Smart ETK Android API Documentation

Release 0.0.19

VIA Technologies

October 29, 2015

CHAPTER

ONE

SMART ETK API GENERAL NOTES

VIA Smart ETK SDK supports the hardware controlling API for GPIO, Watch Dog, and UART (RS-232) modules.

Smart ETK is programmed with the socket IO as the communication between JAVA and C language to control the hardware modules. We implemented the board support service such as `bss_vab820` to meet the request from Smart ETK API.

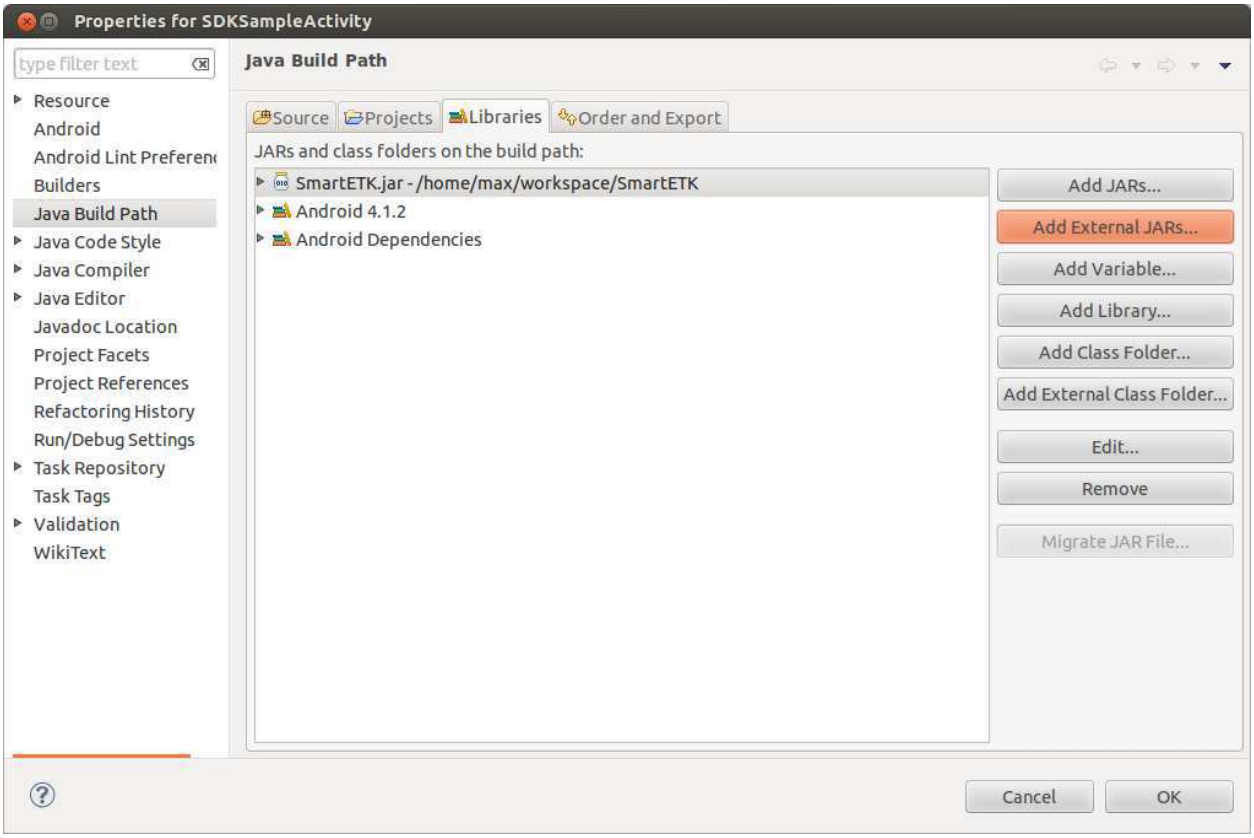
1.1 Compatibility

Model	GPIO	WDT	RTC	WOL	RES	UART	SUS	CEC	I2C	CAN	UPC	DPMS
VAB-820	✓	✓	×	×	×	✓	×	×	×	✓	×	×
VAB-1000	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	×
ALTA DS 2	×	✓	✓	✓	✓	×	✓	×	×	×	×	×
AMOS-820	✓	✓	×	×	×	✓	×	×	×	✓	×	×
ARTiGO A900	✓	✓	✓	✓	✓	×	✓	×	×	×	×	×
Viega	×	✓	×	×	×	×	×	×	×	×	✓	×

