HMIWorks API Reference

Version 1.7.5, Jul. 2022

Warranty

All products manufactured by ICP DAS are under warranty regarding defective materials for a period of one year, beginning from the date of delivery to the original purchaser.

Copyright

ICP DAS assumes no liability for any damage resulting from the use of this product. ICP DAS reserves the right to change this manual at any time without notice. The information furnished by ICP DAS is believed to be accurate and reliable. However, no responsibility is assumed by ICP DAS for its use, not for any infringements of patents or other rights of third parties resulting from its use.

Copyright

Trademark

The names used for identification only may be registered trademarks of their respective companies.

Support

ICP DAS takes your problem as ours.
If you have any problem, please feel free to contact us.
You can count on us for quick response.

Email: service@icpdas.com
Tel: +886-3-5973336

Table of Contents

1. Geometry API

This chapter describes how to draw different shapes, such as rectangles, circles, etc.

The **pContext** described in this chapter is a pointer to the information of drawing space or area, and it is available on the **OnPaint( )** event of a **PaintBox** or **Frame** widget. When drawing on full screen, the **thisContext** pointer can be used instead of **pContext**. The **thisContext** pointer is globally available on each frame, and can be used anywhere.

For example:

```
void PaintBox6OnPaint(tWidget *pWidget, tContext *pContext)
{
    // do somthing with pContext
}
```

When creating a frame, the **OnPaint( )** event is called once. User can also use **WidgetPaint( )** function to activate the **OnPaint( )** event if needed. The **WidgetPaint( )** function rquires a base object address of a widget as parameter, so we have to translate it to tWidget* first.

For example:

```
WidgetPaint((tWidget*)&PaintBox6);      // Activate the OnPaint( ) event of the PaintBox6
WidgetPaint((tWidget*)&Label35);        // Redraw the Label35 widget.
WidgetPaint(WIDGET_ROOT);               // Activate the OnPaint( ) event of all widgets
```

1.1 Widget_Macros

Four macros used in the above example are described below:

1. WidgetLeft gets the x coordinate of the left-top vertex of the widget.
2. WidgetTop gets the y coordinate of the left-top vertex of the widget.
3. WidgetRight gets the x coordinate of the right-bottom vertex of the widget.
4. WidgetBottom gets the y coordinate of the right-bottom vertex of the widget.

1.2 hmi_DrawLine

Draw a line on TouchPAD.

- **Syntax**

```
void hmi_DrawLine(
        tContext *pContext,
        int x1,
        int y1,
        int x2,
        int y2
);
```

- **Parameter**

pContext
[Input] Specify the context

x1
[Input] The x-coordinate of the first vertex of the line segment to draw

y1
[Input] The y-coordinate of the first vertex of the line segment to draw

x2
[Input] The x-coordinate of the second vertex of the line segment to draw

y2
[Input] The y-coordinate of the second vertex of the line segment to draw

- **Return Values**

None

- **Examples**

[C]

```
hmi_DrawLine(pContext, x1, y1, x2, y2);
```

- **Remark**

None

1.3 hmi_DrawRect

Draw a rectangle on TouchPAD.

- **Syntax**

```
void hmi_DrawRect(
        tContext *pContext,
        int x1,
        int y1,
        int x2,
        int y2
);
```

- **Parameter**

pContext
[Input] Specify the context

x1
[Input] The x-coordinate of the first diagonal vertex of the rectangle to draw

y1
[Input] The y-coordinate of the first diagonal vertex of the rectangle to draw

x2
[Input] The x-coordinate of the second diagonal vertex of the rectangle to draw

y2
[Input] The y-coordinate of the second diagonal vertex of the rectangle to draw

- **Return Values**

None

- **Examples**

[C]

```
hmi_DrawRect(pContext, x1, y1, x2, y2);
```

- **Remark**

None

1.4 hmi_FillRect

Draw a rectangle and fill it with a specified color on TouchPAD.

- **Syntax**

```
void hmi_FillRect(
        tContext *pContext,
        int x1,
        int y1,
        int x2,
        int y2
);
```

- **Parameter**

pContext
[Input] Specify the context

x1
[Input] The x-coordinate of the first diagonal vertex of the rectangle to fill

y1
[Input] The y-coordinate of the first diagonal vertex of the rectangle to fill

x2
[Input] The x-coordinate of the second diagonal vertex of the rectangle to fill

y2
[Input] The y-coordinate of the second diagonal vertex of the rectangle to fill

- **Return Values**

None

- **Examples**

[C]

```
hmi_FillRect(pContext, x1, y1, x2, y2);
```

- **Remark**

None

1.5 hmi_DrawCircle

Draw a circle on TouchPAD.

- **Syntax**

```
void hmi_DrawCircle(
        tContext *pContext,
        int x,
        int y,
        int w
);
```

- **Parameter**

pContext
[Input] Specify the context

x
[Input] The x-coordinate of the center of the circle to draw

y
[Input] The y-coordinate of the center of the circle to draw

w
[Input] The radius of the circle to draw

- **Return Values**

None

- **Examples**

[C]

```
hmi_DrawCircle(pContext, x, y, w);
```

- **Remark**

None

1.6 hmi_FillCircle

Draw a circle and fill it with a specified color on TouchPAD.

- **Syntax**

```
void hmi_FillCircle(
        tContext *pContext,
        int x,
        int y,
        int w
);
```

- **Parameter**

pContext

[Input] Specify the context

x

[Input] The x-coordinate of the center of the circle to fill

y

[Input] The y-coordinate of the center of the circle to fill

w

[Input] The radius of the circle to fill

- **Return Values**

None

- **Examples**

[C]

```
hmi_FillCircle(pContext, x, y, w);
```

- **Remark**

None

1.7 hmi_DrawEllipse

Draw an ellipse on TouchPAD.

- **Syntax**

```
void hmi_DrawEllipse(
        tContext *pContext,
        int x1,
        int y1,
        int x2,
        int y2
);
```

- **Parameter**

pContext
[Input] Specify the context

x1
[Input] The x-coordinate of the first diagonal vertex of the rectangular that inscribes the ellipse to draw

y1
[Input] The y-coordinate of the first diagonal vertex of the rectangular that inscribes the ellipse to draw

x2
[Input] The x-coordinate of the second diagonal vertex of the rectangular that inscribes the ellipse to draw

y2
[Input] The y-coordinate of the second diagonal vertex of the rectangular that inscribes the ellipse to draw

- **Return Values**

None

- **Examples**

[C]

```
hmi_DrawEllipse(pContext, x1, y1, x2, y2);
```

- **Remark**

None

1.8 hmi_FillEllipse

Draw an ellipse and fill it with a specified color on TouchPAD.

- **Syntax**

```
void hmi_FillEllipse(
        tContext *pContext,
        int x1,
        int y1,
        int x2,
        int y2
);
```

- **Parameter**

pContext
[Input] Specify the context

x1
[Input] The x-coordinate of the first diagonal vertex of the rectangular that inscribes the ellipse to fill

y1
[Input] The y-coordinate of the first diagonal vertex of the rectangular that inscribes the ellipse to fill

x2
[Input] The x-coordinate of the second diagonal vertex of the rectangular that inscribes the ellipse to fill

y2
[Input] The y-coordinate of the second diagonal vertex of the rectangular that inscribes the ellipse to fill

- **Return Values**

None

- **Examples**

[C]

```
hmi_FillEllipse(pContext, x1, y1, x2, y2);
```

- **Remark**

None

1.9 hmi_DrawPolyLine

Draw a polyline on TouchPAD.

- **Syntax**

```
void hmi_DrawPolyLine(
        tContext *pContext,
        int x,
        int y,
        const int n,
        const short coordinates[]
);
```

- **Parameter**

pContext
[Input] Specify the context

x
[Input] The x-coordinate of the reference point to which all the vertices of the polyline refer

y
[Input] The y-coordinate of the reference point to which all the vertices of the polyline refer

n
[Input] The number of vertices of the polygon to draw

coordinates
[Input] The array of coordinates of the polygon to draw

- **Return Values**

None

- **Examples**

[C]

```
void PaintBox4OnPaint(tWidget *pWidget, tContext *pContext)
{
    const short coordinates[10]={50,50,0,150,100,100,0,100,100,150};
    hmi_DrawPolyLine(pContext, 50, 50, 5, coordinates);
}
// hmi_DrawPolyLine(pContext, 50(reference point), 50(reference point), 5(have 5 points), coordinates);
// const short coordinates[10]={50(point 1, x),50(point 1, y),0(point 2, x),150(point 2, y),100,100,0,100,100,150};
// point1 real x=50+50(reference point)=100
// point1 real y=50+50(reference point)=100
```


- **Remark**

  None

1.10 hmi_DrawPolygon

Draw a polygon on TouchPAD.

- **Syntax**

```
void hmi_DrawPolygon(
        tContext *pContext,
        int x,
        int y,
        const int n,
        const short coordinates[]
);
```

- **Parameter**

pContext
[Input] Specify the context

x
[Input] The x-coordinate of the reference point to which all the vertices of the polygon refer

y
[Input] The y-coordinate of the reference point to which all the vertices of the polygon refer

n
[Input] The number of vertices of the polygon to draw

coordinates
[Input] The array of coordinates of the polygon to draw

- **Return Values**

None

- **Examples**

[C]

```
void PaintBox4OnPaint(tWidget *pWidget, tContext *pContext)
{
    const short coordinates[10]={50,50,0,150,100,100,0,100,100,150};
    hmi_DrawPolygon(pContext, 50, 50, 5, coordinates);
}
// hmi_DrawPolygon(pContext, 50( reference point), 50( reference point), 5(have 5 point), coordinates);
// const short coordinates[10]={50(point 1 ,x),50(point 1,y),0(point 2,x),150(point 2,y),100,100,0,100,100,150};
// point1 real x=50+50(reference point)=100
// point1 real y=50+50(reference point)=100
```

- **Remark**

None

1.11 hmi_FillPolygon

Draw a polygon and fill it with a specified color on TouchPAD.

- **Syntax**

```
void hmi_FillPolygon(
        tContext *pContext,
        int x,
        int y,
        const int n,
        const short coordinates[]
);
```

- **Parameter**

pContext
[Input] Specify the context

x
[Input] The x-coordinate of the reference point to which all the vertices of the polygon refer

y
[Input] The y-coordinate of the reference point to which all the vertices of the polygon refer

n
[Input] The number of vertices of the polygon to fill

coordinates
[Input] The array of coordinates of the polygon to fill

- **Return Values**

None

- **Examples**

[C]

```
void PaintBox4OnPaint(tWidget *pWidget, tContext *pContext)
{
    const short coordinates[10]={50,50,0,150,100,100,0,100,100,150};
    hmi_FillPolygon(pContext, 50, 50, 5, coordinates);
}
// hmi_FillPolygon(pContext, 50( reference point), 50( reference point), 5(have 5 point), coordinates);
// const short coordinates[10]={50(point 1 ,x),50(point 1,y),0(point 2,x),150(point 2,y),100,100,0,100,100,150};
// point1 real x=50+50(reference point)=100      // point1 real y=50+50(reference point)=100
```


- **Remark**

  None

1.12 hmi_SetForeground

Set the foreground color.

- **Syntax**

```
void hmi_SetForeground(
        tContext *pContext,
        unsigned long color
);
```

- **Parameter**

pContext

[Input] Specify the context

color

[Input] Specify the color. A color is represented by a three-byte integer. The order of the bytes from the most significant byte to the least is Red, Green, Blue.
For example, 0x0000FF represents Blue.

- **Return Values**

None

- **Examples**

[C]

```
hmi_SetForeground(pContext, 0x0000FF);      //bule; R-G-B
```

- **Remark**

None

1.13 hmi_DrawImage

Draw images in a PaintBox with images which an ObjectList contains.

- **Syntax**

```
void hmi_DrawImage(
        const tContext *pContext,
        const unsigned char *puclmage,
        long IX,
        long IY
);
```

- **Parameter**

pContext
[Input] Specify the context

puclmage
[Input] Specify the image to draw in the PaintBox

IX
[Input] Specify the x coordinate referenced to the frame's origin (not referenced to the left, top corner of the PaintBox)

IY
[Input] Specify the y coordinate referenced to the frame's origin (not referenced to the left, top corner of the PaintBox)

- **Return Values**

None

- **Examples**

[C]

```
void PaintBox5OnPaint(tWidget *pWidget, tContext *pContext)
{
    // The following code draws image on a PaintBox widget.
    // Images outside the PaintBox area are ingored (invisible).
    // The image is stored on the ObjectList    linked with TextPushButton5 for example.

    hmi_DrawImage(pContext, TextPushButton5.puclmage[0], 0, 0);
}
```

- **Remark**

This function is supported by HMIWorks 2.06.00 or above

2. Frame API

TouchPAD supports multi-frame feature.
The chapter provides APIs to handle multi-frame functions.

2.1 hmi_GotoFrameByName

Goto the frame by specified name

- **Syntax**

```
void hmi_GotoFrameByName(
        const char *frame_name
);
```

- **Parameter**

frame name
[Input] Specify the name of frame to goto

- **Return Values**

None

- **Examples**

[C]

```
void HotSpot44OnClick(tWidget *pWidget)
{
    // After clicking the Hotspot 44, TouchPAD switches to the frame of "Frame4"
    hmi_GotoFrameByName("Frame4");
}
```

- **Remark**

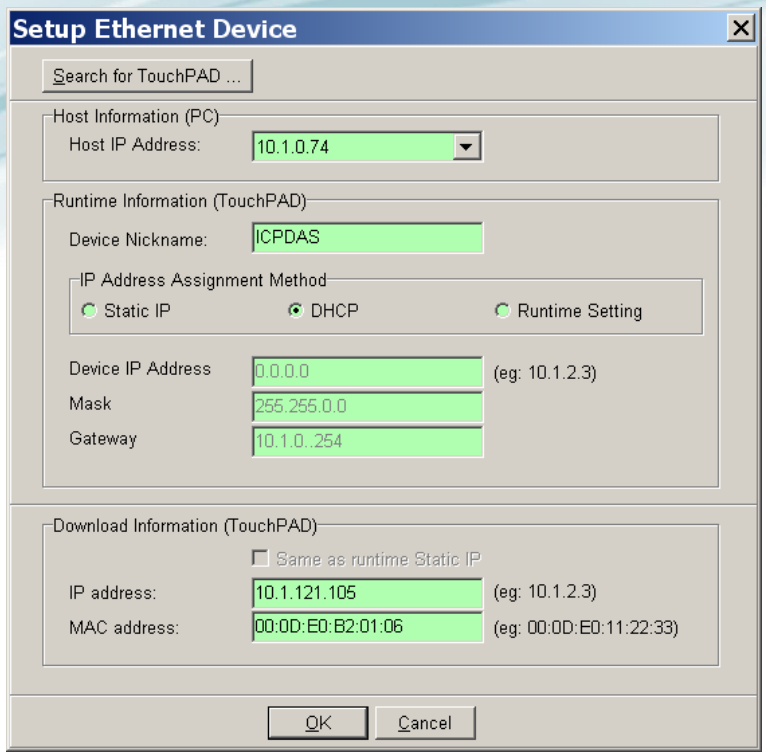
Don't have codes after calling **hmi_GotoFrameByName()**, since you may access wrong widgets between frames.

3. Network Configuration API

This chapter provides network configuration APIs.

Be sure to set up the TouchPAD to have the Ethernet setting as **"Runtime Setting"** as shown below.

⚠ *Note that configured **"Static IP"** or **"DHCP"** disables the network configuration API functions.*



3.1 hmi_NetworkParamsGet

Get the network configuration of the TouchPAD from the parameter area in the internal flash.

- **Syntax**

```
int hmi_NetworkParamsGet(
        unsigned long *ulDHCP,
        unsigned long *ullP,
        unsigned long *ulMask,
        unsigned long *ulGateway
);
```

- **Parameter**

ulDHCP

[Output] Specify the pointer to the variable of the DHCP setting

ulDHCP	IP Type
0	Static IP
1	DHCP Enabled

ulIP

[Output] Specify the pointer to the variable of the IP address

ulMask

[Output] Specify the pointer to the variable of the subnet mask

ulGateway

[Output] Specify the pointer to the variable of gateway address

- **Return Values**

Always 1 (TRUE)

- **Examples**

[C]

```
#define MAX_FUNS 5
#define BUF_SIZE 16

unsigned long gulValue[MAX_FUNS] = {0};
// Funs: Read DHCP, IP, Mask, Gateway, NetID
static char gszIP[BUF_SIZE];    // size: at least 16
```

```
void FConfig2OnShow()
{
	// Get DHCP, IP, Mask, Gateway
	hmi_NetworkParamsGet(gulValue +0, gulValue +1, gulValue +2, gulValue +3);
	gulValue[4] = hmi_NetIDParamsGet();

	// Display the Gateway Address for example
	hmi_IPToStr(gulValue[3], gszIP);
	LabelTextSet(&Label11, gszIP);
}
```

- **Remark**

  None

3.2 hmi_NetworkParamSet

Set the network configuration of the TouchPAD to the parameter area in the internal flash.

- **Syntax**

```
int hmi_NetworkParamsSet(
        unsigned long ulDHCP,
        unsigned long ullP,
        unsigned long ulMask,
        unsigned long ulGateway
);
```

- **Parameter**

ulDHCP

[Input] Specify the variable of the DHCP setting

ulDHCP	IP Type
0	Static IP
1	DHCP Enabled

ulIP

[Input] Specify the variable of the IP address

ulMask

[Input] Specify the variable of the subnet mask

ulGateway

[Input] Specify the variable of gateway address

- **Return Values**

Always 1 (TRUE)

- **Examples**

[C]

```
#define MAX_FUNS 5
unsigned long gulValue[MAX_FUNS] = {0};
// Funs: Read DHCP, IP, Mask, Gateway, NetID
```

```
void btnSaveAll25OnClick(tWidget *pWidget)
{
        gulValue[0] = 0;    // DHCP=0, that is, static IP
        gulValue[1] = hmi_StrToIP("10.1.0.59");     // IP
        gulValue[2] = hmi_StrToIP("255.255.0.0"); // Mask
        gulValue[3] = hmi_StrToIP("10.1.0.254");   // Gateway
        gulValue[4] = 1;    // Net ID

        // Save DHCP, IP, Mask, Gateway
        hmi_NetworkParamsSet(gulValue[0], gulValue[1], gulValue[2], gulValue[3]);

        // Save Net ID
        hmi_NetIDParamsSet(gulValue[4]);
}
```

- **Remark**

  None

3.3 hmi_LocalIPAddrGet

Get the local IP address.

- **Syntax**

```
unsigned long hmi_LocalIPAddrGet();
```

- **Parameter**

None

- **Return Values**

The current used IP address

- **Examples**

[C]

```
#define BUF_SIZE 22
static char gszIP[BUF_SIZE];          // size: at least 16
unsigned long ipaddr = 0;

void FConfig2OnShow()
{
        // Read the current IP address and display it
        ipaddr = hmi_LocalIPAddrGet();
        hmi_IPToStr(ipaddr, gszIP);
        LabelTextSet(&Label11, gszIP);
}
```

- **Remark**

None

3.4 hmi_LocalMACAddrGet

Get the local MAC address.

- **Syntax**

```
void hmi_LocalMACAddrGet(
        unsigned char *pucMACAddr
);
```

- **Parameter**

pucMACAddr

[Output] Specify the pointer to the array of the MAC address. (6 bytes long)

- **Return Values**

None

- **Examples**

[C]

```
unsigned char macaddr[6];
char msg[64];

void BitButton5OnClick(tWidget *pWidget)
{
    hmi_LocalMACAddrGet(macaddr);

    usprintf(msg, "%02x.%02x.%02x.%02x.%02x.%02x", macaddr[0], macaddr[1], macaddr[2], macaddr[3],
        macaddr[4], macaddr[5]);
    LabelTextSet(&Label4, msg);
}
```

- **Remark**

None

3.5 hmi_IPToStr

Convert the IP address of integer to the IP address string.

- **Syntax**

```
int hmi_IPToStr(
        unsigned long ulIP,
        char szIP[]
);
```

- **Parameter**

ulIP
[Input] Specify the variable of the IP address integer

szIP
[Output] Specify the pointer to the string of the IP address (16 bytes including null terminator char)

- **Return Values**

Reserved for future use

- **Examples**

[C]

```
#define MAX_FUNS      5
#define BUF_SIZE      16

unsigned long gulValue[MAX_FUNS] = {0};
// Funs: Read DHCP, IP, Mask, Gateway, NetID
static char gszIP[BUF_SIZE];    // size: at least 16

void FConfig2OnShow()
{
        // Get DHCP, IP, Mask, Gateway
        hmi_NetworkParamsGet(gulValue +0, gulValue +1, gulValue +2, gulValue +3);
        gulValue[4] = hmi_NetIDParamsGet();

        // Display the Gateway Address for example
        hmi_IPToStr(gulValue[3], gszIP);
        LabelTextSet(&Label11, gszIP);
}
```

- **Remark**

None

3.6 hmi_StrToIP

Convert the IP address string to the IP address of integer.

- **Syntax**

```
unsigned long hmi_StrToIP(
                        char szIP[]
);
```

- **Parameter**

szIP

[Input] Specify the pointer to the string of the IP address. (16 bytes including null terminator char)

- **Return Values**

The IP address of integer

- **Examples**

[C]

```
#define MAX_FUNS      5
unsigned long gulValue[MAX_FUNS] = {0};
// Funs: Read IP, DHCP, IP, Mask, Gateway, NetID

void btnSaveAll25OnClick(tWidget *pWidget)
{
        gulValue[0] = 0;    // DHCP=0, that is, static IP
        gulValue[1] = hmi_StrToIP("10.1.0.59");    // IP
        gulValue[2] = hmi_StrToIP("255.255.0.0");    // Mask
        gulValue[3] = hmi_StrToIP("10.1.0.254");    // Gateway
        gulValue[4] = 1;    // Net ID

        // Save DHCP, IP, Mask, Gateway
        hmi_NetworkParamsSet(gulValue[0], gulValue[1], gulValue[2], gulValue[3]);

        // Save Net ID
        hmi_NetIDParamsSet(gulValue[4]);
}
```

- **Remark**

None

4. TCP API

This chapter provides TCP APIs.

4.1 hmi_TCPNew

Allocate a TCP socket if possible.

- **Syntax**

```
tHandle hmi_TCPNew();
```

- **Parameter**

None

- **Return Values**

typedef int tHandle;

If successful, a handle for the socket is returned.

If not successful, -1 is returned.

- **Examples**

[C]

```
tHandle h = hmi_TCPNew();
```

- **Remark**

None

4.2 hmi_TCPListen

This function establishes a TCP socket for listening as a server.

