KETEREX

AN2101
Application Note

USB-910H API DLL and Include File Reference Manual

APPLICABLE ADAPTERS

This Application Note applies to the following Keterex products: KXUSB-910H.

INTRODUCTION

The Keterex USB-910H Embedded Systems Interface provides connectivity between a Host PC and a target system requiring I2C, SMBus, or SPI. Other interfaces can be supported by “bit-banging” up to 9 general-purpose I/O. USB transactions are generated on the Host PC using either the Keterex USB-900 Control Center application or calls to the provided API. The USB-910H Adapter converts these transactions to I2C, SMBus, SPI, or general-purpose I/O operations. Built-in scripting accommodates complex bus protocols, including forcing and/or detecting general-purpose I/O changes during other operations.

The KXUSB-910H API provides a set of functions to control all features in the USB-910H Adapter. These functions are provided in two files, a DLL named “kxusb910h.dll” and an include file name “kxusb910h.h”. An import library file name “kxusb910h.lib” is also provided. This Application Note describes all API functions, including brief examples.

KEY POINTS

- No Drivers Required
- Acts as an I2C, SMBus, or SPI Master or Slave
- Full I2C/SMBus Multi-Master Support
- Supports up to 16 Slave Addresses
- Up to 64kBytes per Operation
- Flexible Scripting Feature
- Firmware Update Supported in the Field.

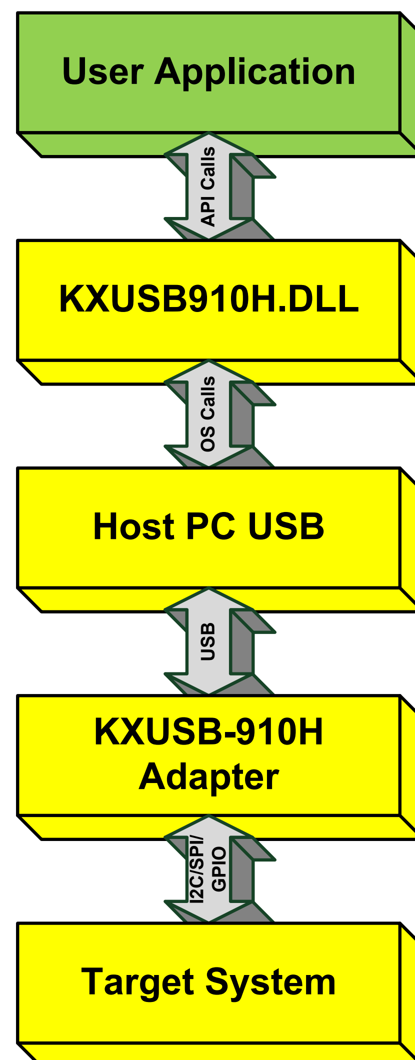

Figure 1. Software Stack between the User's Application and a Target System.

© Keterex, Inc., 2009. All rights reserved.

1. Introduction

The KXUSB-910H API functions are provided in a DLL named “kxusb910h.dll” (generated under Microsoft Visual C++ .NET). This DLL, combined with the include file “kxusb910.h” provides a complete interface to the USB-910H Embedded System Interface product. This is the same DLL used by the Keterex USB-900 Control Center application. An import library file named “kxusb910h.lib” is also provided to support linking of a user’s application. All API files can be found in the API directory of the KXUSB-910H Install CD.

All functions in kxusb910h.dll are implemented as C functions (i.e. they are not contained within a class). The include file “kxusb910h.h” checks if the user’s compiler is C++ compatible and generates the proper import code as follows:

```
#ifdef __cplusplus

#ifdef KXUSB_EXPORTS
#define KXUSB_API extern "C" __declspec(dllexport)
#else
#define KXUSB_API extern "C" __declspec(dllimport)
#endif

#else

#ifdef KXUSB_EXPORTS
#define KXUSB_API extern __declspec(dllexport)
#else
#define KXUSB_API extern __declspec(dllimport)
#endif

#endif
```

This allows the include file to operate with a C or C++ compiler. The KXUSB_EXPORTS macro is used when generating the DLL and should not be defined by the user.

All parameters passed to an API function or returned by an API function use custom data types defined in “kxusb910h.h”. These data types should be used in the host software to guarantee proper operation of the DLL functions. The include file also defines constants used by many of the API functions.

2. Directory of API Functions

The following table provides a brief directory of the available API functions:

API Function (click to follow link)	Description
Setup/Shared Functions	
kxFindAdapters	Finds all unopened USB-910H Adapters connected to the Host PC.
kxGetSerialString	Gets the Serial String for an unopened adapter.
kxOpenAdapter	Opens (i.e. connects to) an adapter.
kxGetVersionString	Gets the Version String for the opened adapter.
kxEnableFeatures	Sets the features to enable for the opened adapter.
kxSetTargetVoltage	Sets the I/O voltage for the opened adapter and configures whether the target system is powered by the adapter.
kxGetStatus	Gets the status of the adapter.
kxGetError	Gets the errors which have occurred since this function was last called.
kxCloseAdapter	Closes the opened adapter.
kxGetInData	Retrieves the incoming data from the adapter.
kxSetSlaveData	Sets the outgoing slave data for the next slave operation.
kxSetSlaveReady	Instructs a slave to start listening for its address or slave-select.
kxSetSlaveReady	Sets the timeout period when waiting for GPIO states.
kxAbort	Attempts to cleanly abort the present operation.
kxStartTimer	Starts a general-purpose timer.
kxTimerExpired	Returns KX_TRUE if the general-purpose timer has expired.
I2C/SMBus Functions	
kxI2CsetBitRate	Sets the I2C bit-rate.
kxI2CsetSCLtimeout	Sets the SCL low timeout period.
kxI2CsetConfig	Sets the I2C configuration.
kxI2CgetConfig	Gets the I2C configuration.
kxI2CfreeBus	Forces the I2C hardware to free the bus.
kxI2CsetAddr	Sets the list of slave addresses the I2C hardware will respond to.
kxI2Cwrite	Instructs the I2C Master to write a block of data to a slave device.
kxI2Cread	Instructs the I2C Master to read a block of data from a slave device.
kxI2Cexecute	Instructs the I2C Master to execute a script.
kxI2CgetAddrIndices	Gets the list of indices at which the I2C Slave stored the incoming addresses (if responded to). These indices correspond to the data array provided by kxGetInData.
SPI Functions	
kxSPIsetBitRate	Sets the SPI bit-rate.
kxSPIsetTiming	Sets the timing of slave-select pin edges.
kxSPIsetConfig	Sets the SPI configuration.
kxSPIgetConfig	Gets the SPI configuration.
kxSPIgetConfig	Instructs the SPI Master to transfer a block of data between itself and the slave.
kxSPIexecute	Instructs the SPI Master to execute a script.
GPIO Functions	
kxGPIOsetDir	Sets the direction of general-purpose I/O pins.
kxGPIOsetState	Sets the output value of general-purpose I/O pins.
kxGPIOgetStatus	Returns the status of all general-purpose I/O pins.

3. Adapter Pin Identifiers

The USB-910H Adapter accepts scripts which supports the ability to manipulate the various pins available on the adapter. Adapter pins (generally referred to as general-purpose I/O, or GPIO, in this document) are identified in scripts using the following two-character identifiers:

Adapter Pin	Pin Identifier (GPIO ID)
SCLK	SK
MISO	MI
MOSI	MO
SS	SS
SDA	SD
SCL	SL
GP1	G1
GP2	G2
GP3	G3
NONE	NO

4. Data Types

The following C language data types are pre-defined in the file “kxusb910h.h”. These data types are used to pass and receive values to/from the various API functions.

4.1. KX_U8

Description:

This data type defines an unsigned 8-bit value. It is used only to generate other data types and is not used directly by any API function.

Definition:

```
typedef unsigned char KX_U8;
```

Details:

The KX_U8 data type is defined to provide an 8-bit value independent of the compiler used.

4.2. KX_U16

Description:

This data type defines an unsigned 16-bit value. It is used only to generate other data types and is not used directly by any API function.

Definition:

```
#ifdef USHRT_MAX == 0xffff
typedef unsigned short KX_U16;
#else
typedef unsigned int KX_U16;
#endif
```

Details:

The KX_U16 data type is defined to provide a 16-bit value independent of the compiler used.

4.3. KX_ADAPTER

Description:

This data type contains a unique identifier passed to adapter-specific functions.

Definition:

```
typedef KX_U8 KX_ADAPTER;
```

Details:

The DLL maintains an internal list of adapters which are established by a call to kxFindAdapters. A specific adapter can be addressed from this list using the KX_ADAPTER data type.

