# SIGMA

## SIGMAP02 – Plugin Developer's Manual

*Application Note*

| | |
|---|---|
| Address: | ASIX s.r.o.<br>Staropramenna 4<br>150 00 Prague<br>Czech Republic |
| E-Mail: | sales@asix.net (sales inquiries, ordering)<br>support@asix.net (technical support) |
| WWW: | www.asix.net (company website) |
| Tel.: | +420-257 312 378 |
| Fax: | +420-257 329 116 |

# Table of Contents

# 1. Plugins overview

*SIGMA* Logic Analyzer plugins can add functionality into software and hardware of *SIGMA*. This includes:

- protocol and data (signal) analyzers and conditioners
- visual and controls enhacements
- user added specific hardware and software functionality

Depending on plugin type you may want to create or modify, plugins can be divided into three groups:

- *SIGMA* specific (will work with and only with *SIGMA*)
  when you want to add some functionality into *SIGMA* hardware or interact directly with *SIGMA* hardware
- Analyzer plugins (will work with different (maybe future) analyzers)
  when you want create, for example, bus analyzer
  there is no need to interact with *SIGMA* hardware
- Generic plugins (no analyzer specific functions are used)
  when you want create, for example, a new trace by combining two another traces (maybe from other plugin, maybe from analyzer – this does not matter)

# 2. Data manipulation schema

All functions are thread unsafe. Plugin can call main application callback functions only when main application call plugin itself. Main application will call plugin always only from its main thread.

All functions in plugin interface use calling convention **stdcall**. This is the same as used in *winapi* calls and different from standard C call.

## 2.1 Traces and Inputs, Busses

There are two basic different types of data handled by the software:

- *Inputs*
- *Traces*

Plugin can obtain each *Input* accessible in the system (from analyzer, another plugin) and plugin can obtain visible *Traces*. *Traces*, which are not displayed in viewer window, are not accessible (plugin's *Trace* will become existing when it is displayed).

Plugin can call `getinput()` and `gettrace()` functions in its own `plugin_getinput()` and `plugin_gettrace()` functions. Number of calls to `getinput()` and `gettrace()` are limited to depth of 20 calls, which prevents stuck from circular infinite call.

### 2.1.1 Inputs

Each *Input* has range, determined by minimum and maximum value it can represent, in range of signed 32 bit integer. For logic signals, range should be from 0 to 1. For each time point has each *Input* its minimum and maximum value, or it is undefined in that time point. (Different minimum and maximum values in one time point represents uncertainty of signal, undefined value means that there was no measuring at this time).

*Inputs* are transfered between main application and plugins in *Input Snapshots*. *Input Snapshot* is array of values between time points from specified time with specified period. All time manipulation is specified in *PU* (=*PicoUnit*, defined in header file as ~~1 *PU* = 1/15015 ns, ~0.0667 ps~~ 1 *PU* = 1/15018 ns, ~0.0666 ps) with „the world“ beginning at 0 *PU* (inputs are always undefined before beginning of the world).

1 PU = 1/15015ns was selected with idea of sample rates like 33.3 MHz. Recent development of hardware shows that also more than 100MHz sample rates are needed. 1 PU = 1/15018ns was selected to allow express sample rate 2.5ns (400MHz).
All newly compiled plugins should use this updated value. All stored files, which have old value which sample rate comforms to `((samplerate % 15015)==0)` are converted to new value.

If the main component, which sources *Inputs* (this means analyzer, source file, etc...) cannot guarantee periodic data sampling (sampled by external clock, source file does not include information on sample frequency, etc...), simple conversion is made, that smallest differetiable unit (sample rate) is 15016 *PU* (as defined in header file). In this case, everything seems that sample rate is equal to 15016 *PU*.

### 2.1.2 Traces

*Trace* is one line in Viewer Window. Invisible (not displayed) *Traces* are not accessible by plugin. Each *Input* which is visible in Viewer Window is also a (very simple) *Trace*.

*Traces* are transfered between plugins and main application in *Trace Snapshots*, data structure simalar to *Input Snapshot*, extended by *Trace*-only related data.

*Trace* can be either *Standard* or *Bus*.

When *Standard Trace* is drawn by Viewer Window, same algorithm is used as when *Traces* composed of single *Input* are drawn. This algorithm does linking of discontinuity courses by vertical bonds (for example on edges of digital signal).

While drawing *Bus Trace*, no such linking is done and the *Trace* is drawed as is. Two different line styles can be used: with fill and without fill depending on order of minimum and maximum values (minimum higher than maximum causes fill).

## 2.2 Highlighting

Each plugin can place *Highlights* into Viewer Window. The *Highlights* may be rectangles and lines placed under waveforms, lines above waveforms and texts.

There are two types of *Highlights*, *Non-indexed Highlights* and *Indexed (tagged) Highlights*. *Indexed Highlight* is only once per index in the system and remains active between calls of `paintnotify()`. *Non-indexed Highlights* must be specified each call of `paintnotify()` function.

