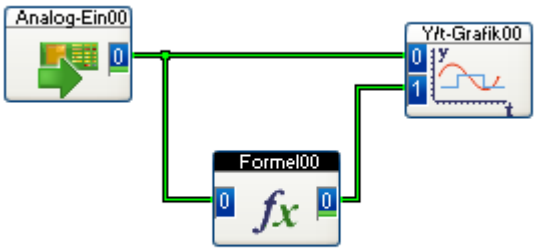
DASYLab®
Data Acquisition System Laboratory

Extension Toolkit

for DASYLab 2016



October 2016

NATIONAL INSTRUMENTS grants you a non-exclusive license to use this Software on one computer at a time. You do not obtain title to the Software or any copyrights or proprietary rights in the Licensed Software. You may not transfer, sub-license, rent, lease, convey, copy, modify, translate, convert to another programming language, decompile or disassemble the Licensed Software for any purpose, except as expressly provided for in this license. You may not copy the Documentation.

You may copy the licensed Software for backup purposes only, in support of your use of the Software in accordance with the terms and conditions of this license.

NATIONAL INSTRUMENTS warrants, for a period of ninety days after your receipt of the product, 1) the disks on which the Software is distributed to be free from defects in materials and workmanship, and 2) that the software will perform substantially in accordance with the Documentation. If the product fails to comply with the warranty set forth above, NATIONAL INSTRUMENTS's entire liability and your exclusive remedy will be replacement of the disks or NATIONAL INSTRUMENTS's reasonable effort to make the product meet the warranty set forth above. If NATIONAL INSTRUMENTS is unable to make the Product conform to the above warranty, then you may return the complete package to NATIONAL INSTRUMENTS or its dealer, and NATIONAL INSTRUMENTS, at its option, may refund all or a fair portion of the price you paid for this package.

Although NATIONAL INSTRUMENTS has tested the Software and reviewed the Documentation, NATIONAL INSTRUMENTS MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SOFTWARE OR DOCUMENTATION, THEIR QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE AND DOCUMENTATION ARE LICENSED "AS IS," AND YOU, THE LICENSEE, ARE ASSUMING THE ENTIRE RISK AS TO THEIR QUALITY AND PERFORMANCE.

IN NO EVENT WILL NATIONAL INSTRUMENTS BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE OR DOCUMENTATION, even if advised of the possibility of such damages. In particular, NATIONAL INSTRUMENTS shall have no liability for any programs or data stored or used with the Software, including the costs of recovering such programs or data.

English Language Support and Distribution:
Worldwide - Outside the Americas: measX GmbH & Co.KG, Trompeterallee 110, D–41189 Moenchengladbach, Germany
www.measx.com · info@measx.com
Americas: Measurement Computing Corporation, 16 Commerce Blvd, Middleboro, MA 02346 USA
www.mccdaq.com · info@mccdaq.com

Table of Contents

1 Introduction

The DASYLab Data Acquisition Laboratory

DASYLab is a Windows-based, interactive software for data acquisition and control. DASYLab 2016 is available for MS Windows 7, 8, 8.1 and 10.

DASYLab was designed for great flexibility by using object-oriented techniques. All of the data processing takes place in independent, encapsulated modules that can be freely connected together. New user designed modules can be easily added to this scheme.

The DASYLab Extension Toolkit

The Extension Toolkit described in this document allows the user and third parties to extend the capabilities of DASYLab by designing their own modules using the C programming language.

This documentation describes all of the internal DASYLab structures necessary to create own modules and explains how to couple them to DASYLab.

Several examples are included with this toolkit to serve as a starting point for your own modules.

Compatibility

The *Extension Toolkit for DASYLab 2016* works with DASYLab 2016 and coming versions. To build an extension module for DASYLab 13, you have to use the *Extension Toolkit 9*. There are no special toolkit versions for DASYLab 10 to 13. You can use the toolkit *Extension Toolkit 9* for these DASYLab versions, too.

2 What is new in Extension Toolkit for DASYLab 2016

For DASYLab 2016 the interface between the main program and the DLLs with the modules went through significant modifications/enhancements. As a result, DLLs created with the *Extension Toolkit 9* and earlier are not compatible with DASYLab 2016. They will not load intentionally - vice versa DLLs created with *Extension Toolkit for DASYLab 2016* will not load in DASYLab 13 and earlier.

Here are the main changes in this version:

- `DLAB_FLOAT` was changed from 4 byte `float` to 8 byte `double` to improve the accuracy of computation results and to minimize errors generated by accumulation of computing errors.
- In the structure `DATA_BLOCK_HEADER` the member `wBlocksize` was renamed to `uiBlockSize` and its data format was enlarged from `WORD` to `UINT`. Block sizes (number of samples in one block) can now be larger than 32768 Samples; see the declaration of `MAX_BLOCKSIZE` in `types.h`.
- In the structure `FIFO_HEADER` the member `wBlockSize` was renamed to `uiBlockSize` – the data format was already `UINT`.
- The structure `Sysinfo` was responsible for authentication of DLLs regarding flags in the serial number. It is renamed to `SerialOpt` and gained additional flags.
- The newly introduced message `DMM_GET_TIMEBASE_ID` supports the handling of unused time bases.
- For later DASYLab releases we provide a new API for data transport at start time (we call this static) and data transport with the data blocks at process time (we call that dynamic). A first version of this API is included in the current toolkit, but all functions (except one) are subject to change. To provide compatibility with later DASYLab releases (as far as we can look into the future) add a call of `EmemBlock_PROCESS_MsgCopyPlain` before using `ReleaseOutputBlock`.
- `ShowPipeStatus` and `AdvancePipeStatus` were removed, since DASYLab handles the animation of the data flow internally. `SetFontSmallProc` was removed, since deprecated from Windows 32 on. Function `CenterDialog` was removed, since handled in DASYLab internally. Some function declarations (that were never exported to this interface) for bitmap handling were also removed.

3 Installation

3.1 Hardware Requirements

To use DASYLab, you need an IBM compatible PC computer with x86 compatible CPU from 1 GHz upwards, a hard disk and at least 1 GB of RAM. This configuration is also adequate for extending DASYLab using the *Extensions Toolkit for DASYLab 2016*.

3.2 Software Requirements

The use of DASYLab is generally restricted to Windows 7, Windows 8/8.1, and Windows 10.

An ANSI-compatible C compiler with complete Windows development support is needed to develop extension modules for DASYLab. We suggest (and have tested) Microsoft Visual C++ 2013 or higher, but other compilers should work as well.
Note: Please ensure that all needed libraries for your module are available on the destination system – e.g. the C runtime library 8.0 (msvcr80.dll), if you have compiled with Visual Studio 2005.

DASYLab currently uses dialog boxes using CTRL3D style. We recommend using this style for any new development. Therefore the *Microsoft Application Studio* or *Borland Resource Workshop* will work best to design resources for DASYLab Extensions.

3.3 Required Knowledge

To extend DASYLab using this Extensions Toolkit we expect you to be an experienced C programmer. You should know the basics of using Windows and should have some experience in using DASYLab.

Experience in Windows programming is not necessary if you write standard modules. You will need to know Windows programming if you want to design your own visualization modules, write modules that directly access hardware, etc. The little Windows programming knowledge that is used for programming the parameter settings dialog box can be learned on the fly by looking at the examples included in this toolkit.

Experience in C++ programming is also unneeded since existing DASYLab Extensions Toolkits are based completely on ANSI-C.

You should have basic knowledge of digital signal processing and good knowledge of the problems you want to solve using your new module classes.

3.4 Installing the Extension Toolkit for DASYLab 2016

The toolkit comes with a standard setup routine. To install it, run the file `setup_DLEXTK_2016.exe` from the Windows Explorer.

The Extension Toolkit will install itself completely into one directory and its subdirectories. No drivers are installed in the Windows directory, nothing is added to other directories. To remove it, use Control Panel (Add/Remove 'Extension Toolkit for DASYLab 2016').

Installation will create a new program manager group with the readme icon.

The directory structure is as follows:

```
– DLAB_EXT_2016
      – Example
            – OBJ32
      – Shared
```

The path *Example* contains all sources for the examples:

- *.c, *.h, the bitmaps, resource file, icons, project files for Visual Studio

The path *Shared* contains:

- all toolkit headers
- all toolkit sources with help routines and functions to exchange data between DASYLab and the toolkit DLL

Once you have installed the *Extension Toolkit for DASYLab 2016* you can try out the demo examples. If you own MS Visual C++ 2013, all you need to do is to load the prepared project file provided with the toolkit. For other compilers you have to set up your own project files. Refer to page 108 for information on how to set up the project parameters.

4 Principles of Operation

4.1 General Overview

The *Extension Toolkit for DASYLab 2016* was designed to allow users and third parties to extend DASYLab by designing their own modules, or module classes (see below). There is no special restriction on what modules designed by this toolkit can do. In fact, all of DASYLab's internal module classes could be rewritten using the toolkit. So the programmer is given the full facilities.

