredAndDrive = _GPIO_P_MODEL_MODE0_WIREDANDDRIVE
gpioModeWiredAndDriveFilter = _GPIO_P_MODEL_MODE0_WIREDANDDRIVEFILTER
gpioModeWiredAndDrivePullUp = _GPIO_P_MODEL_MODE0_WIREDANDDRIVEPULLUP
gpioModeWiredAndDrivePullUpFilter = _GPIO_P_MODEL_MODE0_WIREDANDDRIVEPULLUPFILTER

}

Pin mode.

Functions

void GPIO_DbgLocationSet(unsigned int location)

Sets the pin location of the debug pins (Serial Wire interface).

void GPIO_DbgSWDClkEnable(bool enable)

Enable/disable serial wire clock pin.

void GPIO_DbgSWDIOEnable(bool enable)

Enable/disable serial wire data I/O pin.

void	GPIO_DbgSWOEnable(bool enable) Enable/Disable serial wire output pin.
void	GPIO_DriveModeSet(GPIO_Port_TypeDef port, GPIO_DriveMode_TypeDef mode) Sets drive mode for a GPIO port.
void	GPIO_EM4DisablePinWakeup(uint32_t pinmask) Disable GPIO pin wake-up from EM4.
void	GPIO_EM4EnablePinWakeup(uint32_t pinmask, uint32_t polaritymask) Enable GPIO pin wake-up from EM4.
uint32_t	GPIO_EM4GetPinWakeupCause(void) Check which GPIO pin(s) that caused a wake-up from EM4.
void	GPIO_EM4SetPinRetention(bool enable) Enable GPIO pin retention of output enable, output value, pull enable, and pull direction in EM4.
void	GPIO_ExtIntConfig(GPIO_Port_TypeDef port, unsigned int pin, unsigned int intNo, bool risingEdge, bool fallingEdge, bool enable) Configure the GPIO external pin interrupt.
void	GPIO_InputSenseSet(uint32_t val, uint32_t mask) Enable/disable input sensing.
void	GPIO_IntClear(uint32_t flags) Clear one or more pending GPIO interrupts.
void	GPIO_IntDisable(uint32_t flags) Disable one or more GPIO interrupts.
void	GPIO_IntEnable(uint32_t flags) Enable one or more GPIO interrupts.
uint32_t	GPIO_EnabledIntGet(void) Get enabled GPIO interrupts.
uint32_t	GPIO_IntGet(void) Get pending GPIO interrupts.
uint32_t	GPIO_IntGetEnabled(void) Get enabled and pending GPIO interrupt flags.
void	GPIO_IntSet(uint32_t flags) Set one or more pending GPIO interrupts from SW.
void	GPIO_Lock(void) Lock the GPIO configuration.
unsigned int	GPIO_PinInGet(GPIO_Port_TypeDef port, unsigned int pin) Read the pad value for a single pin in a GPIO port.
void	GPIO_PinLock(GPIO_Port_TypeDef port, unsigned int pin) Lock all GPIO configuration settings for a given pin.
GPIO_Mode_TypeDef	GPIO_PinModeGet(GPIO_Port_TypeDef port, unsigned int pin) Get the mode for a GPIO pin.
void	GPIO_PinModeSet(GPIO_Port_TypeDef port, unsigned int pin, GPIO_Mode_TypeDef mode, unsigned int out) Set the mode for a GPIO pin.
void	GPIO_PinOutClear(GPIO_Port_TypeDef port, unsigned int pin) Set a single pin in GPIO data out port register to 0.
unsigned int	GPIO_PinOutGet(GPIO_Port_TypeDef port, unsigned int pin) Get current setting for a pin in a GPIO port data out register.

void GPIO_PinOutSet(GPIO_Port_TypeDef port, unsigned int pin)
Set a single pin in GPIO data out register to 1.

void GPIO_PinOutToggle(GPIO_Port_TypeDef port, unsigned int pin)
Toggle a single pin in GPIO port data out register.

uint32_t GPIO_PortInGet(GPIO_Port_TypeDef port)
Read the pad values for GPIO port.

void GPIO_PortOutClear(GPIO_Port_TypeDef port, uint32_t pins)
Set bits in DOUT register for a port to 0.

uint32_t GPIO_PortOutGet(GPIO_Port_TypeDef port)
Get the current setting for a GPIO port data out register.

void GPIO_PortOutSet(GPIO_Port_TypeDef port, uint32_t pins)
Set bits GPIO data out register to 1.

void GPIO_PortOutSetVal(GPIO_Port_TypeDef port, uint32_t val, uint32_t mask)
Set GPIO port data out register.

void GPIO_PortOutToggle(GPIO_Port_TypeDef port, uint32_t pins)
Toggle pins in GPIO port data out register.

void GPIO_Unlock(void)
Unlock the GPIO configuration.

Enumeration Documentation

GPIO_Port_TypeDef

> GPIO_Port_TypeDef

GPIO ports IDs.

Enumerator	
gpioPortA	Port A.
gpioPortB	Port B.
gpioPortC	Port C.
gpioPortD	Port D.
gpioPortE	Port E.
gpioPortF	Port F.

GPIO_DriveMode_TypeDef

> GPIO_DriveMode_TypeDef

GPIO drive mode.

Enumerator	
gpioDriveModeStandard	Default 6mA.
gpioDriveModeLowest	0.5 mA.
gpioDriveModeHigh	20 mA.
gpioDriveModeLow	2 mA.

GPIO_Mode_TypeDef

GPIO_Mode_TypeDef

Pin mode.

For more details on each mode, refer to the reference manual.

Enumerator	
gpioModeDisabled	Input disabled.
gpioModeInput	Input enabled.
gpioModeInputPull	Input enabled.
gpioModeInputPullFilter	Input enabled with filter.
gpioModePushPull	Push-pull output.
gpioModePushPullDrive	Push-pull output with drive-strength set by DRIVEMODE.
gpioModeWiredOr	Wired-or output.
gpioModeWiredOrPullDown	Wired-or output with pull-down.
gpioModeWiredAnd	Open-drain output.
gpioModeWiredAndFilter	Open-drain output with filter.
gpioModeWiredAndPullUp	Open-drain output with pull-up.
gpioModeWiredAndPullUpFilter	Open-drain output with filter and pull-up.
gpioModeWiredAndDrive	Open-drain output with drive-strength set by DRIVEMODE.
gpioModeWiredAndDriveFilter	Open-drain output with filter and drive-strength set by DRIVEMODE.
gpioModeWiredAndDrivePullUp	Open-drain output with pull-up and drive-strength set by DRIVEMODE.
gpioModeWiredAndDrivePullUpFilter	Open-drain output with filter, pull-up and drive-strength set by DRIVEMODE.

Function Documentation

GPIO_DbgLocationSet

void GPIO_DbgLocationSet (unsigned int location)

Sets the pin location of the debug pins (Serial Wire interface).

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	location	The debug pin location to use (0-3).

Note

- Changing the pins used for debugging uncontrolled, may result in a lockout.

GPIO_DbgSWDClkEnable

void GPIO_DbgSWDClkEnable (bool enable)

Enable/disable serial wire clock pin.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• false - disable serial wire clock. • true - enable serial wire clock (default after reset).

Note

- Disabling SWDClk will disable the debug interface, which may result in a lockout if done early in startup (before debugger is able to halt core).

GPIO_DbgSWDIOEnable

> void GPIO_DbgSWDIOEnable (bool enable)

Enable/disable serial wire data I/O pin.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• false - disable serial wire data pin. • true - enable serial wire data pin (default after reset).

Note

- Disabling SWDClk will disable the debug interface, which may result in a lockout if done early in startup (before debugger is able to halt core).

GPIO_DbgSWOEnable

> void GPIO_DbgSWOEnable (bool enable)

Enable/Disable serial wire output pin.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• false - disable serial wire viewer pin (default after reset). • true - enable serial wire viewer pin.

Note

- Enabling this pin is not sufficient to fully enable serial wire output, which is also dependent on issues outside the GPIO module. Refer to DBG_SWOEnable().

Warnings

- If debug port is locked, SWO pin is not disabled automatically. To avoid information leakage through SWO, disable SWO pin after locking debug port.

GPIO_DriveModeSet

> void GPIO_DriveModeSet (GPIO_Port_TypeDef port, GPIO_DriveMode_TypeDef mode)

Sets drive mode for a GPIO port.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
GPIO_DriveMode_TypeDef	[in]	mode	Drive mode to use for the port.

GPIO_EM4DisablePinWakeup

```
void GPIO_EM4DisablePinWakeup (uint32_t pinmask)
```

Disable GPIO pin wake-up from EM4.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	pinmask	Bit mask containing the bitwise logic OR of which GPIO pin(s) to disable. Refer to Reference Manuals for pinmask to GPIO port/pin mapping.

GPIO_EM4EnablePinWakeup

```
void GPIO_EM4EnablePinWakeup (uint32_t pinmask, uint32_t polaritymask)
```

Enable GPIO pin wake-up from EM4.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	pinmask	A bitmask containing the bitwise logic OR of which GPIO pin(s) to enable. See Reference Manuals for a pinmask to the GPIO port/pin mapping.
uint32_t	[in]	polaritymask	A bitmask containing the bitwise logic OR of GPIO pin(s) wake-up polarity. See Reference Manuals for pinmask-to-GPIO port/pin mapping.

When the function exits, EM4 mode can be safely entered.

Note

- It is assumed that the GPIO pin modes are set correctly. Valid modes are gpioModeInput and gpioModeInputPull.

GPIO_EM4GetPinWakeupCause

```
uint32_t GPIO_EM4GetPinWakeupCause (void )
```

Check which GPIO pin(s) that caused a wake-up from EM4.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

Bit mask containing the bitwise logic OR of which GPIO pin(s) caused the wake-up. Refer to Reference Manuals for pinmask to GPIO port/pin mapping.

GPIO_EM4SetPinRetention

> void GPIO_EM4SetPinRetention (bool enable)

Enable GPIO pin retention of output enable, output value, pull enable, and pull direction in EM4.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• true - enable EM4 pin retention. • false - disable EM4 pin retention.

Note

- On series 0 devices EM4 gpio retention can either be turned on or off. On series 1 devices there are three EM4 GPIO retention modes available. These modes are "Disabled", "EM4EXIT" and "SWUNLATCH". Use the EMU_EM4Init() to configure the GPIO retention mode on a series 1 device.

The behavior of this function depends on the configured GPIO retention mode. If the GPIO retention mode is configured to be "SWUNLATCH" then this function will not change anything. If the retention mode is anything else then this function will set the GPIO retention mode to "EM4EXIT" when the enable argument is true, and "Disabled" when false.

GPIO_ExtIntConfig

> void GPIO_ExtIntConfig (GPIO_Port_TypeDef port, unsigned int pin, unsigned int intNo, bool risingEdge, bool fallingEdge, bool enable)

Configure the GPIO external pin interrupt.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The port to associate with the `pin`.
unsigned int	[in]	pin	The pin number on the port.
unsigned int	[in]	intNo	The interrupt number to trigger.
bool	[in]	risingEdge	Set to true if the interrupt will be enabled on the rising edge. Otherwise, false.
bool	[in]	fallingEdge	Set to true if the interrupt will be enabled on the falling edge. Otherwise, false.
bool	[in]	enable	Set to true if the interrupt will be enabled after the configuration is complete. False to leave disabled. See GPIO_IntDisable() and GPIO_IntEnable().

It is recommended to disable interrupts before configuring the GPIO pin interrupt. See GPIO_IntDisable() for more information.

The GPIO interrupt handler must be in place before enabling the interrupt.

Notice that any pending interrupt for the selected interrupt is cleared by this function.

Note

- On series 0 devices, the pin number parameter is not used. The pin number used on these devices is hardwired to the interrupt with the same number.
  On series 1 devices, the pin number can be selected freely within a group. Interrupt numbers are divided into 4 groups (intNo / 4) and valid pin number within the interrupt groups are: 0: pins 0-3 (interrupt number 0-3) 1: pins 4-7 (interrupt number 4-7) 2: pins 8-11 (interrupt number 8-11) 3: pins 12-15 (interrupt number 12-15)

GPIO_InputSenseSet

```
void GPIO_InputSenseSet (uint32_t val, uint32_t mask)
```

Enable/disable input sensing.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	val	Bitwise logic OR of one or more of: • GPIO_INSENSE_INT - interrupt input sensing. • GPIO_INSENSE_PRS - peripheral reflex system input sensing.
uint32_t	[in]	mask	Mask containing bitwise logic OR of bits similar as for `val` used to indicate which input sense options to disable/enable.

Disabling input sensing if not used, can save some energy consumption.

GPIO_IntClear

```
void GPIO_IntClear (uint32_t flags)
```

Clear one or more pending GPIO interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Bitwise logic OR of GPIO interrupt sources to clear.

GPIO_IntDisable

```
void GPIO_IntDisable (uint32_t flags)
```

Disable one or more GPIO interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	GPIO interrupt sources to disable.

GPIO_IntEnable

```
void GPIO_IntEnable (uint32_t flags)
```

Enable one or more GPIO interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	GPIO interrupt sources to enable.

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using GPIO_IntClear() prior to enabling the interrupt.

GPIO_EnabledIntGet

```
uint32_t GPIO_EnabledIntGet (void )
```

Get enabled GPIO interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Enabled GPIO interrupt sources.

GPIO_IntGet

```
uint32_t GPIO_IntGet (void )
```

Get pending GPIO interrupts.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- GPIO interrupt sources pending.

GPIO_IntGetEnabled

```
uint32_t GPIO_IntGetEnabled (void )
```

Get enabled and pending GPIO interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled GPIO interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in GPIO_IEN register and
  - the OR combination of valid interrupt flags in GPIO_IF register.

GPIO_IntSet

```
void GPIO_IntSet (uint32_t flags)
```

Set one or more pending GPIO interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	GPIO interrupt sources to set to pending.

GPIO_Lock

```
void GPIO_Lock (void )
```

Lock the GPIO configuration.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

GPIO_PinInGet

```
unsigned int GPIO_PinInGet (GPIO_Port_TypeDef port, unsigned int pin)
```

Read the pad value for a single pin in a GPIO port.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin number to read.

Returns

- The pin value, 0 or 1.

GPIO_PinLock

```
void GPIO_PinLock (GPIO_Port_TypeDef port, unsigned int pin)
```

Lock all GPIO configuration settings for a given pin.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin number to lock.

The lock can only be cleared by a chip reset.

GPIO_PinModeGet

> GPIO_Mode_TypeDef GPIO_PinModeGet (GPIO_Port_TypeDef port, unsigned int pin)

Get the mode for a GPIO pin.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin number in the port.

Returns

- The pin mode.

GPIO_PinModeSet

> void GPIO_PinModeSet (GPIO_Port_TypeDef port, unsigned int pin, GPIO_Mode_TypeDef mode, unsigned int out)

Set the mode for a GPIO pin.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin number in the port.
GPIO_Mode_TypeDef	[in]	mode	The desired pin mode.
unsigned int	[in]	out	A value to set for the pin in the DOUT register. The DOUT setting is important for some input mode configurations to determine the pull-up/down direction.

GPIO_PinOutClear

> void GPIO_PinOutClear (GPIO_Port_TypeDef port, unsigned int pin)

Set a single pin in GPIO data out port register to 0.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin to set.

Note

- To ensure that the setting takes effect on the output pad, the pin must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PinOutGet

> unsigned int GPIO_PinOutGet (GPIO_Port_TypeDef port, unsigned int pin)

Get current setting for a pin in a GPIO port data out register.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin to get setting for.

Returns

- The DOUT setting for the requested pin, 0 or 1.

GPIO_PinOutSet

> void GPIO_PinOutSet (GPIO_Port_TypeDef port, unsigned int pin)

Set a single pin in GPIO data out register to 1.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin to set.

Note

- To ensure that the setting takes effect on the output pad, the pin must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PinOutToggle

> void GPIO_PinOutToggle (GPIO_Port_TypeDef port, unsigned int pin)

Toggle a single pin in GPIO port data out register.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
unsigned int	[in]	pin	The pin to toggle.

Note

- To ensure that the setting takes effect on the output pad, the pin must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PortInGet

```
uint32_t GPIO_PortInGet (GPIO_Port_TypeDef port)
```

Read the pad values for GPIO port.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.

Returns

- The pad values for the GPIO port.

GPIO_PortOutClear

```
void GPIO_PortOutClear (GPIO_Port_TypeDef port, uint32_t pins)
```

Set bits in DOUT register for a port to 0.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
uint32_t	[in]	pins	Bit mask for bits to clear in DOUT register.

Note

- To ensure that the setting takes effect on the output pad, the pin must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PortOutGet

```
uint32_t GPIO_PortOutGet (GPIO_Port_TypeDef port)
```

Get the current setting for a GPIO port data out register.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.

Returns

- The data out setting for the requested port.

GPIO_PortOutSet

```
void GPIO_PortOutSet (GPIO_Port_TypeDef port, uint32_t pins)
```

Set bits GPIO data out register to 1.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
uint32_t	[in]	pins	Bit mask for bits to set to 1 in DOUT register.

Note

- To ensure that the setting takes effect on the respective output pads, the pins must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PortOutSetVal

> void GPIO_PortOutSetVal (GPIO_Port_TypeDef port, uint32_t val, uint32_t mask)

Set GPIO port data out register.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
uint32_t	[in]	val	Value to write to port data out register.
uint32_t	[in]	mask	Mask indicating which bits to modify.

Note

- To ensure that the setting takes effect on the respective output pads, the pins must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_PortOutToggle

> void GPIO_PortOutToggle (GPIO_Port_TypeDef port, uint32_t pins)

Toggle pins in GPIO port data out register.

Parameters

Type	Direction	Argument Name	Description
GPIO_Port_TypeDef	[in]	port	The GPIO port to access.
uint32_t	[in]	pins	Bit mask with pins to toggle.

Note

- To ensure that the setting takes effect on the output pad, the pin must be configured properly. If not, it will take effect whenever the pin has been properly configured.

GPIO_Unlock

> void GPIO_Unlock (void)

Unlock the GPIO configuration.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

I2C - Inter-Integrated Circuit

I2C - Inter-Integrated Circuit

Inter-integrated Circuit (I2C) Peripheral API.

This module contains functions to control the I2C peripheral of Silicon Labs 32-bit MCUs and SoCs. The I2C interface allows communication on I2C buses with the lowest energy consumption possible.

Modules

I2C_Init_TypeDef

I2C_TransferSeq_TypeDef

Enumerations

enum I2C_ClockHLR_TypeDef {

i2cClockHLRStandard = _I2C_CTRL_CLHR_STANDARD
i2cClockHLRAsymetric = _I2C_CTRL_CLHR_ASYMMETRIC
i2cClockHLRFast = _I2C_CTRL_CLHR_FAST

}
Clock low to high ratio settings.

enum I2C_TransferReturn_TypeDef {

i2cTransferInProgress = 1
i2cTransferDone = 0
i2cTransferNack = -1
i2cTransferBusErr = -2
i2cTransferArbLost = -3
i2cTransferUsageFault = -4
i2cTransferSwFault = -5

}
Return codes for single Controller mode transfer function.

Functions

uint32_t I2C_BusFreqGet(I2C_TypeDef *i2c)
Get the current configured I2C bus frequency.

void I2C_BusFreqSet(I2C_TypeDef *i2c, uint32_t freqRef, uint32_t freqScl, I2C_ClockHLR_TypeDef i2cMode)
Set the I2C bus frequency.

void I2C_Enable(I2C_TypeDef *i2c, bool enable)
Enable/disable I2C.

void I2C_Init(I2C_TypeDef *i2c, const I2C_Init_TypeDef *init)
Initialize I2C.

void I2C_Reset(I2C_TypeDef *i2c)
Reset I2C to the same state that it was in after a hardware reset.

I2C_TransferReturn_TypeDef I2C_Transfer(I2C_TypeDef *i2c)
Continue an initiated I2C transfer (single master mode only).

I2C_TransferReturn_TypeDef I2C_TransferInit(I2C_TypeDef *i2c, I2C_TransferSeq_TypeDef *seq)
Prepare and start an I2C transfer (single master mode only).

void I2C_IntClear(I2C_TypeDef *i2c, uint32_t flags)
Clear one or more pending I2C interrupts.

void I2C_IntDisable(I2C_TypeDef *i2c, uint32_t flags)
Disable one or more I2C interrupts.

void I2C_IntEnable(I2C_TypeDef *i2c, uint32_t flags)
Enable one or more I2C interrupts.

uint32_t I2C_IntGet(I2C_TypeDef *i2c)
Get pending I2C interrupt flags.

uint32_t I2C_IntGetEnabled(I2C_TypeDef *i2c)
Get enabled and pending I2C interrupt flags.

void I2C_IntSet(I2C_TypeDef *i2c, uint32_t flags)
Set one or more pending I2C interrupts from SW.

uint8_t I2C_SlaveAddressGet(I2C_TypeDef *i2c)
Get Target address used for I2C peripheral (when operating in Target mode).

void I2C_SlaveAddressSet(I2C_TypeDef *i2c, uint8_t addr)
Set Target address to use for I2C peripheral (when operating in Target mode).

uint8_t I2C_SlaveAddressMaskGet(I2C_TypeDef *i2c)
Get Target address mask used for I2C peripheral (when operating in Target mode).

void I2C_SlaveAddressMaskSet(I2C_TypeDef *i2c, uint8_t mask)
Set Target address mask used for I2C peripheral (when operating in Target mode).

Macros

#define I2C_FREQ_STANDARD_MAX 92000
Standard mode max frequency assuming using 4:4 ratio for Nlow:Nhigh.

#define I2C_FREQ_FAST_MAX 392157
Fast mode max frequency assuming using 6:3 ratio for Nlow:Nhigh.

#define I2C_FREQ_FASTPLUS_MAX 987167
Fast mode+ max frequency assuming using 11:6 ratio for Nlow:Nhigh.

#define I2C_FLAG_WRITE 0x0001
Indicate plain write sequence: S+ADDR(W)+DATA0+P.

#define I2C_FLAG_READ 0x0002
Indicate plain read sequence: S+ADDR(R)+DATA0+P.

#define I2C_FLAG_WRITE_READ 0x0004
Indicate combined write/read sequence: S+ADDR(W)+DATA0+Sr+ADDR(R)+DATA1+P.

#define I2C_FLAG_WRITE_WRITE 0x0008
Indicate write sequence using two buffers: S+ADDR(W)+DATA0+DATA1+P.

#define I2C_FLAG_10BIT_ADDR 0x0010
Use 10 bit address.

#define I2C_INIT_DEFAULT undefined
Suggested default configuration for I2C initialization structure.

Enumeration Documentation

I2C_ClockHLR_TypeDef

I2C_ClockHLR_TypeDef

Clock low to high ratio settings.

Enumerator	
i2cClockHLRStandard	Ratio is 4:4.
i2cClockHLRAsymetric	Ratio is 6:3.
i2cClockHLRFast	Ratio is 11:3.

I2C_TransferReturn_TypeDef

I2C_TransferReturn_TypeDef

Return codes for single Controller mode transfer function.

Enumerator	
i2cTransferInProgress	Transfer in progress.
i2cTransferDone	Transfer completed successfully.
i2cTransferNack	NACK received during transfer.
i2cTransferBusErr	Bus error during transfer (misplaced START/STOP).
i2cTransferArbLost	Arbitration lost during transfer.
i2cTransferUsageFault	Usage fault.
i2cTransferSwFault	SW fault.

Function Documentation

I2C_BusFreqGet

uint32_t I2C_BusFreqGet (I2C_TypeDef * i2c)

Get the current configured I2C bus frequency.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.

This frequency is only relevant when acting as master.

Note

- The actual frequency is a real number, this function returns a rounded down (truncated) integer value.

Returns

- The current I2C frequency in Hz.

I2C_BusFreqSet

void I2C_BusFreqSet (I2C_TypeDef * i2c, uint32_t freqRef, uint32_t freqScl, I2C_ClockHLR_TypeDef i2cMode)

Set the I2C bus frequency.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.
uint32_t	[in]	freqRef	An I2C reference clock frequency in Hz that will be used. If set to 0, HFPERCLK / HFPERCCLK clock is used. Setting it to a higher than actual configured value has the consequence of reducing the real I2C frequency.
uint32_t	[in]	freqScl	A bus frequency to set (bus speed may be lower due to integer prescaling). Safe (according to the I2C specification) maximum frequencies for standard fast and fast+ modes are available using I2C_FREQ_ defines. (Using I2C_FREQ_ defines requires corresponding setting of `type`.) The slowest slave device on a bus must always be considered.
I2C_ClockHLR_TypeDef	[in]	i2cMode	A clock low-to-high ratio type to use. If not using i2cClockHLRStandard, make sure all devices on the bus support the specified mode. Using a non-standard ratio is useful to achieve a higher bus clock in fast and fast+ modes.

The bus frequency is only relevant when acting as master. The bus frequency should not be set higher than the maximum frequency accepted by the slowest device on the bus.

Notice that, due to asymmetric requirements on low and high I2C clock cycles in the I2C specification, the maximum frequency allowed to comply with the specification may be somewhat lower than expected.

See the reference manual, details on I2C clock generation, for maximum allowed theoretical frequencies for different modes.

I2C_Enable

```
void I2C_Enable (I2C_TypeDef * i2c, bool enable)
```

Enable/disable I2C.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.
bool	[in]	enable	True to enable counting, false to disable.

Note

- After enabling the I2C (from being disabled), the I2C is in BUSY state.

I2C_Init

```
void I2C_Init (I2C_TypeDef * i2c, const I2C_Init_TypeDef * init)
```

Initialize I2C.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.
const I2C_Init_TypeDef *	[in]	init	A pointer to the I2C initialization structure.

I2C_Reset

```
void I2C_Reset (I2C_TypeDef * i2c)
```

Reset I2C to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.

Note

- The ROUTE register is NOT reset by this function to allow for centralized setup of this feature.

I2C_Transfer

```
I2C_TransferReturn_TypeDef I2C_Transfer (I2C_TypeDef * i2c)
```

Continue an initiated I2C transfer (single master mode only).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.

This function is used repeatedly after a I2C_TransferInit() to complete a transfer. It may be used in polled mode as the below example shows:

```
I2C_TransferReturn_TypeDef ret;

// Do a polled transfer
ret = I2C_TransferInit(I2C0, seq);
while (ret == i2cTransferInProgress)
{
  ret = I2C_Transfer(I2C0);
}
```

It may also be used in interrupt driven mode, where this function is invoked from the interrupt handler. Notice that, if used in interrupt mode, NVIC interrupts must be configured and enabled for the I2C bus used. I2C peripheral specific interrupts are managed by this software.

Note

- Only single master mode is supported.

Returns

- Returns status for an ongoing transfer.
  - i2cTransferInProgress - indicates that transfer not finished.
  - i2cTransferDone - transfer completed successfully.
  - otherwise some sort of error has occurred.

I2C_TransferInit

> I2C_TransferReturn_TypeDef I2C_TransferInit (I2C_TypeDef * i2c, I2C_TransferSeq_TypeDef * seq)

Prepare and start an I2C transfer (single master mode only).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	A pointer to the I2C peripheral register block.
I2C_TransferSeq_TypeDef *	[in]	seq	A pointer to the sequence structure defining the I2C transfer to take place. The referenced structure must exist until the transfer has fully completed.

This function must be invoked to start an I2C transfer sequence. To complete the transfer, I2C_Transfer() must be used either in polled mode or by adding a small driver wrapper using interrupts.

Note

- Only single master mode is supported.

Returns

- Returns the status for an ongoing transfer:
  - i2cTransferInProgress - indicates that the transfer is not finished.
  - Otherwise, an error has occurred.

I2C_IntClear

> void I2C_IntClear (I2C_TypeDef * i2c, uint32_t flags)

Clear one or more pending I2C interrupts.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint32_t	[in]	flags	Pending I2C interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the I2C module (I2C_IF_nnn).

I2C_IntDisable

> void I2C_IntDisable (I2C_TypeDef * i2c, uint32_t flags)

Disable one or more I2C interrupts.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint32_t	[in]	flags	I2C interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the I2C module (I2C_IF_nnn).

I2C_IntEnable

```
void I2C_IntEnable (I2C_TypeDef * i2c, uint32_t flags)
```

Enable one or more I2C interrupts.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint32_t	[in]	flags	I2C interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the I2C module (I2C_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using I2C_IntClear() prior to enabling the interrupt.

I2C_IntGet

```
uint32_t I2C_IntGet (I2C_TypeDef * i2c)
```

Get pending I2C interrupt flags.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.

Note

- Event bits are not cleared by the use of this function.

Returns

- I2C interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the I2C module (I2C_IF_nnn).

I2C_IntGetEnabled

```
uint32_t I2C_IntGetEnabled (I2C_TypeDef * i2c)
```

Get enabled and pending I2C interrupt flags.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

Pending and enabled I2C interrupt sources Return value is the bitwise AND of
- the enabled interrupt sources in I2Cn_IEN and
- the pending interrupt flags I2Cn_IF

I2C_IntSet

```
void I2C_IntSet (I2C_TypeDef * i2c, uint32_t flags)
```

Set one or more pending I2C interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint32_t	[in]	flags	I2C interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for the I2C module (I2C_IF_nnn).

I2C_SlaveAddressGet

```
uint8_t I2C_SlaveAddressGet (I2C_TypeDef * i2c)
```

Get Target address used for I2C peripheral (when operating in Target mode).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.

For 10-bit addressing mode, the address is split in two bytes, and only the first byte setting is fetched, effectively only controlling the 2 most significant bits of the 10-bit address. Full handling of 10-bit addressing in Target mode requires additional SW handling.

Returns

- I2C Target address in use. The 7 most significant bits define the actual address, the least significant bit is reserved and always returned as 0.

I2C_SlaveAddressSet

```
void I2C_SlaveAddressSet (I2C_TypeDef * i2c, uint8_t addr)
```

Set Target address to use for I2C peripheral (when operating in Target mode).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint8_t	[in]	addr	I2C Target address to use. The 7 most significant bits define the actual address, the least significant bit is reserved and always set to 0.

For 10- bit addressing mode, the address is split in two bytes, and only the first byte is set, effectively only controlling the 2 most significant bits of the 10-bit address. Full handling of 10-bit addressing in Target mode requires additional SW handling.

I2C_SlaveAddressMaskGet

```
uint8_t I2C_SlaveAddressMaskGet (I2C_TypeDef * i2c)
```

Get Target address mask used for I2C peripheral (when operating in Target mode).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.

The address mask defines how the comparator works. A bit position with value 0 means that the corresponding Target address bit is ignored during comparison (don't care). A bit position with value 1 means that the corresponding Target address bit must match.

For 10-bit addressing mode, the address is split in two bytes, and only the mask for the first address byte is fetched, effectively only controlling the 2 most significant bits of the 10-bit address.

Returns

- I2C Target address mask in use. The 7 most significant bits define the actual address mask, the least significant bit is reserved and always returned as 0.

I2C_SlaveAddressMaskSet

```
void I2C_SlaveAddressMaskSet (I2C_TypeDef * i2c, uint8_t mask)
```

Set Target address mask used for I2C peripheral (when operating in Target mode).

Parameters

Type	Direction	Argument Name	Description
I2C_TypeDef *	[in]	i2c	Pointer to I2C peripheral register block.
uint8_t	[in]	mask	I2C Target address mask to use. The 7 most significant bits define the actual address mask, the least significant bit is reserved and should be 0.

The address mask defines how the comparator works. A bit position with value 0 means that the corresponding Target address bit is ignored during comparison (don't care). A bit position with value 1 means that the corresponding Target address bit must match.

For 10-bit addressing mode, the address is split in two bytes, and only the mask for the first address byte is set, effectively only controlling the 2 most significant bits of the 10-bit address.

Macro Definition Documentation

I2C_FREQ_STANDARD_MAX

```
#define I2C_FREQ_STANDARD_MAX
```

Value:

```
92000
```

Standard mode max frequency assuming using 4:4 ratio for Nlow:Nhigh.

From I2C specification: Min Tlow = 4.7us, min Thigh = 4.0us, max Trise=1.0us, max Tfall=0.3us. Since ratio is 4:4, have to use worst case value of Tlow or Thigh as base.

1/(Tlow + Thigh + 1us + 0.3us) = 1/(4.7 + 4.7 + 1.3)us = 93458Hz Note

- Due to chip characteristics, max value is somewhat reduced.

I2C_FREQ_FAST_MAX

```
#define I2C_FREQ_FAST_MAX
```

Value:

```
392157
```

Fast mode max frequency assuming using 6:3 ratio for Nlow:Nhigh.

From I2C specification: Min Tlow = 1.3us, min Thigh = 0.6us, max Trise=0.3us, max Tfall=0.3us. Since ratio is 6:3, have to use worst case value of Tlow or 2xThigh as base.

1/(Tlow + Thigh + 0.3us + 0.3us) = 1/(1.3 + 0.65 + 0.6)us = 392157Hz

I2C_FREQ_FASTPLUS_MAX

```
#define I2C_FREQ_FASTPLUS_MAX
```

Value:

```
987167
```

Fast mode+ max frequency assuming using 11:6 ratio for Nlow:Nhigh.

From I2C specification: Min Tlow = 0.5us, min Thigh = 0.26us, max Trise=0.12us, max Tfall=0.12us. Since ratio is 11:6, have to use worst case value of Tlow or (11/6)xThigh as base.

1/(Tlow + Thigh + 0.12us + 0.12us) = 1/(0.5 + 0.273 + 0.24)us = 987167Hz

I2C_FLAG_WRITE

```
#define I2C_FLAG_WRITE
```

Value:

```
0x0001
```

Indicate plain write sequence: S+ADDR(W)+DATA0+P.

- S - Start
- ADDR(W) - address with W/R bit cleared
- DATA0 - Data taken from buffer with index 0
- P - Stop

I2C_FLAG_READ

```
#define I2C_FLAG_READ
```

Value:

```
0x0002
```

Indicate plain read sequence: S+ADDR(R)+DATA0+P.

- S - Start
- ADDR(R) - Address with W/R bit set
- DATA0 - Data read into buffer with index 0
- P - Stop

I2C_FLAG_WRITE_READ

```
#define I2C_FLAG_WRITE_READ
```

Value:

```
0x0004
```

Indicate combined write/read sequence: S+ADDR(W)+DATA0+Sr+ADDR(R)+DATA1+P.

- S - Start
- Sr - Repeated start
- ADDR(W) - Address with W/R bit cleared
- ADDR(R) - Address with W/R bit set
- DATAn - Data written from/read into buffer with index n
- P - Stop

I2C_FLAG_WRITE_WRITE

```
#define I2C_FLAG_WRITE_WRITE
```

Value:

```
0x0008
```

Indicate write sequence using two buffers: S+ADDR(W)+DATA0+DATA1+P.

- S - Start
- ADDR(W) - Address with W/R bit cleared
- DATAn - Data written from buffer with index n
- P - Stop

I2C_FLAG_10BIT_ADDR

```
#define I2C_FLAG_10BIT_ADDR
```

Value:

```
0x0010
```

Use 10 bit address.

I2C_INIT_DEFAULT

#define I2C_INIT_DEFAULT

Value:

```
{                                                                 \
  true,                 /* Enable when initialization done. */    \
  true,                 /* Set to Controller mode. */              \
  0,                    /* Use currently configured reference clock. */ \
  I2C_FREQ_STANDARD_MAX, /* Set to standard rate assuring being */  \
  /*                       within I2C specification. */            \
  i2cClockHLRStandard   /* Set to use 4:4 low/high duty cycle. */  \
}
```

Suggested default configuration for I2C initialization structure.

I2C_Init_TypeDef

I2C initialization structure.

Public Attributes

bool	enable Enable I2C peripheral when initialization completed.
bool	master Set to Controller (true) or Target (false) mode.
uint32_t	refFreq I2C reference clock assumed when configuring bus frequency setup.
uint32_t	freq (Max) I2C bus frequency to use.
I2C_ClockHLR_TypeDef	clhr Clock low/high ratio control.

Public Attribute Documentation

enable

```
bool I2C_Init_TypeDef::enable
```

Enable I2C peripheral when initialization completed.

master

```
bool I2C_Init_TypeDef::master
```

Set to Controller (true) or Target (false) mode.

refFreq

```
uint32_t I2C_Init_TypeDef::refFreq
```

I2C reference clock assumed when configuring bus frequency setup.

Set it to 0 if currently configured reference clock will be used This parameter is only applicable if operating in Controller mode.

freq

```
uint32_t I2C_Init_TypeDef::freq
```

(Max) I2C bus frequency to use.

This parameter is only applicable if operating in Controller mode.

clhr

```
I2C_ClockHLR_TypeDef I2C_Init_TypeDef::clhr
```

Clock low/high ratio control.

I2C_TransferSeq_TypeDef

Master mode transfer message structure used to define a complete I2C transfer sequence (from start to stop).

The structure allows for defining the following types of sequences (refer to defines for sequence details):

- I2C_FLAG_READ - Data read into buf[0].data
- I2C_FLAG_WRITE - Data written from buf[0].data
- I2C_FLAG_WRITE_READ - Data written from buf[0].data and read into buf[1].data
- I2C_FLAG_WRITE_WRITE - Data written from buf[0].data and buf[1].data

Public Attributes

uint16_t	addr Address to use after (repeated) start.
uint16_t	flags Flags defining sequence type and details, see I2C_FLAG_ defines.
uint8_t *	data Buffer used for data to transmit/receive, must be `len` long.
uint16_t	len Number of bytes in `data` to send or receive.
struct I2C_TransferSeq_TypeDef::@0	buf Buffers used to hold data to send from or receive into, depending on sequence type.

Public Attribute Documentation

addr

```
uint16_t I2C_TransferSeq_TypeDef::addr
```

Address to use after (repeated) start.

Layout details, A = Address bit, X = don't care bit (set to 0):

- 7 bit address - Use format AAAA AAAX
- 10 bit address - Use format XXXX XAAX AAAA AAAA

flags

```
uint16_t I2C_TransferSeq_TypeDef::flags
```

Flags defining sequence type and details, see I2C_FLAG_ defines.

data

```
uint8_t* I2C_TransferSeq_TypeDef::data
```

Buffer used for data to transmit/receive, must be `len` long.

len

> uint16_t I2C_TransferSeq_TypeDef::len

Number of bytes in `data` to send or receive.

Notice that when receiving data to this buffer, at least 1 byte must be received. Setting `len` to 0 in the receive case is considered a usage fault. Transmitting 0 bytes is legal, in which case only the address is transmitted after the start condition.

buf

> struct I2C_TransferSeq_TypeDef::@0 I2C_TransferSeq_TypeDef::buf[2]

Buffers used to hold data to send from or receive into, depending on sequence type.

LESENSE - Low Energy Sensor

LESENSE - Low Energy Sensor

Low Energy Sensor (LESENSE) Peripheral API.

This module contains functions to control the LESENSE peripheral of Silicon Labs 32-bit MCUs and SoCs. LESENSE is a low-energy sensor interface capable of autonomously collecting and processing data from multiple sensors even when in EM2.

Modules

LESENSE_CoreCtrlDesc_TypeDef

LESENSE_TimeCtrlDesc_TypeDef

LESENSE_PerCtrlDesc_TypeDef

LESENSE_DecCtrlDesc_TypeDef

LESENSE_Init_TypeDef

LESENSE_ChDesc_TypeDef

LESENSE_ChAll_TypeDef

LESENSE_AltExDesc_TypeDef

LESENSE_ConfAltEx_TypeDef

LESENSE_DecStCond_TypeDef

LESENSE_DecStDesc_TypeDef

LESENSE_DecStAll_TypeDef

Enumerations

enum LESENSE_ClkPresc_TypeDef {

lesenseClkDiv_1 = 0
lesenseClkDiv_2 = 1
lesenseClkDiv_4 = 2
lesenseClkDiv_8 = 3
lesenseClkDiv_16 = 4
lesenseClkDiv_32 = 5
lesenseClkDiv_64 = 6
lesenseClkDiv_128 = 7

}

Clock divisors for controlling the prescaling factor of the period counter.

enum LESENSE_ScanMode_TypeDef {

lesenseScanStartPeriodic = LESENSE_CTRL_SCANMODE_PERIODIC
lesenseScanStartOneShot = LESENSE_CTRL_SCANMODE_ONESHOT
lesenseScanStartPRS = LESENSE_CTRL_SCANMODE_PRS

}

Scan modes.

enum LESENSE_PRSSel_TypeDef {

lesensePRSCh0 = 0
lesensePRSCh1 = 1
lesensePRSCh2 = 2
lesensePRSCh3 = 3
lesensePRSCh4 = 4
lesensePRSCh5 = 5

lesensePRSCh6
= 6
lesensePRSCh7
= 7
lesensePRSCh8
= 8
lesensePRSCh9
= 9
lesensePRSCh10
= 10
lesensePRSCh11
= 11

}
PRS sources.

enum LESENSE_AltExMap_TypeDef {

lesenseAltExMapALTEX = _LESENSE_CTRL_ALTEXMAP_ALTEX
lesenseAltExMapACMP = _LESENSE_CTRL_ALTEXMAP_ACMP

}
Locations of the alternate excitation function.

enum LESENSE_BufTrigLevel_TypeDef {

lesenseBufTrigHalf = LESENSE_CTRL_BUFIDL_HALFFULL
lesenseBufTrigFull = LESENSE_CTRL_BUFIDL_FULL

}
Result buffer interrupt and DMA trigger levels.

enum LESENSE_DMAWakeUp_TypeDef {

lesenseDMAWakeUpDisable = LESENSE_CTRL_DMAWU_DISABLE
lesenseDMAWakeUpBufValid = LESENSE_CTRL_DMAWU_BUFDATAV
lesenseDMAWakeUpBufLevel = LESENSE_CTRL_DMAWU_BUFLEVEL

}
Modes of operation for DMA wakeup from EM2.

enum LESENSE_BiasMode_TypeDef {

lesenseBiasModeDutyCycle = LESENSE_BIASCTRL_BIASMODE_DUTYCYCLE
lesenseBiasModeHighAcc = LESENSE_BIASCTRL_BIASMODE_HIGHACC
lesenseBiasModeDontTouch = LESENSE_BIASCTRL_BIASMODE_DONTTOUCH

}
Bias modes.

enum LESENSE_ScanConfSel_TypeDef {

lesenseScanConfDirMap = LESENSE_CTRL_SCANCONF_DIRMAP
lesenseScanConfInvMap = LESENSE_CTRL_SCANCONF_INVMAP
lesenseScanConfToggle = LESENSE_CTRL_SCANCONF_TOGGLE
lesenseScanConfDecDef = LESENSE_CTRL_SCANCONF_DECDEF

}
Scan configuration.

enum LESENSE_ControlDACData_TypeDef {

lesenseDACIfData = _LESENSE_PERCTRL_DACCH0DATA_DACDATA
lesenseACMPThres = _LESENSE_PERCTRL_DACCH0DATA_ACMPTHRES

}
DAC CHx data control configuration.

enum LESENSE_ControlDACConv_TypeDef {

lesenseDACConvModeDisable = _LESENSE_PERCTRL_DACCH0CONV_DISABLE
lesenseDACConvModeContinuous = _LESENSE_PERCTRL_DACCH0CONV_CONTINUOUS
lesenseDACConvModeSampleHold = _LESENSE_PERCTRL_DACCH0CONV_SAMPLEHOLD
lesenseDACConvModeSampleOff = _LESENSE_PERCTRL_DACCH0CONV_SAMPLEOFF

}
DAC channel x conversion mode configuration.

enum LESENSE_ControlDACOut_TypeDef {

lesenseDACOutModeDisable = _LESENSE_PERCTRL_DACCH0OUT_DISABLE
lesenseDACOutModePin = _LESENSE_PERCTRL_DACCH0OUT_PIN
lesenseDACOutModeADCACMP = _LESENSE_PERCTRL_DACCH0OUT_ADCACMP
lesenseDACOutModePinADCACMP = _LESENSE_PERCTRL_DACCH0OUT_PINADCACMP

}
DAC channel x output mode configuration.

enum LESENSE_DACRef_TypeDef {

lesenseDACRefVdd = LESENSE_PERCTRL_DACREF_VDD
lesenseDACRefBandGap = LESENSE_PERCTRL_DACREF_BANDGAP

}
DAC reference configuration.

enum LESENSE_ControlACMP_TypeDef {

lesenseACMPModeDisable = _LESENSE_PERCTRL_ACMP0MODE_DISABLE
lesenseACMPModeMux = _LESENSE_PERCTRL_ACMP0MODE_MUX
lesenseACMPModeMuxThres = _LESENSE_PERCTRL_ACMP0MODE_MUXTHRES

}
ACMPx control configuration.

enum LESENSE_WarmupMode_TypeDef {

lesenseWarmupModeNormal = LESENSE_PERCTRL_WARMUPMODE_NORMAL
lesenseWarmupModeACMP = LESENSE_PERCTRL_WARMUPMODE_KEEPACMPWARM
lesenseWarmupModeDAC = LESENSE_PERCTRL_WARMUPMODE_KEEPDACWARM
lesenseWarmupModeKeepWarm = LESENSE_PERCTRL_WARMUPMODE_KEEPACMPDACWARM

}
Warm up modes.

enum LESENSE_DecInput_TypeDef {

lesenseDecInputSensorSt = LESENSE_DECCTRL_INPUT_SENSORSTATE
lesenseDecInputPRS = LESENSE_DECCTRL_INPUT_PRS

}
Decoder input source configuration.

enum LESENSE_ChSampleMode_TypeDef {

lesenseSampleModeCounter = 0x0 << _LESENSE_CH_INTERACT_SAMPLE_SHIFT
lesenseSampleModeACMP = LESENSE_CH_INTERACT_SAMPLE_ACMP

}
Compare source selection for sensor sampling.

enum LESENSE_ChIntMode_TypeDef {

lesenseSetIntNone = LESENSE_CH_INTERACT_SETIF_NONE
lesenseSetIntLevel = LESENSE_CH_INTERACT_SETIF_LEVEL
lesenseSetIntPosEdge = LESENSE_CH_INTERACT_SETIF_POSEDGE
lesenseSetIntNegEdge = LESENSE_CH_INTERACT_SETIF_NEGEDGE

}
Interrupt generation setup for CHx interrupt flag.

enum LESENSE_ChPinExMode_TypeDef {

lesenseChPinExDis = LESENSE_CH_INTERACT_EXMODE_DISABLE
lesenseChPinExHigh = LESENSE_CH_INTERACT_EXMODE_HIGH
lesenseChPinExLow = LESENSE_CH_INTERACT_EXMODE_LOW
lesenseChPinExDACOut = LESENSE_CH_INTERACT_EXMODE_DACOUT

}
Channel pin mode for the excitation phase of the scan sequence.

enum LESENSE_ChPinIdleMode_TypeDef {

lesenseChPinIdleDis = _LESENSE_IDLECONF_CH0_DISABLE
lesenseChPinIdleHigh = _LESENSE_IDLECONF_CH0_HIGH
lesenseChPinIdleLow = _LESENSE_IDLECONF_CH0_LOW
lesenseChPinIdleDACCh0 = _LESENSE_IDLECONF_CH0_DACCH0
lesenseChPinIdleDACCh1 = _LESENSE_IDLECONF_CH12_DACCH1

}
Channel pin mode for the idle phase of scan sequence.

enum LESENSE_ChClk_TypeDef {

lesenseClkLF = _LESENSE_CH_INTERACT_EXCLK_LFACLK
lesenseClkHF = _LESENSE_CH_INTERACT_EXCLK_AUXHFRCO

}
Clock used for excitation and sample delay timing.

enum LESENSE_ChCompMode_TypeDef {

lesenseCompModeLess = LESENSE_CH_EVAL_COMP_LESS
lesenseCompModeGreaterOrEq = LESENSE_CH_EVAL_COMP_GE

}
Compare modes for counter comparison.

enum LESENSE_StoreSample_TypeDef {

lesenseStoreSampleDisable = _LESENSE_CH_EVAL_STRSAMPLE_DEFAULT
lesenseStoreSampleData = _LESENSE_CH_EVAL_STRSAMPLE_MASK >>
_LESENSE_CH_EVAL_STRSAMPLE_SHIFT

}
Mode of Storing of Sensor Sample in Result Buffer.

enum LESENSE_AltExPinIdle_TypeDef {

lesenseAltExPinIdleDis = _LESENSE_ALTEXCONF_IDLECONF0_DISABLE
lesenseAltExPinIdleHigh = _LESENSE_ALTEXCONF_IDLECONF0_HIGH
lesenseAltExPinIdleLow = _LESENSE_ALTEXCONF_IDLECONF0_LOW

}
Sensor evaluation modes.

enum LESENSE_StTransAct_TypeDef {

lesenseTransActNone = LESENSE_ST_TCONFA_PRSACT_NONE
lesenseTransActPRS0 = LESENSE_ST_TCONFA_PRSACT_PRS0
lesenseTransActPRS1 = LESENSE_ST_TCONFA_PRSACT_PRS1
lesenseTransActPRS01 = LESENSE_ST_TCONFA_PRSACT_PRS01
lesenseTransActPRS2 = LESENSE_ST_TCONFA_PRSACT_PRS2
lesenseTransActPRS02 = LESENSE_ST_TCONFA_PRSACT_PRS02
lesenseTransActPRS12 = LESENSE_ST_TCONFA_PRSACT_PRS12
lesenseTransActPRS012 = LESENSE_ST_TCONFA_PRSACT_PRS012
lesenseTransActUp = LESENSE_ST_TCONFA_PRSACT_UP
lesenseTransActDown = LESENSE_ST_TCONFA_PRSACT_DOWN
lesenseTransActUpAndPRS2 = LESENSE_ST_TCONFA_PRSACT_UPANDPRS2
lesenseTransActDownAndPRS2 = LESENSE_ST_TCONFA_PRSACT_DOWNANDPRS2

}
Transition action modes.