*Indexed Highlights* can be placed from any plugin function. *Non-indexed highlights* can be placed only from `paintnotify()` function.

## 2.3 Menu items

Each plugin can allocate *Menu Items* in main window's main menu. Main window is populating plugin *Menu Items* using calls to plugin function `getmenuitem()`. Viewer window pop-up *Menu Items* are populated by calls to plugin function `getpopupitem()` in the moment of user's pop-up. The information about user's menu selection (click on *Menu Item*) is sent to plugin by call to callback funtion specified by plugin.

## 2.4 Hints

Each time main window updates *Hint* (yellow box hovering with mouse), main application calls plugin function `updatehint()` where plugin can add its own *Hint*. Plugin can trigger update of *Hint* by sending a message to main window. All *Hints* from different plugins are displayed together, each plugin's *Hint* on one line.

## 2.5 Unified settings storing

There are several functions to manipulate with *Settings* in unified way. Plugin can store *Settings* into main application database and telling wheather this *Setting* should be stored in registry and wheather this *Setting* should be saved to and loaded from a test file.

There are functions for reading and writing string and integer *Settings* values, for registering *Settings* and for notifying the plugin that *Settings* are going to be stored into registy or test file is going to be saved or that *Settings* were restored from registry or test file has been loaded.

Location where *Settings* are stored in registry is `HKCU\Software\ASIX\LogicAnalyzer`.

## 2.6 Config dialogs

Main application can request plugin's about or config dialogs by calling to plugin's functions `aboutdialog()` or `configdialog()`.

# 3. Messages

Messages are described primarily in `messages.txt` file.

## 3.1 Getting the window's messages

Every plugin can hear main window's messages by knowing *handle* of main window (`hWND` value in `TPluginSettings` structure is given in `plugin_plugin()` function call) and utilizing *WINAPI* function `SetWindowsHookEx()`. All keybord shortcuts, mouse moves, clicks and also internal system broadcast messages and requests goes through here.

## 3.2 WM_UPDATEVIEWER message

`WM_UPDATEVIEWER` is defined in `pluginsIncU.pas`.

Plugin can send message using `SendMessage()` or `PostMessage()` to main window to enforce Viewer Window redraw or similar associated action. `wParam` and `lParam` codes and meanings are in the following table.
When `lParam` is utilized as pointer, message must be sent using `SendMessage()` and must not be sent using `PostMessage()`. Of course, it must be sent by plugin only, mapped in same virtual memory.

| wParam | lParam | Action |
|---|---|---|
| 0 | 0 | Invalidate trace cache, redraw window<br>Use this when plugin's trace is changed (e.g. due to external event, user's interaction, ...) and notify Viewer Window about this |
|  | 2 | Redraw only. No calls to fetch plugin's traces are forced.<br>Use this when you need redraw position of cursor, etc... |
| 1 | 0 | Zoom to fit test into whole Window |
| 2 | 0 | Update hint (yellow box)<br>Use this when plugin's hint changed and you want to force call of `plugin_updatehint()` |
| 3 | pointer | Move Viewer Window. `StartTime` and `PixelTime` are in TWO_LARGE_INTEGER structure located by pointer. (times are in *PU*) |
| 4 | pointer | Add new trace to Viewer Window. Info about new trace is in `TNewTraceInfo` structure located by pointer. |

## 3.3 WM_BROADCASTEVENT message

`WM_BROADCASTEVENT` is defined in `pluginsIncU.pas`.

Plugin can send messages using `SendMessage()` (do not `PostMessage()` ! ) and receive them utilizing `SetWindowsHookEx()` .

Event is identified by NULL terminated string, pointer to that string is `wParam`. In `lParam` may be some additional parameters.

| wParam | lParam |
|---|---|
| „SetMouseCursor“ | pointer to LARGE_INTEGER with new position of mouse cursor (in *PU*) |
| „TriggerPositionChanged“ | pointer to LARGE_INTEGER with new position of trigger (in *PU*) |
| „TestClockTimeChanged“ | pointer to LARGE_INTEGER with new clock period (in *PU*) |
| „TestLengthChanged“ | pointer to LARGE_INTEGER with new test length (in *PU*) |
| „TraceCacheFlush“ | all cached data about traces should be flushed (new test load etc...) |
| "TestDataBeginUpdate" | test data is being intensively changed (i.e. begin of test download), file open |
| "TestDataEndUpdate" | cancels TestDataBeginUpdate event |
| "TestDataClear" | informs about all test data deletion (new test, new file, etc...) |

If any two plugins wants to communicate one with another, they can use new event name. Unknown events should be ignored by others.