On the other hand, the Extension Toolkit is restricted to designing new module classes. It is not possible to add other extensions – such as additional functionality of the main window function bar, or a different routing algorithm – to DASYLab using this toolkit. Currently, new modules and new hardware device drivers are the only extensions one can create to extent DASYLab's possibilities.

4.2 The Module Class Concept

In spite of being written completely in ANSI C, DASYLab internally is an object-oriented system. The modules communicate with each other and with the system, primarily using predefined messages. We distinguish the ***module class*** (that is the type of the module) from the ***module*** itself. If you look at the Generator module class, you may have several modules of that same class in one worksheet. Each module is called an ***instance*** of the module class Generator.

All instances of a module class share the same code to handle messages to the modules. They have different data areas to store their private data, such as parameter box settings or intermediate results.

One of the most important principles of object-oriented design is ***data encapsulation***: The private data of one module may be manipulated only by the module itself. No module may manipulate other module's data and the DASYLab kernel will not touch the private data of the individual modules.

4.3 Coupling new Module Classes to DASYLab

A set of new module classes usually get bound together into one DLL file. One DLL file may contain up to 50 different module classes. After telling the name of this DLL to DASYLab, DASYLab will load the DLL at each program start and will execute one initialization routine in the DLL.

Inside this initialization routine the DLL registers all new module classes it wants to define and possibly extends DASYLab's menu bar with the new modules. This is all there is to it – the new modules now are fully integrated into DASYLab.

By default, DASYLab tries to load the default extension DLL files `DLAB_UX1.DLL`, `DLAB_UX2.DLL`,… up to `DLAB_UX8.DLL`. But you may use any names you want by adding lines to the `DASYLAB.INI` file:

```
[Extend]
DLL1=MY_DLL_1.DLL
DLL2=YOUR_DLL.DLL
…
DLL8=OUR_DLL.DLL
```

The `DASYLAB.INI` file is located in the DASYLab document directory and can be manipulated using a text editor.

DASYLab can load up to eight DLL extensions at a time.

4.4 Functions a Module Class Must Provide for DASYLab

DASYLab will send messages to the individual modules to make them perform actions. These messages include:

- **`CreateModule`** message: The module will receive this message just after it has been created. It can set up its private data with default values, open visualization windows, etc. here.
- **`DeleteModule`** message: The module will receive this message just before it is deleted by the system. It should free any resources it has allocated.
- **`LoadModule`** message: The module will receive this message just after it has been loaded from a previously saved worksheet. It should check the loaded parameters, open visualization windows, etc. here.
- **`StartModule`** message: The module will receive this message just before the experiment is started.
- **`StopModule`** message: The module will receive this message just after the experiment was stopped.
- **`ProcessData`** message: The module will receive this message periodically while the experiment is running. It should check its inputs for incoming data, then process the data and put them to its output.
- **`ParameterSetup`** message: The module should show its parameter setup dialog.
- **`CreateReplace`** message: The module should publish its parameters for replacement.
- **`CheckReplace`** message: The module class should indicate whether it is able to replace a given module.
- **`ReplaceModule`** message: The module should setup its parameters according to a template created from the original module.

Each module class must provide its own code to perform specific actions upon receiving the individual messages. A default handler is provided for the case that one module class does not want or need to perform any action on a specific message.

Generally speaking, a module takes full control and full responsibility for its own private data. For example, it has to handle the parameter setup dialog box completely on its own.

On the other hand, a module does not have to take worry about how it is bound into a worksheet. No special handling is necessary if more than one other module is connected to its output, or if the module is running inside a black box. The inter-module communication is completely handled by the DASYLab kernel.

4.5 Functions Groups Provided by DASYLab

DASYLab provides some functions that can be used inside new module classes. These functions belong to the following groups:

- **Memory management:** Provides some additional consistency checks to detect badly behaving modules earlier.
- **FIFO handling:** Handles the data exchange between the modules.
- **General Utility functions:** Converting a value into a string, performing a basic FFT calculation, etc.
- **Dialog box handling:** Functions supporting the channel selection bar and hotkeys, e.g. F1, F7, and F8.
- **Internal Error Handling:** A module can raise an internal error whenever it gets into a suspicious situation.
- **GDI Stack:** Several modules can share the same GDI objects (fonts, brushes, pens, ...)
- **Global Variables/Strings:** Use and management of global variables and strings
- **Layout connections:** Display windows can support graphical outputs into the layout tool (VI-Tool)

The functions provided by DASYLab can be called from inside the routines handling the messages received by a module. For example, while performing a `ProcessData` message, a module typically calls a FIFO handling function to determine, if any data is waiting on its input side, then process the data, and finally call another FIFO handling function to put the data out to the output side.

Each module class has to provide its own code. It is not possible to call the code of existing DASYLab module classes as subroutines.

4.6 Module Types

Module classes can be divided into several groups which share similar data handling:

a) **Data source modules**: These modules "produce" data, they only have outputs, but do not have any inputs inside the worksheet. Examples of this type are the Software Generator module, the Analog Input module, the File Read module, and the RS232 Input module.

b) **Data processing modules:** These are modules that have inputs and outputs. Many of DASYLab's internal modules are of that type, examples include the Arithmetic module, the Relay module, and Trigger modules.

c) **Data sink modules:** These are modules that have inputs, but no outputs. Examples include visualization modules like the Y/t chart, the File Write module or the Analog Output module. Outputs can be added to some of these modules to chain the incoming data through to simplify the worksheet. For the purpose of this extensions toolkit, we will ignore the chain-through option and use the term data sink for these module classes.

d) **Special module classes:** like Black Box modules, that can produce any of the types mentioned before.

4.7 Things a module should not do

It is very important that no module breaks the rules of object-oriented programming. Most important is the **data encapsulation** that ensures that no module manipulates any other module's data. Ensuring that each module is a closed, self-contained unit is the necessary basis, to allow the user to freely connect modules in a worksheet.

To avoid conflicts with other modules, a module should follow the naming conventions listed below. It is **not** allowed to use DASYLab*'s* internal names for custom modules, to use reserved codes, or to create new module messages, etc.

A module must follow the general rules of Windows programming. For example, when an object allocated by the module is no longer needed, it must be freed by that module.

4.8 Compatibility with future versions of DASYLab

We will try to keep the modules compatible for future versions of DASYLab on a source code basis, that is: when there is need to add new functions to the DASYLab DLL interface, we will try first to keep the interface compatible to old existing DLL's, and, if that's not possible, to try to limit the effort to one recompilation of the modules (without having to change the source code).

If the modules want to take advantage of new facilities, they must eventually provide new functionality.

4.9 Alternatives to using the Extension Toolkit

In addition to the Extension Toolkit there are other ways to extend DASYLab: The Driver Toolkit for the design of device drivers, external utility programs can provide functionality that need not be included in a module, and one can control DASYLab via DDE, or exchange Data via DDE.

Starting with DASYLab 13, you can also use the new script module and the integrated Python Script language to create your own DASYLab modules. Details can be found in the DASYLab online documentation.

Note: Of course, you can use the Extension Toolkit to create device drivers, too. It is even the currently recommended way: The Driver Toolkit is the Extension Toolkit's predecessor. Using the Driver Toolkit you are limited to create drivers for DASYLab that take the place of the "Driver" modules (category "Input/Output") in competition to other "Driver Toolkit Drivers" – only one driver can be selected, only one type of hardware can be used to acquire data. The Extension Toolkit allows you to add additional groups to DASYLab's existing main categories (e.g. Statistics, Inputs/Outputs). It's up to you, if the added modules are just for calculation, or if their purpose is to communicate with diverse hardware devices.

5 A DASYLab Extension Example

In this chapter we present a complete example of a DLL containing some simple, new DASYLab modules: `Derivation, Demand Trigger, Lamp and Generator`.

The internal functions are very similar to the DASYLab's build in functions; in fact, the examples are based upon original DASYLab source code, with modifications to improve its use as an example.

We will not go into too much detail here, but explain the concepts and the general ideas behind the extension. The details can be found in the reference section following this section. Some comments, prototype definitions, etc. are also left out from the source code to make the examples easier to read. You can find the complete examples in the toolkit files.

A DLLs containing new modules for DASYLab consists of two parts:

1. Two global routines with **required**, **fixed** names which are called by DASYLab just after the DLL has been loaded and just before the DLL is going to get unloaded.
2. The code for each of the new module classes.

You will usually have the code for the global routines and for each of the module classes in separate source files. In addition, you need resource files defining the outfit of the dialog boxes, bitmaps, and the like.

5.1 General DLL Routines

The two functions a DLL must provide for DASYLab are called `Init_DLL` and `Exit_DLL`.

