API-Reference for **Bus Interface Modules M130, M131, M132 and M135**
KNX-Processors 184/01, 184/11 and 184/21

Content of this document:

1 Introduction 3
2 API-Reference 3
2.1 Debouncing 3
2.1.1 Function DebounceInit 3
2.1.2 Function Debounce 4
2.2 Object-Handling 5
2.2.1 Function TestAndCopyObject 5
2.2.2 Function SetAndTransmitObject 6
2.2.3 Function TestObject 6
2.2.4 Function TransmitObject 7
2.2.5 Function ReadObject 7
2.2.6 Function SetRAMFlags 8
2.3 Timer 9
2.3.1 Function GetSystemTime 9
2.3.2 Function TmInit 10
2.3.3 Function TmStart 10
2.3.4 Function TmAddStart 11
2.3.5 Function TmIsExpired 11
2.3.6 Function TmIsRunning 12
2.3.7 Function TmStop 12
2.4 AD converter 13
2.4.1 Function ADCInit 13
2.4.2 Function ADCShutdown 13
2.4.3 Function ADCRead 14
2.4.4 Function ADCStop 14
2.4.5 Function ADCIsInterrupted 15
2.4.6 Function ADCResetInterrupted 15
2.4.7 Function CalcPEIType 16
2.5 Pulse width modulation 17
2.5.1 Function PWMInit 17
2.5.2 Function PWMStop 17
2.5.3 Function PWMSetValue 18

2.6 FT12 19
2.6.1 Function FT12Init 19
2.6.2 Function FT12Send 20
2.6.3 Function FT12Get 20
2.7 Handshake 21
2.7.1 Function HSInit 21
2.7.2 Function HSSetFrame 22
2.7.3 Function HSGetFrame 23
2.8 SPI 24
2.8.1 Function SPIInit 24
2.8.2 Function SPISend 24
2.9 Flash 25
2.9.1 Function FlashSegErase 25
2.9.2 Function FlashWrite 26
2.9.3 Function FlashIsBlockBlank 27
2.10 Parameter 28
2.10.1 Function ParamInitVal 28
2.10.2 Function ParamReadVal 29
2.11 Message 30
2.11.1 Function MsgCreate 30
2.11.2 Function MsgDiscard 30
2.11.3 Function MsgGet 31
2.11.4 Function MsgPost 31
2.11.5 Function MsgUndoGet 32
2.11.6 Function MsgSwitchQueue 32
2.11.7 Function MsgResetDynQueues 33
2.12 Utility 34
2.12.1 Function IsApplicationLoaded 34
2.12.2 Function GetPhysAddr 34
2.12.3 Function GetSerialNumber 35
2.12.4 Function ReadBCU2Adr100 35
2.12.5 Function TriggerWatchDog 36
2.13 Interrupts 37
2.13.1 Function IntRegister 37
2.13.2 Function IntUnregister 38
2.13.3 Function IntResetAll 38

Document-Version: 1.3

1 Introduction

This document describes the application programming interface for the
Siemens Bus Interface Modules BIM M130, BIM M131, BIM M132 and BIM M135
as well as for the
Siemens KNX-Processors 184/01, 184/11 and 184/21.
To call any of the following functions write U._<function name>.
Example: if (U._TestObject(3) == TRUE) {…}

2 API-Reference

2.1 Debouncing

The following functions are used to debounce input pins of the BIM in software.

2.1.1 Function DebounceInit

Prototype:

```
void DebounceInit(DEBOUNCEKIT* kit, USHORT initvalue)
```

Description:

This function initializes the structure DEBOUCEKIT. This is necessary before you can call ‘Debounce’ for this DEBOUCEKIT.

Parameters:

`DEBOUNCEKIT* kit:`
A pointer to a structure of type DEBOUNCEKIT

`USHORT initvalue:`
The initial value of the debounced value in structure DEBOUNCEKIT

Return values:

none

Callable in / at:

init, main

Stack used: 18 bytes

Comment:

2.1.2 Function Debounce

Prototype:

```
void Debounce(USHORT sample, USHORT mask, DEBOUNCEKIT* kit,
USHORT debouncetime)
```

Description:

This function is used to debounce the result of the logical AND of ‘sample’ and ‘mask’. If the result has been unchanged for at least ‘debouncetime’, the new value is stored in ‘kit’.