Such message cannot be sent from another virtual memory space, from another application. When such thing is needed, two possible variants are possible:
1) plugin which will translate known strings into codes and re-send messages elsewhere
2) plugin which will copy string into shared memory and then re-send messages elsewhere

# 4. Structures

## TPluginSettings

`TPluginSettings` structure is used with exported `plugin_plugin()` function. Then called by application, plugin must fill its own function entry points and should copy application functions entry points, as well as copy and fill several other information about handle of main window, plugin's name, ID, ...

```
#define PLUGIN_API_VERSION 0x00000001

typedef struct {
// ------- filled by application, must be checked by plugin ---------
 int32 size; // set to sizeof this structure
 int32 version; // set to PLUGIN_API_VERSION
// -------- filled by application ---------
 uint32 hWND; // handle of viewer's window
 int32 pluginID; // plugin ID stored in upper 16bits
 void (*sethighlightrange)(TPluginHighlightRange *range);
 bool (*getintvalue)(char *ident, int32 *value);
 bool (*getstrvalue)(char *ident, int32 *maxlen, char *value);
 bool (*setintvalue)(char *ident, int32 value);
 bool (*setstrvalue)(char *ident, char *value);
 bool (*registerintvalue)(char *ident, int32 default, bool cansave, bool
loaddef, bool loadnow, bool filesave); // returns wheather value already existed
 bool (*registerstrvalue)(char *ident, char *default, bool cansave, bool
loaddef, bool loadnow, bool filesave); // returns wheather value already existed
 bool (*gettrace)(int32 index, TPluginTraceSnapshot *snapshot);
 bool (*gettraceinfo)(int32 index, int32 *maxlen, char *value);
 bool (*getinput)(int32 id, TPluginInputSnapshot *snapshot); // currently only
analyzer inputs are supported
 bool (*getinputinfo)(int32 index, int32 *maxlen, char *value, int32 *id);
// currently only analyzer inputs are supported
 bool (*getanalyzerraw)(int32 id, int32 start, int32 *len, TPluginRawInput
*raw);
 void (*getanalyzerrawlength)(int32 *maxlen);
 bool (*sigma_readregister)(int32 regaddr, int32 *regdata);
 bool (*sigma_writeregister)(int32 regaddr, int32 regdata);
 bool (*sigma_configure)(void *configuration, int32 configurationlength);
 void *reserved1[7]; // reserved for future functions
// ---------- filled by plugin ----------
 bool (*plugin_getnextinterest)(int32 id, int64 *Time);
 int32 plugin_class; // 0 or -1 means without class
 char *plugin_caption;
 bool plugin_alwayswantsinputlist;
 bool (*plugin_init)(bool *allowStartup);
 void (*plugin_free)(void);
 void (*plugin_savesettings)(void);
 void (*plugin_loadsettings)(void);
 bool (*plugin_getmenuitem)(int32 index, TPluginMenuItem *item);
 bool (*plugin_getpopupitem)(int32 index, TPluginMenuItem *item,
TPluginViewRange *range, int32 TraceCount, TraceInfo *infos, POINT
*mousePoint);
 void (*plugin_paintnotify)(TPluginViewRange *range, int32 TraceCount,
TTraceInfo *infos);
 void (*plugin_updatehint)(TPluginViewRange *range, int32 TraceCount,
TTraceInfo *infos, int32 *maxchar, char *text);
 bool (*plugin_getinput)(int32 id, TPluginInputSnapshot *snapshot);
 bool (*plugin_gettrace)(int32 id, TPluginTraceSnapshot *snapshot);
 bool (*plugin_getinputinfo)(int32 index, int32 *id, int32 *maxchar,
char *name);
 bool (*plugin_gettraceinfo)(int32 index, int32 *id, int32 *maxchar,
char *name);
 void (*plugin_aboutdialog)(void);
 void (*plugin_configdialog)(void);
 void (*plugin_tracesconfigdialog)(void);
} TPluginSettings;
```

`size` and `version` must be checked by plugin before touching the structure.

`hWND` is handle to main application's window, which is useful to know when sending and receiving messages.

`pluginID` is unique plugin *ID*. It is stored in upper 16 bits, always positive (only 15 bits are used).

Lower 16 bits of plugin's *Trace* and *Input IDs* are combined with plugin's *ID* to produce unique 32 bit *ID*.

`plugin_caption` is name of plugin displayed to user.

Application will not load two different plugins with same `pluginclass`. This can ensure that two plugins with same functionality will not be loaded.

When `plugin_alwayswantsinputlist` set to `true`, plugin's name will appear in list of avalilable *Traces*/*Inputs* in traces dialog window, even it does not have any *Traces* or *Inputs*. This allows user to select plugin and click to configuration button which will call `plugin_tracesconfigdialog()`.