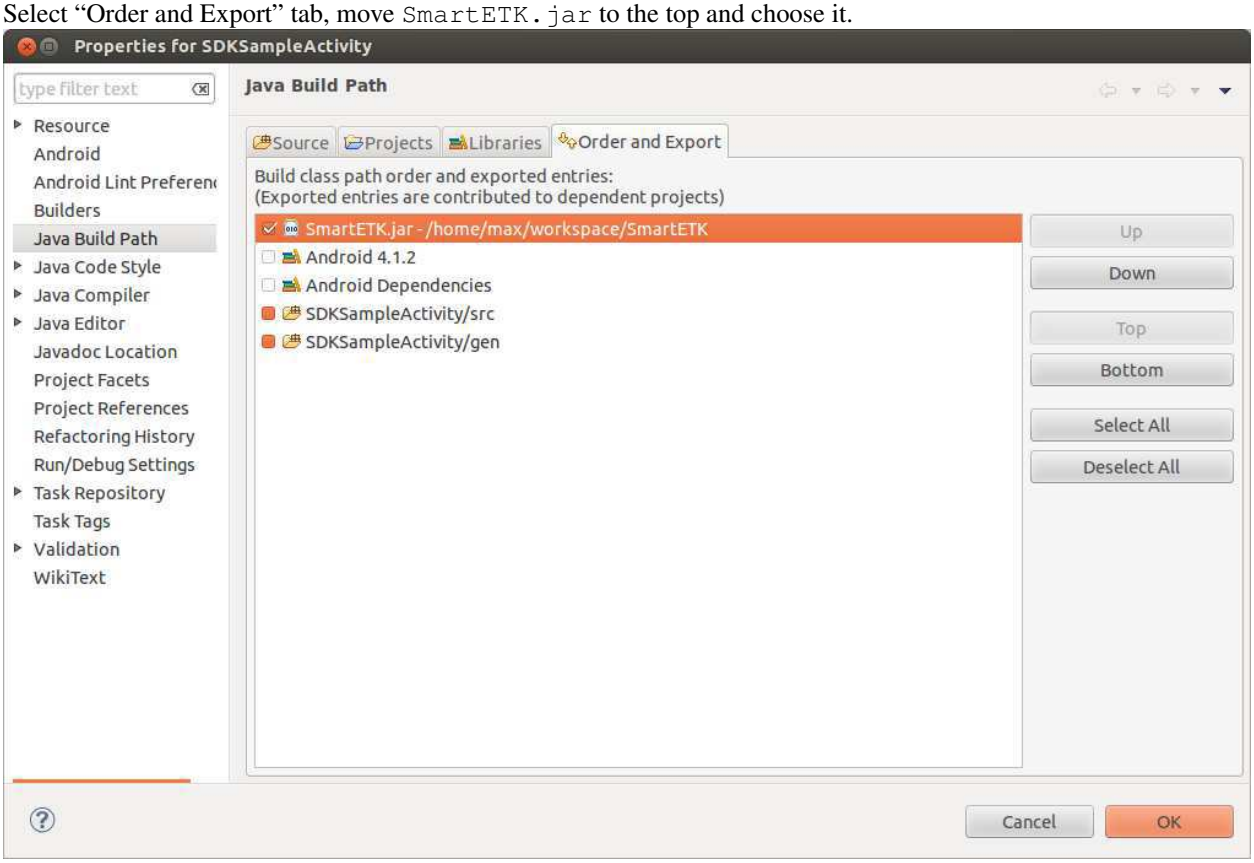
Legend: `GPIO`: *GPIO* support, `WDT`: *WatchDog* timer, `RTC` is *Real-Time Clock Wake-up*, `WOL`: *Wake-on-LAN*, `RES`: `Restart` support, `UART`: *UART* support, `SUS`: `Suspend` support, `CEC`: HDMI CEC support, `CAN`: *CAN* support, `I2C`: *I2C* support, `UPC`: x, `DPMS`: *DPMS* support

1.2 Installation

Open Eclipse IDE and create an Android project. In project properties, import SmartETK.jar by pressing the button "Add External JARs...".



Select "Order and Export" tab, move `SmartETK.jar` to the top and choose it.



1.3 Permissions

Smart ETK is programmed with the socket IO as the communication between JAVA and C language to control the hardware modules, therefore you need to make sure that you have android.permission.INTERNET in AndroidManifest.xml:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

CHAPTER

TWO

SMARTETK CLASS

This class contains general parts used by the rest of the package, such as function return values, and some helper classes.

class **SmartETK**
Helper classes:

class **Timeout**
Timeout configuration, used by *Can.getTimeout* and *Uart.getTimeout*.

boolean **Enable**
Enable or disable timeout function

int **Timeout**
timeout value in multiples of 0.1 seconds, accepted range is 0 – 255 (0 - 25.5 seconds)

2.1 Function Return Values

The return values and error codes used by the methods in Smart ETK.

static int **S_OK**
When a function returns the S_OK value, it indicates that the function has successfully completed.

static int **E_FAIL**
When a function returns the E_FAIL value, it indicates that the function has failed to complete.

static int **E_VERSION_NOT_SUPPORT**
When a function returns the E_VERSION_NOT_SUPPORT value, it indicates that the versions of `SmartETK.jar` and bsservice are not compatible.

static int **E_INVALID_ARG**
When a function returns the E_INVALID_ARG value, it indicates that the arguments are invalid.