- **Syntax**

```
void hmi_TCPListen(
        tHandle h,
        unsigned short usPort
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

usPort

[Input] Specify the port of the TCP communication to listen

- **Return Values**

None

- **Examples**

[C]

```
unsigned short port_listen = 10000;
tHandle h = hmi_TCPNew();
hmi_TCPListen(h, port_listen);
```

- **Remark**

None

4.3 hmi_TCPOpen

This function is used in a client to establish a TCP socket for connecting to a server.

- **Syntax**

```
void hmi_TCPOpen(
        tHandle h,
        unsigned long ulIPAddr,
        unsigned short usRemotePort,
        unsigned short usLocalPort
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

ulIPAddr

[Input] Specify the IP address of the server to connect

usRemotePort

[Input] Specify the remote listening port of the server

usLocalPort

[Input] Specify any value which is greater than zero. This is reserved for future use

- **Return Values**

None

- **Examples**

[C]

```
unsigned short remote_port = 10000;
unsigned short local_port = 10001;

tHandle h = hmi_TCPNew();
hmi_TCPOpen(h, TCP_IPADDR(10,1,0,45), remote_port, local_port);
```

- **Remark**

None

4.4 hmi_TCPClose

This function closes and deallocates a TCP socket.

- **Syntax**

```
void hmi_TCPClose(
        tHandle h
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

- **Return Values**

None

- **Examples**

[C]

```
tHandle h = hmi_TCPNew();
…
hmi_TCPClose (h);
```

- **Remark**

The handle/socket is invalid after calling hmi_TCPClose().

Use hmi_TCPNew() to get a new handle for using socket again.

4.5 hmi_TCPGetLocalPort

This function gets local port of the TCP socket.
If operating as a server, the local port is the listening port. If operating as a client, the local port is meaningless, and reserved for future use.

- **Syntax**

```
unsigned short hmi_TCPGetLocalPort(
                        tHandle h
);
```

- **Parameter**

h
[Input] Specify the handle to the TCP socket

- **Return Values**

The local port of the TCP socket

- **Examples**

[C]

```
tHandle h = hmi_TCPNew();
…
unsigned short local_port = hmi_TCPGetLocalPort (h);
```

- **Remark**

None

4.6 hmi_TCPGetRemotePort

This function gets remote port of the TCP socket.
If operating as a server, the remote port is 0. If operating as a client, the remote port is the port that the server uses for connection.

- **Syntax**

```
unsigned short hmi_TCPGetRemotePort(
                        tHandle h
);
```

- **Parameter**

h
[Input] Specify the handle to the TCP socket

- **Return Values**

The remote port of the TCP socket

- **Examples**

[C]

```
tHandle h = hmi_TCPNew();
…
unsigned short remote_port = hmi_TCPGetRemotePort (h);
```

- **Remark**

None

4.7 hmi_TCPState

This function gets the state of the TCP socket.

- **Syntax**

```
int hmi_TCPState(
        tHandle h
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

- **Return Values**

The state of the TCP socket

STATE_TCP_IDLE	0
STATE_TCP_LISTEN	1
STATE_TCP_CONNECTING	2
STATE_TCP_CONNECTED	3

- **Examples**

[C]

```
tHandle h = hmi_TCPNew();
…
int state = hmi_TCPState (h);
```

- **Remark**

None

4.8 hmi_TCPWrite

This function writes data through a TCP socket. Actually hmi_TCPWrite puts data to the queue for next flush to output to the destination.

- **Syntax**

```
int hmi_TCPWrite(
        tHandle h,
        unsigned char data[],
        int len
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

data

[Input] Specify the array to write

len

[Input] Specify the length of the array to write

- **Return Values**

None

- **Examples**

[C]

```
unsigned short remote_port = 10000;
unsigned short local_port = 10001;
unsigned char data[1024];

tHandle h = hmi_TCPNew();
hmi_TCPOpen(h, TCP_IPADDR(10,1,0,45), remote_port, local_port);
hmi_TCPWrite(h, data, 1024);
```

- **Remark**

None

4.9 hmi_TCPOutput

This function writes data through a TCP socket to the destination immediately (no waiting in the queue).

- **Syntax**

```
void hmi_TCPOutput(
        tHandle h,
        unsigned char data[],
        int len
);
```

- **Parameter**

h
[Input] Specify the handle to the TCP socket

data
[Input] Specify the array to write

len
[Input] Specify the length of the array to write

- **Return Values**

None

- **Examples**

[C]

```
unsigned short remote_port = 10000;
unsigned short local_port = 10001;
unsigned char data[1024];

tHandle h = hmi_TCPNew();
hmi_TCPOpen(h, TCP_IPADDR(10,1,0,45), remote_port, local_port);
hmi_TCPOutput (h, data, 1024);
```

- **Remark**

None

4.10 hmi_TCPReadEx

This function reads data through a TCP socket.

- **Syntax**

```
void hmi_TCPReadEx(
        tHandle h,
        unsigned char data[],
        int len,
        DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

data

[Output] Specify the pointer to the array for the reading data

len

[Input] Specify the length of the array to read

timeout

[Input] Specify the timeout value of the TCP communications

- **Return Values**

The length of the receiving data

- **Examples**

[C]

```
unsigned short remote_port = 10000;
unsigned short local_port = 10001;
DWORD timeout = 200;
unsigned char data[1024];

tHandle h = hmi_TCPNew();
hmi_TCPOpen(h, TCP_IPADDR(10,1,0,45), remote_port, local_port);
int length_received = hmi_TCPReadEx (h, data, 1024, timeout);
```

- **Remark**

  Backward Compatibility:

  The old version for TCPRead is as below:

  int **hmi_TCPRead**(

  tHandle h,

  unsigned char *buf,

  int buf_len);

  It reads the TCP data without waiting, that is, returns immediately.

  It is the same as the hmi_TCPReadEx with timeout = 0.

4.11 hmi_TCPSendCmdEx

This function sends data and then receives data through a TCP socket.

⚠ *Note: The connection will be cut off if there are 10 timeouts continuously happened when calling this function.*

➢ **Syntax**

```
void hmi_TCPSendCmdEx(
        tHandle h,
        unsigned char *send_data,
        int send_data_len,
        unsigned char *receive_data,
        int recv_data_len,
        DWORD timeout
);
```

➢ **Parameter**

h
[Input] Specify the handle to the TCP socket

Send_data
[Input] Specify the pointer to the sending data array

Send_data_len
[Input] Specify the sending data len

receive_data
[Output] Specify the pointer to the receiving data array

recv_data_len
[Input] Specify the length of the recv data len

timeout
[Input] Specify the timeout value of the TCP communications

➢ **Return Values**

The length of the receiving data

- **Examples**

[C]

```
unsigned short remote_port = 10000;
unsigned short local_port = 10001;
DWORD timeout = 200;
unsigned char send_data[1024];
unsigned char receive_data[1024];

tHandle h = hmi_TCPNew();
hmi_TCPOpen(h, TCP_IPADDR(10,1,0,45), remote_port, local_port);
int length_received = hmi_TCPSendCmdEx (h, send_data, 1024, receive_data, 1024, timeout);
```

- **Remark**

Backward Compatibility:

The old version for TCPSendCmd is as below:

int **hmi_TCPSendCmd**(

tHandle h,

unsigned char *send_data,

int send_len,

unsigned char *recv_data,

int recv_len);

It is the same as the hmi_TCPSendCmdEx with timeout = 200.

4.12 hmi_TCPTimeoutBeep

This function beeps when timeout occurs.

- **Syntax**

```
void hmi_TCPTimeoutBeep (
        int iConfig
);
```

- **Parameter**

iConfig

[Input] Specify an integer to configure the timeout beep

iConfig	Description
0	Disable
others	Enable

- **Return Values**

None

- **Examples**

[C]

```
int config = 1;    // Enable the timeout beep

hmi_TCPTimeoutBeep(config);
```

- **Remark**

None

4.13 hmi_TCPAcceptedBy

This function returns the socket number which the server (on TouchPAD) accepted from.

- **Syntax**

```
void hmi_TCPAcceptedBy (
        tHandle h
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

- **Return Values**

If this socket is accepted by the server (on TouchPAD), then the returning value is the socket number which the server accepted from

Else the returning value is 255

- **Examples**

[C]

```
tHandle serverh = -1;
char szipstr[32];
unsigned char *data = "Hello World!";

#define PORT_BIND 23

void BitButton8OnClick(tWidget *pWidget)    // Listen then accept
{
    //
    // Allocate a socket. Check if h < 0 to prevent exhaust tHandle
    //
    if (serverh < 0)
        serverh = hmi_TCPNew();

    //
    // Listen and wait for accepting
    //
    hmi_TCPListen(serverh, PORT_BIND);
}

void BitButton5OnClick(tWidget *pWidget) // Send
{
```

```
        //
        // Loop through all tHandle to pick up the client to communicate
        //
        for (int i = 0; i < MAX_TCP_SOCKET; i++)
        {
                //
                // If the tHandle i is accepted by which the server listens to
                //
                if (serverh == hmi_TCPAcceptedBy(i))
                {
                        if (hmi_TCPState(i) == STATE_TCP_CONNECTED)
                        {
                                hmi_TCPOutput(i, data, strlen(data));
                        }
                }
        }
}


void BitButton9OnClick(tWidget *pWidget) // Receive
{
        static unsigned char buf[32];
        static char msg[32];

        for (int i = 0; i < MAX_TCP_SOCKET; i++)
        {
                if (serverh == hmi_TCPAcceptedBy(i))
                {
                        if (hmi_TCPState(i) == STATE_TCP_CONNECTED)
                        {
                                int cnt = hmi_TCPReadEx(i, buf, 32, 200);
                                if (cnt > 0)
                                {
                                        usprintf(msg, "%d%s", i, buf);
                                        LabelTextSet(&Label11, msg);
                                }
                        }
                }
        }
}

void BitButton10OnClick(tWidget *pWidget) // Close
{
        hmi_TCPClose(serverh);
        //
        // Be sure to set tHandle back to -1 to indicate this tHandle is released
        //
        serverh = -1;
        }
```

- **Remark**

  None

4.14 hmi_TCPAbandon

This function abandons a TCP connection. Different than hmi_TCPClose, it gives up the connection immediately and has an parameter to decide if a reset signal (RST) should be sent to the remote side.

- **Syntax**

```
void hmi_TCPAbandon (
        tHandle h,
        int reset
);
```

- **Parameter**

h

[Input] Specify the handle to the TCP socket

Range: 0-31, except TPD-283 (0-7)

reset

[Input] Tell the remote side of the connection to reset or not when TouchPAD abandons the connection

reset	Description
0	Send no reset (RST) signal to the remote side
Any value other than 0	Send reset

- **Return Values**

None

- **Examples**

[C]

```
tHandle h = 2;    // supposed
int reset = 1;    // send reset when abandoning

hmi_TCPAbandon (h, reset);
```

- **Remark**

The handle/socket is invalid after calling hmi_TCPAbandon().

Use hmi_TCPNew() to get a new handle for using socket again.

5. Modbus TCP Master API

This chapter provides the Modbus TCP Master API functions.
Modbus is a commonly-used communication protocol in the industry field.

The old mbm_ series Modbus TCP master API is still supported for backward compatility.

Mapping between the function code and the Modbus TCP Master API

Function Code	HMIWorks Modbus TCP Master API
1	mtm_ReadDO
2	mtm_ReadDI
3	mtm_ReadAO
4	mtm_ReadAI
5	mtm_WriteDO with ch_count = 1
6	mtm_WriteAO with ch_count = 1
15	mtm_WriteDO with ch_count > 1
16	mtm_WriteAO with ch_count > 1

5.1 mtm_Register

Register a Modbus communication on the TouchPAD.

- **Syntax**

```
tHandle mtm_Register(
                    int NetID,
                    DWORD modbus_ip,
                    int modbus_port
);
```

- **Parameter**

NetID

[Input] Specify the Net ID of the device (Range: 1 ~ 247)

modbus_ip

[Input] Specify the IP of the device to communicate

modbus_port

[Input] Specify the port of the Modbus communication

- **Return Values**

typedef int tHandle

If successful, a handle to a Modbus communication is returned. The range of the possible value of tHandle is 0 ~ 7.

If not, -1 is returned.

- **Examples**

[C]

```
tHandle h = mtm_Register(1, TCP_IPADDR(10,1,102,64), 502);
```

- **Remark**

Backward Compatibility:

The mbm_Register function is exactly the same as the mtm_Register function.

5.2 mtm_Unregister

Unregister a Modbus communication from the TouchPAD.

- **Syntax**

```
BOOL mtm_Unregister(
        tHandle h
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

- **Return Values**

True, if unregistering the modbus communication is successful.

False, if not.

- **Examples**

[C]

```
tHandle h = mtm_Register(1, TCP_IPADDR(10,1,102,64), 502);
…
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The mbm_Unregister function is exactly the same as the mtm_Unregister function.

5.3 mtm_WriteDO

Write DO Values to the digital output module through Modbus communications.

- **Syntax**

```
BOOL mtm_WriteDO(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                char *pcData,
                DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DO module

pcData

[Input] Specify the pointer to an array of char in which each bit of every byte represents the status of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms) When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

- **Return Values**

True, if writing DO successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 16;
char DOValue[2];
DWORD timeout = 200;

// Turn on the ch 0 and ch1.
DOValue[0] = 11;
// Turn on the ch 8 and ch9.
DOValue[1] = 11;

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500);    // Need some time to wait for respond.
mtm_WriteDO(h, NetID, addr, ch_count, DOValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old WriteDO function has the declaration as below:

BOOL **mbm_WriteDO**(
tHandle h,
int addr,
int ch_count,
DWORD DOValue);

The mbm_WriteDO function does the the same job as the mtm_WriteDO function, except with

1. NetID: uses what mtm_Register specifies
2. ch_count: is not greater than 32.
3. DOValue: is replaced with the char array in the mtm_WriteDO function.
4. timeout: uses the default value, 200 ms.

5.4 mtm_ReadDO

Read DO Values from the digital output module through Modbus communications.

- **Syntax**

```
BOOL mtm_ReadDO(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                char *pcData,
                DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DO module

pcData

[Output] Specify the pointer to an array of char in which each bit of every byte represents the status of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms)
When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

- **Return Values**

True, if reading DO successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 16;
char DOValue[2];
DWORD timeout = 200;

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500);    // Need some time to wait for respond.
mtm_ReadDO(h, NetID, addr, ch_count, DOValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old ReadDO function has the declaration as below:

BOOL **mbm_ReadDO**(

tHandle h,

int addr,

int ch_count,

DWORD *DOValue);

The mbm_ReadDO function does the the same job as the mtm_ReadDO function, except with

1. NetID: uses what mtm_Register specifies
2. ch_count: is not greater than 32.
3. DOValue: is replaced with the char array in the mtm_ReadDO function.
4. timeout: uses the default value, 200 ms.

5.5 mtm_ReadDI

Read DI Values from the digital input module through Modbus communications.

- **Syntax**

```
BOOL mtm_ReadDI(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                char *pcData,
                DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DI module

pcData

[Output] Specify the pointer to an array of char in which each bit of every byte represents the status of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms)
When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

- **Return Values**

True, if reading DI successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 16;
char DIValue[2];
DWORD timeout = 200;

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500);    // Need some time to wait for respond.
mtm_ReadDI(h, NetID, addr, ch_count, DIValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old ReadDI function has the declaration as below:

BOOL **mbm_ReadDI**(

tHandle h,

int addr,

int ch_count,

DWORD *DIValue);

The mbm_ReadDI function does the the same job as the mtm_ReadDI function, except with

1. NetID: uses what mtm_Register specifies
2. ch_count: is not greater than 32.
3. DIValue: is replaced with the char array in the mtm_ReadDI function.
4. timeout: uses the default value, 200 ms.

5.6 mtm_WriteAO

Write AO Values to the analog output module through Modbus communications.

- **Syntax**

```
BOOL mtm_WriteAO(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                WORD *pwData,
                DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AO module

pwData

[Input] Specify the pointer to an array of WORD (2 byte) in which each element of the array represents the state of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms)
When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

- **Return Values**

True, if writing AO successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 2;
WORD AOValue[2]; //for example, we have a two-channel AO module
DWORD timeout = 200;

AOValue[0] = 100; //ch 0
AOValue[1] = 120; //ch 1

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500); // Need some time to wait for respond.
mtm_WriteAO(h, NetID, addr, ch_count, AOValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old WriteAO function has the declaration as below:

BOOL **mbm_WriteAO**(
tHandle h,
int addr,
int ch_count,
WORD *AOValue);

The mbm_WriteAO function does the the same job as the mtm_WriteAO function, except with

1. NetID: uses what mtm_Register specifies
2. timeout: uses the default value, 200 ms.

5.7 mtm_ReadAO

Read AO Values from the analog output module through Modbus communications.

- **Syntax**

```
BOOL mtm_ReadAO(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                WORD *pwData,
                DWORD timeout
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AO module

pwData

[Output] Specify the pointer to an array of WORD (2 byte) in which each element of the array represents the state of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms)
When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

- **Return Values**

True, if reading AO successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 2;
WORD AOValue[2]; //for example, we have a two-channel AO module
DWORD timeout = 200;

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500); // Need some time to wait for respond.
mtm_ReadAO(h, NetID, addr, ch_count, AOValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old ReadAO function has the declaration as below:

BOOL **mbm_ReadAO**(

tHandle h,

int addr,

int ch_count,

WORD *AOValue);

The mbm_ReadAO function does the the same job as the mtm_ReadAO function, except with

1. NetID: uses what mtm_Register specifies
2. timeout: uses the default value, 200 ms.

5.8 mtm_ReadAI

Read AI Values from the analog input module through Modbus communications.

➢ **Syntax**

```
BOOL mtm_ReadAI(
                tHandle h,
                int NetID,
                int addr,
                int ch_count,
                WORD *pwData,
                DWORD timeout
);
```

➢ **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Modbus TCP Network ID (usually 1 ~ 247)

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AI module

pwData

[Output] Specify the pointer to an array of WORD (2 byte) in which each element of the array represents the state of a single channel of an I/O module

timeout

[Input] Specify the value of the timeout value for the TCP communications. (unit: ms)
When accessing more I/O points, larger timeout value may be required depending on I/O device's performance

➢ **Return Values**

True, if reading AI successfully.

False, if not.

- **Examples**

[C]

```
int addr = 1;
int NetID = 1;
int ch_count = 2;
WORD AIValue[2]; //for example, we have a two-channel AI module
DWORD timeout = 200;

tHandle h = mtm_Register(NetID, TCP_IPADDR(10,1,102,64), 502);
hmi_DelayUS(500); // Need some time to wait for respond.
mtm_ReadAI(h, NetID, addr, ch_count, AIValue, timeout);
mtm_Unregister(h);
```

- **Remark**

Backward Compatibility:

The old ReadAI function has the declaration as below:

BOOL **mbm_ReadAI**(
tHandle h,
int addr,
int ch_count,
WORD *AIValue);

The mbm_ReadAI function does the the same job as the mtm_ReadAI function, except with

1. NetID: uses what mtm_Register specifies
2. timeout: uses the default value, 200 ms.

6. Modbus TCP Slave API

This chapter provides the Modbus RTU Slave APIs.
Modbus is a commonly-used serial communications in the industry field.

The following DioBuf and AioBuf buffers are data exchange spaces that are accessible by master and slave (TouchPAD). Master uses Modbus to read/write the buffer, while the slave (TouchPAD) diretly read/write the memory buffers. Make sure the master write to and the slave read from the same address (inside the buffer space), then the data exchange between master and slave is working.

6.1 mts_RegisterSlave

Register a Modbus slave operations on the TouchPAD.

- **Syntax**

```
BOOL mts_RegisterSlave(
        Unsigned char NetID,
        WORD DIO_StartAddr,
        WORD DIO_count,
        Char *pcDioBuf,
        WORD AIO_StartAddr,
        WORD AIO_count,
        WORD *pwAioBuf
);
```

- **Parameter**

NetID
[Input] Specify the Net ID of the Modbus communication

DIO_StartAddr
[Input] Specify the starting address for the Modbus communications of DI/DO

DIO_count
[Input] Specify the number of the channels for DI/DO

pcDioBuf
[Input/Output] Specify the pointer to an array of char in which each bit of every byte represents the status of a single DI or DO channel

AIO_StartAddr
[Input] Specify the starting address for the Modbus communications of AI/AO

AIO_count
[Input] Specify the number of the channels for AI/AO

pwAioBuf
[Input/Output] Specify the pointer to an array of WORD in which each word of the array represents the state of a single AI or AO channel

- **Return Values**

TRUE = OK,
FALSE = Parameter Error.

➢ Examples

[C]

```
#define PORT_BIND 502
HANDLE hPort = 0;
char DioBuf[2]; // 8-bit x2 (=16-bit) DIO (can be more)
// We arbitrarily take the first byte as the reserved byte.
// Users NEED NOT to do this if they have their own considerations.
// The first byte is used to store some information which is used by the
// host. In this example, we take
// bit 0: Initial flag, 1=initialized by master, 0 = not yet.
// bit 1~ 7: Reserved.
// bit 8~11: DIs; bit 12~15: DOs

WORD AioBuf[6]; // 16-bit x6 AIO buffers (can be more)

void Frame12OnCreate()
{
	if ( ! hPort ) // port is not opened?
		{
			hPort = hmi_TCPNew(); // Listen and wait for accepting
			hmi_TCPListen(hPort, PORT_BIND);
			memset(DioBuf, 0, sizeof(DioBuf)); // Clear buffer
			memset(AioBuf, 0, sizeof(AioBuf));
			mts_RegisterSlave(1, 0, 16, DioBuf, 0, 6, AioBuf);
		}
}

void Timer4OnExecute(tWidget *pWidget)
{
	unsigned long mts_status;
	static int iCnt = 0;

	if ( hPort ) // port is opened or listening?
	{
	// retrieve Modbus TCP command and process it
		if ( (mts_status & MTS_DIO_DIRTY ) // DIO is updated
		|| ( iCnt == 0 ) ) // First time, or periodically updating
		{
			// Bit 12 ~ 15 as DO
			CheckBoxSelectedSet(&CheckBox5, DioBuf[1] & (1 << 4));
			CheckBoxSelectedSet(&CheckBox7, DioBuf[1] & (1 << 5));
			CheckBoxSelectedSet(&CheckBox8, DioBuf[1] & (1 << 6));
			CheckBoxSelectedSet(&CheckBox9, DioBuf[1] & (1 << 7));
		}
	}
	iCnt ++;
	if (iCnt > 10) iCnt = 0;
}
```

➢ Remark

1. Set 0 to both parameters, DIO_count and AIO_count, to **unregister** those slave functions.
2. Currently, we support only one slave device on TouchPAD.

6.2 mts_ProcessCmd

Process the Modbus TCP Slave command (suggested for every 10 ms).