Functions

void LESENSE_Init(const LESENSE_Init_TypeDef *init, bool reqReset)
Initialize the LESENSE module.

uint32_t LESENSE_ScanFreqSet(uint32_t refFreq, uint32_t scanFreq)
Set the scan frequency for periodic scanning.

void LESENSE_ScanModeSet(LESENSE_ScanMode_TypeDef scanMode, bool start)
Set scan mode of the LESENSE channels.

void LESENSE_StartDelaySet(uint8_t startDelay)
Set the start delay of the sensor interaction on each channel.

void LESENSE_ClkDivSet(LESENSE_ChClk_TypeDef clk, LESENSE_ClkPresc_TypeDef clkDiv)
Set the clock division for LESENSE timers.

void LESENSE_ChannelAllConfig(const LESENSE_ChAll_TypeDef *confChAll)
Configure all (16) LESENSE sensor channels.

void LESENSE_ChannelConfig(const LESENSE_ChDesc_TypeDef *confCh, uint32_t chIdx)
Configure a single LESENSE sensor channel.

void	LESENSE_AltExConfig(const LESENSE_ConfAltEx_TypeDef *confAltEx) Configure the LESENSE alternate excitation modes.
void	LESENSE_ChannelEnable(uint8_t chIdx, bool enaScanCh, bool enaPin) Enable/disable LESENSE scan channel and the pin assigned to it.
void	LESENSE_ChannelEnableMask(uint16_t chMask, uint16_t pinMask) Enable/disable LESENSE scan channel and the pin assigned to it.
void	LESENSE_ChannelTimingSet(uint8_t chIdx, uint8_t exTime, uint8_t sampleDelay, uint16_t measDelay) Set LESENSE channel timing parameters.
void	LESENSE_ChannelThresSet(uint8_t chIdx, uint16_t acmpThres, uint16_t cntThres) Set LESENSE channel threshold parameters.
void	LESENSE_DecoderStateAllConfig(const LESENSE_DecStAll_TypeDef *confDecStAll) Configure all LESENSE decoder states.
void	LESENSE_DecoderStateConfig(const LESENSE_DecStDesc_TypeDef *confDecSt, uint32_t decSt) Configure a single LESENSE decoder state.
void	LESENSE_DecoderStateSet(uint32_t decSt) Set the LESENSE decoder state.
uint32_t	LESENSE_DecoderStateGet(void) Get the current state of the LESENSE decoder.
void	LESENSE_ScanStart(void) Start scanning sensors.
void	LESENSE_ScanStop(void) Stop scanning sensors.
void	LESENSE_DecoderStart(void) Start the LESENSE decoder.
void	LESENSE_ResultBufferClear(void) Clear the result buffer.
void	LESENSE_Reset(void) Reset the LESENSE module.
void	LESENSE_DecoderStop(void) Stop LESENSE decoder.
uint32_t	LESENSE_StatusGet(void) Get the current status of LESENSE.
void	LESENSE_StatusWait(uint32_t flag) Wait until status of LESENSE is equal to what was requested.
uint32_t	LESENSE_ChannelActiveGet(void) Get the currently active channel index.
uint32_t	LESENSE_ScanResultGet(void) Get the latest scan comparison result (1 bit / channel).
uint32_t	LESENSE_ScanResultDataGet(void) Get the oldest unread data from the result buffer.
uint32_t	LESENSE_ScanResultDataBufferGet(uint32_t idx) Get data from the result data buffer.
uint32_t	LESENSE_SensorStateGet(void) Get the current state of the LESENSE sensor.

void LESENSE_RAMPowerDown(void)
Shut off the power to the LESENSE RAM, which disables LESENSE.

void LESENSE_IntClear(uint32_t flags)
Clear one or more pending LESENSE interrupts.

void LESENSE_IntEnable(uint32_t flags)
Enable one or more LESENSE interrupts.

void LESENSE_IntDisable(uint32_t flags)
Disable one or more LESENSE interrupts.

void LESENSE_IntSet(uint32_t flags)
Set one or more pending LESENSE interrupts from SW.

uint32_t LESENSE_IntGet(void)
Get pending LESENSE interrupt flags.

uint32_t LESENSE_IntGetEnabled(void)
Get enabled and pending LESENSE interrupt flags.

Macros

#define LESENSE_NUM_DECODER_STATES (_LESENSE_DECSTATE_DECSTATE_MASK + 1)
Number of decoder states supported by current device.

#define LESENSE_NUM_CHANNELS 16
Number of LESENSE channels.

#define LESENSE_CORECTRL_DESC_DEFAULT undefined
Default configuration for LESENSE_CtrlDesc_TypeDef structure.

#define LESENSE_TIMECTRL_DESC_DEFAULT undefined
Default configuration for LESENSE_TimeCtrlDesc_TypeDef structure.

#define LESENSE_PERCTRL_DESC_DEFAULT undefined
Default configuration for LESENSE_PerCtrl_TypeDef structure.

#define LESENSE_DECCTRL_DESC_DEFAULT undefined
Default configuration for LESENSE_PerCtrl_TypeDef structure.

#define LESENSE_INIT_DEFAULT undefined
Default configuration for LESENSE_Init_TypeDef structure.

#define LESENSE_CH_CONF_DEFAULT undefined
Default configuration for the scan channel.

#define LESENSE_SCAN_CONF_DEFAULT undefined
Default configuration for all the sensor channels.

#define LESENSE_ALTEX_CH_CONF_DEFAULT undefined
Default configuration for the alternate excitation channel.

#define LESENSE_ALTEX_CONF_DEFAULT undefined
Default configuration for all the alternate excitation channels.

#define LESENSE_ST_CONF_DEFAULT undefined
Default configuration for the decoder state condition.

#define LESENSE_DECODER_CONF_DEFAULT undefined
Default configuration for all decoder states.

Enumeration Documentation

LESENSE_ClkPresc_TypeDef

> LESENSE_ClkPresc_TypeDef

Clock divisors for controlling the prescaling factor of the period counter.

Note: These enumeration values are used for different clock division related configuration parameters (hfPresc, lfPresc, pcPresc).

Enumerator	
lesenseClkDiv_1	Divide clock by 1.
lesenseClkDiv_2	Divide clock by 2.
lesenseClkDiv_4	Divide clock by 4.
lesenseClkDiv_8	Divide clock by 8.
lesenseClkDiv_16	Divide clock by 16.
lesenseClkDiv_32	Divide clock by 32.
lesenseClkDiv_64	Divide clock by 64.
lesenseClkDiv_128	Divide clock by 128.

LESENSE_ScanMode_TypeDef

> LESENSE_ScanMode_TypeDef

Scan modes.

Enumerator	
lesenseScanStartPeriodic	New scan is started each time the period counter overflows.
lesenseScanStartOneShot	Single scan is performed when LESENSE_ScanStart() is called.
lesenseScanStartPRS	New scan is triggered by pulse on PRS channel.

LESENSE_PRSSel_TypeDef

> LESENSE_PRSSel_TypeDef

PRS sources.

Note: These enumeration values are being used for different PRS related configuration parameters.

Enumerator	
lesensePRSCh0	PRS channel 0.
lesensePRSCh1	PRS channel 1.
lesensePRSCh2	PRS channel 2.
lesensePRSCh3	PRS channel 3.
lesensePRSCh4	PRS channel 4.
lesensePRSCh5	PRS channel 5.
lesensePRSCh6	PRS channel 6.
lesensePRSCh7	PRS channel 7.
lesensePRSCh8	PRS channel 8.

lesensePRSCh9	PRS channel 9.
lesensePRSCh10	PRS channel 10.
lesensePRSCh11	PRS channel 11.

LESENSE_AltExMap_TypeDef

LESENSE_AltExMap_TypeDef

Locations of the alternate excitation function.

Enumerator	
lesenseAltExMapALTEX	Alternate excitation is mapped to the LES_ALTEX pins.
lesenseAltExMapACMP	Alternate excitation is mapped to the pins of the other ACMP.

LESENSE_BufTrigLevel_TypeDef

LESENSE_BufTrigLevel_TypeDef

Result buffer interrupt and DMA trigger levels.

Enumerator	
lesenseBufTrigHalf	DMA and interrupt flags are set when the result buffer is half-full.
lesenseBufTrigFull	DMA and interrupt flags set when the result buffer is full.

LESENSE_DMAWakeUp_TypeDef

LESENSE_DMAWakeUp_TypeDef

Modes of operation for DMA wakeup from EM2.

Enumerator	
lesenseDMAWakeUpDisable	No DMA wakeup from EM2.
lesenseDMAWakeUpBufValid	DMA wakeup from EM2 when data is valid in the result buffer.
lesenseDMAWakeUpBufLevel	DMA wakeup from EM2 when the result buffer is full/half-full, depending on RESBIDL configuration in the LESENSE_CTRL register (selected by the resBufTrigLevel in LESENSE_ResBufTrigLevel_TypeDef descriptor structure).

LESENSE_BiasMode_TypeDef

LESENSE_BiasMode_TypeDef

Bias modes.

Enumerator	
lesenseBiasModeDutyCycle	Duty cycle bias module between low power and high accuracy mode.
lesenseBiasModeHighAcc	Bias module is always in high accuracy mode.
lesenseBiasModeDontTouch	Bias module is controlled by EMU and not affected by the LESENSE.

LESENSE_ScanConfSel_TypeDef

> LESENSE_ScanConfSel_TypeDef

Scan configuration.

Enumerator	
lesenseScanConfDirMap	Channel configuration registers (CHx_CONF) used are directly mapped to the channel number.
lesenseScanConfInvMap	Channel configuration registers used are CHx+8_CONF for channels 0-7 and CHx-8_CONF for channels 8-15.
lesenseScanConfToggle	Channel configuration registers used toggles between CHX_SCANCONF and CHX+8_SCANCONF when channel x triggers.
lesenseScanConfDecDef	Decoder state defines the channel configuration register (CHx_CONF) to be used.

LESENSE_ControlDACData_TypeDef

> LESENSE_ControlDACData_TypeDef

DAC CHx data control configuration.

Enumerator	
lesenseDACIfData	DAC channel x data is defined by the DAC_CHxDATA register.
lesenseACMPThres	DAC channel x data is defined by the ACMPTHRES in LESENSE_CHx_INTERACT.

LESENSE_ControlDACConv_TypeDef

> LESENSE_ControlDACConv_TypeDef

DAC channel x conversion mode configuration.

Enumerator	
lesenseDACConvModeDisable	LESENSE does not control the DAC channel x.
lesenseDACConvModeContinuous	DAC channel x is driven in continuous mode.
lesenseDACConvModeSampleHold	DAC channel x is driven in sample hold mode.
lesenseDACConvModeSampleOff	DAC channel x is driven in sample off mode.

LESENSE_ControlDACOut_TypeDef

> LESENSE_ControlDACOut_TypeDef

DAC channel x output mode configuration.

Enumerator	
lesenseDACOutModeDisable	DAC CHx output to pin and ACMP/ADC disabled.
lesenseDACOutModePin	DAC CHx output to pin enabled, output to ADC and ACMP disabled.
lesenseDACOutModeADCACMP	DAC CHx output to pin disabled, output to ADC and ACMP enabled.
lesenseDACOutModePinADCACMP	DAC CHx output to pin, ADC, and ACMP enabled.

LESENSE_DACRef_TypeDef

> LESENSE_DACRef_TypeDef

DAC reference configuration.

Enumerator	
lesenseDACRefVdd	DAC uses VDD reference.
lesenseDACRefBandGap	DAC uses band gap reference.

LESENSE_ControlACMP_TypeDef

> LESENSE_ControlACMP_TypeDef

ACMPx control configuration.

Enumerator	
lesenseACMPModeDisable	LESENSE does not control ACMPx.
lesenseACMPModeMux	LESENSE controls input mux of ACMPx.
lesenseACMPModeMuxThres	LESENSE controls input mux of and threshold value of ACMPx.

LESENSE_WarmupMode_TypeDef

> LESENSE_WarmupMode_TypeDef

Warm up modes.

ACMP and DAC duty cycle mode configuration.

Enumerator	
lesenseWarmupModeNormal	ACMPs and DACs are shut down when LESENSE is idle.
lesenseWarmupModeACMP	ACMPs are kept powered up when LESENSE is idle.
lesenseWarmupModeDAC	DAC is kept powered up when LESENSE is idle.
lesenseWarmupModeKeepWarm	ACMPs and DAC are kept powered up when LESENSE is idle.

LESENSE_DecInput_TypeDef

> LESENSE_DecInput_TypeDef

Decoder input source configuration.

Enumerator	
lesenseDecInputSensorSt	SENSORSTATE register is used as input to the decoder.
lesenseDecInputPRS	PRS channels are used as input to the decoder.

LESENSE_ChSampleMode_TypeDef

> LESENSE_ChSampleMode_TypeDef

Compare source selection for sensor sampling.

Enumerator	
lesenseSampleModeCounter	Counter output will be used in comparison.
lesenseSampleModeACMP	ACMP output will be used in comparison.

LESENSE_ChIntMode_TypeDef

```
LESENSE_ChIntMode_TypeDef
```

Interrupt generation setup for CHx interrupt flag.

Enumerator	
lesenseSetIntNone	No interrupt is generated.
lesenseSetIntLevel	Set interrupt flag if the sensor triggers.
lesenseSetIntPosEdge	Set interrupt flag on positive edge of the sensor state.
lesenseSetIntNegEdge	Set interrupt flag on negative edge of the sensor state.

LESENSE_ChPinExMode_TypeDef

```
LESENSE_ChPinExMode_TypeDef
```

Channel pin mode for the excitation phase of the scan sequence.

Enumerator	
lesenseChPinExDis	Channel pin is disabled.
lesenseChPinExHigh	Channel pin is configured as push-pull, driven HIGH.
lesenseChPinExLow	Channel pin is configured as push-pull, driven LOW.
lesenseChPinExDACOut	DAC output (only available on channel 0, 1, 2, 3, 12, 13, 14 and 15)

LESENSE_ChPinIdleMode_TypeDef

```
LESENSE_ChPinIdleMode_TypeDef
```

Channel pin mode for the idle phase of scan sequence.

Enumerator	
lesenseChPinIdleDis	Channel pin is disabled in idle phase.
lesenseChPinIdleHigh	Channel pin is configured as push-pull, driven HIGH in idle phase.
lesenseChPinIdleLow	Channel pin is configured as push-pull, driven LOW in idle phase.
lesenseChPinIdleDACCh0	Channel pin is connected to DAC CH0 output in idle phase.
lesenseChPinIdleDACCh1	Channel pin is connected to DAC CH1 output in idle phase.

LESENSE_ChClk_TypeDef

```
LESENSE_ChClk_TypeDef
```

Clock used for excitation and sample delay timing.

Enumerator	
lesenseClkLF	LFACLK (LF clock) is used.
lesenseClkHF	AUXHFRCO (HF clock) is used.

LESENSE_ChCompMode_TypeDef

```
LESENSE_ChCompMode_TypeDef
```

Compare modes for counter comparison.

Enumerator	
lesenseCompModeLess	Comparison evaluates to 1 if sensor data is less than the counter threshold, or if ACMP output is 0.
lesenseCompModeGreaterOrEq	Comparison evaluates to 1 if sensor data is greater than, or equal to the counter threshold, or if the ACMP output is 1.

LESENSE_StoreSample_TypeDef

```
LESENSE_StoreSample_TypeDef
```

Mode of Storing of Sensor Sample in Result Buffer.

Enumerator	
lesenseStoreSampleDisable	Nothing will be stored in the result buffer.
lesenseStoreSampleData	The sensor sample data will be stored in the result buffer.

LESENSE_AltExPinIdle_TypeDef

```
LESENSE_AltExPinIdle_TypeDef
```

Sensor evaluation modes.

Idle phase configuration of the alternate excitation channels.

Enumerator	
lesenseAltExPinIdleDis	ALTEX output is disabled in idle phase.
lesenseAltExPinIdleHigh	ALTEX output is high in idle phase.
lesenseAltExPinIdleLow	ALTEX output is low in idle phase.

LESENSE_StTransAct_TypeDef

```
LESENSE_StTransAct_TypeDef
```

Transition action modes.

Enumerator	
lesenseTransActNone	No PRS pulses generated (if PRSCOUNT == 0).
lesenseTransActPRS0	Generate pulse on LESPRS0 (if PRSCOUNT == 0).
lesenseTransActPRS1	Generate pulse on LESPRS1 (if PRSCOUNT == 0).

lesenseTransActPRS01	Generate pulse on LESPRS0 and LESPRS1 (if PRSCOUNT == 0).
lesenseTransActPRS2	Generate pulse on LESPRS2 (for both PRSCOUNT == 0 and PRSCOUNT == 1).
lesenseTransActPRS02	Generate pulse on LESPRS0 and LESPRS2 (if PRSCOUNT == 0).
lesenseTransActPRS12	Generate pulse on LESPRS1 and LESPRS2 (if PRSCOUNT == 0).
lesenseTransActPRS012	Generate pulse on LESPRS0, LESPRS1 and LESPRS2 (if PRSCOUNT == 0).
lesenseTransActUp	Count up (if PRSCOUNT == 1).
lesenseTransActDown	Count down (if PRSCOUNT == 1).
lesenseTransActUpAndPRS2	Count up and generate pulse on LESPRS2 (if PRSCOUNT == 1).
lesenseTransActDownAndPRS2	Count down and generate pulse on LESPRS2 (if PRSCOUNT == 1).

Function Documentation

LESENSE_Init

> void LESENSE_Init (const LESENSE_Init_TypeDef * init, bool reqReset)

Initialize the LESENSE module.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_Init_TypeDef *	[in]	init	The LESENSE initialization structure.
bool	[in]	reqReset	Request to call LESENSE_Reset() first to initialize all LESENSE registers with default values.

This function configures the main parameters of the LESENSE interface. See the initialization parameter type definition (LESENSE_Init_TypeDef) for more details.

Note

- LESENSE_Init() is designed to initialize LESENSE once in an operation cycle. Be aware of the effects of reconfiguration if using this function from multiple sources in your code. This function has not been designed to be re-entrant. Requesting reset by setting `reqReset` to true is required in each reset or power-on cycle to configure the default values of the RAM mapped LESENSE registers. Notice that GPIO pins used by the LESENSE module must be properly configured by the user explicitly for the LESENSE to work as intended. (When configuring pins, one should remember to consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

LESENSE_ScanFreqSet

> uint32_t LESENSE_ScanFreqSet (uint32_t refFreq, uint32_t scanFreq)

Set the scan frequency for periodic scanning.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	refFreq	Select reference LFACLK clock frequency in Hz. If set to 0, the current clock frequency is being used as a reference.
uint32_t	[in]	scanFreq	Set the desired scan frequency in Hz.

This function only applies to LESENSE if a period counter is used as a trigger for scan start. The calculation is based on the following formula: Fscan = LFACLKles / ((1+PCTOP)*2^PCPRESC)

Note

- Note that the calculation does not necessarily result in the requested scan frequency due to integer division. Check the return value for the resulted scan frequency.

Returns

- Frequency in Hz calculated and set by this function. Users can use this to compare the requested and set values.

LESENSE_ScanModeSet

```
void LESENSE_ScanModeSet (LESENSE_ScanMode_TypeDef scanMode, bool start)
```

Set scan mode of the LESENSE channels.

Parameters

Type	Direction	Argument Name	Description
LESENSE_ScanMode_TypeDef	[in]	scanMode	Select the location to map LESENSE alternate excitation channels. • lesenseScanStartPeriodic - A new scan is started each time the period counter overflows. • lesenseScanStartOneShot - A single scan is performed when LESENSE_ScanStart() is called. • lesenseScanStartPRS - A new scan is triggered by pulse on the PRS channel.
bool	[in]	start	If true, LESENSE_ScanStart() is immediately issued after configuration.

This function configures how the scan start is triggered. It can be used for re-configuring the scan mode while running the application but it is also used by LESENSE_Init() for initialization.

Note

- Users can configure the scan mode by LESENSE_Init() function, but only with a significant overhead. This simple function serves the purpose of controlling this parameter after the channel has been configured. Be aware of the effects of the non-atomic Read-Modify-Write cycle.

LESENSE_StartDelaySet

```
void LESENSE_StartDelaySet (uint8_t startDelay)
```

Set the start delay of the sensor interaction on each channel.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	startDelay	A number of LFACLK cycles to delay. A valid range: 0-3 (2 bit).

This function sets the start delay of the sensor interaction on each channel. It can be used for adjusting the start delay while running the application but it is also used by LESENSE_Init() for initialization.

Note

- Users can configure the start delay by LESENSE_Init() function, but only with a significant overhead. This simple function serves the purpose of controlling this parameter after the channel has been configured. Be aware of the effects of the non-atomic Read-Modify-Write cycle.

LESENSE_ClkDivSet

void LESENSE_ClkDivSet (LESENSE_ChClk_TypeDef clk, LESENSE_ClkPresc_TypeDef clkDiv)

Set the clock division for LESENSE timers.

Parameters

Type	Direction	Argument Name	Description
LESENSE_ChClk_TypeDef	[in]	clk	Select the clock to prescale. • lesenseClkHF - set AUXHFRCO clock divisor for HF timer. • lesenseClkLF - set LFACLKles clock divisor for LF timer.
LESENSE_ClkPresc_TypeDef	[in]	clkDiv	The clock divisor value. A valid range depends on the `clk` value.

Use this function to configure the clock division for the LESENSE timers used for excitation timing. The division setting is global but the clock source can be selected for each channel using LESENSE_ChannelConfig() function. See documentation for more details.

Note

- If AUXHFRCO is used for excitation timing, LFACLK can't exceed 500 kHz. LFACLK can't exceed 50 kHz if the ACMP threshold level (ACMPTHRES) is not equal for all channels.

LESENSE_ChannelAllConfig

void LESENSE_ChannelAllConfig (const LESENSE_ChAll_TypeDef * confChAll)

Configure all (16) LESENSE sensor channels.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_ChAll_TypeDef *	[in]	confChAll	A configuration structure for all (16) LESENSE sensor channels.

This function configures all sensor channels of the LESENSE interface. See the configuration parameter type definition (LESENSE_ChAll_TypeDef) for more details.

Note

- Channels can be configured individually using LESENSE_ChannelConfig() function. Notice that pins used by the LESENSE module must be properly configured by the user explicitly for LESENSE to work as intended. (When configuring pins, consider the sequence of the configuration to avoid unintended pulses/glitches on output pins.)

LESENSE_ChannelConfig

> void LESENSE_ChannelConfig (const LESENSE_ChDesc_TypeDef * confCh, uint32_t chIdx)

Configure a single LESENSE sensor channel.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_ChDesc_TypeDef *	[in]	confCh	A configuration structure for a single LESENSE sensor channel.
uint32_t	[in]	chIdx	A channel index to configure (0-15).

This function configures a single sensor channel of the LESENSE interface. See the configuration parameter type definition (LESENSE_ChDesc_TypeDef) for more details.

Note

- This function has been designed to minimize the effects of sensor channel reconfiguration while LESENSE is in operation. However, be aware of these effects and the right timing to call this function. Parameter `useAltEx` must be true in the channel configuration to use alternate excitation pins.

LESENSE_AltExConfig

> void LESENSE_AltExConfig (const LESENSE_ConfAltEx_TypeDef * confAltEx)

Configure the LESENSE alternate excitation modes.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_ConfAltEx_TypeDef *	[in]	confAltEx	A configuration structure for LESENSE alternate excitation pins.

This function configures the alternate excitation channels of the LESENSE interface. See the configuration parameter type definition (LESENSE_ConfAltEx_TypeDef) for more details.

Note

- The `useAltEx` parameter must be true in the channel configuration structure (LESENSE_ChDesc_TypeDef) to use alternate excitation pins on the channel.

LESENSE_ChannelEnable

> void LESENSE_ChannelEnable (uint8_t chIdx, bool enaScanCh, bool enaPin)

Enable/disable LESENSE scan channel and the pin assigned to it.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	chIdx	An identifier of the scan channel. A valid range: 0-15.
bool	[in]	enaScanCh	Enable/disable the selected scan channel by setting this parameter to true/false respectively.

Type	Direction	Argument Name	Description
bool	[in]	enaPin	Enable/disable the pin assigned to the channel selected by `chIdx`.

Use this function to enable/disable a selected LESENSE scan channel and the pin assigned to it.

Note

- Users can enable/disable scan channels and the channel pin with the LESENSE_ChannelConfig() function, but only with a significant overhead. This simple function controls these parameters after the channel has been configured.

LESENSE_ChannelEnableMask

> void LESENSE_ChannelEnableMask (uint16_t chMask, uint16_t pinMask)

Enable/disable LESENSE scan channel and the pin assigned to it.

Parameters

Type	Direction	Argument Name	Description
uint16_t	[in]	chMask	Set the corresponding bit to 1 to enable, 0 to disable the selected scan channel.
uint16_t	[in]	pinMask	Set the corresponding bit to 1 to enable, 0 to disable the pin on selected channel.

Use this function to enable/disable LESENSE scan channels and the pins assigned to them using a mask.

Note

- Users can enable/disable scan channels and channel pins by using the LESENSE_ChannelAllConfig() function, but only with a significant overhead. This simple function controls these parameters after the channel has been configured.

LESENSE_ChannelTimingSet

> void LESENSE_ChannelTimingSet (uint8_t chIdx, uint8_t exTime, uint8_t sampleDelay, uint16_t measDelay)

Set LESENSE channel timing parameters.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	chIdx	An identifier of the scan channel. A valid range is 0-15.
uint8_t	[in]	exTime	An excitation time on chIdx. The excitation will last exTime+1 excitation clock cycles. A valid range is 0-63 (6 bits).
uint8_t	[in]	sampleDelay	Sample delay on chIdx. Sampling will occur after sampleDelay+1 sample clock cycles. A valid range is 0-127 (7 bits).
uint16_t	[in]	measDelay	A measure delay on chIdx. Sensor measuring is delayed for measDelay+1 excitation clock cycles. A valid range is 0-127 (7 bits).

Use this function to set timing parameters on a selected LESENSE channel.

Note

-

Users can configure the channel timing parameters with the LESENSE_ChannelConfig() function, but only with a significant overhead. This simple function controls these parameters after the channel has been configured.

LESENSE_ChannelThresSet

```
void LESENSE_ChannelThresSet (uint8_t chIdx, uint16_t acmpThres, uint16_t cntThres)
```

Set LESENSE channel threshold parameters.

Parameters

Type	Direction	Argument Name	Description
uint8_t	[in]	chIdx	An identifier of the scan channel. A valid range is 0-15.
uint16_t	[in]	acmpThres	ACMP threshold. • If perCtrl.dacCh0Data or perCtrl.dacCh1Data is set to lesenseDACIfData, acmpThres defines the 12-bit DAC data in the corresponding data register of the DAC interface (DACn_CH0DATA and DACn_CH1DATA). In this case, the valid range is 0-4095 (12 bits). • If perCtrl.dacCh0Data or perCtrl.dacCh1Data is set to lesenseACMPThres, acmpThres defines the 6-bit Vdd scaling factor of ACMP negative input (VDDLEVEL in ACMP_INPUTSEL register). In this case, the valid range is 0-63 (6 bits).
uint16_t	[in]	cntThres	A decision threshold for counter comparison. A valid range is 0-65535 (16 bits).

Use this function to set threshold parameters on a selected LESENSE channel.

Note

- Users can configure the channel threshold parameters with the LESENSE_ChannelConfig() function, but only with a significant overhead. This simple function serves controls these parameters after the channel has been configured.

LESENSE_DecoderStateAllConfig

```
void LESENSE_DecoderStateAllConfig (const LESENSE_DecStAll_TypeDef * confDecStAll)
```

Configure all LESENSE decoder states.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_DecStAll_TypeDef *	[in]	confDecStAll	A configuration structure for all (16 or 32) LESENSE decoder states.

This function configures all the decoder states of the LESENSE interface. See the configuration parameter type definition (LESENSE_DecStAll_TypeDef) for more details.

Note

- Decoder states can be configured individually using LESENSE_DecoderStateConfig() function. Starting from Series 2 Config 3 (xG23 and higher), this function configures a transition ARC instead of a decoder state.

LESENSE_DecoderStateConfig

> void LESENSE_DecoderStateConfig (const LESENSE_DecStDesc_TypeDef * confDecSt, uint32_t decSt)

Configure a single LESENSE decoder state.

Parameters

Type	Direction	Argument Name	Description
const LESENSE_DecStDesc_TypeDef *	[in]	confDecSt	A configuration structure for a single LESENSE decoder state.
uint32_t	[in]	decSt	A decoder state index to configure (0-15) or (0-31) depending on the device.

This function configures a single decoder state of the LESENSE interface. See the configuration parameter type definition (LESENSE_DecStDesc_TypeDef) for more details.

Note

- Starting from Series 2 Config 3 (xG23 and higher), this function configures a transition ARC instead of a decoder state.

LESENSE_DecoderStateSet

> void LESENSE_DecoderStateSet (uint32_t decSt)

Set the LESENSE decoder state.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	decSt	A decoder state to set as the current state. A valid range is 0-15 or 0-31, depending on the device.

This function can be used for setting the initial state of the LESENSE decoder.

Note

- Make sure the LESENSE decoder state is initialized by this function before enabling the decoder!

LESENSE_DecoderStateGet

> uint32_t LESENSE_DecoderStateGet (void)

Get the current state of the LESENSE decoder.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- This function returns the value of the LESENSE_DECSTATE register that represents the current state of the LESENSE decoder.

LESENSE_ScanStart

```
void LESENSE_ScanStart (void )
```

Start scanning sensors.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function will wait for any pending previous write operation to the CMD register to complete before accessing the CMD register. It will also wait for the write operation to the CMD register to complete before returning. Each write operation to the CMD register may take up to 3 LF clock cycles, so expect some delay. The user may implement a separate function to write multiple command bits in the CMD register in one single operation to optimize an application.

LESENSE_ScanStop

```
void LESENSE_ScanStop (void )
```

Stop scanning sensors.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function will wait for any pending previous write operation to the CMD register to complete before accessing the CMD register. It will also wait for the write operation to the CMD register to complete before returning. Each write operation to the CMD register may take up to 3 LF clock cycles, so the user should expect some delay. The user may implement a separate function to write multiple command bits in the CMD register in one single operation in order to optimize an application.
- If issued during a scan, the command takes effect after scan completion.

LESENSE_DecoderStart

```
void LESENSE_DecoderStart (void )
```

Start the LESENSE decoder.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function will wait for any pending previous write operation to the CMD register to complete before accessing the CMD register. It will also wait for the write operation to the CMD register to complete before returning. Each write operation to the CMD register may take up to 3 LF clock cycles, so expect some delay. The user may implement a separate function to write multiple command bits in the CMD register in one single operation to optimize an application.

LESENSE_ResultBufferClear

```
void LESENSE_ResultBufferClear (void )
```

Clear the result buffer.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function will wait for any pending previous write operation to the CMD register to complete before accessing the CMD register. It will also wait for the write operation to the CMD register to complete before returning. Each write operation to the CMD register may take up to 3 LF clock cycles, so expect some delay. The user may implement a separate function to write multiple command bits in the CMD register in one single operation to optimize an application.

LESENSE_Reset

```
void LESENSE_Reset (void )
```

Reset the LESENSE module.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Use this function to reset LESENSE registers.

Note

- Resetting LESENSE registers is required in each reset or power-on cycle to configure the default values of the RAM mapped LESENSE registers. LESENSE_Reset() can be called on initialization by setting the `reqReset` parameter to true in LESENSE_Init(). Starting from Series 2 Config 3 (xG23 and higher), this function leaves LESENSE in the disabled state.

LESENSE_DecoderStop

```
void LESENSE_DecoderStop (void )
```

Stop LESENSE decoder.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Disables LESENSE decoder by setting the command to LESENSE_DECCTRL register.

LESENSE_StatusGet

```
uint32_t LESENSE_StatusGet (void )
```

Get the current status of LESENSE.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Returns the value of the LESENSE_STATUS register that contains the OR combination of the following status bits for EFR series 0/1:
  - LESENSE_STATUS_BUFDATAV - Result data valid. Set when data is available in result buffer. Cleared when buffer is empty.
  - LESENSE_STATUS_BUFFULL - Result buffer full. Set when result buffer is full.
  - LESENSE_STATUS_BUFHALFFULL - Result buffer half full. Set when result buffer is half full.
  - LESENSE_STATUS_RUNNING - LESENSE is active.
  - LESENSE_STATUS_SCANACTIVE - LESENSE is currently interfacing sensors.

  The OR combination of the following status bits for EFR series 2:
  - LESENSE_STATUS_RESFIFOV - Result Fifo valid. Set when data is available in result Fifo. Cleared when Fifo is empty.
  - LESENSE_STATUS_RESFIFOFULL - Result Fifo full. Set when result Fifo is full.
  - LESENSE_STATUS_RUNNING - LESENSE is active.
  - LESENSE_STATUS_SCANACTIVE - LESENSE is currently interfacing sensors.
  - LESENSE_STATUS_FLUSHING - Fifo flushing
  - LESENSE_STATUS_READBUSY - Fifo Read busy

LESENSE_StatusWait

```
void LESENSE_StatusWait (uint32_t flag)
```

Wait until status of LESENSE is equal to what was requested.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flag	The OR combination of the following status bits for EFR series 0/1: • LESENSE_STATUS_BUFDATAV - Result data valid. Set when data is available in result buffer. Cleared when buffer is empty. • LESENSE_STATUS_BUFHALFFULL - Result buffer half full. Set when result buffer is half full. • LESENSE_STATUS_BUFFULL - Result buffer full. Set when result buffer is full. • LESENSE_STATUS_RUNNING - LESENSE is active. • LESENSE_STATUS_SCANACTIVE - LESENSE is currently interfacing sensors. • LESENSE_STATUS_DACACTIVE - The DAC interface is currently active. The OR combination of the following status bits for EFR series 2: • LESENSE_STATUS_RESFIFOV - Result FIFO valid. Set when data is available in result FIFO. Cleared when FIFO is empty. • LESENSE_STATUS_RESFIFOFULL - Result FIFO full. Set when result FIFO is full. • LESENSE_STATUS_RUNNING - LESENSE is active. • LESENSE_STATUS_SCANACTIVE - LESENSE is currently interfacing sensors. • LESENSE_STATUS_FLUSHING - FIFO flushing • LESENSE_STATUS_READBUSY - FIFO Read busy

Polls LESENSE_STATUS register and waits until requested combination of flags are set.

LESENSE_ChannelActiveGet

```
uint32_t LESENSE_ChannelActiveGet (void )
```

Get the currently active channel index.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Returns the value of the LESENSE_CHINDEX register that contains the index of currently active channel (0-15).

LESENSE_ScanResultGet

```
uint32_t LESENSE_ScanResultGet (void )
```

Get the latest scan comparison result (1 bit / channel).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

-

Returns the value of the LESENSE_SCANRES register that contains the comparison result of last scan on all channels. Bit x is set if a comparison triggered on channel x, which means that LESENSE counter met the comparison criteria set in LESENSE_CHx_EVAL by COMPMODE and CNTTHRES.

LESENSE_ScanResultDataGet

```
uint32_t LESENSE_ScanResultDataGet (void )
```

Get the oldest unread data from the result buffer.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Make sure that the STORERES bit is set in LESENSE_CHx_EVAL, or the STRSCANRES bit is set in LESENSE_CTRL; otherwise, returns the undefined value.

Returns

- Returns the value of LESENSE_RESDATA register that contains the oldest unread counter result from result buffer.

LESENSE_ScanResultDataBufferGet

```
uint32_t LESENSE_ScanResultDataBufferGet (uint32_t idx)
```

Get data from the result data buffer.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	idx	Result data buffer index. Valid range: 0-15.

Note

- Make sure that the STORERES bit is set in LESENSE_CHx_EVAL, or the STRSCANRES bit is set in LESENSE_CTRL; otherwise, returns the undefined value.

Returns

- Returns the selected word from the result data buffer.

LESENSE_SensorStateGet

```
uint32_t LESENSE_SensorStateGet (void )
```

Get the current state of the LESENSE sensor.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

Returns the value of LESENSE_SENSORSTATE register that represents the current state of the LESENSE sensor.

LESENSE_RAMPowerDown

```
void LESENSE_RAMPowerDown (void )
```

Shut off the power to the LESENSE RAM, which disables LESENSE.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Shuts off the LESENSE RAM in order to decrease leakage current of MCU if LESENSE is not used in your application.

Note

- Warning! Once LESENSE RAM is powered down, it cannot be powered up again.

LESENSE_IntClear

```
void LESENSE_IntClear (uint32_t flags)
```

Clear one or more pending LESENSE interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending LESENSE interrupt sources to clear. Use a set of interrupt flags OR-ed together to clear multiple interrupt sources of LESENSE module (LESENSE_IF_nnn).

LESENSE_IntEnable

```
void LESENSE_IntEnable (uint32_t flags)
```

Enable one or more LESENSE interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	LESENSE interrupt sources to enable. Use a set of interrupt flags OR-ed together to enable multiple interrupt sources of LESENSE module (LESENSE_IF_nnn).

LESENSE_IntDisable

```
void LESENSE_IntDisable (uint32_t flags)
```

Disable one or more LESENSE interrupts.

Parameters

Type	Direction	Argument Name	Description
Type	Direction	Argument Name	Description
uint32_t	[in]	flags	LESENSE interrupt sources to disable. Use a set of interrupt flags OR-ed together to disable multiple interrupt sources of LESENSE module (LESENSE_IF_nnn).

LESENSE_IntSet

```
void LESENSE_IntSet (uint32_t flags)
```

Set one or more pending LESENSE interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	LESENSE interrupt sources to set to pending. Use a set of interrupt flags OR-ed together to set multiple interrupt sources of LESENSE module (LESENSE_IFS_nnn).

LESENSE_IntGet

```
uint32_t LESENSE_IntGet (void )
```

Get pending LESENSE interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Event bits are not cleared by the use of this function.

Returns

- Pending LESENSE interrupt sources. The OR combination of valid interrupt flags of the LESENSE module (LESENSE_IF_nnn).

LESENSE_IntGetEnabled

```
uint32_t LESENSE_IntGetEnabled (void )
```

Get enabled and pending LESENSE interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Event bits are not cleared by the use of this function.

Returns

- Pending and enabled LESENSE interrupt sources. Return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in LESENSE_IEN_nnn register (LESENSE_IEN_nnn) and
  - the OR combination of valid interrupt flags of LESENSE module (LESENSE_IF_nnn).

Macro Definition Documentation

LESENSE_NUM_DECODER_STATES

```
#define LESENSE_NUM_DECODER_STATES
```

Value:

```
(_LESENSE_DECSTATE_DECSTATE_MASK + 1)
```

Number of decoder states supported by current device.

LESENSE_NUM_CHANNELS

```
#define LESENSE_NUM_CHANNELS
```

Value:

```
16
```

Number of LESENSE channels.

LESENSE_CORECTRL_DESC_DEFAULT

```
#define LESENSE_CORECTRL_DESC_DEFAULT
```

Value:

```
{                                                                                \
  lesenseScanStartPeriodic,/* Start new scan each time the period counter overflows. */          \
  lesensePRSCh0,           /* Default PRS channel is selected. */                 \
  lesenseScanConfDirMap,   /* Direct mapping SCANCONF register usage strategy. */ \
  false,                   /* Do not invert ACMP0 output. */                      \
  false,                   /* Do not invert ACMP1 output. */                      \
  false,                   /* Disable dual sampling. */                           \
  true,                    /* Store scan result after each scan. */               \
  true,                    /* Overwrite result buffer register even if it is full. */ \
  lesenseBufTrigHalf,      /* Trigger interrupt and DMA request if result buffer is half full. */ \
  lesenseDMAWakeUpDisable, /* Do not wake up on DMA from EM2. */                  \
  lesenseBiasModeDontTouch,/* Do not touch bias configuration. */                 \
  true                     /* Keep LESENSE running in debug mode. */              \
}
```

Default configuration for LESENSE_CtrlDesc_TypeDef structure.

LESENSE_TIMECTRL_DESC_DEFAULT

```
#define LESENSE_TIMECTRL_DESC_DEFAULT
```

Value:

```
{                                          \
  0U, /* No sensor interaction delay. */    \
  false /* Do not delay the AUXHFRCO startup. */ \
}
```

Default configuration for LESENSE_TimeCtrlDesc_TypeDef structure.

LESENSE_PERCTRL_DESC_DEFAULT

```
#define LESENSE_PERCTRL_DESC_DEFAULT
```

Value:

```
{                                                                                \
  lesenseDACIfData,         /* DAC channel 0 data is defined by DAC_CH0DATA register */           \
  lesenseDACConvModeDisable,/* LESENSE does not control DAC CH0. */                              \
  lesenseDACOutModeDisable, /* DAC channel 0 output to pin disabled. */                          \
  lesenseDACIfData,         /* DAC channel 1 data is defined by DAC_CH1DATA register */           \
  lesenseDACConvModeDisable,/* LESENSE does not control DAC CH1. */                              \
  lesenseDACOutModeDisable, /* DAC channel 1 output to pin disabled. */                          \
  0U,                       /* DAC prescaling factor of 1 (0+1). */                              \
  lesenseDACRefVdd,         /* DAC uses VDD reference. */                                        \
  lesenseACMPModeMuxThres,  /* LESENSE controls input mux and threshold value of ACMP0.       */ \
  lesenseACMPModeMuxThres,  /* LESENSE controls input mux and threshold value of ACMP1.       */ \
  lesenseWarmupModeKeepWarm /* Keep both ACMPs and DAC powered up when LESENSE is idle. */       \
}
```

Default configuration for LESENSE_PerCtrl_TypeDef structure.

LESENSE_DECCTRL_DESC_DEFAULT

```
#define LESENSE_DECCTRL_DESC_DEFAULT
```

Value:

```
{                                                                                  \
  lesenseDecInputSensorSt, /* SENSORSTATE register is used as input to decoder.        */ \
  0U,                      /* State 0 is the initial state of decoder. */               \
  false,                   /* Disable check of current state. */                        \
  true,                    /* Enable channel x % 16 interrupt on state x change. */     \
  true,                    /* Enable decoder hysteresis on PRS0 output. */              \
  true,                    /* Enable decoder hysteresis on PRS1 output. */              \
  true,                    /* Enable decoder hysteresis on PRS2 output. */              \
  true,                    /* Enable decoder hysteresis on PRS3 output. */              \
  false,                   /* Disable count mode on decoder PRS channels 0 and 1*/      \
  lesensePRSCh0,           /* PRS Channel 0 as input for bit 0 of LESENSE decoder. */   \
  lesensePRSCh1,           /* PRS Channel 1 as input for bit 1 of LESENSE decoder. */   \
  lesensePRSCh2,           /* PRS Channel 2 as input for bit 2 of LESENSE decoder. */   \
  lesensePRSCh3,           /* PRS Channel 3 as input for bit 3 of LESENSE decoder. */   \
}
```

Default configuration for LESENSE_PerCtrl_TypeDef structure.

LESENSE_INIT_DEFAULT

```
#define LESENSE_INIT_DEFAULT
```

Value:

```
{                                                                    \
  .coreCtrl = LESENSE_CORECTRL_DESC_DEFAULT, /* Default core control parameters. */  \
  .timeCtrl = LESENSE_TIMECTRL_DESC_DEFAULT, /* Default time control parameters. */  \
```

```
    .perCtrl  = LESENSE_PERCTRL_DESC_DEFAULT,  /* Default peripheral control parameters. */ \
    .decCtrl  = LESENSE_DECCTRL_DESC_DEFAULT   /* Default decoder control parameters. */    \
  }
```

Default configuration for LESENSE_Init_TypeDef structure.

LESENSE_CH_CONF_DEFAULT

```
#define LESENSE_CH_CONF_DEFAULT
```

Value:

```
  {                                                                                 \
    false,                   /* Disable scan channel. */                            \
    false,                   /* Disable assigned pin on scan channel. */            \
    false,                   /* Disable interrupts on channel. */                   \
    lesenseChPinExDis,       /* Channel pin is disabled during excitation period. */ \
    lesenseChPinIdleDis,     /* Channel pin is disabled during idle period. */      \
    false,                   /* Do not use alternate excitation pins for excitation. */ \
    false,                   /* Disabled to shift results from this channel to decoder register.  */ \
    false,                   /* Disabled to invert scan result bit. */              \
    lesenseStoreSampleDisable,/* Disabled to store counter value in result buffer. */ \
    lesenseClkLF,            /* Use LF clock for excitation timing. */              \
    lesenseClkLF,            /* Use LF clock for sample timing. */                  \
    0x00U,                   /* Excitation time is set to 0(+1) excitation clock cycles. */ \
    0x00U,                   /* Sample delay is set to 0(+1) sample clock cycles. */ \
    0x00U,                   /* Measure delay is set to 0 excitation clock cycles.*/ \
    0x00U,                   /* ACMP threshold has been set to 0. */               \
    lesenseSampleModeACMP,   /* ACMP output will be used in comparison. */          \
    lesenseSetIntNone,       /* No interrupt is generated by the channel. */        \
    0x00U,                   /* Counter threshold has bee set to 0x00. */          \
    lesenseCompModeLess      /* Compare mode has been set to trigger interrupt on "less". */ \
  }
```

Default configuration for the scan channel.