static int **E_FUNC_NOT_SUPPORT**
When a function returns the E_FUNC_NOT_SUPPORT value, it indicates that the function is not supported by this board.

static int **E_CONNECTION_FAIL**
When a function returns the E_CONNECTION_FAIL value, it indicates that the `bsservice` doesn't respond the request. Please make sure bsservice is running successfully.

static int **E_NOT_RESPOND_YET**
When a function returns the E_NOT_RESPOND_YET value, it indicates that the `bsservice` function is still running and has not finished yet.

static int **E_TIMEOUT**
When a function returns the E_TIMEOUT value, it indicates that no corresponding data has been received within the period.

static int **E_UART_OPENFAIL**
When *Uart.open* returns the E_UART_OPENFAIL value, it indicates that the UART device can't be opened successfully. Please make sure the name of the tty device exists.

static int **E_UART_NOT_OPEN**
When a function returns the E_UART_NOT_OPEN value, it indicates that uart object cannot be operated normally. The reason might be that the application doesn't open uart device before calling other operating function; or it was reset by other uart object.

static int **E_UART_ALREADY_OPENED**
When *Uart.open* returns the E_UART_ALREADY_OPENED value, it indicates that the uart object has been opened. If you need to open other uart device, please call close function to close the current device, then open the other UART again.

static int **E_UART_TTY_BEEN_USED**
When *Uart.open* returns the E_UART_TTY_BEEN_USED value, it indicates that the tty device has been used by other uart object. If you want to use it, you can call reset function to release the resource and open it again.

static int **E_UART_BAUDRATE_NOT_SUPPORT**
When *Uart.setConfig* returns the E_UART_BAUDRATE_NOT_SUPPORT value, it indicates that baud rate is not supported.

static int **E_CAN_OPENFAIL**
When *Can.open* returns the E_CAN_OPENFAIL value, it indicates that the CAN device can't be opened successfully. Please make sure the name of the CAN device exists.

static int **E_CAN_NOT_OPEN**
When a function returns the E_CAN_NOT_OPEN value, it indicates that can object cannot be operated normally. The reason might be that the application doesn't open can device before calling other operating function.

static int **E_CAN_ALREADY_OPENED**
When *Can.open* returns the E_CAN_ALREADY_OPENED value, it indicates that the can object has been opened. If you need to open other can device, please call close function to close the current device, then open the other can again.

static int **E_CAN_BAUDRATE_NOT_SUPPORT**
When *Can.setBitrate* returns the E_CAN_BAUDRATE_NOT_SUPPORT value, it indicates that bit rate is not supported.

CHAPTER

THREE

NETWORK CLASS

class **Network**
Create a new Network object.

```
Network m_network = new Network();
```

int **setWakeOnLan** (boolean *enable*)

Enable or disable Network Wake-on-LAN function from suspend mode.

Parameters

- **enable** (*boolean*) – enable or disable functionality

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

int **getWakeOnLan** (boolean[] *enable*)
Get the status if Network Wake-on-LAN function.

Parameters

- **enable** (*boolean[]*) – variable to update with `true` for enabled, `false` for disabled

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

3.1 Network Code Examples

3.1.1 Set Wake-on-LAN From Suspend Mode

```
boolean bSetEnable = true;

if(null == m_network) {
  m_network = new Network();
}
if(SmartETK.S_OK != m_network.setWakeOnLan(bSetEnable)) {
  return false;
}
```

3.1.2 Get Wake-on-LAN From Suspend Mode Status

```
if(null == m_network) {
  m_network = new Network();
}
boolean[] bGetEnable = new boolean[1];
if(SmartETK.S_OK != m_network.getWakeOnLan(bGetEnable)) {
  return false;
}
return bGetEnable[0];
```

CHAPTER

FOUR

GPIO CLASS

class **GPIO**

Create a new GPIO object with specified pin ID. Ex: 1, 2, 4, 5, 7, 8, 9, 16.

Parameters

- **int pinID** – GPIO's pin ID.

```
GPIO gpio5 = new GPIO(5);
```

int **setEnable**(boolean *enable*)

Enable the specific GPIO pin.

Parameters

- **enable** (*boolean*) – `true` for enable, `false` for disable

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

static int **GM_GPI**

Indicates "input" direction for GPIO pin.

static int **GM_GPO**

Indicates "output" direction for GPIO pin.

int **setDirection**(int *direction*)

Set input/output direction for the specific GPIO pin.

Parameters

- **direction** (*int*) – *GM_GPI* for input direction, *GM_GPO* for output direction.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getDirection**(int[] *direction*)

Get direction state of the specific GPIO Pin.

Parameters

- **direction** (*int[]*) – parameter to set to *GM_GPI* for input, or *GM_GPO* for output depending on the pin's direction

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **setValue**(int *value*)

Set output signal for the specific GPIO Pin.

Parameters

- **value** (*int*) – GPIO signal, *0* for logic low, *1* for logic high.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

int **getValue** (int[] *value*)
Get input signal of the specific GPIO Pin.