➢ **Syntax**

```
unsigned long mts_ProcessCmd(
                    Handle h
);
```

➢ **Parameter**

h

[Input] Specify the handle opened by the TCP_Open function

➢ **Return Values**

The returning unsigned long value has 32 bits, in which

Bit that is set	Representing macros	Descriptions
0	MTS_DIO_DIRTY	DIO is updated
1	MTS_AIO_DIRTY	AIO is updated
8	MTS_DIO_READ	Read DI
9	MTS_AIO_READ	Read AI
31	MTS_CMD_ERROR	Error (0x8000 0000)

➢ **Examples**

[C]

```
#define PORT_BIND 502
HANDLE hPort = 0;
char DioBuf[2]; // 8-bit x2 (=16-bit) DIO (can be more)
// We arbitrarily take the first byte as the reserved byte.
// Users NEED NOT to do this if they have their own considerations.
// The first byte is used to store some information which is used by the
// host. In this example, we take
// bit 0: Initial flag, 1=initialized by master, 0 = not yet.
// bit 1~ 7: Reserved.
// bit 8~11: DIs; bit 12~15: DOs

WORD      AioBuf[6]; // 16-bit x6 AIO buffers (can be more)
```



```
void Frame12OnCreate()
{
   if ( ! hPort )    // port is not opened?
   {
         hPort = hmi_TCPNew(); // Listen and wait for accepting
         hmi_TCPListen(hPort, PORT_BIND);
         memset(DioBuf, 0, sizeof(DioBuf));    // Clear buffer
         memset(AioBuf, 0, sizeof(AioBuf));
         mts_RegisterSlave(1, 0, 16, DioBuf, 0, 6, AioBuf);
   }
}

void Timer4OnExecute(tWidget *pWidget)
{
   unsigned long mts_status;
   static int iCnt = 0;

   if ( hPort )    // port is opened or listening?
   {
       // retrieve Modbus TCP command and process it
      mts_status = mts_ProcessCmd(hPort);
       if ( (mts_status & MTS_DIO_DIRTY )    // DIO is updated
       ||     ( iCnt == 0 )    )    // First time, or periodically updating
       {
           // Bit 12 ~ 15 as DO
           CheckBoxSelectedSet(&CheckBox5, DioBuf[1] & (1 << 4));
           CheckBoxSelectedSet(&CheckBox7, DioBuf[1] & (1 << 5));
           CheckBoxSelectedSet(&CheckBox8, DioBuf[1] & (1 << 6));
            CheckBoxSelectedSet(&CheckBox9, DioBuf[1] & (1 << 7));
       }
   }
   iCnt ++;
   if (iCnt > 10) iCnt = 0;
}
```

- **Remark**

  None

6.3 mts_GetIOStatus

Process the Modbus TCP Slave command (suggested for every 10 ms).

- **Syntax**

```
DWORD mts_GetIOStatus(
                Unsigned char NetID
);
```

- **Parameter**

NetID

[Input] Specify the Net ID of the TouchPAD (in slave mode)

- **Return Values**

The returning unsigned long value has 32 bits, in which

Bit that is set	Representing macros	Descriptions
0	MTS_DIO_DIRTY	DIO is updated
1	MTS_AIO_DIRTY	AIO is updated
8	MTS_DIO_READ	Read DI
9	MTS_AIO_READ	Read AI
31	MTS_CMD_ERROR	Error (0x8000 0000)

- **Examples**

[C]

```
HANDLE h;
…
if(mts_GetIOStatus(h) & MTS_CMD_ERROR)
LabelTextSet(&Label5, "Error!");
…
```

- **Remark**

None

7. Modbus RTU Master API

This chapter provides the Modbus RTU Master APIs.

Modbus is a commonly-used serial communications in the industry field.

Mapping between the function code and the Modbus RTU Master API

Function Code	HMIWorks Modbus RTU Master API
1	mrm_ReadDO
2	mrm_ReadDI
3	mrm_ReadAO
4	mrm_ReadAI
5	mrm_WriteDO with ch_count = 1
6	mrm_WriteAO with ch_count = 1
15	mrm_WriteDO with ch_count > 1
16	mrm_WriteAO with ch_count > 1

7.1 mrm_WriteDO

Write DO Value to the digital output module through Modbus communications.

- **Syntax**

```
BOOL mrm_WriteDO(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        char *data
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DO module

data

[Input] Specify the pointer to an array in which the least significant bit of the first element represents the channel 0, the second lowest bit represents the channel 1, etc. Each 8-channel DI/DO uses a byte to store the data. Channel 0 ~ 7 are stored in data[0], channel 8 ~ 15 are stored in data[1], and so on.

For example,
if we turn on only channel 0 and channel 1, data[0] has the value of 3 (whose binary equivalent is 0000,0011).

- **Return Values**

True, if writing DO successfully.

False, if not.

- **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
int ch_count = 8;
char DO_value[1];
DO_value[0] = 3; //that is, turn on the ch 0 and ch1.

h = uart_Open("COM1,9600,N,8,1");
mrm_WriteDO (h, NetID, addr, ch_count, DO_value);
uart_Close(h);
```

- **Remark**

None

7.2 mrm_ReadDO

Read DO Value from the digital output module through Modbus communications.

- **Syntax**

```
BOOL mrm_ReadDO(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        char *data
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DO module

data

[Output] Specify the pointer to an array in which the least significant bit of the first element represents the channel 0, the second lowest bit represents the channel 1, etc. Each 8-channel DI/DO uses a byte to store the data. Channel 0 ~ 7 are stored in data[0], channel 8 ~ 15 are stored in data[1], and so on.

For example, if we turn on only channel 0 and channel 1, data[0] has the value of 3 (whose binary equivalent is 0000,0011).

- **Return Values**

True, if reading DO successfully.

False, if not.

- **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
int ch_count = 8;
char DO_value[1];

h = uart_Open("COM1,9600,N,8,1");
mrm_ReadDO (h, NetID, addr, ch_count, DO_value);
uart_Close(h);
```

- **Remark**

None

7.3 mrm_ReadDI

Read DI Value from the digital input module through Modbus communications.

- **Syntax**

```
BOOL mrm_ReadDI(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        char *data
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the DI module

data

[Output] Specify the pointer to an array in which the least significant bit of the first element represents the channel 0, the second lowest bit represents the channel 1, etc. Each 8-channel DI/DO uses a byte to store the data. Channel 0 ~ 7 are stored in data[0], channel 8 ~ 15 are stored in data[1], and so on.

For example, if we only channel 0 and channel 1 are input, data[0] has the value of 3 (whose binary equivalent is 0000,0011).

- **Return Values**

True, if reading DI successfully.

False, if not.

- **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
int ch_count = 8;
char DI_value[1];

h = uart_Open("COM1,9600,N,8,1");
mrm_ReadDI (h, NetID, addr, ch_count, DI_value);
uart_Close(h);
```

- **Remark**

None

7.4 mrm_WriteAO

Write AO Value to the analog output module through Modbus communications.

➢ **Syntax**

```
BOOL mrm_WriteAO(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        WORD *AO_value
);
```

➢ **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AO module

AO_value

[Input] Specify the pointer to an array whose values are the Analog Outputs. Each AI/AO channel uses a WORD type to store a data and the data format strongly depends on the devices.

➢ **Return Values**

True, if writing DO successfully.

False, if not.

➢ **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
```

 ✉ E-mail: service@icpdas.com

```
int ch_count = 2;
WORD AO_value[2]; //for example, we have a two-channel AO module

AO_value[0] = 100; //arbitrarily set channel 0, simply for example
AO_value[1] = 120; //arbitrarily set channel 1, simply for example

h = uart_Open("COM1,9600,N,8,1");
mrm_WriteAO (h, NetID, addr, ch_count, AO_value);
uart_Close(h);
```

- **Remark**

  None

7.5 mrm_ReadAO

Read AO Value from the analog output module through Mdbus communications.

- **Syntax**

```
BOOL mrm_ReadAO(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        WORD *AO_value
);
```

- **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AO module

AO_value

[Output] Specify the pointer to an array whose values are the Analog Outputs. Each AI/AO channel uses a WORD type to store a data and the data format strongly depends on the devices.

- **Return Values**

True, if reading AO successfully.

False, if not.

- **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
```



```
int ch_count = 2;
WORD AO_value[2]; //for example, we have a two-channel AO module

h = uart_Open("COM1,9600,N,8,1");
mrm_ReadAO (h, NetID, addr, ch_count, AO_value);
uart_Close(h);
```

- **Remark**

  None

7.6 mrm_ReadAI

Read AI Value from the analog input module through Modbus communications.

➢ **Syntax**

```
BOOL mrm_ReadAI(
        HANDLE h,
        int NetID,
        int addr,
        int ch_count,
        WORD *AI_value
);
```

➢ **Parameter**

h

[Input] Specify the handle to the Modbus communication

NetID

[Input] Specify the Net ID of the Modbus communication

addr

[Input] Specify the starting address of the Modbus communication

ch_count

[Input] Specify the number of the channels of the AI module

AI_value

[Output] Specify the pointer to an array whose values are the Analog Inputs. Each AI/AO channel uses a WORD type to store a data and the data format strongly depends on the devices.

➢ **Return Values**

True, if reading AI successfully.

False, if not.

➢ **Examples**

[C]

```
HANDLE h;
int NetID = 1;
int addr = 1;
```



```
int ch_count = 2;
WORD AI_value[2]; //for example, we have a two-channel AI module

h = uart_Open("COM1,9600,N,8,1");
mrm_ReadAI (h, NetID, addr, ch_count, AI_value);
uart_Close(h);
```

- **Remark**

  None

8. Modbus RTU Slave API

This chapter provides the Modbus RTU Slave APIs.
Modbus is a commonly-used serial communications in the industry field.

The following DioBuf and AioBuf buffers are data exchange spaces that are accessible by master and slave (TouchPAD). Master uses Modbus to read/write the buffer, while the slave (TouchPAD) diretly read/write the memory buffers. Make sure the master write to and the slave read from the same address (inside the buffer space), then the data exchange between master and slave is working.

8.1 mrs_RegisterSlave

Register a Modbus slave operations on the TouchPAD.

- **Syntax**

```
BOOL mrs_RegisterSlave(
        Unsigned char NetID,
        WORD DIO_StartAddr,
        WORD DIO_count,
        Char *pcDioBuf,
        WORD AIO_StartAddr,
        WORD AIO_count,
        WORD *pwAioBuf
);
```

- **Parameter**

NetID
[Input] Specify the Net ID of the Modbus communication

DIO_StartAddr
[Input] Specify the starting address for the Modbus communications of DI/DO

DIO_count
[Input] Specify the number of the channels for DI/DO

pcDioBuf
[Input/Output] Specify the pointer to an array of char in which each bit of every byte represents the status of a single DI or DO channel

AIO_StartAddr
[Input] Specify the starting address for the Modbus communications of AI/AO

AIO_count
[Input] Specify the number of the channels for AI/AO

pwAioBuf
[Input/Output] Specify the pointer to an array of WORD in which each word of the array represents the state of a single AI or AO channel

- **Return Values**

TRUE = OK,
FALSE = Parameter Error.

➢ Examples

[C]

```
HANDLE hPort = 0;
char DioBuf[2]; // 8-bit x2 (=16-bit) DIO (can be more)
// We arbitrarily take the first byte as the reserved byte.
// Users NEED NOT to do this if they have their own considerations.
// The first byte is used to store some information which is used by the
// host. In this example, we take
// bit 0: Initial flag, 1=initialized by master, 0 = not yet.
// bit 1~ 7: Reserved.
// bit 8~11: DIs; bit 12~15: DOs

WORD      AioBuf[6]; // 16-bit x6 AIO buffers (can be more)

void Frame12OnCreate()
{
   if ( ! hPort )    // port is not opened?
   {
         hPort = uart_Open("COM1,115200,N,8,1");    // Open com port
         memset(DioBuf, 0, sizeof(DioBuf));    // Clear buffer
         memset(AioBuf, 0, sizeof(AioBuf));
         mrs_RegisterSlave(1, 0, 16, DioBuf, 0, 6, AioBuf);
   }
}

void Timer4OnExecute(tWidget *pWidget)
{
   unsigned long mrs_status;
   static int iCnt = 0;
    if ( hPort )    // port is opened?
   {
        // retrieve Modbus RTU command and process it
       mrs_status = mrs_ProcessCmd(hPort);
        if ( (mrs_status & MRS_DIO_DIRTY )    // DIO is updated
        ||      ( iCnt == 0 )    )    // First time, or periodically updating
        {
            // Bit 12 ~ 15 as DO
            CheckBoxSelectedSet(&CheckBox5, DioBuf[1] & (1 << 4));
            CheckBoxSelectedSet(&CheckBox7, DioBuf[1] & (1 << 5));
            CheckBoxSelectedSet(&CheckBox8, DioBuf[1] & (1 << 6));
            CheckBoxSelectedSet(&CheckBox9, DioBuf[1] & (1 << 7));
        }
   }
   iCnt ++;
   if (iCnt > 10) iCnt = 0;
}
```

➢ Remark

1. Set 0 to both parameters, DIO_count and AIO_count, to **unregister** those slave functions.
2. Currently, we support only one slave device on TouchPAD.

8.2 mrs_ProcessCmd

Process the Modbus RTU Slave command (suggested for every 10 ms).

- **Syntax**

```
unsigned long mrs_ProcessCmd(
                    Handle h
);
```

- **Parameter**

h

[Input] Specify the handle opened by the uart_Open function

- **Return Values**

The returning unsigned long value has 32 bits, in which

Bit that is set	Representing macros	Descriptions
0	MRS_DIO_DIRTY	DIO is updated
1	MRS_AIO_DIRTY	AIO is updated
8	MRS_DIO_READ	Read DI
9	MRS_AIO_READ	Read AI
31	MRS_CMD_ERROR	Error (0x8000 0000)

- **Examples**

[C]

```
HANDLE hPort = 0;
char DioBuf[2]; // 8-bit x2 (=16-bit) DIO (can be more)
// We arbitrarily take the first byte as the reserved byte.
// Users NEED NOT to do this if they have their own considerations.
// The first byte is used to store some information which is used by the
// host. In this example, we take
// bit 0: Initial flag, 1=initialized by master, 0 = not yet.
// bit 1~ 7: Reserved.
// bit 8~11: DIs; bit 12~15: DOs

WORD      AioBuf[6]; // 16-bit x6 AIO buffers (can be more)
```

```
void Frame12OnCreate()
{
   if ( ! hPort )    // port is not opened?
   {
         hPort = uart_Open("COM1,115200,N,8,1");    // Open com port
         memset(DioBuf, 0, sizeof(DioBuf));    // Clear buffer
         memset(AioBuf, 0, sizeof(AioBuf));
         mrs_RegisterSlave(1, 0, 16, DioBuf, 0, 6, AioBuf);
   }
}

void Timer4OnExecute(tWidget *pWidget)
{
   unsigned long mrs_status;
   static int iCnt = 0;

   if ( hPort )    // port is opened?
   {
       // retrieve Modbus RTU command and process it
      mrs_status = mrs_ProcessCmd(hPort);
       if ( (mrs_status & MRS_DIO_DIRTY )    // DIO is updated
       ||     ( iCnt == 0 )    )    // First time, or periodically updating
       {
           // Bit 12 ~ 15 as DO
           CheckBoxSelectedSet(&CheckBox5, DioBuf[1] & (1 << 4));
           CheckBoxSelectedSet(&CheckBox7, DioBuf[1] & (1 << 5));
           CheckBoxSelectedSet(&CheckBox8, DioBuf[1] & (1 << 6));
           CheckBoxSelectedSet(&CheckBox9, DioBuf[1] & (1 << 7));
       }
}
   iCnt ++;
   if (iCnt > 10) iCnt = 0;
}
```

- **Remark**

  None

8.3 mrs_GetIOStatus

Process the Modbus RTU Slave command (suggested for every 10 ms).

- **Syntax**

```
unsigned long mrs_GetIOStatus(
                    Handle h
);
```

- **Parameter**

h

[Input] Specify the handle opened by the uart_Open function

- **Return Values**

The returning unsigned long value has 32 bits, in which

Bit that is set	Representing macros	Descriptions
0	MRS_DIO_DIRTY	DIO is updated
1	MRS_AIO_DIRTY	AIO is updated
8	MRS_DIO_READ	Read DI
9	MRS_AIO_READ	Read AI
31	MRS_CMD_ERROR	Error (0x8000 0000)

- **Examples**

[C]

```
HANDLE h;
…
if(mrs_GetIOStatus(h) & MRS_CMD_ERROR)
LabelTextSet(&Label5, "Error!");
…
```

- **Remark**

None

9. UART API

This chapter introduces UART APIs.

This set of UART APIs is designed for the serial port in TouchPAD.

Uart Reference

Uart operations include basic management operations, such as opening, sending, receiving, and closing. The following topics describe how you can operate uart programmatically using the uart functions.

Classification

Functions	Descriptions	Memo
uart_Send uart_Recv	Used for sending/receiving the DCON or ASCII commands whichs are NULL terminated strings. They add/check the checksum if enabled. They also add a terminator when sending and reading until the terminator received. The DCON commands always terminated at 0xD (CR, Carriage Return).	-
uart_BinSend uart_BinRecv	Used for sending/receiving binary data or ASCII strings whchs have a specified length of data.	-

Obsoleted Functions

Functions	Descriptions	Memo
uart_Write uart_Read	Used for sending/receiving ASCII string. Both functions are without checksum and the uart_Read returns if 0xD (CR, Carriage Return) is received.	We suggest users to use uart_Send and uart_Recv instead of uart_Write and uart_Read.
uart_BinWrite uart_BinRead	uart_BinWrite is exactly the same as uart_BinSend and uart_BinRead is exactly the same as uart_BinRecv.	We suggest users to use uart_BinSend and uart_BinRecv instead of uart_BinWrite and uart_BinRead.

9.1 uart_Open

This function opens the COM port and specifies the baud rate, parity bits, data bits, and stop bits.

- **Syntax**

```
HANDLE uart_Open(
        LPCSTR ConnectionString
);
```

- **Parameter**

ConnectionString

[Input] Specifies the COM port, baud rate, parity bits, data bits, and stop bits.
The default setting is COM1,115200,N,8,1.
The format of *ConnectionString* is as follows:
"**com_port**, **baud_rate**, **parity_bits**, **data_bits**, **stop_bits**"

com_port	COM1, COM2, COM3….
baud_rate	1200/2400/4800/9600/38400/57600/115200
parity_bits	N (NOPARITY) O (ODDPARITY) E (EVENPARITY) M (MARKPARITY) S (SPACEPARITY)
data_bits	5/6/7/8
stop_bits	1 (ONESTOPBIT) 2 (TWOSTOPBITS)

- **Return Values**

Returns a handle value (> 0) of the opened COM port.
Returns 0 when error.

- **Examples**

[C]

```
HANDLE hOpen;
hOpen = uart_Open("COM1,9600,N,8,1");
```

- **Remark**

None

9.2 uart_Close

This function closes the COM port which has been opened.

- **Syntax**

```
BOOL uart_Close(
        HANDLE hPort
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_Close(hOpen);
```

- **Remark**

None

9.3 uart_Send

This function sends data through the COM port which have been opened. The sending string is automatically appended with the DCON checksum (2 byte) if it is enabled by uart_EnableCheckSum and appended with a terminating character (default: CR) if it is set by uart_SetTerminator.

- **Syntax**

```
BOOL uart_Send(
        HANDLE hPort,
        LPSTR buf
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

buf

[Input] A pointer to a buffer that send the data (null-terminated)

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length = 100;
char buf[Length];
hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_Send(hOpen, buf);
uart_Close(hOpen);
```

- **Remark**

None

9.4 uart_Recv

This function retrieves data through the COM port which have been opened.

- **Syntax**

```
BOOL uart_Recv(
        HANDLE hPort,
        LPSTR buf
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

buf

[Output] A pointer to a buffer that receives the data (null-terminated)

Make sure you have allocated enough buffer space for storing received data.

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length = 100;
char buf[Length];
hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_Recv(hOpen, buf);
uart_Close(hOpen);
```

- **Remark**

Both uart_EnableCheckSum and uart_SetTimeout can influence the behavior of the uart_Send and uart_Recv functions.

Though **FALSE** returned if timeout occurs or checksum has errors, uart_Recv still fill the data into the buffer (the second parameter). Of course, the data of the buffer may not be useful.

9.5 uart_SendCmd

This function sends commands through the COM port which have been opened and then receive data from the COM port.
This function mainly consists of 3 functions, uart_Purge, uart_Send and uart_Recv.

- **Syntax**

```
BOOL uart_SendCmd(
        HANDLE hPort,
        LPSTR cmd,
        LPSTR szResult
);
```

- **Parameter**

hPort
[Input] Handle to the opened COM port

cmd
[Input] A pointer to a command (null-terminated)

szResult
[Output] A pointer to a user allocated buffer that receives the data (null-terminated)
Make sure there is enough buffer space for storing the received data.

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length = 100;
char buf[Length];
hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_SendCmd(hOpen,"$00M", buf);      // $00M: ask the device name
uart_Close(hOpen);
```

- **Remark**

None

9.6 uart_SetTimeout

This function sets the timeout timer. These uart functions will stop receiving/waiting data when timed out.

- **Syntax**

```
BOOL uart_SetTimeout(
        HANDLE hPort,
        DWORD msec
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

msec

[Input] Millisecond to the timer

- **Return Values**

None

- **Examples**

[C]

```
HANDLE hOpen;
DWORD mes;

hOpen = uart_Open("COM1,9600,N,8,1");
uart_SetTimeout(hOpen, mes);
```

- **Remark**

None

9.7 uart_EnableCheckSum

This function turns on the DCON checksum or not.

- **Syntax**

```
BOOL uart_EnableCheckSum(
        HANDLE hPort,
        BOOL bEnable
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

bEnable

[Input] Enable DCON checksum when TRUE, and Disable when FALSE.

- **Return Values**

None

- **Examples**

[C]

```
HANDLE hUart;
char result[32];

hUart = uart_Open("");
uart_EnableCheckSum(hUart , true);
uart_SendCmd(hUart, "$00M", result);
uart_Close(hUart);
```

- **Remark**

uart_EnableCheckSum does not apply to the binary UART API functions, that is, uart_BinSend, uart_BinRecv, and uart_BinSendCmd.

9.8 uart_SetTerminator

This function sets the termination character. This setting is valid for uart_Send, uart_Recv and uart_SendCmd functions.

- **Syntax**

```
BOOL uart_SetTerminator(
        HANDLE hPort,
        LPCSTR szTerm
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

szTerm

[Input] Pointer to the termination character (1-byte string).
Default is CR.

- **Return Values**

None

- **Examples**

[C]

```
HANDLE hUart;
char result[32];

hUart = uart_Open("");
uart_SetTerminator(hUart, "\r");
uart_SendCmd(hUart, "$00M", result);
uart_Close(hUart);
```

- **Remark**

None

9.9 uart_BinSend

This function sends binary data through the COM port which have been opened.