LESENSE_SCAN_CONF_DEFAULT

```
#define LESENSE_SCAN_CONF_DEFAULT
```

Value:

```
  {                                             \
    {                                           \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 0. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 1. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 2. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 3. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 4. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 5. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 6. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 7. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 8. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 9. */  \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 10. */ \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 11. */ \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 12. */ \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 13. */ \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 14. */ \
      LESENSE_CH_CONF_DEFAULT, /* Scan channel 15. */ \
    }                                           \
  }
```

Default configuration for all the sensor channels.

LESENSE_ALTEX_CH_CONF_DEFAULT

```
#define LESENSE_ALTEX_CH_CONF_DEFAULT
```

Value:

```
{                                                                    \
  false,              /* Alternate excitation disabled.*/            \
  lesenseAltExPinIdleDis,/* Alternate excitation pin is disabled in idle. */ \
  false               /* Excite only for corresponding channel. */   \
}
```

Default configuration for the alternate excitation channel.

LESENSE_ALTEX_CONF_DEFAULT

```
#define LESENSE_ALTEX_CONF_DEFAULT
```

Value:

```
{                                                                    \
  lesenseAltExMapACMP,                                               \
  {                                                                  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 0. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 1. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 2. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 3. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 4. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 5. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 6. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 7. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 8. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 9. */  \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 10. */ \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 11. */ \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 12. */ \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 13. */ \
    LESENSE_ALTEX_CH_CONF_DEFAULT, /* Alternate excitation channel 14. */ \
    LESENSE_ALTEX_CH_CONF_DEFAULT  /* Alternate excitation channel 15. */ \
  }                                                                  \
}
```

Default configuration for all the alternate excitation channels.

LESENSE_ST_CONF_DEFAULT

```
#define LESENSE_ST_CONF_DEFAULT
```

Value:

```
{                                                                  \
  0x0FU,              /* Compare value set to 0x0F. */             \
  0x00U,              /* All decoder inputs masked. */             \
  0U,                 /* Next state is state 0. */                 \
  lesenseTransActNone, /* No PRS action performed on compare match. */ \
  false               /* No interrupt triggered on compare match. */  \
}
```

Default configuration for the decoder state condition.

LESENSE_DECODER_CONF_DEFAULT

```
#define LESENSE_DECODER_CONF_DEFAULT
```

Value:

```
{ /* chain |   Descriptor A         |   Descriptor B   */                        \
  {                                                                              \
```

```
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 0. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 1. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 2. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 3. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 4. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 5. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 6. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 7. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 8. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 9. */  \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 10. */ \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 11. */ \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 12. */ \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 13. */ \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }, /* Decoder state 14. */ \
        { false, LESENSE_ST_CONF_DEFAULT, LESENSE_ST_CONF_DEFAULT }  /* Decoder state 15. */ \
      }                                                                                      \
    }
```

Default configuration for all decoder states.

LESENSE_CoreCtrlDesc_TypeDef

Core control (LESENSE_CTRL/CFG) descriptor structure.

Public Attributes

LESENSE_ScanMode_TypeDef	**scanStart** Select scan start mode to control how the scan start is being triggered.
LESENSE_PRSSel_TypeDef	**prsSel** Select PRS source for scan start if scanMode is set to lesensePrsPulse.
LESENSE_ScanConfSel_TypeDef	**scanConfSel** Select scan configuration register usage strategy.
bool	**invACMP0** Set to true to invert ACMP0 output.
bool	**invACMP1** Set to true to invert ACMP1 output.
bool	**dualSample** Set to true to sample both ACMPs simultaneously.
bool	**storeScanRes** Set to true in order to store SCANRES in the RAM (accessible via RESDATA) after each scan.
bool	**bufOverWr** Set to true in order to always make LESENSE write to the result buffer, even if it is full.
LESENSE_BufTrigLevel_TypeDef	**bufTrigLevel** Select trigger conditions for the interrupt and DMA.
LESENSE_DMAWakeUp_TypeDef	**wakeupOnDMA** Configure trigger condition for the DMA wakeup from EM2.
LESENSE_BiasMode_TypeDef	**biasMode** Select bias mode.
bool	**debugRun** Set to true to keep LESENSE running in the debug mode.

Public Attribute Documentation

scanStart

```
LESENSE_ScanMode_TypeDef LESENSE_CoreCtrlDesc_TypeDef::scanStart
```

Select scan start mode to control how the scan start is being triggered.

prsSel

```
LESENSE_PRSSel_TypeDef LESENSE_CoreCtrlDesc_TypeDef::prsSel
```

Select PRS source for scan start if scanMode is set to lesensePrsPulse.

scanConfSel

```
LESENSE_ScanConfSel_TypeDef LESENSE_CoreCtrlDesc_TypeDef::scanConfSel
```

Select scan configuration register usage strategy.

invACMP0

```
bool LESENSE_CoreCtrlDesc_TypeDef::invACMP0
```

Set to true to invert ACMP0 output.

invACMP1

```
bool LESENSE_CoreCtrlDesc_TypeDef::invACMP1
```

Set to true to invert ACMP1 output.

dualSample

```
bool LESENSE_CoreCtrlDesc_TypeDef::dualSample
```

Set to true to sample both ACMPs simultaneously.

storeScanRes

```
bool LESENSE_CoreCtrlDesc_TypeDef::storeScanRes
```

Set to true in order to store SCANRES in the RAM (accessible via RESDATA) after each scan.

bufOverWr

```
bool LESENSE_CoreCtrlDesc_TypeDef::bufOverWr
```

Set to true in order to always make LESENSE write to the result buffer, even if it is full.

bufTrigLevel

```
LESENSE_BufTrigLevel_TypeDef LESENSE_CoreCtrlDesc_TypeDef::bufTrigLevel
```

Select trigger conditions for the interrupt and DMA.

wakeupOnDMA

```
LESENSE_DMAWakeUp_TypeDef LESENSE_CoreCtrlDesc_TypeDef::wakeupOnDMA
```

Configure trigger condition for the DMA wakeup from EM2.

biasMode

```
LESENSE_BiasMode_TypeDef LESENSE_CoreCtrlDesc_TypeDef::biasMode
```

Select bias mode.

debugRun

```
bool LESENSE_CoreCtrlDesc_TypeDef::debugRun
```

Set to true to keep LESENSE running in the debug mode.

LESENSE_TimeCtrlDesc_TypeDef

LESENSE timing control descriptor structure.

Public Attributes

uint8_t	startDelay Set number of LFACLK cycles to delay sensor interaction on each channel.
bool	delayAuxStartup Set to true do delay startup of AUXHFRCO until the system enters excite phase.

Public Attribute Documentation

startDelay

```
uint8_t LESENSE_TimeCtrlDesc_TypeDef::startDelay
```

Set number of LFACLK cycles to delay sensor interaction on each channel.

Valid range: 0-3 (2 bit).

delayAuxStartup

```
bool LESENSE_TimeCtrlDesc_TypeDef::delayAuxStartup
```

Set to true do delay startup of AUXHFRCO until the system enters excite phase.

This will reduce the time AUXHFRCO is enabled and reduce power usage.

LESENSE_PerCtrlDesc_TypeDef

LESENSE peripheral control descriptor structure.

Public Attributes

LESENSE_ControlDACData_TypeDef	dacCh0Data Configure DAC channel 0 data control.
LESENSE_ControlDACConv_TypeDef	dacCh0ConvMode Configure how LESENSE controls conversion on DAC channel 0.
LESENSE_ControlDACOut_TypeDef	dacCh0OutMode Configure how LESENSE controls output on DAC channel 0.
LESENSE_ControlDACData_TypeDef	dacCh1Data Configure DAC channel 1 data control.
LESENSE_ControlDACConv_TypeDef	dacCh1ConvMode Configure how LESENSE controls conversion on DAC channel 1.
LESENSE_ControlDACOut_TypeDef	dacCh1OutMode Configure how LESENSE controls output on DAC channel 1.
uint8_t	dacPresc Configure the prescaling factor for the LESENSE - DAC interface.
LESENSE_DACRef_TypeDef	dacRef Configure the DAC reference to be used.
LESENSE_ControlACMP_TypeDef	acmp0Mode Configure how LESENSE controls ACMP 0.
LESENSE_ControlACMP_TypeDef	acmp1Mode Configure how LESENSE controls ACMP 1.
LESENSE_WarmupMode_TypeDef	warmupMode Configure how LESENSE controls ACMPs and DAC in idle mode.

Public Attribute Documentation

dacCh0Data

```
LESENSE_ControlDACData_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh0Data
```

Configure DAC channel 0 data control.

dacCh0ConvMode

```
LESENSE_ControlDACConv_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh0ConvMode
```

Configure how LESENSE controls conversion on DAC channel 0.

```
LESENSE_ControlDACOut_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh0OutMode
```

Configure how LESENSE controls output on DAC channel 0.

dacCh1Data

```
LESENSE_ControlDACData_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh1Data
```

Configure DAC channel 1 data control.

dacCh1ConvMode

```
LESENSE_ControlDACConv_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh1ConvMode
```

Configure how LESENSE controls conversion on DAC channel 1.

dacCh1OutMode

```
LESENSE_ControlDACOut_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacCh1OutMode
```

Configure how LESENSE controls output on DAC channel 1.

dacPresc

```
uint8_t LESENSE_PerCtrlDesc_TypeDef::dacPresc
```

Configure the prescaling factor for the LESENSE - DAC interface.

Valid range: 0-31 (5-bit).

dacRef

```
LESENSE_DACRef_TypeDef LESENSE_PerCtrlDesc_TypeDef::dacRef
```

Configure the DAC reference to be used.

Set to lesenseDACRefVdd to use VDD and set to lesenseDACRefBandGap to use band gap as reference.

acmp0Mode

```
LESENSE_ControlACMP_TypeDef LESENSE_PerCtrlDesc_TypeDef::acmp0Mode
```

Configure how LESENSE controls ACMP 0.

acmp1Mode

```
LESENSE_ControlACMP_TypeDef LESENSE_PerCtrlDesc_TypeDef::acmp1Mode
```

Configure how LESENSE controls ACMP 1.

warmupMode

```
LESENSE_WarmupMode_TypeDef LESENSE_PerCtrlDesc_TypeDef::warmupMode
```

Configure how LESENSE controls ACMPs and DAC in idle mode.

LESENSE_DecCtrlDesc_TypeDef

LESENSE decoder control descriptor structure.

Public Attributes

LESENSE_DecInput_TypeDef	**decInput** Select input to the LESENSE decoder.
uint32_t	**initState** Initial state of the LESENSE decoder.
bool	**chkState** Set to enable decoder to check the present state in addition to the states defined in TCONF.
bool	**intMap** When set, a transition from state x in decoder will set the interrupt flag CHx.
bool	**hystPRS0** Set to enable hysteresis in decoder for suppressing the changes on PRS channel 0.
bool	**hystPRS1** Set to enable hysteresis in decoder for suppressing the changes on PRS channel 1.
bool	**hystPRS2** Set to enable hysteresis in decoder for suppressing the changes on PRS channel 2.
bool	**hystIRQ** Set to enable hysteresis in decoder for suppressing the interrupt requests.
bool	**prsCount** Set to enable count mode on decoder PRS channels 0 and 1 to produce outputs which can be used by a PCNT to count up or down.
LESENSE_PRSSel_TypeDef	**prsChSel0** Select PRS channel input for bit 0 of LESENSE decoder.
LESENSE_PRSSel_TypeDef	**prsChSel1** Select PRS channel input for bit 1 of LESENSE decoder.
LESENSE_PRSSel_TypeDef	**prsChSel2** Select PRS channel input for bit 2 of LESENSE decoder.
LESENSE_PRSSel_TypeDef	**prsChSel3** Select PRS channel input for bit 3 of LESENSE decoder.

Public Attribute Documentation

decInput

```
LESENSE_DecInput_TypeDef LESENSE_DecCtrlDesc_TypeDef::decInput
```

Select input to the LESENSE decoder.

initState

```
uint32_t LESENSE_DecCtrlDesc_TypeDef::initState
```

Initial state of the LESENSE decoder.

chkState

```
bool LESENSE_DecCtrlDesc_TypeDef::chkState
```

Set to enable decoder to check the present state in addition to the states defined in TCONF.

intMap

```
bool LESENSE_DecCtrlDesc_TypeDef::intMap
```

When set, a transition from state x in decoder will set the interrupt flag CHx.

hystPRS0

```
bool LESENSE_DecCtrlDesc_TypeDef::hystPRS0
```

Set to enable hysteresis in decoder for suppressing the changes on PRS channel 0.

hystPRS1

```
bool LESENSE_DecCtrlDesc_TypeDef::hystPRS1
```

Set to enable hysteresis in decoder for suppressing the changes on PRS channel 1.

hystPRS2

```
bool LESENSE_DecCtrlDesc_TypeDef::hystPRS2
```

Set to enable hysteresis in decoder for suppressing the changes on PRS channel 2.

hystIRQ

```
bool LESENSE_DecCtrlDesc_TypeDef::hystIRQ
```

Set to enable hysteresis in decoder for suppressing the interrupt requests.

prsCount

```
bool LESENSE_DecCtrlDesc_TypeDef::prsCount
```

Set to enable count mode on decoder PRS channels 0 and 1 to produce outputs which can be used by a PCNT to count up or down.

prsChSel0

```
LESENSE_PRSSel_TypeDef LESENSE_DecCtrlDesc_TypeDef::prsChSel0
```

Select PRS channel input for bit 0 of LESENSE decoder.

prsChSel1

```
LESENSE_PRSSel_TypeDef LESENSE_DecCtrlDesc_TypeDef::prsChSel1
```

Select PRS channel input for bit 1 of LESENSE decoder.

prsChSel2

```
LESENSE_PRSSel_TypeDef LESENSE_DecCtrlDesc_TypeDef::prsChSel2
```

Select PRS channel input for bit 2 of LESENSE decoder.

prsChSel3

```
LESENSE_PRSSel_TypeDef LESENSE_DecCtrlDesc_TypeDef::prsChSel3
```

Select PRS channel input for bit 3 of LESENSE decoder.

LESENSE_Init_TypeDef

LESENSE module initialization structure.

Public Attributes

LESENSE_CoreCtrlDesc_TypeDef	coreCtrl LESENSE core configuration parameters.
LESENSE_TimeCtrlDesc_TypeDef	timeCtrl LESENSE timing configuration parameters.
LESENSE_PerCtrlDesc_TypeDef	perCtrl LESENSE peripheral configuration parameters.
LESENSE_DecCtrlDesc_TypeDef	decCtrl LESENSE decoder configuration parameters.

Public Attribute Documentation

coreCtrl

```
LESENSE_CoreCtrlDesc_TypeDef LESENSE_Init_TypeDef::coreCtrl
```

LESENSE core configuration parameters.

timeCtrl

```
LESENSE_TimeCtrlDesc_TypeDef LESENSE_Init_TypeDef::timeCtrl
```

LESENSE timing configuration parameters.

perCtrl

```
LESENSE_PerCtrlDesc_TypeDef LESENSE_Init_TypeDef::perCtrl
```

LESENSE peripheral configuration parameters.

decCtrl

```
LESENSE_DecCtrlDesc_TypeDef LESENSE_Init_TypeDef::decCtrl
```

LESENSE decoder configuration parameters.

LESENSE_ChDesc_TypeDef

Channel descriptor structure.

Public Attributes

bool	enaScanCh Set to enable scan channel CHx.
bool	enaPin Set to enable CHx pin.
bool	enaInt Enable/disable channel interrupts after configuring all the sensor channel parameters.
LESENSE_ChPinExMode_TypeDef	chPinExMode Configure channel pin mode for the excitation phase of the scan sequence.
LESENSE_ChPinIdleMode_TypeDef	chPinIdleMode Configure channel pin idle setup in LESENSE idle phase.
bool	useAltEx Set to use alternate excite pin for excitation.
bool	shiftRes Set to enable result from this channel being shifted into the decoder register.
bool	invRes Set to invert result bit stored in the SCANRES register.
LESENSE_StoreSample_TypeDef	storeCntRes Set to store counter value in the RAM (accessible via RESDATA) and make the comparison result available in the SCANRES register.
LESENSE_ChClk_TypeDef	exClk Select clock used for the excitation timing.
LESENSE_ChClk_TypeDef	sampleClk Select clock used for the sample delay timing.
uint8_t	exTime Configure the excitation time.
uint8_t	sampleDelay Configure the sample delay.
uint16_t	measDelay Configure the measure delay.
uint16_t	acmpThres Configure the ACMP threshold or the DAC data.
LESENSE_ChSampleMode_TypeDef	sampleMode Select if the ACMP output, the ADC output or the counter output should be used in comparison.
LESENSE_ChIntMode_TypeDef	intMode Configure the interrupt generation mode for the CHx interrupt flag.
uint16_t	cntThres Configure the decision threshold for the sensor data comparison.

LESENSE_ChCompMode_TypeDef	compMode Select the mode for counter comparison.

Public Attribute Documentation

enaScanCh

```
bool LESENSE_ChDesc_TypeDef::enaScanCh
```

Set to enable scan channel CHx.

enaPin

```
bool LESENSE_ChDesc_TypeDef::enaPin
```

Set to enable CHx pin.

enaInt

```
bool LESENSE_ChDesc_TypeDef::enaInt
```

Enable/disable channel interrupts after configuring all the sensor channel parameters.

chPinExMode

```
LESENSE_ChPinExMode_TypeDef LESENSE_ChDesc_TypeDef::chPinExMode
```

Configure channel pin mode for the excitation phase of the scan sequence.

Note: OPAOUT is only available on channels 2, 3, 4, and 5.

chPinIdleMode

```
LESENSE_ChPinIdleMode_TypeDef LESENSE_ChDesc_TypeDef::chPinIdleMode
```

Configure channel pin idle setup in LESENSE idle phase.

useAltEx

```
bool LESENSE_ChDesc_TypeDef::useAltEx
```

Set to use alternate excite pin for excitation.

shiftRes

```
bool LESENSE_ChDesc_TypeDef::shiftRes
```

Set to enable result from this channel being shifted into the decoder register.

invRes

```
bool LESENSE_ChDesc_TypeDef::invRes
```

Set to invert result bit stored in the SCANRES register.

storeCntRes

```
LESENSE_StoreSample_TypeDef LESENSE_ChDesc_TypeDef::storeCntRes
```

Set to store counter value in the RAM (accessible via RESDATA) and make the comparison result available in the SCANRES register.

exClk

```
LESENSE_ChClk_TypeDef LESENSE_ChDesc_TypeDef::exClk
```

Select clock used for the excitation timing.

sampleClk

```
LESENSE_ChClk_TypeDef LESENSE_ChDesc_TypeDef::sampleClk
```

Select clock used for the sample delay timing.

exTime

```
uint8_t LESENSE_ChDesc_TypeDef::exTime
```

Configure the excitation time.

Excitation will last exTime+1 excitation clock cycles. Valid range: 0-63 (6 bits).

sampleDelay

```
uint8_t LESENSE_ChDesc_TypeDef::sampleDelay
```

Configure the sample delay.

Sampling will occur after sampleDelay+1 sample clock cycles. Valid range: 0-127 (7 bits) or 0-255 (8 bits) depending on device.

measDelay

```
uint16_t LESENSE_ChDesc_TypeDef::measDelay
```

Configure the measure delay.

Sensor measuring is delayed for measDelay excitation clock cycles. Valid range: 0-127 (7 bits) or 0-1023 (10 bits) depending on device.

acmpThres

```
uint16_t LESENSE_ChDesc_TypeDef::acmpThres
```

Configure the ACMP threshold or the DAC data.

If perCtrl.dacCh0Data or perCtrl.dacCh1Data is set to lesenseDACIfData, acmpThres defines the 12-bit DAC data in the corresponding data register of DAC interface (DACn_CH0DATA and DACn_CH1DATA). In this case, the valid range is: 0-4095 (12 bits). If perCtrl.dacCh0Data or perCtrl.dacCh1Data is set to lesenseACMPThres, acmpThres defines the 6-bit Vdd scaling factor of ACMP negative input (VDDLEVEL in ACMP_INPUTSEL register). In this case, the valid range is: 0-63 (6 bits).

sampleMode

```
LESENSE_ChSampleMode_TypeDef LESENSE_ChDesc_TypeDef::sampleMode
```

Select if the ACMP output, the ADC output or the counter output should be used in comparison.

intMode

```
LESENSE_ChIntMode_TypeDef LESENSE_ChDesc_TypeDef::intMode
```

Configure the interrupt generation mode for the CHx interrupt flag.

cntThres

```
uint16_t LESENSE_ChDesc_TypeDef::cntThres
```

Configure the decision threshold for the sensor data comparison.

Valid range: 0-65535 (16 bits).

compMode

```
LESENSE_ChCompMode_TypeDef LESENSE_ChDesc_TypeDef::compMode
```

Select the mode for counter comparison.

LESENSE_ChAll_TypeDef

Configuration structure for all the scan channels.

Public Attributes

LESENSE_ChDesc_TypeDef	Ch Channel descriptor for all the LESENSE channels.

Public Attribute Documentation

Ch

```
LESENSE_ChDesc_TypeDef LESENSE_ChAll_TypeDef::Ch[LESENSE_NUM_CHANNELS]
```

Channel descriptor for all the LESENSE channels.

LESENSE_AltExDesc_TypeDef

Alternate excitation descriptor structure.

Public Attributes

bool	enablePin Configure alternate excitation pins.
LESENSE_AltExPinIdle_TypeDef	idleConf Configure idle phase setup of alternate excitation pins.
bool	alwaysEx Configure how to control external alternate excitation pins.

Public Attribute Documentation

enablePin

```
bool LESENSE_AltExDesc_TypeDef::enablePin
```

Configure alternate excitation pins.

If set, the corresponding alternate excitation pin/signal is enabled.

idleConf

```
LESENSE_AltExPinIdle_TypeDef LESENSE_AltExDesc_TypeDef::idleConf
```

Configure idle phase setup of alternate excitation pins.

The idleConf parameter is not valid when altExMap==lesenseAltExMapACMP.

alwaysEx

```
bool LESENSE_AltExDesc_TypeDef::alwaysEx
```

Configure how to control external alternate excitation pins.

Only applies if altExMap has been set to lesenseAltExMapALTEX. If true, excitation happens on the corresponding alternate excitation pin during excitation periods of all the enabled channels. If false, excitation happens on the corresponding alternate excitation pin ONLY during excitation period of the corresponding channel. The alwaysEx parameter is not valid when altExMap==lesenseAltExMapACMP.

LESENSE_ConfAltEx_TypeDef

Configuration structure for the alternate excitation.

Public Attributes

LESENSE_AltExMap_TypeDef	altExMap Select alternate excitation mapping.
LESENSE_AltExDesc_TypeDef	AltEx Alternate excitation channel descriptors.

Public Attribute Documentation

altExMap

```
LESENSE_AltExMap_TypeDef LESENSE_ConfAltEx_TypeDef::altExMap
```

Select alternate excitation mapping.

AltEx

```
LESENSE_AltExDesc_TypeDef LESENSE_ConfAltEx_TypeDef::AltEx[16]
```

Alternate excitation channel descriptors.

When altExMap==lesenseAltExMapALTEX, only the 8 first descriptors are used. In this mode they describe the configuration of LES_ALTEX0-7 pins. When altExMap==lesenseAltExMapACMP, all 16 descriptors are used. In this mode they describe the configuration of the 16 possible ACMP0-1 excitation channels. Please refer to the user manual for a complete mapping of routing. NOTE: Some parameters in the descriptors are not valid when altExMap==lesenseAltExMapACMP. Refer to the definition of the LESENSE_AltExDesc_TypeDef structure for details regarding which parameters are valid.

LESENSE_DecStCond_TypeDef

Decoder state condition descriptor structure.

Public Attributes

uint8_t	compVal Configure compare value.
uint8_t	compMask Configure compare mask.
uint8_t	nextState Configure index of state to be entered if the sensor state equals to compVal.
LESENSE_StTransAct_TypeDef	prsAct Configure which PRS action to perform when the sensor state equals to compVal.
bool	setInt If enabled, interrupt flag is set when sensor state equals to compVal.

Public Attribute Documentation

compVal

```
uint8_t LESENSE_DecStCond_TypeDef::compVal
```

Configure compare value.

State transition is triggered when the sensor state equals to this value. Valid range: 0-15 (4 bits).

compMask

```
uint8_t LESENSE_DecStCond_TypeDef::compMask
```

Configure compare mask.

Set bit X to exclude sensor X from evaluation. Note: decoder can handle sensor inputs from up to 4 sensors; therefore, this mask is 4 bit long.

nextState

```
uint8_t LESENSE_DecStCond_TypeDef::nextState
```

Configure index of state to be entered if the sensor state equals to compVal.

Valid range: 0-15 (4 bits).

prsAct

```
LESENSE_StTransAct_TypeDef LESENSE_DecStCond_TypeDef::prsAct
```

Configure which PRS action to perform when the sensor state equals to compVal.

setInt

```
bool LESENSE_DecStCond_TypeDef::setInt
```

If enabled, interrupt flag is set when sensor state equals to compVal.

LESENSE_DecStDesc_TypeDef

Decoder state x configuration structure.

Public Attributes

bool	chainDesc If enabled, the state descriptor pair in next location will also be evaluated.
LESENSE_DecStCond_TypeDef	confA State condition descriptor A (high level descriptor of LESENSE_STx_DECCONFA).
LESENSE_DecStCond_TypeDef	confB State condition descriptor B (high level descriptor of LESENSE_STx_DECCONFB).

Public Attribute Documentation

chainDesc

```
bool LESENSE_DecStDesc_TypeDef::chainDesc
```

If enabled, the state descriptor pair in next location will also be evaluated.

confA

```
LESENSE_DecStCond_TypeDef LESENSE_DecStDesc_TypeDef::confA
```

State condition descriptor A (high level descriptor of LESENSE_STx_DECCONFA).

confB

```
LESENSE_DecStCond_TypeDef LESENSE_DecStDesc_TypeDef::confB
```

State condition descriptor B (high level descriptor of LESENSE_STx_DECCONFB).

LESENSE_DecStAll_TypeDef

Configuration structure for decoder.

Public Attributes

LESENSE_DecStDesc_TypeDef	St Descriptor of the 16 or 32 decoder states depending on the device.

Public Attribute Documentation

St

```
LESENSE_DecStDesc_TypeDef LESENSE_DecStAll_TypeDef::St[LESENSE_NUM_DECODER_STATES]
```

Descriptor of the 16 or 32 decoder states depending on the device.

LETIMER - Low Energy Timer

LETIMER - Low Energy Timer

Low Energy Timer (LETIMER) Peripheral API.

This module contains functions to control the LETIMER peripheral of Silicon Labs 32-bit MCUs and SoCs. The LETIMER is a down-counter that can keep track of time and output configurable waveforms.

Modules

LETIMER_Init_TypeDef

Enumerations

enum LETIMER_RepeatMode_TypeDef {

letimerRepeatFree = _LETIMER_CTRL_REPMODE_FREE
letimerRepeatOneshot = _LETIMER_CTRL_REPMODE_ONESHOT
letimerRepeatBuffered = _LETIMER_CTRL_REPMODE_BUFFERED
letimerRepeatDouble = _LETIMER_CTRL_REPMODE_DOUBLE

}
Repeat mode.

enum LETIMER_UFOA_TypeDef {

letimerUFOANone = _LETIMER_CTRL_UFOA0_NONE
letimerUFOAToggle = _LETIMER_CTRL_UFOA0_TOGGLE
letimerUFOAPulse = _LETIMER_CTRL_UFOA0_PULSE
letimerUFOAPwm = _LETIMER_CTRL_UFOA0_PWM

}
Underflow action on output.

Functions

uint32_t LETIMER_CompareGet(LETIMER_TypeDef *letimer, unsigned int comp)
Get the LETIMER compare register value.

void LETIMER_CompareSet(LETIMER_TypeDef *letimer, unsigned int comp, uint32_t value)
Set the LETIMER compare register value.

uint32_t LETIMER_CounterGet(LETIMER_TypeDef *letimer)
Get LETIMER counter value.

void LETIMER_CounterSet(LETIMER_TypeDef *letimer, uint32_t value)
Set LETIMER counter value.

void LETIMER_Enable(LETIMER_TypeDef *letimer, bool enable)
Start/stop LETIMER.

void LETIMER_FreezeEnable(LETIMER_TypeDef *letimer, bool enable)
LETIMER register synchronization freeze control.

void LETIMER_Init(LETIMER_TypeDef *letimer, const LETIMER_Init_TypeDef *init)
Initialize LETIMER.

void LETIMER_IntClear(LETIMER_TypeDef *letimer, uint32_t flags)
Clear one or more pending LETIMER interrupts.

void LETIMER_IntDisable(LETIMER_TypeDef *letimer, uint32_t flags)
Disable one or more LETIMER interrupts.

void LETIMER_IntEnable(LETIMER_TypeDef *letimer, uint32_t flags)
Enable one or more LETIMER interrupts.

uint32_t LETIMER_IntGet(LETIMER_TypeDef *letimer)
Get pending LETIMER interrupt flags.

uint32_t LETIMER_IntGetEnabled(LETIMER_TypeDef *letimer)
Get enabled and pending LETIMER interrupt flags.

void LETIMER_IntSet(LETIMER_TypeDef *letimer, uint32_t flags)
Set one or more pending LETIMER interrupts from SW.

uint32_t LETIMER_RepeatGet(LETIMER_TypeDef *letimer, unsigned int rep)
Get the LETIMER repeat register value.

void LETIMER_RepeatSet(LETIMER_TypeDef *letimer, unsigned int rep, uint32_t value)
Set the LETIMER repeat counter register value.

void LETIMER_Reset(LETIMER_TypeDef *letimer)
Reset LETIMER to the same state that it was in after a hardware reset.

void LETIMER_SyncWait(LETIMER_TypeDef *letimer)
Wait for the LETIMER to complete all synchronization of register changes and commands.

void LETIMER_TopSet(LETIMER_TypeDef *letimer, uint32_t value)
Set the LETIMER top value.

uint32_t LETIMER_TopGet(LETIMER_TypeDef *letimer)
Get the current LETIMER top value.

Macros

#define LETIMER_INIT_DEFAULT undefined
Default configuration for LETIMER initialization structure.

Enumeration Documentation

LETIMER_RepeatMode_TypeDef

```
LETIMER_RepeatMode_TypeDef
```

Repeat mode.

Enumerator	
letimerRepeatFree	Count until stopped by SW.
letimerRepeatOneshot	Count REP0 times.
letimerRepeatBuffered	Count REP0 times, if REP1 has been written to, it is loaded into REP0 when REP0 is about to be decremented to 0.
letimerRepeatDouble	Run as long as both REP0 and REP1 are not 0.

LETIMER_UFOA_TypeDef

```
LETIMER_UFOA_TypeDef
```

Underflow action on output.

Enumerator	
letimerUFOANone	No output action.
letimerUFOAToggle	Toggle output when counter underflows.
letimerUFOAPulse	Hold output one LETIMER clock cycle when counter underflows.
letimerUFOAPwm	Set output idle when counter underflows, and active when matching COMP1.

Function Documentation

LETIMER_CompareGet

> uint32_t LETIMER_CompareGet (LETIMER_TypeDef * letimer, unsigned int comp)

Get the LETIMER compare register value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
unsigned int	[in]	comp	A compare register to get, either 0 or 1.

Returns

- A compare register value, 0 if invalid register selected.

LETIMER_CompareSet

> void LETIMER_CompareSet (LETIMER_TypeDef * letimer, unsigned int comp, uint32_t value)

Set the LETIMER compare register value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
unsigned int	[in]	comp	A compare register to set, either 0 or 1.
uint32_t	[in]	value	An initialization value (<= 0x0000ffff).

Note

- The setting of a compare register requires synchronization into the low frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the LETIMER_Sync() internal function call.

LETIMER_CounterGet

> uint32_t LETIMER_CounterGet (LETIMER_TypeDef * letimer)

Get LETIMER counter value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to the LETIMER peripheral register block.

Returns

- Current LETIMER counter value.

LETIMER_CounterSet

```
void LETIMER_CounterSet (LETIMER_TypeDef * letimer, uint32_t value)
```

Set LETIMER counter value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to the LETIMER peripheral register block.
uint32_t	[in]	value	New counter value.

LETIMER_Enable

```
void LETIMER_Enable (LETIMER_TypeDef * letimer, bool enable)
```

Start/stop LETIMER.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
bool	[in]	enable	True to enable counting, false to disable.

Note

- The enabling/disabling of the LETIMER modifies the LETIMER CMD register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the LETIMER_Sync() internal function call.

LETIMER_FreezeEnable

```
void LETIMER_FreezeEnable (LETIMER_TypeDef * letimer, bool enable)
```

LETIMER register synchronization freeze control.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.

Type	Direction	Argument Name	Description
bool	[in]	enable	• True - enable freeze, modified registers are not propagated to the LF domain • False - disables freeze, modified registers are propagated to the LF domain

Some LETIMER registers require synchronization into the low-frequency (LF) domain. The freeze feature allows for several such registers to be modified before passing them to the LF domain simultaneously (which takes place when the freeze mode is disabled).

Note

- When enabling freeze mode, this function will wait for all current ongoing LETIMER synchronization to the LF domain to complete (Normally synchronization will not be in progress.) However, for this reason, when using freeze mode, modifications of registers requiring the LF synchronization should be done within one freeze enable/disable block to avoid unecessary stalling.

LETIMER_Init

```
void LETIMER_Init (LETIMER_TypeDef * letimer, const LETIMER_Init_TypeDef * init)
```

Initialize LETIMER.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
const LETIMER_Init_TypeDef *	[in]	init	A pointer to the LETIMER initialization structure.

Note that the compare/repeat values must be set separately with LETIMER_CompareSet() and LETIMER_RepeatSet(). That should probably be done prior using this function if configuring the LETIMER to start when initialization is complete.

Note

- The initialization of the LETIMER modifies the LETIMER CTRL/CMD registers which require synchronization into the low-frequency domain. If any of those registers are modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the LETIMER_Sync() internal function call.

LETIMER_IntClear

```
void LETIMER_IntClear (LETIMER_TypeDef * letimer, uint32_t flags)
```

Clear one or more pending LETIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to LETIMER peripheral register block.

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending LETIMER interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the LETIMER module (LETIMER_IF_nnn).

LETIMER_IntDisable

```
void LETIMER_IntDisable (LETIMER_TypeDef * letimer, uint32_t flags)
```

Disable one or more LETIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to LETIMER peripheral register block.
uint32_t	[in]	flags	LETIMER interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the LETIMER module (LETIMER_IF_nnn).

LETIMER_IntEnable

```
void LETIMER_IntEnable (LETIMER_TypeDef * letimer, uint32_t flags)
```

Enable one or more LETIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to the LETIMER peripheral register block.
uint32_t	[in]	flags	LETIMER interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the LETIMER module (LETIMER_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using LETIMER_IntClear() prior to enabling the interrupt.

LETIMER_IntGet

```
uint32_t LETIMER_IntGet (LETIMER_TypeDef * letimer)
```

Get pending LETIMER interrupt flags.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to LETIMER peripheral register block.

Note

Event bits are not cleared by the use of this function.

Returns

- LETIMER interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the LETIMER module (LETIMER_IF_nnn).

LETIMER_IntGetEnabled

```
uint32_t LETIMER_IntGetEnabled (LETIMER_TypeDef * letimer)
```

Get enabled and pending LETIMER interrupt flags.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to LETIMER peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Event bits are not cleared by the use of this function.

Returns

- Pending and enabled LETIMER interrupt sources. Return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in LETIMER_IEN_nnn register (LETIMER_IEN_nnn) and
  - the OR combination of valid interrupt flags of the LETIMER module (LETIMER_IF_nnn).

LETIMER_IntSet

```
void LETIMER_IntSet (LETIMER_TypeDef * letimer, uint32_t flags)
```

Set one or more pending LETIMER interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	Pointer to LETIMER peripheral register block.
uint32_t	[in]	flags	LETIMER interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for the LETIMER module (LETIMER_IF_nnn).

LETIMER_RepeatGet

```
uint32_t LETIMER_RepeatGet (LETIMER_TypeDef * letimer, unsigned int rep)
```

Get the LETIMER repeat register value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
unsigned int	[in]	rep	Repeat register to get, either 0 or 1.

Returns

- Repeat register value, 0 if invalid register selected.

LETIMER_RepeatSet

```
void LETIMER_RepeatSet (LETIMER_TypeDef * letimer, unsigned int rep, uint32_t value)
```

Set the LETIMER repeat counter register value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
unsigned int	[in]	rep	Repeat counter register to set, either 0 or 1.
uint32_t	[in]	value	An initialization value (<= 0x0000ffff).

Note

- The setting of a repeat counter register requires synchronization into the low-frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the LETIMER_Sync() internal function call.

LETIMER_Reset

```
void LETIMER_Reset (LETIMER_TypeDef * letimer)
```

Reset LETIMER to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.

Note

- The ROUTE register is NOT reset by this function to allow for a centralized setup of this feature.

LETIMER_SyncWait

```
void LETIMER_SyncWait (LETIMER_TypeDef * letimer)
```

Wait for the LETIMER to complete all synchronization of register changes and commands.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.

LETIMER_TopSet

```
void LETIMER_TopSet (LETIMER_TypeDef * letimer, uint32_t value)
```

Set the LETIMER top value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.
uint32_t	[in]	value	The top value. This can be a 16 bit value on series-0 and series-1 devices and a 24 bit value on series-2 devices.

Note

- The LETIMER is a down-counter, so when the counter reaches 0 then the top value will be loaded into the counter. This function can be used to set the top value.

If the LETIMER is not already configured to use a top value then this function will enable that functionality for the user.

LETIMER_TopGet

```
uint32_t LETIMER_TopGet (LETIMER_TypeDef * letimer)
```

Get the current LETIMER top value.

Parameters

Type	Direction	Argument Name	Description
LETIMER_TypeDef *	[in]	letimer	A pointer to the LETIMER peripheral register block.

Returns

- The top value. This will be a 16 bit value on series-0 and series-1 devices and a 24 bit value on series-2 devices.

Macro Definition Documentation

LETIMER_INIT_DEFAULT

```
#define LETIMER_INIT_DEFAULT
```

Value:

```
{                                                                   \
  true,              /* Enable timer when initialization completes. */       \
  false,             /* Stop counter during debug halt. */                   \
  false,             /* Do not start counting on RTC COMP0 match. */         \
  false,             /* Do not start counting on RTC COMP1 match. */         \
  false,             /* Do not load COMP0 into CNT on underflow. */          \
  false,             /* Do not load COMP1 into COMP0 when REP0 reaches 0. */ \
  0,                 /* Idle value 0 for output 0. */                        \
  0,                 /* Idle value 0 for output 1. */                        \
  letimerUFOANone,   /* No action on underflow on output 0. */               \
  letimerUFOANone,   /* No action on underflow on output 1. */               \
  letimerRepeatFree, /* Count until stopped by SW. */                        \
  0                  /* Use default top Value. */                            \
}
```

Default configuration for LETIMER initialization structure.

LETIMER_Init_TypeDef

LETIMER initialization structure.

Public Attributes

bool	**enable** Start counting when initialization completes.
bool	**debugRun** Counter shall keep running during debug halt.
bool	**rtcComp0Enable** Start counting on RTC COMP0 match.
bool	**rtcComp1Enable** Start counting on RTC COMP1 match.
bool	**comp0Top** Load COMP0 register into CNT when counter underflows.
bool	**bufTop** Load COMP1 into COMP0 when REP0 reaches 0.
uint8_t	**out0Pol** Idle value for output 0.
uint8_t	**out1Pol** Idle value for output 1.
LETIMER_UFOA_TypeDef	**ufoa0** Underflow output 0 action.
LETIMER_UFOA_TypeDef	**ufoa1** Underflow output 1 action.
LETIMER_RepeatMode_TypeDef	**repMode** Repeat mode.
uint32_t	**topValue** Top value.

Public Attribute Documentation

enable

```
bool LETIMER_Init_TypeDef::enable
```

Start counting when initialization completes.

debugRun

```
bool LETIMER_Init_TypeDef::debugRun
```

Counter shall keep running during debug halt.

rtcComp0Enable

bool LETIMER_Init_TypeDef::rtcComp0Enable

Start counting on RTC COMP0 match.

rtcComp1Enable

bool LETIMER_Init_TypeDef::rtcComp1Enable

Start counting on RTC COMP1 match.

comp0Top

bool LETIMER_Init_TypeDef::comp0Top

Load COMP0 register into CNT when counter underflows.

bufTop

bool LETIMER_Init_TypeDef::bufTop

Load COMP1 into COMP0 when REP0 reaches 0.

out0Pol

uint8_t LETIMER_Init_TypeDef::out0Pol

Idle value for output 0.

out1Pol

uint8_t LETIMER_Init_TypeDef::out1Pol

Idle value for output 1.

ufoa0

LETIMER_UFOA_TypeDef LETIMER_Init_TypeDef::ufoa0

Underflow output 0 action.

ufoa1

LETIMER_UFOA_TypeDef LETIMER_Init_TypeDef::ufoa1

Underflow output 1 action.

repMode

```
LETIMER_RepeatMode_TypeDef LETIMER_Init_TypeDef::repMode
```

Repeat mode.

topValue

```
uint32_t LETIMER_Init_TypeDef::topValue
```

Top value.

Counter wraps when top value matches counter value is reached.

LEUART - Low Energy UART

LEUART - Low Energy UART

Low Energy Universal Asynchronous Receiver/Transmitter (LEUART) Peripheral API.

This module contains functions to control the LEUART peripheral of Silicon Labs 32-bit MCUs and SoCs. The LEUART module provides the full UART communication using a low frequency 32.768 kHz clock and has special features for communication without the CPU intervention.

Modules

LEUART_Init_TypeDef

Enumerations

enum LEUART_Databits_TypeDef {
  leuartDatabits8 = LEUART_CTRL_DATABITS_EIGHT
  leuartDatabits9 = LEUART_CTRL_DATABITS_NINE
}
Data bit selection.

enum LEUART_Enable_TypeDef {
  leuartDisable = 0x0
  leuartEnableRx = LEUART_CMD_RXEN
  leuartEnableTx = LEUART_CMD_TXEN
  leuartEnable = (LEUART_CMD_RXEN | LEUART_CMD_TXEN)
}
Enable selection.

enum LEUART_Parity_TypeDef {
  leuartNoParity = LEUART_CTRL_PARITY_NONE
  leuartEvenParity = LEUART_CTRL_PARITY_EVEN
  leuartOddParity = LEUART_CTRL_PARITY_ODD
}
Parity selection.

enum LEUART_Stopbits_TypeDef {
  leuartStopbits1 = LEUART_CTRL_STOPBITS_ONE
  leuartStopbits2 = LEUART_CTRL_STOPBITS_TWO
}
Stop bits selection.

Functions

uint32_t LEUART_BaudrateCalc(uint32_t refFreq, uint32_t clkdiv)
Calculate the baudrate for the LEUART given reference frequency and clock division.

uint32_t LEUART_BaudrateGet(LEUART_TypeDef *leuart)
Get the current baudrate for LEUART.

void LEUART_BaudrateSet(LEUART_TypeDef *leuart, uint32_t refFreq, uint32_t baudrate)
Configure the baudrate (or as close as possible to a specified baudrate).

void LEUART_Enable(LEUART_TypeDef *leuart, LEUART_Enable_TypeDef enable)
Enable/disable the LEUART receiver and/or transmitter.

void	LEUART_FreezeEnable(LEUART_TypeDef *leuart, bool enable) LEUART register synchronization freeze control.
void	LEUART_Init(LEUART_TypeDef *leuart, LEUART_Init_TypeDef const *init) Initialize LEUART.
void	LEUART_TxDmaInEM2Enable(LEUART_TypeDef *leuart, bool enable) Enables handling of LEUART TX by DMA in EM2.
void	LEUART_RxDmaInEM2Enable(LEUART_TypeDef *leuart, bool enable) Enables handling of LEUART RX by DMA in EM2.
void	LEUART_IntClear(LEUART_TypeDef *leuart, uint32_t flags) Clear one or more pending LEUART interrupts.
void	LEUART_IntDisable(LEUART_TypeDef *leuart, uint32_t flags) Disable one or more LEUART interrupts.
void	LEUART_IntEnable(LEUART_TypeDef *leuart, uint32_t flags) Enable one or more LEUART interrupts.
uint32_t	LEUART_IntGet(LEUART_TypeDef *leuart) Get pending LEUART interrupt flags.
uint32_t	LEUART_IntGetEnabled(LEUART_TypeDef *leuart) Get enabled and pending LEUART interrupt flags.
void	LEUART_IntSet(LEUART_TypeDef *leuart, uint32_t flags) Set one or more pending LEUART interrupts from SW.
uint32_t	LEUART_StatusGet(LEUART_TypeDef *leuart) Get LEUART STATUS register.
void	LEUART_Reset(LEUART_TypeDef *leuart) Reset LEUART to the same state that it was in after a hardware reset.
uint8_t	LEUART_Rx(LEUART_TypeDef *leuart) Receive one 8 bit frame, (or part of 9 bit frame).
uint16_t	LEUART_RxExt(LEUART_TypeDef *leuart) Receive one 8-9 bit frame with extended information.
void	LEUART_Tx(LEUART_TypeDef *leuart, uint8_t data) Transmit one frame.
void	LEUART_TxExt(LEUART_TypeDef *leuart, uint16_t data) Transmit one 8-9 bit frame with extended control.
uint8_t	LEUART_RxDataGet(LEUART_TypeDef *leuart) Receive one 8 bit frame, (or part of a 9 bit frame).
uint16_t	LEUART_RxDataXGet(LEUART_TypeDef *leuart) Receive one 8-9 bit frame, with extended information.

Macros

#define	LEUART_INIT_DEFAULT undefined Default configuration for LEUART initialization structure.

Enumeration Documentation

LEUART_Databits_TypeDef

LEUART_Databits_TypeDef

Data bit selection.

Enumerator	
leuartDatabits8	8 data bits.
leuartDatabits9	9 data bits.

LEUART_Enable_TypeDef

LEUART_Enable_TypeDef

Enable selection.

Enumerator	
leuartDisable	Disable both receiver and transmitter.
leuartEnableRx	Enable receiver only, transmitter disabled.
leuartEnableTx	Enable transmitter only, receiver disabled.
leuartEnable	Enable both receiver and transmitter.

LEUART_Parity_TypeDef

LEUART_Parity_TypeDef

Parity selection.

Enumerator	
leuartNoParity	No parity.
leuartEvenParity	Even parity.
leuartOddParity	Odd parity.

LEUART_Stopbits_TypeDef

LEUART_Stopbits_TypeDef

Stop bits selection.

Enumerator	
leuartStopbits1	1 stop bits.
leuartStopbits2	2 stop bits.

Function Documentation

LEUART_BaudrateCalc

uint32_t LEUART_BaudrateCalc (uint32_t refFreq, uint32_t clkdiv)

Calculate the baudrate for the LEUART given reference frequency and clock division.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	refFreq	The LEUART peripheral frequency used.
uint32_t	[in]	clkdiv	The clock division factor to be used.