Parameters

- **value** (*int[]*) – GPIO signal, return *0* for logic low, return *1* for logic high.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

4.1 GPIO Code Examples

GPIO1, GPIO2, GPIO4, GPIO5, GPIO7, GPIO8, GPIO9 and GPIO203 are the external GPIO pins for user's own design. An example of setting GPIO1 as input pin and getting its value is shown here.

```
/* Declare variables to get GPIO5 values */
boolean[] bEnable = new boolean[1];
int[] nDirection = new int[1];
int[] nValue = new int[1];

GPIO gpio5 = new GPIO(1); // Create GPIO1 object

gpio5.setEnable(true); // Enable GPIO1
gpio5.setDirection(GPIO.GM_GPI); // Set GPIO1 as input direction
gpio5.getEnable(bEnable); // Get GPIO1's enable status
gpio5.getDirection(nDirection); // Get GPIO1's input/outputdirection
gpio5.getValue(nValue); // Get GPIO1's input value
```

An example of setting GPIO5 as output pin and changing its value is shown here.

```
/* Declare variables to get GPIO6 values */
boolean[] bEnable = new boolean[1];
int[] nDirection = new int[1];
int[] nValue = new int[1];
GPIO gpio6 = new GPIO(5); // Create GPIO5 object

gpio6.setEnable(true); // Enable GPIO5
gpio6.setDirection(GPIO.GM_GPO); // Set GPIO5 as output direction
gpio6.setValue(1); // Set GPIO5's output to high
gpio6.getEnable(bEnable); // Get GPIO5's enable status
gpio6.getDirection(nDirection); // Get GPIO5's input/output direction
gpio6.getValue(nValue); // Get GPIO5's output value
```

Note: Create GPIO203 by following method:

```
GPIO gpio203 = new GPIO(16);
```

CHAPTER

FIVE

RTC CLASS

class **RTC**

Create a new RTC (real-time clock) object.

```
RTC m_rtc = new RTC();
```

static byte **ARG_RTC_MODE_DAY**

Waking up every day.

static byte **ARG_RTC_MODE_WEEK**

Waking up every week.

static byte **ARG_RTC_MODE_MONTH**

Waking up every month.

class **RTCStatus**

RTC wake up time object

byte **Mode**

wake up mode, one of *ARG_RTC_MODE_DAY*, *ARG_RTC_MODE_WEEK*, or *ARG_RTC_MODE_MONTH*.

int **Year**

year of wake up time, counted from 1900, for example 2015 is *iYear = 115*

byte **Month**

month of wake up time, between 1 and 12 accepted

byte **Day**

day of the month for wake up time, between 1 and 31

byte **Hour**

hours for wake up time (24h clock), between 0 and 23

byte **Min**

minutes for wake up time, between 0 and 59

byte **Sec**

seconds for wake up time, between 0 and 59

int **setWakeUpTime**(byte *Mode*, int *Year*, byte *Month*, byte *Day*, byte *Hour*, byte *Min*, byte *Sec*)

Set the wake up time and mode in RTC. The behavior of wake up from suspend mode will start at the wake up time, and it must loop according to the wake up mode.

Parameters

- **Mode** (*byte*) – wake up mode, one of *ARG_RTC_MODE_DAY*, *ARG_RTC_MODE_WEEK*, or *ARG_RTC_MODE_MONTH*.

- **Year** (*int*) – year of wake up time, counted since 1900 for wake up, for example 2015 is *iYear = 115* (???)
- **Month** (*byte*) – month of wake up time, between 1 and 12
- **Day** (*byte*) – day of the month for wake up time, between 1 and 31
- **Hour** (*byte*) – hours for wake up time (24h clock), between 0 and 23
- **Min** (*byte*) – minutes for wake up time, between 0 and 59
- **Sec** (*byte*) – seconds for wake up time, between 0 and 59

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

int **getWakeUpTime** (*RTCStatus RS*)
Get the wake up time and mode set in RTC.

Parameters

- **RS** (RTCStatus) – parameter to return the current wake up time and mode

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

int **setEnable** (boolean *bEnable*)
Enable or disable RTC wake up function from suspend mode.

Parameters

- **bEnable** (*boolean*) – `true` to enable, `false` to disable RTC wake up function from suspend mode

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

int **getEnable** (boolean[] *bEnable*)
Get the status if wake up function from suspend mode is enabled or disabled.