- **Syntax**

```
BOOL uart_BinSend(
                HANDLE hPort,
                LPSTR buf,
                int buf_len
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

buf

[Input] A pointer to a user allocated buffer that send the data

buf_len

[Input] The length of the buffer

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length = 100;
char buf[Length];

hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_BinSend(hOpen, buf, Length);
uart_Close(hOpen);
```

- **Remark**

uart_BinSend does not support uart_EnableCheckSum.

9.10 uart_BinRecv

This function retrieves binary data through the COM port which have been opened.

- **Syntax**

```
BOOL uart_BinRecv(
                HANDLE hPort,
                LPSTR buf,
                int buf_len
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

buf

[Output] A pointer to a user allocated buffer that receives the data

buf_len

[Input] The length of the buffer

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length = 100;
char buf[Length];

hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_BinRecv(hOpen, buf, Length);
uart_Close(hOpen);
```

- **Remark**

uart_BinRecv does not support uart_EnableCheckSum.

9.11 uart_BinSendCmd

This function sends binary commands through the COM port which have been opened and then receive data from the COM port.

- **Syntax**

```
BOOL uart_BinSendCmd(
                HANDLE hPort,
                LPSTR cmd,
                int cmd_len,
                LPSTR buf,
                int buf_len
);
```

- **Parameter**

hPort
[Input] Handle to the opened COM port

cmd
[Input] A pointer to a command

cmd_len
[Input] The length of the command

buf
[Output] A pointer to a user allocated buffer that receives the data

buf_len
[Input] The length of the buffer

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
BOOL ret;
HANDLE hOpen;
int Length1 = 100;
int Length2 = 100;
char cmd[Length1];
char buf[Length2];

hOpen = uart_Open("COM1,9600,N,8,1");
ret = uart_BinSendCmd(hOpen,"$00M", Length1, buf, Length2);
// $00M: ask the device name
uart_Close(hOpen);
```

- **Remark**

uart_BinSendCmd does not support uart_EnableCheckSum.

9.12 uart_GetRxDataCount

This function returns the count of bytes which are presently in the receiver buffer.

- **Syntax**

```
Unsigned long uart_GetRxDataCount(
                        HANDLE hPort
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

- **Return Values**

The count of bytes which are presently in the receiver buffer

- **Examples**

[C]

```
// The example of an echo server

HANDLE hOpen=-1;
#define LENGTH 20 //256
int iRxData=0;

void Timer5OnExecute(tWidget *pWidget)
{
    //--- The buffer to storage the data, it's size is User-defined value ---
    static char recv_str[LENGTH];
    int res=0,ret=0;
    //--- If handle is invalid, return ---
    if(hOpen < 0)    return;
    //--- If no data received, return ---
    if(uart_GetRxDataCount(hOpen)==0) return;
    //--- Check whether the data has been transferred completely      ---
    if (iRxData != uart_GetRxDataCount(hOpen))
    {
      iRxData = uart_GetRxDataCount(hOpen);
      return;
    }

    //--- Make sure the message don't overflow the buffer ---
    iRxData = (iRxData<LENGTH)?iRxData:LENGTH;

    //--- Receive the data from COM port ---
    res = uart_BinRecv(hOpen, recv_str,iRxData);
    recv_str[iRxData]=0;
```

```
        //--- Purge Rx Buffer ---
        uart_Purge(hOpen, 0, 1);

        //--- Process all recevied data ---
        if (res)
        {
            hmi_Beep();
            //--- echo the received message ---
            LabelTextSet(&Label4, recv_str);
            ret = uart_BinSend(hOpen, recv_str, iRxData);
            iRxData = 0;

            if (ret)    LabelTextSet(&Label9, "Echo OK");
            else        LabelTextSet(&Label9, "Echo Error");
        }
    }

    void BitButton10OnClick(tWidget *pWidget)    // Start
    {
        //--- Close the existing handle ---
      if(hOpen>=0)
      {
          uart_Close(hOpen);
          hOpen = -1;
      }

      //--- Establish a new handle ---
      if(hOpen<0)
      {
          hOpen = uart_Open("COM1,115200,N,8,1");
      }

      //---    If success, display current COM Port settings on screen ---
      if(hOpen>=0)
      {
         LabelTextSet(&Label8, "COM1,115200,N,8,1");
         uart_SetTimeOut(hOpen, 300);
      }
    }

    void BitButton11OnClick(tWidget *pWidget)    // Stop
    {
      if(hOpen>=0)
      {
          uart_Close(hOpen);
          hOpen = -1;
          LabelTextSet(&Label8, "Press 'Start' to Begin");
          LabelTextSet(&Label4, "");
          LabelTextSet(&Label9, "");
      }
    }
```

- **Remark**

  None

9.13 uart_Purge

This function sends binary commands through the COM port which have been opened and then receive data from the COM port.

- **Syntax**

```
int uart_Purge(
        HANDLE hPort,
        int ClearTx,
        int ClearRx
);
```

- **Parameter**

hPort

[Input] Handle to the opened COM port

ClearTx

[Input] A integer to tell the uart_Purge to clear the transmitter buffer.

Possible value:	
0	DO NOT clear
Otherwise	DO clear

ClearRx

[Input] A integer to tell the uart_Purge to clear the receiver buffer.

Possible value:	
0	DO NOT clear
Otherwise	DO clear

- **Return Values**

It always returns zero.

The returning value is reserved for future use.

- **Examples**

[C]

```
// The example of an echo server

HANDLE hOpen=-1;
#define LENGTH 20 //256
```

```
int iRxData=0;

void Timer5OnExecute(tWidget *pWidget)
{
    //--- The buffer to storage the data, it's size is User-defined value ---
    static char recv_str[LENGTH];
    int res=0,ret=0;

    //--- If handle is invalid, return ---
    if(hOpen < 0)    return;
    //--- If no data received, return ---
    if(uart_GetRxDataCount(hOpen)==0) return;
    //--- Check whether the data has been transferred completely      ---
    if (iRxData != uart_GetRxDataCount(hOpen))
    {
      iRxData = uart_GetRxDataCount(hOpen);
      return;
    }

    //--- Make sure the message don't overflow the buffer ---
    iRxData = (iRxData<LENGTH)?iRxData:LENGTH;

    //--- Receive the data from COM port ---
    res = uart_BinRecv(hOpen, recv_str,iRxData);
    recv_str[iRxData]=0;
    //--- Purge Rx Buffer ---
    uart_Purge(hOpen, 0, 1);

    //--- Process all recevied data ---
    if (res)
    {
        hmi_Beep();
        //--- echo the received message ---
        LabelTextSet(&Label4, recv_str);
        ret = uart_BinSend(hOpen, recv_str, iRxData);
        iRxData = 0;

        if (ret)    LabelTextSet(&Label9, "Echo OK");
        else        LabelTextSet(&Label9, "Echo Error");
    }
}


void BitButton10OnClick(tWidget *pWidget)    // Start
{
    //--- Close the existing handle ---
   if(hOpen>=0)
   {
       uart_Close(hOpen);
       hOpen = -1;
   }
```

```
    //--- Establish a new handle ---
    if(hOpen<0)
    {
        hOpen = uart_Open("COM1,115200,N,8,1");
    }

    //---    If success, display current COM Port settings on screen ---
    if(hOpen>=0)
    {
       LabelTextSet(&Label8, "COM1,115200,N,8,1");
       uart_SetTimeOut(hOpen, 300);
    }
}

void BitButton11OnClick(tWidget *pWidget)    // Stop
{
    if(hOpen>=0)
    {
        uart_Close(hOpen);
        hOpen = -1;
        LabelTextSet(&Label8, "Press 'Start' to Begin");
        LabelTextSet(&Label4, "");
        LabelTextSet(&Label9, "");
    }
}
```

- **Remark**

  None

10. DCON_IP API

DCON_IO Reference

DCON_IO API supports to operate I-7000 series I/O modules of ICP DAS.
For more details of I-7000 series:
http://www.icpdas.com/products/Remote_IO/i-7000/i-7000_introduction.htm

10.1 dcon_WriteDO

This function writes the DO values to DO modules.

- **Syntax**

```
BOOL dcon_WriteDO(
                HANDLE hPort,
                int iAddress,
                int iDO_TotalCh,
                DWORD iDO_Value
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iDO_TotalCh
[Input] The total number of DO channels of the DO modules

iDO_Value
[Input] The value which is the binary representation of Dos

iDO_Value	Descriptions
1	Turn on the DO channel
0	OFF

- **Return Values**

  TRUE indicates success.

  FALSE indicates failure.

- **Examples**

  [C]

```
HANDLE hPort;
int addr = 1;

int total_channel = 8;
DWORD do_value = 4; // turn on the channel two

hPort = uart_Open("COM3,9600");
BOOL ret = dcon_WriteDO(hPort, addr, total_channel, do_value);
uart_Close(hPort);
```

- **Remark**

  None

10.2 dcon_WriteDOBit

This function writes a single bit of value to the DO module, that is, only the channel corresponding to the bit is changed.

- **Syntax**

```
BOOL dcon_WriteDOBit(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iDO_TotalCh,
                int iBitValue
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iChannel

[Input] The DO channel to change

iDO_TotalCh

[Input] The total number of DO channels of the DO modules

iBitValue

[Input]

iBitValue	Descriptions
1	Turn on the DO channel
0	OFF

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
int iAddress = 1;
int iChannel = 2;
int iDO_TotalCh = 8;
int iBitValue = 1;

hPort = uart_Open("COM1,115200");
BOOL ret = dcon_WriteDOBit(hPort, iAddress, iChannel, iDO_TotalCh, iBitValue);
uart_Close(hPort);
```

- **Remark**

None

10.3 dcon_ReadDO

This function reads the DO value of the DO module.

- **Syntax**

```
BOOL dcon_ReadDO(
                HANDLE hPort,
                int iAddress,
                int iDO_TotalCh,
                DWORD *iDO_Value
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iDO_TotalCh
[Input] The total number of DO channels of the DO modules

iDO_Value
[Output] The pointer to the DO value to read from the DO module.
The DO value is the binary representation of DOs.

- **Return Values**

TRUE indicates success. **FALSE** indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress = 1;
int total_channel = 8;
DWORD do_value;

hPort = uart_Open("COM1,115200");
BOOL ret = dcon_ReadDO(hPort, iAddress, total_channel, &do_value );
uart_Close(hPort);
```

- **Remark**

None

10.4 dcon_ReadDI

This function reads the DI value of the DI module.

- **Syntax**

```
BOOL dcon_ReadDI(
                HANDLE hPort,
                int iAddress,
                int iDI_TotalCh,
                DWORD *iDI_Value
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iDI_TotalCh

[Input] The total number of DI channels of the DI modules

iDI_Value

[Output] The pointer to read-back value which is the binary representation of DIs

iDI_Value	Descriptions
1	High
0	Low

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress = 2;
```

```
int iDI_TotalCh = 8;
DWORD lDI_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadDI(hPort, iAddress, iDI_TotalCh, &lDI_Value);
uart_Close(hPort);
```

- **Remark**

  None

10.5 dcon_ReadDIO

This function reads the DI and the DO values of the DIO module.

➢ **Syntax**

```
BOOL dcon_ReadDIO(
                HANDLE hPort,
                int iAddress,
                int iDI_TotalCh,
                int iDO_TotalCh,
                DWORD *iDI_Value,
                DWORD *iDO_Value
);
```

➢ **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iDI_TotalCh

[Input] The total number of DI channels of the DIO module

iDO_TotalCh

[Input] The total number of DO channels of the DIO module

iDI_Value

[Output] The pointer to the read-back value which is the binary representation of DIs

iDI_Value	Descriptions
1	High
0	Low

iDO_Value

[Output] The pointer to the read-back value which is the binary representation of DOs

iDO_Value	Descriptions
1	High
0	Low

- **Return Values**

  TRUE indicates success.

  FALSE indicates failure.

- **Examples**

  [C]

```
HANDLE hPort;
BYTE iAddress=1;
int iDI_TotalCh=8;
int iDO_TotalCh=8;
DWORD lDI_Value;
DWORD lDO_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadDIO(hPort, iAddress, iDI_TotalCh, iDO_TotalCh, &lDI_Value, &lDO_Value);
uart_Close(hPort);
```

- **Remark**

  None

10.6 dcon_ReadDILatch

This function reads the DI latch value of the DI module.

➢ **Syntax**

```
BOOL dcon_ReadDILatch(
                HANDLE hPort,
                int iAddress,
                int iDI_TotalCh,
                int iLatchType,
                DWORD *iDI_Latch_Value
);
```

➢ **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iDI_TotalCh

[Input] The total number of DI channels of the DI module

iLatchType

[Input] The latch type specified to read latch value back

iLatchType	Descriptions
1	high status latched
0	low status latched

iDI_Latch_Value

[Output] The pointer to the latch value read back from the DI module

The latch value of a particular channel is

1 if there's at least one time that the DI channel is high for latch type = 1;

0 if there's at least one time that the DI channel is low for latch type = 0.

Take latch value of each channel as a bit of a binary value, then the binary value is the DI latch value.

- **Return Values**

  TRUE indicates success.

  FALSE indicates failure.

- **Examples**

  [C]

```
HANDLE hPort;
BYTE iAddress=1;
int iDI_TotalCh=8;
int iLatchType=0;
DWORD lDI_Latch_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadDILatch(hPort, iAddress, iDI_TotalCh, iLatchType, &lDI_Latch_Value);
uart_Close(hPort);
```

- **Remark**

  None

10.7 dcon_ClearDILatch

This function clears the latch value of the DI module.

- **Syntax**

```
BOOL dcon_ClearDILatch(
                HANDLE hPort,
                int iAddress,
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ClearDILatch(hPort, iAddress);
uart_Close(hPort);
```

- **Remark**

None

10.8 dcon_ReadDIOLatch

This function reads the latch values of the DI and DO channels of the DIO module.

➢ **Syntax**

```
BOOL dcon_ReadDIOLatch(
                HANDLE hPort,
                int iAddress,
                int iDI_TotalCh,
                int iDO_TotalCh,
                int iLatchType,
                DWORD *iDI_Latch_Value,
                DWORD *iDO_Latch_Value
);
```

➢ **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iDI_TotalCh

[Input] The total number of DI channels of the DIO module

iDO_TotalCh

[Input] The total number of DO channels of the DIO module

iLatchType

[Input] The type of the latch value read back

iLatchType	Descriptions
1	high status latched
0	low status latched

iDI_Latch_Value

[Output] The pointer to the latch value read back from the DI channels of DIO module.
The latch value of a particular channel is
1 if there's at least one time that the DI channel is high for latch type = 1;
0 if there's at least one time that the DI channel is low for latch type = 0.
Take latch value of each channel as a bit of a binary value, then the binary value is the DI latch value.

iDO_Latch_Value

[Output] The pointer to the latch value read back from the DO channels of DIO module.

The latch value of a particular channel is

1 if there's at least one time that the DO channel is high for latch type = 1;

0 if there's at least one time that the DO channel is low for latch type = 0.

Take latch value of each channel as a bit of a binary value, then the binary value is the DO latch value.

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iDI_TotalCh=8;
int iDO_TotalCh=8;
int iLatchType=0;
DWORD IDI_Latch_Value;
DWORD IDO_Latch_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadDIOLatch(hPort, iAddress, iDI_TotalCh, iDO_TotalCh, iLatchType,
&IDI_Latch_Value,&IDO_Latch_Value);
uart_Close(hPort);
```

- **Remark**

None

10.9 dcon_ClearDIOLatch

This function clears the latch values of DI and DO channels of the DIO module.

- **Syntax**

```
BOOL dcon_ClearDIOLatch(
                HANDLE hPort,
                int iAddress,
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ClearDIOLatch(hPort, iAddress);
uart_Close(hPort);
```

- **Remark**

None

10.10 dcon_ReadDICNT

This function reads the counts of the DI channels of the DI module.

- **Syntax**

```
BOOL dcon_ReadDICNT(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iDI_TotalCh,
                DWORD *iCounter_Value
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iChannel

[Input] The channel to which the counter value belongs

iDI_TotalCh

[Input] Total number of the DI channels of the DI module

iCounter_Value

[Output] The pointer to the counter value

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
```

```
int iChannel =2;
int iDI_TotalCh=8;
DWORD ICounter_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadDICNT(hPort, iAddress,iChannel,iDI_TotalCh, &ICounter_Value);
uart_Close(hPort);
```

- **Remark**

  None

10.11 dcon_ClearDICNT

This function clears the counter value of the DI channel of the DI module.

- **Syntax**

```
BOOL dcon_ClearDICNT(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iDI_TotalCh
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel to which the counter value belongs

iDI_TotalCh
[Input] Total number of the DI channels of the DI module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=2;
int iDI_TotalCh=8;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ClearDICNT(hPort, iAddress,iChannel,iDI_TotalCh);
uart_Close(hPort);
```

- **Remark**

None

10.12 dcon_WriteAO

This function writes the AO value to the AO modules.

- **Syntax**

```
BOOL dcon_WriteAO(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iAO_TotalCh,
                float fValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel to which the AO value is written

iAO_TotalCh
[Input] The total number of the AO channels of the AO module

fValue
[Input] The AO value to write to the AO module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=2;
int iAO_TotalCh=8;
float fValue=5;
```

```
hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_WriteAO(hPort, iAddress, iChannel, iAO_TotalCh, fValue);
uart_Close(hPort);
```

- **Remark**

  None

10.13 dcon_ReadAO

This function reads the AO value of the AO module.

- **Syntax**

```
BOOL dcon_ReadAO(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iAO_TotalCh,
                float *fValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel from which the AO value is read back

iAO_TotalCh
[Input] The total number of the AO channels of the AO module

fValue
[Output] The pointer to the AO value that is read back from the AO module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=2;
int iAO_TotalCh=8;
float fValue;
```



```
hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadAO(hPort, iAddress,iChannel,iAO_TotalCh, &fValue);
uart_Close(hPort);
```

- **Remark**

  None

10.14 dcon_ReadAI

This function reads the AI value of engineering-mode (floating-point) from the AI module.

- **Syntax**

```
BOOL dcon_ReadAI(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iAI_TotalCh,
                float *fValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel from which the AI value is read back

iAI_TotalCh
[Input] The total number of the AI channels of the AI module

fValue
[Output] The pointer to the AI value that is read back from the AI module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=2;
int iAI_TotalCh=8;
float fValue;
```



```
hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadAI(hPort, iAddress, iChannel, iAI_TotalCh, &fValue);
uart_Close(hPort);
```

- **Remark**

  None

10.15 dcon_ReadAIHex

This function reads the AI value of 2's complement-mode (hexadecimal) from the AI module.

- **Syntax**

```
BOOL dcon_ReadAIHex(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int iAI_TotalCh,
                int *iValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel from which the AI value is read back

iAI_TotalCh
[Input] The total number of the AI channels of the AI module

iValue
[Output] The pointer to the AI value that is read back from the AI module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=2;
int iAI_TotalCh=8;
int iValue;
```



```
hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadAIHex(hPort, iAddress, iChannel, iAI_TotalCh, &iValue);
uart_Close(hPort);
```

- **Remark**

  None

10.16 dcon_ReadAIAll

This function reads all the AI values of all channels in engineering-mode (floating-point) from the AI module.

- **Syntax**

```
BOOL dcon_ReadAIAll(
                HANDLE hPort,
                int iAddress,
                float *fValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

fValue
[Output] The array which contains the AI values that read back from the AI module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
float fValue[8];

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadAIAll(hPort, iAddress, fValue);
uart_Close(hPort);
```

- **Remark**

None

10.17 dcon_ReadAIAllHex

This function reads all the AI values of all channels in 2's complement-mode (hexadecimal) from the AI module.

- **Syntax**

```
BOOL dcon_ReadAIAllHex(
                HANDLE hPort,
                int iAddress,
                int *iValue
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iValue
[Output] The array which contains the AI values that read back from the AI module

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iValue[8];

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadAIAllHex(hPort, iAddress, iValue);
uart_Close(hPort);
```

- **Remark**

None

10.18 dcon_ReadCNT

This function reads the counter values of the counter/frequency modules.

- **Syntax**

```
BOOL dcon_ReadCNT(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                DWORD *iCounter_Value
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel from which the count value is read back from the counter/frequency module

iCounter_Value
[Output] The pointer to the counter value that reads back from the counter/frequency module

- **Return Values**

TRUE indicates success. **FALSE** indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=0;
DWORD lCounter_Value;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadCNT(hPort, iAddress, iChannel, &lCounter_Value);
uart_Close(hPort);
```

- **Remark**

None

10.19 dcon_ClearCNT

This function clears the counter values of the counter/frequency modules.

- **Syntax**

```
BOOL dcon_ClearCNT(
                HANDLE hPort,
                int iAddress,
                int iChannel
);
```

- **Parameter**

hPort

[Input] The serial port HANDLE opened by uart_Open()

iAddress

[Input] The address of the command-receiving I/O module

iChannel

[Input] The channel where the count value is cleared

- **Return Values**

TRUE indicates success.

FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=0;

hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ClearCNT(hPort, iAddress, iChannel);
uart_Close(hPort);
```

- **Remark**

None

10.20 dcon_ReadCNTOverflow

This function reads the overflow value of the channel from the counter/frequency modules.

- **Syntax**

```
BOOL dcon_ReadCNTOverflow(
                HANDLE hPort,
                int iAddress,
                int iChannel,
                int *iOverflow
);
```

- **Parameter**

hPort
[Input] The serial port HANDLE opened by uart_Open()

iAddress
[Input] The address of the command-receiving I/O module

iChannel
[Input] The channel from which the overflow value is read back from the counter/frequency module

iOverflow
[Output] The pointer to the overflow value

iOverflow	Descriptions
1	Overflow
0	not

- **Return Values**

TRUE indicates success.
FALSE indicates failure.

- **Examples**

[C]

```
HANDLE hPort;
BYTE iAddress=1;
int iChannel=0;
int iOverflow;
```



```
hPort = uart_Open("COM1,115200");
BOOL iRet = dcon_ReadCNTOverflow(hPort, iAddress, iChannel, &iOverflow);
uart_Close(hPort);
```

- **Remark**

  None

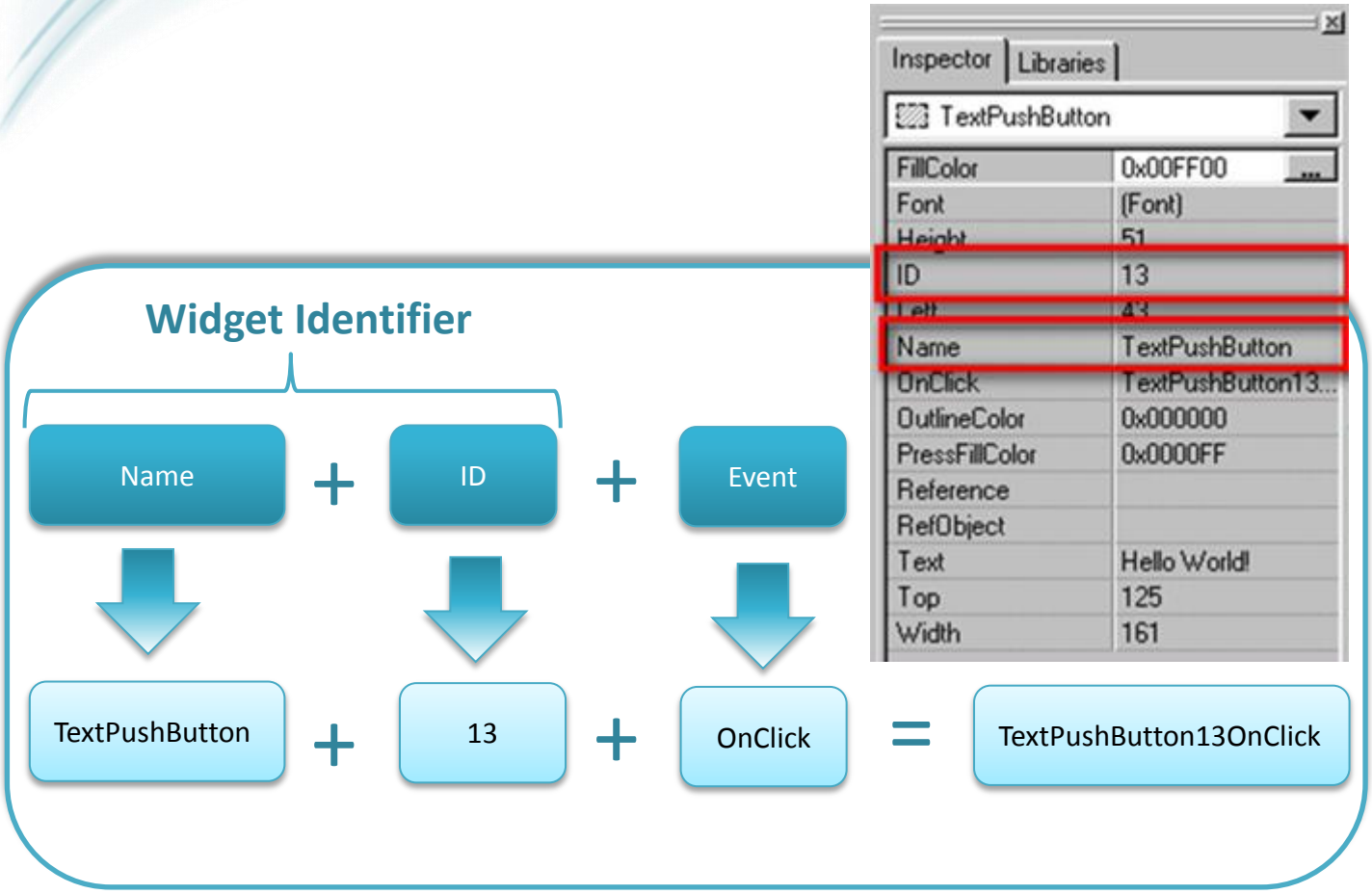
11. Widget API

This chapter provides APIs that are not specified in the Stellaris Graphics Library. (The API functions that we made some modifications)

“The Stellaris Graphics Library is a royalty-free set of graphics primitives and a widget set for creating graphical user interfaces …”

For more details:
http://www.luminarymicro.com/products/stellaris_graphics_library.html

⚠ *Note that the naming convention of the event handler of the widget (here the widget is TextPushButton) is as followings:*



11.1 TextButtonTextGet

Get the **"Text"** property of the TextPushButton.
Users can set the **"Text"** property in the inspector in the design time.

This function is used to get a static string from the **"Text"** property. The widget has no buffer for the **"Text"**, so the string must be static string (static char[]).

- **Syntax**

```
const char *TextButtonTextGet(
                tTextButton *pWidget
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the TextPushButton, to get the **"Text"** property

- **Return Values**

A constant pointer to the static string to store the **"Text"** of the TextPushButton

- **Examples**

[C]

```
int tag = 0;
int count = 0;
static char str[16];
static char str2[16];

void TextPushButton4OnClick(tWidget *pWidget)
{
    count ++;

    //Set the value of count to the "Tag" property
    TextButtonTagSet((tTextButton*)pWidget, count);

    //Get the value of the "Tag" property
    tag = TextButtonTagGet((tTextButton*)pWidget);
```

```
    //Set the "Text" property of the TextPushButton 4
      usprintf(str, "%d", tag);
      TextButtonTextSet((tTextButton*)pWidget, str);

      //Get the "Text" property of the TextPushButton
      //And then show it on the Label 5 (in this example)
      strcpy(str2, TextButtonTextGet((tTextButton*)pWidget));
      LabelTextSet(&Label5, str2);
  }
```

- **Remark**

  None

11.2 TextButtonTextSet

Set the **“Text”** property of the TextPushButton.
Users can set the **“Text”** property in the inspector in the design time.

This function is used to set a static string to the **“Text”** property. The widget has no buffer for the **“Text”**, so the string must be static string (static char[]).

- **Syntax**

```
void TextButtonTextSet(
        tTextButton *pWidget,
        char *text
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the CheckBox, to set the **“Selected”** property

text
[Input] Specify the static string of the **“Text”** property

- **Return Values**

None

- **Examples**

[C]

```
int tag = 0;
int count = 0;
static char str[16];
static char str2[16];

void TextPushButton4OnClick(tWidget *pWidget)
{
    count ++;

    //Set the value of count to the "Tag" property
    TextButtonTagSet((tTextButton*)pWidget, count);
```

```
    //Get the value of the "Tag" property
    tag = TextButtonTagGet((tTextButton*)pWidget);

    //Set the "Text" property of the TextPushButton 4
    usprintf(str, "%d", tag);
    TextButtonTextSet((tTextButton*)pWidget, str);

    //Get the "Text" property of the TextPushButton
    //And then show it on the Label 5 (in this example)
    strcpy(str2, TextButtonTextGet((tTextButton*)pWidget));
    LabelTextSet(&Label5, str2);
}
```

- **Remark**

  None

11.3 SliderRangeGet

This macro gets the range of the Slider.
That is, get the Min and the Max properties of the Slider.

- **Syntax**

```
void SliderRangeGet(
        tWidget *pWidget,
        long lMinimum,
        long lMaximum
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the Slider, to get its Max and Min property

lMinimum
[Output] Specify the integer to store the minimum of the Slider value, that is, the Min property

lMaximum
[Output] Specify the integer to store the maximum of the Slider value, that is, the Max property

- **Return Values**

None

- **Examples**

[C]

```
void BitButton5OnClick(tWidget *pWidget)

{
    static char msg[32];
    long min;
    long max;

    SliderRangeGet(&Slider4, min, max);

    usprintf(msg, "%d, %d", min, max);
    LabelTextSet(&Label6, msg);
}
```

- **Remark**

The parameters lMinimum and lMaximum are not pointers (actually SliderRangeGet is a macro), while the pWidget is a pointer.

11.4 HotSpotLastXGet

Get the last clicked point's coordinate X.
(The left-top vertex of the screen is the origin.)

- **Syntax**

```
int HotSpotLastXGet(
    tHotSpot *pWidget
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the HotSpot, to get the last clicked point's coordinate X

- **Return Values**

The last clicked point's coordinate X

- **Examples**

[C]

```
int tag = 0;
int count = 0;
static char str[16];

void HotSpot4OnClick(tWidget *pWidget)
{
    count ++;

    //Get the last clicked coordinate X, Y
    int x = HotSpotLastXGet((tHotSpot*)pWidget);
    int y = HotSpotLastYGet((tHotSpot*)pWidget);

    //Set the value of count to the "Tag" property
    HotSpotTagSet((tHotSpot*)pWidget, count);

    //Get the value of the "Tag" property
    tag = HotSpotTagGet((tHotSpot*)pWidget);

    //Show the "Tag" and (x, y) on the Label 6 (in this example)
    usprintf(str, "tag=%d, x=%d, y=%d", tag, x, y);
    LabelTextSet(&Label6, str);
}
```

- **Remark**

None

11.5 HotSpotLastYGet

Get the last clicked point's coordinate Y.
(The left-top vertex of the screen is the origin.)

- **Syntax**

```
int HotSpotLastYGet(
    tHotSpot *pWidget
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the HotSpot, to get the last clicked point's coordinate Y

- **Return Values**

The last clicked point's coordinate Y

- **Examples**

[C]

```
int tag = 0;
int count = 0;
static char str[16];

void HotSpot4OnClick(tWidget *pWidget)
{
    count ++;

    //Get the last clicked coordinate X, Y
    int x = HotSpotLastXGet((tHotSpot*)pWidget);
    int y = HotSpotLastYGet((tHotSpot*)pWidget);

    //Set the value of count to the "Tag" property
    HotSpotTagSet((tHotSpot*)pWidget, count);

    //Get the value of the "Tag" property
    tag = HotSpotTagGet((tHotSpot*)pWidget);

    //Show the "Tag" and (x, y) on the Label 6 (in this example)
    usprintf(str, "tag=%d, x=%d, y=%d", tag, x, y);
    LabelTextSet(&Label6, str);
}
```

- **Remark**

None

11.6 CheckBoxSelectedGet

Get the **"Selected"** property of the CheckBox.

- **Syntax**

```
int CheckBoxSelectedGet(
        tCheckBox *pWidget
);
```

- **Parameter**

pWidget

[Input] Specify the pointer to the widget, the CheckBox, to get the **"Selected"** property.

- **Return Values**

Value	Descriptions
0	unchecked state (un-Selected)
1	checked state (Selected)

- **Examples**

[C]

```
int count = 0;
static char str2[16];

void BitButton5OnClick(tWidget *pWidget)
{
    count ++;

    if(count % 2)
    {
        //Set the "Selected" state to un-checked
        CheckBoxSelectedSet(&CheckBox4, 0);
        WidgetPaint((tWidget*)&CheckBox4);
    }
    else
    {
        //Set the "Selected" state to checked
        CheckBoxSelectedSet(&CheckBox4, 1);
        WidgetPaint((tWidget*)&CheckBox4);
    }

    //Show the "Selected" state on the Label 6 (in this example)
    usprintf(str2, "Sel: %d", CheckBoxSelectedGet(&CheckBox4));
    LabelTextSet(&Label6, str2);
}
```

- **Remark**

None

11.7 CheckBoxSelectedSet

Set the **"Selected"** property of the CheckBox.

- **Syntax**

```
void CheckBoxSelectedSet(
        tCheckBox *pWidget,
        int bFlag
);
```

- **Parameter**

pWidget

[Input] Specify the pointer to the widget, the CheckBox, to set the **"Selected"** property.

bFlag

[Input] Specify the state of the **"Selected"** property.

bFlag	Descriptions
0	unchecked state
1	checked state

- **Return Values**

None

- **Examples**

[C]

```
int count = 0;
static char str2[16];

void BitButton5OnClick(tWidget *pWidget)
{
    count ++;

    if(count % 2)

    {
        //Set the "Selected" state to un-checked
        CheckBoxSelectedSet(&CheckBox4, 0);
        WidgetPaint((tWidget*)&CheckBox4);
    }
    else
```

```
        {
            //Set the "Selected" state to checked
            CheckBoxSelectedSet(&CheckBox4, 1);
            WidgetPaint((tWidget*)&CheckBox4);
        }

        //Show the "Selected" state on the Label 6 (in this example)
        usprintf(str2, "Sel: %d", CheckBoxSelectedGet(&CheckBox4));
        LabelTextSet(&Label6, str2);
    }
```

- **Remark**

  None

11.8 LabelTextGet

Get the **"Text"** property of the Label.
Users can set the **"Text"** property in the inspector in the design time.

This function is used to get a static string from the **"Text"** property.
Because the widget has no buffer for the **"Text"**, the string must be static strings (static char[]). Also different Labels must use different static char arrays or these labels (with the same static char array pointers) may display the same content.

- **Syntax**

```
Const char *LabelTextGet(
                tLable *pWidget,
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the Label, to get the **"Text"** property

- **Return Values**

A constant pointer to the static string to store the **"Text"** of the Label

- **Examples**

[C]

```
int count = 0;
static char str[16];

void BitButton6OnClick(tWidget *pWidget)
{
    const char * p;

    count ++;
    //Set the value of count to the Text of the Label
    usprintf(str, "%d", count);
    LabelTextSet(&Label4, str);

    //Get the value of count to compare with 5
    //if count reaches 5, reset it to zero.
    p = LabelTextGet(&Label4);
    if(strcmp("5", p) == 0)
    {
        count = 0;
        usprintf(str, "5, reset to 0 !");
        LabelTextSet(&Label4, str);
    }
}
```

- **Remark**

None

11.9 LabelTextSet

Set the **"Text"** property of the Label.
Users can set the **"Text"** property in the inspector in the design time.

This function is used to set a static string to the **"Text"** property.
Because the widget has no buffer for the **"Text"**, the string must be static strings (static char[]). Also different Labels must use different static char arrays or these labels (with the same static char array pointers) may display the same content.

- **Syntax**

```
void *LabelTextSet(
        tLable *pWidget,
        char *text
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget, the Label, to get the **"Text"** property

text
[Input] Specify the static string of the **"Text"** property

- **Return Values**

None

- **Examples**

[C]

```
int count = 0;
static char str[16];

void BitButton6OnClick(tWidget *pWidget)
{
    const char * p;

    count ++;
    //Set the value of count to the Text of the Label
    usprintf(str, "%d", count);
    LabelTextSet(&Label4, str);

    //Get the value of count to compare with 5
    //if count reaches 5, reset it to zero.
    p = LabelTextGet(&Label4);
```

```
        if(strcmp("5", p) == 0)
        {
            count = 0;
            usprintf(str, "5, reset to 0 !");
            LabelTextSet(&Label4, str);
        }
    }
```

- **Remark**

  None

11.10 TimerEnabledGet

Get the **"Enabled"** property of the Timer.

- **Syntax**

```
int TimerEnabledGet(
    tTimer *pTimer
);
```

- **Parameter**

pTimer

[Input] Specify the pointer to the Timer to set the **"Enabled"** property

- **Return Values**

None

- **Examples**

[C]

```
static char str[16];

void Timer4OnExecute(tWidget *pWidget)
{
    hmi_Beep();
}

void BitButton5OnClick(tWidget *pWidget)
{
    //Get the status of the timer
    int bEnabled = TimerEnabledGet(&Timer4);

    if(bEnabled)
    {
        TimerEnabledSet(&Timer4, 0); //disable the timer
        usprintf(str, "Timer disabled");
        LabelTextSet(&Label6, str);
    }
    else
    {
        TimerEnabledSet(&Timer4, 1); //enable the timer
        usprintf(str, "Timer enabled");
        LabelTextSet(&Label6, str);
    }
}
```

- **Remark**

None

11.11 TimerEnabledSet

Set the **“Enabled”** property of the Timer.

- **Syntax**

```
void TimerEnabledSet(
    tTimer *pTimer,
    int bFlag
);
```

- **Parameter**

pTimer

[Input] Specify the pointer to the Timer to set the **“Enabled”** property

bFlag

[Input] Specify the state of the **“Enabled”** property.

bFlag	Descriptions
0	“Disabled” state
1	“Enabled” state

- **Return Values**

None

- **Examples**

[C]

```
static char str[16];

void Timer4OnExecute(tWidget *pWidget)
{
    hmi_Beep();
}

void BitButton5OnClick(tWidget *pWidget)
{
    //Get the status of the timer
    int bEnabled = TimerEnabledGet(&Timer4);

    if(bEnabled)
    {
        TimerEnabledSet(&Timer4, 0);    //disable the timer
        usprintf(str, "Timer disabled");
        LabelTextSet(&Label6, str);
    }
```

```
        else
        {
            TimerEnabledSet(&Timer4, 1);    //enable the timer
            usprintf(str, "Timer enabled");
            LabelTextSet(&Label6, str);
        }
    }
```

- **Remark**

  None

11.12 TimerIntervalGet

Get the **"Interval"** property of the Timer. (The interval of a timer is its period.)

- **Syntax**

```
int TimerIntervalGet(
    tTimer *pTimer
);
```

- **Parameter**

pTimer

[Input] Specify the pointer to the Timer to set the **"Interval"** property

- **Return Values**

The Interval (period) of the Timer widget

- **Examples**

[C]

```
// Circularly assign 1000 ~ 5000 (ms) to the interval of a Timer
// The Timer beeps from every 1 second to 5 second
#define ONE_SECOND 1000

int count = 0;
void HotSpot68OnClick(tWidget *pWidget)
{
    static char str[16];
    unsigned long intervalGet;

    count++;

    unsigned long interval = (count % 5 + 1) * ONE_SECOND;
    TimerIntervalSet(&Timer69, interval);

    intervalGet = TimerIntervalGet(&Timer69);
    usprintf(str, "beep every %d sec", intervalGet);
    LabelTextSet(&Label64, str);
}

void Timer69OnExecute(tWidget *pWidget)
{
  hmi_Beep();
}
```

- **Remark**

None

11.13 TimerIntervalSet

Set the **“Interval”** property of the Timer. (The interval of a timer is its period.)

- **Syntax**

```
void TimerIntervalSet(
    tTimer *pTimer,
    unsigned long ulInterval
);
```

- **Parameter**

pTimer
[Input] Specify the pointer to the Timer to set the **“Interval”** property

ulInterval
[Input] Specify the **“Interval”** property of the Timer widget.

- **Return Values**

None

- **Examples**

[C]

```
// Circularly assign 1000 ~ 5000 (ms) to the interval of a Timer
// The Timer beeps from every 1 second to 5 second
#define ONE_SECOND      1000

int count = 0;
void HotSpot68OnClick(tWidget *pWidget)
{
    static char str[16];
    unsigned long intervalGet;
    count++;
    unsigned long interval = (count % 5 + 1) * ONE_SECOND;
    TimerIntervalSet(&Timer69, interval);

    intervalGet = TimerIntervalGet(&Timer69);
    usprintf(str, "beep every %d sec", intervalGet);
    LabelTextSet(&Label64, str);
}

void Timer69OnExecute(tWidget *pWidget)
{
    hmi_Beep();
}
```

- **Remark**

None

11.14 Functions for Tag

This section introduces functions (actually macros) of the Tag property for widgets as shown below. For each widget, there are two functions for the Tag property, xTagSet and xTagGet. (x is the widget name)

We can set the **"Tag"** property in the design time, and then read it in the run-time. The **"Tag"** property would be useful when using several widgets in a single event handler function. This property can be used to indicate which widget is clicked at this time. It looks like a widget array index.

Widget	Function
TextPushButton	int **TextButtonTagGet** (tTextButton * pWidget);
	void **TextButtonTagSet** (tTextButton * pWidget, int tag);
Slider	void **SliderTagSet** (tSlider * pWidget, int tag);
	int **SliderTagGet** (tSlider * pWidget);
BitButton	void **BitButtonTagSet** (tBitButton * pWidget, int tag);
	int **BitButtonTagGet** (tBitButton * pWidget);
HotSpot	void **HotSoptTagSet** (tHotSpot * pWidget, int tag);
	int **HotSoptTagGet** (tHotSpot * pWidget);
CheckBox	void **CheckBoxTagSet** (tCheckBox * pWidget, int tag);
	int **CheckBoxTagGet** (tCheckBox * pWidget);

We take TextPushButton for example to introduce how to use these functions.

- **Syntax**

```
void TextButtonTagSet(
        tTextButton *pWidget,
        int tag
);

void TextButtonTagGet(
      tTextButton *pWidget
);
```

- **Parameter**

  pWidget

  [Input] Specify the pointer to the widget (tTextButton is used for TextPushButton without an associated ObjectList) to set/get the Tag property.

  tag

  [Input] Specify the value of the **"Tag"** property

- **Return Values**

  The returning value of TextButtonTagGet is the value of the **"Tag"** property

- **Examples**

  [C]

```
int tag = 0;
int count = 0;
static char str[16];
static char str2[16];

void TextPushButton4OnClick(tWidget *pWidget)
{
    count ++;

    //Set the value of count to the "Tag" property
    TextButtonTagSet((tTextButton*)pWidget, count);

    //Get the value of the "Tag" property
    tag = TextButtonTagGet((tTextButton*)pWidget);

    //Set the "Text" property of the TextPushButton 4
    usprintf(str, "%d", tag);
    TextButtonTextSet((tTextButton*)pWidget, str);

    //Get the "Text" property of the TextPushButton
    //And then show it on the Label 5 (in this example)
    strcpy(str2, TextButtonTextGet((tTextButton*)pWidget));
    LabelTextSet(&Label5, str2);
}
```

- **Remark**

  ObjButtonTagSet and ObjButtonTagGet originally are usd for TextPushButton having its **"RefObject"** property assigned with an ObjectList and they are obsolete after HMIWorks 2.06.00 (included). Use TextButtonTagGet and TextButtonTagSet instead.

11.15 Functions for Value

This section introduces functions (actually macros) of the value of widgets as shown below. For each widget, there are two functions for their values, xValueSet and xValueGet. (x is the widget name.) RadioButton is a little different because of its characteristics of **"one-of-many"** selection.

The value of a widget is used generally with the RefObject and TagName properties.

1. **RefObject:** the value determines which image in the assigned ObjectList to display.
2. **TagName:** the value generally is the value of assigned Tag, it represents usually the state of a particular remote I/O.

Widget	Function
Slider	void **SliderValueSet** (tSlider * pWidget, int value);
	int **SliderValueGet** (tSlider * pWidget);
	Value
	The Position property, it is where the slider locate (between its Max and Min properties)
CheckBox	Function
	void **CheckBoxValueSet** (tCheckBox * pWidget, int value);
	int **CheckBoxValueGet** (tCheckBox * pWidget);
	Value
	Without any assigned ObjectList in the RefObject property, the value of a CheckBox is 0 or 1. With an assigned ObjectList in the RefObject property, the value of the CheckBox is ranged from 0 to (the image count -1), where the image count is the number of images in the assigned ObjectList.
RadioButton	Function
	void **RadioButtonGroupValueSet** (tRadioButton * pWidget, int value);
	int **RadioButtonGroupValueGet** (tRadioButton * pWidget);
	Value
	Since RadioButtons provide a **"one-of-many"** selection, it is meaningless to say that a single RadioButton has a value. On the contrary, we take all the RadioButtons in the same RadioGroup to be considered together and the value of a RadioButton actually become the SerialNumber of the selected RadioButton in the RadioGroup to which the RadioButton belongs.

We take CheckBox for example to introduce how to use these functions.