## `TPluginMenuItem`

`TPluginMenuItem` is used with menu enumeration functions `plugin_getmenuitem()` and `plugin_getpopupitem()`.

```
#define mmFile 0
#define mmView 1
#define mmSettings 2
#define mmPluginSettings 3

typedef struct {
 int32 TopMenu; // mmFile..mmPluginSettings
 char* Caption;
 bool Checked;
 bool Enabled;
 char* ShurtCut;
 void(*Notify)(void); // use stdcall
} TPluginMenuItem;
```

`TPluginTraceSnapshot`

`TPluginTraceSnapshot` is used with trace data manipulation functions `plugin_gettrace()` and `gettrace()`.

When `TextCount!=0`, then `TextTexts`, `TextPositions` and `TextMaxWidths` are mandatory and must not be NULL. `TextFlags` should be ignored when set to NULL. Format of flags are not defined yet with only allowed value of zero which will preserve same behavior after flags will be defined.

`BoundaryMin` and `BoundaryMax` values are theoretical boundaries of this *Trace*. For digital *Trace* this is typically `BoundaryMin=0`, `BoundaryMax=1`.

```
typedef struct {
// --- filled by caller of 'gettrace' function ---
 uint32 StartTimeLo; // All times in PU
 int32 StartTimeHi;
 uint32 PixelTimeLo;
 int32 PixelTimeHi;
 int32 ViewWidth;
// --- filled by called function ---
// arrays below allocated by caller
 int32 BoundaryMin, int32 BoundaryMax; // minimum and maximum trace values
 int32 *MinArray, int32 *MaxArray; // array of values; array length is ViewWidth
 int32 TextCount; // number of texts – length of arrays below
// arrays below allocated by called (remains allocated until next call of any function)
 char **TextTexts; // array of char*
 int32 *TextPositions; // array center positions of texts, in pixels
 int32 *TextMaxWidths; // array of maximum widths of texts, in pixels
 uint32 *TextFlags; // array of flags; currently ignored, set to NULL
 bool BusStyle;
} TPluginTraceSnapshot;
```

`TPluginInputSnapshot`

`TPluginInputSnapshot` is simplified `TPluginTraceSnapshot`. It can be interpreted as `TPluginTraceSnapshot` with fields `TextCount` and further as reserved.

```
typedef struct {
// --- filled by caller of 'gettrace' function ---
 uint32 StartTimeLo; // All times in PU
 int32 StartTimeHi;
 uint32 PixelTimeLo;
 int32 PixelTimeHi;
 int32 ViewWidth;
// --- filled by called function ---
// arrays allocated by caller
 int32 BoundaryMin, int32 BoundaryMax; // minimum and maximum trace values
 int32 *MinArray, int32 *MaxArray; // array of values; array length is ViewWidth
} TPluginInputSnapshot;
```

## TPluginHighlightRange

`TPluginHighlightRange` is used with `sethighlightrange()` function.

```
#define hsBackground 0
#define hsSelection 1
#define hsLine 2
#define hsTopLine 3
#define hsText 4

typedef struct {
 int32 Tag; // allowed is -1 and 0x0000..0xFFFF
 int32 Style; // style hsBackground..hsText
 uint32 Color;
 int32 TraceNumber; // -1 means all
// SelectTime[0]=-1 means no (remove) selection
// SelectTime[1] is applicable only for hsBackground and hsLine
 struct { uint32 Lo; int32 Hi } SelectTime[2];
 int32 SelectSampleValue[2]; // applicable only if TraceNumber is not -1
 char *Text;
} TPluginHighlightRange;
```

*Highlights* with `Tag!=-1` are persistent. They must be removed with call where `SelectTime[0]=-1`. On other hand, highlights with `Tag==-1` can be called only from `plugin_paintnotify()` function and duration is only until next call of `plugin_paintnotify()`. Number of highlights with `Tag==-1` is unlimited.

- `hsBackground` is highlight box, which is below traces and other highlights.
- `hsSelection` highlight is box, which is above traces and other highlights, drawed using pen style xor.
- `hsLine` is vertical line, which is under traces. Example is trigger position.
- `hsTopLine` is vertical line, which is above traces. Example is cursor position.
- `hsText` is vertical line with text box next to time ruler. Example are bookmarks.

## TNewTraceInfo

`TNewTraceInfo` structure is used with message `WM_UPDATEVIEWER`, `wParam=4`, add new trace.

```
#define ttBus 1
#define ttInput 2
#define ttTrace 4

typedef struct {
 char *Caption;
 int32 TraceType; // ttBus, ttInput or ttTrace
 int32 InputCount;
 int32 Inputs[]; // InputCount integers – input ids
} TNewTraceInfo;
```

- When `TraceType==ttBus`, `Inputs` is array of ***input** ids*, of chich is consisted the new bus.
- When `TraceType==ttInput`, `Inputs` has only one field with ***input** id*.
- When `TraceType==ttTrace`, `Inputs` has only one field with ***trace** id*.