Parameters:

`USHORT sample:`
The actual value that should be debounced

`USHORT mask:`
Specifies the bits in sample that should be debounced

`DEBOUNCEKIT* kit:`
A pointer to a structure of type DEBOUNCEKIT

`USHORT debouncetime:`
The minimum number of ticks the value must remain unchanged. One tick is equal to $416.\overline{6}\,\mu s$.

Return values:

none

Callable in / at:

main

Stack used: 22 bytes

Comment:

The mask must always be the same for a specific DEBOUNCEKIT.
The maximum debounce time is 10 seconds (24,000 ticks).

2.2 Object-Handling

The following functions are used to get and set flags on communication objects.

2.2.1 Function TestAndCopyObject

Prototype:

```
BOOL TestAndCopyObject(USHORT objectNr, void* dst, BYTE len)
```

Description:

This function tests if there was an update for an object given by ‘objectNr’. If this is the case ‘len’ bytes of the object value are copied to ‘dst’.

Parameters:

`USHORT objectNr:`
The object number

`void* dst:`
A pointer to RAM where the object data should be copied

`BYTE len:`
The number of bytes that should be copied from the object data to ‘dst’

Return values:

true: there was an update and the object data was copied to 'dst’
false: there was no update

Callable in / at:

interrupts, main

Stack used: 24 bytes

Comment:

This function handles the lock of the task switch in the critical sections automatically.

2.2.2 Function SetAndTransmitObject

Prototype:

```
BOOL SetAndTransmitObject(USHORT objectNr, void* src, BYTE len)
```

Description:

Copies ‘len’ bytes from ‘src’ to the object data and sets the flags to transmit the object specified by ‘objectNr’.

Parameters:

`USHORT objectNr:`
The object number

`void* src:`
A pointer to the new object value

`BYTE len:`
The number of bytes that should be copied to the object data before transmitting

Return values:

true: the data was copied and the transmit flags are set
false: the operation could not be performed; that for example could be the case if the specified object is still transmitting

Callable in / at:

interrupts, main

Stack used: 22 bytes

Comment:

This function handles the lock of the task switch in the critical sections automatically.

2.2.3 Function TestObject

Prototype:

```
BOOL TestObject(USHORT objectNr)
```

Description:

Tests if there was an update for the object specified by ‘objectNr’.

Parameters:

`USHORT objectNr:`
The object number

Return values:

true: there was an update
false: there was no update

Callable in / at:

interrupts, main

Stack used: 22 bytes

Comment:

2.2.4 Function TransmitObject

Prototype:

```
BOOL TransmitObject(USHORT objectNr)
```

Description:

This function sets the RAM flags to transmit the object specified by ‘objectNr’.

Parameters:

`USHORT objectNr:`

The object number

Return values:

true: the RAM flags were set successfully
false: the RAM flags were not set; that could be the case if the object is still transmitting

Callable in / at:

interrupts, main

Stack used: 20 bytes

Comment:

2.2.5 Function ReadObject

Prototype:

```
BOOL ReadObject(USHORT objectNr)
```

Description:

Sets the RAM flags to generate a GroupValueRead on the object specified by ‘objectNr’.

Parameters:

`USHORT objectNr:`

The object number

Return values:

true: the RAM flags were set successfully
false: the RAM flags were not set; that could be the case if the object is still transmitting

Callable in / at:

interrupts, main

Stack used: 14 bytes

Comment:

2.2.6 Function SetRAMFlags

Prototype:

```
BYTE SetRAMFlags(BYTE objectNr, BYTE flags)
```

Description:

Set the RAM flags for the object specified by 'objectNr'. The high nibble of 'flags' acts as mask where a logical one indicates that these flags should be modified for the object specified by 'objectNr'. The low nibble of 'flags' must contain the value for the flags that should be modified. The low nibble of the return value is the actual value of the RAM flags for this object.
To do a simple read of the RAM flags for one object, all bits in the high nibble of 'flags' must be zero.