Parameters

- **bEnable** (*boolean[]*) – parameter to return `true` for enabled, `false` for disabled

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

5.1 RTC Code Examples

5.1.1 Set RTC Wake Up From Suspend mode

```
boolean bSetEnable = true;

if(null == m_rtc) {
  m_rtc = new RTC();
}
if(SmartETK.S_OK != m_rtc.setEnable(bSetEnable)) {
  return false;
}
```

5.1.2 Get RTC Wake Up Status

```
if(null == m_rtc) {
  m_rtc = new RTC();
}
boolean[] bGetEnable = new boolean[1];
if(SmartETK.S_OK != m_rtc.getEnable(bGetEnable)) {
  return false;
}
```

5.1.3 Set RTC Wake Up Time

The folloing code sets the wake up behaviour to wake up from suspend starting from 2015/5/1, every day at 12:00.

```
byte Mode = RTC.ARG_RTC_MODE_DAY;
int Year = 2015;
byte Month = IntToByte(5);
byte Day = IntToByte(1);
byte Hour = IntToByte(12);
byte Min = IntToByte(0);
byte Sec = IntToByte(0);

if(null == m_rtc) {
  m_rtc = new RTC();
}

if(SmartETK.S_OK != m_rtc.setWakeUpTime(Mode, Year, Month, Day, Hour, Min, Sec)) {
  return false;
}
```

5.1.4 Get RTC Wake Up Time

```
if(null == m_rtc) {
  m_rtc = new RTC();
}
m_RS = new RTCStatus();
if(SmartETK.S_OK != m_rtc.getWakeUpTime(m_RS)) {
  return false;
}
```

CHAPTER

SIX

I2C CLASS

class **I2C**

Create a new I2C object with specified bus number, slave address and the length of the offset address.

Parameters

- **int I2CBusNum** – I2C bus number, for example: `0` is for `i2c-0` bus
- **byte I2CAddress** – I2C slave address, support 7 bits slave addresses
- **int OffsetLen** – the length of the registers' offset in number of bytes, accepted values are 0 to 4, `0`: no registers, `1`: 1 byte = 8 bit registers, `2`: 2 bytes = 16 bit registers, `3`: 3 bytes = 24 bit registers, `4`: 4 bytes = 32 bit registers

For example, create an I2C object in I2C bus 1 and I2C slave address 10, and the offset length is 0

```
I2C m_i2c = new I2C(1,10,0);
```

Another example, create an I2C object in I2C bus 1 and I2C slave address 52, and the offset length is 2 (16 bit registers).

```
I2C m_i2c = new I2C(1,52,2);
```

int **read**(byte[] *Buf*, int *Offset*, int *ReadLen*)

Read data from specified offset with a given length, and store the data in buffer.

Parameters

- **Buf** (*byte[]*) – buffer to store the read data
- **Offset** (*int*) – the registers' offset to read from a specified I2C bus number and slave address, accepted values are from 0 to 0x7FFFFFFF
- **ReadLen** (*int*) – number of bytes to read, maximum 255 bytes per transfer.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*

int **write**(byte[] *byBuf*, int *iOffset*, int *iWriteLen*)

Write data to a specified offset with a given length.

Parameters

- **Buf** (*byte*) – the write buffer
- **Offset** (*int*) – the registers' offset of writing to a specified I2C bus number and slave address, accepted values are from 0 to 7FFFFFFF
- **WriteLen** (*int*) – the written data length, maximum 255 bytes per transfer

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*

6.1 I2C Code Examples

6.1.1 Initializate I2C

Create an I2C object in I2C bus 1 and I2C slave address 52, and the offset length is 2.

```
int iBusNum = 1;
byte byAddress = IntToByte(52);
int iOffsetLen = 2;

if(iBusNum < 0 || byAddress < 0 || iOffsetLen < 0) {
  return false;
}
m_i2c = new I2C(iBusNum, byAddress, iOffsetLen);
```

6.1.2 Read I2C Data

Read data from offset "0" with length "2" bytes, and store data in byRead byte array buffer.

```
byte[] byRead = new byte[255]
int iOffset = 0;
int iReadLen = 2;
Arrays.fill(byRead, 0);
if(SmartETK.S_OK != m_i2c.read(byRead, iOffset, iReadLen) || null == byRead) {
  return false;
}
```

6.1.3 Write I2C Data

Write data to offset 0 with length 2 bytes and data value 0x0101. The written data is stored in byWrite byte array buffer.

```
byte[] byWrite = new byte[2]
byWrite[0] = 0x01;
byWrite[1] = 0x01;
int iOffset = 0;
int iWriteLen = 2;

if(SmartETK.S_OK != m_i2c.write(byWrite, iOffset, iWriteLen)) {
  return false;
}
```

CHAPTER

SEVEN

WATCHDOG CLASS

class **WatchDog**
Create a new WatchDog object.

```
WatchDog m_watchdog = new WatchDog();
```

int **setEnable** (boolean *bEnable*)
Enable or disable watch dog function. SmartETK service will feed the watch dog within a period automatically. Once watch dog function is enabled, *keepAlive* needs to be called within the timeout period set by *setTimeout*, otherwise the system will reboot.

Parameters

- **enable** (*boolean*) – `true` for enable, `false` for disable

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getEnable** (boolean[] *enable*)
Get enable state of the watch dog function.

