LAUTERBACH
DEVELOPMENT TOOLS

Protocol Analyzer Application Note

MANUAL

Protocol Analyzer Application Note

Protocol Analyzer Application Note

Version 06-Jun-2024

Introduction

Typically the PowerProbe or PowerIntegrator is used to trace a target-specific interface which follows a certain protocol. This can be a serial channel, USB, CAN bus, memory busses for SDRAM or FLASH or any other protocol you can think of.

In many cases it is tedious to analyze the traced raw data manually. It is far more convenient to let the computer transform the traced raw data into a higher level of abstraction, for example into the display of transferred bytes, or even into a summary of sent/received packets.

The TRACE32 software offers an extremely flexible feature to support such a protocol analysis: There is an open programming interface which allows to transfer the raw trace data to an user made analysis software (DLL) which returns the results to the TRACE32 software. The TRACE32 software can display the results in a separate window and it is easy to switch between different levels of abstraction.

Furthermore for memory busses the returned (calculated) buscycles (address bus, data bus, cycle type) can be combined with the application source code to show the program flow on ASM or HLL basis.

The analysis of protocols like I2C, JTAG, CAN or USB is already part of the TRACE32 software.

The analysis of other protocols have to be implemented by the user.

This document describes the principles how to implement a user specific protocol analysis.

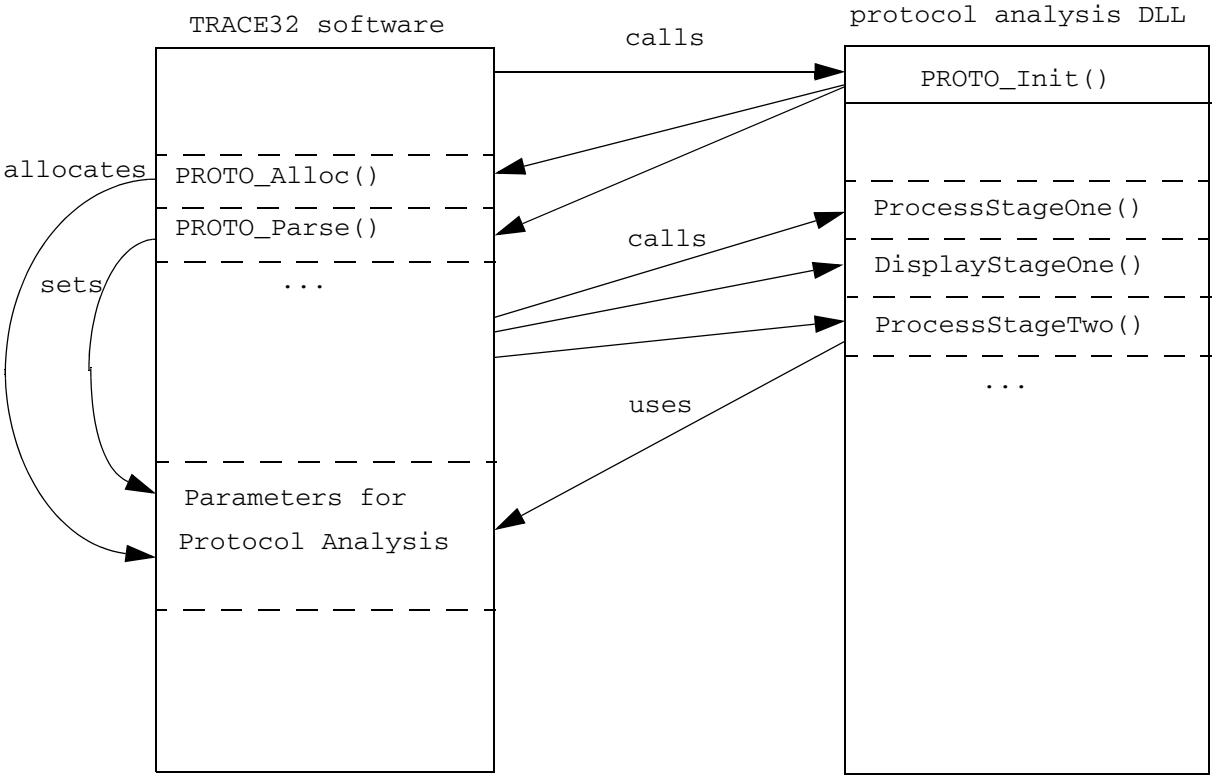
A short Introduction to DLLs

To develop a protocol analysis DLL, you have to be familiar with the programming language C.