DASYLab will call the `Init_DLL` function immediately after the DLL has been loaded. This function may then add a menu to DASYLab's menu bar (containing all of the module classes defined in this DLL) and call `init` functions for all of the modules it wants to define. Each module class can alternatively integrate itself into the DASYLab menu tree and browser.

DASYLab will call the `Exit_DLL` function just before the DLL is to be unloaded. In most cases, the `Exit_DLL` function can be left empty, because the exit functions of the individual modules (which are called independently by DASYLab) will do all the work that is necessary.

Typically the two functions get bound together in one source file looking like this:

```
#include <WINDOWS.H>
#include "DLAB.H"
BOOL Init_DLL (void)
{
   /* Expand DASYLab's module bar for the new modules */
   ExpandModuleBar();

   /* Init all new Modules here */
   Init_DERIVATION ();
   Init_MORE ();
   Init_AND_MORE ();
   Init_AND_MORE_MODULES ();
   return TRUE;
}
BOOL Exit_DLL (void)
{
   return TRUE;
}
```

Calling the `INIT` function once for each module class is sufficient for the DLL. One function of the `INIT` function of the module class is to register itself with DASYLab as shown below.

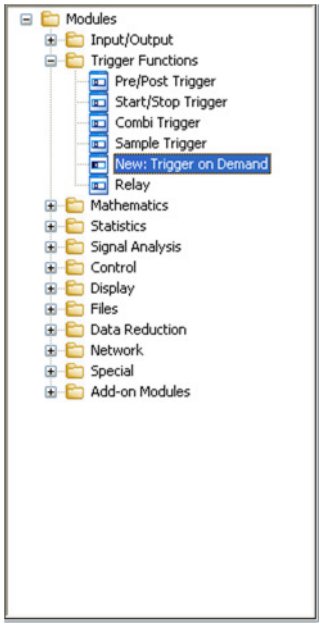
Inside the new submenu, the extension must always use the fixed range from 2950 to 2974 of module class menu IDs to distinguish the different module classes. DASYLab then maps this range onto a different range for internal use. So there is no problem when different extension DLL's use the same menu bar IDs.

5.2 DLL Menu Entries

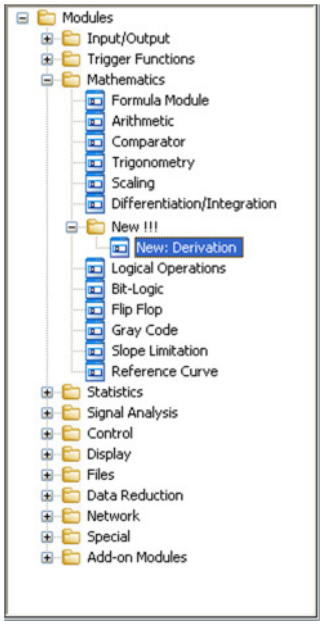
With DASYLab 9, the interface has been completely reworked. The DASYLab screen comprises the Module Browser that contains the **Modules** tab, the **Black-Box** tab, and the **Navigator** tab. DASYLab provides all available modules in a tree structure, on the **Module** tab. You can drag and drop the modules from the Browser into the worksheet. Integrate the modules that you create with the Extension Toolkit in the thematically related module group.

The following section describes the steps you must complete in your sources, to insert your Extension Toolkit modules and additional configuration dialog boxes in the menus of the browser. The description uses the source code examples of the Extension Toolkit to include the example modules **Trigger on Demand**, **Derivation**, **Generator, a**nd **Status lamp**

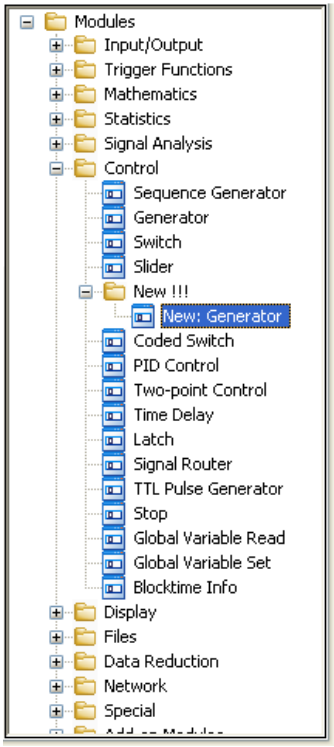
New: Trigger on Demand



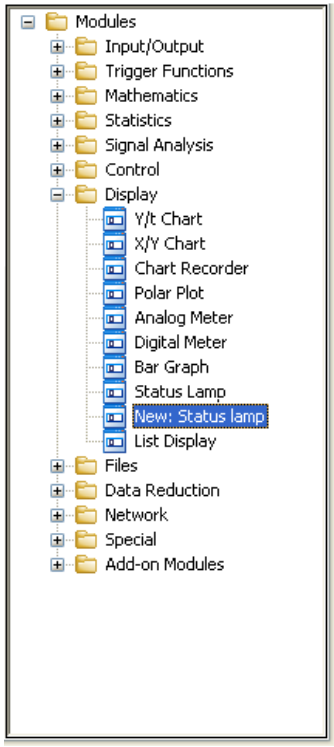
New: Derivation



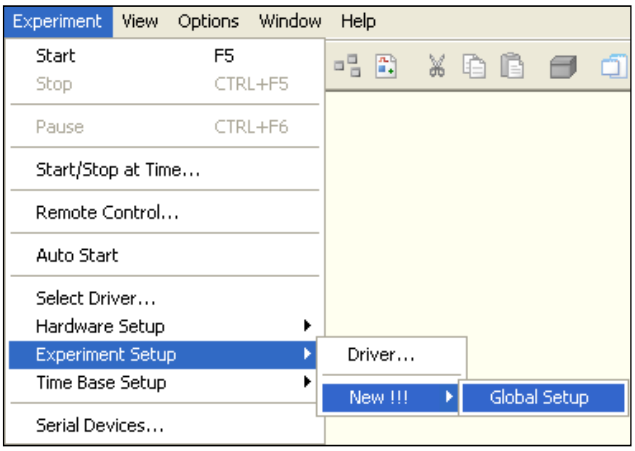
New: Generator



New: Status lamp



The examples also demonstrate how you can include the comprehensive configuration dialog box **Global Setup** in the menu: **Experiment»Experiment Setup»New !!! » Global Setup**.



The **Global Setup** dialog box is integrated in the `ExpandModuleBar` function in the file `DlabExtTkExample.c`. The function uses `szMainMenuEntry` to forward the main menu entry that contains the submenus you want to change to the `DASYLAB_INSERT_MENU` structure. The following entries in `szMainMenuEntry` are valid:

MOD	Submenu in the **Module** menu
ME	Submenu in the **Experiment** menu
HLP	Submenu in the **Help** menu

```
typedef struct
{
   UINT uiSize;
   UINT uiVersion;
   char szMainMenuEntry[MAIN_MENU_ENTRY_MAXLEN];
   char szSubMenusEntry[SUB_MENUS_ENTRY_MAXLEN];
   char szNewMenuEntries[SUB_MENUS_ENTRY_MAXLEN];
   char szMenuGroupEntries[MAIN_MENU_ENTRY_MAXLEN];
   UINT uiNewMenuID;
   char cMenuAccessKey;
} DASYLAB_INSERT_MENU;
```

Use `szSubMenusEntry` to specify into which submenu you want your module integrated.

The following submenus are defined for `"MOD"`:

IO	Modules » Input / Output
TRIG	Modules » Trigger Functions
MATH	Modules » Mathematics
STAT	Modules » Statistics
SIG	Modules » Signal Analysis
CONTR	Modules » Control
DISPL	Modules » Display
FILE	Modules » Files
REDUCT	Modules » Data Reduction
NET	Modules » Network
SPEC	Modules » Special
ADDON*	Modules » Add-on Modules

*Only available for DASYLab versions installed with Add-on serial number.

The following submenus are defined for `"ME"`:

HS	Experiment » Hardware Setup
ES	Experiment » Experiment Setup
TB	Experiment » Time Base Setup

The following submenus are defined for `"HLP"`:

HW	Help

Use the separator `//` in `szSubMenusEntry` to create further submenus in the browser.

The modules are integrated in the respective Init-function of the menu structure as described in chapter 5.3.2. To do so, check the following functions:

- `Init_DERIVATION` (file `Deriv.c`, function `FillTkDllsMenu_DERIVATION`)
- `Init_DEMANDTRIG` (file `dmd_trig.c`, function `FillTkDllsMenu_DEMANDTRIG`)
- `Init_GENERATOR` (file `Generat.c`, function `FillTkDllsMenu_GENERATOR`)
- `Init_LAMP` (file `Lamp.c`, function `FillTkDllsMenu_LAMP`).

5.3 An example module class: **Derivation**

Each of the module classes defined inside a DLL is an independent unit. You will usually have a separate source file for each module class.