This function returns the baudrate that a LEUART module will use if configured with the given frequency and clock divisor. Notice that this function will not use the hardware configuration. It can be used to determine if a given configuration is sufficiently accurate for the application.

Returns

- A baudrate with given settings.

LEUART_BaudrateGet

```
uint32_t LEUART_BaudrateGet (LEUART_TypeDef * leuart)
```

Get the current baudrate for LEUART.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.

This function returns the actual baudrate (not considering the oscillator inaccuracies) used by the LEUART peripheral.

Returns

- The current baudrate.

LEUART_BaudrateSet

```
void LEUART_BaudrateSet (LEUART_TypeDef * leuart, uint32_t refFreq, uint32_t baudrate)
```

Configure the baudrate (or as close as possible to a specified baudrate).

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
uint32_t	[in]	refFreq	The LEUART reference clock frequency in Hz that will be used. If set to 0, the currently configured reference clock is assumed.
uint32_t	[in]	baudrate	A baudrate to try to achieve for LEUART.

Note

- The baudrate setting requires synchronization into the low-frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed.

LEUART_Enable

```
void LEUART_Enable (LEUART_TypeDef * leuart, LEUART_Enable_TypeDef enable)
```

Enable/disable the LEUART receiver and/or transmitter.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
LEUART_Enable_TypeDef	[in]	enable	Select status for receiver/transmitter.

Notice that this function does not do any configuration. Enabling should normally be done after the initialization is done (if not enabled as part of initialization).

Note

- Enabling/disabling requires synchronization into the low-frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed.

LEUART_FreezeEnable

```
void LEUART_FreezeEnable (LEUART_TypeDef * leuart, bool enable)
```

LEUART register synchronization freeze control.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
bool	[in]	enable	• True - enable freeze, modified registers are not propagated to the LF domain • False - disables freeze, modified registers are propagated to the LF domain

Some LEUART registers require synchronization into the low-frequency (LF) domain. The freeze feature allows for several such registers to be modified before passing them to the LF domain simultaneously (which takes place when the freeze mode is disabled).

Note

- When enabling freeze mode, this function will wait for all current ongoing LEUART synchronization to the LF domain to complete (Normally synchronization will not be in progress.) However, for this reason, when using freeze mode, modifications of registers requiring LF synchronization should be done within one freeze enable/disable block to avoid unnecessary stalling.

LEUART_Init

```
void LEUART_Init (LEUART_TypeDef * leuart, LEUART_Init_TypeDef const * init)
```

Initialize LEUART.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.

Type	Direction	Argument Name	Description
LEUART_Init_TypeDef const *	[in]	init	A pointer to the initialization structure used to configure basic async setup.

This function will configure basic settings to operate in normal asynchronous mode. Consider using LEUART_Reset() prior to this function if the state of configuration is not known, since only configuration settings specified by `init` are set.

Special control setup not covered by this function may be done either before or after using this function (but normally before enabling) by direct modification of the CTRL register.

Notice that pins used by the LEUART module must be properly configured by the user explicitly for the LEUART to work as intended. (When configuring pins consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

Note

- Initializing requires synchronization into the low-frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed.

LEUART_TxDmaInEM2Enable

```
void LEUART_TxDmaInEM2Enable (LEUART_TypeDef * leuart, bool enable)
```

Enables handling of LEUART TX by DMA in EM2.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
bool	[in]	enable	True - enables functionality False - disables functionality

LEUART_RxDmaInEM2Enable

```
void LEUART_RxDmaInEM2Enable (LEUART_TypeDef * leuart, bool enable)
```

Enables handling of LEUART RX by DMA in EM2.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
bool	[in]	enable	True - enables functionality False - disables functionality

LEUART_IntClear

```
void LEUART_IntClear (LEUART_TypeDef * leuart, uint32_t flags)
```

Clear one or more pending LEUART interrupts.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending LEUART interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for LEUART module (LEUART_IF_nnn).

LEUART_IntDisable

```
void LEUART_IntDisable (LEUART_TypeDef * leuart, uint32_t flags)
```

Disable one or more LEUART interrupts.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.
uint32_t	[in]	flags	LEUART interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for LEUART module (LEUART_IF_nnn).

LEUART_IntEnable

```
void LEUART_IntEnable (LEUART_TypeDef * leuart, uint32_t flags)
```

Enable one or more LEUART interrupts.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.
uint32_t	[in]	flags	LEUART interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for LEUART module (LEUART_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using LEUART_IntClear() prior to enabling the interrupt.

LEUART_IntGet

```
uint32_t LEUART_IntGet (LEUART_TypeDef * leuart)
```

Get pending LEUART interrupt flags.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Note

- The event bits are not cleared by the use of this function.

Returns

- LEUART interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for LEUART module (LEUART_IF_nnn).

LEUART_IntGetEnabled

```
uint32_t LEUART_IntGetEnabled (LEUART_TypeDef * leuart)
```

Get enabled and pending LEUART interrupt flags.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled LEUART interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in LEUARTx_IEN_nnn register (LEUARTx_IEN_nnn) and
  - the OR combination of valid interrupt flags of LEUART module (LEUARTx_IF_nnn).

LEUART_IntSet

```
void LEUART_IntSet (LEUART_TypeDef * leuart, uint32_t flags)
```

Set one or more pending LEUART interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.
uint32_t	[in]	flags	LEUART interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for LEUART module (LEUART_IF_nnn).

LEUART_StatusGet

```
uint32_t LEUART_StatusGet (LEUART_TypeDef * leuart)
```

Get LEUART STATUS register.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Returns

STATUS register value.

LEUART_Reset

```
void LEUART_Reset (LEUART_TypeDef * leuart)
```

Reset LEUART to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.

LEUART_Rx

```
uint8_t LEUART_Rx (LEUART_TypeDef * leuart)
```

Receive one 8 bit frame, (or part of 9 bit frame).

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.

This function is normally used to receive one frame when operating with frame length 8 bits. See LEUART_RxExt() for reception of 9 bit frames.

Notice that possible parity/stop bits are not considered a part of the specified frame bit length.

Note

- This function will stall if the buffer is empty until data is received.

Returns

- Data received.

LEUART_RxExt

```
uint16_t LEUART_RxExt (LEUART_TypeDef * leuart)
```

Receive one 8-9 bit frame with extended information.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.

This function is normally used to receive one frame and additional RX status information is required.

Note

- This function will stall if buffer is empty until data is received.

Returns

- Data received.

LEUART_Tx

```
void LEUART_Tx (LEUART_TypeDef * leuart, uint8_t data)
```

Transmit one frame.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
uint8_t	[in]	data	Data to transmit. See details above for more info.

Depending on the frame length configuration, 8 (least significant) bits from `data` are transmitted. If the frame length is 9, 8 bits are transmitted from `data` and one bit as specified by the CTRL register, BIT8DV field. See LEUART_TxExt() for transmitting 9 bit frame with full control of all 9 bits.

Notice that possible parity/stop bits in asynchronous mode are not considered a part of the specified frame bit length.

Note

- This function will stall if buffer is full until the buffer becomes available.

LEUART_TxExt

```
void LEUART_TxExt (LEUART_TypeDef * leuart, uint16_t data)
```

Transmit one 8-9 bit frame with extended control.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	A pointer to the LEUART peripheral register block.
uint16_t	[in]	data	Data to transmit with extended control. Least significant bit contains frame bits and additional control bits are available as documented in the reference manual (set to 0 if not used).

Notice that possible parity/stop bits in asynchronous mode are not considered a part of the specified frame bit length.

Note

- This function will stall if the buffer is full until the buffer becomes available.

LEUART_RxDataGet

```
uint8_t LEUART_RxDataGet (LEUART_TypeDef * leuart)
```

Receive one 8 bit frame, (or part of a 9 bit frame).

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Used to quickly receive one 8 bit frame by reading RXDATA register directly, without checking STATUS register for RXDATAV flag. This can be useful from RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read received data. Please refer to LEUART_RxDataXGet() for reception of 9 bit frames.

Note

- Since this function does not check if the RXDATA register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The LEUART_Rx() is normally a better choice if the validity of the RXDATA register is not certain.
- Notice that possible parity/stop bits are not considered part of specified frame bit length.

Returns

- Data received.

LEUART_RxDataXGet

```
uint16_t LEUART_RxDataXGet (LEUART_TypeDef * leuart)
```

Receive one 8-9 bit frame, with extended information.

Parameters

Type	Direction	Argument Name	Description
LEUART_TypeDef *	[in]	leuart	Pointer to LEUART peripheral register block.

Used to quickly receive one 8-9 bit frame with extended information by reading RXDATAX register directly, without checking STATUS register for RXDATAV flag. This can be useful from RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read received data.

Note

- Since this function does not check if the RXDATAX register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The LEUART_RxExt() is normally a better choice if the validity of the RXDATAX register is not certain.
- Notice that possible parity/stop bits are not considered part of specified frame bit length.

Returns

- Data received.

Macro Definition Documentation

LEUART_INIT_DEFAULT

```
#define LEUART_INIT_DEFAULT
```

Value:

```
{                                                                     \
  leuartEnable,    /* Enable RX/TX when initialization completed. */  \
  0,               /* Use current configured reference clock for configuring baud rate.*/ \
  9600,            /* 9600 bits/s. */                                 \
  leuartDatabits8, /* 8 data bits. */                                 \
  leuartNoParity,  /* No parity. */                                   \
  leuartStopbits1  /* 1 stop bit. */                                  \
}
```

Default configuration for LEUART initialization structure.

LEUART_Init_TypeDef

Initialization structure.

Public Attributes

LEUART_Enable_TypeDef	**enable** Specifies whether TX and/or RX will be enabled when initialization completes.
uint32_t	**refFreq** LEUART reference clock assumed when configuring baud rate setup.
uint32_t	**baudrate** Desired baud rate.
LEUART_Databits_TypeDef	**databits** Number of data bits in frame.
LEUART_Parity_TypeDef	**parity** Parity mode to use.
LEUART_Stopbits_TypeDef	**stopbits** Number of stop bits to use.

Public Attribute Documentation

enable

```
LEUART_Enable_TypeDef LEUART_Init_TypeDef::enable
```

Specifies whether TX and/or RX will be enabled when initialization completes.

refFreq

```
uint32_t LEUART_Init_TypeDef::refFreq
```

LEUART reference clock assumed when configuring baud rate setup.

Set to 0 if using currently configured reference clock.

baudrate

```
uint32_t LEUART_Init_TypeDef::baudrate
```

Desired baud rate.

databits

```
LEUART_Databits_TypeDef LEUART_Init_TypeDef::databits
```

Number of data bits in frame.

parity

```
LEUART_Parity_TypeDef LEUART_Init_TypeDef::parity
```

Parity mode to use.

stopbits

```
LEUART_Stopbits_TypeDef LEUART_Init_TypeDef::stopbits
```

Number of stop bits to use.

MSC - Memory System Controller

MSC - Memory System Controller

Memory System Controller API.

Contains functions to control the MSC, primarily the Flash. Users can perform Flash memory write and erase operations, as well as optimization of the CPU instruction fetch interface for the application. Available instruction fetch features depends on the MCU or SoC family, but features such as instruction pre-fetch, cache, and configurable branch prediction are typically available.

Note

- Flash wait-state configuration is handled by CMU - Clock Management Unit. When core clock configuration is changed by a call to functions such as CMU_ClockSelectSet() or CMU_HFRCOBandSet(), then Flash wait-state configuration is also updated.

MSC resets into a safe state. To initialize the instruction interface to recommended settings:

```
MSC_ExecConfig_TypeDef execConfig = MSC_EXECCONFIG_DEFAULT;
MSC_ExecConfigSet(&execConfig);
```

Note

- The optimal configuration is highly application dependent. Performance benchmarking is supported by most families. See MSC_StartCacheMeasurement() and MSC_GetCacheMeasurement() for more details.
- The flash write and erase runs from RAM on the EFM32G devices. On all other devices the flash write and erase functions run from flash.
- Flash erase may add ms of delay to interrupt latency if executing from Flash.

Flash write and erase operations are supported by MSC_WriteWord(), MSC_ErasePage(), and MSC_MassErase(). Mass erase is supported for MCU and SoC families with larger Flash sizes.

Note

- MSC_Init() must be called prior to any Flash write or erase operation.

The following steps are necessary to perform a page erase and write:

```
uint32_t * userDataPage = (uint32_t *) USERDATA_BASE;
uint32_t userData[] = {
  0x01020304,
  0x05060708
};
MSC_ErasePage(userDataPage);
MSC_WriteWord(userDataPage, userData, sizeof(userData));
```

Note

- The configuration EM_MSC_RUN_FROM_RAM is used to allocate the flash write functions in RAM. By default, flash write functions are placed in RAM on EFM32G and Series 2 devices unless SL_RAMFUNC_DISABLE is defined. For other devices, flash write functions are placed in FLASH by default unless EM_MSC_RUN_FROM_RAM is defined and SL_RAMFUNC_DISABLE is not defined.

Modules

MSC_ExecConfig_TypeDef

Enumerations

enum MSC_Status_TypeDef {

mscReturnOk = 0
mscReturnInvalidAddr = -1
mscReturnLocked = -2
mscReturnTimeOut = -3
mscReturnUnaligned = -4

}
Return codes for writing/erasing Flash.

enum MSC_BusStrategy_Typedef {

mscBusStrategyCPU = MSC_READCTRL_BUSSTRATEGY_CPU
mscBusStrategyDMA = MSC_READCTRL_BUSSTRATEGY_DMA
mscBusStrategyDMAEM1 = MSC_READCTRL_BUSSTRATEGY_DMAEM1
mscBusStrategyNone = MSC_READCTRL_BUSSTRATEGY_NONE

}
Strategy for prioritized bus access.

Functions

bool MSC_LockGetLocked(void)
Get the status of the MSC register lock.

void MSC_LockSetLocked(void)
Set the MSC register lock to a locked state.

void MSC_LockSetUnlocked(void)
Set the MSC register lock to an unlocked state.

uint32_t MSC_ReadCTRLGet(void)
Get the current value of the read control register (MSC_READCTRL).

void MSC_ReadCTRLSet(uint32_t value)
Write a value to the read control register (MSC_READCTRL).

void MSC_IntClear(uint32_t flags)
Clear one or more pending MSC interrupts.

void MSC_IntDisable(uint32_t flags)
Disable one or more MSC interrupts.

void MSC_IntEnable(uint32_t flags)
Enable one or more MSC interrupts.

uint32_t MSC_IntGet(void)
Get pending MSC interrupt flags.

uint32_t MSC_IntGetEnabled(void)
Get enabled and pending MSC interrupt flags.

void MSC_IntSet(uint32_t flags)
Set one or more pending MSC interrupts from SW.

void MSC_StartCacheMeasurement(void)
Start measuring the cache hit ratio.

int32_t MSC_GetCacheMeasurement(void)
Stop measuring the hit rate.

void MSC_FlushCache(void)
Flush contents of instruction cache.

void MSC_EnableCache(bool enable)
Enable or disable instruction cache functionality.

void	MSC_EnableCacheIRQs(bool enable) Enable or disable instruction cache functionality in IRQs.
void	MSC_EnableAutoCacheFlush(bool enable) Enable or disable instruction cache flushing when writing to flash.
void	MSC_BusStrategy(mscBusStrategy_Typedef mode) Configure which unit should get priority on system bus.
void	MSC_ExecConfigSet(MSC_ExecConfig_TypeDef *execConfig) Set the MSC code execution configuration.
MSC_RAMFUNC_DECLARATOR MSC_Status_TypeDef	MSC_WriteWordFast(uint32_t *address, void const *data, uint32_t numBytes) Writes data to flash memory.
SL_RAMFUNC_DECLARATORMSC_Status_TypeDef	MSC_MassErase(void) Erase the entire Flash in one operation.
MSC_RAMFUNC_DECLARATOR MSC_Status_TypeDef	MSC_ErasePage(uint32_t *startAddress) Erases a page in flash memory.
MSC_RAMFUNC_DECLARATOR MSC_Status_TypeDef	MSC_WriteWord(uint32_t *address, void const *data, uint32_t numBytes) Writes data to flash memory.
void	MSC_Init(void) Enables the flash controller for writing.
void	MSC_Deinit(void) Disables the flash controller for writing.

Macros

#define	MSC_PROGRAM_TIMEOUT 10000000UL Timeout used while waiting for Flash to become ready after a write.
#define	MSC_EXECCONFIG_DEFAULT undefined Default MSC ExecConfig initialization.

Enumeration Documentation

MSC_Status_TypeDef

> MSC_Status_TypeDef

Return codes for writing/erasing Flash.

Enumerator	
mscReturnOk	Flash write/erase successful.
mscReturnInvalidAddr	Invalid address.
mscReturnLocked	Flash address is locked.
mscReturnTimeOut	Timeout while writing to Flash.
mscReturnUnaligned	Unaligned access to Flash.

MSC_BusStrategy_Typedef

MSC_BusStrategy_Typedef

Strategy for prioritized bus access.

Enumerator	
mscBusStrategyCPU	Prioritize CPU bus accesses.
mscBusStrategyDMA	Prioritize DMA bus accesses.
mscBusStrategyDMAEM1	Prioritize DMAEM1 for bus accesses.
mscBusStrategyNone	No unit has bus priority.

Function Documentation

MSC_LockGetLocked

bool MSC_LockGetLocked (void)

Get the status of the MSC register lock.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Boolean true if register lock is applied, false otherwise.

MSC_LockSetLocked

void MSC_LockSetLocked (void)

Set the MSC register lock to a locked state.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

MSC_LockSetUnlocked

void MSC_LockSetUnlocked (void)

Set the MSC register lock to an unlocked state.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

MSC_ReadCTRLGet

uint32_t MSC_ReadCTRLGet (void)

Get the current value of the read control register (MSC_READCTRL).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The 32-bit value read from the MSC_READCTRL register.

MSC_ReadCTRLSet

```
void MSC_ReadCTRLSet (uint32_t value)
```

Write a value to the read control register (MSC_READCTRL).

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	value	The 32-bit value to write to the MSC_READCTRL register.

MSC_IntClear

```
void MSC_IntClear (uint32_t flags)
```

Clear one or more pending MSC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending MSC interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the MSC module (MSC_IF_nnn).

MSC_IntDisable

```
void MSC_IntDisable (uint32_t flags)
```

Disable one or more MSC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	MSC interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for the MSC module (MSC_IF_nnn).

MSC_IntEnable

```
void MSC_IntEnable (uint32_t flags)
```

Enable one or more MSC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	MSC interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for the MSC module (MSC_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using MSC_IntClear() prior to enabling the interrupt.

MSC_IntGet

```
uint32_t MSC_IntGet (void )
```

Get pending MSC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- The event bits are not cleared by the use of this function.

Returns

- MSC interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for the MSC module (MSC_IF_nnn).

MSC_IntGetEnabled

```
uint32_t MSC_IntGetEnabled (void )
```

Get enabled and pending MSC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled MSC interrupt sources. The return value is the bitwise AND of
  - the enabled interrupt sources in MSC_IEN and
  - the pending interrupt flags MSC_IF

MSC_IntSet

```
void MSC_IntSet (uint32_t flags)
```

Set one or more pending MSC interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	MSC interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for the MSC module (MSC_IF_nnn).

MSC_StartCacheMeasurement

> void MSC_StartCacheMeasurement (void)

Start measuring the cache hit ratio.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Starts performance counters. It is defined inline to minimize the impact of this code on the measurement itself.

MSC_GetCacheMeasurement

> int32_t MSC_GetCacheMeasurement (void)

Stop measuring the hit rate.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Defined inline to minimize the impact of this code on the measurement itself. Only works for relatively short sections of code. To measure longer sections of code, implement an IRQ Handler for the CHOF and CMOF overflow interrupts. These overflows need to be counted and included in the total. Functions can then be implemented as follows:

```
* volatile uint32_t hitOverflows
* volatile uint32_t missOverflows
*
* void MSC_IRQHandler(void)
* {
*   uint32_t flags;
*   flags = MSC->IF;
*   if (flags & MSC_IF_CHOF) {
*      MSC->IFC = MSC_IF_CHOF;
*      hitOverflows++;
*   }
*   if (flags & MSC_IF_CMOF) {
*     MSC->IFC = MSC_IF_CMOF;
*     missOverflows++;
*   }
* }
*
* void startPerformanceCounters(void)
* {
```

```
*   hitOverflows =0;*   missOverflows =0;**MSC_IntEnable(MSC_IF_CHOF |
MSC_IF_CMOF);*NVIC_EnableIRQ(MSC_IRQn);**MSC_StartCacheMeasurement();*}*
```

Returns

- Returns -1 if there has been no cache accesses. Returns -2 if there has been an overflow in the performance counters. If not, it will return the percentage of hits versus misses.

MSC_FlushCache

```
void MSC_FlushCache (void )
```

Flush contents of instruction cache.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

MSC_EnableCache

```
void MSC_EnableCache (bool enable)
```

Enable or disable instruction cache functionality.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	Enable instruction cache. Default is on.

MSC_EnableCacheIRQs

```
void MSC_EnableCacheIRQs (bool enable)
```

Enable or disable instruction cache functionality in IRQs.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	Enable instruction cache. Default is on.

MSC_EnableAutoCacheFlush

```
void MSC_EnableAutoCacheFlush (bool enable)
```

Enable or disable instruction cache flushing when writing to flash.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	Enable automatic cache flushing. Default is on.

MSC_BusStrategy

```
void MSC_BusStrategy (mscBusStrategy_Typedef mode)
```

Configure which unit should get priority on system bus.

Parameters

Type	Direction	Argument Name	Description
mscBusStrategy_Typedef	[in]	mode	Unit to prioritize bus accesses for.

MSC_ExecConfigSet

```
void MSC_ExecConfigSet (MSC_ExecConfig_TypeDef * execConfig)
```

Set the MSC code execution configuration.

Parameters

Type	Direction	Argument Name	Description
MSC_ExecConfig_TypeDef *	[in]	execConfig	The code execution configuration.

MSC_WriteWordFast

```
MSC_RAMFUNC_DEFINITION_END MSC_RAMFUNC_DEFINITION_BEGIN MSC_Status_TypeDef
MSC_WriteWordFast (uint32_t * address, void const * data, uint32_t numBytes)
```

Writes data to flash memory.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[in]	address	A pointer to the flash word to write to. Must be aligned to words.
void const *	[in]	data	Data to write to flash.
uint32_t	[in]	numBytes	A number of bytes to write from the Flash. NB: Must be divisible by four.

This function is faster than MSC_WriteWord(), but it disables interrupts. Write data must be aligned to words and contain a number of bytes that is divisible by four. Warnings

- This function is only available for certain devices.

Note

- It is recommended to erase the flash page before performing a write. It is required to run this function from RAM on parts that include a flash write buffer.

For IAR Embedded Workbench, Simplicity Studio and GCC, this is done automatically by using attributes in the function proctype. For Keil uVision IDE, define a section called "ram_code" and place this manually in the project's scatter file.

Returns

- Returns the status of the write operation.

```
*   flashReturnOk - The operation completed successfully.
*   flashReturnInvalidAddr - The operation tried to erase a non-flash area.
*   flashReturnLocked - The operation tried to erase a locked area of the flash.
*   flashReturnTimeOut - The operation timed out waiting for flash operation
*       to complete. Or the MSC timed out waiting for the software to write
*       the next word into the DWORD register.
*
```

MSC_MassErase

```
MSC_RAMFUNC_DEFINITION_END SL_RAMFUNC_DEFINITION_BEGIN MSC_Status_TypeDef MSC_MassErase (void
)
```

Erase the entire Flash in one operation.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This command will erase the entire contents of the device. Use with care, both a debug session and all contents of the flash will be lost. The lock bit, MLW will prevent this operation from executing and might prevent a successful mass erase.

Returns

- Returns the status of the operation.

MSC_ErasePage

```
MSC_RAMFUNC_DEFINITION_BEGIN MSC_Status_TypeDef MSC_ErasePage (uint32_t * startAddress)
```

Erases a page in flash memory.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[in]	startAddress	A pointer to the flash page to erase. Must be aligned to the beginning of the page boundary.

Note

- For the Gecko family, it is required to run this function from RAM.

For IAR Embedded Workbench, Simplicity Studio and GCC, this is achieved automatically by using attributes in the function proctype. For Keil uVision IDE, define a section called "ram_code" and place this manually in the project's scatter file.

Returns

- Returns the status of erase operation, MSC_Status_TypeDef

```
*   mscReturnOk - The operation completed successfully.
*   mscReturnInvalidAddr - The operation tried to erase a non-flash area.
*   mscReturnLocked - The operation tried to erase a locked area of the flash.
*   mscReturnTimeOut - The operation timed out waiting for the flash operation
```

```
*       to complete.*
```

MSC_WriteWord

MSC_RAMFUNC_DEFINITION_END MSC_RAMFUNC_DEFINITION_BEGIN MSC_Status_TypeDef MSC_WriteWord (uint32_t * address, void const * data, uint32_t numBytes)

Writes data to flash memory.

Parameters

Type	Direction	Argument Name	Description
uint32_t *	[in]	address	A pointer to the flash word to write to. Must be aligned to words.
void const *	[in]	data	Data to write to flash.
uint32_t	[in]	numBytes	A number of bytes to write from flash. NB: Must be divisible by four.

This function is interrupt-safe, but slower than MSC_WriteWordFast(), which writes to flash with interrupts disabled. Write data must be aligned to words and contain a number of bytes that is divisible by four. Note

- It is recommended to erase the flash page before performing a write.

For the Gecko family, it is required to run this function from RAM.

For IAR Embedded Workbench, Simplicity Studio and GCC, this is done automatically by using attributes in the function proctype. For Keil uVision IDE, define a section called "ram_code" and place it manually in the project's scatter file.

This function requires a system core clock at 1 MHz or higher.

Returns

- Returns the status of the write operation.

```
*   flashReturnOk - The operation completed successfully.
*   flashReturnInvalidAddr - The operation tried to erase a non-flash area.
*   flashReturnLocked - The operation tried to erase a locked area of the Flash.
*   flashReturnTimeOut - The operation timed out waiting for the flash operation
*       to complete, or the MSC module timed out waiting for the software to write
*       the next word into the DWORD register.
*
```

MSC_Init

void MSC_Init (void)

Enables the flash controller for writing.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function must be called before flash operations when AUXHFRCO clock has been changed from a default band.

MSC_Deinit

```
void MSC_Deinit (void )
```

Disables the flash controller for writing.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Macro Definition Documentation

MSC_PROGRAM_TIMEOUT

```
#define MSC_PROGRAM_TIMEOUT
```

Value:

```
10000000UL
```

Timeout used while waiting for Flash to become ready after a write.

This number indicates the number of iterations to perform before issuing a timeout.

Note

- Timeout is set very large (in the order of 100x longer than necessary). This is to avoid any corner case.

MSC_EXECCONFIG_DEFAULT

```
#define MSC_EXECCONFIG_DEFAULT
```

Value:

```
{                           \
  false,                    \
  true,                     \
  false,                    \
  false,                    \
  false,                    \
  false,                    \
}
```

Default MSC ExecConfig initialization.

MSC_ExecConfig_TypeDef

Code execution configuration.

Public Attributes

bool	scbtEn	Enable Suppressed Conditional Branch Target Prefetch.
bool	prefetchEn	Enable MSC prefetching.
bool	ifcDis	Disable instruction cache.
bool	aiDis	Disable automatic cache invalidation on write or erase.
bool	iccDis	Disable automatic caching of fetches in interrupt context.
bool	useHprot	Use ahb_hprot to determine if the instruction is cacheable or not.

Public Attribute Documentation

scbtEn

```
bool MSC_ExecConfig_TypeDef::scbtEn
```

Enable Suppressed Conditional Branch Target Prefetch.

prefetchEn

```
bool MSC_ExecConfig_TypeDef::prefetchEn
```

Enable MSC prefetching.

ifcDis

```
bool MSC_ExecConfig_TypeDef::ifcDis
```

Disable instruction cache.

aiDis

```
bool MSC_ExecConfig_TypeDef::aiDis
```

Disable automatic cache invalidation on write or erase.

iccDis

```
bool MSC_ExecConfig_TypeDef::iccDis
```

Disable automatic caching of fetches in interrupt context.

useHprot

```
bool MSC_ExecConfig_TypeDef::useHprot
```

Use ahb_hprot to determine if the instruction is cacheable or not.

OPAMP - Operational Amplifier

OPAMP - Operational Amplifier

Operational Amplifier (OPAMP) peripheral API.

This module contains functions to:

- OPAMP_Enable() Configure and enable OPAMP.
- OPAMP_Disable() Disable OPAMP.

All OPAMP functions assume that the DAC clock is running. If DAC is not used, the clock can be turned off when OPAMPs are configured.

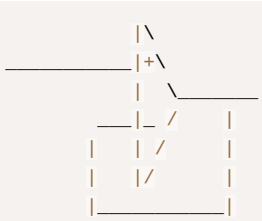
If the available gain values don't suit the application at hand, the resistor ladders can be disabled and external gain programming resistors used.

A number of predefined OPAMP setup macros are available for configuration of the most common OPAMP topologies (see figures below).

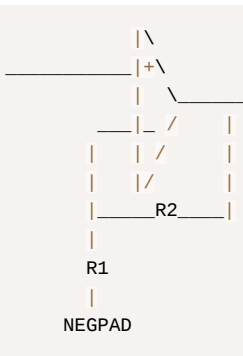
Note

- The terms POSPAD and NEGPAD in the figures are used to indicate that these pads should be connected to a suitable signal ground.

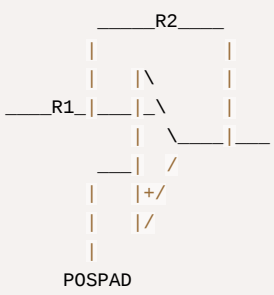
Unity gain voltage follower.
Use predefined macros OPA_INIT_UNITY_GAIN and OPA_INIT_UNITY_GAIN_OPA2.



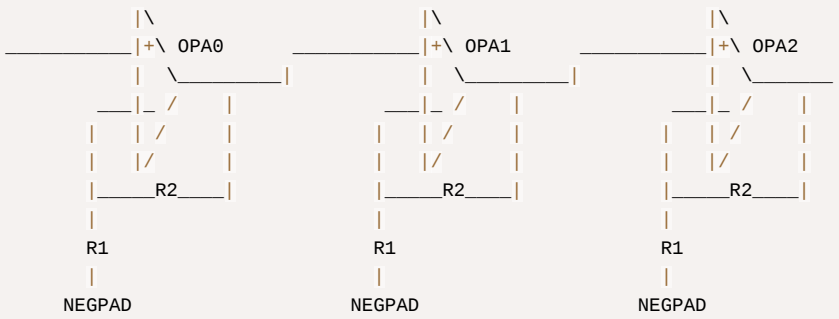
Non-inverting amplifier.
Use predefined macros OPA_INIT_NON_INVERTING and OPA_INIT_NON_INVERTING_OPA2.



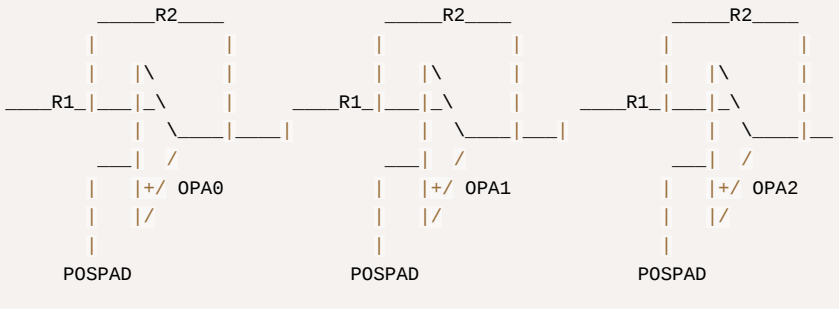
Inverting amplifier.
Use predefined macros OPA_INIT_INVERTING and OPA_INIT_INVERTING_OPA2.



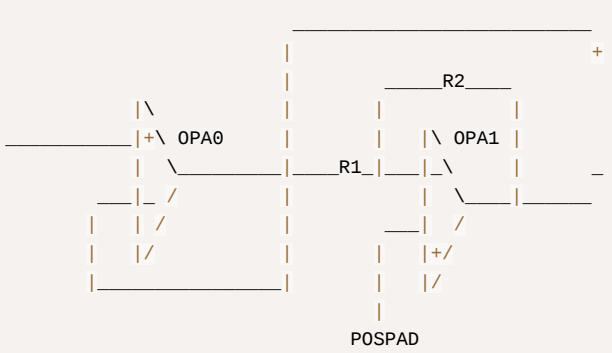
Cascaded non-inverting amplifiers.
Use predefined macros OPA_INIT_CASCADED_NON_INVERTING_OPA0, OPA_INIT_CASCADED_NON_INVERTING_OPA1 and OPA_INIT_CASCADED_NON_INVERTING_OPA2.



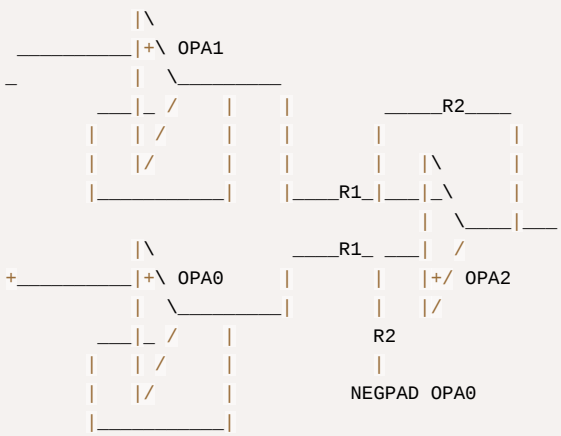
Cascaded inverting amplifiers.
Use predefined macros OPA_INIT_CASCADED_INVERTING_OPA0, OPA_INIT_CASCADED_INVERTING_OPA1 and OPA_INIT_CASCADED_INVERTING_OPA2.



Differential driver with two opamp's.
Use predefined macros OPA_INIT_DIFF_DRIVER_OPA0 and OPA_INIT_DIFF_DRIVER_OPA1.



Differential receiver with three opamp's.
Use predefined macros OPA_INIT_DIFF_RECEIVER_OPA0, OPA_INIT_DIFF_RECEIVER_OPA1 and OPA_INIT_DIFF_RECEIVER_OPA2.



Modules

OPAMP_Init_TypeDef

Enumerations

enum OPAMP_TypeDef {

OPA0 = 0
OPA1 = 1
OPA2 = 2

}
OPAMP selector values.

enum OPAMP_NegSel_TypeDef {

opaNegSelDisable = DAC_OPA0MUX_NEGSEL_DISABLE
opaNegSelUnityGain = DAC_OPA0MUX_NEGSEL_UG
opaNegSelResTap = DAC_OPA0MUX_NEGSEL_OPATAP
opaNegSelNegPad = DAC_OPA0MUX_NEGSEL_NEGPAD

}
OPAMP negative terminal input selection values.

enum OPAMP_PosSel_TypeDef {

opaPosSelDisable = DAC_OPA0MUX_POSSEL_DISABLE
opaPosSelDac = DAC_OPA0MUX_POSSEL_DAC
opaPosSelPosPad = DAC_OPA0MUX_POSSEL_POSPAD
opaPosSelOpaIn = DAC_OPA0MUX_POSSEL_OPA0INP
opaPosSelResTapOpa0 = DAC_OPA0MUX_POSSEL_OPATAP

}
OPAMP positive terminal input selection values.

enum OPAMP_OutMode_TypeDef {

opaOutModeDisable = DAC_OPA0MUX_OUTMODE_DISABLE
opaOutModeMain = DAC_OPA0MUX_OUTMODE_MAIN
opaOutModeAlt = DAC_OPA0MUX_OUTMODE_ALT
opaOutModeAll = DAC_OPA0MUX_OUTMODE_ALL

}
OPAMP output terminal selection values.

enum OPAMP_ResSel_TypeDef {

opaResSelDefault = DAC_OPA0MUX_RESSEL_DEFAULT
opaResSelR2eq0_33R1 = DAC_OPA0MUX_RESSEL_RES0
opaResSelR2eqR1 = DAC_OPA0MUX_RESSEL_RES1
opaResSelR1eq1_67R1 = DAC_OPA0MUX_RESSEL_RES2
opaResSelR2eq2R1 = DAC_OPA0MUX_RESSEL_RES3
opaResSelR2eq3R1 = DAC_OPA0MUX_RESSEL_RES4
opaResSelR2eq4_33R1 = DAC_OPA0MUX_RESSEL_RES5
opaResSelR2eq7R1 = DAC_OPA0MUX_RESSEL_RES6
opaResSelR2eq15R1 = DAC_OPA0MUX_RESSEL_RES7

}
OPAMP gain values.

enum OPAMP_ResInMux_TypeDef {

opaResInMuxDisable = DAC_OPA0MUX_RESINMUX_DISABLE
opaResInMuxOpaIn = DAC_OPA0MUX_RESINMUX_OPA0INP
opaResInMuxNegPad = DAC_OPA0MUX_RESINMUX_NEGPAD
opaResInMuxPosPad = DAC_OPA0MUX_RESINMUX_POSPAD
opaResInMuxVss = DAC_OPA0MUX_RESINMUX_VSS

}
OPAMP resistor ladder input selector values.

Functions

void OPAMP_Disable(DAC_TypeDef *dac, OPAMP_TypeDef opa)
Disable an Operational Amplifier.

void OPAMP_Enable(DAC_TypeDef *dac, OPAMP_TypeDef opa, const OPAMP_Init_TypeDef *init)
Configure and enable an Operational Amplifier.

Macros

#define OPA_INIT_UNITY_GAIN undefined
Configuration of OPA0/1 in unity gain voltage follower mode.

#define OPA_INIT_UNITY_GAIN_OPA2 undefined
Configuration of OPA2 in unity gain voltage follower mode.

#define OPA_INIT_NON_INVERTING undefined
Configuration of OPA0/1 in non-inverting amplifier mode.

#define OPA_INIT_NON_INVERTING_OPA2 undefined
Configuration of OPA2 in non-inverting amplifier mode.

#define OPA_INIT_INVERTING undefined
Configuration of OPA0/1 in inverting amplifier mode.

#define OPA_INIT_INVERTING_OPA2 undefined
Configuration of OPA2 in inverting amplifier mode.

#define OPA_INIT_CASCADED_NON_INVERTING_OPA0 undefined
Configuration of OPA0 in cascaded non-inverting amplifier mode.

#define OPA_INIT_CASCADED_NON_INVERTING_OPA1 undefined
Configuration of OPA1 in cascaded non-inverting amplifier mode.

#define OPA_INIT_CASCADED_NON_INVERTING_OPA2 undefined
Configuration of OPA2 in cascaded non-inverting amplifier mode.

#define OPA_INIT_CASCADED_INVERTING_OPA0 undefined
Configuration of OPA0 in cascaded inverting amplifier mode.

#define OPA_INIT_CASCADED_INVERTING_OPA1 undefined
Configuration of OPA1 in cascaded inverting amplifier mode.

#define OPA_INIT_CASCADED_INVERTING_OPA2 undefined
Configuration of OPA2 in cascaded inverting amplifier mode.

#define OPA_INIT_DIFF_DRIVER_OPA0 undefined
Configuration of OPA0 in two-opamp differential driver mode.

#define OPA_INIT_DIFF_DRIVER_OPA1 undefined
Configuration of OPA1 in two-opamp differential driver mode.

#define OPA_INIT_DIFF_RECEIVER_OPA0 undefined
Configuration of OPA0 in three-opamp differential receiver mode.

#define OPA_INIT_DIFF_RECEIVER_OPA1 undefined
Configuration of OPA1 in three-opamp differential receiver mode.

#define OPA_INIT_DIFF_RECEIVER_OPA2 undefined
Configuration of OPA2 in three-opamp differential receiver mode.

Enumeration Documentation

OPAMP_TypeDef

> OPAMP_TypeDef

OPAMP selector values.

Enumerator	
OPA0	Select OPA0.
OPA1	Select OPA1.
OPA2	Select OPA2.

OPAMP_NegSel_TypeDef

> OPAMP_NegSel_TypeDef

OPAMP negative terminal input selection values.

Enumerator	
opaNegSelDisable	Input disabled.
opaNegSelUnityGain	Unity gain feedback path.
opaNegSelResTap	Feedback resistor ladder tap.
opaNegSelNegPad	Negative pad as input.

OPAMP_PosSel_TypeDef

> OPAMP_PosSel_TypeDef

OPAMP positive terminal input selection values.

Enumerator	
opaPosSelDisable	Input disabled.
opaPosSelDac	DAC as input (not OPA2).
opaPosSelPosPad	Positive pad as input.
opaPosSelOpaIn	Input from OPAx.

opaPosSelResTapOpa0	Feedback resistor ladder tap from OPA0.

OPAMP_OutMode_TypeDef

> OPAMP_OutMode_TypeDef

OPAMP output terminal selection values.

Enumerator	
opaOutModeDisable	OPA output disabled.
opaOutModeMain	Main output to pin enabled.
opaOutModeAlt	Alternate output(s) enabled (not OPA2).
opaOutModeAll	Both main and alternate enabled (not OPA2).

OPAMP_ResSel_TypeDef

> OPAMP_ResSel_TypeDef

OPAMP gain values.

Enumerator	
opaResSelDefault	Default value when resistor ladder is unused.
opaResSelR2eq0_33R1	R2 = 0.33 * R1.
opaResSelR2eqR1	R2 = R1
opaResSelR1eq1_67R1	R2 = 1.67 R1
opaResSelR2eq2R1	R2 = 2 * R1
opaResSelR2eq3R1	R2 = 3 * R1
opaResSelR2eq4_33R1	R2 = 4.33 * R1.
opaResSelR2eq7R1	R2 = 7 * R1
opaResSelR2eq15R1	R2 = 15 * R1

OPAMP_ResInMux_TypeDef

> OPAMP_ResInMux_TypeDef

OPAMP resistor ladder input selector values.

Enumerator	
opaResInMuxDisable	Resistor ladder disabled.
opaResInMuxOpaIn	Input from OPAx.
opaResInMuxNegPad	Input from negative pad.
opaResInMuxPosPad	Input from positive pad.
opaResInMuxVss	Input connected to Vss.

Function Documentation

OPAMP_Disable

void OPAMP_Disable (DAC_TypeDef * dac, OPAMP_TypeDef opa)

Disable an Operational Amplifier.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
OPAMP_TypeDef	[in]	opa	Selects an OPA, valid values are OPA0, OPA1, and OPA2.

OPAMP_Enable

void OPAMP_Enable (DAC_TypeDef * dac, OPAMP_TypeDef opa, const OPAMP_Init_TypeDef * init)

Configure and enable an Operational Amplifier.

Parameters

Type	Direction	Argument Name	Description
DAC_TypeDef *	[in]	dac	A pointer to the DAC peripheral register block.
OPAMP_TypeDef	[in]	opa	Selects an OPA, valid values are OPA0, OPA1, and OPA2.
const OPAMP_Init_TypeDef *	[in]	init	A pointer to a structure containing OPAMP initialization information.

Note

- The value of the alternate output enable bit mask in the OPAMP_Init_TypeDef structure should consist of one or more of the DAC_OPA[opa#]MUX_OUTPEN_OUT[output#] flags (defined in <part_name>_dac.h) OR'ed together.

  For OPA0:
  - DAC_OPA0MUX_OUTPEN_OUT0
  - DAC_OPA0MUX_OUTPEN_OUT1
  - DAC_OPA0MUX_OUTPEN_OUT2
  - DAC_OPA0MUX_OUTPEN_OUT3
  - DAC_OPA0MUX_OUTPEN_OUT4

  For OPA1:
  - DAC_OPA1MUX_OUTPEN_OUT0
  - DAC_OPA1MUX_OUTPEN_OUT1
  - DAC_OPA1MUX_OUTPEN_OUT2
  - DAC_OPA1MUX_OUTPEN_OUT3
  - DAC_OPA1MUX_OUTPEN_OUT4

  For OPA2:
  - DAC_OPA2MUX_OUTPEN_OUT0
  - DAC_OPA2MUX_OUTPEN_OUT1

  E.g:
  init.outPen = DAC_OPA0MUX_OUTPEN_OUT0 | DAC_OPA0MUX_OUTPEN_OUT4;

Macro Definition Documentation

OPA_INIT_UNITY_GAIN

#define OPA_INIT_UNITY_GAIN

Value:

```
{                                                                   \
  opaNegSelUnityGain,           /* Unity gain.                    */ \
  opaPosSelPosPad,              /* Positive input from pad.       */ \
  opaOutModeMain,               /* Main output enabled.           */ \
  opaResSelDefault,             /* Resistor ladder is not used.   */ \
  opaResInMuxDisable,           /* Resistor ladder disabled.      */ \
  0,                            /* No alternate outputs enabled.  */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.        */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.   */ \
  false,                        /* No low pass filter on positive pad. */ \
  false,                        /* No low pass filter on negative pad. */ \
  false,                        /* No nextout output enabled.     */ \
  false,                        /* Negative pad disabled.         */ \
  true,                         /* Positive pad enabled, used as signal input. */ \
  false,                        /* No shorting of inputs.         */ \
  false,                        /* Rail-to-rail input enabled.    */ \
  true,                         /* Use factory calibrated opamp offset. */ \
  0                             /* Opamp offset value (not used). */ \
}
```

Configuration of OPA0/1 in unity gain voltage follower mode.

OPA_INIT_UNITY_GAIN_OPA2

```
#define OPA_INIT_UNITY_GAIN_OPA2
```

Value:

```
{                                                                   \
  opaNegSelUnityGain,           /* Unity gain.                    */ \
  opaPosSelPosPad,              /* Positive input from pad.       */ \
  opaOutModeMain,               /* Main output enabled.           */ \
  opaResSelDefault,             /* Resistor ladder is not used.   */ \
  opaResInMuxDisable,           /* Resistor ladder disabled.      */ \
  DAC_OPA0MUX_OUTPEN_OUT0,      /* Alternate output 0 enabled.    */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.        */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.   */ \
  false,                        /* No low pass filter on positive pad. */ \
  false,                        /* No low pass filter on negative pad. */ \
  false,                        /* No nextout output enabled.     */ \
  false,                        /* Negative pad disabled.         */ \
  true,                         /* Positive pad enabled, used as signal input. */ \
  false,                        /* No shorting of inputs.         */ \
  false,                        /* Rail-to-rail input enabled.    */ \
  true,                         /* Use factory calibrated opamp offset. */ \
  0                             /* Opamp offset value (not used). */ \
}
```

Configuration of OPA2 in unity gain voltage follower mode.

OPA_INIT_NON_INVERTING

```
#define OPA_INIT_NON_INVERTING
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap. */ \
  opaPosSelPosPad,              /* Positive input from pad.       */ \
  opaOutModeMain,               /* Main output enabled.           */ \
  opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                    */ \
  opaResInMuxNegPad,            /* Resistor ladder input from negative pad. */ \
  0,                            /* No alternate outputs enabled.  */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.        */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.   */ \
  false,                        /* No low pass filter on positive pad. */ \
  false,                        /* No low pass filter on negative pad. */ \
  false,                        /* No nextout output enabled.     */ \
  true,                         /* Negative pad enabled, used as signal ground. */ \
  true,                         /* Positive pad enabled, used as signal input. */ \
  false,                        /* No shorting of inputs.         */ \
  false,                        /* Rail-to-rail input enabled.    */ \
  true,                         /* Use factory calibrated opamp offset. */ \
```

```
  0                           /* Opamp offset value (not used).              */ \
  }
```

Configuration of OPA0/1 in non-inverting amplifier mode.