The most convenient way to produce a DLL is to use Microsoft's Visual Studio and the Makefile provided in the example section. You can also use the free GNU tools for building the DLL. (If you are using Cygwin, ensure that the Mingw32 packages are installed.)

You must have installed the TRACE32 software to use the DLL.

"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 protocol analysis DLL provides a collection of functions for the TRACE32 software, which analyze the raw trace data of the TRACE32 software and which display the analyzed data. "Dynamic Link" means that the TRACE32 software loads the library only when you want to use the protocol analysis functions of the library.



When a protocol analysis DLL is loaded by the TRACE32 software, the TRACE32 software only knows that the DLL contains the PROTO_Init function. The PROTO_Init function has to tell the TRACE32 software about the other analysis functions, which are provided by the DLL. These functions are called “callback functions”, because PROTO_Init tells the TRACE32 software about the availability of the “callback functions”, and the TRACE32 software calls them back later.

PROTO_Init will be called each time a protocol analysis window is opened. So each protocol analysis window has its own set of callback functions.

If the analysis functions need additional parameters, these parameters have to be stored somewhere. The parameters can’t be stored at a fixed position, because it is possible to open several protocol analysis windows, which use the same protocol analysis functions, but have different parameter sets. So each protocol analysis window has to have its own parameter set.

If the analysis functions need parameters, the PROTO_Init function calls a function in the TRACE32 software to allocate memory for one parameter set. The next step is to call functions in the TRACE32 software to read in the values of the different parameters. The last step is to tell the TRACE32 software about the callback functions and also which parameter set the callback functions require.

The user specific DLL has to include the following three functions minimum.

PROTO_Init	• Define the required data channels needed for the analysis. • If required, allocate memory for parameters needed by the analysis. • If required, read additional parameters from command line. • Inform the TRACE32 software about all implemented callback functions in the DLL
ProcessCallBack-1	• Processes the raw trace data and generate a data array, which contains the extracted information.
DisplayCallBack-1	• Display the data array, which was generated by **ProcessCallBack-1**

Additional Process and Display callback functions can be implemented (up to four)

ProcessCallBack-2	• Processes the data array of **ProcessCallBack-1** and generates a second data array, which contains the extracted information.
DisplayCallBack-2	• Displays the data array, which was generated by **ProcessCallBack-2**
ProcessCallBack-x	• Processes the data array of **ProcessCallBack-(x-1)** and generates a data array, which contains the extracted information.
DisplayCallBack-x	• Displays the data array, which was generated by **ProcessCallBack-x**

This software structure allows to split up the analysis of trace data into several stages. For example: When analyzing USB, ProcessCallBack-1, extracts the transferred bytes out of the raw trace data. ProcessCallBack-2 uses the output of ProcessCallBack-1 to extract information about the USB frames.

PROTO_Init

PROTO_Init is the first function which is called by the TRACE32 software when entering the command:

Trace.Proto.* <protocol name>

Step 1: Definition of PROTO_Init:

```
int PROTO_Init(protoContext context, int command)

context  : Communication Structure for TRACE32 software
command  : Command to be executed (list/chart/statistic/find/export)
```

All callback functions in the DLL get the “context” as first parameter. The reason is that the callback functions need to call functions which are inside the TRACE32 software. The DLL can’t know where these functions reside inside the TRACE32 software. The “context” contains this information.

Step 2: It might be necessary to allocate memory for protocol specific parameters. Memory allocation for this parameter set is done with the function PROTO_Alloc.

```
void * PROTO_Alloc(protoContext context, int size);

context      : Communication Structure for TRACE32 software
size         : Byte-Size of the required memory
return       : Pointer to allocated memory
```

For Example: The protocol analysis for an asynchronous serial protocol requires three parameters: The baudrate, the number of bits and the parity (none, odd, even). So at the beginning of the source code of the DLL there is a definition of a structure which defines the layout of the memory which will hold the parameters:

```
typedef struct
{
     int baudrate;
     int bits;
     int parity;
} AsyncParameters;
```

The **PROTO_Init** function of the asynchronous protocol analysis DLL calls **PROTO_Alloc** to allocate memory for this Parameters:

```
AsyncParameters *params;
params = (AsyncParameters*)PROTO_Alloc(context,sizeof(AsyncParameters));
```