4.4. KX_BOOL

Description:

This data type contains a true/false boolean value.

Definition:

```
typedef KX_U8 KX_BOOL;

#define KX_TRUE      ((KX_BOOL)1)
#define KX_FALSE     ((KX_BOOL)0)
```

Details:

Most API functions return a KX_BOOL value indicating whether the function succeeded (i.e. return KX_TRUE) or failed (i.e. return KX_FALSE).

4.5. KX_VOLTAGE

Description:

This data type contains a voltage value specified in volts.

Definition:

```
typedef double KX_VOLTAGE;
```

Details:

The KX_VOLTAGE type is used, for example, when setting the target voltage of an adapter.

4.6. KX_RATE

Description:

This data type contains a rate value specified in kbits/sec.

Definition:

```
typedef double KX_RATE;
```

Details:

The KX_RATE type is used, for example, when setting the bit-rate of an adapter.

4.7. KX_TIME

Description:

This data type contains a time value specified in seconds.

Definition:

```
typedef double KX_TIME;
```

Details:

The KX_TIME type is used, for example, when setting the I2C timeout period of an adapter.

4.8. KX_ERROR

Description:

This data type contains a bit field indicating the cause of an error.

Definition:

```
typedef KX_U16 KX_ERROR;

#define KX_ERROR_INVALID_COMMAND          ((KX_ERROR)0x0001U)
#define KX_ERROR_FEATURE_NOT_SUPPORTED    ((KX_ERROR)0x0002U)
#define KX_ERROR_FEATURE_COMBINATION      ((KX_ERROR)0x0004U)
#define KX_ERROR_FEATURE_NOT_ENABLED      ((KX_ERROR)0x0008U)
#define KX_ERROR_INVALID_PARAMETER        ((KX_ERROR)0x0010U)
#define KX_ERROR_BUFFER_OVERFLOW          ((KX_ERROR)0x0020U)
#define KX_ERROR_RX_NACK                  ((KX_ERROR)0x0040U)
#define KX_ERROR_BAD_EXPECTED_VALUE       ((KX_ERROR)0x0080U)
#define KX_ERROR_GPIO_TIMEOUT             ((KX_ERROR)0x0100U)
#define KX_ERROR_SCL_TIMEOUT              ((KX_ERROR)0x0200U)
#define KX_ERROR_BUS_ERROR                ((KX_ERROR)0x0400U)
#define KX_ERROR_RX_OVERRUN               ((KX_ERROR)0x0800U)
#define KX_ERROR_ABORT_LOST_ARB           ((KX_ERROR)0x1000U)
#define KX_ERROR_ADAPTER_NOT_FOUND        ((KX_ERROR)0x2000U)
#define KX_ERROR_ADAPTER_NOT_OPEN         ((KX_ERROR)0x4000U)
#define KX_ERROR_ADAPTER_TIMEOUT          ((KX_ERROR)0x8000U)
```

Details:

A KX_ERROR type is returned by the kxGetError function. This function is generally called after another API function returns a KX_FALSE, indicating an error occurred. The following table describes each error value:

KX_ERROR Value	Description
KX_ERROR_INVALID_COMMAND	Indicates that an invalid command was rejected by the adapter.
KX_ERROR_FEATURE_NOT_SUPPORTED	Indicates that an unsupported feature (e.g. a feature not defined by one of the KX_FEATURE values) was rejected by the adapter.
KX_ERROR_FEATURE_COMBINATION	Indicates that an illegal combination of features was requested. For example, only one of the I2C Master, I2C Slave, SPI Master, or SPI Slave features may be enabled at a time.
KX_ERROR_FEATURE_NOT_ENABLED	Indicates that a command which requires a particular feature was rejected since that feature was not enabled.
KX_ERROR_INVALID_PARAMETER	Indicates that a command was rejected because one or more of it parameters were invalid, e.g. out of range.

KX_ERROR_BUFFER_OVERFLOW	Indicates that incoming data was lost due to the adapter buffer being full.
KX_ERROR_RX_NACK	Indicates that an I2C Master operation failed because an unexpected NACK was received.
KX_ERROR_BAD_EXPECTED_VALUE	Indicates that an expected I2C or SPI value was incorrect.
KX_ERROR_GPIO_TIMEOUT	Indicates that an I2C or SPI operation terminated because a wait for a GPIO value timed-out.
KX_ERROR_SCL_TIMEOUT	Indicates that an SCL low timeout event was detected on the I2C bus.
KX_ERROR_BUS_ERROR	Indicates that an I2C protocol error was detected (e.g. an unexpected STOP).
KX_ERROR_RX_OVERRUN	Indicates that incoming data was lost while operating as a SPI slave.
KX_ERROR_ABORT_LOST_ARB	Indicates that an I2C operation was aborted because arbitration was lost.
KX_ERROR_ADAPTER_NOT_FOUND	Indicates that a requested adapter (e.g. during an kxOpenAdapter call) was not found.
KX_ERROR_ADAPTER_NOT_OPEN	Indicates that an API call was rejected since no adapter was open.
KX_ERROR_ADAPTER_TIMEOUT	Indicates that an API call failed due to a timeout while waiting for an adapter response.

4.9. KX_STATUS

Description:

This data type contains a bit field indicating the present status of an adapter.

Definition:

```
typedef KX_U16 KX_STATUS;

#define KX_STATUS_I2C_MASTER_PENDING        ((KX_STATUS)0x0001)
#define KX_STATUS_I2C_SLAVE_PENDING         ((KX_STATUS)0x0002)
#define KX_STATUS_I2C_LOST_ARBITRATION      ((KX_STATUS)0x0004)
#define KX_STATUS_SPI_MASTER_PENDING        ((KX_STATUS)0x0008)
#define KX_STATUS_SPI_SLAVE_PENDING         ((KX_STATUS)0x0010)
#define KX_STATUS_MEMORY_FULL               ((KX_STATUS)0x0020)
#define KX_STATUS_ADAPTER_UPDATED           ((KX_STATUS)0x0040)
#define KX_STATUS_ERROR                     ((KX_STATUS)0x8000)
```

Details:

The KX_STATUS data type is used when calling the kxGetStatus function. Such a call is made to determine the status of a requested action by an adapter. The following table describes of each status value:

KX_STATUS Value	Description
KX_STATUS_I2C_MASTER_PENDING	Indicates that a requested I2C Master operation is pending, i.e. either waiting to execute or is still executing.

KX_STATUS_I2C_SLAVE_PENDING	Indicates that a requested I2C Slave operation is pending, i.e. either waiting to execute or is still executing.
KX_STATUS_I2C_LOST_ARBITRATION	Indicates that the I2C Master lost arbitration during the present operation.
KX_STATUS_SPI_MASTER_PENDING	Indicates that a requested SPI Master operation is pending, i.e. either waiting to execute or is still executing.
KX_STATUS_SPI_SLAVE_PENDING	Indicates that a requested SPI Slave operation is pending, i.e. either waiting to execute or is still executing.
KX_STATUS_MEMORY_FULL	Indicates that the outgoing buffer is full. This bit is used by the DLL to handshake with the adapter when transferring data. In general, it will not be used by the user's application.
KX_STATUS_ADAPTER_UPDATED	Indicates that the adapter firmware was updated during to a call to kxOpenAdapter.
KX_STATUS_ERROR	Indicates that one or more errors have occurred since the last call to kxGetError.

4.10. KX_DATA

Description:

This data type contains data (e.g. read or write data) passed to an adapter.

Definition:

```
typedef KX_U8 KX_DATA;
```

Details:

The KX_DATA type is used to send and receive byte data to various API functions.

4.11. KX_ADDR

Description:

This data type contains an I2C address passed to an adapter.

Definition:

```
typedef KX_U16 KX_ADDR;
```

Details:

The KX_ADDR type is used to send I2C Slave addresses to various API functions.

4.12. KX_GPIOIDX

Description:

This data type contains an index value indicating a single adapter pin.

Definition:

```
typedef KX_U8 KX_GPIOIDX;

#define KX_GPIOIDX_SCLK                 ((KX_GPIOIDX)0U)
```

```
#define KX_GPIOIDX_MISO          ((KX_GPIOIDX)1U)
#define KX_GPIOIDX_MOSI          ((KX_GPIOIDX)2U)
#define KX_GPIOIDX_SS            ((KX_GPIOIDX)3U)
#define KX_GPIOIDX_SDA           ((KX_GPIOIDX)4U)
#define KX_GPIOIDX_SCL           ((KX_GPIOIDX)5U)
#define KX_GPIOIDX_GP1           ((KX_GPIOIDX)6U)
#define KX_GPIOIDX_GP2           ((KX_GPIOIDX)7U)
#define KX_GPIOIDX_GP3           ((KX_GPIOIDX)8U)
#define KX_GPIOIDX_NONE          ((KX_GPIOIDX)255U)
```

Details:

The KX_ GPIOIDX data type is used when passing a GPIO identifier to various API functions and in execution strings. The "NONE" pin is used, for example, when setting a Slave-Select to "NONE", i.e. to no Slave-Select.