We present a complete new DASYLab module class named `Derivation`, which is very similar to DASYLab*'s* internal Derivation class. The user can select between two functions: Calculating the derivation of a channel or calculating the integral of a channel.

Modules of the Derivation class are typical data processing modules. We will show examples of data source and data sink module classes later.

5.3.1 Internal data structure of a module class

A module class should not use any global or static variables, except possibly some space to save global strings loaded from the resource. Therefore it should not define any (static or non-static) variables outside of function scope and it should not declare any static variables inside a function scope.

It is very important not to mix up different modules' data, so every instance (= module) of a class must get its own data space.

When creating a new module of some class, DASYLab allocates data space for three different types of private data for that module which are described in the following sequences. The DASYLab kernel does not know the internal structure of this data; it just needs to know the sizes of these three different structures to allocate the memory.

5.3.1.1 Non temporary parameters of a module class

`ThisModule->ModuleParameters`: Parameters of the module that can be set up using the module parameter dialog box. We use the `MODULE_DERIVATION` structure in the example below to define that data.

```
typedef struct
{
   char szChannelName[MAX_CHANNEL][24];      /* Channel Name (all Modules)  */
   UINT wFunction[MAX_CHANNEL];              /* Derivation, Integral, ...   */
   BOOL___bPhaseCorrection[MAX_CHANNEL];     /* Phase Correction (OLD)      */
   BOOL bRestart[MAX_CHANNEL];               /* Restart after...            */
   UINT wBlocks[MAX_CHANNEL];                /* ... 'Blocks' ?              */
   WORD wDerivVersion;                       /* Internal version of this module */
   BOOL bCopyChannelName[MAX_CHANNEL];       /* Copy channel names */
   char szUnit[MAX_CHANNEL][MAX_UNIT_LEN];   /* Unit */
} MODULE_DERIVATION;
```

This structure only is necessary if a module is saved with the worksheet: Since DASYLab version 5.0 three kinds of worksheet files can be saved:

1. Binary worksheet

The DASYLab kernel will save the data as one big chunk, not looking at the contents of the structure. When the worksheet is loaded, the DASYLab kernel restores this chunk before sending a `Load` message to the module.

For this reason it is important **never** to change the data structure once it has been defined. It is, however allowed to extend the structure at the end. If you absolutely need to change the size of a component in the structure, you must rename, but leave intact the old field, and add a new field at the end. As you can see the parameter `__bPhaseCorrection` is not needed in the module `DERIVATION`. Two underlines show this fact and preserve the size of the module structure.

An example of this would be to increase the length of the channel names from 24 to, say 36. You would make a change looking like this:

```
typedef struct
{
   char __old_ChannelName[MAX_CHANNEL][24];
   UINT wFunction[MAX_CHANNEL];    /* Derivation, Integral, ...  */
   char ChannelName[MAX_CHANNEL][36];      /* new Channel Name  */
   ...
} MODULE_DERIVATION;
```

When loading an old worksheet, the DASYLab Kernel loads the old block, and fills up the remaining bytes with zeroes before sending the `Load` message to the module.

On receipt of the `Load` message, the module should check if the new names are all empty, and, if it is true, assume an old worksheet was loaded and copy all old names into their new places.

Modules classes which do not follow these rules will have serious problems when loading old worksheets.

2. ASCII worksheet for compatibility with previous versions of DASYLab

Binary worksheets cannot be loaded in previous versions of DASYLab. The solution for this problem is the use of ASCII-worksheets. To save the parameters of a module in ASCII-format, the `PARAMETER_INFO` structure is introduced. This structure contains the module parameters in a special manner. A pointer to this structure has to be returned if a worksheet is loaded or saved as ASCII worksheet.

Refer to chapter 6.2 for a detailed description of this structure.

3. Text worksheet for documentation purposes only

The use of ASCII-worksheets contains the possibility to save worksheets as documentation text files. All module parameters given in the `PARAMETER_INFO` structure are used except those, whose description starts with a '*' sign. DASYLab cannot load a worksheet from this file type.

5.3.1.2 Temporary parameters of a module class

`ThisModule->TempModuleData`: Intermediate data for the module, such as window handles, pointers to additional allocated memory blocks, etc. These are not saved with the worksheet. We use the `VAR_DERIVATION` struct in the example below to define that data.

```
typedef struct
{
   /* This module supports event actions */
   int NumSyncActions;                 /* Number of supported synchron events */
   struct
   {
      BOOL     bReceived;                    /* Sync event has happened        */
      BOOL     bFulFilled;                   /* Sync event with/without action */
      int      ChanMask;                     /* Mask only connected channels   */
      int      DoneMask;                     /* Mark ready channels            */
      UINT     ActionNumber;                 /* Which action is wanted         */
      double   ActionTime;                   /* When did the event appear      */
   } SyncAction[MAX_SYNC_ACTIONS];
} VAR_DERIVATION;
```

5.3.1.3 Temporary parameters of each channel

`ThisModule->TempChannelData[channel]`: Intermediate data for each channel, such as intermediate results of computation. These are also not saved with the worksheet. We use the `CHANNEL_DERIVATION` structure in the example below to define that data.

```
typedef struct
{
   double y1;                    /* Last Data Value of the last Data Block  */
   double z;                     /* Value of the Integral                   */
   DWORD    dwCount;             /* Actual Block Counter */
} CHANNEL_DERIVATION;
```

5.3.2 Registering a new module class with DASYLab

When DASYLab executes the `INIT` function of a DLL, the `INIT` function will call the `INIT` functions of all of the module classes contained within. At this point the module class must register itself with DASYLab.

The module class must fill a `MODCLASS` structure containing

- The instance handle of the DLL containing the module class
- The internal module class name for identification
- The respective sizes of the data structures mentioned above
- A string for the default module name
- A string describing the function of the module
- The help index in the help file
- The name of the help file that contains the help for this module.
- The ID of the Black Box type the module can appear in; always `BB_UNIVERSAL` at present.
- An icon to be shown in the module bar.
- An icon for the module block in the worksheet
- A `ProcessData` function to do all the data processing while the experiment is running. This function therefore handles the ProcessData message.
- **A** `PerformAction` function to handle all other messages sent to the module

All unused fields must be filled up with zeroes. This should be done by setting the entire structure to zero first, and then fill out the fields needed.

Then call the `RegisterModuleClass` function:

```
static char IdString_DERIVATION[MODULE_NAME_LENGTH+1];
static char StatusString_DERIVATION[MODULE_STATUS_LENGTH];
void Init_DERIVATION ( void )
{
   MODCLASS mc;
   memset ( &mc, 0, sizeof(mc) );

   LoadString (hInst, MN_MODULE_DERIVATION, IdString_DERIVATION, MODULE_NAME_LENGTH+1);
   LoadString (hInst, MN_STATUS_DERIVATION, StatusString_DERIVATION, MODULE_STATUS_LENGTH-1);
   mc.hInst         = hInst;
   mc.Name          = "NEW:DERIV";
   mc.DataSize      = sizeof (MODULE_DERIVATION);
   mc.VarSize       = sizeof(VAR_DERIVATION);
   mc.ChannelSize   = sizeof (CHANNEL_DERIVATION);
   mc.MenuId        = MN_MODULE_DERIVATION;
   mc.IdString      = IdString_DERIVATION;
   mc.StatusString  = StatusString_DERIVATION;
   mc.HelpId        = 0;
   mc.HelpFileName  = NULL;
   mc.BBoxId        = BB_UNIVERSAL;
   mc.ModIcon       = LoadBitmap (hInst, "DERIVATION_ICO");
```

```
   mc.BlkIcon       = LoadBitmap (hInst, "DERIVATION");
   mc.PerformAction = PerformAction_DERIVATION;
   mc.ProcessData   = ProcessData_DERIVATION;

   // Insert menu item in the DASYLab menu.
   if ( mc.MenuId != 0 )
      FillTkDllsMenu_DERIVATION ( mc.MenuId );

   RegisterModuleClass ( &mc );
}
```

If there are any additional handles allocated, they must be freed when the `DQM_UNREGISTER_CLASS` message is sent to the module.

The `ModIcon` and `BlkIcon` items need not (and must not) be freed by the module class because this is done by the DASYLab kernel.

After having successfully called the `RegisterModuleClass` function, the module class is known to DASYLab and can now be used like any other module class: you can add the module class to the function bar, create modules of that class, etc.

The new module class does not automatically appear in the standard menu bar of DASYLab, but since the menu bar can be extended by the DLL, the DLL can also add the new module classes to the menu bar.

5.3.3 Processing messages sent by DASYLab

All messages sent to a module are sent to the `PerformAction` function. The only exception is the `ProcessData` message which has its own handling function for efficiency reasons.

DASYLab will send messages to a module in response to certain events, for example, the creation/deletion of a module. These events will be described below.