Parameters:

`BYTE objectNr:`
The object number

`BYTE flags:`
The high nibble must contain the mask that specifies which flags should be modified; the low nibble must contain the new values for the flags

Return values:

The flags for the specified object; stored in the low nibble of return value

Callable in / at:

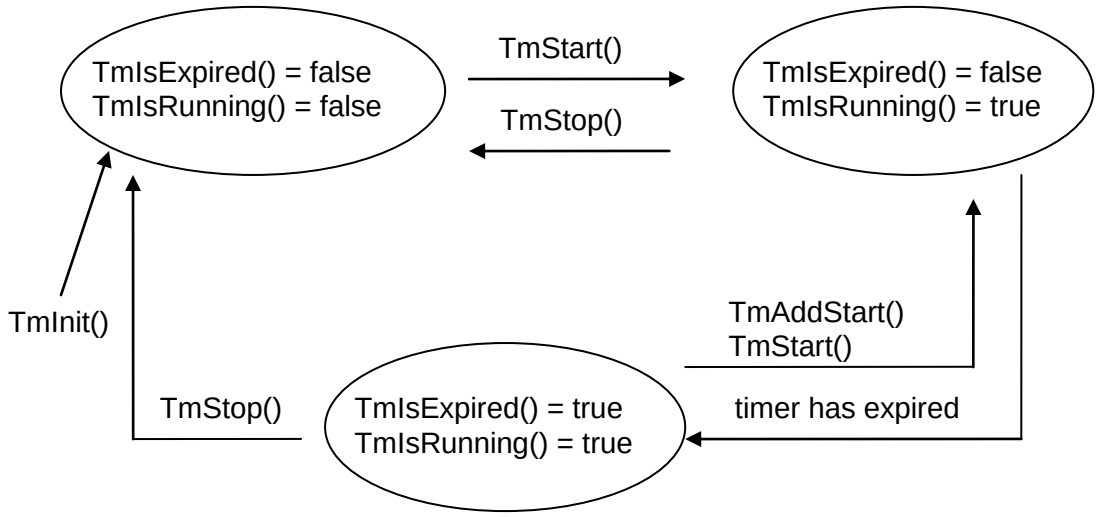
interrupts, main

Stack used: 12 bytes

Comment:

2.3 Timer

The following functions can be used for working with time and timers managed by the BIM operation system. The following figure shows the usage of the timer functions:



2.3.1 Function GetSystemTime

Prototype:

```
ULONG GetSystemTime(void)
```

Description:

This function returns a 4 byte value that contains the ticks that are expired since the last start up. One tick is equal to $416.\overline{6}\,\mu s$.

Parameters:

none

Return values:

The 4 byte value that contains the ticks

Callable in / at:

init, main

Stack used: 12 bytes

Comment:

2.3.2 Function TmInit

Prototype:

```
void TmInit (BYTE NumOfTimers)
```

Description:

This function initializes the timer table specified in the application info block. It sets the number of timers and sets each timer to expired and stopped.

Parameters:

`BYTE NumOfTimers:`

The number of timers that are used in the application program

Return values:

none

Callable in / at:

init

Stack used: 8 bytes

Comment:

This function must be called before any other timer function is called.

2.3.3 Function TmStart

Prototype:

```
void TmStart(TIMER* pTimer, ULONG ticks)
```

Description:

Starts a timer specified by the pointer ‘pTimer’. The timer will be expired after ‘ticks’ measured from the point where ‘TmStart’ was called. Use ‘TmIsExpired’ to test whether a timer is expired or not.

Parameters:

`TIMER* pTimer:`

A pointer to a timer that should be started

`ULONG ticks:`

The number of ticks the timer should run before it will be expired

Return values:

none

Callable in / at:

init, main

Stack used: 18 bytes

Comment:

2.3.4 Function TmAddStart

Prototype:

```
void TmAddStart(TIMER* pTimer, ULONG ticks)
```

Description:

This function restarts a timer for the specified 'ticks' measured from the last expiration of the timer. Use this function if you want to get timer intervals that do not drift away from the real time like it would be if you use 'TmStart'.

Parameters:

`TIMER* pTimer:`

A pointer to a timer that should be started

`ULONG ticks:`

The number of ticks the timer should run before it will be expired

Return values:

none

Callable in / at:

main

Stack used: 18 bytes

Comment:

Before a call of 'TmAddStart' there must be at least one call of 'TmStart'

2.3.5 Function TmIsExpired

Prototype:

```
BOOL TmIsExpired(TIMER* pTimer)
```

Description:

Tests if the timer specified by the pointer 'pTimer' is expired or not. The timer remains expired until the next call of 'TmStart', 'TmAddStart' or 'TmStop'.