4.13. KX_GPIOMASK

Description:

This data type contains a bit field indicating a collection of GPIO signals.

Definition:

```
typedef KX_U16 KX_GPIOMASK;

#define KX_GPIOMASK_SCLK         ((KX_GPIOMASK)0x0001U)
#define KX_GPIOMASK_MISO         ((KX_GPIOMASK)0x0002U)
#define KX_GPIOMASK_MOSI         ((KX_GPIOMASK)0x0004U)
#define KX_GPIOMASK_SS           ((KX_GPIOMASK)0x0008U)
#define KX_GPIOMASK_SDA          ((KX_GPIOMASK)0x0010U)
#define KX_GPIOMASK_SCL          ((KX_GPIOMASK)0x0020U)
#define KX_GPIOMASK_GP1          ((KX_GPIOMASK)0x0040U)
#define KX_GPIOMASK_GP2          ((KX_GPIOMASK)0x0080U)
#define KX_GPIOMASK_GP3          ((KX_GPIOMASK)0x0100U)
```

Details:

The KX_GPIOMASK data type is used when passing GPIO identifiers to various API functions.

4.14. KX_STRING

Description:

This data type contains a null-terminated ASCII string used to contain Version and Serial strings retrieved from an adapter.

Definition:

```
typedef KX_U8 KX_STRING;
```

Details:

The KX_STRING type is used to retrieve ASCII strings from various API functions.

4.15. KX_I2CEXESTR

Description:

This data type contains a null-terminated ASCII string to serve as a script of I2C actions to be executed by an adapter.

Definition:

typedef KX_U8 KX_I2CEXESTR;

Details:

A KX_I2CEXESTR string consists of a script of tokens, each 2 characters long. White space (spaces or tabs) can be placed between tokens and are ignored by the DLL. The following tokens are supported:

"/F" Set the I2C configuration. The following token is expected to represent a hexadecimal byte that is an OR of the desired KX_I2CCONFIG values. This configuration remains in effect until changed by a call to kxI2CsetConfig or another /F token.

"/S" Generate a START condition on the I2C bus.

"/P" Generate a STOP condition on the I2C bus.

"/D" Delay a period of time before continuing the script. The next two tokens are assumed to indicate the delay in microseconds listed in hexadecimal, MSB first.

"/C" Either write a SMBus PEC byte (if performing a write) or read a SMBus PEC (if performing a read). If a read, the byte is compared against the expected value and the KX_ERROR_BAD_EXPECTED_VALUE error bit is set if incorrect. The script is aborted if the KX_I2CCONFIG_ABORT_ON_EXPECT configuration flag is set. The PEC is calculated using all addresses and bytes transmitted or received since the last bus-free condition using the SMBus CRC-8 algorithm.

"/E" Expect a value. This token requires that the previous byte read match the following token value. This token can be used, for example, to expect PEC values calculated using alternative algorithms to SMBus CRC-8. If an expected value fails, the KX_ERROR_BAD_EXPECTED_VALUE status bit is set and the execution will be aborted if enabled to do so.

"/R" Read a block of data. The next two tokens indicate the number of bytes to read in hexadecimal, listed MSB first.

"/1" Drive a GPIO pin High. The next token indicates the GPIO pin to drive using a GPIO ID (see Adapter Pin Identifiers). The GPIO is automatically set as an output.

"/0" Drive a GPIO pin Low. The next token indicates the GPIO pin to drive using a GPIO ID (see Adapter Pin Identifiers). The GPIO is automatically set as an output.

"/Z" Float a GPIO pin. The next token indicates the GPIO pin to float (i.e. set as an input) using a GPIO ID (see Adapter Pin Identifiers). The GPIO is automatically set as an input.

"/H" Wait for a GPIO pin to go High. The next token indicates the GPIO pin to detect using a GPIO ID (see Adapter Pin Identifiers). The GPIO is automatically set as an input. Waiting for GP3 is not allowed.

"/L" Wait for a GPIO pin to go Low. The next token indicates the GPIO pin to detect using a GPIO ID (see Adapter Pin Identifiers). The GPIO is automatically set as an input. Waiting for GP3 is not allowed.

"HH" A valid hexadecimal byte to serve as a write byte.

Examples:

"/S 80 00 11 22 5A /P"	Write [0x00,0x11,0x22,0x5A] to address 0x40.
"/S81/R0100/P"	Read 256 bytes from address 0x40.
"/S0255/S03/R0004/C/P"	Write a command (0x55) to address 0x01 followed by a repeated start and a read of 4 bytes plus a PEC byte.
"/HG1/S02AA/P/D0100/S09/R0010/P"	Wait for GP1 to go High, write 0xAA to address 0x01, delay 256µs, then read 16 bytes from address 0x04.

4.16. KX_SPIEXESTR

Description:

This data type contains a null-terminated ASCII string to serve as a script of SPI Master actions to be executed by an adapter.

Definition:

typedef KX_U8 KX_SPIEXESTR;

Details:

A KX_SPIEXESTR string consists of a script of tokens, each 2 characters long. White space (spaces or tabs) can be placed between tokens and are ignored by the DLL. The following tokens are supported:

"/F" Set the SPI configuration. The following token is expected to represent a hexadecimal byte that is an OR of the desired KX_SPICONFIG values. The token after that provides the GPIO pin to use as the Slave-Select using a GPIO ID (see Adapter Pin Identifiers). If the Slave-Select pin is "NO" the master will not assert a Slave-Select. The next pair of tokens provide the number of bytes to transfer for each Slave-Select cycle in hexadecimal (listed MSB first). The master will transfer this number of bytes each time the Slave-Select pin is asserted (whether the pin is "NO" or not). It will then de-assert the pin, delay the configured period of time, and repeat until all data is transferred. This configuration remains in effect until changed by a call to kxSPIsetConfig or another /F token.

"/R" Read a block of data with a repeated outgoing byte. The next token in the string is repeatedly transmitted. The next pair of tokens provide the number of times to repeat this byte in hexadecimal (listed MSB first).

"/1" Drive a GPIO pin High. The next token indicates the GPIO pin to drive using a GPIO ID value (see Adapter Pin Identifiers). The GPIO is automatically set as an output.

"/0" Drive a GPIO pin Low. The next token indicates the GPIO pin to drive using a GPIO ID value (see Adapter Pin Identifiers). The GPIO is automatically set as an output.

"/Z" Float a GPIO pin (i.e. set as an input). The next token indicates the GPIO pin to float using a GPIO ID value (see Adapter Pin Identifiers). The GPIO is automatically set as an output.

"/H" Wait for a GPIO pin to go High. The next token indicates the GPIO pin to detect using a GPIO ID value (see Adapter Pin Identifiers). The GPIO is automatically set as an input. Waiting for GP3 is not allowed.

"/L" Wait for a GPIO pin to go Low. The next token indicates the GPIO pin to detect using a GPIO ID value (see Adapter Pin Identifiers). The GPIO is automatically set as an input. Waiting for GP3 is not allowed.

"/E" Expect a value. This token requires that the next token in the execution string match the previous incoming byte value. If an expected value fails, the KX_ERROR_BAD_EXPECTED_VALUE error bit is set and the execution will be aborted if enabled to do so by the present configuration (see KX_SPICONFIG).

"/D" Delay a period of time before continuing the execution. The next two tokens are assumed to indicate the delay in microseconds listed as hexadecimal MSB first.

"HH" A valid hexadecimal byte to serve as a outgoing byte.

Examples:

"00 11 22 A5" Transfer [0x00,0x11,0x22,0xA5].

"00112233/E55" Transfer [0x00,0x11,0x22,0x33] – require that the last byte received is 0x55.

"/LG2/R0000FF/0G3" Wait for GP2 to go Low, receive 255 bytes while sending 0x00, then drive GP3 low.

4.17. KX_SPICONFIG

Description:

This data type contains a bit field indicating a SPI transfer mode.