OPA_INIT_NON_INVERTING_OPA2

```
#define OPA_INIT_NON_INVERTING_OPA2
```

Value:

```
  {                                                                           \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelPosPad,              /* Positive input from pad.                     */ \
    opaOutModeMain,               /* Main output enabled.                         */ \
    opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                                  */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    DAC_OPA0MUX_OUTPEN_OUT0,      /* Alternate output 0 enabled.                  */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    false,                        /* No nextout output enabled.                   */ \
    true,                         /* Negative pad enabled, used as signal ground. */ \
    true,                         /* Positive pad enabled, used as signal input.  */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA2 in non-inverting amplifier mode.

OPA_INIT_INVERTING

```
#define OPA_INIT_INVERTING
```

Value:

```
  {                                                                           \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelPosPad,              /* Positive input from pad.                     */ \
    opaOutModeMain,               /* Main output enabled.                         */ \
    opaResSelR2eqR1,              /* R2 = R1                                      */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    0,                            /* No alternate outputs enabled.                */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    false,                        /* No nextout output enabled.                   */ \
    true,                         /* Negative pad enabled, used as signal input.  */ \
    true,                         /* Positive pad enabled, used as signal ground. */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA0/1 in inverting amplifier mode.

OPA_INIT_INVERTING_OPA2

```
#define OPA_INIT_INVERTING_OPA2
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
  opaPosSelPosPad,              /* Positive input from pad.                     */ \
  opaOutModeMain,               /* Main output enabled.                         */ \
  opaResSelR2eqR1,              /* R2 = R1                                      */ \
  opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
  DAC_OPA0MUX_OUTPEN_OUT0,      /* Alternate output 0 enabled.                  */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  false,                        /* No nextout output enabled.                   */ \
  true,                         /* Negative pad enabled, used as signal input.  */ \
  true,                         /* Positive pad enabled, used as signal ground. */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
}
```

Configuration of OPA2 in inverting amplifier mode.

OPA_INIT_CASCADED_NON_INVERTING_OPA0

```
#define OPA_INIT_CASCADED_NON_INVERTING_OPA0
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
  opaPosSelPosPad,              /* Positive input from pad.                     */ \
  opaOutModeAll,                /* Both main and alternate outputs.             */ \
  opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                                  */ \
  opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
  0,                            /* No alternate outputs enabled.                */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  true,                         /* Pass output to next stage (OPA1).            */ \
  true,                         /* Negative pad enabled, used as signal ground. */ \
  true,                         /* Positive pad enabled, used as signal input.  */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
}
```

Configuration of OPA0 in cascaded non-inverting amplifier mode.

OPA_INIT_CASCADED_NON_INVERTING_OPA1

```
#define OPA_INIT_CASCADED_NON_INVERTING_OPA1
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
  opaPosSelOpaIn,               /* Positive input from OPA0 output.             */ \
  opaOutModeAll,                /* Both main and alternate outputs.             */ \
  opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                                  */ \
  opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
  0,                            /* No alternate outputs enabled.                */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  true,                         /* Pass output to next stage (OPA2).            */ \
  true,                         /* Negative pad enabled, used as signal ground. */ \
  false,                        /* Positive pad disabled.                       */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
```

```
  }
```

Configuration of OPA1 in cascaded non-inverting amplifier mode.

OPA_INIT_CASCADED_NON_INVERTING_OPA2

```
#define OPA_INIT_CASCADED_NON_INVERTING_OPA2
```

Value:

```
  {                                                                 \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelOpaIn,               /* Positive input from OPA1 output.             */ \
    opaOutModeMain,               /* Main output enabled.                         */ \
    opaResSelR2eq0_33R1,          /* R2 = 1/3 R1                                  */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    DAC_OPA0MUX_OUTPEN_OUT0,      /* Alternate output 0 enabled.                  */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    false,                        /* No nextout output enabled.                   */ \
    true,                         /* Negative pad enabled, used as signal ground. */ \
    false,                        /* Positive pad disabled.                       */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA2 in cascaded non-inverting amplifier mode.

OPA_INIT_CASCADED_INVERTING_OPA0

```
#define OPA_INIT_CASCADED_INVERTING_OPA0
```

Value:

```
  {                                                                 \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelPosPad,              /* Positive input from pad.                     */ \
    opaOutModeAll,                /* Both main and alternate outputs.             */ \
    opaResSelR2eqR1,              /* R2 = R1                                      */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    0,                            /* No alternate outputs enabled.                */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    true,                         /* Pass output to next stage (OPA1).            */ \
    true,                         /* Negative pad enabled, used as signal input.  */ \
    true,                         /* Positive pad enabled, used as signal ground. */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA0 in cascaded inverting amplifier mode.

OPA_INIT_CASCADED_INVERTING_OPA1

```
#define OPA_INIT_CASCADED_INVERTING_OPA1
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
  opaPosSelPosPad,              /* Positive input from pad.                     */ \
  opaOutModeAll,                /* Both main and alternate outputs.             */ \
  opaResSelR2eqR1,              /* R2 = R1                                      */ \
  opaResInMuxOpaIn,             /* Resistor ladder input from OPA0.             */ \
  0,                            /* No alternate outputs enabled.                */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  true,                         /* Pass output to next stage (OPA2).            */ \
  false,                        /* Negative pad disabled.                       */ \
  true,                         /* Positive pad enabled, used as signal ground. */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
}
```

Configuration of OPA1 in cascaded inverting amplifier mode.

OPA_INIT_CASCADED_INVERTING_OPA2

```
#define OPA_INIT_CASCADED_INVERTING_OPA2
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
  opaPosSelPosPad,              /* Positive input from pad.                     */ \
  opaOutModeMain,               /* Main output enabled.                         */ \
  opaResSelR2eqR1,              /* R2 = R1                                      */ \
  opaResInMuxOpaIn,             /* Resistor ladder input from OPA1.             */ \
  DAC_OPA0MUX_OUTPEN_OUT0,      /* Alternate output 0 enabled.                  */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  false,                        /* No nextout output enabled.                   */ \
  false,                        /* Negative pad disabled.                       */ \
  true,                         /* Positive pad enabled, used as signal ground. */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
}
```

Configuration of OPA2 in cascaded inverting amplifier mode.

OPA_INIT_DIFF_DRIVER_OPA0

```
#define OPA_INIT_DIFF_DRIVER_OPA0
```

Value:

```
{                                                                   \
  opaNegSelUnityGain,           /* Unity gain.                                  */ \
  opaPosSelPosPad,              /* Positive input from pad.                     */ \
  opaOutModeAll,                /* Both main and alternate outputs.             */ \
  opaResSelDefault,             /* Resistor ladder is not used.                 */ \
  opaResInMuxDisable,           /* Resistor ladder disabled.                    */ \
  0,                            /* No alternate outputs enabled.                */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
  false,                        /* No low pass filter on positive pad.          */ \
  false,                        /* No low pass filter on negative pad.          */ \
  true,                         /* Pass output to next stage (OPA1).            */ \
  false,                        /* Negative pad disabled.                       */ \
  true,                         /* Positive pad enabled, used as signal input.  */ \
  false,                        /* No shorting of inputs.                       */ \
  false,                        /* Rail-to-rail input enabled.                  */ \
  true,                         /* Use factory calibrated opamp offset.         */ \
  0                             /* Opamp offset value (not used).               */ \
```

```
  }
```

Configuration of OPA0 in two-opamp differential driver mode.

OPA_INIT_DIFF_DRIVER_OPA1

```
#define OPA_INIT_DIFF_DRIVER_OPA1
```

Value:

```
  {                                                                 \
    opaNegSelResTap,              /* Negative input from resistor ladder tap.     */ \
    opaPosSelPosPad,              /* Positive input from pad.                     */ \
    opaOutModeMain,               /* Main output enabled.                         */ \
    opaResSelR2eqR1,              /* R2 = R1                                      */ \
    opaResInMuxOpaIn,             /* Resistor ladder input from OPA0.             */ \
    0,                            /* No alternate outputs enabled.                */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    false,                        /* No nextout output enabled.                   */ \
    false,                        /* Negative pad disabled.                       */ \
    true,                         /* Positive pad enabled, used as signal ground. */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA1 in two-opamp differential driver mode.

OPA_INIT_DIFF_RECEIVER_OPA0

```
#define OPA_INIT_DIFF_RECEIVER_OPA0
```

Value:

```
  {                                                                 \
    opaNegSelUnityGain,           /* Unity gain.                                  */ \
    opaPosSelPosPad,              /* Positive input from pad.                     */ \
    opaOutModeAll,                /* Both main and alternate outputs.             */ \
    opaResSelR2eqR1,              /* R2 = R1                                      */ \
    opaResInMuxNegPad,            /* Resistor ladder input from negative pad.     */ \
    0,                            /* No alternate outputs enabled.                */ \
    _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.                      */ \
    _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.                 */ \
    false,                        /* No low pass filter on positive pad.          */ \
    false,                        /* No low pass filter on negative pad.          */ \
    true,                         /* Pass output to next stage (OPA2).            */ \
    true,                         /* Negative pad enabled, used as signal ground. */ \
    true,                         /* Positive pad enabled, used as signal input.  */ \
    false,                        /* No shorting of inputs.                       */ \
    false,                        /* Rail-to-rail input enabled.                  */ \
    true,                         /* Use factory calibrated opamp offset.         */ \
    0                             /* Opamp offset value (not used).               */ \
  }
```

Configuration of OPA0 in three-opamp differential receiver mode.

OPA_INIT_DIFF_RECEIVER_OPA1

```
#define OPA_INIT_DIFF_RECEIVER_OPA1
```

Value:

```
{                                                                   \
  opaNegSelUnityGain,           /* Unity gain.                    */ \
  opaPosSelPosPad,              /* Positive input from pad.       */ \
  opaOutModeAll,                /* Both main and alternate outputs.*/ \
  opaResSelDefault,             /* Resistor ladder is not used.   */ \
  opaResInMuxDisable,           /* Disable resistor ladder.       */ \
  0,                            /* No alternate outputs enabled.  */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.        */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.   */ \
  false,                        /* No low pass filter on positive pad.*/ \
  false,                        /* No low pass filter on negative pad.*/ \
  true,                         /* Pass output to next stage (OPA2).*/ \
  false,                        /* Negative pad disabled.         */ \
  true,                         /* Positive pad enabled, used as signal input.*/ \
  false,                        /* No shorting of inputs.         */ \
  false,                        /* Rail-to-rail input enabled.    */ \
  true,                         /* Use factory calibrated opamp offset.*/ \
  0                             /* Opamp offset value (not used). */ \
}
```

Configuration of OPA1 in three-opamp differential receiver mode.

OPA_INIT_DIFF_RECEIVER_OPA2

```
#define OPA_INIT_DIFF_RECEIVER_OPA2
```

Value:

```
{                                                                   \
  opaNegSelResTap,              /* Input from resistor ladder tap.*/ \
  opaPosSelResTapOpa0,          /* Input from OPA0 resistor ladder tap.*/ \
  opaOutModeMain,               /* Main output enabled.           */ \
  opaResSelR2eqR1,              /* R2 = R1                        */ \
  opaResInMuxOpaIn,             /* Resistor ladder input from OPA1.*/ \
  DAC_OPA0MUX_OUTPEN_OUT0,      /* Enable alternate output 0.     */ \
  _DAC_BIASPROG_BIASPROG_DEFAULT, /* Default bias setting.        */ \
  _DAC_BIASPROG_HALFBIAS_DEFAULT, /* Default half-bias setting.   */ \
  false,                        /* No low pass filter on positive pad.*/ \
  false,                        /* No low pass filter on negative pad.*/ \
  false,                        /* No nextout output enabled.     */ \
  false,                        /* Negative pad disabled.         */ \
  false,                        /* Positive pad disabled.         */ \
  false,                        /* No shorting of inputs.         */ \
  false,                        /* Rail-to-rail input enabled.    */ \
  true,                         /* Use factory calibrated opamp offset.*/ \
  0                             /* Opamp offset value (not used). */ \
}
```

Configuration of OPA2 in three-opamp differential receiver mode.

OPAMP_Init_TypeDef

OPAMP init structure.

Public Attributes

OPAMP_NegSel_TypeDef	**negSel** Select input source for negative terminal.
OPAMP_PosSel_TypeDef	**posSel** Select input source for positive terminal.
OPAMP_OutMode_TypeDef	**outMode** Output terminal connection.
OPAMP_ResSel_TypeDef	**resSel** Select R2/R1 resistor ratio.
OPAMP_ResInMux_TypeDef	**resInMux** Select input source for resistor ladder.
uint32_t	**outPen** Alternate output enable bit mask.
uint32_t	**bias** Set OPAMP bias current.
bool	**halfBias** Divide OPAMP bias current by 2.
bool	**lpfPosPadDisable** Disable low pass filter on positive pad.
bool	**lpfNegPadDisable** Disable low pass filter on negative pad.
bool	**nextOut** Enable NEXTOUT signal source.
bool	**npEn** Enable positive pad.
bool	**ppEn** Enable negative pad.
bool	**shortInputs** Short OPAMP input terminals.
bool	**hcmDisable** Disable input rail-to-rail capability.
bool	**defaultOffset** Use factory calibrated opamp offset value.
uint32_t	**offset** Opamp offset value when defaultOffset is false.

Public Attribute Documentation

negSel

```
OPAMP_NegSel_TypeDef OPAMP_Init_TypeDef::negSel
```

Select input source for negative terminal.

posSel

```
OPAMP_PosSel_TypeDef OPAMP_Init_TypeDef::posSel
```

Select input source for positive terminal.

outMode

```
OPAMP_OutMode_TypeDef OPAMP_Init_TypeDef::outMode
```

Output terminal connection.

resSel

```
OPAMP_ResSel_TypeDef OPAMP_Init_TypeDef::resSel
```

Select R2/R1 resistor ratio.

resInMux

```
OPAMP_ResInMux_TypeDef OPAMP_Init_TypeDef::resInMux
```

Select input source for resistor ladder.

outPen

```
uint32_t OPAMP_Init_TypeDef::outPen
```

Alternate output enable bit mask.

This value should consist one or more of the DAC_OPA[opa#]MUX_OUTPEN_OUT[output#] flags (defined in <part_name>_dac.h) OR'ed together.

For OPA0:

- DAC_OPA0MUX_OUTPEN_OUT0
- DAC_OPA0MUX_OUTPEN_OUT1
- DAC_OPA0MUX_OUTPEN_OUT2
- DAC_OPA0MUX_OUTPEN_OUT3

- DAC_OPA0MUX_OUTPEN_OUT4

For OPA1:

- DAC_OPA1MUX_OUTPEN_OUT0
- DAC_OPA1MUX_OUTPEN_OUT1
- DAC_OPA1MUX_OUTPEN_OUT2
- DAC_OPA1MUX_OUTPEN_OUT3
- DAC_OPA1MUX_OUTPEN_OUT4

For OPA2:

- DAC_OPA2MUX_OUTPEN_OUT0
- DAC_OPA2MUX_OUTPEN_OUT1

E.g:
init.outPen = DAC_OPA0MUX_OUTPEN_OUT0 | DAC_OPA0MUX_OUTPEN_OUT2 | DAC_OPA0MUX_OUTPEN_OUT4;

bias

> uint32_t OPAMP_Init_TypeDef::bias

Set OPAMP bias current.

halfBias

> bool OPAMP_Init_TypeDef::halfBias

Divide OPAMP bias current by 2.

lpfPosPadDisable

> bool OPAMP_Init_TypeDef::lpfPosPadDisable

Disable low pass filter on positive pad.

lpfNegPadDisable

> bool OPAMP_Init_TypeDef::lpfNegPadDisable

Disable low pass filter on negative pad.

nextOut

bool OPAMP_Init_TypeDef::nextOut

Enable NEXTOUT signal source.

npEn

bool OPAMP_Init_TypeDef::npEn

Enable positive pad.

ppEn

bool OPAMP_Init_TypeDef::ppEn

Enable negative pad.

shortInputs

bool OPAMP_Init_TypeDef::shortInputs

Short OPAMP input terminals.

hcmDisable

bool OPAMP_Init_TypeDef::hcmDisable

Disable input rail-to-rail capability.

defaultOffset

bool OPAMP_Init_TypeDef::defaultOffset

Use factory calibrated opamp offset value.

offset

```
uint32_t OPAMP_Init_TypeDef::offset
```

Opamp offset value when defaultOffset is false.

PCNT - Pulse Counter

PCNT - Pulse Counter

Pulse Counter (PCNT) Peripheral API.

This module contains functions to control the PCNT peripheral of Silicon Labs 32-bit MCUs and SoCs. The PCNT decodes incoming pulses. The module has a quadrature mode which may be used to decode the speed and direction of a mechanical shaft.

Modules

PCNT_Init_TypeDef

Enumerations

enum PCNT_Mode_TypeDef {

pcntModeDisable = _PCNT_CTRL_MODE_DISABLE
pcntModeOvsSingle = _PCNT_CTRL_MODE_OVSSINGLE
pcntModeExtSingle = _PCNT_CTRL_MODE_EXTCLKSINGLE
pcntModeExtQuad = _PCNT_CTRL_MODE_EXTCLKQUAD

}
Mode selection.

enum PCNT_CntEvent_TypeDef {

pcntCntEventBoth = _PCNT_CTRL_CNTEV_BOTH
pcntCntEventUp = _PCNT_CTRL_CNTEV_UP
pcntCntEventDown = _PCNT_CTRL_CNTEV_DOWN
pcntCntEventNone = _PCNT_CTRL_CNTEV_NONE

}
Counter event selection.

enum PCNT_PRSSel_TypeDef {

pcntPRSCh0 = 0
pcntPRSCh1 = 1
pcntPRSCh2 = 2
pcntPRSCh3 = 3
pcntPRSCh4 = 4
pcntPRSCh5 = 5
pcntPRSCh6 = 6
pcntPRSCh7 = 7
pcntPRSCh8 = 8
pcntPRSCh9 = 9
pcntPRSCh10 = 10
pcntPRSCh11 = 11

}
PRS sources for `s0PRS` and `s1PRS`.

enum PCNT_PRSInput_TypeDef {

pcntPRSInputS0 = 0
pcntPRSInputS1 = 1

}
PRS inputs of PCNT.

Functions

void PCNT_CounterReset(PCNT_TypeDef *pcnt)
Reset PCNT counters and TOP register.

void PCNT_Enable(PCNT_TypeDef *pcnt, PCNT_Mode_TypeDef mode)
Set PCNT operational mode.

bool PCNT_IsEnabled(PCNT_TypeDef *pcnt)
Returns if the PCNT module is enabled or not.

void PCNT_CounterTopSet(PCNT_TypeDef *pcnt, uint32_t count, uint32_t top)
Set the counter and top values.

void PCNT_PRSInputEnable(PCNT_TypeDef *pcnt, PCNT_PRSInput_TypeDef prsInput, bool enable)
Enable/disable the selected PRS input of PCNT.

void PCNT_FreezeEnable(PCNT_TypeDef *pcnt, bool enable)
PCNT register synchronization freeze control.

void PCNT_Init(PCNT_TypeDef *pcnt, const PCNT_Init_TypeDef *init)
Initialize the pulse counter.

void PCNT_Reset(PCNT_TypeDef *pcnt)
Reset PCNT to the same state that it was in after a hardware reset.

void PCNT_TopBufferSet(PCNT_TypeDef *pcnt, uint32_t val)
Set top buffer value.

void PCNT_TopSet(PCNT_TypeDef *pcnt, uint32_t val)
Set the top value.

uint32_t PCNT_CounterGet(PCNT_TypeDef *pcnt)
Default configuration for PCNT initialization structure.

uint32_t PCNT_AuxCounterGet(PCNT_TypeDef *pcnt)
Get the auxiliary counter value.

void PCNT_CounterSet(PCNT_TypeDef *pcnt, uint32_t count)
Set a counter value.

void PCNT_IntClear(PCNT_TypeDef *pcnt, uint32_t flags)
Clear one or more pending PCNT interrupts.

void PCNT_IntDisable(PCNT_TypeDef *pcnt, uint32_t flags)
Disable one or more PCNT interrupts.

void PCNT_IntEnable(PCNT_TypeDef *pcnt, uint32_t flags)
Enable one or more PCNT interrupts.

uint32_t PCNT_IntGet(PCNT_TypeDef *pcnt)
Get pending PCNT interrupt flags.

uint32_t PCNT_IntGetEnabled(PCNT_TypeDef *pcnt)
Get enabled and pending PCNT interrupt flags.

void PCNT_IntSet(PCNT_TypeDef *pcnt, uint32_t flags)
Set one or more pending PCNT interrupts from SW.

uint32_t PCNT_TopBufferGet(PCNT_TypeDef *pcnt)
Get the pulse counter top buffer value.

uint32_t PCNT_TopGet(PCNT_TypeDef *pcnt)
Get the pulse counter top value.

void PCNT_Sync(PCNT_TypeDef *pcnt, uint32_t mask)
Wait for an ongoing sync of register(s) to low-frequency domain to complete.

Macros

#define PCNT0_CNT_SIZE (16)
PCNT0 Counter register size.

#define PCNT1_CNT_SIZE (8)
PCNT1 Counter register size.

#define PCNT2_CNT_SIZE (8)
PCNT2 Counter register size.

#define PCNT_MODE_DISABLE 0xFF
PCNT mode disable.

#define PCNT_CNT_EVENT_NONE 0xFF
PCNT count event is none.

#define DEFAULT_DEBUG_HALT
Default Debug.

#define DEFAULT_MODE pcntModeDisable,
Default Mode.

#define DEFAULT_HYST false,
Default Hysteresis.

#define DEFAULT_CDIR true,
Default counter direction.

#define DEFAULT_CNTEV pcntCntEventUp,
Default count event.

#define DEFAULT_AUXCNTEV pcntCntEventNone,
Default auxiliary count event.

#define DEFAULT_PRS_CH pcntPRSCh0,
Default selected PRS channel as S0IN and S1IN.

#define PCNT_INIT_DEFAULT undefined
Default configuration for PCNT initialization structure.

Enumeration Documentation

PCNT_Mode_TypeDef

```
PCNT_Mode_TypeDef
```

Mode selection.

Enumerator	
pcntModeDisable	Disable pulse counter.
pcntModeOvsSingle	Single input LFACLK oversampling mode (available in EM0-EM2).
pcntModeExtSingle	Externally clocked single input counter mode (available in EM0-EM3).
pcntModeExtQuad	Externally clocked quadrature decoder mode (available in EM0-EM3).

PCNT_CntEvent_TypeDef

```
PCNT_CntEvent_TypeDef
```

Counter event selection.

Note: unshifted values are being used for enumeration because multiple configuration structure members use this type definition.

Enumerator

pcntCntEventBoth	Counts up on up-count and down on down-count events.
pcntCntEventUp	Only counts up on up-count events.
pcntCntEventDown	Only counts down on down-count events.
pcntCntEventNone	Never counts.

PCNT_PRSSel_TypeDef

> PCNT_PRSSel_TypeDef

PRS sources for `s0PRS` and `s1PRS`.

Enumerator	
pcntPRSCh0	PRS channel 0.
pcntPRSCh1	PRS channel 1.
pcntPRSCh2	PRS channel 2.
pcntPRSCh3	PRS channel 3.
pcntPRSCh4	PRS channel 4.
pcntPRSCh5	PRS channel 5.
pcntPRSCh6	PRS channel 6.
pcntPRSCh7	PRS channel 7.
pcntPRSCh8	PRS channel 8.
pcntPRSCh9	PRS channel 9.
pcntPRSCh10	PRS channel 10.
pcntPRSCh11	PRS channel 11.

PCNT_PRSInput_TypeDef

> PCNT_PRSInput_TypeDef

PRS inputs of PCNT.

Enumerator	
pcntPRSInputS0	
pcntPRSInputS1	PRS input 0.

Function Documentation

PCNT_CounterReset

> void PCNT_CounterReset (PCNT_TypeDef * pcnt)

Reset PCNT counters and TOP register.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.

Note

- Notice that special SYNCBUSY handling is not applicable for the RSTEN bit of the control register, so we don't need to wait for it when only modifying RSTEN. (It would mean undefined wait time if clocked by an external clock.) The SYNCBUSY bit will however be set, leading to a synchronization in the LF domain, with, in reality, no changes.

PCNT_Enable

```
void PCNT_Enable (PCNT_TypeDef * pcnt, PCNT_Mode_TypeDef mode)
```

Set PCNT operational mode.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
PCNT_Mode_TypeDef	[in]	mode	An operational mode to use for PCNT.

Notice that this function does not do any configuration. Setting operational mode is normally only required after initialization is done, and if not done as part of initialization or if requiring to disable/reenable pulse counter.

Note

- This function may stall until synchronization to low-frequency domain is completed. For that reason, it should normally not be used when an external clock is used for the PCNT module, since stall time may be undefined.

PCNT_IsEnabled

```
bool PCNT_IsEnabled (PCNT_TypeDef * pcnt)
```

Returns if the PCNT module is enabled or not.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.

Notice that this function does not do any configuration.

Returns

- Returns TRUE if the module is enabled.

PCNT_CounterTopSet

```
void PCNT_CounterTopSet (PCNT_TypeDef * pcnt, uint32_t count, uint32_t top)
```

Set the counter and top values.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
uint32_t	[in]	count	A value to set in the counter register.

Type	Direction	Argument Name	Description
uint32_t	[in]	top	A value to set in the top register.

The pulse counter is disabled while changing these values and reenabled (if originally enabled) when values have been set.

Note

- This function will stall until synchronization to low-frequency domain is completed. For that reason, it should normally not be used when an external clock is used for the PCNT module, since stall time may be undefined. The counter should normally only be set when operating in (or about to enable) pcntModeOvsSingle mode.

PCNT_PRSInputEnable

> void PCNT_PRSInputEnable (PCNT_TypeDef * pcnt, PCNT_PRSInput_TypeDef prsInput, bool enable)

Enable/disable the selected PRS input of PCNT.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
PCNT_PRSInput_TypeDef	[in]	prsInput	PRS input (S0 or S1) of the selected PCNT module.
bool	[in]	enable	Set to true to enable, false to disable the selected PRS input.

Notice that this function does not do any configuration.

PCNT_FreezeEnable

> void PCNT_FreezeEnable (PCNT_TypeDef * pcnt, bool enable)

PCNT register synchronization freeze control.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
bool	[in]	enable	• True - enable freeze, modified registers are not propagated to the LF domain. • False - disables freeze, modified registers are propagated to LF domain.

Some PCNT registers require synchronization into the low-frequency (LF) domain. The freeze feature allows for several registers to be modified before passing them to the LF domain simultaneously, which takes place when the freeze mode is disabled.

Note

- When enabling freeze mode, this function will wait for all current ongoing PCNT synchronization to the LF domain to complete (normally synchronization will not be in progress). However, for this reason, when using freeze mode, modifications of registers requiring the LF synchronization should be done within one freeze enable/disable block to avoid unnecessary stalling.

PCNT_Init

```
void PCNT_Init (PCNT_TypeDef * pcnt, const PCNT_Init_TypeDef * init)
```

Initialize the pulse counter.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
const PCNT_Init_TypeDef *	[in]	init	A pointer to the initialization structure.

This function will configure the pulse counter. The clock selection is configured as follows, depending on operational mode:

- pcntModeOvsSingle - Use LFACLK.
- pcntModeExtSingle - Use external PCNTn_S0 pin.
- pcntModeExtQuad - Use external PCNTn_S0 pin.

Notice that the LFACLK must be enabled in all modes, since some basic setup is done with this clock even if the external pin clock usage mode is chosen. The pulse counter clock for the selected instance must also be enabled prior to initialization.

Notice that pins used by the PCNT module must be properly configured by the user explicitly through setting the ROUTE register for the PCNT to work as intended.

Writing to CNT will not occur in external clock modes (EXTCLKQUAD and EXTCLKSINGLE) because the external clock rate is unknown. The user should handle it manually depending on the application.

TOPB is written for all modes but in external clock mode it will take 3 external clock cycles to sync to TOP.

Note

- Initializing requires synchronization into the low-frequency domain. This may cause a delay.

PCNT_Reset

```
void PCNT_Reset (PCNT_TypeDef * pcnt)
```

Reset PCNT to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.

Notice the LFACLK must be enabled, since some basic reset is done with this clock. The pulse counter clock for the selected instance must also be enabled prior to initialization.

Note

- The ROUTE register is NOT reset by this function to allow for centralized setup of this feature.

PCNT_TopBufferSet

```
void PCNT_TopBufferSet (PCNT_TypeDef * pcnt, uint32_t val)
```

Set top buffer value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
uint32_t	[in]	val	A value to set in the top buffer register.

Note

- This function may stall until synchronization to low-frequency domain is completed. For that reason, it should normally not be used when an external clock is used for the PCNT module since stall time may be undefined.

PCNT_TopSet

```
void PCNT_TopSet (PCNT_TypeDef * pcnt, uint32_t val)
```

Set the top value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
uint32_t	[in]	val	A value to set in the top register.

Note

- This function will stall until synchronization to low-frequency domain is completed. For that reason, it should normally not be used when an external clock is used for the PCNT module since stall time may be undefined.

PCNT_CounterGet

```
uint32_t PCNT_CounterGet (PCNT_TypeDef * pcnt)
```

Default configuration for PCNT initialization structure.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Get the pulse counter value.

Returns

- Current pulse counter value.

PCNT_AuxCounterGet

```
uint32_t PCNT_AuxCounterGet (PCNT_TypeDef * pcnt)
```

Get the auxiliary counter value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Returns

- Current auxiliary counter value.

PCNT_CounterSet

```
void PCNT_CounterSet (PCNT_TypeDef * pcnt, uint32_t count)
```

Set a counter value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.
uint32_t	[in]	count	Value to set in counter register.

Pulse counter is disabled while changing counter value and re-enabled (if originally enabled) when counter value has been set.

Note

- This function will stall until synchronization to low-frequency domain is completed. For that reason, it should normally not be used when using an external clock to clock the PCNT module since stall time may be undefined in that case. The counter should normally only be set when operating in (or about to enable) pcntModeOvsSingle mode.

PCNT_IntClear

```
void PCNT_IntClear (PCNT_TypeDef * pcnt, uint32_t flags)
```

Clear one or more pending PCNT interrupts.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.
uint32_t	[in]	flags	Pending PCNT interrupt source to clear. Use a bitwise logic OR combination of valid interrupt flags for the PCNT module (PCNT_IF_nnn).

PCNT_IntDisable

```
void PCNT_IntDisable (PCNT_TypeDef * pcnt, uint32_t flags)
```

Disable one or more PCNT interrupts.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	PCNT interrupt sources to disable. Use a bitwise logic OR combination of valid interrupt flags for PCNT module (PCNT_IF_nnn).

PCNT_IntEnable

```
void PCNT_IntEnable (PCNT_TypeDef * pcnt, uint32_t flags)
```

Enable one or more PCNT interrupts.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.
uint32_t	[in]	flags	PCNT interrupt sources to enable. Use a bitwise logic OR combination of valid interrupt flags for PCNT module (PCNT_IF_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using PCNT_IntClear() prior to enabling the interrupt.

PCNT_IntGet

```
uint32_t PCNT_IntGet (PCNT_TypeDef * pcnt)
```

Get pending PCNT interrupt flags.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Note

- The event bits are not cleared by the use of this function.

Returns

- PCNT interrupt sources pending. A bitwise logic OR combination of valid interrupt flags for PCNT module (PCNT_IF_nnn).

PCNT_IntGetEnabled

```
uint32_t PCNT_IntGetEnabled (PCNT_TypeDef * pcnt)
```

Get enabled and pending PCNT interrupt flags.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to thePCNT peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- The event bits are not cleared by the use of this function.

Returns

- Pending and enabled PCNT interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in PCNT_IEN_nnn register (PCNT_IEN_nnn) and
  - the OR combination of valid interrupt flags of the PCNT module (PCNT_IF_nnn).

PCNT_IntSet

```
void PCNT_IntSet (PCNT_TypeDef * pcnt, uint32_t flags)
```

Set one or more pending PCNT interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.
uint32_t	[in]	flags	PCNT interrupt sources to set to pending. Use a bitwise logic OR combination of valid interrupt flags for PCNT module (PCNT_IF_nnn).

PCNT_TopBufferGet

```
uint32_t PCNT_TopBufferGet (PCNT_TypeDef * pcnt)
```

Get the pulse counter top buffer value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Returns

- Current pulse counter top buffer value.

PCNT_TopGet

```
uint32_t PCNT_TopGet (PCNT_TypeDef * pcnt)
```

Get the pulse counter top value.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	Pointer to the PCNT peripheral register block.

Returns

- Current pulse counter top value.

PCNT_Sync

```
void PCNT_Sync (PCNT_TypeDef * pcnt, uint32_t mask)
```

Wait for an ongoing sync of register(s) to low-frequency domain to complete.

Parameters

Type	Direction	Argument Name	Description
PCNT_TypeDef *	[in]	pcnt	A pointer to the PCNT peripheral register block.
uint32_t	[in]	mask	A bitmask corresponding to SYNCBUSY register defined bits indicating registers that must complete any ongoing synchronization.

Macro Definition Documentation

PCNT0_CNT_SIZE

```
#define PCNT0_CNT_SIZE
```

Value:

```
(16)
```

PCNT0 Counter register size.

PCNT0 counter is 16 bits.

PCNT1_CNT_SIZE

```
#define PCNT1_CNT_SIZE
```

Value:

```
(8)
```

PCNT1 Counter register size.

PCNT1 counter is 8 bits.

PCNT2_CNT_SIZE

```
#define PCNT2_CNT_SIZE
```

Value:

```
(8)
```

PCNT2 Counter register size.

PCNT2 counter is 8 bits.

PCNT_MODE_DISABLE

```
#define PCNT_MODE_DISABLE
```

Value:

```
0xFF
```

PCNT mode disable.

PCNT_CNT_EVENT_NONE

```
#define PCNT_CNT_EVENT_NONE
```

Value:

```
0xFF
```

PCNT count event is none.

DEFAULT_DEBUG_HALT

```
#define DEFAULT_DEBUG_HALT
```

Default Debug.

DEFAULT_MODE

```
#define DEFAULT_MODE
```

Value:

```
pcntModeDisable,
```

Default Mode.

Disabled by default.

DEFAULT_HYST

```
#define DEFAULT_HYST
```

Value:

```
false,
```

Default Hysteresis.

Hysteresis disabled.

DEFAULT_CDIR

```
#define DEFAULT_CDIR
```

Value:

```
true,
```

Default counter direction.

Counter direction is given by CNTDIR.

DEFAULT_CNTEV

```
#define DEFAULT_CNTEV
```

Value:

```
pcntCntEventUp,
```

Default count event.

Regular counter counts up on upcount events.

DEFAULT_AUXCNTEV

```
#define DEFAULT_AUXCNTEV
```

Value:

```
pcntCntEventNone,
```

Default auxiliary count event.

Auxiliary counter doesn't respond to events.

DEFAULT_PRS_CH

```
#define DEFAULT_PRS_CH
```

Value:

```
pcntPRSCh0,
```

Default selected PRS channel as S0IN and S1IN.

PRS channel 0 selected as S0IN and as S1IN.

PCNT_INIT_DEFAULT

```
#define PCNT_INIT_DEFAULT
```

Value:

```
{                                                                   \
  DEFAULT_MODE                        /* Default mode. */           \
    _PCNT_CNT_RESETVALUE,             /* Default counter HW reset value. */ \
    _PCNT_TOP_RESETVALUE,             /* Default counter HW reset value. */ \
    false,                            /* Use positive edge. */      \
    false,                            /* Up-counting. */            \
    false,                            /* Filter disabled. */        \
  DEFAULT_DEBUG_HALT                  /* Debug Halt enabled. */     \
  DEFAULT_HYST                        /* Default Hysteresis. */     \
```

```
    DEFAULT_CDIR                        /* Default CNTDIR. */               \
    DEFAULT_CNTEV                       /* Faults CNTEV. */                 \
    DEFAULT_AUXCNTEV                    /* Default AUXCNTEV. */             \
    DEFAULT_PRS_CH                      /* PRS channel 0 selected as S0IN. */ \
      DEFAULT_PRS_CH                    /* PRS channel 0 selected as S1IN. */ \
}
```

Default configuration for PCNT initialization structure.

PCNT_Init_TypeDef

Initialization structure.

Public Attributes

PCNT_Mode_TypeDef	**mode** Mode to operate in.
uint32_t	**counter** Initial counter value (refer to reference manual for max value allowed).
uint32_t	**top** Initial top value (refer to reference manual for max value allowed).
bool	**negEdge** Polarity of incoming edge.
bool	**countDown** Counting direction, only applicable for pcntModeOvsSingle and pcntModeExtSingle modes.
bool	**filter** Enable filter, only available in pcntModeOvsSingle* mode.
bool	**hyst** Set to true to enable hysteresis.
bool	**s1CntDir** Set to true to enable S1 to determine the direction of counting in OVSSINGLE or EXTCLKSINGLE modes.
PCNT_CntEvent_TypeDef	**cntEvent** Selects whether the regular counter responds to up-count events, down-count events, both, or none.
PCNT_CntEvent_TypeDef	**auxCntEvent** Selects whether the auxiliary counter responds to up-count events, down-count events, both, or none.
PCNT_PRSSel_TypeDef	**s0PRS** Select PRS channel as input to S0IN in PCNTx_INPUT register.
PCNT_PRSSel_TypeDef	**s1PRS** Select PRS channel as input to S1IN in PCNTx_INPUT register.

Public Attribute Documentation

mode

```
PCNT_Mode_TypeDef PCNT_Init_TypeDef::mode
```

Mode to operate in.

counter

```
uint32_t PCNT_Init_TypeDef::counter
```

Initial counter value (refer to reference manual for max value allowed).

Only used for pcntModeOvsSingle (and possibly pcntModeDisable) modes. If using pcntModeExtSingle or pcntModeExtQuad modes, counter value is reset to HW reset value.

top

```
uint32_t PCNT_Init_TypeDef::top
```

Initial top value (refer to reference manual for max value allowed).

Only used for pcntModeOvsSingle (and possibly pcntModeDisable) modes. If using pcntModeExtSingle or pcntModeExtQuad modes, top value is reset to HW reset value.

negEdge

```
bool PCNT_Init_TypeDef::negEdge
```

Polarity of incoming edge.

- pcntModeExtSingle mode - if false, positive edges are counted, otherwise negative edges.
- pcntModeExtQuad mode - if true, counting direction is inverted.

countDown

```
bool PCNT_Init_TypeDef::countDown
```

Counting direction, only applicable for pcntModeOvsSingle and pcntModeExtSingle modes.

filter

```
bool PCNT_Init_TypeDef::filter
```

Enable filter, only available in pcntModeOvsSingle* mode.

hyst

```
bool PCNT_Init_TypeDef::hyst
```

Set to true to enable hysteresis.

When enabled, PCNT will always overflow and underflow to TOP/2.

s1CntDir

```
bool PCNT_Init_TypeDef::s1CntDir
```

Set to true to enable S1 to determine the direction of counting in OVSSINGLE or EXTCLKSINGLE modes.

When S1 is high, the count direction is given by CNTDIR, and when S1 is low, the count direction is the opposite.

cntEvent

```
PCNT_CntEvent_TypeDef PCNT_Init_TypeDef::cntEvent
```

Selects whether the regular counter responds to up-count events, down-count events, both, or none.

auxCntEvent

```
PCNT_CntEvent_TypeDef PCNT_Init_TypeDef::auxCntEvent
```

Selects whether the auxiliary counter responds to up-count events, down-count events, both, or none.

s0PRS

```
PCNT_PRSSel_TypeDef PCNT_Init_TypeDef::s0PRS
```

Select PRS channel as input to S0IN in PCNTx_INPUT register.

s1PRS