Parameters:

`TIMER* pTimer:`

A pointer to a timer that should be tested whether it is expired or not

Return values:

true: the timer is expired
false: the timer is not expired

Callable in / at:

interrupts, main, save

Stack used: 8 bytes

Comment:

2.3.6 Function TmIsRunning

Prototype:

```
BOOL TmIsRunning(TIMER* pTimer)
```

Description:

Tests if a timer is running. A timer is running if it is not expired and not stopped.

Parameters:

`TIMER* pTimer:`

A pointer to a timer that should be tested whether it is running or not

Return values:

true: the timer is running
false: the timer is not running

Callable in / at:

interrupts, main, save

Stack used: 8 bytes

Comment:

2.3.7 Function TmStop

Prototype:

```
void TmStop(TIMER* pTimer)
```

Description:

Stops a timer given by the pointer ‘pTimer’.

Parameters:

`TIMER* pTimer:`

A pointer to a timer that should be stopped

Return values:

none

Callable in / at:

interrupts, main

Stack used: 8 bytes

Comment:

2.4 AD converter

The following functions are used to control the AD converter of the BIM.

2.4.1 Function ADCInit

Prototype:

```
void ADCInit(ADCSpeedModeType speed)
```

Description:

Inits the AD converter and sets the speed specified by the parameter ‘speed’.

Parameters:

`ADCSpeedModeType speed:`

The conversion speed of the AD converter

Return values:

none

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.4.2 Function ADCShutdown

Prototype:

```
void ADCShutdown(void)
```

Description:

Disables the AD converter.

Parameters:

none

Return values:

none

Callable in / at:

interrupts, init, main, save, unload

Stack used: 8 bytes

Comment:

After ADCShutdown a new call of ‘ADCInit’ is necessary before ‘ADCRead’ can be called again.

2.4.3 Function ADCRead

Prototype:

```
USHORT ADCRead(BYTE port, BYTE ADCcount)
```

Description:

Returns the accumulated AD converter value of the specified ‘port’; divide the return value through ‘ADCcount’ to get the average value.

Parameters:

`BYTE port:`

The ad converter channel that should be converted

`BYTE ADCcount:`

The number of conversions that should be accumulated

Return values:

The sum of the values of all read operations

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.4.4 Function ADCStop

Prototype:

```
void ADCStop(void)
```

Description:

Stops the AD converter.

Parameters:

none

Return values:

none

Callable in / at:

interrupts, main

Stack used: 8 bytes

Comment:

After a call of ‘ADCStop’ you can call ‘ADCRead’ without a new call of ‘ADCInit’.

2.4.5 Function ADCIsInterrupted

Prototype:

```
BOOL ADCIsInterrupted(void)
```

Description:

If you manually control the AD converter you have to call this function, because the system is able to stop the ad converter to save current (this is done if a flash write operation is necessary). If 'ADCGetInterrupted' returns 'true' call 'ADCResetInterrupted' and do the AD conversion again.

Parameters:

none

Return values:

true: last AD conversion was interrupted
false: last AD conversion ends without interruption

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.4.6 Function ADCResetInterrupted

Prototype:

```
void ADCResetInterrupted(void)
```

Description:

Resets the flag that indicates that an AD conversion was stopped by the system

Parameters:

none

Return values:

none

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.4.7 Function CalcPEIType

Prototype:

```
BYTE CalcPEIType(BYTE adval)
```

Description:

This function can be used to calculate the PEI type that corresponds to the given AD converter value.

Parameters:

`BYTE adval:`

The measured AD converter value

Return values:

none

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.5 Pulse width modulation

The following functions are used to control to pulse width modulation output of the BIM.

2.5.1 Function PWMInit

Prototype:

```
void PWMInit(PWMChannelType channel, PWMPolType mode,
PWMSpeedModeType speed)
```

Description:

Inits the pulse width modulation output for the specified channel. Select the polarity via ‘mode’ and a speed via ‘speed’.