Definition:

```
typedef KX_U8 KX_SPICONFIG;

#define KX_SPICONFIG_LSB_FIRST              ((KX_SPICONFIG)0x01)
#define KX_SPICONFIG_ABORT_ON_EXPECT        ((KX_SPICONFIG)0x02)
#define KX_SPICONFIG_SS_ACTIVE_HIGH         ((KX_SPICONFIG)0x04)
#define KX_SPICONFIG_PHASE                  ((KX_SPICONFIG)0x08)
#define KX_SPICONFIG_POLARITY               ((KX_SPICONFIG)0x10)
#define KX_SPICONFIG_SS_HIZ_MISO            ((KX_SPICONFIG)0x20)
```

Details:

The KX_SPICONFIG data type is used when calling the kxSPIsetConfig function. The following table describes each configuration bit value:

KX_SPICONFIG Value	Description
KX_SPICONFIG_LSB_FIRST	Instructs the adapter to transmit and receive all data bytes least-significant-bit first. Otherwise, the MSB is transmitted / received first.
KX_SPICONFIG_ABORT_ON_EXPECT	Instructs the adapter to abort the operation if an expected value (i.e. when using the /E token) is incorrect.
KX_SPICONFIG_SS_ACTIVE_HIGH	Instructs the Adapter to assert the Slave-

	Select pin HIGH during each transaction. Otherwise, the pin is asserted LOW.
KX_SPICONFIG_PHASE	Instructs the adapter to transition/sample MISO/MOSI on the leading edge of SCLK. Otherwise, MISO/MOSI transitions/samples of the trailing edge of SCLK.
KX_SPICONFIG_POLARITY	Instructs the adapter to either idle the SCLK signal HIGH (when a master) or expect the SCLK to idle HIGH (when a slave). Otherwise, SCLK idles low.
KX_SPICONFIG_SS_HIZ_MISO	When operating as a SPI slave, this value instructs the adapter to tri-state the MISO pin when its assigned Slave-Select pin is de-asserted.

4.18. KX_I2CCONFIG

Description:

This data type contains a bit field indicating an I2C configuration.

Definition:

```
typedef KX_U8 KX_I2CCONFIG;

#define KX_I2CCONFIG_LSB_FIRST              ((KX_I2CCONFIG)0x01)
#define KX_I2CCONFIG_ABORT_ON_EXPECT        ((KX_I2CCONFIG)0x02)
#define KX_I2CCONFIG_RETRY_ON_LOST_ARB      ((KX_I2CCONFIG)0x04)
#define KX_I2CCONFIG_FREE_BUS_ENABLE        ((KX_I2CCONFIG)0x08)
```

Details:

The KX_I2CCONFIG data type is used when calling the kxI2CsetConfig function. The following table describes the I2CCONFIG values:

KX_I2CCONFIG Value	Description
KX_I2CCONFIG_LSB_FIRST	Instructs the adapter to transmit and receiver all data bytes least-significant-bit first. Otherwise, the MSB is transmitted / received first. Address bytes are always transmitted MSB first.
KX_I2CCONFIG_ABORT_ON_EXPECT	Instructs the adapter to abort the operation if an expected value (e.g. using the /C token) is incorrect.
KX_I2CCONFIG_RETRY_ON_LOST_ARB	Instructs the adapter to retry if an operation terminates due to loss of arbitration.
KX_I2CCONFIG_FREE_BUS_ENABLE	Instructs the adapter to detect the bus as free if SCL remains high for 3.3 clock periods.

4.19. KX_FEATURE

Description:

This data type contains a bit field indicating a collection of supported features.

Definition:

```
typedef KX_U8 KX_FEATURE;
```

```
#define KX_FEATURE_I2CMST        ((KX_FEATURE)0x0001U)
#define KX_FEATURE_I2CSLV        ((KX_FEATURE)0x0002U)
#define KX_FEATURE_SPIMST        ((KX_FEATURE)0x0004U)
#define KX_FEATURE_SPISLV        ((KX_FEATURE)0x0008U)
#define KX_FEATURE_I2CRPU        ((KX_FEATURE)0x0010U)
```

Details:

The KX_FEATURE data type is used when setting which features of the adapter are to be enabled. The following table describes the available features:

KX_FEATURE Value	Description
KX_FEATURE_I2CMST	Enables the adapter as an I2C or SMBus Master device.
KX_FEATURE_I2CSLV	Enables the adapter as an I2C or SMBus Slave device.
KX_FEATURE_SPIMST	Enables the adapter as a SPI Master device.
KX_FEATURE_SPISLV	Enables the adapter as a SPI Slave device.
KX_FEATURE_I2CRPU	Connects 2.2kΩ resistors between the I2C pins (SCL and SDA) and the configured I/O voltage. This feature is allowed regardless of whether the adapter is an I2C Master or Slave. It is generally best to enable this feature (assuming the pull-ups are needed) before enabling the I2C Master or I2C Slave feature. If they are enabled in the same call to kxEnableFeatures, the adapter will automatically enable the pull-ups before the I2C interface.

5. Setup/Shared API Functions

5.1. kxFindAdapters

Description:

This function returns the number of connected, unopened adapters.

Prototype:

KX_ADAPTER kxFindAdapters(void);

Arguments:

none.

Return:

This function returns the number of USB-connected, unopened adapters. If no adapters are found, a value of 0 is returned. Calls to functions requiring a KX_ADAPTER parameter must be passed a value between 0 and the returned value minus one.

Example:

KX_ADAPTER numAdapters = kxFindAdapters();

5.2. kxGetSerialString

Description:

This function retrieves the Serial string from a specified, unopened adapter. The adapter must be unopened to retrieve its serial string.

Prototype:

KX_BOOL kxGetSerialString(KX_ADAPTER adapter, KX_STRING **str);

Arguments:

adapter The ID of the adapter to retrieve the serial string from.
str The function sets this pointer to the address of a static array in memory containing the serial string. Do not pass this value to free().

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_ADAPTER adapter;
KX_STRING *pStr;
…
if(kxGetSerialString(adapter,&pStr) == KX_TRUE){
        printf("Serial = %s\n",pStr);
}
```

5.3. kxOpenAdapter

Description:

This function opens and initializes a connected adapter. A call to this function is required before any additional adapter-specific functions are called for this adapter.

Prototype:

KX_BOOL kxOpenAdapter(KX_ADAPTER adapter);

Arguments:

adapter The ID of the adapter to open.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxOpenAdapter(0) == KX_TRUE){
        …
}
```

5.4. kxGetVersionString

Description:

This function returns the Version string from a given adapter.

Prototype:

KX_BOOL kxGetVersionString(KX_STRING **str);

Arguments:

str The function sets this pointer to the address of a static array in memory containing the version string. Do not pass this value to free().

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_STRING *str;

if(kxGetVersionString(&str) == KX_TRUE){
        printf("Version = %s\n",str);
}
```

5.5. kxEnableFeatures

Description:

This function sets which features are enabled in an adapter.

Prototype:

KX_BOOL kxEnableFeatures(KX_FEATURE features);

Arguments:

features A KX_FEATURE value indicating the features to enable.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxEnableFeatures(KX_FEATURE_I2CMST | KX_FEATURE_I2CRPU) ==
        KX_TRUE){
        …
}
```

5.6. kxSetTargetVoltage

Description:

This function sets the programmable target voltage level for the adapter and sets if the target voltage is connected to the target. This level establishes the I/O voltage thresholds of the adapter as well as the voltage applied to the target when enabled. It can take several milliseconds for the requested voltage to stabilize. Therefore, it is generally recommend to set the voltage level, delay, then set the enable if needed.

Prototype:

KX_BOOL kxSetTargetVoltage(KX_VOLTAGE volts, KX_BOOL enable);

Arguments:

volts The requested target voltage in volts. This parameter must be either exactly 5.0, or between 1.65 and 3.6, otherwise an error is returned.

enable If KX_TRUE, the requested target voltage is connected to the target.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSetTargetVoltage(1.8, KX_TRUE) == KX_TRUE){
        …
}
```

5.7. kxGetStatus

Description:

This function provides the current status of an adapter.

Prototype:

KX_BOOL kxGetStatus(KX_STATUS *status);

Arguments:

status The memory location to store the requested status value.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_STATUS status;

if(kxGetStatus(&status) == KX_TRUE &&
        (status & KX_STATUS_I2C_MASTER_PENDING)){
        …
}
```

5.8. kxGetError

Description:

This function returns the cause of errors, if any, which have occurred since the last call to kxGetError.

Prototype:

KX_ERROR kxGetError(void);

Arguments:

none

Return:

Returns a KX_ERROR value which indicates the cause of all errors since the last call of this function. A return value of 0 indicates no error has occurred. The errors are automatically cleared after each call to kxGetError.

Example:

```
if(kxGetError() & KX_ERROR_ADAPTER_NOT_FOUND){
        …
}
```

5.9. kxCloseAdapter

Description:

This function sets all outputs from the adapter to high-impedance, disables the target VDD and closes communications with the adapter.