Generally, do not perform message handling inside the `PerformAction` function, but use the `PerformAction` function only as a switch to the desired location.

We suggest the following scheme for the `PerformAction` function:

```
unsigned long PerformAction_DERIVATION ( MODULE *ThisModule, int wMsg, int wParam, long lParam )
{
   switch ( wMsg )
   {
      /* DASYLab Class Messages */
      case DQM_UNREGISTER_CLASS:
         return Exit_DERIVATION();

      /* DASYLab Module Messages */
      case DMM_CREATE_MODULE:
         return Create_DERIVATION ( ThisModule );

      case DMM_DELETE_MODULE:
         return Delete_DERIVATION ( ThisModule );

      case DMM_PARAM_MODULE:
         return Param_DERIVATION ( ThisModule );

      case DMM_SAVE_MODULE:
         return Save_DERIVATION ( ThisModule );

      case DMM_LOAD_MODULE:
         return Load_DERIVATION ( ThisModule );

      case DMM_START_MODULE:
         return Start_DERIVATION ( ThisModule );

      case DMM_STOP_MODULE:
```

```
            return Stop_DERIVATION ( ThisModule );

         case DMM_GET_PARAMETERS_INFO:
         {
            return (long) ParameterDerivation;
         }
         break;

         case DMM_SET_CHANNEL_NAME_COPY_OPTION:
         {
            MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;
            UINT i;

            for (i=0; i<MAX_CHANNEL; i++)
               ModuleInfo->bCopyChannelName[i] = wParam;
         }
         break;

         case DMM_GET_CHANNEL_NAME:
         {
            MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;

            if (wParam<MAX_CHANNEL)
            {
               if ( ModuleInfo->bCopyChannelName[wParam] )
                  strcpy ( ModuleInfo->szChannelName[wParam], (LPSTR)lParam);
               CopyChannelName (ThisModule->Fifo[wParam],
                                (LPSTR *)ModuleInfo->szChannelName[wParam]);
            }
         }
         break;

         /* Register here all possible event messages for the action module */
         case DMM_QUERY_MODULE_ACTION:
         case DMM_QUERY_SYNC_MODULE_ACTION:
            return ActionQuery_DERIVATION ( ThisModule, wMsg, wParam,
                                            (ONE_ACTION FAR *) lParam );

         /* Process event messages */
         case DMM_MODULE_ACTION:
         case DMM_CLEAR_SYNC_MODULE_ACTIONS:
         case DMM_INIT_SYNC_MODULE_ACTION:
         case DMM_SYNC_MODULE_ACTION:
         case DMM_SYNC_MODULE_NO_ACTION:
            return Action_DERIVATION ( ThisModule, wMsg, wParam,
                                       (ONE_ACTION FAR *) lParam );

         /* DASYLab Channel Messages */
         case DCM_SETUP_FIFO:
            return SetupFifo_DERIVATION ( ThisModule, wParam );

         case DCM_GET_UNIT:
         {
            MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;

            return (long) ModuleInfo->szUnit[wParam];
         }
         break;

         default:
            break;
      }

      return PerformDefaultAction ( ThisModule, wMsg, wParam, lParam );
   }
```

A module may receive several other messages which are sent by the DASYLab kernel for Black Box handling, etc. Standard modules should not handle these messages by themselves.

It is important to call the `PerformDefaultAction` function for every message the module class does not want to process on its own. This also ensures proper operation in the case that we may introduce new messages in the future.

5.3.4 Processing the DQM_UNREGISTER_CLASS message

The DQM_UNREGISTER_CLASS message is sent to the module class only one time at the very end, immediately before DASYLab exits. All modules were deleted before so the worksheet is empty at the time the DQM_UNREGISTER_CLASS message occurs.

For standard data processing modules, there is no need to process this message. Modules classes that allocate additional resources in the INIT function should free them here.

A typical example of this are visualization modules which register a window class inside the INIT function and must unregister it here again.

The return value in response to this message must always be TRUE.

```
static BOOL Exit_DERIVATION ( void )
{
   /* Nothing to do for most module classes */
   return TRUE;
}
```

5.3.5 Processing the DMM_CREATE_MODULE message

The DMM_CREATE_MODULE message is sent to each newly created instance of a module class immediately after it has been created. The DASYLab kernel has allocated space for the module's private data (three structures, see above) before sending this message.

The module must check, if its number of inputs and outputs are zero, and if so, define default values for them. In one special case (loading an old worksheet when the structure of the module's data has badly changed) DASYLab will pre-set values for the number of inputs and outputs and the module is not allowed to change these. The standard case, however, is that DASYLab sends the message with number of inputs and outputs set to zero before.

The module must then set up default values for all of its parameter settings. So every module class has to process this message.

The module should return TRUE if the message was processed successfully, and FALSE in case of error. Before returning FALSE, it should display a message box explaining what went wrong.

```
static BOOL Create_DERIVATION (MODULE *ThisModule)
{
   MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;
   CHANNEL_DERIVATION   *ChannelInfo;
   char                 DummyBuf[10];
   UINT                 i;

   /* Setup Default Values for Number of Inputs and Outputs */
   if ( ThisModule->wNumInpChan == 0 && ThisModule->wNumOutChan == 0 )
   {
      ThisModule->wNumOutChan = 1;
      ThisModule->wNumInpChan = 1;
   }

   /* Setup Default Values for Private Data */
   for ( i=0; i<MAX_CHANNEL; i++ )
   {
      /* Setup Channel Name */
      strcpy (ModuleInfo->szChannelName[i], IdString_DERIVATION);
      strcat (ModuleInfo->szChannelName[i], " ");
      itoa (i, DummyBuf, 10);
      strcat (ModuleInfo->szChannelName[i], DummyBuf);

      /* Default Function */
      ModuleInfo->wFunction[i] = IDD_DERIVATION_DIFF;

      strcpy ( ModuleInfo->szUnit[i], "#0/s" );
```

```
   }

   return TRUE;
}
```

5.3.6 Processing the DMM_DELETE_MODULE message

The DMM_DELETE_MODULE message is sent to each instance of a module class immediately before it will be deleted by the DASYLab kernel. It should free any allocated memory, handles, etc. that were allocated while processing the Create or Load message.

There is only one Delete message that applies to both: modules that were newly created; and modules that were loaded from disk. For this reason care must be taken to ensure that new and loaded modules allocate the same objects.

The module must always return TRUE in response to this message.

```
static BOOL Delete_DERIVATION (MODULE *ThisModule)
{
   /* Since we didn't allocate Memory or GDI Objects: */
   /* Nothing to do */
   UNUSED (ThisModule); /* Prevent Compiler Warning */
   return TRUE;
}
```

5.3.7 Processing the DMM_LOAD_MODULE message

The DMM_LOAD_MODULE message is sent to each newly created instance of a module class immediately after it has been loaded from a saved worksheet. The DASYLab kernel has allocated space for the module's private data (three structures, see above) and has loaded the first of them (ThisModule->ModuleParameters) with the values found on the disk before sending this message.

The module must do initialization setup similar to that of the Create message. After having processed this message, all necessary settings, opening windows, allocating space, etc. should be done that also take place for the Create message.

The module also handles changes in the ThisModule->ModuleParameters structure here. See the comments above for how this must be done.

The module should return TRUE if the message was processed successfully, and FALSE in case of error. Before returning FALSE, it should display a message box explaining what went wrong.

```
static BOOL Load_DERIVATION (MODULE *ThisModule)
{
   MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;
   int                  i;

   /* Here the use of the wDerivVersion-parameter is obvious: Older versions of */
   /* this module class won't have the szUnit-parameter. By checking the */
   /* internal version we can set this parameter to default values. */
   if ( ModuleInfo->wDerivVersion < 1 )
   {
      for ( i=0; i<MAX_CHANNEL; i++ )
         strcpy ( ModuleInfo->szUnit[i],
                  GetDefaultString ( ModuleInfo->wFunction[i] ) );
   }
   ModuleInfo->wDerivVersion = DERIV_VERSION;

   return TRUE;
}
```

5.3.8 Processing the DMM_SAVE_MODULE message

The DMM_SAVE_MODULE message is sent to each module immediately before the DASYLab kernel saves a worksheet. The Module may look up intermediate values like window positions and add

them to the ThisModule->ModuleParameters field. That field is then saved en-bloc by the DASYLab kernel.

The module must always return TRUE in response to this message.

```
static BOOL Save_DERIVATION (MODULE *ThisModule)
{
   /* Store Window Sizes, ...  but for this Module: */
   /* Nothing to do */
   UNUSED (ThisModule); /* Prevent Compiler Warning */
   return TRUE;
}
```

5.3.9 Processing the **DMM_GET_PARAMETERS_INFO** message

The DMM_GET_PARAMETERS_INFO message is sent before a worksheet is saved in ASCII-format or text format for documentation purposes or saved a worksheet is loaded from an ASCII-file. The returned value to this message is a pointer to an array of the PARAMETER_INFO structure which describes the private variables of the module in question. An example is given in the PerformAction_DERIVATION function in chapter 5.3.3.