Parameters:

`PWMChannelType channel:`
: One of the two available pwm channels

`PWMPolType mode:`
: The polarity of the generated pwm signal

`PWMSpeedModeType speed:`
: The speed of the pwm signal

Return values:
: none

Callable in / at:
: interrupts, init, main, save

Stack used: 8 bytes

Comment:
: ---

2.5.2 Function PWMStop

Prototype:

```
void PWMStop(PWMChannelType channel)
```

Description:
: This function stops the pulse width modulation for the specified channel.

Parameters:

`PWMChannelType channel:`
: The pwm channel that should be stopped

Return values:
: none

Callable in / at:
: interrupts, init, main, save, unload

Stack used: 8 bytes

Comment:
: ---

2.5.3 Function PWMSetValue

Prototype:

```
void PWMSetValue(PWMChannelType channel, BYTE value)
```

Description:

This function sets a new value to the pulse width modulation for the specified channel.

Parameters:

`PWMChannelType channel:`

The pwm channel whose value should be updated

`BYTE value:`

The new value for the selected pwm channel

Return values:

none

Callable in / at:

interrupts, init, main, save

Stack used: 8 bytes

Comment:

2.6 FT12

The following functions are used to send and receive FT12 frames over UART0 of the bim.

2.6.1 Function FT12Init

Prototype:

```
void FT12Init(USHORT TimeoutTime, BYTE len, BYTE* rcvBuffer,
BYTE* trmBuffer, BYTE baud, BYTE config)
```

Description:

This function is used to initialize the driver for sending and receiving FT12 frames over UART0. It must be called before any other function of FT12 is called.

Parameters:

`USHORT TimeoutTime:`
The time out time for the FT12 acknowledge

`BYTE len:`
The size of the transmit and receive buffer

`BYTE* rcvBuffer:`
A pointer to a receive buffer

`BYTE* trmBuffer:`
A pointer to a transmit buffer

`BYTE baud:`
The baud rate; the possible values can be looked up in the NEC78K0 / KE2 datasheet

`BYTE config:`
The configuration of the FT12 UART; the possible configurations can be looked up in the NEC78K0 / KE2 datasheet

Return values:

none

Callable in / at:

init

Stack used: 18 bytes

Comment:

2.6.2 Function FT12Send

Prototype:

```
BOOL FT12Send(BYTE* src, BYTE len, BYTE* result)
```

Description:

Copies ‘len’ bytes from ‘src’ to the send buffer if it is empty. After sending the result is stored in ‘result’.

Parameters:

`BYTE* src:`
A pointer to data that should be transmitted

`BYTE len:`
The number of bytes that should be transmitted

`BYTE* result:`
A pointer to a byte where the result of the FT12 transmit will be stored

Return values:

true: the data was copied to send buffer
false: the send buffer was not free, so no data has been copied

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.6.3 Function FT12Get

Prototype:

```
BOOL FT12Get(BYTE* dst, BYTE* len)
```

Description:

Tests if there was a frame received over FT12. If this is the case the received data is copied to ‘dst’ and the length is written to ‘len’.

Parameters:

`BYTE* dst:`
A pointer to a data buffer where the received data should be copied

`BYTE* len:`
A pointer to a byte where the number of copied bytes is written

Return values:

true: there was a frame received over FT12 and the data has been copied to ‘dst’
false: no data was received over FT12

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.7 Handshake

The following functions are used to send and receive frames via handshake protocol. It is not recommend using these functions for new implementations.

2.7.1 Function HSInit

Prototype:

```
void HSInit(USHORT TimeoutTime, BYTE len, BYTE* rcvBuffer,
BYTE* trmBuffer, BYTE baud, BYTE config)
```

Description:

This function is used to initialize the driver for sending and receiving frames with handshake flow control over UART0.

Parameters:

`USHORT TimeoutTime:`
The time in between a send operation must be completed

`BYTE len:`
The size of the transmit and receive buffer

`BYTE* rcvBuffer:`
A pointer to a receive buffer

`BYTE* trmBuffer:`
A pointer to a transmit buffer

`BYTE baud:`
The baud rate; the possible values must be looked up in the NEC78K0 / KE2 datasheet

`BYTE config:`
The configuration of the FT12 uart; the possible configurations must be looked up in the NEC78K0 / KE2 datasheet

Return values:

none

Callable in / at:

init

Stack used: 14 bytes

Comment:

It is not recommended using this function for new implementations.