Parameters

- **enable** (*boolean[]*) – parameter to put the return value of the watchdog status, `true` for enabled, `false` for disabled

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **keepAlive** ()
Keep watch dog alive to avoid rebooting the system.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **setTimeout** (int *iTimeout*)
Set watch dog timeout value

Parameters

- **iTimeout** (*int*) – timeout in seconds, accepted values are between `1` and `128`.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getTimeout** (int[] *iTimeout*)
Get watchdog timeout value.

Parameters

- **iTimeout** (*int[]*) – parameter to put the return value of the watchdog timeout in seconds

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

7.1 WatchDog Code Examples

7.1.1 Enable WatchDog

```
if(null == m_watchdog) {
  m_watchdog = new WatchDog();
}
if(SmartETK.S_OK != m_watchdog.enable(true)) {
  return false;
}
```

7.1.2 Get WatchDog status

```
if(null == m_watchdog) {
  m_watchdog = new WatchDog();
}

boolean[] bGetEnable = new boolean[1];

if(SmartETK.S_OK != m_watchdog.getEnable(bGetEnable)) {
  return false;
}
return bGetEnable[0];
```

7.1.3 Keep WatchDog alive

```
if(null == m_watchdog) {
  m_watchdog = new WatchDog();
}
if(SmartETK.S_OK != m_watchdog. keepAlive()){
  return false;
}
```

CHAPTER

EIGHT

UART CLASS

class **Uart**
Create a new UART object.

```
Uart m_uart = new Uart();
```

int **open** (String *sDev*)
Open the specified UART device.

Parameters

- **sDev** (*String*) – UART device name, for example `ttyUSB0`.

Returns *S_OK* if function succeeds

Returns *E_UART_OPENFAIL* if failed to open the device

Returns *E_UART_ALREADY_OPENED* if the device has already has been opened

Returns *E_UART_TTY_BEEN_USED* if the device has been used by other object

Returns `E_*` otherwise, see *Function Return Values*.

int **close** ()
Close the UART device that is currently opened.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

class **UartConfig**
Class to contain the Uart configuration values for `getConfig`.

int **BaudRate**
baud rate, for example `115200`

byte **DataBits**
data bits, `7` for 7 data bits, `8` for 8 data bits

byte **StopBits**
stop bits, `1` for 1 stop bit, `2` for 2 stop bits

byte **Parity**
parity, `0` for none, `1` for odd, `2` even parity

byte **FlowControl**
flow control, `0` for none, `1` for CTS/RTS flow control

int **setConfig** (int *iBaudRate*, byte *byDataBIts*, byte *byStopBits*, byte *byParity*, byte *byFlowCtrl*)
Configure an already opened UART device.

Parameters

- **iBaudRate** (*int*) – baud rate, e.g. `115200`
- **byDataBits** (*byte*) – data bits, *7* for 7-bit data bits, `8` for 8-bit data bits
- **byStopBits** (*byte*) – stop bits, `1` for 1 stop bit, `2`: 2 stop bits
- **byParity** (*byte*) – parity, `0` for none, `1` for odd, `2` for even parity
- **byFlowControl** (*byte*) – flow control, `0` for none, `1` for CTS/RTS flow control

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getConfig**(*UartConfig UC*)

Get the configurations of the opened Uart device and store them in passed UartConfig Class.

Parameters

- **UC** (UartConfig) – Uart Configuration

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
UartConfig UC = m_uart.new UartConfig();

if (SmartETK.S_OK != m_uart.getConfig(UC)) {
  cleanStatus();
  return;
}
```

int **setTimeout**(boolean *bEnable*, int *iTimeout*)

Set the timeout of the opened UART device.

If `bEnable` is set to `true`, the UART read method depends on the `iTimeout` value. If timeout is set to `0` then polling read is used, if `1-255` then the data is read with the corresponding timeout.

If `bEnable` is set to `false` then blocking read is performed.

Parameters

- **bEnable** (*boolean*) – `true` if enable the timeout function, `false` otherwise.
- **iTimeout** (*int*) – timeout value in multiples of 0.1 seconds, accepted range is 0 – 255 (0 - 25.5 seconds)

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getTimeout**(*Timeout T*)

Get the timeout configuration of the opened Uart device and store them in passed Timeout Class.

Parameters

- **T** (Timeout) – timeout configuration

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
Timeout T = m_uart.new Timeout();

if(SmartETK.S_OK != m_uart.getTimeout(T)) {
  cleanStatus();
  return;
}
```

class **ReturnChar**
Used by *getReturnChar*.

boolean **enabled**
Whether the terlmination character function is enabled.

byte **returnChar**
The termination character

int **setReturnChar**(boolean *bEnable*, byte *byReturnChar*)
Set the termination character of the opened UART device.

If `bEnable` is `true`, then read will block until a character equal to‘‘byReturnChar‘‘ is received, or read buffer is full. If `bEnable` is `false` then read will ignore byReturnChar checking when reading data.

Parameters

- **bEnable** (*boolean*) – enable or disable the termination character function.
- **byReturnChar** (*byte*) – the termination character

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getReturnChar**(*ReturnChar RC*)
Get the termination character configuration of the opened Uart device and store them in passed ReturnChar Class.