Prototype:

KX_BOOL kxCloseAdapter(void);

Arguments:

none

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxCloseAdapter() == KX_TRUE){
        …
}
```

5.10. kxGetInData

Description:

This function returns the incoming data from any SPI transaction or I2C read transaction (e.g. kxI2Cexecute, kxSPItransfer, etc.).

Prototype:

```
KX_BOOL kxGetInData(KX_DATA **rdata, KX_COUNT *rcnt);
```

Arguments:

rdata The function sets this pointer to the location of data retrieved from the adapter. The pointer value points to a static array - do not pass to free().

rcnt The function stores the number of bytes at this address (0 if no data is available).

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_DATA *rdata;
KX_COUNT rcnt;

if(kxGetInData(&rdata,&rcnt) == KX_TRUE) {
        …
}
```

5.11. kxSetSlaveData

Description:

This function sets the outgoing data to be sent on the next I2C or SPI Slave operation.

Prototype:

```
KX_BOOL kxSetSlaveData(KX_DATA *data, KX_COUNT cnt);
```

Arguments:

data A pointer to the slave data.

cnt The number of bytes of slave data.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_DATA rdata[] = {0x00,0x11,0x22,0x33};

if(kxSetSlaveData(rdata,sizeof(rdata)) == KX_TRUE &&
        kxSetSlaveReady(0) == KX_TRUE) {
        …
}
```

5.12. kxSetSlaveReady

Description:

This function arms the slave for the next incoming request. This function is typically called just after kxSetSlaveData has been called to set the next outgoing slave data. The adapter automatically disarms the slave after a completed slave operation (either after responding to an I2C address and receiving a STOP, or after all outgoing SPI data is transfer and the Slave-Select pin de-asserts).

Prototype:

KX_BOOL kxSetSlaveReady(KX_GPIOIDX ss);

Arguments:

ss When used to arm the SPI Slave, this value sets the Slave-Select pin used to activate the slave. When used to arm the I2C Slave, this value has no effect. If the SPI Slave is to transfer as soon as SCLK toggles, set *ss* to KX_GPIOIDX_NONE.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSetSlaveReady (KX_GPIOIDX_SS) == KX_TRUE) {
        …
}
```

5.13. kxSetWaitTimeout

Description:

This function sets the maximum allowed time to wait for a GPIO pin state during a SPI or I2C operation (e.g. due to a “/H” token). If this time is exceeded, the operation is aborted and an error is generated.

Prototype:

KX_BOOL kxSetWaitTimeout(KX_TIME timeout);

Arguments:

timeout The maximum allowed delay in seconds. If ≤ 0.0, the timeout is set to infinity. The maximum supported value is 8.388 seconds.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSetWaitTimeout(0.1) == KX_TRUE) {
        …
}
```

5.14. kxAbort

Description:

This function aborts the active operation (SPI or I2C, Master or Slave).

Prototype:

KX_BOOL kxAbort(void);

Arguments:

none

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxAbort() == KX_TRUE) {
        …
}
```

5.15. kxStartTimer

Description:

This function starts a count-down timer. The count-down timer is provided as a convenience and can be used, for example, to detect when an operation is taking too long to complete. Only one timer can be active at a time.

Prototype:

```
void kxStartTimer(KX_TIME time);
```

Arguments:

time The timer will expire after this number of seconds.

Return:

none.

Example:

```
kxStartTimer(5.0);                        // set the timer to expire in 5 seconds.
while(kxTimerExpired() != KX_TRUE) ;      // wait the 5 seconds
```

5.16. kxTimerExpired

Description:

This function indicates whether the count-down timer has expired.

Prototype:

```
KX_BOOL kxTimerExpired(void);
```

Arguments:

none

Return:

Returns KX_TRUE if the timer has expired, KX_FALSE otherwise.

Example:

```
kxStartTimer(5.0);                        // set the timer to expire in 5 seconds.
while(kxTimerExpired() != KX_TRUE) ;      // wait the 5 seconds
```

6. I2C API Functions

6.1. kxI2CsetBitRate

Description:

This function sets the I2C bit-rate for an adapter.

Prototype:

KX_BOOL kxI2CsetBitRate(KX_RATE kbitsPerSec);

Arguments:

kbitsPerSec The bit-rate of the adapter in kbits/second, from 31.25 to 1500 kHz.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSetBitRate(100.0) == KX_TRUE) {
        …
}
```

6.2. kxI2CsetSCLtimeout

Description:

This function sets the I2C SCL low timeout period for an adapter. The I2C interface will automatically perform a kxI2CfreeBus operation if the SCL signal remains low for a period exceeding this timeout value. If the requested time is ≤ 0.0, the timeout feature is disabled (i.e. set to infinity)

Prototype:

KX_BOOL kxI2CsetSCLtimeout(KX_TIME timeout);

Arguments:

timeout The timeout period of the adapter in seconds (up to 0.0327sec).

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxI2CsetSCLtimeout(30e-3) == KX_TRUE) {
        …
}
```

6.3. kxI2CsetConfig

Description:

This function configures the I2C interface by passing a OR'ed collection of KX_I2CCONFIG values.

Prototype:

KX_BOOL kxI2CsetConfig(KX_I2CCONFIG config);

Arguments:

config A set of KX_I2CCONFIG fields indicating which options to enable.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxI2CsetConfig (KX_I2CCONFIG_ABORT_ON_EXPECT |
KX_I2CCONFIG_FREE_BUS_ENABLE) == KX_TRUE) {
        …
}
```

6.4. kxI2CgetConfig

Description:

This function gets the present I2C interface configuration.

Prototype:

KX_BOOL kxI2CgetConfig(KX_I2CCONFIG *config);

Arguments:

config The function will store the I2C configuration as an OR'ed collection of KX_I2CCONFIG flags at this address.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_I2CCONFIG config;

if(kxI2CgetConfig(&config) == KX_TRUE) {
        …
}
```

6.5. kxI2CfreeBus

Description:

This function aborts any active I2C operation and forces the adapter to immediately release the I2C bus.

Prototype:

KX_BOOL kxI2CfreeBus(void);

Arguments:

none

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxI2CfreeBus() == KX_TRUE) {
        …
}
```

6.6. kxI2CsetAddr

Description:

This function sets the list of I2C bus addresses for which the I2C Slave feature will respond. Up to 16 addresses can be enabled using this API function. If the address count is 0, the adapter responds to all addresses (either 7-bit or 10-bit address depending on the value of tenBitAddr). The addresses should be right-justified.

Prototype:

KX_BOOL kxI2CsetAddr(KX_ADDR *addr, KX_COUNT cnt, KX_BOOL tenBitAddr);

Arguments:

addr	A pointer to a list of address values (each address is right justified).
tenBitAddr	Indicates that the addresses are 10-bits wide rather than 7-bits wide. The adapter cannot respond to a mix of 7-bit and 10-bit addresses. This parameter also controls what type of addresses the slave expects when responding to all addresses.
cnt	The number of address values in the list.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_ADDR addr[] = {0x10,0x20,0x21};
…
if(kxI2CsetAddr(addr,sizeof(addr),KX_FALSE) == KX_TRUE) {
        …
}
```

6.7. kxI2Cwrite

Description:

This function initiates an I2C master write operation by an adapter. The bytes stored at *wdata* are written to the given I2C address when the bus becomes free. Note that the function generally returns before the write completes. A call to kxGetStatus should be issued to determine when the write is complete (by checking the KX_STATUS_I2C_MASTER_PENDING bit).

Prototype:

KX_BOOL kxI2Cwrite(KX_ADDR addr, KX_BOOL tenBitAddr, KX_DATA *wdata,
KX_COUNT wcnt);

Arguments:

addr	The slave address to write (right-justified).
tenBitAddr	Indicates that the address is 10-bits wide.
wdata	A pointer to the write data.
wcnt	The number of bytes to write.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure. Note that KX_TRUE indicates that the write was scheduled successfully, not that the write completed successfully. The actual write status is determined by a call to kxGetStatus.

Example:

```
KX_DATA wdata[4];
KX_STATUS status;

if(kxI2Cwrite(0x50,KX_FALSE,wdata,sizeof(wdata)) == KX_TRUE) {
        while(kxGetStatus(&status) == KX_TRUE &&
                (status & KX_STATUS_I2C_MASTER_PENDING));
        …
```

}

6.8. kxI2Cread

Description:

This function initiates an I2C master read operation by an adapter. The requested number of bytes is read from the given I2C address when the bus becomes free. Note that the function generally returns before the read completes. A call to kxGetStatus should be issued to determine when the read is complete. A call to kxGetInData is then used to retrieve the read data when complete.