See chapter 6.2 for detailed information about the PARAMETER_INFO structure.

5.3.10 Processing the **DMM_GET_EXTRA_PARAM_SIZE** message

Some modules have parameters of variable size which have to be stored with the worksheet. The DMM_GET_EXTRA_PARAM_SIZE message is sent to receive the size of those parameters. If a module returns a value greater than 0 as response to this message the DMM_GET_EXTRA_PARAMS is sent to this module afterwards to get a pointer to the parameters of variable size.

```
case DMM_GET_EXTRA_PARAM_SIZE:
   return ExtraParamSize;
```

5.3.11 Processing the **DMM_GET_EXTRA_PARAMS** message

The DMM_GET_EXTRA_PARAMS message is sent to store parameters of variable size with the worksheet. The module must return a pointer to these parameters in response of this message.

```
case DMM_GET_EXTRA_PARAMS:
   return (long) ExtraParams;
```

5.3.12 Processing the **DMM_SET_EXTRA_PARAMS** message

The DMM_SET_EXTRA_PARAMS message is sent to load parameters of variable size from the worksheet. The lParam parameter with an offset of four is the pointer to these parameters.

```
case DMM_SET_EXTRA_PARAMS:
   if ( lParam != 0 )
   {
      ExtraParams = (void *) ( lParam + 4 );
   }
   break;
```

5.3.13 Processing the **DCM_SETUP_FIFO** message

The DCM_SETUP_FIFO messages are sent to every module that contains output channels, before the experiment gets started, and after the DMM_PREPARE_START_MODULE message is sent to the module.

One DCM_SETUP_FIFO message is sent for every output channel of the module. The messages may be sent in any order, so you must not assume any specific order there. In particular, the message may be sent for a higher channel before a lower channel. You should treat the output channels completely independent of each other.

Before sending the DCM_SETUP_FIFO message for one specific channel, the DASYLab kernel will set up default values for the ChannelType, ChannelFlags, MaxBlockSize and SampleDistance on that channel. The module then should check if the values are in the expected range, and may change any of them as appropriate.

For example: FFT calculation will change the ChannelType to FFT data, averaging data of 4 points into one will then multiply the sample rate by that factor of four (because the samples at the output of the module are four times as wide apart from each other than the input samples are), etc.

Note: It is very important to keep the channel information consistent with the data on the channel. If your module changes the data rate, the type of data, the block size, etc., you must set up the correct values here.

The module should return TRUE if the message was processed successfully, and FALSE in case of error. Before returning FALSE, it should display a message box explaining what went wrong.

```
static BOOL SetupFifo_DERIVATION (MODULE *ThisModule, UINT wFifoNr)
{
   /* For Demonstration Purpose we added some future Data Types */
   switch ( ThisModule->Fifo[wFifoNr]->ChannelType )
   {
      case KT_NORMAL:          /* Standard Channel Type */
      case KT_SPEC:            /* Spectral Data, full Length */
      case KT_SPEC2:           /* Spectral Data, half Length */
      case KT_CLASS:           /* Histogram Data old Type */
      case KT_CLASS2:          /* Histogram Data new Type with Time Info */
         /* These were the allowed Data Types, were we have nothing else to do */
         break;
      /* case KT_BINARY:          TTL Data */
      /* case KT_SPEC3:           Spectral Data with half Length + 1 */
      default:
         /* Only the above Types of Data are allowed; otherwise we return */
         /* FALSE; so an Error Message would be sent */
         return FALSE;
   }

   return TRUE;
}
```

We have added the validity check for the channel type here. As an alternative, it could be added to the handling of the DMM_START_MODULE message. Modules that don't have output channels will never see a DCM_SETUP_FIFO message, and must therefore do all checks inside the DMM_START_MODULE message handler.

5.3.14 Processing the DMM_PREPARE_START_MODULE message

The DMM_PREPARE_START_MODULE message is sent to every module before the DMM_START_MODULE message is sent to the module. A module can process this message to check if the experiment can be started. If the module cannot start the experiment it must return FALSE after processing this message.

5.3.15 Processing the DMM_START_MODULE message

The DMM_START_MODULE message is sent to every module before the experiment gets started, but after the DMM_PREPARE_START_MODULE messages have been sent to the module.

The module should set up intermediate results, reset counters etc. here to prepare for the coming experiment. At this point the characteristics of each input channel are also known, and the module should check, if its input channels have the right characteristics (type, flags), if they match each other, etc. If it allocates any object here, these objects must be freed again inside the handling of the DMM_STOP_MODULE message.

The module should return TRUE if the message was processed successfully, and FALSE in case of error. Before returning FALSE, it should display a message box explaining what went wrong.

```
static BOOL Start_DERIVATION (MODULE *ThisModule)
{
   MODULE_DERIVATION *ModuleInfo = ThisModule->ModuleParameters;
   CHANNEL_DERIVATION *ChannelInfo;
   UINT i;

   for (i=0; i<ThisModule->wNumOutChan; i++)
   {
      /* Per Channel Data */
      ChannelInfo = ThisModule->TempChannelData[i];
      /* Initialize internal Values every time */
      ChannelInfo->y1 = 0.0;
      ChannelInfo->z  = 0.0;
      ChannelInfo->dwCount = 0;
   }
   return TRUE;
}
```

5.3.16 Processing the **DMM_GO_MODULE** message

The DMM_GO_MODULE message is sent to every module immediately before the experiment starts. The corresponding function which is processing this message must return quickly.

5.3.17 Processing the **DMM_PREPARE_STOP** message

This message is sent when the worksheet stops. For example, the message can terminate driver module threads before DASYLab- FIFOs logs off. This prevents crashes within a thread when accessing FIFOs that no longer exist. The DASYLab sequence control calls DMM_PREPARE_STOP before DMM_STOP_MODULE.

5.3.18 Processing the **DMM_STOP_MODULE** message

The DMM_STOP_MODULE message is sent to every module after the experiment is stopped. The module should free resources allocated inside the DMM_START_MODULE message handler. The module must always return TRUE in response to this message.

```
static BOOL Stop_DERIVATION (MODULE *ThisModule)
{
   UNUSED ( ThisModule );
   return TRUE;
}
```

5.3.19 Processing the **DMM_PAUSE_MODULE** message

The DMM_PAUSE_MODULE message is sent to every module when the user paused the experiment.

5.3.20 Processing the **DMM_RESUME_MODULE** message

The DMM_RESUME_MODULE message is sent to every module when the user resumed the experiment after it has been paused.

5.3.21 Processing the **DMM_PARAM_MODULE** message

The DMM_PARAM_MODULE message is sent to a module when the user double-clicks on its block in the worksheet. The module should display a dialog box allowing the user to set up and modify the module parameters.

Most of the work is done in the dialog box function (i.e. DERIVATIONProc) as explained below. So we just call the dialog box here.

The module must always return TRUE in response to this message.

```
static BOOL Param_DERIVATION (MODULE *ThisModule)
```

```
{
   HWND hwnd = ThisModule->hwndModule;
   DialogBox (hInst, "DERIVATION", hwnd, DERIVATIONProc);
   return TRUE;
}
```

The above form of the Dialog Box call assumes that DERIVATIONProc is equipped with windows prolog/epilog code. Most of today's Windows compilers can generate this prolog/epilog code automatically for every function.

For old compilers not supporting this, you have to use an additional MakeProcInstance:

```
static void Param_DERIVATION (MODULE *ThisModule)
{
   HWND hwnd = ThisModule->hwndModule;
   FARPROC lpDlgProc;
   lpDlgProc = MakeProcInstance ((FARPROC) DERIVATIONProc, hInst);
   DialogBox (hInst, "DERIVATION", hwnd, (DLGPROC) lpDlgProc);
   FreeProcInstance (lpDlgProc);
}
```

Using the old form does no harm when used with the new compilers. So if you have problems with dialog boxes, you may safely test the old form to see if the problems are gone. If that does not resolve your problems and you still don't see the box appearing on the screen, then the dialog box template is probably missing in your resource file. We will discuss the DialogBox handling later.

5.3.22 Processing the **DMM_QUERY_MODULE_ACTION** message

The DMM_QUERY_MODULE_ACTION message is sent to every module in order to ask, if the module supports asynchronous event driven actions.