## `TPluginRawInputField`

`TPluginRawInputField` structure is used with `getanalyzerraw()` function.

```
typedef struct {
 uint32 TimeLo; // time in PU
 int32 TimeHi;
 int32 min, max;
} TPluginRawInputField;
```

## `TPluginViewRange`

`TPluginViewRange` structure is used with `plugin_getpopupitem()`, `plugin_paintnotify()` and `plugin_updatehint()` functions and carries current context.

```
typedef struct {
 uint32 StartTimeLo; // All times in PU
 int32 StartTimeHi;
 uint32 PixelTimeLo;
 int32 PixelTimeHi;
 int32 ViewWidth; // visible area of viewer window, in pixels
 POINT ViewerScreenPoint; // POINT is WINAPI defined structure
} TPluginViewRange;
```

# 5. Functions reference

Basic data types used in text below.

| name | width | meaning |
|---|---|---|
| int32 | 32 bits | signed integer |
| uint32 | 32 bits | unsigned integer |
| bool | 32 bits | 0 means false, other value means true |

## 5.1 Exported functions

Only one function is exported by plugin's DLL file.

---

bool plugin_plugin(settings *TPlugInSettings);

**Parameters:**
`settings` is structure containing basic global variables and function entry points. Some of them are prefilled by application, other must be filled by plugin itself. If some function is not implemented by plugin, its entry point may be left `NULL`. See `TPlugInSettings` structure for further information.

Data contained in the structure must be copied to plugin's local memory, because the structure is removed from the memory after function returns.

**Description:**
This function can be called multiple times.

This function is always called before everything else (even before plugin_init()).

**Return value:**
Plugin returns `true`. If `false` is returned, plugin is not encountered by application.

## 5.2 Plugin functions

These functions can be called by application. These functions are served by plugin.

These functions entry points are filled in `TPlugInSettings` structure during `plugin_plugin` call.

---

bool plugin_init(bool *allowStartup);

**Parameters:**
`allowStartup` is pointer to bool which plugin may fill with `false` value to prevent application from loading (in case of fatal hardware related error related to plugin, etc...). The value is pre-filled with `true`.

**Description:**
Plugin initialization is here. On other function (except plugin_plugin) is called before this.

**Return value:**
If initialization fails, plugin returns `false`, plugin is listed in application plugins list, but nothing can be done with it. Otherwise, plugin should return `true`.

---

void plugin_free();

**Parameters:**
No parameters.

**Description:**
Complementar to `plugin_init`(). Nothing more will be called after this.

**Return value:**
Nothing.

---

void plugin_savesettings();

**Parameters:**
No parameters.

**Description:**
This function notifies plugin that this is best time to store all settings using `setintvalue()` and `setstrvalue()` functions. This happens typically on application close, before file save etc...

**Return value:**
Nothing.

---

void plugin_loadsettings();

**Parameters:**
No parameters.

**Description:**
This function notifies plugin that this is best time to load all settings using `getintvalue()` and `getstrvalue()` functions. This happens typically on application load and after file load.

**Return value:**
Nothing.

---

void plugin_aboutdialog();

**Parameters:**
No parameters.

**Description:**
This function notifies plugin that user wants plugin's about dialog. This function can do as small as show dialog using *winapi* `MessageBox()`.

If entry point of this function is left `NULL`, graphic user controls to show about dialog are disabled (greyed).

**Return value:**
Nothing.

---

void plugin_configdialog();

**Parameters:**
No parameters.

**Description:**
This function notifies plugin that user wants plugin's configuration dialog.

If entry point of this function is left `NULL`, graphic user contols to show about dialog are disabled (greyed).

**Return value:**
Nothing.

void plugin_tracesconfigdialog();

**Parameters:**
No parameters.

**Description:**
This function is very similar to `plugin_configdialog()`, only difference is that this function is bound with user graphic controls in Traces Setup Dialog (Menu > Settings > Traces Setup...).
If entry point of this function is left `NULL`, graphic user contols to show about dialog are disabled (greyed).
There is no reason why `plugin_tracesconfigdialog()` and `plugin_configdialog()` functions cannot be the same.

**Return value:**
Nothing.

bool plugin_getmenuitem(int32 index, TMenuItem *item);

**Parameters:**
`index` is indexed from zero of menu item to be filled in structure `item`.
`item` has to be filled by the function.

**Description:**
If `index` is less than number of menu items owned by the plugin, function returns the menu item in the `item` structure and returns true, otherwise fill nothing and returns `false`.
If function entry point is left `NULL`, plugin will not have any menu items.

```
TMainMenu = (mmFile=0, mmView, mmSettings, mmPluginSettings);
TPluginMenuItem = packed record
                   MainMenu: TMainMenu;
                   Caption: PChar;    // allocated by plugin
                   Checked: LongBool;
                   Enabled: LongBool;
                   ShortCut: PChar;   // can be NULL
                   Notify: procedure; stdcall;
                  end;
```

**Return value:**
`true` if structure `item` is filled, otherwise `false`.

bool plugin_getpopupitem(int32 index, TMenuItem *item, TPluginViewRange *range,
int32 tracecount, TTraceInfo *traceinfos, POINT *mousepoint);

**Parameters:**
`index` is indexed from zero if menu item to be filled in structure `item`.
`item` has to be filled by the function.
`range` is range of active view area in the Viewer Window.
`tracecount` is total number of traces in the Viewer Window including trace which are not visible („behind corner“).
`traceinfos` is array of `TTraceInfo`, number of fields equals `tracecount`.
`mousepoint` is structure containing point where menu is popped up.

**Description:**
This function is called when user pops up a menu in Viewer Window to enumerate menu items to be displayed from this plugin. Number of menu items can differ from each user's click.
Context of user's click is in several parameters.
If function entry point is left `NULL`, plugin will not have any popup menu items.

**Return value:**
`true` if structure `item` is filled, otherwise `false`.

void plugin_paintnotify(TPluginViewRange *range, int32 tracecount, TTraceInfo *traceinfos);

**Parameters:**
`range` is range of active view area in the Viewer Window.
`tracecount` is total number of traces in the Viewer Window including trace which are not visible („behind corner“).
`traceinfos` is array of `TTraceInfo`, number of fields equals `tracecount`.

**Description:**
Function is called to notify plugin before Viewer Window painting about context including traces to be painted and visible range in the Viewer Window. List of traces include also non visible traces („behind corner“).
If function entry point is left `NULL`, plugin will not be notofied about paint.

**Return value:**
Nothing.

---

void plugin_updatehint(TPluginViewRange *range, int32 tracecount, TTraceInfo *traceinfos, int32 *maxchar, char *text);

**Parameters:**
`range` is range of active view area in the Viewer Window.
`tracecount` is total number of traces in the Viewer Window including trace which are not visible („behind corner“).
`traceinfos` is array of `TTraceInfo`, number of fields equals `tracecount`.
`maxchar` is maximum number of characters which can be stored into `text`. Allocated number of bytes in `text` is `maxchar`+1 (to include `NULL` char).
`text` is buffer to store plugin's hint terminated by `NULL` char.

**Description:**
Function store plugin's hint to buffer `text`. If hint is larger than `maxchar`, only first characters are stored and function updates `maxchar` value with needed amount of characters. Application will call function again with reallocated `text` buffer.
If plugin does not have hint, returned string must be empty (contain only `NULL` char).
If function entry point is left `NULL`, no hints are used at all.

**Return value:**
Nothing.

---

bool plugin_getinputinfo(int32 index, int32 *id, int32 *maxchar, char *name);

**Parameters:**
`index` is indexed from zero.
`id` is id of input of plugin indexed by `index`. Id is number in range `0x0000` and `0xFFFF`.
`maxchar` is maximum number of characters which can be stored into `name`. Allocated number of bytes in `name` is `maxchar`+1 (to include `NULL` char).
`name` is buffer to store name to be displayed to user in Traces Setup dialog.

**Description:**
This function returns plugin's available inputs. Inputs are indexed from zero. If `index` is equal or larger to number of available inputs from plugin, function should return `false` and fill nothing.

**Return value:**
`true` if `id`, `name` (and `maxchar`) are filled, otherwise `false`.

---

bool plugin_getinput(int32 id, TPluginInputSnapshot *snapshot);

**Parameters:**
`id` is id of requested input. It is `id` returned by `plugin_getinputinfo()`.

**Description:**
Application requests plugin's input identified by `id`. Min/max arrays in `snapshot` structure are allocated by application.

**Return value:**
`true` if snapshot is filled, otherwise `false`.
Return value `false` may not be considered as „fatal error“, only as state when is nothing to display, e.g. before first data capture.

---

bool plugin_gettraceinfo(int32 index, int32 *id, int32 *maxchar, char *name);

**Parameters:**
`index` is indexed from zero.
`id` is id of trace of plugin indexed by `index`. Id is number in range `0x0000` and `0xFFFF`.
`maxchar` is maximum number of characters which can be stored into `name`. Allocated number of bytes in `name` is `maxchar`+1 (to include `NULL` char).
`name` is buffer to store name to be displayed to user in Traces Setup dialog.

**Description:**
This function returns plugin's available traces. Traces are indexed from zero. If `index` is equal or larger to number of available traces from plugin, function should return `false` and fill nothing.

**Return value:**
`true` if `id`, `name` (and `maxchar`) are filled, otherwise `false`.

---

bool plugin_gettrace(int32 id, TPluginTraceSnapshot *snapshot);

**Parameters:**
`id` is id of requested trace. It is `id` returned by `plugin_gettraceinfo()`.

**Description:**
Application requests plugin's trace identified by `id`. Min/max arrays in `snapshot` structure are allocated by application. Other optional arrays (`Text`...) must be allocated by plugin. Plugin can free these arrays in next call of any `plugin_`... function.

**Return value:**
`true` if snapshot is filled, otherwise `false`.
Return value `false` may not be considered as „fatal error“, only as state when is nothing to display, e.g. before first data capture.

bool plugin_getnextinterest(int32 id, int64 *Time);

**Parameters:**
`id` is id of requested trace. It is `id` returned by `plugin_gettraceinfo()`.

**Description:**
This optional function is used for effective export of text files. Plugin can tell application there is no interesting data by skipping some `Time`. Used for example by protocol decoders in spaces between characters.

**Return value:**
`true` if new Time is filled, otherwise `false`.

## 5.3 Application callback functions

These functions can be called by plugin. These functions are served by application.

These functions can be called only from plugin from application main thread. Functions `plugin_`... are always called from there.

---

void sethighlightrange(TPluginHighlightRange *highlight);

**Parameters:**
`highlight` has to be filled by caller

**Description:**
Plugin calls this function to add or remove highlight in Viewer Window.
This function can be called asynchronously from any thread, or from function `plugin_paintnotify()`. When called asynchronously, `highlight->Tag` must not be -1.

**Return value:**
Nothing.

---

bool getintvalue(char *ident, int32 *value);

**Parameters:**
`ident` is identifier of value to get
`value` is pointer to location where to receive the value

**Description:**
Plugin calls this function to obtain integer value identified by string `ident`. Owner of the value (e.g. plugin itself) must register the value by `registerintvalue()` function.

**Return value:**
If value exists and value is integer type, return value is `true`.

---

bool getstrvalue(char *ident, int32 *maxlen, char *value);

**Parameters:**
`ident` is identifier of value to get
`maxlen` is number of allocated bytes for value
`value` is pointer to location where to receive the value; allocated by plugin

**Description:**
Plugin calls this function to obtain integer value identified by string `ident`. Owner of the value (e.g. plugin itself) must register the value by `registerstrvalue()` function. If number of allocated bytes are not enough, `maxlen` is filled with value of needed bytes, first bytes of value are filled, `NULL` char is always included (string is truncated), return value if `true`.

**Return value:**
If value exists and value is string type, return value is `true`.

bool setintvalue(char *ident, int32 value);

**Parameters:**
`ident` is identifier of value to set
`value` is to set

**Description:**
Plugin calls this function to set integer value identified by string `ident`. Owner of the value (e.g. plugin itself) must register the value by `registerintvalue()` function.

**Return value:**
If value exists and value is integer type, return value is `true`.

bool setstrvalue(ident: PChar; value: PChar);

**Parameters:**
`ident` is identifier of value to set
`value` is to set

**Description:**
Plugin calls this function to set string value identified by string `ident`. Owner of the value (e.g. plugin itself) must register the value by `registerstrvalue()` function.

**Return value:**
If value exists and value is string type, return value is `true`.

bool registerintvalue(char *ident, int32 default, bool cansave, bool loaddef, bool loadnow, bool filesave);

**Parameters:**
`ident` is identifier of value to register
`default` is default value (e.g. immediately after register, when value was not stored in file, after user's action load defaults)
`cansave` is wheather value is stored into registry
`loaddef` is wheather value is restored to default when user does action load defaults
`loadnow` is wheather value should be read from registry immediately
`filesave` is wheather value is stored into STF file

**Description:**
Plugin calls this function to register integer value identified by string `ident`.

**Return value:**
Return value is wheather the variable existed before call to this funtion.

---

bool registerstrvalue(ident: PChar; default: PChar; cansave, loaddef, loadnow, filesave: LongBool);
{ returns wheather value already existed }

**Parameters:**
`ident` is identifier of value to register
`default` is default value (e.g. immediately after register, when value was not stored in file, after user's action load defaults)
`cansave` is wheather value is stored into registry
`loaddef` is wheather value is restored to default when user does action load defaults
`loadnow` is wheather value should be read from registry immediately
`filesave` is wheather value is stored into STF file

**Description:**
Plugin calls this function to register string value identified by string `ident`.

**Return value:**
Return value is wheather the variable existed before call to this funtion.

---

bool gettrace(int32 index, TPluginTraceSnapshot *snapshot);

**Parameters:**
`index` is index of trace in Viewer Window
`snapshot` is pointer to TPluginTraceSnapshot structure. The structure is filled by application

**Description:**
Plugin calls this function to get trace data. `snapshot` structure is filled by application. All arrays and fields are allocated by application and remains allocated until next call to application or exit from plugin's function.

**Return value:**
Returned value is `true` if `snapshot` structure was filled. Return value `false` is not fatal error, it may indicate that no data was acquired yet.

---

bool gettraceinfo(int32 index, int32 *maxlen, char *name);

**Parameters:**
`index` is index of trace in Viewer Window
`maxlen` is pointer to number of allocated bytes for name
`value` is pointer to location where to receive the name; allocated by plugin

**Description:**
Plugin calls this function to obtain name of trace. If number of allocated bytes are not enough, `maxlen` is filled with value of needed bytes, first bytes of name are filled, `NULL` char is always included (string is truncated), return value is `true`.
If `index` is out of range, return value is `false`.

**Return value:**
If `index` is out of range, return value is `false`, otherwise `true`.

> To obtain number of traces, call `gettraceinfo()` multiple times with increasing number of `index`.

---

bool getinput(int32 id, TPluginInputSnapshot *snapshot);

**Parameters:**
`id` is id of input
`snapshot` is pointer to structure TPluginInputSnapshot. The structure is filled by application

**Description:**
Plugin calls this function to get trace data. `snapshot` structure is filled by application. All arrays and fields are allocated by application and remains allocated until next call to application or exit from plugin's function.
`Id` consists of lower 16 bits and upper 15 bits (sign bit is excluded). Upper 15 bits determine source – each plugin has unique *id* occupied by upper 15 bits. Internal *inputs* (*SIGMA* itself, special inputs) has upper 15 bits 0x00000000. Logic analyzer's *inputs* are always *ids* in range 0x00000000 – 0x000000FF. *Ids* 0x00000100 – 0x0000FFFF are internal special.

**Return value:**
Returned value is `true` if `snapshot` structure was filled. Return value `false` is not fatal error, it may indicate that no data was acquired yet.

> Note: getinput() and getinputinfo() is supported only for internal inputs yet.

bool getinputinfo(int32 index, int32 *maxlen, char *value, int32 *id);

**Parameters:**
`index` is index of input in the system; index values are consecutive from zero
`maxlen` is pointer to number of allocated bytes for name
`value` is pointer to location where to receive the name; allocated by plugin
`id` is pointer to *id* of that *input*. use this *id* in calls to `getinput()` function.

**Description:**
Plugin calls this function to obtain name and *ids* of input. If number of allocated bytes are not enough, `maxlen` is filled with value of needed bytes, first bytes of name are filled, `NULL` char is always included (string is truncated), return value is `true`.
If `index` is out of range, return value is `false`.

**Return value:**
If `index` is out of range, return value is `false`, otherwise `true`.

To obtain number of inputs, call `getinputinfo()` multiple times with increasing number of `index`.

bool getanalyzerraw(int32 id, int32 start, int32* len, TPluginRawInputField *rawinput);

**Parameters:**
`id` is *id* of input
`start` is first sample to get
`len` is number of samples to get
`rawinput` is pointer to array of TPluginRawInputField; allocated by caller (plugin)

**Description:**
Plugin calls this function to obtain *raw samples* of analyzer. This method is much faster than calling `getinput()`, but is dependent on analyzer. Currently, this function is supported only with *SIGMA* logic analyzer.
`len` is adjusted down if reading beyond end.

**Return value:**
If `len` is at least one, return value is `true`.

---

void getanalyzerrawlength(int32 *len);

**Parameters:**
len is pointer to location where to receive length

**Description:**
Plugin calls this function to obtain number of *raw samples* of acquired test of analyzer, needed to know boundary of `getanalyzerraw()` function.

**Return value:**
none

---

bool sigma_readregister(int32 regaddr, int32 *regdata);

**Parameters:**
`regaddr` is address of register. Allowed range is 16 – 255.
`regdata` is pointer where to receive value of that register

**Description:**
Call this function to read user defined registers in *SIGMA*.

**Return value:**
When register was read, return value is `true`.

---

bool sigma_writeregister(int32 regaddr, int32 regdata);

**Parameters:**
`regaddr` is address of register. Allowed range is 16 – 255.
`regdata` is value of that register. Allowed range is 0 – 255.

**Description:**
Call this function to write user defined registers in *SIGMA*.

**Return value:**
When register was written, return value is `true`.

bool sigma_configure(void *configuration, int32 configurationlength);

**Parameters:**
`configuration` is pointer to location of configuration bitstream in memory
`configurationlength` is length of that bitstream

**Description:**
For information on format of configuration bitstream, contact ASIX at support@asix.net.
Providing bad bitstream can cause stuck of USB communication without ability to reset *SIGMA* and send another configuration. In this case, *SIGMA* needs to be pulled out and in USB to be resetted manually.
Bad bitstream can also cause malfunction of hardware and result in its damage.
Use with caution!

**Return value:**
When *configuration* was accepted, return value is `true`.

## 6. File resources

Header file `pluginsIncU.pas` is included together with some plugin example files in *SIGMA* software installation package. These files are installed when source files option is checked to be installed. Files are installed into directory <*installation_dir*>`\src`.

## 7. Document revision history

| **Version** | **When** | **What** |
|---|---|---|
| 1.0 | 23.7.2008 | First official release |
| 1.1 | 18.9.2009 | Added new *WM_BROADCASTEVENT* messages |
| 1.2 | 12.12.2011 | Corrected plugin_init function<br>New function plugin_getnextinterest<br>Update of PicoUnitsInNs |