Prototype:

```
KX_BOOL kxI2Cread(KX_DATA addr, KX_BOOL tenBitAddr, KX_COUNT rcnt);
```

Arguments:

addr	The slave address to read (right justified).
tenBitAddr	Indicates that the address is 10-bits wide.
rcnt	The number of bytes to read.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure. Note that KX_TRUE indicates that the read was scheduled successfully, not that the read completed successfully.

Example:

```
KX_STATUS status;
KX_DATA *rdata;
KX_COUNT rcnt;

if(kxI2Cread(0x50,KX_FALSE,8) == KX_TRUE) {
        while(kxGetStatus(&status) == KX_TRUE &&
                (status & KX_STATUS_I2C_MASTER_PENDING));
        if(kxGetInData(&rdata,&rcnt) == KX_TRUE){
                …
        }
}
```

6.9. kxI2Cexecute

Description:

This function initiates an I2C script execution by an adapter. Note that the function generally returns before the execution completes. A call to kxGetStatus should be issued to determine when the execution is complete. When complete, a call to kxGetInData can be used to obtain any incoming data generated by the script.

Prototype:

```
KX_BOOL kxI2Cexecute(KX_I2CEXESTR *estr);
```

Arguments:

estr	A pointer to the script to execute.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure. Note that KX_TRUE indicates that the execution was scheduled successfully, not that the execution

completed successfully. A call to kxGetStatus should be made to determine the status of the execution.

Example:

```
KX_I2CEXESTR *estr = "/S 80 55 /S 81 /R 0008 /P";
if(kxI2Cexecute(estr) == KX_TRUE) {
        while(kxGetStatus(&status) == KX_TRUE &&
                (status & KX_STATUS_I2C_MASTER_PENDING));
        …
}
```

6.10. kxI2CgetAddrIndices

Description:

When operating as an I2C slave device, an adapter stores all incoming bytes while addressed, including the addresses it responded to. This function provides a list of index values. These index values identify which locations in the data array provided by kxGetInData contain the addresses. For example, if an index value is 2, the value at data[2] is the MSB of the incoming address the slave responded to. If the slave is operating in 10-bit address mode, the next byte holds the LSB of the address. Otherwise the address only occurs one byte in the array.

Prototype:

```
KX_BOOL kxI2CgetAddrIndices(KX_COUNT **indices, KX_COUNT *cnt);
```

Arguments:

indices	The function sets this pointer to the location of a static array holding the address indices. This value should not be passed to free().
cnt	The function stores the number of address index values at this location.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_COUNT *indices
KX_COUNT cnt;
…
if(kxI2CgetAddrIndices(&indices,&cnt) == KX_TRUE) {
        …
}
```

7. SPI API Functions

7.1. kxSPIsetBitRate

Description:

This function sets the SPI bit-rate for an adapter during master transactions. The bit rate does not need to be set for slave operations.

Prototype:

```
KX_BOOL kxSPIsetBitRate(KX_RATE kbitsPerSec);
```

Arguments:

rate	The bit-rate of the adapter in kbits/second, ranging from 93.75kHz to 24MHz.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSetBitRate(1000.0) == KX_TRUE) {
        …
}
```

7.2. kxSPIsetTiming

Description:

This function sets the minimum delay between asserting the SPI Slave-Select signal and generating the bit period (TSU) and the delay between the last bit period and de-asserting the slave-select signal (THD). This function also sets the minimum time the Slave-Select signal must remain de-asserted between SPI transfers (TMIN).

Prototype:

```
KX_BOOL kxSPIsetTiming(KX_TIME tmin, KX_TIME tsu, KX_TIME thd);
```

Arguments:

tmin	The requested TMIN in seconds, from 2e-6 to 55e-6.
tsu	The requested TSU in seconds, from 2e-6 to 55e-6.
thd	The requested THD in seconds, from 2e-6 to 55e-6.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSPIsetTiming(5e-6,5e-6,20e-6) == KX_TRUE) {
        …
}
```

7.3. kxSPIsetConfig

Description:

This function configures the SPI operations.

Prototype:

```
KX_BOOL kxSPIsetConfig(KX_SPICONFIG config, KX_GPIOIDX ss,
                       KX_COUNT bytesPerSS);
```

Arguments:

config	The requested SPI mode provided as an OR'd collection of KX_SPICONFIG values.
ss	The Slave-Select pin to use during master operations. If set to KX_GPIOIDX_NONE, a master will not assert a Slave-Select pin.
bytesPerSS	The number of bytes to transmit per Slave-Select cycle. A master will de-assert and re-assert the Slave-Select pin after transferring this number of bytes.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxSPIsetConfig(KX_SPICONFIG_SS_ACTIVE_HIGH|KX_SPICONFIG_LSB_FIRST,
        KX_GPIOIDX_SS,8) == KX_TRUE) {
        …
}
```

7.4. kxSPIgetConfig

Description:

This function provides the present configure for SPI operations.

Prototype:

```
KX_BOOL kxSPIgetConfig(KX_SPICONFIG *config, KX_GPIOIDX *ss,
                KX_COUNT *bytesPerSS);
```

Arguments:

config	The function stores the SPI mode at this location as an OR'd collection of KX_SPICONFIG values.
ss	The function stores the Slave-Select pin to use during master operations at this location (see KX_GPIOIDX).
bytesPerSS	The function stores the number of bytes to transmit per Slave-Select cycle at this location.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_SPICONFIG config;
KX_GPIOIDX ss;
KX_COUNT bytesPerSS;

if(kxSPIgetConfig(&config,&ss,&bytesPerSS) == KX_TRUE) {
        …
}
```

7.5. kxSPItransfer

Description:

This function initiates a SPI transfer by an adapter. The bytes stored at *wdata* are shifted out the MOSI pin. The data received on the MISO pin is shifted into the adapter's SPI buffer. This data can be retrieved by a call to the kxGetInData function. Note that the function generally returns before the transfer completes. A call to kxGetStatus should be issued to determine when the transfer is complete (by checking the KX_STATUS_SPI_MASTER_PENDING flag).

Prototype:

```
KX_BOOL kxSPItransfer(KX_DATA *wdata, KX_COUNT wcnt);
```

Arguments:

wdata	A pointer to the MOSI data.
wcnt	The number of bytes to transfer.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure. Note that KX_TRUE

indicates that the transfer was scheduled successfully, not that the transfer completed successfully. The actual transfer status is determined by a call to kxGetStatus.

Example:

```
KX_DATA wdata[4];
KX_STATUS status;
…
if(kxSPItransfer(wdata,sizeof(wdata)) == KX_TRUE) {
        while(kxGetStatus(&status) == KX_TRUE &&
                (status & KX_STATUS_SPI_MASTER_PENDING));
        …
}
```

7.6. kxSPIexecute

Description:

This function initiates a SPI master script execution by an adapter. Note that the function generally returns before the execution completes. A call to kxGetStatus should be issued to determine when the execution is complete (by checking the KX_STATUS_SPI_MASTER_PENDING flag).

Prototype:

KX_BOOL kxSPIexecute(KX_SPIEXESTR *estr);

Arguments:

estr A pointer to the SPI script to execute (see KX_SPIEXESTR).

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure. Note that KX_TRUE indicates that the execution was scheduled successfully, not that the execution completed successfully. A call to kxGetStatus should be made to determine the status of the execution.

Example:

```
KX_SPIEXESTR *estr = "/1 G1 80 55 81 01 00 /0 G1";
if(kxSPIexecute(estr) == KX_TRUE) {
        while(kxGetStatus(&status) == KX_TRUE &&
                (status & KX_STATUS_SPI_MASTER_PENDING));
        …
}
```

8. GPIO API Functions

8.1. kxGPIOsetDir

Description:

This function sets the input/output direction of adapter pins not used by an enabled I2C or SPI feature.

Prototype:

KX_BOOL kxGPIOsetDir(KX_GPIOMASK outs);

Arguments:

outs Pins indicated by this value are set as outputs, otherwise the pins are inputs.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxGPIOsetDir(KX_GPIOMASK_SS | KX_GPIOMASK_GP1) == KX_TRUE) {
        …
}
```

8.2. kxGPIOsetState

Description:

This function sets the output state of various pins for an adapter. If a pins is presently configured as an output, the change will occur immediately. Otherwise, the requested output state will not take effect until the signal is set as an output using the kxGPIOsetDir function. Signals presently used by an enabled feature are not affected until that feature is disabled.