Step 3: All required trace data channels and parameters have to be read from the command line. This is done with the function **PROTO_Parse**:

```
void PROTO_Parse(protoContext context, protoPtr result, char * names,
                 int flags);

context      : Communication Structure for TRACE32 software
result       : Pointer to a variable, which will hold the read value
names        : String, which defines the parameter name or possible
               selections of a selection list
flags        : Defines the parameter type which should be read
```

First of all the user of the TRACE32 software has to define which physical trace channels correspond to the signals of the protocol that will be analyzed. The **PROTO_Init** function of the DLL calls **PROTO_Parse** for each protocol signal which must be defined by the user. The TRACE32 software displays the symbolic name of the signal in the state line of the TRACE32 window and the user has to enter the corresponding trace channel of the PowerProbe or PowerIntegrator.

For Example: The I2C Protocol has two physical signals called SDA and SCL (Serial Data and Serial Clock). So the **PROTO_Init** function which is responsible for the I2C Protocol calls **PROTO_Parse** two times:

```
PROTO_Parse(context, (protoPtr) 0, "<sda>", PROTO_PARSE_CHANNEL);
PROTO_Parse(context, (protoPtr) 0, "<scl>", PROTO_PARSE_CHANNEL);
```

For the definition of a protocol channel there is no result value, so the **PROTO_Parse** function is called with a null result pointer.

The TRACE32 software stores the definition of the protocol channels internally. Later when the **ProcessCallback-1** function needs the bit values of the protocol channels, the TRACE32 software will use this definition to store the bit values of the protocol channels in the order in which they were defined into an integer array. The first data channel which was defined with **PROTO_Parse** will be saved at index “0” of the integer array.

Optional channels can be defined by using the flag **PROTO_PARSE_OPTIONAL**. For Example the JTAG Protocol doesn’t always have a **TRST** signal. So **PROTO_Parse** is called in the following way:

```
PROTO_Parse(context, (protoPtr) 0, "<trst>",
     PROTO_PARSE_CHANNEL|PROTO_PARSE_OPTIONAL);
```

After all channel definitions were entered by the user, **PROTO_Parse** can be called to request further parameters from the user for the protocol analysis if required. The following examples show the different methods to request a parameter.

If the user has to select between different choices you call **PROTO_Parse** with the flag **PROTO_PARSE_SELECTION**. For example for an asynchronous serial protocol the user has to select the parity. The following code fragment uses the declaration of **params** in the **PROTO_Alloc** example:

```
int parity;
PROTO_Parse(context, (protoPtr)(&parity), "NONE,ODD,EVEN",
     PROTO_PARSE_SELECTION);
params->parity = parity;
```

The user has to enter “NONE”, “ODD” or “EVEN”. If the user chooses **NONE**, the TRACE32 software will store the value 0 (zero) into the variable **parity**. If the user chooses **ODD**, the TRACE32 software will store the value 1 into the variable **parity**, and so on. The value is then transferred into the allocated parameter memory (this is done in the last line of the above code fragment). Later the **ProcessCallback1** function will access the parameter memory and use the parity value to analyze the raw trace data.

If the user has to enter a frequency, you call **PROTO_Parse** with the flag **PROTO_PARSE_FREQUENCY**. In the asynchronous serial protocol example, the user has to enter the baudrate. The following code fragment prompts the user to enter the baudrate:

```
int baudrate;
PROTO_Parse(context, (protoPtr)(&baudrate), "<baudrate>",
     PROTO_PARSE_FREQUENCY);
params->baudrate = baudrate;
```

The TRACE32 software understands expressions like “1Hz”, “140Khz”, “14Mhz”. For example: if the user enters “14Mhz” the TRACE32 software will store the value 14000000 into the variable “baudrate”. As in the previous example, the last line of the code fragment transfers the entered value into the allocated parameter memory.

If the user has to enter a time value, you call **PROTO_Parse** with the flag **PROTO_PARSE_TIME**. For example the user could be asked to enter the bit-time in an asynchronous protocol, instead of the frequency. The following code fragment prompts the user to enter a time value:

```
protoTime bittime;
PROTO_Parse(context, (protoPtr)(&bittime), "<bittime>",
     PROTO_PARSE_TIME);
params->bittime = bittime;
```

As you can see in the first line of the code fragment, time values are stored in a special “protoTime” format. The header files for the DLL contain functions for manipulating time values (for example adding time values, comparing time values and so on.)