```
PCNT_PRSSel_TypeDef PCNT_Init_TypeDef::s1PRS
```

Select PRS channel as input to S1IN in PCNTx_INPUT register.

PRS - Peripheral Reflex System

PRS - Peripheral Reflex System

Peripheral Reflex System (PRS) Peripheral API.

This module contains functions to control the PRS peripheral of Silicon Labs 32-bit MCUs and SoCs. The PRS allows configurable, fast, and autonomous communication between peripherals on the MCU or SoC.

Enumerations

enum PRS_ChType_t {
  prsTypeAsync
  prsTypeSync
}
PRS Channel type.

enum PRS_Edge_TypeDef {
  prsEdgeOff
  prsEdgePos
  prsEdgeNeg
  prsEdgeBoth
}
Edge detection type.

enum PRS_Signal_t {

prsSignalNone = PRS_CH_CTRL_SOURCESEL_NONE | (0x0 << _PRS_CH_CTRL_SIGSEL_SHIFT)
prsSignalSW = PRS_CH_CTRL_SOURCESEL_NONE | (0x1 << _PRS_CH_CTRL_SIGSEL_SHIFT)
prsSignalADC0_SINGLE = PRS_ADC0_SINGLE
prsSignalADC0_SCAN = PRS_ADC0_SCAN
prsSignalTIMER0_UF = PRS_TIMER0_UF
prsSignalTIMER0_OF = PRS_TIMER0_OF
prsSignalTIMER0_CC0 = PRS_TIMER0_CC0
prsSignalTIMER0_CC1 = PRS_TIMER0_CC1
prsSignalTIMER0_CC2 = PRS_TIMER0_CC2
prsSignalTIMER1_UF = PRS_TIMER1_UF
prsSignalTIMER1_OF = PRS_TIMER1_OF
prsSignalTIMER1_CC0 = PRS_TIMER1_CC0
prsSignalTIMER1_CC1 = PRS_TIMER1_CC1
prsSignalTIMER1_CC2 = PRS_TIMER1_CC2
prsSignalTIMER2_UF = PRS_TIMER2_UF
prsSignalTIMER2_OF = PRS_TIMER2_OF
prsSignalTIMER2_CC0 = PRS_TIMER2_CC0
prsSignalTIMER2_CC1 = PRS_TIMER2_CC1
prsSignalTIMER2_CC2 = PRS_TIMER2_CC2
prsSignalTIMER3_UF = PRS_TIMER3_UF
prsSignalTIMER3_OF = PRS_TIMER3_OF
prsSignalTIMER3_CC0 = PRS_TIMER3_CC0
prsSignalTIMER3_CC1 = PRS_TIMER3_CC1
prsSignalTIMER3_CC2 = PRS_TIMER3_CC2
prsSignalLETIMER0_CH0 = PRS_LETIMER0_CH0
prsSignalLETIMER0_CH1 = PRS_LETIMER0_CH1
prsSignalRTC_OF = PRS_RTC_OF
prsSignalRTC_COMP0 = PRS_RTC_COMP0
prsSignalRTC_COMP1 = PRS_RTC_COMP1
prsSignalBURTC_COMP = PRS_BURTC_COMP
prsSignalBURTC_OF = PRS_BURTC_OF
prsSignalACMP0_OUT = PRS_ACMP0_OUT
prsSignalACMP1_OUT = PRS_ACMP1_OUT
prsSignalLESENSE_SCANRES0 = PRS_LESENSE_SCANRES0
prsSignalLESENSE_SCANRES1 = PRS_LESENSE_SCANRES1
prsSignalLESENSE_SCANRES2 = PRS_LESENSE_SCANRES2
prsSignalLESENSE_SCANRES3 = PRS_LESENSE_SCANRES3
prsSignalLESENSE_SCANRES4 = PRS_LESENSE_SCANRES4
prsSignalLESENSE_SCANRES5 = PRS_LESENSE_SCANRES5
prsSignalLESENSE_SCANRES6 = PRS_LESENSE_SCANRES6
prsSignalLESENSE_SCANRES7 = PRS_LESENSE_SCANRES7
prsSignalLESENSE_SCANRES8 = PRS_LESENSE_SCANRES8
prsSignalLESENSE_SCANRES9 = PRS_LESENSE_SCANRES9
prsSignalLESENSE_SCANRES10 = PRS_LESENSE_SCANRES10
prsSignalLESENSE_SCANRES11 = PRS_LESENSE_SCANRES11
prsSignalLESENSE_SCANRES12 = PRS_LESENSE_SCANRES12
prsSignalLESENSE_SCANRES13 = PRS_LESENSE_SCANRES13
prsSignalLESENSE_SCANRES14 = PRS_LESENSE_SCANRES14
prsSignalLESENSE_SCANRES15 = PRS_LESENSE_SCANRES15
prsSignalLESENSE_DEC0 = PRS_LESENSE_DEC0
prsSignalLESENSE_DEC1 = PRS_LESENSE_DEC1
prsSignalLESENSE_DEC2 = PRS_LESENSE_DEC2
prsSignalUSART1_TXC = PRS_USART1_TXC
prsSignalUSART1_RXDATAV = PRS_USART1_RXDATAV
prsSignalUSART2_TXC = PRS_USART2_TXC
prsSignalUSART2_RXDATAV = PRS_USART2_RXDATAV
prsSignalUART0_TXC = PRS_UART0_TXC
prsSignalUART0_RXDATAV = PRS_UART0_RXDATAV
prsSignalUART1_TXC = PRS_UART1_TXC
prsSignalUART1_RXDATAV = PRS_UART1_RXDATAV
prsSignalGPIO_PIN0 = PRS_GPIO_PIN0
prsSignalGPIO_PIN1 = PRS_GPIO_PIN1
prsSignalGPIO_PIN2 = PRS_GPIO_PIN2
prsSignalGPIO_PIN3 = PRS_GPIO_PIN3
prsSignalGPIO_PIN4 = PRS_GPIO_PIN4
prsSignalGPIO_PIN5 = PRS_GPIO_PIN5
prsSignalGPIO_PIN6 = PRS_GPIO_PIN6
prsSignalGPIO_PIN7 = PRS_GPIO_PIN7
prsSignalGPIO_PIN8 = PRS_GPIO_PIN8
prsSignalGPIO_PIN9 = PRS_GPIO_PIN9
prsSignalGPIO_PIN10 = PRS_GPIO_PIN10
prsSignalGPIO_PIN11 = PRS_GPIO_PIN11
prsSignalGPIO_PIN12 = PRS_GPIO_PIN12
prsSignalGPIO_PIN13 = PRS_GPIO_PIN13
prsSignalGPIO_PIN14 = PRS_GPIO_PIN14
prsSignalGPIO_PIN15 = PRS_GPIO_PIN15

}

PRS Signal.

Functions

void PRS_SourceSignalSet(unsigned int ch, uint32_t source, uint32_t signal, PRS_Edge_TypeDef edge)
Set a source and signal for a channel.

void PRS_SourceAsyncSignalSet(unsigned int ch, uint32_t source, uint32_t signal)
Set the source and asynchronous signal for a channel.

void PRS_GpioOutputLocation(unsigned int ch, unsigned int location)
Send the output of a PRS channel to a GPIO pin.

int PRS_GetFreeChannel(PRS_ChType_t type)
Search for the first free PRS channel.

void PRS_Reset(void)
Reset all PRS channels.

void PRS_ConnectSignal(unsigned int ch, PRS_ChType_t type, PRS_Signal_t signal)
Connect a PRS signal to a channel.

void PRS_LevelSet(uint32_t level, uint32_t mask)
Set level control bit for one or more channels.

uint32_t PRS_LevelGet(void)
Get level control bit for all channels.

void PRS_PulseTrigger(uint32_t channels)
Trigger a high pulse (one HFPERCLK) for one or more channels.

void PRS_ChannelLevelSet(unsigned int ch, bool level)
Set the PRS channel level for one asynchronous PRS channel.

void PRS_ChannelPulse(unsigned int ch)
Trigger a pulse on one PRS channel.

Macros

#define PRS_SYNC_CHAN_COUNT PRS_CHAN_COUNT
PRS Synchronous channel count.

#define PRS_ASYNC_CHAN_COUNT PRS_CHAN_COUNT
PRS Asynchronous channel count.

#define PRS_ASYNC_SUPPORTED 1
PRS asynchronous support.

Enumeration Documentation

PRS_ChType_t

> PRS_ChType_t

PRS Channel type.

Enumerator	
prsTypeAsync	Asynchronous channel type.
prsTypeSync	Synchronous channel type.

PRS_Edge_TypeDef

> PRS_Edge_TypeDef

Edge detection type.

Enumerator	
prsEdgeOff	Leave signal as is.
prsEdgePos	Generate pulses on positive edge.
prsEdgeNeg	Generate pulses on negative edge.
prsEdgeBoth	Generate pulses on both edges.

PRS_Signal_t

> PRS_Signal_t

PRS Signal.

Enumerator	
prsSignalNone	No Signal.
prsSignalSW	Software-reserved Signal.
prsSignalADC0_SINGLE	ADC0_SINGLE signal.
prsSignalADC0_SCAN	ADC0_SCAN signal.
prsSignalTIMER0_UF	TIMER0 underflow Signal.
prsSignalTIMER0_OF	TIMER0 overflow Signal.
prsSignalTIMER0_CC0	TIMER0 capture/compare channel 0 Signal.
prsSignalTIMER0_CC1	TIMER0 capture/compare channel 1 Signal.
prsSignalTIMER0_CC2	TIMER0 capture/compare channel 2 Signal.
prsSignalTIMER1_UF	TIMER1 underflow Signal.
prsSignalTIMER1_OF	TIMER1 overflow Signal.
prsSignalTIMER1_CC0	TIMER1 capture/compare channel 0 Signal.
prsSignalTIMER1_CC1	TIMER1 capture/compare channel 1 Signal.
prsSignalTIMER1_CC2	TIMER1 capture/compare channel 2 Signal.
prsSignalTIMER2_UF	TIMER2 underflow Signal.
prsSignalTIMER2_OF	TIMER2 overflow Signal.
prsSignalTIMER2_CC0	TIMER2 capture/compare channel 0 Signal.
prsSignalTIMER2_CC1	TIMER2 capture/compare channel 1 Signal.
prsSignalTIMER2_CC2	TIMER2 capture/compare channel 2 Signal.
prsSignalTIMER3_UF	TIMER3 underflow Signal.
prsSignalTIMER3_OF	TIMER3 overflow Signal.
prsSignalTIMER3_CC0	TIMER3 capture/compare channel 0 Signal.
prsSignalTIMER3_CC1	TIMER3 capture/compare channel 1 Signal.
prsSignalTIMER3_CC2	TIMER3 capture/compare channel 2 Signal.
prsSignalLETIMER0_CH0	LETIMER0 channel 0 Signal.
prsSignalLETIMER0_CH1	LETIMER0 channel 1 Signal.
prsSignalRTC_OF	RTC_OF signal.
prsSignalRTC_COMP0	RTC_COMP0 signal.
prsSignalRTC_COMP1	RTC_COMP1 signal.
prsSignalBURTC_COMP	BURTC compare Signal.
prsSignalBURTC_OF	BURTC overflow Signal.

prsSignalACMP0_OUT	ACMP0 Signal.
prsSignalACMP1_OUT	ACMP1 output Signal.
prsSignalLESENSE_SCANRES0	LESENSE_SCANRES0 Signal.
prsSignalLESENSE_SCANRES1	LESENSE_SCANRES1 Signal.
prsSignalLESENSE_SCANRES2	LESENSE_SCANRES2 Signal.
prsSignalLESENSE_SCANRES3	LESENSE_SCANRES3 Signal.
prsSignalLESENSE_SCANRES4	LESENSE_SCANRES4 Signal.
prsSignalLESENSE_SCANRES5	LESENSE_SCANRES5 Signal.
prsSignalLESENSE_SCANRES6	LESENSE_SCANRES6 Signal.
prsSignalLESENSE_SCANRES7	LESENSE_SCANRES7 Signal.
prsSignalLESENSE_SCANRES8	LESENSE_SCANRES8 Signal.
prsSignalLESENSE_SCANRES9	LESENSE_SCANRES9 Signal.
prsSignalLESENSE_SCANRES10	LESENSE_SCANRES10 Signal.
prsSignalLESENSE_SCANRES11	LESENSE_SCANRES11 Signal.
prsSignalLESENSE_SCANRES12	LESENSE_SCANRES12 Signal.
prsSignalLESENSE_SCANRES13	LESENSE_SCANRES13 Signal.
prsSignalLESENSE_SCANRES14	LESENSE_SCANRES14 Signal.
prsSignalLESENSE_SCANRES15	LESENSE_SCANRES15 Signal.
prsSignalLESENSE_DEC0	LESENSE_DEC0 Signal.
prsSignalLESENSE_DEC1	LESENSE_DEC1 Signal.
prsSignalLESENSE_DEC2	LESENSE_DEC2 Signal.
prsSignalUSART1_TXC	USART1 TX complete Signal.
prsSignalUSART1_RXDATAV	USART1 RX data available Signal.
prsSignalUSART2_TXC	USART2 TX complete Signal.
prsSignalUSART2_RXDATAV	USART2 RX data available Signal.
prsSignalUART0_TXC	UART0 TX complete Signal.
prsSignalUART0_RXDATAV	UART0 RX data available Signal.
prsSignalUART1_TXC	UART1 TX complete Signal.
prsSignalUART1_RXDATAV	UART1 RX data available Signal.
prsSignalGPIO_PIN0	GPIO Pin 0 Signal.
prsSignalGPIO_PIN1	GPIO Pin 1 Signal.
prsSignalGPIO_PIN2	GPIO Pin 2 Signal.
prsSignalGPIO_PIN3	GPIO Pin 3 Signal.
prsSignalGPIO_PIN4	GPIO Pin 4 Signal.
prsSignalGPIO_PIN5	GPIO Pin 5 Signal.
prsSignalGPIO_PIN6	GPIO Pin 6 Signal.
prsSignalGPIO_PIN7	GPIO Pin 7 Signal.
prsSignalGPIO_PIN8	GPIO Pin 8 Signal.
prsSignalGPIO_PIN9	GPIO Pin 9 Signal.
prsSignalGPIO_PIN10	GPIO Pin 10 Signal.
prsSignalGPIO_PIN11	GPIO Pin 11 Signal.
prsSignalGPIO_PIN12	GPIO Pin 12 Signal.
prsSignalGPIO_PIN13	GPIO Pin 13 Signal.

prsSignalGPIO_PIN14	GPIO Pin 14 Signal.
prsSignalGPIO_PIN15	GPIO Pin 15 Signal.

Function Documentation

PRS_SourceSignalSet

> void PRS_SourceSignalSet (unsigned int ch, uint32_t source, uint32_t signal, PRS_Edge_TypeDef edge)

Set a source and signal for a channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	A channel to define the signal and source for.
uint32_t	[in]	source	A source to select for the channel. Use one of PRS_CH_CTRL_SOURCESEL_x defines.
uint32_t	[in]	signal	A signal (for selected `source`) to use. Use one of PRS_CH_CTRL_SIGSEL_x defines.
PRS_Edge_TypeDef	[in]	edge	An edge (for selected source/signal) to generate the pulse for.

PRS_SourceAsyncSignalSet

> void PRS_SourceAsyncSignalSet (unsigned int ch, uint32_t source, uint32_t signal)

Set the source and asynchronous signal for a channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	A channel to define the source and asynchronous signal for.
uint32_t	[in]	source	A source to select for the channel. Use one of PRS_CH_CTRL_SOURCESEL_x defines.
uint32_t	[in]	signal	An asynchronous signal (for selected `source`) to use. Use one of the PRS_CH_CTRL_SIGSEL_x defines that support asynchronous operation.

Asynchronous reflexes are not clocked on HFPERCLK and can be used even in EM2/EM3. There is a limitation to reflexes operating in asynchronous mode in that they can only be used by a subset of the reflex consumers. See the PRS chapter in the reference manual for the complete list of supported asynchronous signals and consumers.

Note

- This function is not supported on EFM32GxxxFyyy parts. In asynchronous mode, the edge detector only works in EM0 and should not be used. The EDSEL parameter in PRS_CHx_CTRL register is set to 0 (OFF) by default.

PRS_GpioOutputLocation

> void PRS_GpioOutputLocation (unsigned int ch, unsigned int location)

Send the output of a PRS channel to a GPIO pin.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	PRS channel number.
unsigned int	[in]	location	PRS routing location.

This function is used to send the output of a PRS channel to a GPIO pin. Note that there are certain restrictions to where a PRS channel can be routed. Consult the datasheet of the device to see if a channel can be routed to the requested GPIO pin.

PRS_GetFreeChannel

> int PRS_GetFreeChannel (PRS_ChType_t type)

Search for the first free PRS channel.

Parameters

Type	Direction	Argument Name	Description
PRS_ChType_t	[in]	type	PRS channel type. This can be either prsTypeAsync or prsTypeSync.

Returns

- Channel number >= 0 if an unused PRS channel was found. If no free PRS channel was found then -1 is returned.

PRS_Reset

> void PRS_Reset (void)

Reset all PRS channels.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function will reset all the PRS channel configuration.

PRS_ConnectSignal

> void PRS_ConnectSignal (unsigned int ch, PRS_ChType_t type, PRS_Signal_t signal)

Connect a PRS signal to a channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	PRS channel number.
PRS_ChType_t	[in]	type	PRS channel type. This can be either prsTypeAsync or prsTypeSync.

Type	Direction	Argument Name	Description
PRS_Signal_t	[in]	signal	This is the PRS signal that should be placed on the channel.

This function will make the PRS signal available on the specific channel. Only a single PRS signal can be connected to any given channel.

PRS_LevelSet

> void PRS_LevelSet (uint32_t level, uint32_t mask)

Set level control bit for one or more channels.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	level	Level to use for channels indicated by `mask`. Use logical OR combination of PRS_SWLEVEL_CHnLEVEL defines for channels to set high level, otherwise 0.
uint32_t	[in]	mask	Mask indicating which channels to set level for. Use logical OR combination of PRS_SWLEVEL_CHnLEVEL defines.

The level value for a channel is XORed with both the pulse possibly issued by PRS_PulseTrigger() and the PRS input signal selected for the channel(s).

PRS_LevelGet

> uint32_t PRS_LevelGet (void)

Get level control bit for all channels.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The current software level configuration.

PRS_PulseTrigger

> void PRS_PulseTrigger (uint32_t channels)

Trigger a high pulse (one HFPERCLK) for one or more channels.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	channels	Logical ORed combination of channels to trigger a pulse for. Use PRS_SWPULSE_CHnPULSE defines.

Setting a bit for a channel causes the bit in the register to remain high for one HFPERCLK cycle. Pulse is XORed with both the corresponding bit in PRS SWLEVEL register and the PRS input signal selected for the channel(s).

PRS_ChannelLevelSet

```
void PRS_ChannelLevelSet (unsigned int ch, bool level)
```

Set the PRS channel level for one asynchronous PRS channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	PRS channel number.
bool	[in]	level	true to set the level high (1) and false to set the level low (0).

PRS_ChannelPulse

```
void PRS_ChannelPulse (unsigned int ch)
```

Trigger a pulse on one PRS channel.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	ch	PRS channel number.

Macro Definition Documentation

PRS_SYNC_CHAN_COUNT

```
#define PRS_SYNC_CHAN_COUNT
```

Value:

```
PRS_CHAN_COUNT
```

PRS Synchronous channel count.

PRS_ASYNC_CHAN_COUNT

```
#define PRS_ASYNC_CHAN_COUNT
```

Value:

```
PRS_CHAN_COUNT
```

PRS Asynchronous channel count.

PRS_ASYNC_SUPPORTED

```
#define PRS_ASYNC_SUPPORTED
```

Value:

```
1
```

PRS asynchronous support.

RAMFUNC - RAM Function Support

RAMFUNC - RAM Function Support

RAM code support.

Provides support for executing code from RAM. Provides a unified method to manage RAM code across all supported tools.

Note

- Other cross-compiler support macros are implemented in COMMON.
- Functions executing from RAM should not be declared as static.

Warnings

- Standard library facilities are available to the tool with GCC in hosted mode (default), regardless of the section attribute. Calls to standard libraries placed in the default section may therefore occur. To disable hosted mode, add '-ffreestanding' to the build command line. This is the only way to guarantee no calls to standard libraries with GCC. Read more at www.gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/Standards.html
- Keil/ARM uVision users must add a section named "ram_code" in their linker scatter file. This section must be in RAM memory. Look in the MCU SDK for example scatter files (ram_code.sct).

Usage

In your .h file:

```
#include "em_ramfunc.h"

SL_RAMFUNC_DECLARATOR
void MyPrint(const char* string);
```

Issues have been observed with ARM GCC when there is no declarator. It is recommended to have a declarator also for internal functions but move the declarator to the .c file.

In your .c file:

```
#include "em_ramfunc.h"

SL_RAMFUNC_DEFINITION_BEGIN
void MyPrint(const char* string)
{
...
}
SL_RAMFUNC_DEFINITION_END
```

Macros

#define SL_RAMFUNC_DISABLE
This define is not present by default.

#define SL_RAMFUNC_DECLARATOR
Compiler ported function declarator for RAM code.

#define SL_RAMFUNC_DEFINITION_BEGIN
Compiler ported function definition begin marker for RAM code.

#define SL_RAMFUNC_DEFINITION_END
Compiler ported function definition end marker for RAM code.

Macro Definition Documentation

SL_RAMFUNC_DISABLE

```
#define SL_RAMFUNC_DISABLE
```

This define is not present by default.

By compiling with define SL_RAMFUNC_DISABLE, code placed in RAM by SL_RAMFUNC macros will be placed in default code space (Flash) instead.

Note

- This define is not present by default.

SL_RAMFUNC_DECLARATOR

```
#define SL_RAMFUNC_DECLARATOR
```

Compiler ported function declarator for RAM code.

SL_RAMFUNC_DEFINITION_BEGIN

```
#define SL_RAMFUNC_DEFINITION_BEGIN
```

Compiler ported function definition begin marker for RAM code.

SL_RAMFUNC_DEFINITION_END

```
#define SL_RAMFUNC_DEFINITION_END
```

Compiler ported function definition end marker for RAM code.

RMU - Reset Management Unit

RMU - Reset Management Unit

Reset Management Unit (RMU) Peripheral API.

This module contains functions to control the RMU peripheral of Silicon Labs 32-bit MCUs and SoCs. RMU ensures correct reset operation and is responsible for connecting the different reset sources to the reset lines of the MCU or SoC.

Enumerations

enum RMU_ResetMode_TypeDef {
  rmuResetModeClear = 0
  rmuResetModeSet = 1
}
RMU reset modes.

enum RMU_Reset_TypeDef {
  rmuResetBU = _RMU_CTRL_BURSTEN_MASK
  rmuResetLockUp = _RMU_CTRL_LOCKUPRDIS_MASK
}
RMU controlled peripheral reset control and reset source control.

Functions

void RMU_ResetControl(RMU_Reset_TypeDef reset, RMU_ResetMode_TypeDef mode)
Disable/enable reset for various peripherals and signal sources.

void RMU_ResetCauseClear(void)
Clear the reset cause register.

uint32_t RMU_ResetCauseGet(void)
Get the cause of the last reset.

Macros

#define RMU_LockupResetDisable (A)
RMU_LockupResetDisable kept for backwards compatibility.

Enumeration Documentation

RMU_ResetMode_TypeDef

```
RMU_ResetMode_TypeDef
```

RMU reset modes.

Enumerator	
rmuResetModeClear	Reset mode clear.
rmuResetModeSet	Reset mode set.

RMU_Reset_TypeDef

RMU_Reset_TypeDef

RMU controlled peripheral reset control and reset source control.

Enumerator	
rmuResetBU	Reset control over Backup Power domain select.
rmuResetLockUp	Cortex lockup reset select.

Function Documentation

RMU_ResetControl

void RMU_ResetControl (RMU_Reset_TypeDef reset, RMU_ResetMode_TypeDef mode)

Disable/enable reset for various peripherals and signal sources.

Parameters

Type	Direction	Argument Name	Description
RMU_Reset_TypeDef	[in]	reset	Reset types to enable/disable.s
RMU_ResetMode_TypeDef	[in]	mode	Reset mode.

RMU_ResetCauseClear

void RMU_ResetCauseClear (void)

Clear the reset cause register.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

This function clears all the reset cause bits of the RSTCAUSE register. The reset cause bits must be cleared by software before a new reset occurs. Otherwise, reset causes may accumulate. See RMU_ResetCauseGet().

RMU_ResetCauseGet

uint32_t RMU_ResetCauseGet (void)

Get the cause of the last reset.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

To be useful, the reset cause must be cleared by software before a new reset occurs. Otherwise, reset causes may accumulate. See RMU_ResetCauseClear(). This function call will return the main cause for reset, which can be a bit mask (several causes) and clear away "noise".

Returns

- A reset cause mask. See the reference manual for a description of the reset cause mask.

Macro Definition Documentation

RMU_LockupResetDisable

```
#define RMU_LockupResetDisable
```

Value:

```
(A)
```

RMU_LockupResetDisable kept for backwards compatibility.

RTC - Real Time Counter

RTC - Real Time Counter

Real Time Counter (RTC) Peripheral API.

This module contains functions to control the RTC peripheral of Silicon Labs 32-bit MCUs and SoCs. The RTC ensures timekeeping in low energy modes.

Modules

RTC_Init_TypeDef

Functions

uint32_t RTC_CompareGet(unsigned int comp)
Get the RTC compare register value.

void RTC_CompareSet(unsigned int comp, uint32_t value)
Set the RTC compare register value.

uint32_t RTC_CounterGet(void)
Get RTC counter value.

void RTC_CounterSet(uint32_t value)
Set the RTC counter value.

void RTC_CounterReset(void)
Restart the RTC counter from zero.

void RTC_Enable(bool enable)
Enable/disable RTC.

void RTC_FreezeEnable(bool enable)
RTC register synchronization freeze control.

void RTC_Init(const RTC_Init_TypeDef *init)
Initialize RTC.

void RTC_IntClear(uint32_t flags)
Clear one or more pending RTC interrupts.

void RTC_IntDisable(uint32_t flags)
Disable one or more RTC interrupts.

void RTC_IntEnable(uint32_t flags)
Enable one or more RTC interrupts.

uint32_t RTC_IntGet(void)
Get pending RTC interrupt flags.

uint32_t RTC_IntGetEnabled(void)
Get enabled and pending RTC interrupt flags.

void RTC_IntSet(uint32_t flags)
Set one or more pending RTC interrupts from SW.

void RTC_Reset(void)
Restore RTC to reset state.

Macros

#define NUM_RTC_CHANNELS 2U
The RTC peripheral on series 0 devices support 2 compare channels while the RTC peripheral on series 1 devices support 6 compare channels.

#define RTC_INIT_DEFAULT undefined
Suggested default configuration for RTC initialization structure.

Function Documentation

RTC_CompareGet

```
uint32_t RTC_CompareGet (unsigned int comp)
```

Get the RTC compare register value.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	comp	A compare register to get. This value must be less than NUM_RTC_CHANNELS.

Returns

- A compare register value, 0 if invalid register selected.

RTC_CompareSet

```
void RTC_CompareSet (unsigned int comp, uint32_t value)
```

Set the RTC compare register value.

Parameters

Type	Direction	Argument Name	Description
unsigned int	[in]	comp	A compare register to set. This value must be less than NUM_RTC_CHANNELS.
uint32_t	[in]	value	An initialization value (<= 0x00ffffff).

Note

- The setting of a compare register requires synchronization into the low-frequency domain. If the same register is modified before a previous update has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the regSync() internal function call.

RTC_CounterGet

```
uint32_t RTC_CounterGet (void )
```

Get RTC counter value.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Current RTC counter value.

RTC_CounterSet

```
void RTC_CounterSet (uint32_t value)
```

Set the RTC counter value.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	value	The new RTC counter value.

RTC_CounterReset

```
void RTC_CounterReset (void )
```

Restart the RTC counter from zero.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

RTC_Enable

```
void RTC_Enable (bool enable)
```

Enable/disable RTC.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	True to enable counting, false to disable.

Note

- The enabling/disabling of RTC modifies the RTC CTRL register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the regSync() internal function call.

RTC_FreezeEnable

```
void RTC_FreezeEnable (bool enable)
```

RTC register synchronization freeze control.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	• True - enable freeze, modified registers are not propagated to the LF domain • False - disables freeze, modified registers are propagated to LF domain

Some RTC registers require synchronization into the low-frequency (LF) domain. The freeze feature allows for several registers to be modified before passing them to the LF domain simultaneously (which takes place when the freeze mode is disabled).

Note

- When enabling freeze mode, this function will wait for all current ongoing RTC synchronization to LF domain to complete (normally synchronization will not be in progress.) However, for this reason, when using freeze mode, modifications of registers requiring LF synchronization should be done within one freeze enable/disable block to avoid unnecessary stalling. This only applies to the Gecko Family. See the reference manual chapter about Access to Low Energy Peripherals (Asynchronos Registers) for details.

RTC_Init

```
void RTC_Init (const RTC_Init_TypeDef * init)
```

Initialize RTC.

Parameters

Type	Direction	Argument Name	Description
const RTC_Init_TypeDef *	[in]	init	A pointer to the RTC initialization structure.

Note that the compare values must be set separately with RTC_CompareSet() prior to the use of this function if configuring the RTC to start when initialization is completed.

Note

- The initialization of the RTC modifies the RTC CTRL register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed. This only applies to the Gecko Family. See comments in the regSync() internal function call.

RTC_IntClear

```
void RTC_IntClear (uint32_t flags)
```

Clear one or more pending RTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	RTC interrupt sources to clear. Use a set of interrupt flags OR-ed together to clear multiple interrupt sources for the RTC module (RTC_IFS_nnn).

RTC_IntDisable

> void RTC_IntDisable (uint32_t flags)

Disable one or more RTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	RTC interrupt sources to disable. Use a set of interrupt flags OR-ed together to disable multiple interrupt sources for the RTC module (RTC_IFS_nnn).

RTC_IntEnable

> void RTC_IntEnable (uint32_t flags)

Enable one or more RTC interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	RTC interrupt sources to enable. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the RTC module (RTC_IFS_nnn).

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using RTC_IntClear() prior to enabling the interrupt.

RTC_IntGet

> uint32_t RTC_IntGet (void)

Get pending RTC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- Event bits are not cleared by using this function.

Returns

- Pending RTC interrupt sources. Returns a set of interrupt flags OR-ed together for multiple interrupt sources in the RTC module (RTC_IFS_nnn).

RTC_IntGetEnabled

> uint32_t RTC_IntGetEnabled (void)

Get enabled and pending RTC interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by using this function.

Returns

- Pending and enabled RTC interrupt sources. The return value is the bitwise AND of
  - the enabled interrupt sources in RTC_IEN and
  - the pending interrupt flags RTC_IF.

RTC_IntSet

```
void RTC_IntSet (uint32_t flags)
```

Set one or more pending RTC interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	RTC interrupt sources to set to pending. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the RTC module (RTC_IFS_nnn).

RTC_Reset

```
void RTC_Reset (void )
```

Restore RTC to reset state.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Macro Definition Documentation

NUM_RTC_CHANNELS

```
#define NUM_RTC_CHANNELS
```

Value:

```
2U
```

The RTC peripheral on series 0 devices support 2 compare channels while the RTC peripheral on series 1 devices support 6 compare channels.

RTC_INIT_DEFAULT

```
#define RTC_INIT_DEFAULT
```

Value:

```
{                                                 \
  true,  /* Start counting when initialization is done. */  \
  false, /* Disable updating during debug halt. */          \
  true   /* Restart counting from 0 when reaching COMP0. */ \
}
```

Suggested default configuration for RTC initialization structure.

RTC_Init_TypeDef

RTC initialization structure.

Public Attributes

bool	enable Start counting when initialization is completed.
bool	debugRun Counter shall keep running during debug halt.
bool	comp0Top Use compare register 0 as max count value.

Public Attribute Documentation

enable

```
bool RTC_Init_TypeDef::enable
```

Start counting when initialization is completed.

debugRun

```
bool RTC_Init_TypeDef::debugRun
```

Counter shall keep running during debug halt.

comp0Top

```
bool RTC_Init_TypeDef::comp0Top
```

Use compare register 0 as max count value.

SYSTEM - System Utils

SYSTEM - System Utils

System API.

This module contains functions to read information such as RAM and Flash size, device unique ID, chip revision, family, and part number from DEVINFO and SCB blocks. Functions to configure and read status from FPU are available for compatible devices.

Modules

SYSTEM_ChipRevision_TypeDef

SYSTEM_CalAddrVal_TypeDef

Enumerations

enum SYSTEM_PartFamily_TypeDef {

```
systemPartFamilyEfm32Gecko = _DEVINFO_PART_DEVICE_FAMILY_EFM32G
systemPartFamilyEfm32Giant = _DEVINFO_PART_DEVICE_FAMILY_EFM32GG
systemPartFamilyEfm32Tiny = _DEVINFO_PART_DEVICE_FAMILY_EFM32TG
systemPartFamilyEfm32Leopard = _DEVINFO_PART_DEVICE_FAMILY_EFM32LG
systemPartFamilyEfm32Wonder = _DEVINFO_PART_DEVICE_FAMILY_EFM32WG
systemPartFamilyEfm32Zero = _DEVINFO_PART_DEVICE_FAMILY_EFM32ZG
systemPartFamilyEfm32Happy = _DEVINFO_PART_DEVICE_FAMILY_EFM32HG
systemPartFamilyEzr32Wonder = _DEVINFO_PART_DEVICE_FAMILY_EZR32WG
systemPartFamilyEzr32Leopard = _DEVINFO_PART_DEVICE_FAMILY_EZR32LG
systemPartFamilyEzr32Happy = _DEVINFO_PART_DEVICE_FAMILY_EZR32HG
systemPartFamilyGecko = _DEVINFO_PART_DEVICE_FAMILY_G
systemPartFamilyGiant = _DEVINFO_PART_DEVICE_FAMILY_GG
systemPartFamilyTiny = _DEVINFO_PART_DEVICE_FAMILY_TG
systemPartFamilyLeopard = _DEVINFO_PART_DEVICE_FAMILY_LG
systemPartFamilyWonder = _DEVINFO_PART_DEVICE_FAMILY_WG
systemPartFamilyZero = _DEVINFO_PART_DEVICE_FAMILY_ZG
systemPartFamilyHappy = _DEVINFO_PART_DEVICE_FAMILY_HG
systemPartFamilyUnknown = 0xFF
```

}

Family identifiers.

enum SYSTEM_FpuAccess_TypeDef {

```
fpuAccessDenied = (0x0 << 20)
fpuAccessPrivilegedOnly = (0x5 << 20)
fpuAccessReserved = (0xA << 20)
fpuAccessFull = (0xF << 20)
```

}

Floating point co-processor access modes.

enum SYSTEM_SecurityCapability_TypeDef {

```
securityCapabilityUnknown
securityCapabilityNA
securityCapabilityBasic
securityCapabilityRoT
securityCapabilitySE
securityCapabilityVault
```

}

Family security capability.

Functions

void	SYSTEM_ChipRevisionGet(SYSTEM_ChipRevision_TypeDef *rev) Get a chip major/minor revision.
SYSTEM_PartFamily_TypeDef	SYSTEM_GetFamily(void) Get the MCU family identifier.
void	SYSTEM_FpuAccessModeSet(SYSTEM_FpuAccess_TypeDef accessMode) Set floating point co-processor (FPU) access mode.
bool	SYSTEM_GetCalibrationValue(volatile uint32_t *regAddress) Get a factory calibration value for a given peripheral register.
SYSTEM_SecurityCapability_TypeDef	SYSTEM_GetSecurityCapability(void) Get family security capability.
uint64_t	SYSTEM_GetUnique(void) Get the unique number for this device.
uint8_t	SYSTEM_GetProdRev(void) Get the production revision for this part.
uint32_t	SYSTEM_GetSRAMBaseAddress(void) Get the SRAM Base Address.
uint16_t	SYSTEM_GetSRAMSize(void) Get the SRAM size (in KB).
uint16_t	SYSTEM_GetFlashSize(void) Get the flash size (in KB).
uint32_t	SYSTEM_GetFlashPageSize(void) Get the flash page size in bytes.
uint16_t	SYSTEM_GetPartNumber(void) Get the MCU part number.
uint8_t	SYSTEM_GetCalibrationTemperature(void) Get the calibration temperature (in degrees Celsius).

Enumeration Documentation

SYSTEM_PartFamily_TypeDef

> SYSTEM_PartFamily_TypeDef

Family identifiers.

Enumerator	
systemPartFamilyEfm32Gecko	EFM32 Gecko Device Family.
systemPartFamilyEfm32Giant	EFM32 Giant Gecko Series 0 Device Family.
systemPartFamilyEfm32Tiny	EFM32 Tiny Gecko Device Family.
systemPartFamilyEfm32Leopard	EFM32 Leopard Gecko Device Family.
systemPartFamilyEfm32Wonder	EFM32 Wonder Gecko Device Family.
systemPartFamilyEfm32Zero	EFM32 Zero Gecko Device Family.
systemPartFamilyEfm32Happy	EFM32 Happy Gecko Device Family.
systemPartFamilyEzr32Wonder	EZR32 Wonder Device Family.
systemPartFamilyEzr32Leopard	EZR32 Leopard Device Family.
systemPartFamilyEzr32Happy	EZR32 Happy Device Family.
systemPartFamilyGecko	Gecko Device Family.

systemPartFamilyGiant	Giant Gecko Device Family.
systemPartFamilyTiny	Tiny Gecko Device Family.
systemPartFamilyLeopard	Leopard Gecko Device Family.
systemPartFamilyWonder	Wonder Gecko Device Family.
systemPartFamilyZero	Zero Gecko Device Family.
systemPartFamilyHappy	Happy Gecko Device Family.
systemPartFamilyUnknown	Unknown Device Family.

SYSTEM_FpuAccess_TypeDef

> SYSTEM_FpuAccess_TypeDef

Floating point co-processor access modes.

Enumerator	
fpuAccessDenied	Access denied, any attempted access generates a NOCP UsageFault.
fpuAccessPrivilegedOnly	Privileged access only, an unprivileged access generates a NOCP UsageFault.
fpuAccessReserved	Reserved.
fpuAccessFull	Full access.

SYSTEM_SecurityCapability_TypeDef

> SYSTEM_SecurityCapability_TypeDef

Family security capability.

Enumerator	
securityCapabilityUnknown	Unknown security capability.
securityCapabilityNA	Security capability not applicable.
securityCapabilityBasic	Basic security capability.
securityCapabilityRoT	Root of Trust security capability.
securityCapabilitySE	Secure Element security capability.
securityCapabilityVault	Secure Vault security capability.

Function Documentation

SYSTEM_ChipRevisionGet

> void SYSTEM_ChipRevisionGet (SYSTEM_ChipRevision_TypeDef * rev)

Get a chip major/minor revision.

Parameters

Type	Direction	Argument Name	Description
SYSTEM_ChipRevision_TypeDef *	[out]	rev	A location to place the chip revision information.

SYSTEM_GetFamily

```
SYSTEM_PartFamily_TypeDef SYSTEM_GetFamily (void )
```

Get the MCU family identifier.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function retrieves family ID by reading the chip's device info structure in flash memory. Users can retrieve family ID directly by reading DEVINFO->PART item and decode with mask and shift #defines defined in <part_family>_devinfo.h (refer to code below for details).

Returns

- Family identifier of MCU.

SYSTEM_FpuAccessModeSet

```
void SYSTEM_FpuAccessModeSet (SYSTEM_FpuAccess_TypeDef accessMode)
```

Set floating point co-processor (FPU) access mode.

Parameters

Type	Direction	Argument Name	Description
SYSTEM_FpuAccess_TypeDef	[in]	accessMode	Floating point co-processor access mode. See SYSTEM_FpuAccess_TypeDef for details.

SYSTEM_GetCalibrationValue

```
bool SYSTEM_GetCalibrationValue (volatile uint32_t * regAddress)
```

Get a factory calibration value for a given peripheral register.

Parameters

Type	Direction	Argument Name	Description
volatile uint32_t *	[in]	regAddress	The peripheral calibration register address to get a calibration value for. If the calibration value is found, this register is updated with the calibration value.

Returns

- True if a calibration value exists, false otherwise.

SYSTEM_GetSecurityCapability

```
SYSTEM_SecurityCapability_TypeDef SYSTEM_GetSecurityCapability (void )
```

Get family security capability.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function retrieves the family security capability based on the device number. The device number is one letter and 3 digits: DEVICENUMBER = (alpha-'A')*1000 + numeric. i.e. 0d = "A000"; 1123d = "B123". The security capabilities are represented by SYSTEM_SecurityCapability_TypeDef.

Returns

- Security capability of MCU.

SYSTEM_GetUnique

```
uint64_t SYSTEM_GetUnique (void )
```

Get the unique number for this device.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Unique number for this device.

SYSTEM_GetProdRev

```
uint8_t SYSTEM_GetProdRev (void )
```

Get the production revision for this part.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Production revision for this part.

SYSTEM_GetSRAMBaseAddress

```
uint32_t SYSTEM_GetSRAMBaseAddress (void )
```

Get the SRAM Base Address.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function is used to retrieve the base address of the SRAM.

Returns

- Base address SRAM (32-bit unsigned integer).

SYSTEM_GetSRAMSize

```
uint16_t SYSTEM_GetSRAMSize (void )
```

Get the SRAM size (in KB).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function retrieves SRAM size by reading the chip device info structure. If your binary is made for one specific device only, use SRAM_SIZE instead.

Returns

- Size of internal SRAM (in KB).

SYSTEM_GetFlashSize

```
uint16_t SYSTEM_GetFlashSize (void )
```

Get the flash size (in KB).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function retrieves flash size by reading the chip device info structure. If your binary is made for one specific device only, use FLASH_SIZE instead.

Returns

- Size of internal flash (in KB).

SYSTEM_GetFlashPageSize

```
uint32_t SYSTEM_GetFlashPageSize (void )
```

Get the flash page size in bytes.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- This function retrieves flash page size by reading the chip device info structure. If your binary is made for one specific device only, use FLASH_PAGE_SIZE instead.

Returns

- Page size of internal flash in bytes.

SYSTEM_GetPartNumber

```
uint16_t SYSTEM_GetPartNumber (void )
```

Get the MCU part number.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- The part number of MCU.

SYSTEM_GetCalibrationTemperature

```
uint8_t SYSTEM_GetCalibrationTemperature (void )
```

Get the calibration temperature (in degrees Celsius).

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- Calibration temperature in Celsius.

SYSTEM_ChipRevision_TypeDef

Chip revision details.

Public Attributes

uint8_t	minor Minor revision number.
uint8_t	major Major revision number.
uint8_t	family Device family number.

Public Attribute Documentation

minor

```
uint8_t SYSTEM_ChipRevision_TypeDef::minor
```

Minor revision number.

major

```
uint8_t SYSTEM_ChipRevision_TypeDef::major
```

Major revision number.

family

```
uint8_t SYSTEM_ChipRevision_TypeDef::family
```

Device family number.

SYSTEM_CalAddrVal_TypeDef

DEVINFO calibration address/value pair.

Public Attributes

uint32_t	address Peripheral calibration register address.
uint32_t	calValue Calibration value for register at address.

Public Attribute Documentation

address

```
uint32_t SYSTEM_CalAddrVal_TypeDef::address
```

Peripheral calibration register address.

calValue

```
uint32_t SYSTEM_CalAddrVal_TypeDef::calValue
```

Calibration value for register at address.

TIMER - Timer/Counter

TIMER - Timer/Counter

Timer/Counter (TIMER) Peripheral API.

The timer module consists of three main parts:

- General timer configuration and enable control.
- Compare/capture control.
- Dead time insertion control (may not be available for all timers).

Modules

TIMER_Init_TypeDef

TIMER_InitCC_TypeDef

TIMER_InitDTI_TypeDef

Enumerations

enum TIMER_CCMode_TypeDef {

  timerCCModeOff = _TIMER_CC_CTRL_MODE_OFF
  timerCCModeCapture = _TIMER_CC_CTRL_MODE_INPUTCAPTURE
  timerCCModeCompare = _TIMER_CC_CTRL_MODE_OUTPUTCOMPARE
  timerCCModePWM = _TIMER_CC_CTRL_MODE_PWM

}

Timer compare/capture mode.

enum TIMER_ClkSel_TypeDef {

  timerClkSelHFPerClk = _TIMER_CTRL_CLKSEL_PRESCHFPERCLK
  timerClkSelCC1 = _TIMER_CTRL_CLKSEL_CC1
  timerClkSelCascade = _TIMER_CTRL_CLKSEL_TIMEROUF

}

Clock select.

enum TIMER_Edge_TypeDef {

  timerEdgeRising = _TIMER_CC_CTRL_ICEDGE_RISING
  timerEdgeFalling = _TIMER_CC_CTRL_ICEDGE_FALLING
  timerEdgeBoth = _TIMER_CC_CTRL_ICEDGE_BOTH
  timerEdgeNone = _TIMER_CC_CTRL_ICEDGE_NONE

}

Input capture edge select.

enum TIMER_Event_TypeDef {

  timerEventEveryEdge = _TIMER_CC_CTRL_ICEVCTRL_EVERYEDGE
  timerEventEvery2ndEdge = _TIMER_CC_CTRL_ICEVCTRL_EVERYSECONDEDGE
  timerEventRising = _TIMER_CC_CTRL_ICEVCTRL_RISING
  timerEventFalling = _TIMER_CC_CTRL_ICEVCTRL_FALLING

}

Input capture event control.

enum TIMER_InputAction_TypeDef {

timerInputActionNone = _TIMER_CTRL_FALLA_NONE
timerInputActionStart = _TIMER_CTRL_FALLA_START
timerInputActionStop = _TIMER_CTRL_FALLA_STOP
timerInputActionReloadStart = _TIMER_CTRL_FALLA_RELOADSTART

}
Input edge action.

enum TIMER_Mode_TypeDef {

timerModeUp = _TIMER_CTRL_MODE_UP
timerModeDown = _TIMER_CTRL_MODE_DOWN
timerModeUpDown = _TIMER_CTRL_MODE_UPDOWN
timerModeQDec = _TIMER_CTRL_MODE_QDEC

}
Timer mode.

enum TIMER_OutputAction_TypeDef {

timerOutputActionNone = _TIMER_CC_CTRL_CUFOA_NONE
timerOutputActionToggle = _TIMER_CC_CTRL_CUFOA_TOGGLE
timerOutputActionClear = _TIMER_CC_CTRL_CUFOA_CLEAR
timerOutputActionSet = _TIMER_CC_CTRL_CUFOA_SET

}
Compare/capture output action.

enum TIMER_Prescale_TypeDef {

timerPrescale1 = _TIMER_CTRL_PRESC_DIV1
timerPrescale2 = _TIMER_CTRL_PRESC_DIV2
timerPrescale4 = _TIMER_CTRL_PRESC_DIV4
timerPrescale8 = _TIMER_CTRL_PRESC_DIV8
timerPrescale16 = _TIMER_CTRL_PRESC_DIV16
timerPrescale32 = _TIMER_CTRL_PRESC_DIV32
timerPrescale64 = _TIMER_CTRL_PRESC_DIV64
timerPrescale128 = _TIMER_CTRL_PRESC_DIV128
timerPrescale256 = _TIMER_CTRL_PRESC_DIV256
timerPrescale512 = _TIMER_CTRL_PRESC_DIV512
timerPrescale1024 = _TIMER_CTRL_PRESC_DIV1024

}
Prescaler.

enum TIMER_DtiFaultAction_TypeDef {

timerDtiFaultActionNone = _TIMER_DTFC_DTFA_NONE
timerDtiFaultActionInactive = _TIMER_DTFC_DTFA_INACTIVE
timerDtiFaultActionClear = _TIMER_DTFC_DTFA_CLEAR
timerDtiFaultActionTristate = _TIMER_DTFC_DTFA_TRISTATE

}
DT (Dead Time) Fault Actions.

enum TIMER_PrsOutput_t {

timerPrsOutputPulse = 0
timerPrsOutputLevel = 1
timerPrsOutputDefault = timerPrsOutputPulse

}
PRS Output configuration.

Typedefs

typedef uint8_t TIMER_PRSSEL_TypeDef
Peripheral Reflex System signal.

Functions

void TIMER_Init(TIMER_TypeDef *timer, const TIMER_Init_TypeDef *init)
Initialize TIMER.

void TIMER_InitCC(TIMER_TypeDef *timer, unsigned int ch, const TIMER_InitCC_TypeDef *init)
Initialize the TIMER compare/capture channel.

void TIMER_InitDTI(TIMER_TypeDef *timer, const TIMER_InitDTI_TypeDef *init)
Initialize the TIMER DTI unit.

void TIMER_Reset(TIMER_TypeDef *timer)
Reset the TIMER to the same state that it was in after a hardware reset.

bool TIMER_Valid(const TIMER_TypeDef *ref)
Validate the TIMER register block pointer.

bool TIMER_SupportsDTI(const TIMER_TypeDef *ref)
Check whether the TIMER is valid and supports Dead Timer Insertion (DTI).

uint32_t TIMER_MaxCount(const TIMER_TypeDef *ref)
Get the Max count of the timer.

uint32_t TIMER_CaptureGet(TIMER_TypeDef *timer, unsigned int ch)
Get compare/capture value for the compare/capture channel.

uint32_t TIMER_CaptureBufGet(TIMER_TypeDef *timer, unsigned int ch)
Get the buffered compare/capture value for compare/capture channel.

void TIMER_CompareBufSet(TIMER_TypeDef *timer, unsigned int ch, uint32_t val)
Set the compare value buffer for the compare/capture channel when operating in compare or PWM mode.

void TIMER_CompareSet(TIMER_TypeDef *timer, unsigned int ch, uint32_t val)
Set the compare value for compare/capture channel when operating in compare or PWM mode.

uint32_t TIMER_CounterGet(TIMER_TypeDef *timer)
Get the TIMER counter value.

void TIMER_CounterSet(TIMER_TypeDef *timer, uint32_t val)
Set the TIMER counter value.

void TIMER_Enable(TIMER_TypeDef *timer, bool enable)
Start/stop TIMER.

void TIMER_EnableDTI(TIMER_TypeDef *timer, bool enable)
Enable or disable DTI unit.

uint32_t TIMER_GetDTIFault(TIMER_TypeDef *timer)
Get DTI fault source flags status.

void TIMER_ClearDTIFault(TIMER_TypeDef *timer, uint32_t flags)
Clear DTI fault source flags.

void TIMER_IntClear(TIMER_TypeDef *timer, uint32_t flags)
Clear one or more pending TIMER interrupts.

void TIMER_IntDisable(TIMER_TypeDef *timer, uint32_t flags)
Disable one or more TIMER interrupts.

void TIMER_IntEnable(TIMER_TypeDef *timer, uint32_t flags)
Enable one or more TIMER interrupts.

uint32_t TIMER_IntGet(TIMER_TypeDef *timer)
Get pending TIMER interrupt flags.

uint32_t TIMER_IntGetEnabled(TIMER_TypeDef *timer)
Get enabled and pending TIMER interrupt flags.

void TIMER_IntSet(TIMER_TypeDef *timer, uint32_t flags)
Set one or more pending TIMER interrupts from SW.

void TIMER_Lock(TIMER_TypeDef *timer)
Lock some TIMER registers to protect them from being modified.

void TIMER_TopBufSet(TIMER_TypeDef *timer, uint32_t val)
Set the top value buffer for the timer.

uint32_t TIMER_TopGet(TIMER_TypeDef *timer)
Get the top value setting for the timer.

void TIMER_TopSet(TIMER_TypeDef *timer, uint32_t val)
Set the top value for timer.

void TIMER_Unlock(TIMER_TypeDef *timer)
Unlock TIMER to enable writing to locked registers again.

Macros

#define TIMER_INIT_DEFAULT undefined
Default configuration for TIMER initialization structure.

#define TIMER_INITCC_DEFAULT undefined
Default configuration for TIMER compare/capture initialization structure.

#define TIMER_INITDTI_DEFAULT undefined
Default configuration for TIMER DTI initialization structure.

Enumeration Documentation

TIMER_CCMode_TypeDef

> TIMER_CCMode_TypeDef

Timer compare/capture mode.

Enumerator	
timerCCModeOff	Channel turned off.
timerCCModeCapture	Input capture.
timerCCModeCompare	Output compare.
timerCCModePWM	Pulse-Width modulation.

TIMER_ClkSel_TypeDef

> TIMER_ClkSel_TypeDef

Clock select.

Enumerator	
timerClkSelHFPerClk	Prescaled HFPER / HFPERB clock.
timerClkSelCC1	Compare/Capture Channel 1 Input.
timerClkSelCascade	Cascaded clocked by underflow or overflow by lower numbered timer.

TIMER_Edge_TypeDef

> TIMER_Edge_TypeDef

Input capture edge select.

Enumerator	
timerEdgeRising	Rising edges detected.
timerEdgeFalling	Falling edges detected.
timerEdgeBoth	Both edges detected.
timerEdgeNone	No edge detection, leave signal as is.

TIMER_Event_TypeDef

```
TIMER_Event_TypeDef
```

Input capture event control.

Enumerator	
timerEventEveryEdge	PRS output pulse, interrupt flag, and DMA request set on every capture.
timerEventEvery2ndEdge	PRS output pulse, interrupt flag, and DMA request set on every second capture.
timerEventRising	PRS output pulse, interrupt flag, and DMA request set on rising edge (if input capture edge = BOTH).
timerEventFalling	PRS output pulse, interrupt flag, and DMA request set on falling edge (if input capture edge = BOTH).

TIMER_InputAction_TypeDef

```
TIMER_InputAction_TypeDef
```

Input edge action.

Enumerator	
timerInputActionNone	No action taken.
timerInputActionStart	Start counter without reload.
timerInputActionStop	Stop counter without reload.
timerInputActionReloadStart	Reload and start counter.

TIMER_Mode_TypeDef

```
TIMER_Mode_TypeDef
```

Timer mode.

Enumerator	
timerModeUp	Up-counting.
timerModeDown	Down-counting.
timerModeUpDown	Up/down-counting.
timerModeQDec	Quadrature decoder.

TIMER_OutputAction_TypeDef

TIMER_OutputAction_TypeDef

Compare/capture output action.

Enumerator	
timerOutputActionNone	No action.
timerOutputActionToggle	Toggle on event.
timerOutputActionClear	Clear on event.
timerOutputActionSet	Set on event.

TIMER_Prescale_TypeDef

TIMER_Prescale_TypeDef

Prescaler.

Enumerator	
timerPrescale1	Divide by 1.
timerPrescale2	Divide by 2.
timerPrescale4	Divide by 4.
timerPrescale8	Divide by 8.
timerPrescale16	Divide by 16.
timerPrescale32	Divide by 32.
timerPrescale64	Divide by 64.
timerPrescale128	Divide by 128.
timerPrescale256	Divide by 256.
timerPrescale512	Divide by 512.
timerPrescale1024	Divide by 1024.

TIMER_DtiFaultAction_TypeDef

TIMER_DtiFaultAction_TypeDef

DT (Dead Time) Fault Actions.

Enumerator	
timerDtiFaultActionNone	No action on fault.
timerDtiFaultActionInactive	Set outputs inactive.
timerDtiFaultActionClear	Clear outputs.
timerDtiFaultActionTristate	Tristate outputs.

TIMER_PrsOutput_t

TIMER_PrsOutput_t

PRS Output configuration.

Enumerator	
timerPrsOutputPulse	Pulse PRS output from a channel.
timerPrsOutputLevel	PRS output follows CC out level.
timerPrsOutputDefault	Default PRS output behavior.

Typedef Documentation

TIMER_PRSSEL_TypeDef