5.3.23 Processing the **DMM_QUERY_SYNC_MODULE_ACTION** message

The DMM_QUERY_SYNC_MODULE_ACTION message is sent to every module in order to ask, if the module supports synchronous event driven actions. Depending on the action number, the module can choose, if the specified action is supported. As shown in the example, the messages DMM_QUERY_SYNC_MODULE_ACTION and DMM_QUERY_MODULE_ACTION are treated in the same function, because they are both supported. If a module has to differ between synchronous and asynchronous actions, there should be two different functions in order to check the action support. Here the predefined ACTION_RESET variable is supported.

```
static BOOL ActionQuery_DERIVATION ( MODULE *ThisModule, int wAction, int wChannel, ACTION *Action )
{
   switch ( Action->Number )
   {
      /* We can process a Reset action */
      case ACTION_NULL:
      case ACTION_RESET:
         return TRUE;
      default:
         break;
   }

   return FALSE;

   UNUSED(ThisModule);
   UNUSED(wAction);
   UNUSED(wChannel);
}
```

5.3.24 Processing the **DMM_EXPLAIN_USER_ACTION** message

The DMM_EXPLAIN_USER_ACTION message is sent to the specific module to declare new user defined event driven actions. Each module class has the possibility to define up to 25 private event driven actions. The IDs ACTION_USER_0,…,ACTION_USER_24 are defined in the file CONST.H. The example GENERAT.C describes the usage of user defined event driven actions. Here the frequency and

amplitude can be changed by the action module. For better syntax understanding the IDs ACTION_USER_0 and ACTION_USER_1 are renamed to ACTION_SET_FREQ and ACTION_SET_AMPLITUDE.

The following parameters can be defined:

- Actual instance handle hInst of the DLL.
- The action description string InternalName.
- Number and kind of parameters Params which can be changed. Please use the predefined IDs from TYPES.H (ACTION_NO_PARAMS, ACTION_1_PARAMS, ACTION_2_PARAMS, ACTION_3_PARAMS, ACTION_4_PARAMS or ACTION_STRING_PARAMS). As you see, you can use no parameters, up to 4 *double* or *integer* parameters or a single *string* parameter in one action channel.
- The string ID IdName of the event driven action. DASYLab reads with the help of the instance handle and the IdName the name of the action out of the resource file of the DLL.
- Buffer of string parameter IDs for each parameter that can be defined. This string is shown in the dialog box of the action module above each parameter edit field.
- The parameter ParamType describes the kind of variable the action module should show in the dialog box. Possible are ACTION_PARAM_TYPE_INT for integer and ACTION_PARAM_TYPE_DOUBLE for double parameters.

```
static BOOL ExplainAction_GENERATOR ( MODULE *ThisModule, int wAction, int wChannel,
                                      ACTION_DESCRIPTION *Acd )
{
   switch ( Acd->Number )
   {
     case ACTION_SET_FREQ:
       Acd->hInst = hInst;
       strcpy (Acd->InternalName, "GEN_SET_FREQ");
       Acd->Params = ACTION_1_PARAMS;
       Acd->IdName = STR_ACTION_SET_FREQ;
       Acd->IdParams[0] = STR_FREQUENZ;
       Acd->ParamType[0] = ACTION_PARAM_TYPE_DOUBLE;
       return TRUE;

     case ACTION_SET_AMPLITUDE:
       Acd->hInst = hInst;
       strcpy (Acd->InternalName, "GEN_SET_AMPL");
       Acd->Params = ACTION_1_PARAMS;
       Acd->IdName = STR_ACTION_SET_AMPLITUDE;
       Acd->IdParams[0] = STR_AMPLITUDE;
       Acd->ParamType[0] = ACTION_PARAM_TYPE_DOUBLE;
       return TRUE;

     default:
       break;
   }

   return FALSE;

   UNUSED (ThisModule);
   UNUSED (wAction);
   UNUSED (wChannel);
}
```

5.3.25 Processing the DMM_MODULE_ACTION message

The DMM_MODULE_ACTION message is sent to the specific module to perform an asynchronous action. Depending on the action number the module can select the concerning action. This and the

following messages are analyzed in the function `Action_DERIVATION` depending on the specified action.

```
....
switch ( wAction )
   {
      /* Asynchron event */
      case DMM_MODULE_ACTION:
      {
         /* Which event */
         switch ( Action->Number )
         {
            /* Reset */
            case ACTION_RESET:
            {
               CHANNEL_DERIVATION * ChannelInfo;
               UINT i;

               /* Select all channels */
               if ( wChannel == 0 )
                  wChannel = -1;

               for ( i=0; i<ThisModule->wNumOutChan; i++ )
               {
                  if ( wChannel & ( 1L << i ) )
                  {
                     ChannelInfo = ThisModule->TempChannelData[i];

                     /* Reset variable */
                     ChannelInfo->z = 0.0;
                  }
               }
            }
            return TRUE;

            default:
               break;
         }
      }
      break;
....
```

5.3.26 Processing the **DMM_CLEAR_SYNC_MODULE_ACTIONS** message

The `DMM_CLEAR_SYNC_MODULE_ACTIONS` message is sent to the specified module to initialize the variables needed for the synchronous actions.

```
...
/* Clear all list of synchronous actions */
case DMM_CLEAR_SYNC_MODULE_ACTIONS:
{
   VAR_DERIVATION *PrivatVars = ThisModule->TempModuleData;

   PrivatVars->NumSyncActions = 0;
}
break;
...
```

5.3.27 Processing the **DMM_INIT_SYNC_MODULE_ACTION** message

The `DMM_INIT_SYNC_MODULE_ACTION` message is sent to the specified module to initialize the variables needed for the synchronous actions. Only `MAX_SYNC_ACTIONS` actions are possible to create for each module. The variable `PrivatVars->NumSyncActions` counts the used synchronous actions.

```
/* Init synchronous actions */
case DMM_INIT_SYNC_MODULE_ACTION:
{
   VAR_DERIVATION *PrivatVars = ThisModule->TempModuleData;

   /* We support only MAX_SYNC_ACTIONS actions */
   if ( PrivatVars->NumSyncActions >= MAX_SYNC_ACTIONS )
      return FALSE;
```

```
      /* Select all channels */
      if ( wChannel == 0 )
         wChannel = -1;

      /* Deselect not connected channels */
      wChannel &= (int) ( (1L << ThisModule->wNumOutChan) - 1 );

      /* Sign action index */
      Action->ReceiveID = PrivatVars->NumSyncActions;

      /* Init varialbles */
      PrivatVars->SyncAction[Action->ReceiveID].ActionNumber = Action->Number;
      PrivatVars->SyncAction[Action->ReceiveID].ActionTime = 0.0;
      PrivatVars->SyncAction[Action->ReceiveID].bReceived = FALSE;
      PrivatVars->SyncAction[Action->ReceiveID].bFulFilled = FALSE;
      PrivatVars->SyncAction[Action->ReceiveID].ChanMask = wChannel;
      PrivatVars->SyncAction[Action->ReceiveID].DoneMask = 0;

      /* Prepare next action number */
      PrivatVars->NumSyncActions += 1;

      return TRUE;
   }
   break;
```

5.3.28 Processing the **DMM_SYNC_MODULE_ACTION** message

The `DMM_SYNC_MODULE_ACTION` message is sent to the specified module when a synchronous event has happened. If the module supports synchronous actions, the data flow depends on the messages `DMM_SYNC_MODULE_ACTION` or `DMM_SYNC_MODULE_NO_ACTION`. These messages appear continuously, in order to process new data blocks and in the case of `DMM_SYNC_MODULE_ACTION` start the specified event driven action.

```
/* It's a sychron event driven action */
case DMM_SYNC_MODULE_ACTION:
case DMM_SYNC_MODULE_NO_ACTION:
{
   VAR_DERIVATION *PrivatVars = ThisModule->TempModuleData;

   /* First we process old actions */
   if ( PrivatVars->SyncAction[Action->ReceiveID].bReceived )
      return FALSE;

   /* Have we to do any actions */
   if ( wAction == DMM_SYNC_MODULE_ACTION )
      PrivatVars->SyncAction[Action->ReceiveID].bFulFilled = TRUE;
   else
      PrivatVars->SyncAction[Action->ReceiveID].bFulFilled = FALSE;

   /* Set received flag */
   PrivatVars->SyncAction[Action->ReceiveID].bReceived = TRUE;

   /* Set ready mask */
   PrivatVars->SyncAction[Action->ReceiveID].DoneMask = 0;

   /* Set action start time */
   PrivatVars->SyncAction[Action->ReceiveID].ActionTime = Action->fStartTime;
}
break;
```

First of all the variable `PrivatVars->SyncAction[Action->ReceiveID].bReceived` is tested, if the old action has been processed. If the message `DMM_SYNC_MODULE_ACTION` has been sent, the action has to be performed. The flag `PrivatVars->SyncAction[Action->ReceiveID].bFulFilled` is set to `TRUE` and will be worked out in the `ProcessData` function.

5.3.29 Processing the **DMM_SYNC_MODULE_NO_ACTION** message

The `DMM_SYNC_MODULE_NO_ACTION` message is sent to the specified module when no synchronous action has to be performed. See example above.