2.7.2 Function HSSetFrame

Prototype:

```
BOOL HSSetFrame(BYTE* src, BYTE len, BYTE* result)
```

Description:

Copies ‘len’ bytes from ‘src’ to the send buffer if it is empty. After sending the result is stored in ‘result’.

Parameters:

`BYTE* src:`
A pointer to data that should be transmitted

`BYTE len:`
The number of bytes that should be transmitted

`BYTE* result:`
A pointer to a byte where the result of the send operation is stored

Return values:

true: the data was copied to send buffer
false: the send buffer was not free, so no data has been copied

Callable in / at:

main

Stack used: 8 bytes

Comment:

It is not recommended using this function for new implementations.

2.7.3 Function HSGetFrame

Prototype:

```
BOOL HSGetFrame(BYTE* dst, BYTE* len)
```

Description:

Tests if there was a frame received over FT12. If this is the case the received data is copied to ‘dst’ and the length is written to ‘len’.

Parameters:

`BYTE* dst:`

A pointer to a buffer where the received bytes should be copied

`BYTE* len:`

A pointer to a byte where the number of copied bytes will be written

Return values:

true: there was a frame received and the data has been copied to ‘dst’
false: no data was received

Callable in / at:

main

Stack used: 8 bytes

Comment:

It is not recommended using this function for new implementations.

2.8 SPI

2.8.1 Function SPIInit

Prototype:

```
void SPIInit(enum SPISpeed speed, BYTE CKPDAP, BOOL MSBFirst)
```

Description:

This function is used to initialize the driver for transmitting data over SPI.

Parameters:

`enum SPISpeed speed:`
The speed of the SPI communication

`BYTE CKPDAP:`
value: 0x00: clock is idle high, data is valid if clock is high
value: 0x01: clock is idle high, data is valid if clock is low
value: 0x02: clock is idle low, data is valid if clock is low
value: 0x03: clock is idle low, data is valid if clock is high

`BOOL MSBFirst:`
Specifies whether the data bytes are transmitted with msb first or lsb first

Return values:

none

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.8.2 Function SPISend

Prototype:

```
BOOL SPISend(BYTE* pData, BYTE length)
```

Description:

This function is used to transmit ‘legth’ bytes from ‘pData’ over SPI. The received data overwrites the data in ‘pData’.

Parameters:

`BYTE* pData:`
A pointer to data that should be transmitted and where the received data is stored

`BYTE length:`
The number of bytes that should be exchanged over SPI

Return values:

true: the SPI data exchange was successful
false: the SPI data exchange was not successful

Callable in / at:

interrupts, init, main, save, unload

Stack used: 8 bytes

Comment:

2.9 Flash

The following functions are used to write data to the flash memory of the BIM.

2.9.1 Function FlashSegErase

Prototype:

```
BOOL FlashSegErase(ULONG dst)
```

Description:

This function is used to erase a segment in flash memory where ‘dst’ is in.

Parameters:

`ULONG dst:`

Specifies which flash segment should be erased. The address has not to be the start address of the flash segment.

Return values:

true: the segment was successful erased

false: the erase operation could not be done; this is normally the case if ‘dst’ is not o.k.

Callable in / at:

main, save

Stack used: 8 bytes

Comment:

After a call of this function the system performs a task switch and the application program is called again when the flash erase operation has been completed.

2.9.2 Function FlashWrite

Prototype:

```
BOOL FlashWrite(ULONG dst, void* src, BYTE count)
```

Description:

This function writes ‘count’ bytes from ‘src’ to ‘dst’ in flash. The ‘dst’ flash memory must be blank flash space.

Parameters:

`ULONG dst:`

The destination address as a four byte value where the data should be written; the flash space where ‘dst’ points to must be blank flash and ‘dst’ must be aligned to a four byte address.

`void* src:`

A pointer to data that should be written to flash

`BYTE count:`

The number of bytes that should be written

Return values:

true: the data has been written to flash

false: the data has not been written to flash; this could be the case if the address is not o.k.

Callable in / at:

main, save

Stack used: 8 bytes

Comment:

The maximum value for ‘count’ is 124. After a call of this function the system performs a task switch and the application program is called again when the flash write operation has been completed.

2.9.3 Function FlashIsBlockBlank

Prototype:

```
BOOL FlashIsBlockBlank(BYTE* pVal, USHORT count)
```

Description:

Tests if a block of bytes in flash memory is blank.