If the user has to enter a number, you call **PROTO_Parse** with the flag **PROTO_PARSE_INTEGER**:

```
int number;
PROTO_Parse(context, (protoPtr)(&number), "<baudrate>",
     PROTO_PARSE_INTEGER);
```

Step 4: The TRACE32 software has to be informed about the CallBackFunctions which exist in the DLL. This is done with the functions **PROTO_RegisterProcessCallback** and **PROTO_RegisterDisplayCallback**:

The declaration of **PROTO_RegisterProcessCallback** looks like this:

```
void PROTO_RegisterProcessCallback(protoContext context,
     int (PROTOAPI *callback)(…), protoPtr localdata, int size,
     int flags);

context      : Communication Structure for TRACE32 software
callback     : Pointer to the ProcessCallback function.
localdata    : Pointer to the allocated parameter memory of PROTO_Init
               (this pointer passed as a parameter to 'callback')
size         : Byte Size of one array element which is generated by the
               callback function
               (necessary to calculate the "arrayOutSize" of 'callback')
flags        : Level (1 - 5)
```

As an example, assume we use below definition of **stageOneEntry**, and we have a process callback function named **processStageOne**. The following code fragment in **PROTO_Init** would be used to tell the TRACE32 software about this function:

```
PROTO_RegisterProcessCallback(context, processStageOne, (protoPtr) 0,
     sizeof(stageOneEntry), 1);
```

In this example, the processStageOne function doesn't use any parameters, so we pass a null pointer ("**(ptoroPtr) 0**") as parameter memory.

The definition of PROTO_RegisterDisplayCallback is very similar:

```
void PROTO_RegisterDisplayCallback(protoContext context,
     void (PROTOAPI *callback)(…), protoPtr localdata, int flags);

context      : Communication Structure for TRACE32 software
callback     : Pointer to the DisplayCallback function.
localdata    : Pointer to the allocated parameter memory of PROTO_Init
               (is transferred to 'callback')
flags        : Level (1 - 5)
```

To analyze a protocol, the TRACE32 software will first call all process callback functions, which extract information out of the raw trace data. To display the extracted information in an analyzer.proto.list window, the TRACE32 software will call the display callback functions.

Step 5: Define default display level.

The protocol analysis support different display levels. With the function **PROTO_SetDefaultLevel** the user can define the default level when opening the protocol list window for the first time.

```
void PROTO_SetDefaultLevel(protoContext context, int level);

context      : Communication Structure for TRACE32 software
level        : default level number
```

Process Callback Functions

The processing of the raw trace data is done in several stages. Each stage is realized by one process callback function. The first process callback function (stage one) processes the raw trace data and extracts information which is stored in an array. The second process callback function (stage two) processes the output of the first process callback function and extracts information which is stored in a second array. The third process callback function (stage three) processes the output of the second process callback function, and so on.

Up to 5 stages are possible.

If there is only one process callback function, the processing is done in only one stage and only one array with extracted information is generated.

As mentioned a process callback function stores the extracted information in an array. Therefore for each stage you have to define the type of the array entries of this stage.

An array entry must be a structure in which the first member is a time value of type “protoTime”. This time value describes the point of time in the trace data to which the array entry refers. As an example: Assume that a stage one process callback function extracts 8 bit values out of the raw trace data (imagine that a simple serial protocol is analyzed, which transfers 8 Bit values.) In this case the type definition of a stage one array entry might look like this:

```
/* Define the array entry type of the
     stage one process callback function*/
typedef struct {
     protoTime timestamp;
     unsigned char dataByte;  /* this is the data which will be extracted
                                 out of the raw trace data */
} stageOneEntry;
```

Just as another example, assume that a second stage will generate 32-bit values out of four 8-bit values from the first stage. In this case the array entry type for the second stage might look like this:

```
/* Define the array entry type of the
     stage two process callback function*/
typedef struct {
     protoTime timestamp;
     unsigned int dataDword;  /* this is the data which will be extracted
                                out of the stage one data */
} stageTwoEntry;
```

All process callback functions have to use the same parameter interface. Lets assume we want to declare a stage one process callback function with the name “processStageOne”. The declaration would look like this:

```
static int processStageOne(protoContext context,
     protoPtr arrayOut, int arrayOutSize,
     protoPtr arrayIn,  int arrayInSize,
     protoPtr localdata)
```