- **Syntax**

```
void CheckBoxValueSet(
        tCheckBox *pWidget,
        int value
);

void CheckBoxValueGet(
        tCheckBox *pWidget
);
```

- **Parameter**

pWidget
[Input] Specify the pointer to the widget to set/get its value.

value
[Input] Specify the value of the widget

- **Return Values**

The returning value of CheckBoxValueGet is the value of the widget (here, CheckBox)

- **Examples**

[C]

```
int value = 0;
int count = 0;
static char str[16];

// the image count of the ObjectList assigned in the RefObject property
int imageCount = 5;

void TextPushButton4OnClick(tWidget *pWidget)
{
    count++;

    //Set the value of count to the "Tag" property
    CheckBoxValueSet ((tCheckBox *)pWidget, (count % imageCount));

    //Get the value of the value
    value = CheckBoxValueGet ((tCheckBox *)pWidget);

    //And then show it on the Label 5 (in this example)
    usprintf(str, "%d", value);
    LabelTextSet(&Label5, str);
}
```

- **Remark**

None

11.16 Functions for Enabled

This section introduces functions (actually macros) of the Enabled property for widgets as shown below. For each widget, there are two functions for the Enabled property, xEnabledSet and xEnabledGet. (x is the widget name)

Widget	Function
TextPushButton	void **TextButtonEnabledSet**(tTextButton *pWidget, BOOL bEnabled);
	BOOL **TextButtonEnabledGet**(tTextButton *pWidget);
Slider	void **SliderEnabledSet**(tSlider *pWidget, BOOL bEnabled);
	BOOL **SliderEnabledGet**(tSlider *pWidget);
BitButton	void **BitButtonEnabledSet**(tBitButton *pWidget, BOOL bEnabled);
	BOOL **BitButtonEnabledGet**(tBitButton *pWidget);
HotSpot	void **HotSpotEnabledSet**(tHotSpot *pWidget, BOOL bEnabled);
	BOOL **HotSpotEnabledGet**(tHotSpot *pWidget);
CheckBox	void **CheckBoxEnabledSet**(tCheckBox *pWidget, BOOL bEnabled);
	BOOL **CheckBoxEnabledGet**(tCheckBox *pWidget);
Label	void **LabelEnabledSet**(tLabel *pWidget, BOOL bEnabled);
	BOOL **LabelEnabledGet**(tLabel *pWidget);
RadioButton	void **RadioButtonEnabledSet**(tLabel *pWidget, BOOL bEnabled);
	BOOL **RadioButtonEnabledGet**(tLabel *pWidget);

We take TextPushButton for example to introduce how to use these functions.

- **Syntax**

```
void TextButtonEnabledSet(
        tTextButton *pWidget,
        BOOL bEnabled
);

BOOL TextButtonEnabledGet(
        tTextButton *pWidget
);
```

- **Parameter**

  pWidget

  [Input] Specify the pointer to the widget (tTextButton is used for TextPushButton without an associated ObjectList) to set/get the Enabled property.

  bEnabled

  [Input] Specify the **"Enabled"** property of the widget.
  Possbile value: TRUE or FALSE

- **Return Values**

  The returning value of TextButtonEnabledGet is the status of the Enabled property.
  TRUE: the widget is enabled
  FALSE: the widget is not enabled

- **Examples**

  [C]

- **Remark**

  ObjButtonEnabledSet and ObjButtonEnabledGet originally are usd for TextPushButton having its **"RefObject"** property assigned with an ObjectList and they are obsolete after HMIWorks 2.06.00 (included). Use TextButtonEnabledGet and TextButtonEnabledSet instead.

11.17 Functions for Visible

This section introduces functions (actually macros) of the Visible property for widgets as shown below. For each widget, there are two functions for the Visible property, xVisibleSet and xVisibleGet. (x is the widget name)

Widget	Function
TextPushButton	void **TextButtonVisibleSet**(tTextButton *pWidget, BOOL bVisible);
	BOOL **TextButtonVisibleGet**(tTextButton *pWidget);
Slider	void **SliderVisibleSet**(tSlider *pWidget, BOOL bVisible);
	BOOL **SliderVisibleGet**(tSlider *pWidget);
BitButton	void **BitButtonVisibleSet**(tBitButton *pWidget, BOOL bVisible);
	BOOL **BitButtonVisibleGet**(tBitButton *pWidget);
CheckBox	void **CheckBoxVisibleSet**(tCheckBox *pWidget, BOOL bVisible);
	BOOL **CheckBoxVisibleGet**(tCheckBox *pWidget);
Label	void **LabelVisibleSet**(tLabel *pWidget, BOOL bVisible);
	BOOL **LabelVisibleGet**(tLabel *pWidget);
RadioButton	void **RadioButtonVisibleSet**(tLabel *pWidget, BOOL bVisible);
	BOOL **RadioButtonVisibleGet**(tLabel *pWidget);

We take TextPushButton for example to introduce how to use these functions.

- **Syntax**

```
void TextButtonEnabledSet(
        tTextButton *pWidget,
        BOOL bVisible
);

BOOL TextButtonEnabledGet(
        tTextButton *pWidget
);
```

- **Parameter**

  pWidget

  [Input] Specify the pointer to the widget (tTextButton is used for TextPushButton without an associated ObjectList) to set/get the Visible property.

  bVisible

  [Input] Specify the **"Visible"** property of the widget.

  Possbile value: TRUE or FALSE

- **Return Values**

  The returning value of TextButtonVisibleGet is the status of the Visible property.

  TRUE: the widget is visible

  FALSE: the widget is invisible

- **Examples**

  [C]

```
int status = 0;

void TextPushButton5OnClick(tWidget *pWidget)
{
   hmi_Beep();
}

void BitButton4OnClick(tWidget *pWidget)
{
      BOOL en = FALSE;
      BOOL vi = FALSE;
      tTextButton * pt = &TextPushButton5;
      static char msg[32];

      status++;

      switch (status % 4)

      {
            default:
            case 0:
                  TextButtonEnabledSet(pt, TRUE);
                  TextButtonVisibleSet(pt, TRUE);
                  break;
            case 1:
                  TextButtonEnabledSet(pt, TRUE);
                  TextButtonVisibleSet(pt, FALSE);
                  break;
            case 2:
                  TextButtonEnabledSet(pt, FALSE);
                  TextButtonVisibleSet(pt, TRUE);
                  break;
            case 3:
                  TextButtonEnabledSet(pt, FALSE);
                  TextButtonVisibleSet(pt, FALSE);
                  break;
```



```
        }

        // WidgetPaint must be executed to redraw full screen after
        // changing the visibility of any widget.
        WidgetPaint(WIDGET_ROOT);

        en = TextButtonEnabledGet(pt);
        vi = TextButtonVisibleGet(pt);
        strcpy(msg, en ? "Enabled, " : "Disenabled, ");
        strcat(msg, vi ? "Visible" : "Unvisible");
        LabelTextSet(&Label8, msg);
    }
```

- **Remark**
  1. WidgetPaint(WIDGET_ROOT) must be called to redraw all widgets after changing visibility of the widget to make the changing take effect. WidgetPaint costs system resources so we don't suggest users frequently changing the visibility of widgets.
  2. ObjButtonVisibleSet and ObjButtonVisibleGet originally are used for TextPushButton having its **"RefObject"** property assigned with an ObjectList and they are obsolete after HMIWorks 2.06.00 (included). Use TextButtonVisibleGet and TextButtonVisibleSet instead.

12. Flash API

This chapter introduces API functions for flash reading and writing.

For users' convenience, there are two sets of API functions for data storage in the flash on the TouchPAD devices. One is for the MCU (micro-controller unit) internal flash and the other is the external serial flash (total 8 MB).

To user these features, install the HMIWorks software with **version 2.03 or above**.
ftp://ftp.icpdas.com/pub/cd/touchpad/setup/

No.	1	2
Target Flash	MCU internal flash	External serial flash
Possible Target Device	All devices in the TouchPAD series	All devices in the TouchPAD series, except TPD-280 and TPD-283 (for those having external flash)
API Functions Provided	hmi_UserParamsGet, hmi_UserParamsSet	hmi_UserFlashReadEx, hmi_UserFlashWriteEx, hmi_UserFlashConfig, hmi_UserFlashErase
Size of Storage	256 byte	4 KB ~ 7 MB
Suggested Users	Any TouchPAD users	**For advanced users only.** Any undetermined use will damage the application image.

To user these features, install the HMIWorks software with **version 2.10.30 or above**.
ftp://ftp.icpdas.com/pub/cd/touchpad/setup/

No.	3
EEPROM	The EEPROM is available on 32-bit operation, so there are 512 DWORD.
Possible Target Device/ Suggested Users	Provides 2 KB EEPROM user space for H versions and 7'' models.
API Functions Provided	hmi_UserEepromRead, hmi_UserEepromWrite, hmi_UserEepromErase
Size of Storage	512 DWORD

12.1 hmi_UserParamsGet

Get data from the 256-byte parameter area in the MCU (MicroController Unit) internal flash.

- **Syntax**

```
int hmi_UserParamsGet(
        int iOffset,
        int iSize,
        char *pcBuffer
);
```

- **Parameter**

iOffset

[Input] Specify the offset to the base of the 256-byte parameter area to read data from it.
Possible range: 0 ~ 255. ⚠ *Note: iOffset + iSize cannot be larger than 256*

iSize

[Input] Specify the size of the data to read from the 256-byte parameter area.
Possible range: 1 ~ 256. ⚠ *Note: iOffset + iSize cannot be larger than 256*

pcBuffer

[Output] Specify the pointer to the char array to store the data got from the 256-byte parameter area.

- **Return Values**

1 (true) = OK;
0 (false) = Failure

- **Examples**

[C]

```
void BitButton4OnClick(tWidget *pWidget)
{
    static int iLog = 0;
    static char szMsg[30];

    // User Parameter Area is 256 bytes only.
    // Get the last record
    if ( hmi_UserParamsGet(0, 4, (char *)&iLog) )
    {
```



```
                iLog++;
                usprintf(szMsg, "%d", iLog);

                 LabelTextSet(&Label5, szMsg);

                // Update data
                if ( hmi_UserParamsSet(0, 4, (char *)&iLog) )
                {
                    hmi_Beep();
                }
            }
        }
```

- **Remark**
  1. There is a write/erase limit for the flashes.
     Frequent uses may damage the flash.
  2. The old g_sParameters. UserParamsData is no longer available.

12.2 hmi_UserParamsSet

Set data to the 256-byte parameter area in the MCU (MicroController Unit) internal flash.

- **Syntax**

```
int hmi_UserParamsSet(
        int iOffset,
        int iSize,
        char *pcBuffer
);
```

- **Parameter**

iOffset

[Input] Specify the offset to the base of the 256-byte parameter area to write data to it.
Possible range: 0 ~ 255. ⚠ *Note: iOffset + iSize cannot be larger than 256*

iSize

[Input] Specify the size of the data to write to the 256-byte parameter area.
Possible range: 1 ~ 256. ⚠ *Note: iOffset + iSize cannot be larger than 256*

pcBuffer

[Input] Specify the pointer to the char array which is used to write to the 256-byte parameter area.

- **Return Values**

1 (true) = OK;
0 (false) = Failure

- **Examples**

[C]

```
void BitButton4OnClick(tWidget *pWidget)
{
    static int iLog = 0;
    static char szMsg[30];

    // User Parameter Area is 256 bytes only.
    // Get the first record
    if ( hmi_UserParamsGet(0, 4, (char *)&iLog) )
    {
```



```
                iLog++;
                usprintf(szMsg, "%d", iLog);
                LabelTextSet(&Label5, szMsg);

                // Update data
                if ( hmi_UserParamsSet(0, 4, (char *)&iLog) )
                {
                    hmi_Beep();
                }
            }
        }
```

- **Remark**
  1. There is a write/erase limit for the flashes.
     Frequent uses may damage the flash.
  2. The old g_sParameters. UserParamsData is no longer available.
  3. Since hardware limitation, each write operation (calling the hmi_UserParamsSet function) cause a full block updated. For better performance, please use single write operation with all parameters in a buffer instead of several write operations

12.3 hmi_UserFlashConfig

Configure how many blocks of the external serial flash can be used for reading and writing. Each block has size of 4 KB.

➢ **Syntax**

```
int hmi_UserFlashConfig(
        unsigned long iNumberOfBlocks
);
```

➢ **Parameter**

iNumberOfBlocks

[Input] Specify the number of blocks to be used for data storage.

Possible range: 1 ~ 1792 blocks (= 4 KB ~ 7 MB, 0 block = disable)

Number of Blocks	Size	Number of Blocks	Size
1	4 KB	768	3 MB
2	8 KB	1024	4 MB
4	16 KB	1280	5 MB
8	32 KB	1536	6 MB
256	1 MB	1792	7 MB
512	2 MB		

⚠ *Note:*

1. *The application image is put on the external flash started from the lowest address (that is, zero). Be careful to configure an area in order not to overwrite the application image.*
2. *The area claimed by hmi_UserFlashConfig occupies the highest addresses of the external flash and started from its lowest address inside the claimed area.*
3. *The application image and the hmi_UserFlashConfig share the same external flash. If the sum of them larger than the specification, the data will be overwritten, the image may break.*
4. *User can replace the hmi_UserFlashConfig API by configuring the flash size at "Project→Project Configuration→Other→User Flash Config→Size". By this method, the compiler will check whether the flash is enough, the compiler will report an error if the flash is not enough.*

➢ **Return Values**

The number of blocks that are claimed

➢ **Examples**

[C]

```
void btnConfig4OnClick(tWidget *pWidget)
{
    // Enable 1792 blocks (=7 MB) for reading/writing by user
    if ( hmi_UserFlashConfig(1792) == 1792 )
        LabelTextSet(&Label7, "Configure OK");
}

void btnWrite5OnClick(tWidget *pWidget)
{
    int i;
    for (i=0; i<1792; i++)    // loop through the upper 7 MB
    {
        // MUST: Erase the external serial flash block
        hmi_UserFlashErase(i);
        // Write 4 bytes to each block for example
        hmi_UserFlashWriteEx(i, 0, 4, (char *)&i);
    }
}

void btnRead6OnClick(tWidget *pWidget)
{
    static char szMsg[30];
    static int iBlock = 0;
    int iVal;

    // Read 4 bytes from a block
    if ( hmi_UserFlashReadEx(iBlock, 0, 4, (char *)&iVal) )
    {
        usprintf(szMsg, "%d", iVal);
        LabelTextSet(&Label7, szMsg);
     }
    else
        LabelTextSet(&Label7, "Read Error");

    iBlock++;
}
```

➢ **Remark**

1. There is a write/erase limit for the flashes.
   Frequent uses may damage the flash.
2. The old functions, such as hmi_UserFlashRead and hmi_UserFlashWrite, are depreciated.
3. **For advanced users only.** Any wrongly-configured area will overwrite the application area and damage the application image.

12.4 hmi_UserFlashReadEx

Read data from the user flash area that configured by hmi_UserFlashConfig.

- **Syntax**

```
unsigned long hmi_UserFlashReadEx(
                        unsigned long iBlock,
                        unsigned long iOffset,
                        unsigned long iLength,
                        char *pBuffer
);
```

- **Parameter**

iBlock

[Input] Specify the block index to read data from the configured user flash area.
Possible range: 0 to iNumberOfBlocks - 1.
⚠ *Note: where iNumberOfBlocks is the number of blocks claimed by the hmi_UserFlashConfig function*

iOffset

[Input] Specify the offset to the base of the block to read data from that block which has index equal to iBlock. Possible range: 0 ~ 4095. ⚠ *Note: iOffset + iLength cannot be larger than 4096*

iLength

[Input] Specify the size of the data to read from the block of the flash whose index is iBlock.
Possible range: 1 ~ 4096. ⚠ *Note: iOffset + iLength cannot be larger than 4096*

pBuffer

[Output] Specify the pointer to the char array to store the data read from the configured user flash area.

- **Return Values**

Data length read;
0 (false) = failure

- **Examples**

[C]

```
void btnConfig4OnClick(tWidget *pWidget)
{
    // Enable 1792 blocks (=7 MB) for reading/writing by user
if ( hmi_UserFlashConfig(1792) == 1792 )
        LabelTextSet(&Label7, "Configure OK");
}

void btnWrite5OnClick(tWidget *pWidget)
{
    int i;
    for (i=0; i<1792; i++)    // loop through the upper 7 MB
    {
        // MUST: Erase the external serial flash block
        hmi_UserFlashErase(i);
        // Write 4 bytes to each block for example
        hmi_UserFlashWriteEx(i, 0, 4, (char *)&i);
    }
}

void btnRead6OnClick(tWidget *pWidget)
{
    static char szMsg[30];
    static int iBlock = 0;
    int iVal;

    // Read 4 bytes from a block
    if ( hmi_UserFlashReadEx(iBlock, 0, 4, (char *)&iVal) )
    {
        usprintf(szMsg, "%d", iVal);
        LabelTextSet(&Label7, szMsg);
    }
    else
        LabelTextSet(&Label7, "Read Error");

    iBlock++;
}
```

- **Remark**
  1. There is a write/erase limit for the flashes.
     Frequent uses may damage the flash.
  2. The old functions, such as hmi_UserFlashRead and hmi_UserFlashWrite, are depreciated.
  3. **For advanced users only.** Any wrongly-configured area will overwrite the application area and damage the application image.

12.5 hmi_UserFlashErase

Erase a block of data in the user flash area that configured by hmi_UserFlashConfig. The data becomes all 0xFF after erased. This function **MUST** be used before hmi_UserFlashWriteEx is used.

- **Syntax**

```
int hmi_UserFlashErase(
        unsigned long iBlock
);
```

- **Parameter**

iBlock

[Input] Specify the block index to erase in the configured user flash area.

Possible range: 0 to iNumberOfBlocks - 1.

⚠ *Note: where iNumberOfBlocks is the number of blocks claimed by the hmi_UserFlashConfig function*

- **Return Values**

TRUE = Success;

0 (FALSE) = Failure

- **Examples**

[C]

```
void btnConfig4OnClick(tWidget *pWidget)
{
    // Enable 1792 blocks (=7 MB) for reading/writing by user
    if ( hmi_UserFlashConfig(1792) == 1792 )
        LabelTextSet(&Label7, "Configure OK");
}

void btnWrite5OnClick(tWidget *pWidget)
{
    int i;
    for (i=0; i<1792; i++)    // loop through the upper 7 MB
    {
        // MUST: Erase the external serial flash block
        hmi_UserFlashErase(i);
        // Write 4 bytes to each block for example
        hmi_UserFlashWriteEx(i, 0, 4, (char *)&i);
    }
}
```

```
void btnRead6OnClick(tWidget *pWidget)
{
    static char szMsg[30];
    static int iBlock = 0;
    int iVal;

    // Read 4 bytes from a block
    if ( hmi_UserFlashReadEx(iBlock, 0, 4, (char *)&iVal) )
    {
        usprintf(szMsg, "%d", iVal);
        LabelTextSet(&Label7, szMsg);
    }
    else
        LabelTextSet(&Label7, "Read Error");

    iBlock++;
}
```

- **Remark**
  1. There is a write/erase limit for the flashes.
     Frequent uses may damage the flash.
  2. The old functions, such as hmi_UserFlashRead and hmi_UserFlashWrite, are depreciated.
  3. **For advanced users only.** Any wrongly-configured area will overwrite the application area and damage the application image.

12.6 hmi_UserFlashWriteEx

Write data to the user flash area that configured by hmi_UserFlashConfig.
The write operation only change the related bits from 1 to 0, but not from 0 to 1. So erase function is required to set all data 0xFF before write operation.

- **Syntax**

```
unsigned long hmi_UserFlashWriteEx(
                        unsigned long iBlock,
                        unsigned long iOffset,
                        unsigned long iLength,
                        char *pBuffer
);
```

- **Parameter**

iBlock

[Input] Specify the block index to write data to the configured user flash area.
Possible range: 0 to iNumberOfBlocks - 1.
⚠ *Note: where iNumberOfBlocks is the number of blocks claimed by the hmi_UserFlashConfig function*

iOffset

[Input] Specify the offset to the base of the block to write data to that block which has index equal to iBlock. Possible range: 0 ~ 4095. ⚠ *Note: iOffset + iLength cannot be larger than 4096*

iLength

[Input] Specify the size of the data to write to the block of the flash whose index is iBlock.
Possible range: 1 ~ 4096. ⚠ *Note: iOffset + iLength cannot be larger than 4096*

pBuffer

[Output] Specify the pointer to the char array which is used to write to the configured user flash area.

- **Return Values**

Data length written

- **Examples**

[C]

```
void btnConfig4OnClick(tWidget *pWidget)
{
    // Enable 1792 blocks (=7 MB) for reading/writing by user
```

```
    if ( hmi_UserFlashConfig(1792) == 1792 )
        LabelTextSet(&Label7, "Configure OK");
}

void btnWrite5OnClick(tWidget *pWidget)
{
    int i;
    for (i=0; i<1792; i++)    // loop through the upper 7 MB
    {
        // MUST: Erase the external serial flash block
        hmi_UserFlashErase(i);
        // Write 4 bytes to each block for example
        hmi_UserFlashWriteEx(i, 0, 4, (char *)&i);
    }
}

void btnRead6OnClick(tWidget *pWidget)
{
    static char szMsg[30];
    static int iBlock = 0;
    int iVal;

    // Read 4 bytes from a block
    if ( hmi_UserFlashReadEx(iBlock, 0, 4, (char *)&iVal) )
    {
        usprintf(szMsg, "%d", iVal);
        LabelTextSet(&Label7, szMsg);
    }
    else
        LabelTextSet(&Label7, "Read Error");

    iBlock++;
}
```

- **Remark**
  1. There is a write/erase limit for the flashes.
     Frequent uses may damage the flash.
  2. The old functions, such as hmi_UserFlashRead and hmi_UserFlashWrite, are depreciated.
  3. **For advanced users only.** Any wrongly-configured area will overwrite the application area and damage the application image.

12.7 hmi_UserEepromRead

Read 32-bit data from the 2-KB EEPROM.

- **Syntax**

```
void hmi_UserEepromRead(
        unsigned long ui32Address,
        unsigned long ui32Count,
        unsigned long *pui32Data
);
```

- **Parameter**

Ui32Address

[Input] Defines the offset address of the EEPROM to be read.
The valid offset address is between 0 to 511.

Ui32Count

[Input] Defines the number of 32-bit data that is to be read.
The valid value is between 1 to 512.

Pui32Data

[Output] Points to the first 32-bit data buffer to store the data read from EEPROM

- **Return Values**

None

12.8 hmi_UserEepromWrite

Write 32-bit data to the 2-KB EEPROM.

Be careful! The EEPROM has about 100,000 times erase/write limitation on each address.

- **Syntax**

```
unsigned long hmi_UserEepromWrite(
                unsigned long ui32Address,
                unsigned long ui32Count,
                 unsigned long *pui32Data
);
```

- **Parameter**

Ui32Address

[Input] Defines the offset address of the EEPROM to be written to.
The valid offset address is between 0 to 511.

Ui32Count

[Input] Defines the number of 32-bit data that is to be written.
The valid value is between 1 to 512.

Pui32Data

[Output] Points to the first 32-bit data to write to the EEPROM.

- **Return Values**

Returns 0 on success or non-zero values on failure.