Parameters:

`BYTE* pVal:`

The start address of the block

`USHORT count:`

The number of bytes that should be tested

Return values:

true: the specified block is blank

false: the specified block is not blank

Callable in / at:

interrupts, init, main, save, unload

Stack used: 8 bytes

Comment:

2.10 Parameter

The following functions are used to initialize and read parameter values that are used for properties in the application program

2.10.1 Function ParamInitVal

Prototype:

```
BOOL ParamInitVal(BYTE* src, BYTE ValID, BYTE ValLength)
```

Description:

If you use parameter management you can use this function to set an initial value for a parameter value specified by 'ValID'. The value is only written in the parameter management if it does not already exists.

Parameters:

`BYTE* src:`
A pointer to data that should be written to the specified parameter value

`BYTE ValID:`
The ID that identifies the value

`BYTE ValLength:`
The size of the value in bytes

Return values:

true: operation was successful
false: operation was not successful

Callable in / at:

init

Stack used: 8 bytes

Comment:

2.10.2 Function ParamReadVal

Prototype:

```
BOOL ParamReadVal(BYTE ValID, void* dst, BYTE len)
```

Description:

Reads a value specified by 'ValID' from the parameter management and copies 'len' bytes to 'dst'.

Parameters:

`BYTE ValID:`
The ID that identifies the value

`void* dst:`
A pointer to a buffer in RAM where the data of the specified parameter value should be copied

`BYTE len:`
The number of bytes that should be copied

Return values:

true: the data of the specified parameter value was copied to 'dst'
false: an error occurred; this could be the case when the specified parameter value was not found

Callable in / at:

main

Stack used: 26 bytes

Comment:

2.11 Message

The following functions are used the work with system messages.

2.11.1 Function MsgCreate

Prototype:

```
MESSAGE* MsgCreate(void)
```

Description:

This function tries to get a message from the message pool and returns a pointer to the free message.

Parameters:

none

Return values:

A pointer to the message; if there was no free message, the return value is NULL.

Callable in / at:

main

Stack used: 12 bytes

Comment:

2.11.2 Function MsgDiscard

Prototype:

```
void MsgDiscard(MESSAGE* pMsg)
```

Description:

This function posts a message back to the message pool.

Parameters:

`MESSAGE* pMsg:`

A pointer to the message that should be discarded

Return values:

none

Callable in / at:

main, unload

Stack used: 10 bytes

Comment:

2.11.3 Function MsgGet

Prototype:

```
MESSAGE* MsgGet(MESSAGEQUEUE* pQueue)
```

Description:

This function returns the next message from the specified message queue.

Parameters:

`MESSAGEQUEUE* pQueue:`

A pointer to the queue from where a message should be put out

Return values:

A pointer to the message that was put out from the queue; if there was no message in the queue the return value is NULL.

Callable in / at:

interrupts, init, main, save, unload

Stack used: 10 bytes

Comment:

2.11.4 Function MsgPost

Prototype:

```
void MsgPost(MESSAGE* pMsg, MESSAGEQUEUE* pQueue)
```

Description:

This function posts a message specified by pMsg to the specified message queue.

Parameters:

`MESSAGE* pMsg:`

A pointer to the message that should be posted in the specified queue

`MESSAGEQUEUE* pQueue:`

A pointer to the message queue where the specified message should be posted

Return values:

none

Callable in / at:

main, unload

Stack used: 8 bytes

Comment:

2.11.5 Function MsgUndoGet

Prototype:

```
void MsgUndoGet(MESSAGE* pMsg, MESSAGEQUEUE* pQueue)
```

Description:

This function puts a message witch was put out from a message queue via 'MsgGet' back in the specified queue.

Parameters:

`MESSAGE* pMsg:`

A pointer to the message that should be put back in the specified message queue

`MESSAGEQUEUE* pQueue:`

A pointer to the message queue where the message should be put back

Return values:

none

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.11.6 Function MsgSwitchQueue

Prototype:

```
void MsgSwitchQueue(DYNMESSAGEQUEUE* dynq, MESSAGEQUEUE*
staticq)
```

Description:

This function can be used to redirect a message. After a call of this function a message that is posted to 'dynq' will be posted to 'staticq'.