```
typedef uint8_t TIMER_PRSSEL_TypeDef
```

Peripheral Reflex System signal.

Function Documentation

TIMER_Init

```
void TIMER_Init (TIMER_TypeDef * timer, const TIMER_Init_TypeDef * init)
```

Initialize TIMER.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	A pointer to the TIMER peripheral register block.
const TIMER_Init_TypeDef *	[in]	init	A pointer to the TIMER initialization structure.

Notice that the counter top must be configured separately with, for instance TIMER_TopSet(). In addition, compare/capture and dead-time insertion initialization must be initialized separately if used, which should probably be done prior to using this function if configuring the TIMER to start when initialization is completed.

TIMER_InitCC

```
void TIMER_InitCC (TIMER_TypeDef * timer, unsigned int ch, const TIMER_InitCC_TypeDef * init)
```

Initialize the TIMER compare/capture channel.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	A pointer to the TIMER peripheral register block.
unsigned int	[in]	ch	A compare/capture channel to initialize for.
const TIMER_InitCC_TypeDef *	[in]	init	A pointer to the TIMER initialization structure.

Notice that if operating the channel in compare mode, the CCV and CCVB register must be set separately, as required.

TIMER_InitDTI

```
void TIMER_InitDTI (TIMER_TypeDef * timer, const TIMER_InitDTI_TypeDef * init)
```

Initialize the TIMER DTI unit.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	A pointer to the TIMER peripheral register block.
const TIMER_InitDTI_TypeDef *	[in]	init	A pointer to the TIMER DTI initialization structure.

TIMER_Reset

```
void TIMER_Reset (TIMER_TypeDef * timer)
```

Reset the TIMER to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	A pointer to the TIMER peripheral register block.

Note

- The ROUTE register is NOT reset by this function to allow for a centralized setup of this feature.

TIMER_Valid

```
bool TIMER_Valid (const TIMER_TypeDef * ref)
```

Validate the TIMER register block pointer.

Parameters

Type	Direction	Argument Name	Description
const TIMER_TypeDef *	[in]	ref	Pointer to the TIMER peripheral register block.

Returns

- True if ref points to a valid timer, false otherwise.

TIMER_SupportsDTI

```
bool TIMER_SupportsDTI (const TIMER_TypeDef * ref)
```

Check whether the TIMER is valid and supports Dead Timer Insertion (DTI).

Parameters

Type	Direction	Argument Name	Description
const TIMER_TypeDef *	[in]	ref	Pointer to the TIMER peripheral register block.

Returns

- True if ref points to a valid timer that supports DTI, false otherwise.

TIMER_MaxCount

```
uint32_t TIMER_MaxCount (const TIMER_TypeDef * ref)
```

Get the Max count of the timer.

Parameters

Type	Direction	Argument Name	Description
const TIMER_TypeDef *	[in]	ref	Pointer to the TIMER peripheral register block.

Returns

- The max count value of the timer. This is 0xFFFF for 16 bit timers and 0xFFFFFFFF for 32 bit timers.

TIMER_CaptureGet

```
uint32_t TIMER_CaptureGet (TIMER_TypeDef * timer, unsigned int ch)
```

Get compare/capture value for the compare/capture channel.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
unsigned int	[in]	ch	Compare/capture channel to access.

Returns

- Current capture value.

TIMER_CaptureBufGet

```
uint32_t TIMER_CaptureBufGet (TIMER_TypeDef * timer, unsigned int ch)
```

Get the buffered compare/capture value for compare/capture channel.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
unsigned int	[in]	ch	Compare/capture channel to access.

Returns

- Current buffered capture value.

TIMER_CompareBufSet

```
void TIMER_CompareBufSet (TIMER_TypeDef * timer, unsigned int ch, uint32_t val)
```

Set the compare value buffer for the compare/capture channel when operating in compare or PWM mode.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
unsigned int	[in]	ch	Compare/capture channel to access.
uint32_t	[in]	val	Value to set in compare value buffer register.

The compare value buffer holds the value which will be written to TIMERn_CCx_CCV on an update event if the buffer has been updated since the last event.

TIMER_CompareSet

```
void TIMER_CompareSet (TIMER_TypeDef * timer, unsigned int ch, uint32_t val)
```

Set the compare value for compare/capture channel when operating in compare or PWM mode.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
unsigned int	[in]	ch	Compare/capture channel to access.
uint32_t	[in]	val	Value to set in compare value register.

TIMER_CounterGet

```
uint32_t TIMER_CounterGet (TIMER_TypeDef * timer)
```

Get the TIMER counter value.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to TIMER peripheral register block.

Returns

- Current TIMER counter value.

TIMER_CounterSet

```
void TIMER_CounterSet (TIMER_TypeDef * timer, uint32_t val)
```

Set the TIMER counter value.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	val	Value to set counter to.

TIMER_Enable

```
void TIMER_Enable (TIMER_TypeDef * timer, bool enable)
```

Start/stop TIMER.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
bool	[in]	enable	Set to true to enable counting; set to false otherwise.

TIMER_EnableDTI

```
void TIMER_EnableDTI (TIMER_TypeDef * timer, bool enable)
```

Enable or disable DTI unit.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
bool	[in]	enable	Set to true to enable DTI unit; set to false otherwise.

TIMER_GetDTIFault

```
uint32_t TIMER_GetDTIFault (TIMER_TypeDef * timer)
```

Get DTI fault source flags status.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.

Note

- Event bits are not cleared by this function.

Returns

- Status of the DTI fault source flags. Returns one or more valid DTI fault source flags (TIMER_DTFAULT_nnn) OR'ed together.

TIMER_ClearDTIFault

```
void TIMER_ClearDTIFault (TIMER_TypeDef * timer, uint32_t flags)
```

Clear DTI fault source flags.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	flags	DTI fault source(s) to clear. Use one or more valid DTI fault source flags (TIMER_DTFAULT_nnn) OR'ed together.

TIMER_IntClear

```
void TIMER_IntClear (TIMER_TypeDef * timer, uint32_t flags)
```

Clear one or more pending TIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	flags	Pending TIMER interrupt source(s) to clear. Use one or more valid interrupt flags for the TIMER module (TIMER_IF_nnn) OR'ed together.

TIMER_IntDisable

```
void TIMER_IntDisable (TIMER_TypeDef * timer, uint32_t flags)
```

Disable one or more TIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	flags	TIMER interrupt source(s) to disable. Use one or more valid interrupt flags for the TIMER module (TIMER_IF_nnn) OR'ed together.

TIMER_IntEnable

```
void TIMER_IntEnable (TIMER_TypeDef * timer, uint32_t flags)
```

Enable one or more TIMER interrupts.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	flags	TIMER interrupt source(s) to enable. Use one or more valid interrupt flags for the TIMER module (TIMER_IF_nnn) OR'ed together.

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using TIMER_IntClear() prior to enabling the interrupt.

TIMER_IntGet

```
uint32_t TIMER_IntGet (TIMER_TypeDef * timer)
```

Get pending TIMER interrupt flags.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.

Note

- Event bits are not cleared by this function.

Returns

- TIMER interrupt source(s) pending. Returns one or more valid interrupt flags for the TIMER module (TIMER_IF_nnn) OR'ed together.

TIMER_IntGetEnabled

```
uint32_t TIMER_IntGetEnabled (TIMER_TypeDef * timer)
```

Get enabled and pending TIMER interrupt flags.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by this function.

Returns

- Pending and enabled TIMER interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in TIMERx_IEN_nnn register (TIMERx_IEN_nnn) and
  - the OR combination of valid interrupt flags of the TIMER module (TIMERx_IF_nnn).

TIMER_IntSet

```
void TIMER_IntSet (TIMER_TypeDef * timer, uint32_t flags)
```

Set one or more pending TIMER interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	flags	TIMER interrupt source(s) to set to pending. Use one or more valid interrupt flags for the TIMER module (TIMER_IF_nnn) OR'ed together.

TIMER_Lock

```
void TIMER_Lock (TIMER_TypeDef * timer)
```

Lock some TIMER registers to protect them from being modified.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to TIMER peripheral register block.

Refer to the reference manual for TIMER registers that will be locked.

Note

- If locking the TIMER registers, they must be unlocked prior to using any TIMER API function that modifies TIMER registers protected by the lock.

TIMER_TopBufSet

```
void TIMER_TopBufSet (TIMER_TypeDef * timer, uint32_t val)
```

Set the top value buffer for the timer.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	val	Value to set in top value buffer register.

When top value buffer register is updated, value is loaded into top value register at the next wrap around. This feature is useful in order to update top value safely when timer is running.

TIMER_TopGet

```
uint32_t TIMER_TopGet (TIMER_TypeDef * timer)
```

Get the top value setting for the timer.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.

Returns

- Current top value.

TIMER_TopSet

```
void TIMER_TopSet (TIMER_TypeDef * timer, uint32_t val)
```

Set the top value for timer.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.
uint32_t	[in]	val	Value to set in top value register.

TIMER_Unlock

```
void TIMER_Unlock (TIMER_TypeDef * timer)
```

Unlock TIMER to enable writing to locked registers again.

Parameters

Type	Direction	Argument Name	Description
TIMER_TypeDef *	[in]	timer	Pointer to the TIMER peripheral register block.

Macro Definition Documentation

TIMER_INIT_DEFAULT

```
#define TIMER_INIT_DEFAULT
```

Value:

```
{                                                                     \
  true,                /* Enable timer when initialization completes. */          \
  false,               /* Stop counter during debug halt. */                      \
  timerPrescale1,      /* No prescaling. */                                       \
  timerClkSelHFPerClk, /* Select HFPER / HFPERB clock. */                         \
  false,               /* Not 2x count mode. */                                   \
  false,               /* No ATI. */                                              \
  false,               /* No RSSCOIST. */                                         \
  timerInputActionNone, /* No action on falling input edge. */                    \
  timerInputActionNone, /* No action on rising input edge. */                     \
  timerModeUp,         /* Up-counting. */                                         \
  false,               /* Do not clear DMA requests when DMA channel is active. */ \
  false,               /* Select X2 quadrature decode mode (if used). */          \
  false,               /* Disable one shot. */                                    \
  false                /* Not started/stopped/reloaded by other timers. */        \
}
```

Default configuration for TIMER initialization structure.

TIMER_INITCC_DEFAULT

```
#define TIMER_INITCC_DEFAULT
```

Value:

```
{                                                                   \
  timerEventEveryEdge,    /* Event on every capture. */                 \
  timerEdgeRising,        /* Input capture edge on rising edge. */      \
  0,                      /* Not used by default, select PRS channel 0. */ \
  timerOutputActionNone,  /* No action on underflow. */                 \
  timerOutputActionNone,  /* No action on overflow. */                  \
  timerOutputActionNone,  /* No action on match. */                     \
  timerCCModeOff,         /* Disable compare/capture channel. */        \
  false,                  /* Disable filter. */                         \
  false,                  /* No PRS input. */                           \
  false,                  /* Clear output when counter disabled. */     \
  false,                  /* Do not invert output. */                   \
  timerPrsOutputDefault,  /* Use default PRS output configuration. */   \
}
```

Default configuration for TIMER compare/capture initialization structure.

TIMER_INITDTI_DEFAULT

```
#define TIMER_INITDTI_DEFAULT
```

Value:

```
{                                                                     \
  true,                   /* Enable the DTI. */                       \
  false,                  /* CC[0|1|2] outputs are active high. */    \
  false,                  /* CDTI[0|1|2] outputs are not inverted. */ \
  false,                  /* No auto restart when debugger exits. */  \
  false,                  /* No PRS source selected. */               \
  0,                      /* Not used by default, select PRS channel 0. */ \
  timerPrescale1,         /* No prescaling.  */                       \
  0,                      /* No rise time. */                         \
  0,                      /* No fall time. */                         \
  TIMER_DTOGEN_DTOGCC0EN | TIMER_DTOGEN_DTOGCDTI0EN, /* Enable CC0 and CDTI0. */ \
  true,                   /* Enable core lockup as fault source. */   \
  true,                   /* Enable debugger as fault source. */      \
  false,                  /* Disable PRS fault source 0. */           \
  0,                      /* Not used by default, select PRS channel 0. */ \
  false,                  /* Disable PRS fault source 1. */           \
  0,                      /* Not used by default, select PRS channel 0. */ \
  timerDtiFaultActionInactive, /* No fault action. */                 \
}
```

Default configuration for TIMER DTI initialization structure.

TIMER_Init_TypeDef

TIMER initialization structure.

Public Attributes

bool	**enable** Start counting when initialization completed.
bool	**debugRun** Counter shall keep running during debug halt.
TIMER_Prescale_TypeDef	**prescale** Prescaling factor, if HFPER / HFPERB clock used.
TIMER_ClkSel_TypeDef	**clkSel** Clock selection.
bool	**count2x** 2x Count mode, counter increments/decrements by 2, meant for PWM mode.
bool	**ati** ATI (Always Track Inputs) makes CCPOL always track the polarity of the inputs.
bool	**rssCoist** Reload-Start Sets COIST When enabled, compare output is set to COIST value on a Reload-Start event.
TIMER_InputAction_TypeDef	**fallAction** Action on falling input edge.
TIMER_InputAction_TypeDef	**riseAction** Action on rising input edge.
TIMER_Mode_TypeDef	**mode** Counting mode.
bool	**dmaClrAct** DMA request clear on active.
bool	**quadModeX4** Select X2 or X4 quadrature decode mode (if used).
bool	**oneShot** Determines if only counting up or down once.
bool	**sync** Timer can be start/stop/reload by other timers.

Public Attribute Documentation

enable

```
bool TIMER_Init_TypeDef::enable
```

Start counting when initialization completed.

debugRun

```
bool TIMER_Init_TypeDef::debugRun
```

Counter shall keep running during debug halt.

prescale

```
TIMER_Prescale_TypeDef TIMER_Init_TypeDef::prescale
```

Prescaling factor, if HFPER / HFPERB clock used.

clkSel

```
TIMER_ClkSel_TypeDef TIMER_Init_TypeDef::clkSel
```

Clock selection.

count2x

```
bool TIMER_Init_TypeDef::count2x
```

2x Count mode, counter increments/decrements by 2, meant for PWM mode.

ati

```
bool TIMER_Init_TypeDef::ati
```

ATI (Always Track Inputs) makes CCPOL always track the polarity of the inputs.

rssCoist

```
bool TIMER_Init_TypeDef::rssCoist
```

Reload-Start Sets COIST When enabled, compare output is set to COIST value on a Reload-Start event.

fallAction

```
TIMER_InputAction_TypeDef TIMER_Init_TypeDef::fallAction
```

Action on falling input edge.

riseAction

```
TIMER_InputAction_TypeDef TIMER_Init_TypeDef::riseAction
```

Action on rising input edge.

mode

```
TIMER_Mode_TypeDef TIMER_Init_TypeDef::mode
```

Counting mode.

dmaClrAct

```
bool TIMER_Init_TypeDef::dmaClrAct
```

DMA request clear on active.

quadModeX4

```
bool TIMER_Init_TypeDef::quadModeX4
```

Select X2 or X4 quadrature decode mode (if used).

oneShot

```
bool TIMER_Init_TypeDef::oneShot
```

Determines if only counting up or down once.

sync

```
bool TIMER_Init_TypeDef::sync
```

Timer can be start/stop/reload by other timers.

TIMER_InitCC_TypeDef

TIMER compare/capture initialization structure.

Public Attributes

TIMER_Event_TypeDef	**eventCtrl** Input capture event control.
TIMER_Edge_TypeDef	**edge** Input capture edge select.
TIMER_PRSSEL_TypeDef	**prsSel** Peripheral reflex system trigger selection.
TIMER_OutputAction_TypeDef	**cufoa** Counter underflow output action.
TIMER_OutputAction_TypeDef	**cofoa** Counter overflow output action.
TIMER_OutputAction_TypeDef	**cmoa** Counter match output action.
TIMER_CCMode_TypeDef	**mode** Compare/capture channel mode.
bool	**filter** Enable digital filter.
bool	**prsInput** Select TIMERnCCx (false) or PRS input (true).
bool	**coist** Compare output initial state.
bool	**outInvert** Invert output from compare/capture channel.
TIMER_PrsOutput_t	**prsOutput** PRS output configuration.

Public Attribute Documentation

eventCtrl

```
TIMER_Event_TypeDef TIMER_InitCC_TypeDef::eventCtrl
```

Input capture event control.

edge

```
TIMER_Edge_TypeDef TIMER_InitCC_TypeDef::edge
```

Input capture edge select.

prsSel

> TIMER_PRSSEL_TypeDef TIMER_InitCC_TypeDef::prsSel

Peripheral reflex system trigger selection.

Only applicable if `prsInput` is enabled.

cufoa

> TIMER_OutputAction_TypeDef TIMER_InitCC_TypeDef::cufoa

Counter underflow output action.

cofoa

> TIMER_OutputAction_TypeDef TIMER_InitCC_TypeDef::cofoa

Counter overflow output action.

cmoa

> TIMER_OutputAction_TypeDef TIMER_InitCC_TypeDef::cmoa

Counter match output action.

mode

> TIMER_CCMode_TypeDef TIMER_InitCC_TypeDef::mode

Compare/capture channel mode.

filter

> bool TIMER_InitCC_TypeDef::filter

Enable digital filter.

prsInput

> bool TIMER_InitCC_TypeDef::prsInput

Select TIMERnCCx (false) or PRS input (true).

coist

bool TIMER_InitCC_TypeDef::coist

Compare output initial state.

Only used in Output Compare and PWM mode. When true, the compare/PWM output is set high when the counter is disabled. When counting resumes, this value will represent the initial value for the compare/PWM output. If the bit is cleared, the output will be cleared when the counter is disabled.

outInvert

bool TIMER_InitCC_TypeDef::outInvert

Invert output from compare/capture channel.

prsOutput

TIMER_PrsOutput_t TIMER_InitCC_TypeDef::prsOutput

PRS output configuration.

PRS output from a timer can either be a pulse output or a level output that follows the CC out value.

TIMER_InitDTI_TypeDef

TIMER Dead Time Insertion (DTI) initialization structure.

Public Attributes

bool	enable Enable DTI or leave it disabled until TIMER_EnableDTI() is called.
bool	activeLowOut DTI Output Polarity.
bool	invertComplementaryOut DTI Complementary Output Invert.
bool	autoRestart Enable Automatic Start-up functionality (when debugger exits).
bool	enablePrsSource Enable/disable PRS as DTI input.
TIMER_PRSSEL_TypeDef	prsSel Select which PRS channel as DTI input.
TIMER_Prescale_TypeDef	prescale DTI prescaling factor, if HFPER / HFPERB clock used.
unsigned int	riseTime DTI Rise Time.
unsigned int	fallTime DTI Fall Time.
uint32_t	outputsEnableMask DTI outputs enable bit mask, consisting of one bit per DTI output signal, i.e., CC0, CC1, CC2, CDTI0, CDTI1, and CDTI2.
bool	enableFaultSourceCoreLockup Enable core lockup as a fault source.
bool	enableFaultSourceDebugger Enable debugger as a fault source.
bool	enableFaultSourcePrsSel0 Enable PRS fault source 0 (`faultSourcePrsSel0`).
TIMER_PRSSEL_TypeDef	faultSourcePrsSel0 Select which PRS signal to be PRS fault source 0.
bool	enableFaultSourcePrsSel1 Enable PRS fault source 1 (`faultSourcePrsSel1`).
TIMER_PRSSEL_TypeDef	faultSourcePrsSel1 Select which PRS signal to be PRS fault source 1.
TIMER_DtiFaultAction_TypeDef	faultAction Fault Action.

Public Attribute Documentation

enable

bool TIMER_InitDTI_TypeDef::enable

Enable DTI or leave it disabled until TIMER_EnableDTI() is called.

activeLowOut

bool TIMER_InitDTI_TypeDef::activeLowOut

DTI Output Polarity.

invertComplementaryOut

bool TIMER_InitDTI_TypeDef::invertComplementaryOut

DTI Complementary Output Invert.

autoRestart

bool TIMER_InitDTI_TypeDef::autoRestart

Enable Automatic Start-up functionality (when debugger exits).

enablePrsSource

bool TIMER_InitDTI_TypeDef::enablePrsSource

Enable/disable PRS as DTI input.

prsSel

TIMER_PRSSEL_TypeDef TIMER_InitDTI_TypeDef::prsSel

Select which PRS channel as DTI input.

Only valid if `enablePrsSource` is enabled.

prescale

TIMER_Prescale_TypeDef TIMER_InitDTI_TypeDef::prescale

DTI prescaling factor, if HFPER / HFPERB clock used.

riseTime

unsigned int TIMER_InitDTI_TypeDef::riseTime

DTI Rise Time.

fallTime

unsigned int TIMER_InitDTI_TypeDef::fallTime

DTI Fall Time.

outputsEnableMask

uint32_t TIMER_InitDTI_TypeDef::outputsEnableMask

DTI outputs enable bit mask, consisting of one bit per DTI output signal, i.e., CC0, CC1, CC2, CDTI0, CDTI1, and CDTI2.

This value should consist of one or more TIMER_DTOGEN_DTOGnnnEN flags (defined in <part_name>_timer.h) OR'ed together.

enableFaultSourceCoreLockup

bool TIMER_InitDTI_TypeDef::enableFaultSourceCoreLockup

Enable core lockup as a fault source.

enableFaultSourceDebugger

bool TIMER_InitDTI_TypeDef::enableFaultSourceDebugger

Enable debugger as a fault source.

enableFaultSourcePrsSel0

bool TIMER_InitDTI_TypeDef::enableFaultSourcePrsSel0

Enable PRS fault source 0 (`faultSourcePrsSel0`).

faultSourcePrsSel0

TIMER_PRSSEL_TypeDef TIMER_InitDTI_TypeDef::faultSourcePrsSel0

Select which PRS signal to be PRS fault source 0.

enableFaultSourcePrsSel1

> bool TIMER_InitDTI_TypeDef::enableFaultSourcePrsSel1

Enable PRS fault source 1 (`faultSourcePrsSel1`).

faultSourcePrsSel1

> TIMER_PRSSEL_TypeDef TIMER_InitDTI_TypeDef::faultSourcePrsSel1

Select which PRS signal to be PRS fault source 1.

faultAction

> TIMER_DtiFaultAction_TypeDef TIMER_InitDTI_TypeDef::faultAction

Fault Action.

USART - Synchronous/Asynchronous Serial

USART - Synchronous/Asynchronous Serial

Universal Synchronous/Asynchronous Receiver/Transmitter Peripheral API.

The Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is a very flexible serial I/O module. It supports full duplex asynchronous UART communication as well as RS-485, SPI, MicroWire, and 3-wire. It can also interface with ISO7816 Smart-Cards, and IrDA devices.

The USART has a wide selection of operating modes, frame formats, and baud rates. All features are supported through the API of this module.

Triple buffering and DMA support makes high data-rates possible with minimal CPU intervention. It is possible to transmit and receive large frames while the MCU remains in EM1 Sleep.

This module does not support DMA configuration. The UARTDRV and SPIDRV drivers provide full support for DMA and more.

The following steps are necessary for basic operation:

Clock enable:

```
#if !defined(_SILICON_LABS_32B_SERIES_2)
/* USART is a HFPERCLK peripheral. Enable HFPERCLK domain and USART0.
 * We also need to enable the clock for GPIO to configure pins. */
CMU_ClockEnable(cmuClock_HFPER, true);
CMU_ClockEnable(cmuClock_USART0, true);
CMU_ClockEnable(cmuClock_GPIO, true);
#endif
```

To initialize the USART for asynchronous operation (e.g., UART):

```
/* Initialize with default settings and then update fields according to application requirements. */
USART_InitAsync_TypeDef initAsync = USART_INITASYNC_DEFAULT;
initAsync.baudrate = 38400;
USART_InitAsync(USART0, &initAsync);
```

To initialize the USART for synchronous operation (e.g., SPI):

```
/* Initialize with default settings and then update fields according to application requirements. */
USART_InitSync_TypeDef initSync = USART_INITSYNC_DEFAULT;
/* Operate as SPI master */
initSync.master = true;
/* Clock idle low, sample on falling edge. */
initSync.clockMode = usartClockMode1;
USART_InitSync(USART0, &initSync);
```

After pins are assigned for the application/board, enable pins at the desired location. Available locations can be obtained from the Pin Definitions section in the data sheet.

```
/* Enable I/O and set location */
USART0->ROUTE = USART_ROUTE_RXPEN | USART_ROUTE_TXPEN | USER_LOCATION;

/* Also enable CS and CLK pins if the USART is configured for synchronous mode.
 * Set GPIO mode. */
if (USART0->CTRL & USART_CTRL_SYNC) {
  USART0->ROUTE |= USART_ROUTE_CSPEN | USART_ROUTE_CLKPEN;

  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_TX_PORT(USER_LOCATION), AF_USART0_TX_PIN(USER_LOCATION),
gpioModePushPull, 0);
  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_RX_PORT(USER_LOCATION), AF_USART0_RX_PIN(USER_LOCATION),
gpioModeInput, 0);
  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_CS_PORT(USER_LOCATION), AF_USART0_CS_PIN(USER_LOCATION),
gpioModePushPull, 0);
  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_CLK_PORT(USER_LOCATION), AF_USART0_CLK_PIN(USER_LOCATION),
gpioModePushPull, 0);
} else {
  /* To avoid false start, configure TX pin as initial high */
  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_TX_PORT(USER_LOCATION), AF_USART0_TX_PIN(USER_LOCATION),
gpioModePushPull, 1);
  GPIO_PinModeSet((GPIO_Port_TypeDef)AF_USART0_RX_PORT(USER_LOCATION), AF_USART0_RX_PIN(USER_LOCATION),
gpioModeInput, 0);
  /* Don't enable CTS/RTS hardware flow control pins in this example. */
}
```

Note

- UART hardware flow control is not directly supported in hardware on _SILICON_LABS_32B_SERIES_0 parts.
- UARTDRV supports all types of UART flow control. Software assisted hardware flow control is available for parts without true UART hardware flow control.

Modules

USART_InitAsync_TypeDef

USART_PrsTriggerInit_TypeDef

USART_InitSync_TypeDef

USART_InitIrDA_TypeDef

USART_InitI2s_TypeDef

Enumerations

enum USART_Databits_TypeDef {

usartDatabits4 = USART_FRAME_DATABITS_FOUR
usartDatabits5 = USART_FRAME_DATABITS_FIVE
usartDatabits6 = USART_FRAME_DATABITS_SIX
usartDatabits7 = USART_FRAME_DATABITS_SEVEN
usartDatabits8 = USART_FRAME_DATABITS_EIGHT
usartDatabits9 = USART_FRAME_DATABITS_NINE
usartDatabits10 = USART_FRAME_DATABITS_TEN
usartDatabits11 = USART_FRAME_DATABITS_ELEVEN
usartDatabits12 = USART_FRAME_DATABITS_TWELVE
usartDatabits13 = USART_FRAME_DATABITS_THIRTEEN
usartDatabits14 = USART_FRAME_DATABITS_FOURTEEN
usartDatabits15 = USART_FRAME_DATABITS_FIFTEEN
usartDatabits16 = USART_FRAME_DATABITS_SIXTEEN

}

Databit selection.

enum USART_Enable_TypeDef {

usartDisable = 0x0
usartEnableRx = USART_CMD_RXEN
usartEnableTx = USART_CMD_TXEN
usartEnable = (USART_CMD_RXEN | USART_CMD_TXEN)

}
Enable selection.

enum USART_OVS_TypeDef {

usartOVS16 = USART_CTRL_OVS_X16
usartOVS8 = USART_CTRL_OVS_X8
usartOVS6 = USART_CTRL_OVS_X6
usartOVS4 = USART_CTRL_OVS_X4

}
Oversampling selection, used for asynchronous operation.

enum USART_Parity_TypeDef {

usartNoParity = USART_FRAME_PARITY_NONE
usartEvenParity = USART_FRAME_PARITY_EVEN
usartOddParity = USART_FRAME_PARITY_ODD

}
Parity selection, mainly used for asynchronous operation.

enum USART_Stopbits_TypeDef {

usartStopbits0p5 = USART_FRAME_STOPBITS_HALF
usartStopbits1 = USART_FRAME_STOPBITS_ONE
usartStopbits1p5 = USART_FRAME_STOPBITS_ONEANDAHALF
usartStopbits2 = USART_FRAME_STOPBITS_TWO

}
Stop bits selection, used for asynchronous operation.

enum USART_ClockMode_TypeDef {

usartClockMode0 = USART_CTRL_CLKPOL_IDLELOW | USART_CTRL_CLKPHA_SAMPLELEADING
usartClockMode1 = USART_CTRL_CLKPOL_IDLELOW | USART_CTRL_CLKPHA_SAMPLETRAILING
usartClockMode2 = USART_CTRL_CLKPOL_IDLEHIGH | USART_CTRL_CLKPHA_SAMPLELEADING
usartClockMode3 = USART_CTRL_CLKPOL_IDLEHIGH | USART_CTRL_CLKPHA_SAMPLETRAILING

}
Clock polarity/phase mode.

enum USART_IrDAPw_Typedef {

usartIrDAPwONE = USART_IRCTRL_IRPW_ONE
usartIrDAPwTWO = USART_IRCTRL_IRPW_TWO
usartIrDAPwTHREE = USART_IRCTRL_IRPW_THREE
usartIrDAPwFOUR = USART_IRCTRL_IRPW_FOUR

}
Pulse width selection for IrDA mode.

enum USART_I2sFormat_TypeDef {

usartI2sFormatW32D32 = USART_I2SCTRL_FORMAT_W32D32
usartI2sFormatW32D24M = USART_I2SCTRL_FORMAT_W32D24M
usartI2sFormatW32D24 = USART_I2SCTRL_FORMAT_W32D24
usartI2sFormatW32D16 = USART_I2SCTRL_FORMAT_W32D16
usartI2sFormatW32D8 = USART_I2SCTRL_FORMAT_W32D8
usartI2sFormatW16D16 = USART_I2SCTRL_FORMAT_W16D16
usartI2sFormatW16D8 = USART_I2SCTRL_FORMAT_W16D8
usartI2sFormatW8D8 = USART_I2SCTRL_FORMAT_W8D8

}
I2S format selection.

enum USART_I2sJustify_TypeDef {

usartI2sJustifyLeft = USART_I2SCTRL_JUSTIFY_LEFT
usartI2sJustifyRight = USART_I2SCTRL_JUSTIFY_RIGHT

}
I2S frame data justify.

Typedefs

typedef uint8_t **USART_PRS_Channel_t**
PRS Channel type.

Functions

void **USART_BaudrateAsyncSet**(USART_TypeDef *usart, uint32_t refFreq, uint32_t baudrate, USART_OVS_TypeDef ovs)
Configure USART/UART operating in asynchronous mode to use a given baudrate (or as close as possible to a specified baudrate).

uint32_t **USART_BaudrateCalc**(uint32_t refFreq, uint32_t clkdiv, bool syncmode, USART_OVS_TypeDef ovs)
Calculate baudrate for USART/UART given reference frequency, clock division, and oversampling rate (if async mode).

uint32_t **USART_BaudrateGet**(USART_TypeDef *usart)
Get the current baudrate for USART/UART.

void **USART_BaudrateSyncSet**(USART_TypeDef *usart, uint32_t refFreq, uint32_t baudrate)
Configure the USART operating in synchronous mode to use a given baudrate (or as close as possible to a specified baudrate).

void **USART_Enable**(USART_TypeDef *usart, USART_Enable_TypeDef enable)
Enable/disable USART/UART receiver and/or transmitter.

void **USART_InitAsync**(USART_TypeDef *usart, const USART_InitAsync_TypeDef *init)
Initialize USART/UART for normal asynchronous mode.

void **USART_InitSync**(USART_TypeDef *usart, const USART_InitSync_TypeDef *init)
Initialize USART for synchronous mode.

void **USARTn_InitIrDA**(USART_TypeDef *usart, const USART_InitIrDA_TypeDef *init)
Initialize USART for asynchronous IrDA mode.

void **USART_InitI2s**(USART_TypeDef *usart, USART_InitI2s_TypeDef *init)
Initialize USART for I2S mode.

void **USART_InitPrsTrigger**(USART_TypeDef *usart, const USART_PrsTriggerInit_TypeDef *init)
Initialize the automatic transmissions using PRS channel as a trigger.

void **USART_Reset**(USART_TypeDef *usart)
Reset USART/UART to the same state that it was in after a hardware reset.

uint8_t **USART_Rx**(USART_TypeDef *usart)
Receive one 4-8 bit frame, (or part of 10-16 bit frame).

uint16_t **USART_RxDouble**(USART_TypeDef *usart)
Receive two 4-8 bit frames or one 10-16 bit frame.

uint32_t **USART_RxDoubleExt**(USART_TypeDef *usart)
Receive two 4-9 bit frames, or one 10-16 bit frame with extended information.

uint16_t **USART_RxExt**(USART_TypeDef *usart)
Receive one 4-9 bit frame (or part of 10-16 bit frame) with extended information.

uint8_t **USART_SpiTransfer**(USART_TypeDef *usart, uint8_t data)
Perform one 8 bit frame SPI transfer.

void **USART_Tx**(USART_TypeDef *usart, uint8_t data)
Transmit one 4-9 bit frame.

void **USART_TxDouble**(USART_TypeDef *usart, uint16_t data)
Transmit two 4-9 bit frames or one 10-16 bit frame.

void **USART_TxDoubleExt**(USART_TypeDef *usart, uint32_t data)
Transmit two 4-9 bit frames or one 10-16 bit frame with extended control.

void USART_TxExt(USART_TypeDef *usart, uint16_t data)
Transmit one 4-9 bit frame with extended control.

void USART_IntClear(USART_TypeDef *usart, uint32_t flags)
Clear one or more pending USART interrupts.

void USART_IntDisable(USART_TypeDef *usart, uint32_t flags)
Disable one or more USART interrupts.

void USART_IntEnable(USART_TypeDef *usart, uint32_t flags)
Enable one or more USART interrupts.

uint32_t USART_IntGet(USART_TypeDef *usart)
Get pending USART interrupt flags.

uint32_t USART_IntGetEnabled(USART_TypeDef *usart)
Get enabled and pending USART interrupt flags.

void USART_IntSet(USART_TypeDef *usart, uint32_t flags)
Set one or more pending USART interrupts from SW.

uint32_t USART_StatusGet(USART_TypeDef *usart)
Get USART STATUS register.

uint8_t USART_RxDataGet(USART_TypeDef *usart)
Receive one 4-8 bit frame, (or part of 10-16 bit frame).

uint16_t USART_RxDoubleGet(USART_TypeDef *usart)
Receive two 4-8 bit frames, or one 10-16 bit frame.

uint32_t USART_RxDoubleXGet(USART_TypeDef *usart)
Receive two 4-9 bit frames, or one 10-16 bit frame with extended information.

uint16_t USART_RxDataXGet(USART_TypeDef *usart)
Receive one 4-9 bit frame, (or part of 10-16 bit frame) with extended information.

Macros

#define USART_INITASYNC_DEFAULT undefined
Default configuration for USART asynchronous initialization structure.

#define USART_INITPRSTRIGGER_DEFAULT undefined
Default configuration for USART PRS triggering structure.

#define USART_INITSYNC_DEFAULT undefined
Default configuration for USART sync initialization structure.

#define USART_INITIRDA_DEFAULT undefined
Default configuration for IrDA mode initialization structure.

#define USART_INITI2S_DEFAULT undefined
Default configuration for I2S mode initialization structure.

Enumeration Documentation

USART_Databits_TypeDef

> USART_Databits_TypeDef

Databit selection.

Enumerator	
usartDatabits4	4 data bits (not available for UART).
usartDatabits5	5 data bits (not available for UART).

usartDatabits6	6 data bits (not available for UART).
usartDatabits7	7 data bits (not available for UART).
usartDatabits8	8 data bits.
usartDatabits9	9 data bits.
usartDatabits10	10 data bits (not available for UART).
usartDatabits11	11 data bits (not available for UART).
usartDatabits12	12 data bits (not available for UART).
usartDatabits13	13 data bits (not available for UART).
usartDatabits14	14 data bits (not available for UART).
usartDatabits15	15 data bits (not available for UART).
usartDatabits16	16 data bits (not available for UART).

USART_Enable_TypeDef

> USART_Enable_TypeDef

Enable selection.

Enumerator	
usartDisable	Disable both receiver and transmitter.
usartEnableRx	Enable receiver only, transmitter disabled.
usartEnableTx	Enable transmitter only, receiver disabled.
usartEnable	Enable both receiver and transmitter.

USART_OVS_TypeDef

> USART_OVS_TypeDef

Oversampling selection, used for asynchronous operation.

Enumerator	
usartOVS16	16x oversampling (normal).
usartOVS8	8x oversampling.
usartOVS6	6x oversampling.
usartOVS4	4x oversampling.

USART_Parity_TypeDef

> USART_Parity_TypeDef

Parity selection, mainly used for asynchronous operation.

Enumerator	
usartNoParity	No parity.
usartEvenParity	Even parity.
usartOddParity	Odd parity.

USART_Stopbits_TypeDef

USART_Stopbits_TypeDef

Stop bits selection, used for asynchronous operation.

Enumerator	
usartStopbits0p5	0.5 stop bits.
usartStopbits1	1 stop bits.
usartStopbits1p5	1.5 stop bits.
usartStopbits2	2 stop bits.

USART_ClockMode_TypeDef

USART_ClockMode_TypeDef

Clock polarity/phase mode.

Enumerator	
usartClockMode0	Clock idle low, sample on rising edge.
usartClockMode1	Clock idle low, sample on falling edge.
usartClockMode2	Clock idle high, sample on falling edge.
usartClockMode3	Clock idle high, sample on rising edge.

USART_IrDAPw_Typedef

USART_IrDAPw_Typedef

Pulse width selection for IrDA mode.

Enumerator	
usartIrDAPwONE	IrDA pulse width is 1/16 for OVS=0 and 1/8 for OVS=1.
usartIrDAPwTWO	IrDA pulse width is 2/16 for OVS=0 and 2/8 for OVS=1.
usartIrDAPwTHREE	IrDA pulse width is 3/16 for OVS=0 and 3/8 for OVS=1.
usartIrDAPwFOUR	IrDA pulse width is 4/16 for OVS=0 and 4/8 for OVS=1.

USART_I2sFormat_TypeDef

USART_I2sFormat_TypeDef

I2S format selection.

Enumerator	
usartI2sFormatW32D32	32-bit word, 32-bit data.
usartI2sFormatW32D24M	32-bit word, 32-bit data with 8 lsb masked.
usartI2sFormatW32D24	32-bit word, 24-bit data.
usartI2sFormatW32D16	32-bit word, 16-bit data.
usartI2sFormatW32D8	32-bit word, 8-bit data.

usartI2sFormatW16D16	16-bit word, 16-bit data.
usartI2sFormatW16D8	16-bit word, 8-bit data.
usartI2sFormatW8D8	8-bit word, 8-bit data.

USART_I2sJustify_TypeDef

```
USART_I2sJustify_TypeDef
```

I2S frame data justify.

Enumerator	
usartI2sJustifyLeft	Data is left-justified within the frame.
usartI2sJustifyRight	Data is right-justified within the frame.

Typedef Documentation

USART_PRS_Channel_t

```
typedef uint8_t USART_PRS_Channel_t
```

PRS Channel type.

Function Documentation

USART_BaudrateAsyncSet

```
void USART_BaudrateAsyncSet (USART_TypeDef * usart, uint32_t refFreq, uint32_t baudrate,
USART_OVS_TypeDef ovs)
```

Configure USART/UART operating in asynchronous mode to use a given baudrate (or as close as possible to a specified baudrate).

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
uint32_t	[in]	refFreq	USART/UART reference clock frequency in Hz. If set to 0, the currently configured reference clock is assumed.
uint32_t	[in]	baudrate	Baudrate to try to achieve for USART/UART.
USART_OVS_TypeDef	[in]	ovs	Oversampling to be used. Normal is 16x oversampling but lower oversampling may be used to achieve higher rates or better baudrate accuracy in some cases. Notice that lower oversampling frequency makes the channel more vulnerable to bit faults during reception due to clock inaccuracies compared to the link partner.

USART_BaudrateCalc

```
uint32_t USART_BaudrateCalc (uint32_t refFreq, uint32_t clkdiv, bool syncmode, USART_OVS_TypeDef ovs)
```

Calculate baudrate for USART/UART given reference frequency, clock division, and oversampling rate (if async mode).

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	refFreq	USART/UART HF peripheral frequency used.
uint32_t	[in]	clkdiv	A clock division factor to be used.
bool	[in]	syncmode	• True - synchronous mode operation. • False - asynchronous mode operation.
USART_OVS_TypeDef	[in]	ovs	Oversampling used if in asynchronous mode. Not used if `syncmode` is true.

This function returns the baudrate that a USART/UART module will use if configured with the given frequency, clock divisor, and mode. Notice that this function will not use the hardware configuration. It can be used to determine if a given configuration is sufficiently accurate for the application.

Returns

- Baudrate with given settings.

USART_BaudrateGet

```
uint32_t USART_BaudrateGet (USART_TypeDef * usart)
```

Get the current baudrate for USART/UART.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.

This function returns the actual baudrate (not considering oscillator inaccuracies) used by a USART/UART peripheral.

Returns

- The current baudrate.

USART_BaudrateSyncSet

```
void USART_BaudrateSyncSet (USART_TypeDef * usart, uint32_t refFreq, uint32_t baudrate)
```

Configure the USART operating in synchronous mode to use a given baudrate (or as close as possible to a specified baudrate).

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART peripheral register block. (Cannot be used on UART modules.)
uint32_t	[in]	refFreq	A USART reference clock frequency in Hz that will be used. If set to 0, the currently-configured reference clock is assumed.

Type	Direction	Argument Name	Description
uint32_t	[in]	baudrate	Baudrate to try to achieve for USART.

The configuration will be set to use a baudrate <= the specified baudrate to ensure that the baudrate does not exceed the specified value.

The fractional clock division is suppressed, although the hardware design allows it. It could cause half clock cycles to exceed a specified limit and thus potentially violate specifications for the slave device. In some special situations, a fractional clock division may be useful even in synchronous mode, but in those cases it must be directly adjusted, possibly assisted by USART_BaudrateCalc():

Warnings

- The consequence of the aforementioned suppression of the fractional part of the clock divider is that some frequencies won't be achievable. The divider will only be able to be an integer value so the reference clock will only be dividable by N (where N is a positive integer).

USART_Enable

> void USART_Enable (USART_TypeDef * usart, USART_Enable_TypeDef enable)

Enable/disable USART/UART receiver and/or transmitter.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
USART_Enable_TypeDef	[in]	enable	Select the status for the receiver/transmitter.

Notice that this function does not do any configuration. Enabling should normally be done after initialization (if not enabled as part of initialization).

USART_InitAsync

> void USART_InitAsync (USART_TypeDef * usart, const USART_InitAsync_TypeDef * init)

Initialize USART/UART for normal asynchronous mode.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
const USART_InitAsync_TypeDef *	[in]	init	A pointer to the initialization structure used to configure the basic async setup.

This function will configure basic settings to operate in normal asynchronous mode.

A special control setup not covered by this function must be done after using this function by direct modification of the CTRL register.

Notice that pins used by the USART/UART module must be properly configured by the user explicitly for the USART/UART to work as intended. (When configuring pins, remember to consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

USART_InitSync

> void USART_InitSync (USART_TypeDef * usart, const USART_InitSync_TypeDef * init)

Initialize USART for synchronous mode.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART peripheral register block. (UART does not support this mode.)
const USART_InitSync_TypeDef *	[in]	init	A pointer to the initialization structure used to configure basic async setup.

This function will configure basic settings to operate in synchronous mode.

A special control setup not covered by this function must be done after using this function by direct modification of the CTRL register.

Notice that pins used by the USART module must be properly configured by the user explicitly for the USART to work as intended. (When configuring pins remember to consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

USARTn_InitIrDA

> void USARTn_InitIrDA (USART_TypeDef * usart, const USART_InitIrDA_TypeDef * init)

Initialize USART for asynchronous IrDA mode.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART peripheral register block.
const USART_InitIrDA_TypeDef *	[in]	init	A pointer to the initialization structure used to configure async IrDA setup.

This function will configure basic settings to operate in asynchronous IrDA mode.

A special control setup not covered by this function must be done after using this function by direct modification of the CTRL and IRCTRL registers.

Notice that pins used by the USART/UART module must be properly configured by the user explicitly for the USART/UART to work as intended. (When configuring pins, remember to consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

Note

- Not all USART instances support IrDA. See the data sheet for your device.

USART_InitI2s

> void USART_InitI2s (USART_TypeDef * usart, USART_InitI2s_TypeDef * init)

Initialize USART for I2S mode.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART peripheral register block. (UART does not support this mode.)
USART_InitI2s_TypeDef *	[in]	init	A pointer to the initialization structure used to configure the basic I2S setup.

This function will configure basic settings to operate in I2S mode.

A special control setup not covered by this function must be done after using this function by direct modification of the CTRL and I2SCTRL registers.

Notice that pins used by the USART module must be properly configured by the user explicitly for the USART to work as intended. (When configuring pins, remember to consider the sequence of configuration to avoid unintended pulses/glitches on output pins.)

Note

- This function does not apply to all USART's. See the chip Reference Manual.

USART_InitPrsTrigger

> void USART_InitPrsTrigger (USART_TypeDef * usart, const USART_PrsTriggerInit_TypeDef * init)

Initialize the automatic transmissions using PRS channel as a trigger.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to USART to configure.
const USART_PrsTriggerInit_TypeDef *	[in]	init	A pointer to the initialization structure.

Note

- Initialize USART with USART_Init() before setting up the PRS configuration.

USART_Reset

> void USART_Reset (USART_TypeDef * usart)

Reset USART/UART to the same state that it was in after a hardware reset.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to USART/UART peripheral register block.

USART_Rx

> uint8_t USART_Rx (USART_TypeDef * usart)

Receive one 4-8 bit frame, (or part of 10-16 bit frame).

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.

This function is normally used to receive one frame when operating with frame length 4-8 bits. See USART_RxExt() for reception of 9 bit frames.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is empty until data is received. Alternatively, the user can explicitly check whether data is available. If data is available, call USART_RxDataGet() to read the RXDATA register directly.

Returns

- Data received.

USART_RxDouble

```
uint16_t USART_RxDouble (USART_TypeDef * usart)
```

Receive two 4-8 bit frames or one 10-16 bit frame.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.

This function is normally used to receive one frame when operating with frame length 10-16 bits. See USART_RxDoubleExt() for reception of two 9 bit frames.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is empty until data is received. Alternatively, the user can explicitly check whether data is available. If data is available, call USART_RxDoubleGet() to read the RXDOUBLE register directly.

Returns

- Data received.

USART_RxDoubleExt