Parameters

- **RC** (ReturnChar) – termination character configuration

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
ReturnChar RC = new ReturnChar();
if(SmartETK.S_OK != m_uart.getReturnChar(RC)) {
  cleanStatus();
  return;
}
```

int **readData**(int *iReadLen*, byte[] *byRead*, int[] *iActualLen*)
Receive data from the opened UART device.

Parameters

- **iReadLen** (*int*) – number of bytes to read, maximum 1024 bytes per transfer.
- **byRead** (*byte[]*) – pointer to the buffer pointer.
- **iActualLen** (*int[]*) – the actual number of bytes received

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **writeData** (int *iWriteLen*, byte[] *byWrite*)
Send the data to the opened UART device.

Parameters

- **iWriteLen** (*int*) – number of bytes to transmit, maximum 1024 bytes per transfer.
- **byWrite** (*byte[]*) – pointer to data buffer.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

int **reset** ()
Reset the opened or failed to open UART device. If the uart device has been used by other object, *Uart.open* will return an *E_UART_ALREADY_OPENED*. The object could call this reset function to release the UART resource and try to open the device again by calling *Uart.open*.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

8.1 UART Code Examples

8.1.1 UART Initialization

Note: In the sample code below, mETBaudRate refers to an EditText widget.

```
private Uart m_uart = null;
m_uart = new Uart();
if(null == m_uart) {
  cleanStatus();
  return;
}
if(SmartETK.S_OK != m_uart.open((m_sDev = mETDev.getText().toString()))) {
  cleanStatus();
  return;
}
if(SmartETK.S_OK != m_uart.setConfig((m_iBaudRate = Integer.valueOf(mETBaudRate.getText().toString()),
                                     (byte)8,
                                     (byte)1,
                                     (byte)0,
                                     (byte)0)) {
  cleanStatus();
  return;
}
```

8.1.2 Write UART Data

Note: In the sample code below, mETWrite is an EditText widget.

```
if(SmartETK.S_OK != m_uart.writeData(mETWrite.getText().toString().getBytes().length,
                                     mETWrite.getText().toString().getBytes())) {
  return;
}
```

8.1.3 Read UART Data

```
int iReadLen = LENGTH;
byte[] byRead = new byte[LENGTH];
int[] iActualLen = new int[1];

while(SmartETK.S_OK == m_mainThreadUart.readData(iReadLen,
                                                 byRead,
                                                 iActualLen)) {
  if(0 == iActualLen[0]) {
     continue;
  }
 /* Process received byRead byte array ... */
 for(int i = 0; i < byRead.length; i++) {
   byRead[i] = 0;
 }
 iActualLen[0] = 0;
}
```

CHAPTER

NINE

CAN CLASS

class **Can**

Create a new CAN object.

```
Can m_can = newCan();
```

int **open** (String *sName*)

Open the specified CAN device.

Parameters

- **sname** (*String*) – CAN device name, for exammple `can0`, `can1`.

Returns *S_OK* if function succeeds

Returns *E_CAN_OPENFAIL* if opening device has failed

Returns *E_CAN_ALREADY_OPENED* if the object is already open

Returns `E_*` otherwise, see *Function Return Values*.

int **close** ()

Close the CAN device that is currently opened.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **setBitrate** (int *iBitrate*)

Set the bitrate of the opened CAN device.

Parameters

- **iBitrate** (*int*) – bit rate, e.g. `125000`. The default rate is `500000`

Returns *S_OK* if function succeeds

Returns *E_CAN_BAUDRATE_NOT_SUPPORT* if the given bitrate is not supported.

Returns `E_*` otherwise, see *Function Return Values*.

int **getBitrate** (int[] *iBitrate*)

Get the bitrate of the opened CAN device.

Parameters

- **iBitrate** (*int*) – store the return bit rate

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **setTimeout** (boolean *bEnable*, int *iTimeout*)

If `bEnable` is set to `true`, the UART read method depends on the `iTimeout` value. If timeout is set to `0` then polling read is used, if `1-255` then the data is read with the corresponding timeout.

If `bEnable` is set to `false` then blocking read is performed.

Parameters

- **bEnable** (*boolean*) – `true` if enable the timeout function, `false` otherwise.
- **iTimeout** (*int*) – timeout value in multiples of 0.1 seconds, accepted range is 0 – 255 (0 - 25.5 seconds)

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getTimeout** (*Timeout timeout*)

Get the timeout configuration of the opened CAN device and store them in passed Timeout Class.

Parameters

- **T** (Timeout) – timeout configuration

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
Import com.viaembedded.smartetk.SmartETK.Timeout;

Can m_can = new Can();
Timeout timeout = new Timeout();

if(SmartETK.S_OK != m_can.getTimeout(timeout)) {
  cleanStatus();
  return;
}
```

int **setLoopback** (boolean *bEnable*)

The loopback functionality is enabled by default to reflect standard networking behavior for CAN applications. A local loopback functionality is similar to the local echo e.g. of tty devices.