Parameters:

`DYNMESSAGEQUEUE* dynq:`

A pointer to a dynamic message queue whose messages should be redirected

`MESSAGEQUEUE* staticq:`

A pointer to a static message queue where the redirected messages should be put in

Return values:

none

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.11.7 Function MsgResetDynQueues

Prototype:

```
void MsgResetDynQueues(void)
```

Description:

Use this function to restore the original system message redirection system.

Parameters:

none

Return values:

none

Callable in / at:

Main, unload

Stack used: 8 bytes

Comment:

2.12 Utility

The following functions are useful utilities.

2.12.1 Function IsApplicationLoaded

Prototype:

```
BOOL IsApplicationLoaded(void)
```

Description:

This function indicates if the application is loaded.

Parameters:

none

Return values:

true: the application is loaded
false: the application is not loaded

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

2.12.2 Function GetPhysAddr

Prototype:

```
void GetPhysAddr(BYTE* dst)
```

Description:

Use this function to read the physical address of this device.

Parameters:

`BYTE* dst:`

A pointer to a buffer with at least 2 bytes in RAM where the actual physical address of the device will be copied

Return values:

none

Callable in / at:

main

Stack used: 8 bytes

Comment:

2.12.3 Function GetSerialNumber

Prototype:

```
void GetSerialNumber(BYTE* dst)
```

Description:

Use this function to read the serial number of this device.

Parameters:

`BYTE* dst:`

A pointer to a buffer with at least 6 bytes in RAM where the actual serial number of the device will be copied

Return values:

none

Callable in / at:

main

Stack used: 26 bytes

Comment:

2.12.4 Function ReadBCU2Adr100

Prototype:

```
BOOL ReadBCU2Adr100(BYTE offset, BYTE count, void* dst)
```

Description:

Use this function to copy ‘count’ bytes to ‘dst’ from the simulated address 0x0100 from the bcu 2.

Parameters:

`BYTE offset:`

The offset starting from address 0x0100

`BYTE count:`

The number of bytes that should be copied

`void* dst:`

A pointer to a buffer in RAM where the data will be copied

Return values:

true: the read operation was successful
false: an error occurred; this would be the case if something is wrong with the parameters

Callable in / at:

main

Stack used: 26 bytes

Comment:

2.12.5 Function TriggerWatchDog

Prototype:

```
void TriggerWatchDog(void)
```

Description:

Call this function cyclic to retrigger the user application watchdog.

Parameters:

none

Return values:

none

Callable in / at:

interrupts, init, main, save, unload

Stack used: 8 bytes

Comment:

The watchdog time for the application watchdog is configured in application info block.

2.13 Interrupts

The interrupt functions enable the application programmer to use some interrupts of the BIM M 13x microcontroller. The maximum execution time for an interrupt handling routine in the application program must not exceed **100µsec**! The following interrupt vector addresses could be used and are defines in the header file 'io78f053x_64.h' which is already included in 'BIM_M13x.h':

INTP4_vect	INTKR_vect
INTST0_vect	INTSR0_vect
INTTM000_vect	INTTM010_vect
INTTM51_vect	
INTCSI11_vect	

2.13.1 Function IntRegister

Prototype:

```
void IntRegister(pIntFunc func, BYTE IntAddr)
```

Description:

This function is used to register interrupt service routines for the application program.

Parameters:

`pIntFunc func:`

A pointer to the function that will handle the specified interrupt

`BYTE IntAddr:`

One of the allowed interrupt vector addresses that should be handled by the specified function

Return values:

none

Callable in / at:

init, main

Stack used: 8 bytes

Comment:

The maximum execution time for an interrupt handling routine in the application program must not exceed 100µsec.

2.13.2 Function IntUnregister

Prototype:

```
void IntUnregister(BYTE IntAddr)
```

Description:

Use this function to unregister a interrupt service routine.

Parameters:

`BYTE IntAddr:`

The interrupt vector address that will not longer be handled by the user application

Return values:

none

Callable in / at:

main, unload

Stack used: 8 bytes

Comment:

2.13.3 Function IntResetAll

Prototype:

```
void IntResetAll(void)
```

Description:

This function unregisters all interrupts that could be registered from user application

Parameters:

void

Return values:

none

Callable in / at:

main, unload

Stack used: 8 bytes

Comment:
