How to use a DLL to process Trace Data from the CombiProbe

The Trace32 GUI allows to load customer provided DLLs to process Trace Data received by the CombiProbe. The DLL can either forward the Trace Data to other customer implemented software or it can feed back data to the Trace32 GUI for displaying.

Additionally the CombiProbe supports to export Trace Data to the Host (the PC) on the fly, that means simultaneously while data is recorded. The DLL can be also used to process the data on the fly or to forward it to customer implemented software on the fly.

A short introduction to DLLs (Dynamic Link Libraries)

"DLL" is the abbreviation for "Dynamic Link Library". "Library" in this context means a collection of functions, which can be used by another software. "Dynamic Link" means, that the Library is loaded on demand. A Custom Trace DLL provides a collection of functions for the Trace32 Software, which can either export the Trace Data received by the CombiProbe to somewhere else or feed back the processed data to the Trace32 Software for displaying.

Since the DLL is a freely modifiable piece of code, a customer can also use a DLL to export the data in any way which is supported by the Operating System.

Some examples to export Trace Data to another customer implemented software are

- Preprocess the Trace Data and store the Trace Data into a file.
- Send the Trace Data to a UDP socket.
- Send the Trace Data via a TCP connection.
- Send the Trace Data via a Named Pipe.
- Copy Trace Data into Shared Memory.
- ...

The motivation for using a customer implemented DLL is to be absolutely flexible in the way the data is actually send to a customer specific application. Because the customer can implement the DLL in any way, any kind of inter process communication can be used.

Additionally instead of exporting the data to another piece of customer software, the data can be directly logged back into the Trace32 Software, which gives you the means to implement completely customer defined traces for Trace32.

How to load a DLL for Custom Trace Processing

To load a Custom Trace DLL, use the following Trace32 command:

```
CAnalyzer.CustomTraceLoad "label" <DLL Name>
        <Command Line Options>
```

The command line options are passed to the DLL and are intended to be used by the DLL for configuration.

Trace32 can load up to 8 CustomTrace DLLs simultaneously in total and the same DLL can be loaded more than once.

After loading a DLL, you can execute commands related to the DLL with

`CAnalyzer.CustomTrace.<label>.<command>`

The `<label>` has to be the same label as given in the `CAnalyzer.CustomTraceLoad` command.

If you call the `CAnalyzer.CustomTraceLoad` command with an empty label (“”), all loaded DLLs will be unloaded.

To unload a specific DLL, you can use the `CAnalyzer.CustomTrace.<label>.unload` command.

Displaying data logged by the DLL

If the DLL logs data to Trace32, then this data can be displayed with the commands

```
CAnalyzer.CustomTrace.<label>.List
CAnalyzer.CustomTrace.<label>.ListString
```

The List command will display cycles logged by the DLL, the ListString command will display ASCII strings, logged by the DLL.

Trace32 API for a Custom Trace DLL

The “pipeproto.h” C header file defines the API for the Custom Trace DLL.

The “pipeproto.c” C file implements the API.

When the DLL is loaded the Trace32 Software will call the function `PIPE_Interface` (implemented in “pipeproto.c”), which in turn will call `PIPE_Init`, which has to be implemented inside the DLL. The DLL cannot be linked without changing “pipeproto.c” if `PIPE_Init` is not implemented in the DLL.

The `PIPE_Init` function should process the command line options and initialize DLL specific control structures. Additionally the `PIPE_Init` function uses the API to register DLL specific functions with the Trace32 Software. These callback functions are then later called by Trace32 to

process Trace Data.

The following functions are provided by the API implemented in “pipeproto.c” to be called by the DLL code:

- `PIPE_RegisterCallback`
  Call this function to register a callback function with Trace32.
- `PIPE_Puts`
  Call this function to print a message to the Area Window of Trace32.
- `PIPE_Printf`
  Call this function to print a message to the Area Window of Trace32.
  Similar to PIPE_Puts, but with the syntax of a regular printf.
- `PIPE_Command`
  Call this function to execute an arbitrary Trace32 command line command.
- `PIPE_UartWrite`
  Call this function to send a block of data via the CombiProbe UART.
- `PIPE_LogCycle`
  Call this function to log a (bus) cycle for display in the Trace32 GUI.
- `PIPE_LogCustom`
  Call this function to log data in a custom format for later display in the Trace32 GUI.

The DLL can register five different callback functions:

- A processing function. This callback function will be called by Trace32 each time a Trace Message (STP Message or ITM Message) is received. This callback function then can export the message by any means, just throw the message away or it can use the `PIPE_LogCycle` or `PIPE_LogCustom` functions to feed back the decoded data to Trace32.
- An exit function. This callback function will be called by Trace32 when the DLL is unloaded. This function should close all connections and files and free all memory used by the DLL.
- A task function. This callback function will be called by Trace32 periodically. The task function can be used to implement server like functionality, for example to implement a server which can be reached via TCP.
- A command function. This callback function will be called by Trace32, when a user executes a CAnalyzer.`CustomTrace.<label>.COMMAND <Command Line Options>` command.
- A display function. This display function will be called by Trace32, when Trace32 wants to display custom data which was logged with `PIPE_LogCustom`. The display function has to convert the custom data into a format which Trace32 understands.

PIPE_Init

This function has to be defined inside the DLL code. It is called by Trace32 when the DLL is loaded.

Detailed definition:

```
int PIPE_Init(SPipeProtoInfo *info, int argc, char **argv);
```

Parameters:

`info`	This is a pointer to an internal communication structure which is used to call API functions. This pointer has to be passed to all API functions as first parameter.
`argc`	Number of command line options.
`argv`	Pointer to array of command line options. The first command line option contains the name of the DLL.

Return Value:

The function has to return zero if initialization was successful. If a non zero value is returned, Trace32 will report an error and unload the DLL.

Note: The unloading of the DLL due to an error will **not** call potentially registered exit callback function.

PIPE_RegisterCallback

Call this function to register a callback function.

Detailed definition:

```
int PIPE_RegisterCallback(
   SPipeProtoInfo *info,int type,void *callback,
   void *localdata,unsigned int flags);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`type`	Type of callback function which is registered.
`callback`	Pointer to callback function.
`localdata`	Pointer to DLL specific data. Each time the callback function is called by Trace32, this pointer will be passed to the callback function as parameter.
`flags`	Optional flags for callback function. Currently this parameter is not used, so set it to zero.

Return value:

If the call is successful, the function will return zero. If an error happened a value different from zero will be returned.

The following callback function types are defined:

- PIPE_PROCESS_CALLBACK
- PIPE_EXIT_CALLBACK
- PIPE_TASK_CALLBACK
- PIPE_CMD_CALLBACK
- PIPE_DISPLAYCUSTOM_CALLBACK

Note: A DLL should never use static variables, because the DLL can be loaded several times. In this case all "instances" of the DLL might share the static variables. Instead allocate memory with `malloc` and store DLL specific data inside this allocated memory. Let Trace32 pass this DLL specific data to your callback functions by using the `localdata` parameter when calling `PIPE_RegisterCallback`.

PIPE_Puts

Call this function to print a message to the Trace32 Area Window.

Detailed definition:

```
void PIPE_Puts(SPipeProtoInfo *info,const char *str);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`str`	Needs to point to a null terminated C string. This string will be printed.

Note: Trace32 will automatically print a newline after the string. So the string to be printed should not include a newline.

PIPE_Printf

Call this function to print a formatted message to the Trace32 Area Window.

Detailed definition:

```
void PIPE_Printf(SPipeProtoInfo *info,const char *format,... );
```

Parameters:

`info`	Needs to point to the internal communication structure.
`format`	Needs to point to the null terminated format string. Has the same meaning as the format string in a regular printf call.
`...`	Optional parameters to be printed as in a regular printf call.

Note: Trace32 will automatically print a newline after the string. So the format string should not include a newline.

PIPE_Command

Call this function to execute an arbitrary Trace32 command line command.

Detailed definition:

```
int PIPE_Command(SPipeProtoInfo *info,const char *str);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`str`	Needs to point to a null terminated string, which contains the Trace32 command line command.

Return value:

If the command is executed successfully, the function will return zero. If an error happened a value different from zero will be returned.

Do not call a Trace32 command that would result in the DLL being unloaded. For example avoid calling `CAnalyzer.CustomTraceLoad ""`.

PIPE_UartWrite

Call this function to send a block of bytes via the CombiProbe UART.

Detailed definition:

```
int PIPE_UartWrite(
   SPipeProtoInfo *info,const unsigned char * buf,int len);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`buf`	Needs to point to a block of bytes to be sent via the CombiProbe UART.
`len`	Number of bytes to sent via the CombiProbe UART.

Return value:

The function will return the number of bytes written to the UART.

PIPE_LogCycle

Call this function to log a bus cycle for display in the Trace32 GUI.

Detailed definition:

```
void PIPE_LogCycle(
   SPipeProtoInfo *info,const unsigned char *ts,
   const SPipeProtoCycleInfo *cycle);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`ts`	64 bit (that is 8 bytes) timestamp. The timestamp should have the same format as the first 8 byte of the buffer passed to the process callback; see also process callback function.
`cycle`	Pointer to structure which describes the cycle, which should be logged.

The SPipeProtoCycleInfo structure is defined in the “pipeproto.h” file as:

```
typedef struct {
  int type;
  unsigned int address;
  unsigned int spaceid;
  int datasize;
  unsigned char data[8];
} SPipeProtoCycleInfo;
```

You have to fill in at least the type field, before calling `PIPE_LogCycle`. The following types are allowed: `PIPE_LOG_TYPE_PROGRAM`, `PIPE_LOG_TYPE_WRITE`, `PIPE_LOG_TYPE_READ`.

Program cycles must have a valid `address` field.

For Read and Write cycles you have to logical OR the `PIPE_LOG_TYPE_WITHADDRESS` and/or the `PIPE_LOG_TYPE_WITHDATA` bits to the type to indicate if the cycle contains information about address and/or data.

The `address` field optionally defines the address of the cycle.

The `spaceid` field can be used in systems with MMU to describe, which virtual to physical translation table must be used. You have to logical OR the `PIPE_LOG_TYPE_WITHSPACEID` bit to the type to indicate that the cycle has a spaceid.

The `datasize` field optionally describes how many bytes of data were transmitted in the cycle. Valid values are 1 (8 bit), 2 (16 bit), 4 (32 bit) and 8 (64 bit).

The `data` field defines the data which was transmitted in the cycle and needs to store the data in little-endian format.

PIPE_LogCustom

Call this function to log custom data for later display in the Trace32 GUI.

Detailed definition:

```
void PIPE_LogCustom(
   SPipeProtoInfo *info,const unsigned char *ts,
   int type, const unsigned char *buf, int len);
```

Parameters:

`info`	Needs to point to the internal communication structure.
`ts`	64 bit (that is 8 bytes) timestamp. The timestamp should have the same format as the first 8 byte of the buffer passed to the process callback; see also process callback function.
`type`	Type of data. The following four types are allowed: `PIPE_LOG_TYPE_STRING`, `PIPE_LOG_TYPE_CUSTOM`, `PIPE_LOG_TYPE_CUSTOMSTRING`, `PIPE_LOG_TYPE_CUSTOMCYCLE`
`buf`	data to log.
`len`	length of data to log in bytes.

Calling this function will log the data provided in the buf array for later display. Data which was logged with the type `PIPE_LOG_TYPE_STRING` will be displayed by Trace32 as ASCII strings.

All other types of custom data, require that you implement a display callback function, which is responsible to translate the custom data into a format which Trace32 can display. See the description about the display callback function.

Process Callback Function

The DLL should register a Process Callback Function. Trace32 will call this function each time a Trace Message is received (either a STP Message or an ITM Message).

The Process Callback Function must adhere to the following function definition:

```
void process(
   SPipeProtoInfo *info,localDataP data,
   unsigned char *buffer,int len);
```

Paremeters:

`info`	Points to the internal communication structure. Pass this pointer when calling an API function.
`localdata`	Points to the data which was passed to `PIPE_RegisterCallback` when this Process Callback Function was registered.

`buffer`	Points to the byte buffer containing the received Trace Message. `buffer` may be a NULL pointer: This is an indication to flush all data buffered inside the DLL to make sure that all buffered data is pushed to the applications which might be connected to the DLL.
`len`	Defines the length of the received Trace Message if `buffer` is not a NULL pointer. If `buffer` is a NULL pointer, `len` encodes what to do: If `len` equals 0, flush all outstanding data. If `len` equals -1, re-initialize your DLL internal data structures used for processing the trace data.

The exact format of the byte buffer depends on the type of trace which is processed.

For an STP trace the buffer has the following format:

Byte 0-7	Contains 64 bit absolute Timestamp in little-endian order. Scale: 0x100 = 256 = 20 ns. Example: 0x12345678 means 305419896/256*20ns = 23860929,375 ns
Byte 8	Type of message. See source code examples.
Byte 9	STP Master which produced this message.
Byte 10	STP Channel which produced this message.
Byte 11...	Payload of message (length is dependent on type of message).

For an ITM trace the buffer has the following format:

Byte 0-7	Contains 64 bit absolute Timestamp in little-endian order. Scale: 0x100 = 256 = 20 ns
Byte 8	Type of message. See source code examples.
Byte 10	ITM Channel which produced this message.
Byte 11...	Payload of message (length is dependent on type of message).

Display Callback function

The DLL uses PIPE_LogCustom to log custom data, the DLL also should register a Display Callback Function. Trace32 will call this function each time wants to display or interpret custom data.

The display callback function must adhere to the following interface.

```
void display(
   SPipeProtoInfo *info,localDataP data,
   unsigned char *buffer,int len,
   SPipeProtoCustomInfo *disp);
```

Parameters:

`info`	Points to the internal communication structure. Pass this pointer when calling an API function.
`localdata`	Points to the data which was passed to `PIPE_RegisterCallback` when the Display Callback Function was registered.
`buffer`	Points to the byte buffer containing the custom data.
`len`	Defines the length of custom data in bytes.
`disp`	Structure which needs to be filled in by the display callback function.

The display callback function must process the custom data and fill in the `disp` structure, which is then used by Trace32 to display the data. When Trace32 calls the display callback function, it sets the type field of the `disp` structure to indicate to the display callback function what kind of data is needed.

If the type field is set to `PIPE_LOG_TYPE_CUSTOMSTRING` Trace32 expects that the display callback function converts the custom data into an ASCII string. The display callback function must set the `type` field to `PIPE_LOG_TYPE_STRING` to indicate a successful conversion.

If the type fiels is set to `PIPE_LOG_TYPE_CUSTOMCYCLE` Trace32 expects that the display callback function converts the custom data into cycle information. The display callback function must set the type field to `PIPE_LOG_TYPE_CYCLE` to indicate a successful conversion.

Data which was logged as `PIPE_LOG_TYPE_CUSTOM` will be later used as both `PIPE_LOG_TYPE_CUSTOMSTRING` and `PIPE_LOG_TYPE_CUSTOMCYCLE` depending on how Trace32 wants to display the data. The display callback function should be able to convert this kind of data into both: cycle information and ASCII strings.

Please have a look at the source code examples to see how to fill in the `disp` structure.

Exit Callback Function

If the DLL registers an Exit Callback Function, Trace32 will call this function when the DLL is unloaded.

The Exit Callback Function must adhere to the following function definition:

```
void exit_dll(SPipeProtoInfo *info, localDataP data);
```

Parameters:

`info` Points to the internal communication structure.

`data` Points to the data which was passed to PIPE_RegisterCallback when this Exit Callback Function was registered.

Task Callback Function

If the DLL registers a Task Callback Function, Trace32 will call this function periodically while the DLL is loaded.

The Task Callback Function must adhere to the following function definition:

```
void task(SPipeProtoInfo *info,localDataP data);
```

Parameters:

`info` Points to the internal communication structure.

`data` Points to the data which was passed to `PIPE_RegisterCallback` when this Task Callback Function was registered.

By using a Task Callback Function the DLL can implement server like functionality.

The Task Callback Function acts like the infinite main processing loop of a server process.

Command Callback Functionality

If the DLL registers a Command Callback Function, Trace32 will call this function, when a user executes a

```
CAnalyzer.CustomTrace.<label>.COMMAND <Command Line Options>
```

command in the Trace32 command line or from a PRACTICE script.

The Command Callback Function must adhere to the following function definition:

```
void command( SPipeProtoInfo *info, localDataP data,
              int argc, char**argv);
```

Parameters:

`info`	Points to the internal communication structure.
`data`	Points to the data which was passed to PIPE_RegisterCallback when this Command Callback Function was registered.
`argc`	Number of command line options passed in the `argv` array.
`argv`	Array of null terminated strings, which contain the command line options.

By registering a Command Callback Function a DLL can implement user specific commands which can be called from the command line of Trace32.