The parameters have the following meaning:

context The communication structure, which is necessary to call functions of the TRACE32 software.

arrayOut This is a pointer to the array where the output data of the process callback function will be stored. This array is already allocated by the TRACE32 software.

arrayOutSize This parameter tells how many array entries the TRACE32 software allocated for output data. If the process callback function finds out during processing, that more entries are needed, it must abort and return the value **PROTO_PROCESS_OUTOFMEMORY**.

arrayIn This is a pointer to the array of the output data of the previous processing stage. If the process callback function processes raw trace data (because it's the stage one process callback function), this parameter is unused.

arrayInSize This parameter tells how many array entries the arrayIn array contains. For raw trace data this parameter tells how many raw records must be processed.

localdata This is a pointer to the parameter memory that was allocated in **PROTO_Init**. If no memory was allocated, because no additional parameters are needed, this parameter is unused.

Normally it is a good idea to define two local variables inside the process callback function which will be used to cast the "arrayIn" and "arrayOut" parameters to the correct type. For example this code fragment could be used in a stage two process callback function:

```
stageOneEntry   *stageOneArray; /* input data  */
stageTwoEntry   *stageTwoArray; /* output data */

stageOneArray = (stageOneEntry *)(arrayIn);
stageTwoArray = (stageTwoEntry *)(arrayOut);
```

Process callback functions for stage two and higher simply access the entries in the **arrayIn** array to process data from the previous stage.

If an entry is stored into the output data array, the timestamp of this entry (the first member in the structure of an output array entry) should be copied from the first input data entry which was analyzed to generate this output data entry. This way it is possible to align the processed data to the raw trace data via using the **TRACK** option of the analyzer.list and analyzer.proto.list windows.

Stage one process callback functions are special, because they process the raw trace data.

A stage one process callback function has to call the function **PROTO_ReadTrace** to access the raw trace data. The declaration of **PROTO_ReadTrace** looks like this:

```
extern void PROTO_ReadTrace(protoContext context,
     int index, protoTime * timestamp, int * data);
```

The parameters have the following meaning:

context	The communication structure, which is necessary to call functions of the TRACE32 software.
index	The number of the raw trace record you want to read out. The value of this parameter must be in the range 0 to (arrayInSize-1).
timestamp	The point of time in the trace data to which the record refers. This variable will be set by **PROTO_ReadTrace**.
data	A pointer to an integer array in which **PROTO_ReadTrace** will store the bit values of the protocol signals, which were defined in **PROTO_Init**.

PROTO_ReadTrace will store the values (0 or 1) of the protocol signals into the integer array **data**. The values will be stored in the order in which the protocol signals were defined. So **data[0]** will hold the value of the first protocol signal that was defined, **data[1]** to the second signal, and so on.

As a general rule, if you implement a stage one process callback function, you should declare an array as a local variable, which is able to hold the values of all protocol signals. For example: The I2C Protocol has two physical signals, namely SDA and SCL. So you might find the following code fragment in a stage one process function for an I2C analyzer:

```
unsigned int traceChannel[2];  /* Only 2 channels are defined for I2C */
/* Define two preprocessor macros to access the array */
#define SDA traceChannel[0]
#define SCL traceChannel[1]
```

All process callback functions should call **PROTO_Cancel** periodically to check if the user wants to abort the protocol analysis. The declaration of **PROTO_Cancel** looks like this:

```
extern int PROTOAPI PROTO_Cancel(protoContext context);
```

If **PROTO_Cancel** returns a non-zero value the process callback function should abort the processing and return **PROTO_PROCESS_CANCEL**.

If a process callback function finishes the processing, it **must** return the number of output array entries, which were stored in the **arrayOut** array.

Display Callback Function

The Display callback functions are called, when the analyzed data has to be displayed in the “analayzer.proto.list” window. The window can hide the data from lower stages. So for example if you have two stages, the window can hide the extracted data from stage one, and only display the data from stage two. This way the granularity of the display can be changed.

For each processing stage there has to be one display function. Each display function handles the display of one array entry of the data array, which holds the extracted data of the current stage.

So an array entry must contain all information which is necessary to display it.