Prototype:

```
KX_BOOL kxGPIOsetState(KX_GPIOMASK states);
```

Arguments:

states Pins indicated by this value are driven HIGH when configured as an output.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
if(kxGPIOsetState(KX_GPIOMASK_SS | KX_GPIOMASK_GP1) == KX_TRUE) {
        …
}
```

8.3. kxGPIOgetStatus

Description:

This function returns the present state and direction of all pins as sets of bit masks using the KX_GPIOMASK data type. The function returns the actual sensed value on the pin regardless of whether the pin is configured as an input or output or used by another feature.

Prototype:

```
KX_BOOL kxGPIOgetStatus(KX_GPIOMASK *state, KX_GPIOMASK *dir,
        KX_GPIOMASK *sensed);
```

Arguments:

state The function stores the GPIO outgoing state values (set by kxGPIOsetState or scripts) at the indicated location. These are the values the adapter will drive when each pin is configured as an output and not used by an enabled feature.

dir The function stores the GPIO directions at the indicated location. Signals whose direction is output have their associated bit set to a 1.

sensed The function stores the GPIO sensed input values at the indicated location. The sensed value is the value sensed at the pin regardless of whether the GPIO is an input or output.

Return:

Returns KX_TRUE is successful, KX_FALSE otherwise. If KX_FALSE is returned, a call to kxGetError can be used to provide the cause of the failure.

Example:

```
KX_GPIOMASK states,dir,sensed;

if(kxGPIOgetState(&state,&dir,&sensed) == KX_TRUE && (state &
KX_GPIOMASK_GP1)){
        …
}
```

9. Examples

The following sections provide code segments which show how to use most features of the USB-910H API. This code segments include comments where additional error handling would be required in a real application.

9.1. Finding and Opening a Specific Adapter as an I2C Master

```
//-----------------------------------------------------------------
// Find and open adapter “000100” and enable as an I2C Master with
// the internal pull-ups enabled.
//
KX_STRING *str;

// find all available adapters
KX_ADAPTER devCount = kxFindAdapters();

// search for an adapter with serial string “000100”
for(KX_ADAPTER i = 0 ; i < devCount ; i++){
      if(kxGetSerialString(i,&str) == KX_TRUE &&
            !strcmp(str,”000100”)) break;
}

// if found, try to open the adapter
if(i == devCount || kxOpenAdapter(i) != KX_TRUE){
      // we failed to find or open the adapter!
}

// enable the I2C master mode with pullups
else if(kxEnableFeatures(KX_FEATURE_I2CMST| KX_FEATURE_I2CRPU) != KX_TRUE){
      // failed to enable the desired features!
}
```

9.2. Performing an I2C Master Write Operation

```
//-----------------------------------------------------------------
// Write {0x00,0x01,0x02,0x03} to address 0x20.
// Assumes an adapter is opened and the I2C Master
// feature has been enabled.

KX_STATUS status;
KX_DATA wdata[] = {0x00,0x01,0x02,0x03};

// set the bit-rate to 100kbits/sec
```

```
if(kxI2CsetBitRate(100.0) != KX_TRUE){
      // failed to set bit-rate
}

// configure the I2C to:
//    detect a free-bus condition
//    retry if it loses arbitration
if(kxI2CsetConfig(KX_I2CCONFIG_FREE_BUS_ENABLE |
                  KX_I2CCONFIG_RETRY_ON_LOST_ARB) != KX_TRUE){
      // failed to set I2C configuration
}

// initiate the write operation
if(kxI2Cwrite(0x20,KX_FALSE,wdata,sizeof(wdata)) == KX_TRUE){

      // don't wait more than 5 seconds for the operation to complete
      kxStartTimer(5.0);

      do {
            // are we taking too long?
            if(kxTimerExpired() == KX_TRUE){
                  // operation is taking too long!
                  kxAbort();
                  exit(-1);
            }

            // get the adapter status value
            if(kxGetStatus(&status) != KX_TRUE){
                  // failed to get the status!
                  exit(-1);
            }
      } while(status & KX_STATUS_I2C_MASTER_PENDING);
}
else {
      // the write failed!
}
```

9.3. Performing an I2C Master Read Operation

```
//-----------------------------------------------------------------
// Read four bytes from address 0x20.
// Assumes an adapter is opened and the I2C Master
// feature has been enabled.

KX_DATA *rdata;
KX_COUNT *count;
KX_STATUS status;

// set the I2C bit-rate
if(kxI2CsetBitRate(100.0) != KX_TRUE){
      // failed to set bit-rate
}

// configure the I2C to:
//    detect a free-bus condition
//    retry if it loses arbitration
if(kxI2CsetConfig(KX_I2CCONFIG_FREE_BUS_ENABLE |
```

```
                  KX_I2CCONFIG_RETRY_ON_LOST_ARB) != KX_TRUE){
      // failed to set I2C configuration
}

// initiate a read of 4 bytes
if(kxI2Cread(0x20,KX_FALSE,4) == KX_TRUE){

      // don't wait more than 5 seconds for the operation to complete
      kxStartTimer(5.0);

      do {
            // are we taking too long?
            if(kxTimerExpired() == KX_TRUE){
                  // operation is taking too long!
                  kxAbort();
                  exit(-1);
            }

            // get the adapter status value
            if(kxGetStatus(&status) != KX_TRUE){
                  // failed to get the status!
                  exit(-1);
            }
      } while(status & KX_STATUS_I2C_MASTER_PENDING);

      // retrieve the read data
      if(kxGetInData(&rdata,&count) != KX_TRUE){
            // data retrieval failed
      }
}
else {
      // the write failed!
}
```

9.4. Performing an I2C Master Write/Read Operation

```
//----------------------------------------------------------------
// Write {0x00,0x01,0x02,0x03} to address 0x20, then
// read four bytes back from address 0x20 after a repeated start.
// Assumes an adapter is opened and the I2C Master
// feature has been enabled.

KX_DATA *rdata;
KX_COUNT *count;
KX_STATUS status;

// set the bit-rate
if(kxI2CsetBitRate(100.0) != KX_TRUE){
      // failed to set bit-rate
}

// configure the I2C to:
//    detect a free-bus condition
//    retry if it loses arbitration
if(kxI2CsetConfig(KX_I2CCONFIG_FREE_BUS_ENABLE |
                  KX_I2CCONFIG_RETRY_ON_LOST_ARB) != KX_TRUE){
      // failed to set I2C configuration
```

```
}

// Execute a script to perform the required operations.
// Address 0x20 becomes 0x40 when left-justified.
if(kxI2Cexecute("/S4000010203/S41/R0004/P") == KX_TRUE){

	// don't wait more than 5 seconds for the operation to complete
	kxStartTimer(5.0);

	do {
		// are we taking too long?
		if(kxTimerExpired() == KX_TRUE){
			// operation is taking too long!
			kxAbort();
			exit(-1);
		}

		// get the adapter status value
		if(kxGetStatus(&status) != KX_TRUE){
			// failed to get the status!
			exit(-1);
		}
	} while(status & KX_STATUS_I2C_MASTER_PENDING);

	// retrieve the read data
	if(kxGetInData(&rdata,&count) != KX_TRUE){
		// data retrieval failed
	}
}
else {
	// the write failed!
}
```

9.5. Performing an I2C Slave Write/Read Operation

```
//-----------------------------------------------------------------
// Arm the Adapter as an I2C slave prepared to send (i.e. respond
// to a read request) of 4 bytes.  The Slave will respond to address
// 0x20 and 0x40.
// Assumes an adapter is opened and the I2C Slave feature has
// been enabled.

KX_DATA wdata[]   = {0x00,0x01,0x02,0x03};
KX_ADDR addr[]    = {0x20,0x40};
KX_DATA *rdata;
KX_COUNT *count;
KX_STATUS status;

// only used by the free-bus detector
if(kxI2CsetBitRate(100.0) != KX_TRUE){
	// failed to set bit-rate
}

// configure the I2C to detect a free-bus condition
if(kxI2CsetConfig(KX_I2CCONFIG_FREE_BUS_ENABLE) != KX_TRUE){
	// failed to set I2C configuration
}
```

```
// set the outgoing slave data
if(kxSetSlaveData(wdata,sizeof(wdata)) != KX_TRUE){
	// failed to set slave data!
}

// set the slave addresses (7-bit)
if(kxI2CsetAddr(addr,sizeof(addr),KX_FALSE) != KX_TRUE){
	// failed to set slave addresses!
}

// Arm the slave and wait for the operation to complete.
// For I2C, the parameter value passed to kxSetSlaveReady is not used.
if(kxSetSlaveReady(0) == KX_TRUE){

	// don't wait more than 5 seconds for the operation to complete
	kxStartTimer(5.0);

	do {
		// are we taking too long?
		if(kxTimerExpired() == KX_TRUE){
			// operation is taking too long!
			kxAbort();
			exit(-1);
		}

		// get the adapter status value
		if(kxGetStatus(&status) != KX_TRUE){
			// failed to get the status!
			exit(-1);
		}
	} while(status & KX_STATUS_I2C_SLAVE_PENDING);

	// retrieve any write (i.e. incoming) data
	if(kxGetInData(&rdata,&count) != KX_TRUE){
		// data retrieval failed
	}
}
else {
	// the operation failed!
}
```

9.6. Performing a SPI Master Transfer

```
//-----------------------------------------------------------------
// Transfer {0x00,0x01,0x02,0x03} to a SPI slave using Slave-Select
// pin SS.
// Assumes an adapter is opened and the SPI Master
// feature has been enabled.