```
uint32_t USART_RxDoubleExt (USART_TypeDef * usart)
```

Receive two 4-9 bit frames, or one 10-16 bit frame with extended information.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.

This function is normally used to receive one frame when operating with frame length 10-16 bits and additional RX status information is required.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if buffer is empty until data is received. Alternatively, the user can explicitly check whether data is available. If data is available, call USART_RxDoubleXGet() to read the RXDOUBLEX register directly.

Returns

- Data received.

USART_RxExt

```
uint16_t USART_RxExt (USART_TypeDef * usart)
```

Receive one 4-9 bit frame (or part of 10-16 bit frame) with extended information.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.

This function is normally used to receive one frame when operating with frame length 4-9 bits and additional RX status information is required.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is empty until data is received. Alternatively, the user can explicitly check whether data is available. If data is available, call USART_RxDataXGet() to read the RXDATAX register directly.

Returns

- Data received.

USART_SpiTransfer

```
uint8_t USART_SpiTransfer (USART_TypeDef * usart, uint8_t data)
```

Perform one 8 bit frame SPI transfer.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART peripheral register block.
uint8_t	[in]	data	Data to transmit.

Note

- This function will stall if the transmit buffer is full. When a transmit buffer becomes available, data is written and the function will wait until data is fully transmitted. The SPI return value is then read out and returned.

Returns

- Data received.

USART_Tx

```
void USART_Tx (USART_TypeDef * usart, uint8_t data)
```

Transmit one 4-9 bit frame.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
uint8_t	[in]	data	Data to transmit. See details above for more information.

Depending on the frame length configuration, 4-8 (least significant) bits from `data` are transmitted. If the frame length is 9, 8 bits are transmitted from `data` and one bit as specified by CTRL register, BIT8DV field. See USART_TxExt() for transmitting 9 bit frame with full control of all 9 bits.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is full until the buffer becomes available.

USART_TxDouble

```
void USART_TxDouble (USART_TypeDef * usart, uint16_t data)
```

Transmit two 4-9 bit frames or one 10-16 bit frame.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
uint16_t	[in]	data	Data to transmit, the least significant byte holds the frame transmitted first. See details above for more info.

Depending on the frame length configuration, 4-8 (least significant) bits from each byte in `data` are transmitted. If frame length is 9, 8 bits are transmitted from each byte in `data` adding one bit as specified by the CTRL register, BIT8DV field, to each byte. See USART_TxDoubleExt() for transmitting two 9 bit frames with full control of all 9 bits.

If the frame length is 10-16, 10-16 (least significant) bits from `data` are transmitted.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is full until the buffer becomes available.

USART_TxDoubleExt

```
void USART_TxDoubleExt (USART_TypeDef * usart, uint32_t data)
```

Transmit two 4-9 bit frames or one 10-16 bit frame with extended control.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
uint32_t	[in]	data	Data to transmit with extended control. Contains two 16 bit words concatenated. Least significant word holds the frame transmitted first. If the frame length is 4-9, two frames with 4-9 least significant bits from each 16 bit word are transmitted.

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is full until the buffer becomes available.
- If the frame length is 10-16 bits, 8 data bits are taken from the least significant 16 bit word and the remaining bits from the other 16 bit word.
- Additional control bits are available as documented in the reference manual (set to 0 if not used). For 10-16 bit frame length, these control bits are taken from the most significant 16 bit word.

USART_TxExt

```
void USART_TxExt (USART_TypeDef * usart, uint16_t data)
```

Transmit one 4-9 bit frame with extended control.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	A pointer to the USART/UART peripheral register block.
uint16_t	[in]	data	Data to transmit with extended control. Least significant bit contains frame bits. Additional control bits are available as documented in the reference manual (set to 0 if not used).

Notice that possible parity/stop bits in asynchronous mode are not considered part of a specified frame bit length.

Note

- This function will stall if the buffer is full until the buffer becomes available.

USART_IntClear

```
void USART_IntClear (USART_TypeDef * usart, uint32_t flags)
```

Clear one or more pending USART interrupts.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	Pending USART/UART interrupt source(s) to clear. Use one or more valid interrupt flags for the USART module (USART_IF_nnn) OR'ed together.

USART_IntDisable

```
void USART_IntDisable (USART_TypeDef * usart, uint32_t flags)
```

Disable one or more USART interrupts.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.
uint32_t	[in]	flags	USART/UART interrupt source(s) to disable. Use one or more valid interrupt flags for the USART module (USART_IF_nnn) OR'ed together.

USART_IntEnable

```
void USART_IntEnable (USART_TypeDef * usart, uint32_t flags)
```

Enable one or more USART interrupts.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.
uint32_t	[in]	flags	USART/UART interrupt source(s) to enable. Use one or more valid interrupt flags for the USART module (USART_IF_nnn) OR'ed together.

Note

- Depending on the use, a pending interrupt may already be set prior to enabling the interrupt. To ignore a pending interrupt, consider using USART_IntClear() prior to enabling the interrupt.

USART_IntGet

```
uint32_t USART_IntGet (USART_TypeDef * usart)
```

Get pending USART interrupt flags.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.

Note

- The event bits are not cleared by the use of this function.

Returns

- USART/UART interrupt source(s) pending. Returns one or more valid interrupt flags for the USART module (USART_IF_nnn) OR'ed together.

USART_IntGetEnabled

```
uint32_t USART_IntGetEnabled (USART_TypeDef * usart)
```

Get enabled and pending USART interrupt flags.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.

Useful for handling more interrupt sources in the same interrupt handler.

Note

- Interrupt flags are not cleared by the use of this function.

Returns

- Pending and enabled USART interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in USARTx_IEN_nnn register (USARTx_IEN_nnn) and
  - the OR combination of valid interrupt flags of the USART module (USARTx_IF_nnn).

USART_IntSet

```
void USART_IntSet (USART_TypeDef * usart, uint32_t flags)
```

Set one or more pending USART interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.
uint32_t	[in]	flags	USART/UART interrupt source(s) to set to pending. Use one or more valid interrupt flags for the USART module (USART_IF_nnn) OR'ed together.

USART_StatusGet

```
uint32_t USART_StatusGet (USART_TypeDef * usart)
```

Get USART STATUS register.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to the USART/UART peripheral register block.

Returns

STATUS register value.

USART_RxDataGet

```
uint8_t USART_RxDataGet (USART_TypeDef * usart)
```

Receive one 4-8 bit frame, (or part of 10-16 bit frame).

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to USART/UART peripheral register block.

This function is used to quickly receive one 4-8 bits frame by reading the RXDATA register directly, without checking the STATUS register for the RXDATAV flag. This can be useful from the RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read the received data. Please refer to USART_RxDataXGet() for reception of 9 bit frames.

Note

- Because this function does not check whether the RXDATA register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The USART_Rx() is normally a better choice if the validity of the RXDATA register is not certain.
- Notice that possible parity/stop bits in asynchronous mode are not considered part of specified frame bit length.

Returns

- Data received.

USART_RxDoubleGet

```
uint16_t USART_RxDoubleGet (USART_TypeDef * usart)
```

Receive two 4-8 bit frames, or one 10-16 bit frame.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to USART/UART peripheral register block.

This function is used to quickly receive one 10-16 bits frame or two 4-8 bit frames by reading the RXDOUBLE register directly, without checking the STATUS register for the RXDATAV flag. This can be useful from the RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read the received data. This function is normally used to receive one frame when operating with frame length 10-16 bits. Please refer to USART_RxDoubleXGet() for reception of two 9 bit frames.

Note

- Because this function does not check whether the RXDOUBLE register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The USART_RxDouble() is normally a better choice if the validity of the RXDOUBLE register is not certain.
- Notice that possible parity/stop bits in asynchronous mode are not considered part of specified frame bit length.

Returns

Data received.

USART_RxDoubleXGet

```
uint32_t USART_RxDoubleXGet (USART_TypeDef * usart)
```

Receive two 4-9 bit frames, or one 10-16 bit frame with extended information.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to USART/UART peripheral register block.

This function is used to quickly receive one 10-16 bits frame or two 4-9 bit frames by reading the RXDOUBLEX register directly, without checking the STATUS register for the RXDATAV flag. This can be useful from the RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read the received data.

Note

- Because this function does not check whether the RXDOUBLEX register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The USART_RxDoubleExt() is normally a better choice if the validity of the RXDOUBLEX register is not certain.
- Notice that possible parity/stop bits in asynchronous mode are not considered part of specified frame bit length.

Returns

- Data received.

USART_RxDataXGet

```
uint16_t USART_RxDataXGet (USART_TypeDef * usart)
```

Receive one 4-9 bit frame, (or part of 10-16 bit frame) with extended information.

Parameters

Type	Direction	Argument Name	Description
USART_TypeDef *	[in]	usart	Pointer to USART/UART peripheral register block.

This function is used to quickly receive one 4-9 bit frame, (or part of 10-16 bit frame) with extended information by reading the RXDATAX register directly, without checking the STATUS register for the RXDATAV flag. This can be useful from the RXDATAV interrupt handler, i.e., waiting is superfluous, in order to quickly read the received data.

Note

- Because this function does not check whether the RXDATAX register actually holds valid data, it should only be used in situations when it is certain that there is valid data, ensured by some external program routine, e.g., when handling an RXDATAV interrupt. The USART_RxExt() is normally a better choice if the validity of the RXDATAX register is not certain.
- Notice that possible parity/stop bits in asynchronous mode are not considered part of specified frame bit length.

Returns

- Data received.

Macro Definition Documentation

USART_INITASYNC_DEFAULT

```
#define USART_INITASYNC_DEFAULT
```

Value:

```
{                                                                                \
  usartEnable,          /* Enable RX/TX when initialization is complete. */      \
  0,                    /* Use current configured reference clock for configuring baud rate. */ \
  115200,               /* 115200 bits/s. */                                     \
  usartOVS16,           /* 16x oversampling. */                                  \
  usartDatabits8,       /* 8 data bits. */                                       \
  usartNoParity,        /* No parity. */                                         \
  usartStopbits1,       /* 1 stop bit. */                                        \
  false,                /* Do not disable majority vote. */                      \
  false,                /* Not USART PRS input mode. */                          \
  0,                    /* PRS channel 0. */                                     \
  false,                /* Auto CS functionality enable/disable switch. */       \
  false,                /* No CS invert. */                                      \
}
```

Default configuration for USART asynchronous initialization structure.

USART_INITPRSTRIGGER_DEFAULT

```
#define USART_INITPRSTRIGGER_DEFAULT
```

Value:

```
{                                                   \
  false,            /* Do not enable autoTX triggering. */   \
  false,            /* Do not enable receive triggering. */  \
  false,            /* Do not enable transmit triggering. */ \
  0                 /* Set default channel to zero. */       \
}
```

Default configuration for USART PRS triggering structure.

USART_INITSYNC_DEFAULT

```
#define USART_INITSYNC_DEFAULT
```

Value:

```
{                                                                               \
  usartEnable,     /* Enable RX/TX when initialization is complete. */          \
  0,               /* Use current configured reference clock for configuring baud rate. */ \
  1000000,         /* 1 Mbits/s. */                                             \
  usartDatabits8,  /* 8 data bits. */                                           \
  true,            /* Master mode. */                                           \
  false,           /* Send least significant bit first. */                      \
  usartClockMode0, /* Clock idle low, sample on rising edge. */                 \
  false,           /* Not USART PRS input mode. */                              \
  0,               /* PRS channel 0. */                                         \
  false,           /* No AUTOTX mode. */                                        \
  false,           /* No AUTOCS mode. */                                        \
  false,           /* No CS invert. */                                          \
}
```

Default configuration for USART sync initialization structure.

USART_INITIRDA_DEFAULT

```
#define USART_INITIRDA_DEFAULT
```

Value:

```
{                                                                                \
  {                                                                              \
    usartEnable,           /* Enable RX/TX when initialization is complete. */   \
    0,                     /* Use current configured reference clock for configuring baud rate. */ \
    115200,                /* 115200 bits/s. */                                  \
    usartOVS16,            /* 16x oversampling. */                               \
    usartDatabits8,        /* 8 data bits. */                                    \
    usartEvenParity,       /* Even parity. */                                    \
    usartStopbits1,        /* 1 stop bit. */                                     \
    false,                 /* Do not disable majority vote. */                   \
    false,                 /* Not USART PRS input mode. */                       \
    0,                     /* PRS channel 0. */                                  \
    false,                 /* Auto CS functionality enable/disable switch */     \
    false,                 /* No CS invert. */                                   \
  },                                                                             \
  false,            /* Rx invert disabled. */                                    \
  false,            /* Filtering disabled. */                                    \
  usartIrDAPwTHREE, /* Pulse width is set to ONE. */                             \
  false,            /* Routing to PRS is disabled. */                            \
  0                 /* PRS channel 0. */                                         \
}
```

Default configuration for IrDA mode initialization structure.

USART_INITI2S_DEFAULT

```
#define USART_INITI2S_DEFAULT
```

Value:

```
{                                                                                \
  {                                                                              \
    usartEnable,     /* Enable RX/TX when initialization is complete. */         \
    0,               /* Use current configured reference clock for configuring baud rate. */ \
    1000000,         /* 1 Mbits/s. */                                            \
    usartDatabits16, /* 16 databits. */                                          \
    true,            /* Master mode. */                                          \
    true,            /* Most significant bit first. */                           \
    usartClockMode0, /* Clock idle low, sample on rising edge. */                \
    false,           /* Not USART PRS input mode. */                             \
    0,               /* PRS channel 0. */                                        \
    false,           /* No AUTOTX mode. */                                       \
    false,           /* No AUTOCS mode */                                        \
    false,           /* No CS invert. */                                         \
  },                                                                             \
  usartI2sFormatW16D16, /* 16-bit word, 16-bit data */                           \
  true,                /* Delay on I2S data. */                                  \
  false,               /* No DMA split. */                                       \
  usartI2sJustifyLeft,/* Data is left-justified within the frame */             \
  false                /* Stereo mode. */                                        \
}
```

Default configuration for I2S mode initialization structure.

USART_InitAsync_TypeDef

Asynchronous mode initialization structure.

Public Attributes

USART_Enable_TypeDef	**enable** Specifies whether TX and/or RX is enabled when initialization is completed.
uint32_t	**refFreq** USART/UART reference clock assumed when configuring baud rate setup.
uint32_t	**baudrate** Desired baud rate.
USART_OVS_TypeDef	**oversampling** Oversampling used.
USART_Databits_TypeDef	**databits** Number of data bits in frame.
USART_Parity_TypeDef	**parity** Parity mode to use.
USART_Stopbits_TypeDef	**stopbits** Number of stop bits to use.
bool	**mvdis** Majority Vote Disable for 16x, 8x and 6x oversampling modes.
bool	**prsRxEnable** Enable USART Rx via PRS.
USART_PRS_Channel_t	**prsRxCh** Select PRS channel for USART Rx.
bool	**autoCsEnable** Auto CS enabling.
bool	**csInv** Enable CS invert.

Public Attribute Documentation

enable

```
USART_Enable_TypeDef USART_InitAsync_TypeDef::enable
```

Specifies whether TX and/or RX is enabled when initialization is completed.

refFreq

```
uint32_t USART_InitAsync_TypeDef::refFreq
```

USART/UART reference clock assumed when configuring baud rate setup.

Set to 0 to use the currently configured reference clock.

baudrate

```
uint32_t USART_InitAsync_TypeDef::baudrate
```

Desired baud rate.

oversampling

```
USART_OVS_TypeDef USART_InitAsync_TypeDef::oversampling
```

Oversampling used.

databits

```
USART_Databits_TypeDef USART_InitAsync_TypeDef::databits
```

Number of data bits in frame.

Notice that UART modules only support 8 or 9 data bits.

parity

```
USART_Parity_TypeDef USART_InitAsync_TypeDef::parity
```

Parity mode to use.

stopbits

```
USART_Stopbits_TypeDef USART_InitAsync_TypeDef::stopbits
```

Number of stop bits to use.

mvdis

```
bool USART_InitAsync_TypeDef::mvdis
```

Majority Vote Disable for 16x, 8x and 6x oversampling modes.

prsRxEnable

```
bool USART_InitAsync_TypeDef::prsRxEnable
```

Enable USART Rx via PRS.

prsRxCh

```
USART_PRS_Channel_t USART_InitAsync_TypeDef::prsRxCh
```

Select PRS channel for USART Rx.

(Only valid if prsRxEnable is true).

autoCsEnable

```
bool USART_InitAsync_TypeDef::autoCsEnable
```

Auto CS enabling.

csInv

```
bool USART_InitAsync_TypeDef::csInv
```

Enable CS invert.

By default, chip select is active low. Set to true to make chip select active high.

USART_PrsTriggerInit_TypeDef

USART PRS trigger enable.

Public Attributes

bool	autoTxTriggerEnable Enable AUTOTX.
bool	rxTriggerEnable Trigger receive via PRS channel.
bool	txTriggerEnable Trigger transmit via PRS channel.
USART_PRS_Channel_t	prsTriggerChannel PRS channel to be used to trigger auto transmission.

Public Attribute Documentation

autoTxTriggerEnable

```
bool USART_PrsTriggerInit_TypeDef::autoTxTriggerEnable
```

Enable AUTOTX.

rxTriggerEnable

```
bool USART_PrsTriggerInit_TypeDef::rxTriggerEnable
```

Trigger receive via PRS channel.

txTriggerEnable

```
bool USART_PrsTriggerInit_TypeDef::txTriggerEnable
```

Trigger transmit via PRS channel.

prsTriggerChannel

```
USART_PRS_Channel_t USART_PrsTriggerInit_TypeDef::prsTriggerChannel
```

PRS channel to be used to trigger auto transmission.

USART_InitSync_TypeDef

Synchronous mode initialization structure.

Public Attributes

USART_Enable_TypeDef	**enable** Specifies whether TX and/or RX shall be enabled when initialization is completed.
uint32_t	**refFreq** USART/UART reference clock assumed when configuring baud rate setup.
uint32_t	**baudrate** Desired baud rate.
USART_Databits_TypeDef	**databits** Number of data bits in frame.
bool	**master** Select if to operate in master or slave mode.
bool	**msbf** Select if to send most or least significant bit first.
USART_ClockMode_TypeDef	**clockMode** Clock polarity/phase mode.
bool	**prsRxEnable** Enable USART Rx via PRS.
USART_PRS_Channel_t	**prsRxCh** Select PRS channel for USART Rx.
bool	**autoTx** Enable AUTOTX mode.
bool	**autoCsEnable** Auto CS enabling.
bool	**csInv** Enable CS invert.

Public Attribute Documentation

enable

```
USART_Enable_TypeDef USART_InitSync_TypeDef::enable
```

Specifies whether TX and/or RX shall be enabled when initialization is completed.

refFreq

```
uint32_t USART_InitSync_TypeDef::refFreq
```

USART/UART reference clock assumed when configuring baud rate setup.

Set to 0 to use the currently configured reference clock.

baudrate

```
uint32_t USART_InitSync_TypeDef::baudrate
```

Desired baud rate.

databits

```
USART_Databits_TypeDef USART_InitSync_TypeDef::databits
```

Number of data bits in frame.

master

```
bool USART_InitSync_TypeDef::master
```

Select if to operate in master or slave mode.

msbf

```
bool USART_InitSync_TypeDef::msbf
```

Select if to send most or least significant bit first.

clockMode

```
USART_ClockMode_TypeDef USART_InitSync_TypeDef::clockMode
```

Clock polarity/phase mode.

prsRxEnable

```
bool USART_InitSync_TypeDef::prsRxEnable
```

Enable USART Rx via PRS.

prsRxCh

```
USART_PRS_Channel_t USART_InitSync_TypeDef::prsRxCh
```

Select PRS channel for USART Rx.

(Only valid if prsRxEnable is true).

autoTx

```
bool USART_InitSync_TypeDef::autoTx
```

Enable AUTOTX mode.

Transmits as long as RX is not full. Generates underflows if TX is empty.

autoCsEnable

```
bool USART_InitSync_TypeDef::autoCsEnable
```

Auto CS enabling.

csInv

```
bool USART_InitSync_TypeDef::csInv
```

Enable CS invert.

By default, chip select is active low. Set to true to make chip select active high.

USART_InitIrDA_TypeDef

IrDA mode initialization structure.

Inherited from asynchronous mode initialization structure.

Public Attributes

USART_InitAsync_TypeDef	**async** General Asynchronous initialization structure.
bool	**irRxInv** Set to invert Rx signal before IrDA demodulator.
bool	**irFilt** Set to enable filter on IrDA demodulator.
USART_IrDAPw_Typedef	**irPw** Configure the pulse width generated by the IrDA modulator as a fraction of the configured USART bit period.
bool	**irPrsEn** Enable the PRS channel selected by irPrsSel as input to IrDA module instead of TX.
USART_PRS_Channel_t	**irPrsSel** PRS can be used as input to the pulse modulator instead of TX.

Public Attribute Documentation

async

```
USART_InitAsync_TypeDef USART_InitIrDA_TypeDef::async
```

General Asynchronous initialization structure.

irRxInv

```
bool USART_InitIrDA_TypeDef::irRxInv
```

Set to invert Rx signal before IrDA demodulator.

irFilt

```
bool USART_InitIrDA_TypeDef::irFilt
```

Set to enable filter on IrDA demodulator.

irPw

```
USART_IrDAPw_Typedef USART_InitIrDA_TypeDef::irPw
```

Configure the pulse width generated by the IrDA modulator as a fraction of the configured USART bit period.

irPrsEn

```
bool USART_InitIrDA_TypeDef::irPrsEn
```

Enable the PRS channel selected by irPrsSel as input to IrDA module instead of TX.

irPrsSel

```
USART_PRS_Channel_t USART_InitIrDA_TypeDef::irPrsSel
```

PRS can be used as input to the pulse modulator instead of TX.

This value selects the channel to use.

USART_InitI2s_TypeDef

I2S mode initialization structure.

Inherited from synchronous mode initialization structure.

Public Attributes

USART_InitSync_TypeDef	**sync** General Synchronous initialization structure.
USART_I2sFormat_TypeDef	**format** I2S mode.
bool	**delay** Delay on I2S data.
bool	**dmaSplit** Separate DMA Request For Left/Right Data.
USART_I2sJustify_TypeDef	**justify** Justification of I2S data within the frame.
bool	**mono** Stereo or Mono, set to true for mono.

Public Attribute Documentation

sync

```
USART_InitSync_TypeDef USART_InitI2s_TypeDef::sync
```

General Synchronous initialization structure.

format

```
USART_I2sFormat_TypeDef USART_InitI2s_TypeDef::format
```

I2S mode.

delay

```
bool USART_InitI2s_TypeDef::delay
```

Delay on I2S data.

Set to add a one-cycle delay between a transition on the word-clock and the start of the I2S word. Should be set for standard I2S format.

dmaSplit

```
bool USART_InitI2s_TypeDef::dmaSplit
```

Separate DMA Request For Left/Right Data.

justify

```
USART_I2sJustify_TypeDef USART_InitI2s_TypeDef::justify
```

Justification of I2S data within the frame.

mono

```
bool USART_InitI2s_TypeDef::mono
```

Stereo or Mono, set to true for mono.

VCMP - Voltage Comparator

VCMP - Voltage Comparator

Supply Voltage Comparator (VCMP) Peripheral API.

This module contains functions to control the VCMP peripheral of Silicon Labs 32-bit MCUs and SoCs. The VCMP monitors the input voltage supply and generates interrupts on events using as little as 100 nA.

Modules

VCMP_Init_TypeDef

Enumerations

enum VCMP_WarmTime_TypeDef {

vcmpWarmTime4Cycles = _VCMP_CTRL_WARMTIME_4CYCLES
vcmpWarmTime8Cycles = _VCMP_CTRL_WARMTIME_8CYCLES
vcmpWarmTime16Cycles = _VCMP_CTRL_WARMTIME_16CYCLES
vcmpWarmTime32Cycles = _VCMP_CTRL_WARMTIME_32CYCLES
vcmpWarmTime64Cycles = _VCMP_CTRL_WARMTIME_64CYCLES
vcmpWarmTime128Cycles = _VCMP_CTRL_WARMTIME_128CYCLES
vcmpWarmTime256Cycles = _VCMP_CTRL_WARMTIME_256CYCLES
vcmpWarmTime512Cycles = _VCMP_CTRL_WARMTIME_512CYCLES

}

Warm-up Time in High Frequency Peripheral Clock cycles.

enum VCMP_Hysteresis_TypeDef {

vcmpHystNone
vcmpHyst20mV

}

Hysteresis configuration.

Functions

void VCMP_Init(const VCMP_Init_TypeDef *vcmpInit)
Configure and enable Voltage Comparator.

void VCMP_LowPowerRefSet(bool enable)
Enable or disable Low Power Reference setting.

void VCMP_TriggerSet(int level)
Configure the trigger level of the voltage comparator.

void VCMP_Enable(void)
Enable Voltage Comparator.

void VCMP_Disable(void)
Disable Voltage Comparator.

uint32_t VCMP_VoltageToLevel(float v)
Calculate voltage to trigger level.

bool VCMP_VDDLower(void)
Check whether VDD is lower than the trigger level.

bool VCMP_VDDHigher(void)
Check whether VDD is higher than the trigger level.

bool VCMP_Ready(void)
Check whether the VCMP output is ready.

void VCMP_IntClear(uint32_t flags)
Clear one or more pending VCMP interrupts.

void VCMP_IntSet(uint32_t flags)
Set one or more pending VCMP interrupts from SW.

void VCMP_IntDisable(uint32_t flags)
Disable one or more VCMP interrupts.

void VCMP_IntEnable(uint32_t flags)
Enable one or more VCMP interrupts.

uint32_t VCMP_IntGet(void)
Get pending VCMP interrupt flags.

uint32_t VCMP_IntGetEnabled(void)
Get enabled and pending VCMP interrupt flags.

Macros

#define VCMP_INIT_DEFAULT undefined
Default VCMP initialization structure.

Enumeration Documentation

VCMP_WarmTime_TypeDef

> VCMP_WarmTime_TypeDef

Warm-up Time in High Frequency Peripheral Clock cycles.

Enumerator	
vcmpWarmTime4Cycles	4 cycles
vcmpWarmTime8Cycles	8 cycles
vcmpWarmTime16Cycles	16 cycles
vcmpWarmTime32Cycles	32 cycles
vcmpWarmTime64Cycles	64 cycles
vcmpWarmTime128Cycles	128 cycles
vcmpWarmTime256Cycles	256 cycles
vcmpWarmTime512Cycles	512 cycles

VCMP_Hysteresis_TypeDef

> VCMP_Hysteresis_TypeDef

Hysteresis configuration.

Enumerator	
vcmpHystNone	Normal operation, no hysteresis.
vcmpHyst20mV	Digital output will not toggle until positive edge is at least 20 mV above or below negative input voltage.

Function Documentation

VCMP_Init

```
void VCMP_Init (const VCMP_Init_TypeDef * vcmpInit)
```

Configure and enable Voltage Comparator.

Parameters

Type	Direction	Argument Name	Description
const VCMP_Init_TypeDef *	[in]	vcmpInit	The VCMP Initialization structure.

VCMP_LowPowerRefSet

```
void VCMP_LowPowerRefSet (bool enable)
```

Enable or disable Low Power Reference setting.

Parameters

Type	Direction	Argument Name	Description
bool	[in]	enable	If true, enables low power reference. If false, disable low power reference.

VCMP_TriggerSet

```
void VCMP_TriggerSet (int level)
```

Configure the trigger level of the voltage comparator.

Parameters

Type	Direction	Argument Name	Description
int	[in]	level	A trigger value, in range 0-63.

VCMP_Enable

```
void VCMP_Enable (void )
```

Enable Voltage Comparator.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

VCMP_Disable

```
void VCMP_Disable (void )
```

Disable Voltage Comparator.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

VCMP_VoltageToLevel

```
uint32_t VCMP_VoltageToLevel (float v)
```

Calculate voltage to trigger level.

Parameters

Type	Direction	Argument Name	Description
float	[in]	v	Voltage Level for trigger.

Note

- Soft float support is needed for this function to work.

Returns

- Trigger level.

VCMP_VDDLower

```
bool VCMP_VDDLower (void )
```

Check whether VDD is lower than the trigger level.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if Voltage Comparator indicated VDD < trigger level, else returns false.

VCMP_VDDHigher

```
bool VCMP_VDDHigher (void )
```

Check whether VDD is higher than the trigger level.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if Voltage Comparator indicated VDD > trigger level, else returns false.

VCMP_Ready

```
bool VCMP_Ready (void )
```

Check whether the VCMP output is ready.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Returns

- True if VCMP output is ready.

VCMP_IntClear

```
void VCMP_IntClear (uint32_t flags)
```

Clear one or more pending VCMP interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	VCMP interrupt sources to clear. Use a set of interrupt flags OR-ed together to clear multiple interrupt sources for the VCMP module (VCMP_IFS_nnn).

VCMP_IntSet

```
void VCMP_IntSet (uint32_t flags)
```

Set one or more pending VCMP interrupts from SW.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	VCMP interrupt sources to set to pending. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the VCMP module (VCMP_IFS_nnn).

VCMP_IntDisable

```
void VCMP_IntDisable (uint32_t flags)
```

Disable one or more VCMP interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	VCMP interrupt sources to enable. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the VCMP module (VCMP_IFS_nnn).

VCMP_IntEnable

```
void VCMP_IntEnable (uint32_t flags)
```

Enable one or more VCMP interrupts.

Parameters

Type	Direction	Argument Name	Description
uint32_t	[in]	flags	VCMP interrupt sources to enable. Use a set of interrupt flags OR-ed together to set multiple interrupt sources for the VCMP module (VCMP_IFS_nnn).

VCMP_IntGet

```
uint32_t VCMP_IntGet (void )
```

Get pending VCMP interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Note

- The event bits are not cleared by the use of this function.

Returns

- Pending VCMP interrupt sources. Returns a set of interrupt flags OR-ed together for multiple interrupt sources in the VCMP module (VCMP_IFS_nnn).

VCMP_IntGetEnabled

```
uint32_t VCMP_IntGetEnabled (void )
```

Get enabled and pending VCMP interrupt flags.

Parameters

Type	Direction	Argument Name	Description
void	N/A		

Useful for handling more interrupt sources in the same interrupt handler.

Note

- The event bits are not cleared by the use of this function.

Returns

- Pending and enabled VCMP interrupt sources. The return value is the bitwise AND combination of
  - the OR combination of enabled interrupt sources in VCMP_IEN_nnn register (VCMP_IEN_nnn) and
  - the OR combination of valid interrupt flags of the VCMP module (VCMP_IF_nnn).

Macro Definition Documentation

VCMP_INIT_DEFAULT

```
#define VCMP_INIT_DEFAULT
```

Value:

```
{                                                                   \
  true,              /** Half Bias enabled. */                      \
  0x7,               /** Bias current 0.7 uA when half bias is enabled. */ \
  false,             /** Falling edge sense not enabled. */         \
  false,             /** Rising edge sense not enabled. */          \
  vcmpWarmTime4Cycles, /** 4 clock cycles warm-up time. */          \
  vcmpHystNone,      /** No hysteresis. */                          \
  0,                 /** 0 in digital output when inactive. */      \
  true,              /** Do not use low power reference. */         \
  39,                /** Trigger level just below 3V. */            \
  true,              /** Enable after initialization. */            \
}
```

Default VCMP initialization structure.

VCMP_Init_TypeDef

VCMP Initialization structure.

Public Attributes

bool	**halfBias** If set to true, reduces the bias current by half.
int	**biasProg** BIAS current configuration, depends on Half Bias setting above, see reference manual for more information.
bool	**irqFalling** Enable interrupt for falling edge.
bool	**irqRising** Enable interrupt for rising edge.
VCMP_WarmTime_TypeDef	**warmup** Warm-up time in clock cycles.
VCMP_Hysteresis_TypeDef	**hyst** Hysteresis configuration.
int	**inactive** Output value when comparator is inactive, should be 0 or 1.
bool	**lowPowerRef** Enable low power mode for VDD and band gap reference.
int	**triggerLevel** Trigger level, according to formula: VDD Trigger Level = 1.667V + 0.034V x triggerLevel.
bool	**enable** Enable VCMP after configuration.

Public Attribute Documentation

halfBias

```
bool VCMP_Init_TypeDef::halfBias
```

If set to true, reduces the bias current by half.

biasProg

```
int VCMP_Init_TypeDef::biasProg
```

BIAS current configuration, depends on Half Bias setting above, see reference manual for more information.

irqFalling

```
bool VCMP_Init_TypeDef::irqFalling
```

Enable interrupt for falling edge.

irqRising

```
bool VCMP_Init_TypeDef::irqRising
```

Enable interrupt for rising edge.

warmup

```
VCMP_WarmTime_TypeDef VCMP_Init_TypeDef::warmup
```

Warm-up time in clock cycles.

hyst

```
VCMP_Hysteresis_TypeDef VCMP_Init_TypeDef::hyst
```

Hysteresis configuration.

inactive

```
int VCMP_Init_TypeDef::inactive
```

Output value when comparator is inactive, should be 0 or 1.

lowPowerRef

```
bool VCMP_Init_TypeDef::lowPowerRef
```

Enable low power mode for VDD and band gap reference.

triggerLevel

```
int VCMP_Init_TypeDef::triggerLevel
```

Trigger level, according to formula: VDD Trigger Level = 1.667V + 0.034V x triggerLevel.

enable

```
bool VCMP_Init_TypeDef::enable
```

Enable VCMP after configuration.

VERSION - Version Defines

VERSION - Version Defines

Version API.

Macros specifying the EMLIB and CMSIS version.

Macros

#define _CMSIS_VERSION 5.8.0
Version number of targeted CMSIS package.

#define _CMSIS_VERSION_MAJOR 5
Major version of CMSIS.

#define _CMSIS_VERSION_MINOR 8
Minor version of CMSIS.

#define _CMSIS_VERSION_PATCH 0
Patch revision of CMSIS.

Macro Definition Documentation

_CMSIS_VERSION

```
#define _CMSIS_VERSION
```

Value:

```
5.8.0
```

Version number of targeted CMSIS package.

_CMSIS_VERSION_MAJOR

```
#define _CMSIS_VERSION_MAJOR
```

Value:

```
5
```

Major version of CMSIS.

_CMSIS_VERSION_MINOR

```
#define _CMSIS_VERSION_MINOR
```

Value:

```
8
```

Minor version of CMSIS.

_CMSIS_VERSION_PATCH

```
#define _CMSIS_VERSION_PATCH
```

Value:

```
0
```

Patch revision of CMSIS.

WDOG - Watchdog

WDOG - Watchdog

Watchdog (WDOG) Peripheral API.

This module contains functions to control the WDOG peripheral of Silicon Labs 32-bit MCUs and SoCs. The WDOG resets the system in case of a fault condition.

Modules

WDOG_Init_TypeDef

Enumerations

enum WDOG_ClkSel_TypeDef {

wdogClkSelULFRCO = _WDOG_CTRL_CLKSEL_ULFRCO
wdogClkSelLFRCO = _WDOG_CTRL_CLKSEL_LFRCO
wdogClkSelLFXO = _WDOG_CTRL_CLKSEL_LFXO

}
Watchdog clock selection.

enum WDOG_PeriodSel_TypeDef {

wdogPeriod_9 = 0x0
wdogPeriod_17 = 0x1
wdogPeriod_33 = 0x2
wdogPeriod_65 = 0x3
wdogPeriod_129 = 0x4
wdogPeriod_257 = 0x5
wdogPeriod_513 = 0x6
wdogPeriod_1k = 0x7
wdogPeriod_2k = 0x8
wdogPeriod_4k = 0x9
wdogPeriod_8k = 0xA
wdogPeriod_16k = 0xB
wdogPeriod_32k = 0xC
wdogPeriod_64k = 0xD
wdogPeriod_128k = 0xE
wdogPeriod_256k = 0xF

}
Watchdog period selection.

Functions

void WDOGn_Enable(WDOG_TypeDef *wdog, bool enable)
Enable/disable the watchdog timer.

void WDOGn_Feed(WDOG_TypeDef *wdog)
Feed WDOG.

void WDOGn_Init(WDOG_TypeDef *wdog, const WDOG_Init_TypeDef *init)
Initialize WDOG (assuming the WDOG configuration has not been locked).

void WDOGn_Lock(WDOG_TypeDef *wdog)
Lock the WDOG configuration.

void WDOGn_SyncWait(WDOG_TypeDef *wdog)
Wait for the WDOG to complete all synchronization of register changes and commands.

void WDOGn_Unlock(WDOG_TypeDef *wdog)
Unlock the WDOG configuration.

bool WDOGn_IsEnabled(WDOG_TypeDef *wdog)

Get enabled status of the Watchdog.

bool WDOGn_IsLocked(WDOG_TypeDef *wdog)
Get locked status of the Watchdog.

Macros

#define DEFAULT_WDOG WDOG
Default WDOG instance for deprecated functions.

#define WDOG_INIT_DEFAULT undefined
Suggested default configuration for WDOG initialization structure.

#define WDOG_SYNC_TIMEOUT 30000
In some scenarioes when the watchdog is disabled the synchronization register might be set and not be cleared until the watchdog is enabled again.

Enumeration Documentation

WDOG_ClkSel_TypeDef

> WDOG_ClkSel_TypeDef

Watchdog clock selection.

Enumerator	
wdogClkSelULFRCO	Ultra low frequency (1 kHz) clock.
wdogClkSelLFRCO	Low frequency RC oscillator.
wdogClkSelLFXO	Low frequency crystal oscillator.

WDOG_PeriodSel_TypeDef

> WDOG_PeriodSel_TypeDef

Watchdog period selection.

Enumerator	
wdogPeriod_9	9 clock periods
wdogPeriod_17	17 clock periods
wdogPeriod_33	33 clock periods
wdogPeriod_65	65 clock periods
wdogPeriod_129	129 clock periods
wdogPeriod_257	257 clock periods
wdogPeriod_513	513 clock periods
wdogPeriod_1k	1025 clock periods
wdogPeriod_2k	2049 clock periods
wdogPeriod_4k	4097 clock periods
wdogPeriod_8k	8193 clock periods
wdogPeriod_16k	16385 clock periods
wdogPeriod_32k	32769 clock periods
wdogPeriod_64k	65537 clock periods
wdogPeriod_128k	131073 clock periods

wdogPeriod_256k	262145 clock periods

Function Documentation

WDOGn_Enable

```
void WDOGn_Enable (WDOG_TypeDef * wdog, bool enable)
```

Enable/disable the watchdog timer.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	A pointer to the WDOG peripheral register block.
bool	[in]	enable	True to enable Watchdog, false to disable. Watchdog cannot be disabled if it's been locked.

Note

- This function modifies the WDOG CTRL register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed.

WDOGn_Feed

```
void WDOGn_Feed (WDOG_TypeDef * wdog)
```

Feed WDOG.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	A pointer to the WDOG peripheral register block.

When WDOG is activated, it must be fed (i.e., clearing the counter) before it reaches the defined timeout period. Otherwise, WDOG will generate a reset.

Note

- Note that WDOG is an asynchronous peripheral and when calling the WDOGn_Feed() function the hardware starts the process of clearing the counter. This process takes some time before it completes depending on the selected oscillator (up to 4 peripheral clock cycles). When using the ULFRCO for instance as the oscillator the watchdog runs on a 1 kHz clock and a watchdog clear operation might take up to 4 ms.

If the device enters EM2 or EM3 while a command is in progress then that command will be aborted. An application can use WDOGn_SyncWait() to wait for a command to complete.

WDOGn_Init

```
void WDOGn_Init (WDOG_TypeDef * wdog, const WDOG_Init_TypeDef * init)
```

Initialize WDOG (assuming the WDOG configuration has not been locked).

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	Pointer to the WDOG peripheral register block.
const WDOG_Init_TypeDef *	[in]	init	The structure holding the WDOG configuration. A default setting WDOG_INIT_DEFAULT is available for initialization.

Note

- This function modifies the WDOG CTRL register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed.

WDOGn_Lock

```
void WDOGn_Lock (WDOG_TypeDef * wdog)
```

Lock the WDOG configuration.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	A pointer to WDOG peripheral register block.

This prevents errors from overwriting the WDOG configuration, possibly disabling it. Only a reset can unlock the WDOG configuration once locked.

If the LFRCO or LFXO clocks are used to clock WDOG, consider using the option of inhibiting those clocks to be disabled. See the WDOG_Enable() initialization structure.

Note

- This function modifies the WDOG CTRL register which requires synchronization into the low-frequency domain. If this register is modified before a previous update to the same register has completed, this function will stall until the previous synchronization has completed.

WDOGn_SyncWait

```
void WDOGn_SyncWait (WDOG_TypeDef * wdog)
```

Wait for the WDOG to complete all synchronization of register changes and commands.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	A pointer to WDOG peripheral register block.

WDOGn_Unlock

```
void WDOGn_Unlock (WDOG_TypeDef * wdog)
```

Unlock the WDOG configuration.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	A pointer to WDOG peripheral register block.

Note that this function will have no effect on devices where a reset is the only way to unlock the watchdog.

WDOGn_IsEnabled

```
bool WDOGn_IsEnabled (WDOG_TypeDef * wdog)
```

Get enabled status of the Watchdog.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	Pointer to the WDOG peripheral register block.

Returns

- True if Watchdog is enabled.

WDOGn_IsLocked

```
bool WDOGn_IsLocked (WDOG_TypeDef * wdog)
```

Get locked status of the Watchdog.

Parameters

Type	Direction	Argument Name	Description
WDOG_TypeDef *	[in]	wdog	Pointer to the WDOG peripheral register block.

Returns

- True if Watchdog is locked.

Macro Definition Documentation

DEFAULT_WDOG

```
#define DEFAULT_WDOG
```

Value:

```
WDOG
```

Default WDOG instance for deprecated functions.

WDOG_INIT_DEFAULT

```
#define WDOG_INIT_DEFAULT
```

Value:

```
0 |  {                                                                \
```

```
    true,                   /* Start Watchdog when initialization is done. */ \
    false,                  /* WDOG is not counting during debug halt. */     \
    false,                  /* WDOG is not counting when in EM2. */           \
    false,                  /* WDOG is not counting when in EM3. */           \
    false,                  /* EM4 can be entered. */                         \
    false,                  /* Do not block disabling LFRCO/LFXO in CMU. */   \
    false,                  /* Do not lock WDOG configuration. */             \
    wdogClkSelLFRCO,        /* Select 32.768 kHz WDOG oscillator. */          \
    wdogPeriod_256k         /* Set longest possible timeout period. */        \
  }
```

Suggested default configuration for WDOG initialization structure.

WDOG_SYNC_TIMEOUT

```
#define WDOG_SYNC_TIMEOUT
```

Value:

```
30000
```

In some scenarioes when the watchdog is disabled the synchronization register might be set and not be cleared until the watchdog is enabled again.

This will happen when for instance some watchdog register is modified while the watchdog clock is disabled. In these scenarioes we need to make sure that the software does not wait forever.

WDOG_Init_TypeDef

Watchdog initialization structure.

Public Attributes

bool	enable Enable Watchdog when initialization completed.
bool	debugRun Counter keeps running during debug halt.
bool	em2Run Counter keeps running when in EM2.
bool	em3Run Counter keeps running when in EM3.
bool	em4Block Block EMU from entering EM4.
bool	swoscBlock Block SW from disabling LFRCO/LFXO oscillators.
bool	lock Block SW from modifying the configuration (a reset is needed to reconfigure).
WDOG_ClkSel_TypeDef	clkSel Clock source to use for Watchdog.
WDOG_PeriodSel_TypeDef	perSel Watchdog timeout period.

Public Attribute Documentation

enable

```
bool WDOG_Init_TypeDef::enable
```

Enable Watchdog when initialization completed.

debugRun

```
bool WDOG_Init_TypeDef::debugRun
```

Counter keeps running during debug halt.

em2Run

```
bool WDOG_Init_TypeDef::em2Run
```

Counter keeps running when in EM2.

em3Run

bool WDOG_Init_TypeDef::em3Run

Counter keeps running when in EM3.

em4Block

bool WDOG_Init_TypeDef::em4Block

Block EMU from entering EM4.

swoscBlock

bool WDOG_Init_TypeDef::swoscBlock

Block SW from disabling LFRCO/LFXO oscillators.

lock

bool WDOG_Init_TypeDef::lock

Block SW from modifying the configuration (a reset is needed to reconfigure).

clkSel

WDOG_ClkSel_TypeDef WDOG_Init_TypeDef::clkSel

Clock source to use for Watchdog.

perSel

WDOG_PeriodSel_TypeDef WDOG_Init_TypeDef::perSel

Watchdog timeout period.

EZR32WG

API Documentation

List of modules	Description
ACMP - Analog Comparator	Analog comparator (ACMP) Peripheral API.
ADC - Analog to Digital Converter	Analog to Digital Converter Analog to Digital Converter (ADC) Peripheral API.
AES - AES Accelerator	Advanced Encryption Standard Accelerator (AES) Peripheral API.
BURTC - Backup RTC	Backup Real Time Counter (BURTC) Peripheral API.
BUS - Bitfield Read/Write	BUS register and RAM bit/field read/write API.
CHIP - Chip Errata Workarounds	Chip errata workaround APIs.
CMU - Clock Management Unit	Clock management unit (CMU) Peripheral API.
CORE - Core Interrupt	Core interrupt handling API.
DAC - Digital to Analog Converter	Digital to Analog Converter (DAC) Peripheral API.
DBG - Debug	Debug (DBG) Peripheral API.
DMA - Direct Memory Access	Direct Memory Access (DMA) Peripheral API.
EBI - External Bus Interface	EBI External Bus Interface (EBI) Peripheral API.
EMU - Energy Management Unit	Energy Management Unit (EMU) Peripheral API.
GPIO - General Purpose Input/Output	General Purpose Input/Output (GPIO) API.
I2C - Inter-Integrated Circuit	Inter-integrated Circuit (I2C) Peripheral API.
LESENSE - Low Energy Sensor	Low Energy Sensor (LESENSE) Peripheral API.
LETIMER - Low Energy Timer	Low Energy Timer (LETIMER) Peripheral API.
LEUART - Low Energy UART	Low Energy Universal Asynchronous Receiver/Transmitter (LEUART) Peripheral API.
MSC - Memory System Controller	Memory System Controller API.
OPAMP - Operational Amplifier	Operational Amplifier (OPAMP) peripheral API.
PCNT - Pulse Counter	Pulse Counter (PCNT) Peripheral API.
PRS - Peripheral Reflex System	Peripheral Reflex System (PRS) Peripheral API.
RAMFUNC - RAM Function Support	RAM code support.
RMU - Reset Management Unit	Reset Management Unit (RMU) Peripheral API.
RTC - Real Time Counter	Real Time Counter (RTC) Peripheral API.
SYSTEM - System Utils	System API.
TIMER - Timer/Counter	Timer/Counter (TIMER) Peripheral API.
USART - Synchronous/Asynchronous Serial	Universal Synchronous/Asynchronous Receiver/Transmitter Peripheral API.
VCMP - Voltage Comparator	Supply Voltage Comparator (VCMP) Peripheral API.
VERSION - Version Defines	Version API.
WDOG - Watchdog	Watchdog (WDOG) Peripheral API.