12.9 hmi_UserEepromErase

Erase the content of the 2-KB EEPROM.

The hmi_UserEepromErase() function is not required before any write operation, but can be used to clear garbage data first.

Be careful! The EEPROM has about 100,000 times erase/write limitation on each address.

- **Syntax**

```
unsigned long hmi_UserEepromErase(
        unsigned long ui32Address,
        unsigned long ui32Count
    );
```

- **Parameter**

Ui32Address

[Input] Defines the offset address of the EEPROM to be erased.
The valid offset address is between 0 to 511.

Ui32Count

[Input] Defines the number of 32-bit data space that is to be erased.
The valid value is between 1 to 512.

- **Return Values**

Returns 0 on success or non-zero values on failure.

13. MQTT API

This chapter introduces API functions for the MQTT protocol.
These API functions support MQTT client only. And secure MQTT (TLS, RFC5246) is not supported too.

For information about MQTT,
http://mqtt.org/
“MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. …”

From the above site you can find,
MQTT v3.1.1, OASIS Standard, 29 October 2014
https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard

The MQTT source for HMIWorks is ported from the Paho project.
http://www.eclipse.org/paho/
http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.embedded-c.git/
The MQTT API functions for HMIWorks are only a thin layer of interface. Their target is hiding the Paho source details from users. If users want to customize themselves, they can try to use the API functions provided by the Paho source without using our MQTT API functions.

TouchPAD models that support MQTT

- ✓ Not all the models in the TouchPAD series support MQTT
- ✓ The models that support MQTT are
  TPD-283-H, TPD-283-Mx,
  TPD-283U-H, TPD-283U-Mx,
  TPD-433-H, TPD-433-Mx,
  TPD-703, TPD-703-64,
  VPD-133-H.

13.1 hmi_MQTTConnect

Connect to a broker (MQTT server).

- **Syntax**

```
int hmi_MQTTConnect(
        char *ip_addr,
        int port,
        char *client_id,
        char *username,
        char *password
);
```

- **Parameter**

ip_addr

[Input] Specify the dotted IP address string, e.g. “10.1.0.236”.

port

[Input] Specify the TCP port that the MQTT protocol use.

Be sure to specify the port to 1883.

We reserve this parameter only for possible future updates.

client_id

[Input] Specify the string for a server to identify the client.

The *client_id* are between 1 and 23 bytes in length, and that contain only the characters "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

username

[Input] This parameter is used when the MQTT broker requires authentication.

password

[Input] This parameter is used when the MQTT broker requires authentication.

- **Return Values**

Reserved for future use

- **Examples**

[C]

```
//
// This file, __Frame1.h, is encoded in UTF-8, so that we can support
// unicode messages.
//

#define HANDLER_TIMEOUT                400

//
// The library of MQTT (mqtt.a) for HMIWorks allocates 256 bytes for
// both topics and messages. Here we must declare a buffer of char not
// shorter than 256.
//

#define BUF_LEN                256

//
// The buffers of topics and messages that we subscribe.
//

char topic_recv[BUF_LEN];
char msg_recv[BUF_LEN];

void BitButton7OnClick(tWidget *pWidget)
{
        //
        // Connect to remote MQTT broker. MQTT port = 1883.
        // client_id is used to be identified by the server. We suggest users to
        // make client_id from [0-9A-Za-z] and make client_id of length 1 to 23
        // characters because the MQTT standard (MQTT Version 3.1.1) tells
        // the broker must support these.
        //
        char* client_id = "ICPDAS";
        hmi_MQTTConnect("10.1.0.25", 1883, client_id, "", "");
}
void BitButton8OnClick(tWidget *pWidget)
{
        hmi_MQTTDisconnect();
}

void BitButton10OnClick(tWidget *pWidget)    // Publish
{
        int qos = 1;
        bool retain = false;
        char* topic = "Hello";
        char* msg = "Bye!";

        hmi_MQTTPublish(qos, retain, topic, msg);
}
```

```
void BitButton16OnClick(tWidget *pWidget)    // Subscribe Hello
{
        int qos = 1;
        hmi_MQTTSubscribe(qos, "Hello");
}

void BitButton18OnClick(tWidget *pWidget)    // Unsubscribe Hello
{
        hmi_MQTTUnsubscribe("Hello");
}

void Timer6OnExecute(tWidget *pWidget)
{
        //
        // HANDLER_TIMEOUT (here, 400) we use for hmi_MQTTHandler
        // must be less than the interval of this timer (here, Timer6). It waits for
        // messages sent by the server until HANDLER_TIMEOUT ms gone.
        //
        int ret = hmi_MQTTHandler(HANDLER_TIMEOUT, topic_recv, msg_recv);

        //
        // If something has been received though the received data are not
        // necessary what we subscribed, we update the topic and message to
        // Labels anyway. If the data is not what we subscribed, topic_recv and
        // msg_recv are simply left the same as before.
        //
        if (ret > 0)
        {
                LabelTextSet(&Label4, topic_recv);
                LabelTextSet(&Label5, msg_recv);
        }
}
```

- **Remark**

  None

13.2 hmi_MQTTDisconnect

Disconnect from a broker (MQTT server).

- **Syntax**

```
int hmi_MQTTDisconnect();
```

- **Parameter**

None

- **Return Values**

Reserved for future use

- **Examples**

[C]

```
// the same as hmi_MQTTConnect
```

- **Remark**

None

13.3 hmi_MQTTPublish

Pusblish the specified topic with messages to a broker (MQTT server).

- **Syntax**

```
int hmi_MQTTPublish(
        int qos,
        bool retain,
        char *topic,
        char *message
);
```

- **Parameter**

qos

[Input] Specify the level of **"Quality of Service"**.

qos	Descriptions
0	at most once delivery
1	at least once delivery
2	exactly once delivery

retain

[Input] Specify the retain flag for this publish action.

"If the RETAIN flag is set to 1, in a PUBLISH Packet sent by a Client to a Server, the Server MUST store the Application Message and its QoS, so that it can be delivered to future subscribers whose subscriptions match its topic name."

topic

[Input] Specify the topic of the published message.

"The Topic Name identifies the information … published."

message

[Input] Specify the message that is published.

"The content and format of the data is application specific."

⚠ *Note: For more information about QoS, retain, topic, message, refer to "MQTT v3.1.1, OASIS Standard" as described above.*

- **Return Values**

  Reserved for future use

- **Examples**

  [C]

```
// the same as hmi_MQTTConnect
```

- **Remark**

  None

13.4 hmi_MQTTSubscribe

Subscribe messages of the specified topic from a broker (MQTT server).

- **Syntax**

```
int hmi_MQTTSubscribe(
        int qos,
        char *topicFilter
);
```

- **Parameter**

qos

[Input] Specify the level of **"Quality of Service"**.

qos	Descriptions
0	at most once delivery
1	at least once delivery
2	exactly once delivery

topicFilter

[Input] Specify the topic that we want to subscribe.

"The Server sends PUBLISH Packets to the Client in order to forward Application Messages that were published to Topics that match these Subscriptions."

There are at most 5 topics for one TouchPAD device can subscribe.

⚠ *Note: For more information about QoS, retain, topic, message, refer to "MQTT v3.1.1, OASIS Standard" as described above.*

- **Return Values**

Reserved for future use

- **Examples**

[C]

```
// the same as hmi_MQTTConnect
```

- **Remark**

None

13.5 hmi_MQTTUnsubscribe

Unsubscribe messages of the specified topic from a broker (MQTT server).

```
int hmi_MQTTUnsubscribe(
        const char *topic
);
```

- **Parameter**

  topic

  [Input] Specify the topic that we want to unsubscribe.

  "The Server sends PUBLISH Packets to the Client in order to forward Application Messages that were published to Topics that match these Subscriptions."

⚠ *Note: For more information about QoS, retain, topic, message, refer to "MQTT v3.1.1, OASIS Standard" as described above.*

- **Return Values**

  Reserved for future use

- **Examples**

  [C]

```
// the same as hmi_MQTTConnect
```

- **Remark**

  None

13.6 hmi_MQTTHandler

The function that should be called periodically to handle the received messages of the specified topic from a broker (MQTT server).

- **Syntax**

```
int hmi_MQTTHandler(
        int timeout_ms,
        char *topic_recv,
        char *msg_recv
);
```

- **Parameter**

timeout_ms

[Input] Specify the timeout value of this handler function. (unit: millisecond)

The handler function goes on receiving any data from a broker until a timeout value of time is reached since it starts receiving.

Usually we use a Timer widget to call periodically hmi_MQTTHandler and be sure to let the interval of the Timer widget greater than the timeout_ms of the hmi_MQTTHandler function.

topic_recv

[Output] Specify the pointer to an array that is used to store the receiving topic from a brokder.

msg_recv

[Output] the pointer to an array that is used to store the receiving messages from a brokder.

- **Return Values**

0: there's no data read.

1: there comes receiving data.

- **Examples**

[C]

```
// the same as hmi_MQTTConnect
```

- **Remark**

None

14. Miscellaneous API

This chapter provides APIs that are not classified.

14.1 hmi_Beep

Sound the beep.

- **Syntax**

```
void hmi_Beep();
```

- **Parameter**

None

- **Return Values**

None

- **Examples**

[C]

```
hmi_Beep ();
```

- **Remark**

None

14.2 hmi_PlaySong

Play a song through a buzzer.

The range the buzzer can play is C4 -- B7.

But TPD-280/TPD-283 has only the range G5 -- B7 because of hardware limitation.

Refer to http://en.wikipedia.org/wiki/Piano_key_frequencies for more information.

⚠ *Note: TPD-430/TPD-430-EU does **NOT** support this function.*

- **Syntax**

```
int hmi_PlaySong(
        unsigned short *g_pusKeyClick,
        unsigned long len
);
```

- **Parameter**

g_pusKeyClick

[Input] Specify the song in an array to have a series of the (length, pitch) pair.

⚠ *Note that the length has unit: 10 ms.*

len

[Input] Specify the length of the array specified in the first parameter.

- **Return Values**

None

- **Examples**

[C]

```
//
// Do Re Mi Fa So La Si
// In the example below, the “Do” is issued 25 x 10 ms long and then the
// “Re” issued. The “Re” is issued for (50 - 25) x 10 ms long, then the “Mi”
// issued. And so on.
//
unsigned short g_pusKeyClick[] =
    {
        0, C7,
       25, D7,
       50, E7,
       75, F7,
```



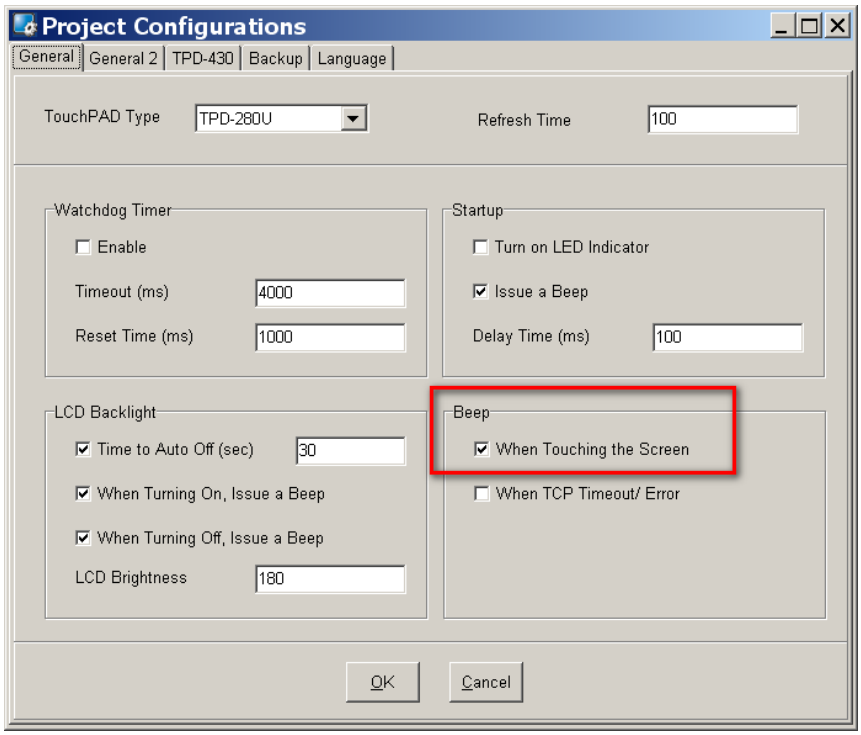
```
                100, G7,
                125, A7,
                150, B7,
                 175, SILENCE
            };    // length = 16

        void BitButton4OnClick(tWidget *pWidget)
        {
            hmi_PlaySong(g_pusKeyClick, sizeof(g_pusKeyClick) / 2);
        }
```

- **Remark**

⚠ *Note that when the **"When Touching the Screen"** item is checked, this function is disabled automatically.*

The **"When Touching the Screen"** option can be found at the project configuration form, from the **"Project Configuration"** item in the **"HMI"** menu, as shown below.



14.3 hmi_ConfigBeep

Configure the beep of the TPD-430.

- **Syntax**

```
void hmi_ConfigBeep(
        unsigned short usFreq,
        unsigned short usTicksMS
);
```

- **Parameter**

usFreq

[Input] Specify the pitch (the frequency value) of the beep.
Range: 30 ~ 4,000 Hz.

usTickMS

[Input] Specify the elapsing interval of the beep.
Range: 1 ~ 30,000 ms.

- **Return Values**

None

- **Examples**

[C]

```
//beep at the the specified pitch 100 Hz, and 5ms long a beep
hmi_ConfigBeep (100, 5);
```

- **Remark**

None

14.4 hmi_GetRotaryID

Get the ID of the rotary switch.

- **Syntax**

```
int hmi_GetRotaryID();
```

- **Parameter**

None

- **Return Values**

None

- **Examples**

[C]

```
int id = hmi_GetRotaryID();
```

- **Remark**

None

14.5 hmi_SetLED

Set the LED indicator of a TouchPAD device.
The supported TouchPAD devices include
TPD-430/TPD-430-EU/TPD-433/TPD-433-EU/TPD-432F/TPD-433F/VPD-130/
VPD-130N/VPD-132/VPD-132N/VPD-133/VPD-133N/VPD-142/VPD-142N/VPD-143/ VPD-143N.

- **Syntax**

```
void hmi_SetLED(
        int status
);
```

- **Parameter**

status

[Input] Specify the status of the LED indicator. There are two states of the LED indicator, HMI_LED_ON and HMI_LED_OFF.

- **Return Values**

None

- **Examples**

[C]

```
int count = 0;

if(count % 2)
hmi_SetLED (HMI_LED_ON);    //turn on the LED indicator
else
hmi_SetLED (HMI_LED_OFF);    //turn off the LED indicator
```

- **Remark**

The original hmi_ShowPanelLed function for the VPD series now is depreciated. Replace hmi_ShowPanelLed with hmi_SetLED.

14.6 hmi_BacklightSet

Set the brightness of all the models in the TouchPAD series.
Or turn on or off the back light of other devices in the TouchPAD series.

- **Syntax**

```
void hmi_BacklightSet(
        unsigned char ucBrightness
);
```

- **Parameter**

ucBrightness

[Input] Specify the brightness of TouchPAD.
Range: 0 ~ 255.
0=the darkest, ..., 255=the brightest.

- **Return Values**

None

- **Examples**

[C]

```
unsigned char b = 128;

hmi_BacklightSet(b);    //set the brightness to 128
```

- **Remark**

None

14.7 hmi_ReadPanelKey

Get the states of the panel key on the front panel.

- **Syntax**

```
int hmi_ReadPanelKey();
```

- **Parameter**

None

- **Return Values**

When the state of the key is pressed, the corresponding value in the table below is returned.

Key	Value	Define #1	Define #2
No press	0	N/A	N/A
	1	PANEL_KEY_F1	PANEL_KEY_UP
	2	PANEL_KEY_F2	PANEL_KEY_DOWN
	4	PANEL_KEY_F3	PANEL_KEY_LEFT
	8	PANEL_KEY_F4	PANEL_KEY_RIGHT
	16	PANEL_KEY_F5	PANEL_KEY_ENTER

- **Examples**

[C]

```
void Timer6OnExecute(tWidget *pWidget)
{
    static char str[20];
    int reading = -1;

    reading = hmi_ReadPanelKey();

    usprintf(str, "%d", reading);
    LabelTextSet(&Label5, str);
}
```

- **Remark**

None

14.8 hmi_GetTickCount

Get the tick count of the TouchPAD.

- **Syntax**

```
unsigned long hmi_GetTickCount();
```

- **Parameter**

None

- **Return Values**

The system tick count in the unit of milisecond. The resolution is about 10 ms.

That is, this hmi_GetTickCount function is based on a fixed time interval of 1000 ticks/second.

- **Examples**

[C]

```
void BitButton4OnClick(tWidget *pWidget)
{
    static char str[16];
    unsigned long tick = hmi_GetTickCount();

    usprintf(str, "tick= %d", tick);
    LabelTextSet(&Label5, str);
}
```

- **Remark**

None

14.9 hmi_DelayUS

Delay a specified interval in micro-second.

- **Syntax**

```
void hmi_DelayUS(
        unsigned long ulDelayTime
);
```

- **Parameter**

ulDelayTime

[Input] Specify the delay time in micro-second.

Suggested range: 1 ~ 50 (us) **in order not to block the system**.

- **Return Values**

None

- **Examples**

[C]

```
hmi_DelayUS(10); //delay 10 us
```

- **Remark**

TouchPAD is not a multitasking system.

Delay too much severely blocks the system.

14.10 hmi_GetDateTime

Get the date and time from the RTC chip on the TouchPAD devices.
Not all the devices in the TouchPAD series equips with a RTC chip.

- **Syntax**

```
void hmi_GetDateTime(
        int *year,
        int *month,
        int *day,
        int *hour,
        int *minute,
        int *second
);
```

- **Parameter**

year
[Output] Specify the pointer to the integer used to represent the year.

month
[Output] Specify the pointer to the integer used to represent the month.

day
[Output] Specify the pointer to the integer used to represent the day.

hour
[Output] Specify the pointer to the integer used to represent the hour.

minute
[Output] Specify the pointer to the integer used to represent the minute.

second
[Output] Specify the pointer to the integer used to represent the second.

- **Return Values**

None

- **Examples**

[C]

```
void TextPushButton4OnClick(tWidget *pWidget)
{
    static char temp[32];
    int yy, mm, dd, hh, nn, ss;

    // set date and time to RTC (Real Time Clock)
    // Set to 13:31:22, Jan 2nd, 2012
    yy = 2012;    mm = 1;    dd = 2;
    hh = 13;       nn =31;    ss = 22;
    hmi_SetDateTime(yy, mm, dd, hh, nn, ss);

    // get date and time from RTC
    hmi_GetDateTime(&yy, &mm, &dd, &hh, &nn, &ss);

    usprintf(temp, "%04d/%02d/%02d %02d:%02d:%02d\n", yy, mm, dd, hh, nn, ss);
    LabelTextSet(&Label5, temp);
}
```

- **Remark**

A RTC chip is required for hmi_GetDateTime and hmi_SetDateTime.

14.11 hmi_SetDateTime

Set the date and time to the RTC chip on the TouchPAD devices.
Not all the devices in the TouchPAD series equips with a RTC chip.

- **Syntax**

```
void hmi_SetDateTime(
        int year,
        int month,
        int day,
        int hour,
        int minute,
        int second
);
```

- **Parameter**

year
[Input] Specify the value of the year to set to the RTC chip.

month
[Input] Specify the value of the month to set to the RTC chip.

day
[Input] Specify the value of the day to set to the RTC chip.

hour
[Input] Specify the value of the hour to set to the RTC chip.

minute
[Input] Specify the value of the minute to set to the RTC chip.

second
[Input] Specify the value of the second to set to the RTC chip.

- **Return Values**

None

- **Examples**

[C]

```
void TextPushButton4OnClick(tWidget *pWidget)
{
    static char temp[32];
    int yy, mm, dd, hh, nn, ss;

    // set date and time to RTC (Real Time Clock)
    // Set to 13:31:22, Jan 2nd, 2012
    yy = 2012;    mm = 1;    dd = 2;
    hh = 13;      nn =31;    ss = 22;
    hmi_SetDateTime(yy, mm, dd, hh, nn, ss);

    // get date and time from RTC
    hmi_GetDateTime(&yy, &mm, &dd, &hh, &nn, &ss);

    usprintf(temp, "%04d/%02d/%02d %02d:%02d:%02d\n", yy, mm, dd, hh, nn, ss);
    LabelTextSet(&Label5, temp);
}
```

- **Remark**

A RTC chip is required for hmi_GetDateTime and hmi_SetDateTime.

14.12 CRC16

Get a 16-bit cyclic redundancy check value of an array of bytes.

- **Syntax**

```
Unsigned short CRC16(
                    const imsogmed char *nData,
                    unsigned short wLength
);
```

- **Parameter**

nData

[Input] Specify the pointer to the data to calculate its CRC value

wLength

[Input] the length of data to calculate

- **Return Values**

The 16-bit CRC value

- **Examples**

[C]

```
//
// Use the CRC16 in the Modbus coil command
//
// DO command (Modbus Coil)
//str[0] = slave address
//str[1] = function
//str[2,3] = start address
//str[4,5] = length
//str[6] = Byte ((length+7)/8)
//str[7] = value
//str[9, 10] = checksum

char recv[1+1+2+2+2];

// a cmd of DO (Modbus coil) for example
unsigned char cmd[] = {0x01, 0x0F, 0x00, 0x00, 0x00, 0x10, 0x03, 0x11, 0, 0};

// calculate the 16-bit CRC value of the cmd
unsigned short ret_crc = CRC16(cmd, sizeof(cmd)-2);
```

```
// put the CRC value in the last 2 bytes of the cmd
cmd[sizeof(cmd)-2]      = ret_crc & 0xff;
cmd[sizeof(cmd)-1]      = ret_crc >>8;
recv[0] = 0;

HANDLE h = uart_Open("COM1,115200,N,8,1");
uart_BinSendCmd(h, cmd, sizeof(cmd), recv, sizeof(recv));
uart_Close(h);
```

- **Remark**

  None

14.13 FloatToStr

Convert a floating-point number of the type float to string.

- **Syntax**

```
int FloatToStr(
    char *buf,
    float fVal,
    int precision
);
```

- **Parameter**

buf

[Output] Specify the pointer to a char array to represent the floating-point number.

fVal

[Input] the floating-point number to be converted

precision

[Input] the number of digit after the decimal point (round off)

Possible range: 0 ~ 5.

- **Return Values**

0 (False),

1 (OK)

- **Examples**

[C]

```
void BitButton4OnClick(tWidget *pWidget)
{
    static char buf[20];
    float fVal = -2.3617;
    int precision = 3;    // 0 ~ 5 assigned by user

    FloatToStr(buf, fVal, precision);
    LabelTextSet(&Label5, buf);    // the result is -2.362
}
```

- **Remark**

None

14.14 hmi_LCDIdleSetCallback

This function is used to configure the callback function as TouchPAD's idle or wake up event handlers.