KX_DATA *rdata;
KX_COUNT *count;
KX_STATUS status;

// set the SPI bit-rate to 1Mbit/sec
if(kxSPIsetBitRate(1000.0) != KX_TRUE){
	// failed to set bit-rate
```

```
}
// set the TMIN, TSU, and THD timing in seconds
if(kxSPIsetTiming(5e-6,5e-6,5e-6) != KX_TRUE){
      // failed to set SPI timing
}

// configure the SPI to:
//    Assert the Slave-Select active-high.
//    Use SS as the Slave-Select pin.
//    Transfer 2 bytes per SS assertion.
if(kxSPIsetConfig(KX_SPICONFIG_SS_ACTIVE_HIGH,KX_GPIOIDX_SS,2) != KX_TRUE){
      // failed to set SPI configuration
}

// Initiate the transfer.
if(kxSPItransfer(wdata,sizeof(wdata)) == KX_TRUE){

      // don't wait more than 5 seconds for the operation to complete
      kxStartTimer(5.0);

      do {
            // are we taking too long?
            if(kxTimerExpired() == KX_TRUE){
                  // operation is taking too long!
                  kxAbort();
                  exit(-1);
            }

            // get the adapter status value
            if(kxGetStatus(&status) != KX_TRUE){
                  // failed to get the status!
                  exit(-1);
            }
      } while(status & KX_STATUS_SPI_MASTER_PENDING);

      // retrieve the incoming data
      if(kxGetInData(&rdata,&count) != KX_TRUE){
            // data retrieval failed
      }
}
else {
      // the transfer failed!
}
```

9.7. Performing a SPI Master Script operation

```
//-----------------------------------------------------------------
// Use a SPI Script to perform the following:
//    Wait for pin GP1 to go HIGH.
//    Transfer {0x00,0x01,0x02,0x03} to a SPI slave.
//    Drive pin GP2 HIGH to 128usec.
//
// Assumes an adapter is opened and the SPI Master
// feature has been enabled.

KX_DATA *rdata;
KX_COUNT *count;
```

```
KX_STATUS status;

// set the bit-rate to 1Mbit/sec
if(kxSPIsetBitRate(1000.0) != KX_TRUE){
      // failed to set bit-rate
}

// set the TMIN, TSU, and THD timing in seconds
if(kxSPIsetTiming(5e-6,5e-6,5e-6) != KX_TRUE){
      // failed to set SPI timing
}

// configure the SPI to:
//    Assert the Slave-Select active-high.
//    Use SS as the Slave-Select pin.
//    Transfer 2 bytes per SS assertion.
if(kxSPIsetConfig(KX_SPICONFIG_SS_ACTIVE_HIGH,KX_GPIOIDX_SS,2) != KX_TRUE){
      // failed to set SPI configuration
}

// Initiate the operation.
if(kxSPIexecute("/HG1 00010203 /1G2 /D0080 /0G2") == KX_TRUE){

      // don't wait more than 5 seconds for the operation to complete
      kxStartTimer(5.0);

      do {
            // are we taking too long?
            if(kxTimerExpired() == KX_TRUE){
                  // operation is taking too long!
                  kxAbort();
                  exit(-1);
            }

            // get the adapter status value
            if(kxGetStatus(&status) != KX_TRUE){
                  // failed to get the status!
                  exit(-1);
            }
      } while(status & KX_STATUS_SPI_MASTER_PENDING);

      // retrieve the incoming data
      if(kxGetInData(&rdata,&count) != KX_TRUE){
            // data retrieval failed
      }
}
else {
      // the transfer failed!
}
```

9.8. Performing a SPI Slave Operation

```
//-----------------------------------------------------------------
// Arm the Adapter as a SPI slave prepared to transfer 4 bytes.
// The Slave will respond to Slave-Select pin GP1.
// Assumes an adapter is opened and the SPI Slave feature has
// been enabled.
```


KETEREX

```
KX_DATA wdata[]   = {0x00,0x01,0x02,0x03};
KX_DATA *rdata;
KX_COUNT *count;
KX_STATUS status;

// configure the SPI to:
//    Assert the Slave-Select active-high.
// The other values only affect Master operations.
if(kxSPIsetConfig(KX_SPICONFIG_SS_ACTIVE_HIGH,KX_GPIOIDX_SS,2) != KX_TRUE){
      // failed to set SPI configuration
}

// set the outgoing slave data
if(kxSetSlaveData(wdata,sizeof(wdata)) != KX_TRUE){
      // failed to set slave data!
}

// Arm the slave with GP1 as the Slave-Select and wait for the
// operation to complete
if(kxSetSlaveReady(KX_GPIOIDX_GP1) == KX_TRUE){

      // don't wait more than 5 seconds for the operation to complete
      kxStartTimer(5.0);

      do {
            // are we taking too long?
            if(kxTimerExpired() == KX_TRUE){
                  // operation is taking too long!
                  kxAbort();
                  exit(-1);
            }

            // get the adapter status value
            if(kxGetStatus(&status) != KX_TRUE){
                  // failed to get the status!
                  exit(-1);
            }
      } while(status & KX_STATUS_I2C_SLAVE_PENDING);

      // retrieve any write (i.e. incoming) data
      if(kxGetInData(&rdata,&count) != KX_TRUE){
            // data retrieval failed
      }
}
else {
      // the operation failed!
}
```

9.9. Use General-Purpose I/O to Write Bytes to a Port

```
//-----------------------------------------------------------------
// Use the general-purpose I/O to write bytes to a port.
// GP3 is used as a write strobe.  All other pins are used to
// hold the write byte.

// declare the list of data to write
```

```
unsigned char data[] = {0x11,0x22,0x33,0x44};

// set all GPIO as outputs
if(kxGPIOsetDir(  KX_GPIOMASK_SCLK|KX_GPIOMASK_MISO|KX_GPIO_MOSI|
                  KX_GPIOMASK_SS|KX_GPIOMASK_SDA|KX_GPIOMASK_SCL|
                  KX_GPIOMASK_GP1|KX_GPIOMASK_GP2|KX_GPIOMASK_GP3) !=
            KX_TRUE){
      // failed to set GPIO state
}

// start with all pins low
if(kxGPIOsetState(0) != KX_TRUE){
      // failed to set GPIO state
}

// for each data byte to write
for(int i = 0 ; i < sizeof(data) ; i++){

      // set the data to write and set GP3 HIGH
      if(kxGPIOsetState(data[i] | KX_GPIOMASK_GP3) != KX_TRUE){
            // failed to set GPIO state
      }

      // drive GP3 back low
      if(kxGPIOsetState(data[i]) != KX_TRUE){
            // failed to set GPIO state
      }
}
```

10. Contact Information

Keterex, Inc.

7320 N. Mo Pac Expressway

Suite 201

Austin, Texas 78731

Tel: 512-346-8800

www.keterex.com

Email: support@keterex.com

Information furnished by Keterex, Inc. in this document is believed to be accurate and reliable. However, no responsibility is assumed for its use. Information in this document is subject to change without notice.

Trademarks: The Keterex name and logo are registered trademarks of Keterex Incorporated. Other products or brand names mentioned herein are trademarks or registered trademarks of their respective holders.

Use in life support systems: Keterex adapters are not designed for use in life support and/or safety equipment where malfunction of the product can result in personal injury or death. Your use or sale of these adapters for use in life support and/or safety applications is at your own risk. You agree to defend and hold us harmless from any and all damages, claims, suits or expenses resulting from such use.

Limitation of Liability: You acknowledge and agree that, in no event, shall Keterex be liable, whether in contract, warranty, tort (including negligence or breach of statutory duty), strict liability, indemnity, contribution, or otherwise, for any indirect, special, punitive, exemplary, incidental or consequential loss, damage, cost or expense of any kind whatsoever, howsoever caused, or for any loss of production, cost of procurement of substitute goods, loss of capital, loss of software, loss of profit, loss of revenues, contracts, business, cost of rework, loss of goodwill or anticipated savings, or wasted management time, even if Keterex has been advised of the possibility or they are foreseeable. The total liability of Keterex on all claims, whether in contract, warranty, tort (including negligence or breach of statutory duty), strict liability, indemnity, contribution, or otherwise, shall not exceed the purchase price of these adapters