`bEnable = true` (if setRecvOwnMsgs() also set to true, it will receive its own msgs after transmit)

`bEnable = false` (no matter setRecvOwnMsgs() set to true or false, it won't receive its onw msgs after transmit)

Parameters

- **bEnable** (*boolean*) – `true` to enable loopback, `false` otherwise.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getLoopback** (boolean[] *bEnable*)

Get loopback state.

Parameters

- **bEnable** (*boolean[]*) – to variable to place the loopback state, `true` for enabled, `false` for disabled

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
boolean[] bEnable_getlbk = null;

if(SmartETK.S_OK != m_uart.getLoopback(bEnable_getlbk)) {
  cleanStatus();
  return;
}
```

int **setRecvOwnMsgs** (boolean *bEnable*)

Set `CAN_RAW_RECV_OWN_MSGS` flag to decide whether the socket receives frames its own sent or not. As the local loopback is enabled, the reception of the CAN frames on the same socket that was sending the CAN frame is assumed to be unwanted and therefore disabled by default.

`bEnable = true` (if *setLoopback* set to false, it won't receive its own msgs after sending Can frame)

`bEnable = false` (default)

Parameters

- **bEnable** (*boolean*) – `true` if receiving own frames, `false` otherwise

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

int **getRecvOwnMsgs** (Boolean[] *bEnable*)

Get the state of receiving its own sent frames or not.

Parameters

- **bEnable** (*boolean[]*) – variable to put results, `true` if function is enabled, `false` if not.

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*.

Example:

```
boolean[] bEnable_recvOwn = null;

if(SmartETK.S_OK != m_uart.getRecvOwnMsgs(bEnable_recvOwn)) {
  cleanStatus();
  return;
}
```

class **CanFilter**

CAN filter object

static final int **PAYLOAD_SIZE**

`8`, payload data size

static final int **CAN_INV_FILTER**

`0x20000000`, the filter can be inverted (`CAN_INV_FILTER` bit is set in can_id)

int **iCanID**

The CAN ID

int **iCanMask**

Valid bits in CAN ID for frame formats

int **setFilter**(*CanFilter*[] *canFilter*, int *iLength*)

The reception of CAN frames can be controlled by defining 0 .. n filters with the *CanFilter* object array buffer. A filter matches, when:

```
[received_can_id] & CanFilter.iCanMask == CanFilter.iCanID & CanFilter.iCanMask
```

To disable the reception of CAN frames:

```
setFilter(null, 0);
```

Parameters

- **canFilter** (*CanFilter[]*) – *CanFilter* object array
- **iLength** – number of *CanFilter* objects to set, 0 represents to disable the reception of CAN frames.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

class **CanFrame**

CAN frame objec

static final int **PAYLOAD_SIZE**

16, Payload data size

int **iCanID**

32 bit CAN_ID + EFF/RTR flags

final byte[] **byData**

8-byte (byte[8]) frame payload data. The object had been created by byte[8] array buffer. Users can modify data byte array, but cannot modify the object.

int **readFrame**(*CanFrame canFrame*)

Reading CAN frame from the opened CAN device.

Parameters

- **canFrame** (CanFrame) – CAN frame object to read

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

int **writeFrame**(*CanFrame canFrame*)

Write a CAN frame to the opened CAN device.

Parameters

- **canFrame** (CanFrame) – CAN frame object to write

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*.

CHAPTER

TEN

SYSTEMETK CLASS

class **SystemETK**
Create a new SystemETK object.

```
SystemETK m_system = new SystemETK();
```

int **reboot**()
Reboot the machine.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

int **suspend**()
Suspend the machine.

Returns *S_OK* if function succeeds

Returns E_* otherwise, see *Function Return Values*

10.1 SystemETK Code Examples

10.1.1 Reboot the Machine

```
private SystemETK m_system = null;
if(null == m_system) {
  m_system = new SystemETK();
}
if(SmartETK.S_OK != m_system.reboot()) {
  return;
}
```

10.1.2 Suspend the Machine

```
private SystemETK m_system = null;
if(null == m_system) {
  m_system = new SystemETK();
}
if(SmartETK.S_OK != m_system.suspend()) {
  return;
}
```

CHAPTER

ELEVEN

DPMS CLASS

class **Dpms**
Create a new DPMS object.

```
m_dpms = new Dpms();
```

int **setDpms** (boolean *bEnable*)
Enable or disable the DPMS mode of the HDMI output

Parameters

- **bEnable** (*boolean*) – `true` to enable the DPMS mode, `false` to disable

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*

int **getDpms** (boolean[] *bEnable*)
Get the status if DPMS function.

Parameters

- **bEnable** (*boolean[]*) – parameter to contain the return value, `true` for enabled, `false` for disabled

Returns *S_OK* if function succeeds

Returns `E_*` otherwise, see *Function Return Values*