- **Syntax**

```
void hmi_LCDIdleSetCallback(
        unsigned long ulTimeoutMS,
        PFN_VOIDCALLBACK pfnIdle,
        PFN_VOIDCALLBACK pfnWakeup
);
```

- **Parameter**

ulTimeoutMS

[Input] Specify the timeout value to enter the idle mode. (unit: ms)

pfnIdle

[Input] Specify the function pointer to the callback function that is fired when entering the idle state.

pfnWakeup

[Input] Specify the function pointer to the callback function that is fired when the touch screen wakes up.

⚠ *Note: The prototype of the idle or wakeup callback functions is defined as below:*

```
typedef void (*PFN_VOIDCALLBACK)(void);
```

- **Return Values**

None

- **Examples**

[C]

```
// Callback function when TouchPAD entering the idle state.
void TLCDIdle()
{
    // go to the frame of the screen saver when idle
    hmi_GotoFrameByName("FScreenSaver");
}
```

⚠ *Note: Don't have codes after calling **hmi_GotoFrameByName()**.*

```
// Callback function when TouchPAD wakes up.
void TLCDWakeup()
{
    hmi_BacklightSet(255); // Enable LCD
    hmi_GotoFrameByName("FHome");
}
```

⚠ *Note: Don't have codes after calling **hmi_GotoFrameByName()**.*

```
void FHome2OnCreate()
{
    // Configuration for the callback functions when idle atfter 5 seconds.
    hmi_LCDIdleSetCallback(5 * 1000, TLCDIdle, TLCDWakeup);
}
```

- **Remark**

  How to disable the screen saver?

  Simply set the first argument of hmi_LCDIdleSetCallback to zero and run it.

  That is,

  ---

  hmi_LCDIdleSetCallback(**0**, TLCDIdle, TLCDWakeup);

14.15 hmi_LCDIdleStatusReset

This function is used to reset the idle state of TouchPAD and enable the next callback when idle state is entered again.

When to use? For example, assume that we have an application that make TouchPAD change from the idle page to the alarm page when needed. And after the alarm turned off, TouchPAD then goes back to the home page automatically.

In this case at this time, hmi_LCDIdleStatusReset is used to tell TouchPAD to start to listen to the idle callback again for the next possible idleness because if you don't do this, TouchPAD is still in its idle state and cannot turn off the screen.

Remember that the idle and wakeup state is based on touch or no-touch on the LCD. When in the idle state, the program is still running, not sleeping.

- **Syntax**

```
void hmi_LCDIdleStatusReset();
```

- **Parameter**

None

- **Return Values**

None

- **Examples**

[C]

```
hmi_LCDIdleStatusReset();
```

- **Remark**

None

14.16 hmi_Pack

- **Syntax**

```
void hmi_Pack(
        char *pcData,
        BOOL *pbDio,
        int tot_ch
);
```

- **Parameter**

pcData
[Output] The data array of type char which is used in Modbus API functions

pbDio
[Input] The array of the binary data usually representing digital I/Os

tot_ch
[Input] Total channel

- **Return Values**

None

- **Examples**

[C]

```
char input[1];
char bytes[1];
BOOL bits[8];

input[0]=1;

// Unpack the simulated Modbus data to bits[]
hmi_Unpack (input,bits,8);

if(    bits[0] == 0)LabelTextSet(&Label5,"Unpack failed");
else LabelTextSet(&Label5,"Unpack OK");

// Pack the unpacked bits[] back to Modbus data, bytes[]
hmi_Pack (bytes,bits,8);

if(input[0] != bytes[0]) LabelTextSet(&Label6,"Pack failed");

else LabelTextSet(&Label6,"Pack OK");
```

- **Remark**

None

14.17 hmi_Unpack

- **Syntax**

```
void hmi_UnPack(
        char *pcData,
        BOOL *pbDio,
        int tot_ch
);
```

- **Parameter**

pcData
[Input] The data array of type char which is used in Modbus API functions

pbDio
[Output] The array of the binary data usually representing digital I/Os

tot_ch
[Input] Total channel

- **Return Values**

None

- **Examples**

[C]

```
char input[1];
char bytes[1];
BOOL bits[8];

input[0]=1;

// Unpack the simulated Modbus data to bits[]
hmi_Unpack (input,bits,8);

if( bits[0] == 0)LabelTextSet(&Label5,"Unpack failed");

else LabelTextSet(&Label5,"Unpack OK");

// Pack the unpacked bits[] back to Modbus data, bytes[]
hmi_Pack (bytes,bits,8);

if(input[0] != bytes[0]) LabelTextSet(&Label6,"Pack failed");

else LabelTextSet(&Label6,"Pack OK");
```

- **Remark**

None

14.18 hmi_SoftwareReset

Reset the TouchPAD device by software (using this API function).

- **Syntax**

```
void hmi_SoftwareReset();
```

- **Parameter**

None

- **Return Values**

None

- **Examples**

[C]

```
hmi_SoftwareReset();
```

- **Remark**

None

14.19 hmi_SerialNumberGet

Get the unique hardware serial number for each TouchPAD device.

⚠ *Note: support **only** for TPD-*-H or TPD-*-Mx. (where * means several characters)*

- **Syntax**

```
int hmi_SerialNumberGet(
        unsigned long *uniqid,
        int len
);
```

- **Parameter**

uniqid

[Output] An array of type **"unsigned long"** which is used to store the unique ID

len

[Input] The length of the array of uniqid (unit: 4-byte word)

Currently, the unique ID has length of 4, that is, total 16 bytes.

- **Return Values**

None (Reserved for future use)

- **Examples**

[C]

```
unsigned long uniqid[4];
tLabel* lb[4];
char msg[4][16];

void BitButton4OnClick(tWidget *pWidget)
{
hmi_SerialNumberGet(uniqid, 4);

    for (int i = 0; i < 4; i++)
    {
        usprintf(msg[i], "%08x", uniqid[i]);
        LabelTextSet(lb[i], msg[i]);
    }
}
```

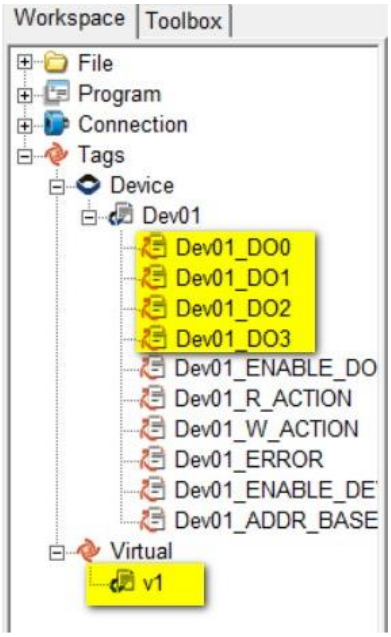
```
void Frame12OnCreate()
{
        lb[0] = &Label5;
        lb[1] = &Label6;
        lb[2] = &Label7;
        lb[3] = &Label8;
}
```

- **Remark**

  Support only for TPD-*-H or TPD-*-Mx. (where * means several characters)

  Other models in the TouchPAD series do not support this function.

14.20 Macros for Device I/O Tag and Virtual Tag



VAR_VALUE( )

Get the value of a device I/O Tag or Virtual Tag.

- **Examples**

[C]

```
int iValue = VAR_VALUE(Dev01_DO1);
```

VAR_SET( )

Set the value of a device I/O Tag or Virtual Tag.
When read-back a status from a device, the VAR_SET( ) macro is used to change the value of a device I/O Tag.

- **Examples**

[C]

```
int iValue = 1;
VAR_SET( Dev01_DO0, iValue);
```


VAR_SET_WRITE_DATA( )

Set the value of a device I/O Tag or Virtual Tag with WRITE flag.
When application want to change the remote device I/O status, the VAR_SET_WRITE_DATA( ) macro can be used.
Don't use VAR_SET_WRITE_DATA() macro on a read-back status from a device,
since the TouchPAD may update the device later because the WRITE flag is set.
This can cause a busy loop of read-back and update.

- **Examples**

[C]

```
int iValue = 1;
VAR_SET_WRITE_DATA( Dev01_DO0, iValue);
```

15. DGW-521 API

15.1 dgw_Init

Initialize a DGW-521 device.

- **Syntax**

```
HANDLE dgw_Init(
        HANDLE hSerial,
        unsigned char ModbusID,
        DGW_CONTROL_BLOCK *pDGW
);
```

- **Parameter**

hSerial
[Input] The handle value of the serial port connected with the DGW-521 module.

ModbusID
[Input] The Modbus ID of the DGW-521 module.

pDGW
[Input] The data buffer of the DGW-521 module.

- **Return Values**

Handle value 0 - 15 for DGW-521 module, or -1 for out of range.

- **Examples**

[C]

```
DGW_CONTROL_BLOCK dcb;
HANDLE hDGW = INVALID_HANDLE;
HANDLE hSerial = INVALID_HANDLE;
Int ModbusID=1;

hSerial = uart_Open("COM1,9600,8N1");
hDGW = dgw_Init(hSerial, ModbusID, &dcb);
```

- **Remark**

None

15.2 dgw_Remove

Remove a DGW-521 handle.

- **Syntax**

```
int dgw_Remove(
    HANDLE hDGW
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

- **Return Values**

error code.

- **Examples**

[C]

```
dgw_Remove( hDGW);
```

- **Remark**

None

15.3 dgw_SetLampLevel

Set lamp output level.

- **Syntax**

```
int dgw_SetLampLevel(
    HANDLE hDGW,
    unsigned char address,
    unsigned char data
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

address
[Input] DALI address 0 - 63, same as DALI spec.

data
[Input] 0 - 254 for lamp output level, same as DALI spec.

- **Return Values**

error code.

- **Examples**

[C]

```
Int lamp_addr=1;

dgw_SetLampLevel(hDGW, lamp_addr, 10);
```

- **Remark**

None

15.4 dgw_SetAllLampsLevel

Set all lamps output level.

- **Syntax**

```
int dgw_SetAllLampsLevel(
    HANDLE hDGW,
    unsigned char *datalist
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

datalist

[Input] Pointer to a list of 64-lamp output level (0 - 254), no change (255), same as DALI spec.

- **Return Values**

error code.

- **Examples**

[C]

```
unsigned char data[64] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
                          17, 18, 19, 20, 21}; // all others are 0 (off)

dgw_SetAllLampsLevel(hDGW, data);
```

;

- **Remark**

None

15.5 dgw_SetGroupLevel

Set group output level.

- **Syntax**

```
int dgw_SetGroupLevel(
    HANDLE hDGW,
    unsigned char group,
    unsigned char data
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

group
[Input] Group address 0 - 15, same as DALI spec.

data
[Input] 0 - 254 for group output level, same as DALI spec.

- **Return Values**

error code.

- **Examples**

[C]

```
Int group_num=1;

dgw_SetGroupLevel (hDGW, group_num, 10);
```

- **Remark**

None

15.6 dgw_SetAllGroupsLevel

Set all group output level.

- **Syntax**

```
int dgw_SetAllGroupsLevel(
    HANDLE hDGW,
    unsigned char *datalist
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

datalist

[Input] Pointer to a list of 16-group output level (0-245), no change (255), same as DALI spec.

- **Return Values**

error code.

- **Examples**

[C]

```
unsigned char data[16] = {10, 20, 30}; // all others are 0 (off)

dgw_SetAllGroupsLevel(hDGW, data);
```

- **Remark**

None

15.7 dgw_Scan

Scan DALI slave devices connected on the DGW-521 module. The dgw_GetScanState() function is required until scan finished.

- **Syntax**

```
int dgw_Scan(
    HANDLE hDGW,
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

- **Return Values**

error code.

- **Examples**

[C]

```
int ScanState=0;
unsigned int u32[2];

void BitButton26OnClick(tWidget *pWidget)
{
    dgw_Scan(hDGW);
    ScanState = -1;
}

void Timer6OnExecute(tWidget *pWidget)
{
    if(ScanState != 0)
        ScanState = dgw_GetScanState(hDGW, u32);
    else
        dgw_GetPresences(hDGW, u32);
}
```

- **Remark**

None

15.8 dgw_GetScanState

Get state and result of scan DALI slave device. This function is required until scan finished.

- **Syntax**

```
int dgw_GetScanState(
    HANDLE hDGW,
    unsigned int *presence
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

presence

[Input] Pointers to two 32-bit data buffer for presence of 64 DALI slaves.

- **Return Values**

0=Scan Finish, 1=Scan Busy, Others for error codes.

- **Examples**

[C]

```
int ScanState=0;
unsigned int u32[2];

void BitButton26OnClick(tWidget *pWidget)
{
    dgw_Scan(hDGW);
    ScanState = -1;
}
void Timer6OnExecute(tWidget *pWidget)
{
    if(ScanState != 0)
        ScanState = dgw_GetScanState(hDGW, u32);
    else
        dgw_GetPresences(hDGW, u32);
}
```

- **Remark**

None

15.9 dgw_Process

This function should be periodic called (for example: 500 ms) to process a single DGW-521 module at a time.
Users can add delay for the RS-485 bus silent-time if needed before switching communications to another DGW-521 module..

- **Syntax**

```
int dgw_Process(
    HANDLE hDGW
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

- **Return Values**

error code.

- **Examples**

[C]

```
void Timer19OnExecute(tWidget *pWidget)
{
  dgw_Process(hDGW); // Process Modbus RTU commands (response) to (from) the DGW module.
};
```

- **Remark**

None

15.10 dgw_GetAllLampsLevel

Get lamp output level recorded in the TPD.

- **Syntax**

```
int dgw_GetAllLampsLevel(
    HANDLE hDGW,
    unsigned char *datalist
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

datalist

[Input] Pointer to the buffer to store output levels of 64 lamps.

- **Return Values**

error code.

- **Examples**

[C]

```
unsigned char LampLevel[64];

dgw_GetAllLampsLevel(hDGW, LampLevel);
```

;

- **Remark**

None

15.11 dgw_SendReadback

To read back a lamp output level. Since the TouchPAD and DGW-521 module do not maintain the relationship between group and lamps, there is no lamp output level information after group output operations.
Users may use this optional function to trigger the DGW-521 module to read back the lamp output level if the information is required. The dgw_Sate() function is used to see if the operation is finished or not.

- **Syntax**

```
int dgw_SendReadback(
    HANDLE hDGW,
    unsigned char addr
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

addr
[Input] The DALI address of lamp.

- **Return Values**

error code.

- **Examples**

[C]

```
Int addr =1;
unsigned char data[16] = {10, 20, 30}; // all others are 0 (off)

dgw_SetAllGroupsLevel(hDGW, data);
dgw_SendReadback(hDGW,addr); // Manually read back after set group level

void Timer6OnExecute(tWidget *pWidget)
{
    static char szMsg[10];
    usprintf(szMsg, "St: %d", dgw_State(hDGW) ); // show DGW state
    LabelTextSet(&Label5, szMsg);
}
```

- **Remark**

None

15.12 dgw_SendReadbackAll

To read back all lamp output level. Since the TouchPAD and DGW-521 module do not maintain the relationship between group and lamps, there is no lamp output level information after group output operations.
Users may use this optional function to trigger the DGW-521 module to read back the lamp output level if the information is required. The dgw_Sate() function is used to see if the operation is finished or not.

- **Syntax**

```
int dgw_SendReadbackAll(
    HANDLE hDGW,
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

- **Return Values**

error code.

- **Examples**

[C]

```
unsigned char data[16] = {10, 20, 30}; // all others are 0 (off)

dgw_SetAllGroupsLevel(hDGW, data);
dgw_SendReadbackAll(hDGW); // Manually read back after set group level

void Timer6OnExecute(tWidget *pWidget)
{
    static char szMsg[10];
    usprintf(szMsg, "St: %d", dgw_State(hDGW) ); // show DGW state
    LabelTextSet(&Label5, szMsg);
}
```

- **Remark**

None

15.13 dgw_ResendOutput

To resend lamp output level. This function may be used after DGW module and lamps are rebooted that requires TPD to resend last output level to the DGW module.

- **Syntax**

```
int dgw_ResendOutput (
    HANDLE hDGW,
    unsigned char addr
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

addr

[Input] The DALI address 0 – 63 of lamp.

- **Return Values**

error code.

- **Examples**

[C]

```
Int lamp_addr=1;

dgw_ResendOutput(hDGW, lamp_addr);
```

- **Remark**

None

15.14 dgw_ResendOutputAll

To resend all lamps output level. This function may be used after DGW module and lamps are rebooted requires TPD to resend last output level to the DGW module.

- **Syntax**

```
int dgw_ResendOutputAll(
    HANDLE hDGW
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

- **Return Values**

error code.

- **Examples**

[C]

```
dgw_ResendOutputAll(hDGW);
```

- **Remark**

None

15.15 dgw_State

Get DGW-521 state. For example, after sending commands, you may want to check if it is finished or still in busy state.

- **Syntax**

```
int dgw_State (
    HANDLE hDGW,
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

- **Return Values**

DGW_INIT	**0**
DGW_SCAN	**1**
DGW_IDLE / DGW_READY	**2**
DGW_BUSY	**3**

- **Examples**

[C]

```
Int d_state;

d_state = dgw_State(hDGW);
```

- **Remark**

None

15.16 dgw_GetPresences

Get presences state of DALI slaves.

- **Syntax**

```
int dgw_GetPresences(
    HANDLE hDGW,
    unsigned int *datalist
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

Datalist

[Input] Pointer to a list of 32-bit data buffers to store 64-bit states.

- **Return Values**

error code.

- **Examples**

[C1] When initialize a DGW-521 device.

```
DGW_CONTROL_BLOCK dcb;
HANDLE hDGW = INVALID_HANDLE;
HANDLE hSerial = INVALID_HANDLE;
Int ModbusID=1;
Int d_state;
unsigned int u32[2];

hSerial = uart_Open("COM1,9600,8N1");
hDGW = dgw_Init(hSerial, ModbusID, &dcb);

void Timer6OnExecute(tWidget *pWidget)
{
   d_state = dgw_State(hDGW);

   if(d_state == 2 ) // DGW_IDLE    or    DGW_READY
      dgw_GetPresences(hDGW, u32);
}
```


[C2] The DALI slave device has changed, it is necessary to rescan the DALI slave device connected to the DGW-521 module, for example: replace the device.

```
int ScanState=0;
unsigned int u32[2];

void BitButton26OnClick(tWidget *pWidget)
{
        dgw_Scan(hDGW); //Rescan the DALI slave device
        ScanState = -1;
}

void Timer6OnExecute(tWidget *pWidget)
{
         if(ScanState != 0)
                ScanState = dgw_GetScanState(hDGW, u32);
         else
                dgw_GetPresences(hDGW, u32);
}
```

- **Remark**

  None

15.17 dgw_SendCommand

Send DALI command.

- **Syntax**

```
int dgw_ SendCommand (
    HANDLE hDGW,
    int idxBuf,
    unsigned short wCMD
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

idxBuf
[Input] 1 - 8 for command buffer index.

wCMD
[Input] high-byte= DALI address, low-byte= command or data.
Refer to DALI specification and DGW-521 manaul for details.

- **Return Values**

error code.

15.18 dgw_SendCommand2

Send DALI command.

- **Syntax**

```
int dgw_ SendCommand2 (
    HANDLE hDGW,
    int idxBuf,
    char addrType,
    char address,
    char cmdType,
    unsigned char cmdByte
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

idxBuf
[Input] 1 - 8 for command buffer index.

addrType
[Input] 0=DALI Address, 1=Group Address, 2=Broadcast.

address
[Input] 0-63 for DALI address, 0-15 for group address.

cmdType
[Input] 0=Direct Lamp Power Level, 1=Command.

cmdByte
[Input] Lamp power level or command.

- **Return Values**

error code.

15.19 dgw_GotoScene

Get DALI response.

- **Syntax**

```
int dgw_ GotoScene (
    HANDLE hDGW,
    int idxBuf,
    char addrType,
    char address,
    char scene
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

idxBuf
[Input] 1 - 8 for command buffer index.

addrType
[Input] 0=DALI Address, 1=Group Address, 2=Broadcast.

address
[Input] 0-63 for DALI address, 0-15 for group address.

scene
[Input] 0-15.

- **Return Values**

error code.

15.20 dgw_CheckResponse

Get DALI response.

- **Syntax**

```
int dgw_ CheckResponse (
    HANDLE hDGW,
    int idxBuf,
    unsigned short *data
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

idxBuf
[Input] 1 - 8 for command buffer index.

data
[Input] Pointer to a 16-bit data buffer for storing response data and state.
0-7 bits for DALI response state, 8-15 bits DALI response data. Same as DGW-521.

- **Return Values**

error code.

15.21 dgw_GetCommandBuffer

This function gets an available command buffer index (1 - 8).
User can also manage the 8 command buffers by them-self, so the get/release command buffer functions can be ignored.

- **Syntax**

```
int dgw_ GetCommandBuffer (
    HANDLE hDGW
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

- **Return Values**

1 - 8 for command buffer index, or error code.

15.22 dgw_ReleaseCommandBuffer

The dgw_ReleaseCommandBuffer() function is required when using dgw_GetCommandBuffer(), so the library can get which command buffer is available.
Users can also skip these two functions and manage the command buffer state by them-self.
For example: Only use a single command with the command buffer #1 to send / receive response, so there is no need to get and release command buffer operations.

- **Syntax**

```
int dgw_ GotoScene (
    HANDLE hDGW,
    int idxBuf
);
```

- **Parameter**

hDGW
[Input] Handle value of the DGW-521 module.

idxBuf
[Input] 1 - 8 for command buffer index.

- **Return Values**

error code.

15.23 dgw_SetOperationMode

This defines the TPD operations after DGW-521 recovering from communication errors.

- **Syntax**

```
int dgw_SetOperationMode(
    HANDLE hDGW,
    unsigned int iMode
);
```

- **Parameter**

hDGW

[Input] Handle value of the DGW-521 module.

iMode

[Input]

Value	Parameter	Describe
0	DGW_USER_CONTROL	No action, User control
1	DGW_READBACK_LAMPS	Readback level of lamps
2	DGW_LAST_LAMPS_LEVEL	Output last level of lamps
3	DGW_LAST_GROUPS_LEVEL	Output last level of groups
4	DGW_AUTO_LAMPS_GROUPS	Auto switching between Last-Lamps-Level and Last-Groups-Level modes when output lamp level or group level.

- **Return Values**

error code.

- **Examples**

[C]

```
DGW_CONTROL_BLOCK dcb;
HANDLE hDGW = INVALID_HANDLE;
HANDLE hSerial = INVALID_HANDLE;
Int ModbusID=1;

hDGW = dgw_Init(hSerial, ModbusID, &dcb);
dgw_SetOperationMode(hDGW, DGW_AUTO_LAMPS_GROUPS);
```

- **Remark**

None