All display callback functions use the same interface:

```
void displayCallback(protoContext context, protoPtr pdata,
     protoPtr localdata)

context      : Communication Structure for TRACE32 software
pdata        : Pointer to the array entry which should
               be displayed.
localdata    : the allocated memory of PROTO_Init which may hold
               parameters
```

To display the data, the display callback function can call the following functions which are part of the TRACE32 software:

```
void PROTO_Puts(protoContext context, const char *text);

context      : Communication Structure for TRACE32 software
text         : the string which will be displayed in the 'Trace.PROTO'
               window.
```

PROTO_Puts can be used to output descriptive text into the display window.

```
void PROTO_Printf(protoContext context, const char *format, …);

context      : Communication Structure for TRACE32 software
format       : format string similar to the standard C library 'printf'
```

This function is probably the easiest to use. It behaves like the printf of the standard C library.

```
void PROTO_Control(protoContext context, int controlcode);

context      : Communication Structure for TRACE32 software
controlcode  : Code for various text attributes (color…)
```

The output can be displayed in different color combinations. To change the current color you use the PROTO_Control function.

The following values for **controlcode** change the color or font of the output:

```
PROTO_ATTRIBUTE_NORM                normal color (black).

PROTO_ATTRIBUTE_LIGHT               bright color (for comments …)

PROTO_ATTRIBUTE_BOLD                bold text type

PROTO_ATTRIBUTE_ERROR               red text color for ERROR display

PROTO_ATTRIBUTE_ASCII               change to ASCII font
```

Additionally PROTO_Control supports some special operations, which can be executed with the following control code values:

```
PROTO_CONTROL_PRINTNIL              writes the character '\0'. (cannot be
                                    done with PROTO_Puts).

PROTO_CONTROL_LINEFEED              start a new line

PROTO_CONTROL_INDENT                Print spaces to indent line. Indent
                                    depends on processing stage

PROTO_CONTROL_LINETILLEND           draw a line till end of line
```

As an example of the usage of the different functions: Assume that we want to display a stage one array entry, which contains only one member, which is called **dataByte** (see above for the declaration of **stageOneEntry**). The following declaration could be used:

```
static void displayStageOne(protoContext context, protoPtr pdata,
     protoPtr localdata)
{
     stageOneEntry *entry;
     entry=(stageOneEntry *)pdata;

     PROTO_Control(context, PROTO_CONTROL_INDENT);
     PROTO_Control(context, PROTO_ATTRIBUTE_LIGHT);
     PROTO_Puts(context, "BYTE : ");
     PROTO_Control(context, PROTO_ATTRIBUTE_NORM);
     PROTO_Printf(context, "0x%02x ", entry->dataByte);
}
```

Draw Callback Function

The Draw callback functions are called, when the analyzed data has to be displayed as graphic in the “analayzer.proto.draw” window. Up to six graphs are supported and can be displayed in separate windows or in a single window.

Step 1: The TRACE32 software has to be informed about the presence of a DrawCallBack function. This is done in PROTO_Init with the function **PROTO_RegisterDrawCallback:**

```
void DrawCallback(protoContext context, protoPtr pdata,
     protoPtr localdata, int flags, int channels, int low, int high)

context      : Communication Structure for TRACE32 software
pdata        : Pointer to the array entry which should
               be displayed.
localdata    : The allocated memory of PROTO_Init which may hold
               parameters
flags        : Protocol level that calculates the data to be displayed
channels     : Number of channels to be displayed
low          : Min-value to be displayed
high         : Max-value to be displayed
```

Step 2: The graph values have to be calculated with the DrawCallback function. The calculation results are stored in the presult array (presult[0] for graph-0, presult[1] for graph-1, …). The return value of the DrawCallback function marks valid values of the presult array. All marked graphs are displayed.

Example:

- The current calculation gives valid results for graph-2 and graph-0. The return value is 0x05 then.
- The current calculation gives valid results for graph-1 and graph-0. The return value is 0x03 then.

The return value defines which graphs are displayed. A value of 0x3f causes the display of six graphs in a single window. A value of 0x01 causes the display of only one graph.

```
int DrawCallback(protoContext context, protoPtr pdata,
     protoPtr localdata, int* presult)

context      : Communication Structure for TRACE32 software
pdata        : Pointer to the array entry which should
               be displayed.
localdata    : the allocated memory of PROTO_Init which may hold
               parameters
presult      : pointer to result array which holds the graph values
return value : bit mask which marks valid values of presult
```