5.3.30 Processing the **DMM_COPY_CHANNEL_NAME** message

DASYLab supports copying channel names from one module to another. This message is sent to **data source modules** to copy the name of each channel to the connected module(s) by using the `CopyChannelName` function. The processing of this message is given in the Generator example:

```
case DMM_COPY_CHANNEL_NAME:
      {
         MODUL_GENERATOR *PrivatInfo = ThisModule->ModuleParameters;
         UINT i;

         for (i=0; i<ThisModule->wNumOutChan; i++)
         {
            CopyChannelName (ThisModule->Fifo[i], (LPSTR *)PrivatInfo->szChannelName[i]);
         }
      }
      break;
```

5.3.31 Processing the **DMM_GET_CHANNEL_NAME** message

This message is sent to data processing and data sink modules: The `wParam` value specifies the current channel number and the `lParam` value is a pointer to the channel name of the previous module. If the channel names of the previous module should be copied this message should be processed like this:

```
case DMM_GET_CHANNEL_NAME:
      {
         MODULE_DERIVATION    *ModuleInfo = ThisModule->ModuleParameters;

         if (wParam<MAX_CHANNEL)
         {
            if ( ModuleInfo->bCopyChannelName[wParam] )
               strcpy ( ModuleInfo->szChannelName[wParam], (LPSTR)lParam);

            /* If the module has output channels: copy channel name to the connected module(s) */
            CopyChannelName ( ThisModule->Fifo[wParam], (LPSTR *)ModuleInfo->szChannelName[wParam] );
         }
      }
      break;
```

5.3.32 Processing the **DMM_SET_CHANNEL_NAME_COPY_OPTION** message

When worksheets of older DASYLab versions are loaded, the user is asked if he wants to set the option to copy channel names from one module to another. Depending on the user's choice, the corresponding variable is initialized with the value of `wParam` of this message:

```
case DMM_SET_CHANNEL_NAME_COPY_OPTION:
      {
         MODULE_DERIVATION *ModuleInfo = ThisModule->ModuleParameters;
         UINT i;

         for (i=0; i<MAX_CHANNEL; i++)
            ModuleInfo->bCopyChannelName[i] = wParam;
      }
      break;
```

5.3.33 Processing the **DCM_GET_UNIT** message

The handling of physical units for each channel of a module is similar to copying channel names:

```
case DCM_GET_UNIT:
      {
         MODULE_DERIVATION *ModuleInfo = ThisModule->ModuleParameters;

         return (long) ModuleInfo->szUnit[wParam];
      }
      break;
```

Processing this message means to copy the unit of each channel to the connected module(s). The `wParam` value specifies the current channel number.

Data source modules should provide a list box in which the physical unit of the channel can be selected. The unit can be copied through data processing modules by using the placeholder `#0`. If a data processing module performs arithmetic operations that have effect on the physical unit different operations can be done with the placeholder. See the DASYLab manual for details. Data sink modules should simplify the unit by using the `ExpandUnitString` function (→ example `Lamp.c`).

5.3.34 Processing the **DMM_SAVE_WINDOW_POS** message

The `DMM_SAVE_WINDOW_POS` message is sent to the specified module when the user wants to create a new window arrangement. The actual window position and dimensions are saved under the actual number stored in the `wParam` variable. The function `SaveHwndPos` of the toolkit has to be called in order to save the actual window position. The data for storing the positions are placed in the `ModuleWndPos` structure. In order to recall the positions, the data have to be placed in the module parameters which are copied into the worksheet. See the example `LAMP.C` where this function is used.

5.3.35 Processing the **DMM_DEL_WINDOW_POS** message

The `DMM_DEL_WINDOW_POS` message is sent to the specified module when the actual arrangement should be deleted. No further action has to be done here.

5.3.36 Processing the **DMM_SHOW_WINDOW_POS** message

The `DMM_SHOW_WINDOW_POS` message is sent to the specified module when the actual arrangement number `wParam` should be shown. Call here the toolkit function `ShowHwndPos`.

5.3.37 Processing the **DMM_NORMALIZE_ALL_WINDOWS** message

DASYLab sends this message to modules, if function **View » All Windows » Restore** was executed.

5.3.38 Processing the **DMM_HIDE_ALL_WINDOWS** message

DASYLab sends this message to modules, if function **View » All Windows » Hide** was executed.

5.3.39 Processing the **DMM_SHOW_ALL_WINDOWS** message

DASYLab sends this message to modules, if function **View » All Windows » Show** was executed.

5.3.40 Processing the **DMM_MINIMIZE_ALL_WINDOWS** message

DASYLab sends this message to modules, if function **View » All Windows » Minimize** was executed.

5.3.41 Processing the **DMM_ACTIVATE_LAYOUT** message

The `DMM_ACTIVATE_LAYOUT` message is sent when the layout window is activated. Any child window should be hidden.

5.3.42 Processing the **DMM_ACTIVATE_WORKSHEET** message

The `DMM_ACTIVATE_WORKSHEET` message is sent when the worksheet window is activated. The actual window arrangement for this view should be restored.

5.3.43 Processing the **DMM_ACTIVATE_DISPLAY_WND** message

The `DMM_ACTIVATE_DISPLAY_WND` message is sent when the display window is activated. The actual window arrangement for this view should be restored.

5.3.44 Processing the **DMM_GLOBAL_VAR_CHANGED** and **DMM_GLOBAL_STRING_CHANGED** message

The DMM_GLOBAL_VAR_CHANGED and DMM_GLOBAL_STRING_CHANGED message is sent, if the value of a global variable or a global string has changed. This message is sent to the module only if it registered global variables and/or global strings. See chapter *5.4* Using global strings or variables with DASYLab for details.

5.3.45 Processing the **DMM_CHANGE_VAR_NAME** message

The DMM_CHANGE_VAR_NAME message is sent if the name of a global variable or a global string has changed. The message will have the old name as wParam and the new name as lParam.

If the module uses global variables by number, no action needs to be taken, since the global variable/string functions will handle the new name correctly.

However, if the module stores placeholders for global variables or strings in text strings (like for file names etc.), the module should call the ChangeNameInString function to change the string to the use of the new name and return the appropriate result. If several such strings have to be processed, return a Boolean AND of all those return values to indicate any failure.

Here is an example of how to handle the DMM_CHANGE_VAR_NAME message:

```
case DMM_CHANGE_VAR_NAME:
{
   MODULE_FILESAVE *ModuleInfo = ThisModule->ModuleParameters;

   return ChangeNameInString (ModuleInfo->szFileName, sizeof (ModuleInfo->szFileName),
                              (LPSTR) wParam, (LPSTR) lParam);
}
```

5.3.46 Replacing modules

*5.3.46.1 Processing the **DMM_CREATE_REPLACE** message*

The DMM_CREATE_REPLACE message is sent to a module if the user clicks **Replace module** in the context menu of the module block. The lParam of the message contains a pointer to a ModuleReplaceTemplate structure. This is a MODULE structure which has all relevant parameters copied from the module's MODULE structure. The ModuleParameters member however does not point to the module's private parameters, but to a MODULE_DATA_TEMPLATE structure, which is an abstraction of the module's parameters. This can be used by a module which will replace the existing one to setup its parameters correctly. Parameters supported for transfer to the new module are: channel names, units, and CopyChannelName settings.

If the module supports several subtypes which could replace each other (e.g. the Arithmetic module), the module can set the bReplaceItself member of the MODULE_DATA_TEMPLATE structure to TRUE. In this case, the same module will appear in the list of module available for replacement.

It is the module's responsibility to copy its private parameters to the MODULE_DATA_TEMPLATE where appropriate. If the creation of the template fails or the module does not support replacing, return FALSE to this message, otherwise TRUE. FALSE is default.

See the toolkit examples for setting up a DMM_CREATE_REPLACE handler for special types of modules.

*5.3.46.2 Processing the **DQM_CHECK_REPLACE** message*

If the user has clicked **Replace module** in the context menu of the module block and a ModuleReplaceTemplate has successfully been created by the above message, all module classes

are inquired if they can replace the existing module by the DQM_CHECK_REPLACE module class message. The lParam of the message contains a pointer to the ModuleReplaceTemplate.

The module class should check the settings in the ModuleReplaceTemplate and return TRUE if it can replace a module described by the template, and FALSE otherwise. Default return value is FALSE.

Things especially to check here are number of input and output channels and the ChannelRelation type. Note that e.g. a relay module cannot replace an arithmetic module with two inputs and one output although it might have the same number of inputs and outputs, but a different ChannelRelation.

See the toolkit examples for setting up a DQM_CHECK_REPLACE handler for special types of modules.

*5.3.46.3 Processing the **DMM_REPLACE_MODULE** message*

If the user has selected a module for replacing an existing one, the MODULE structure of the new module will be created by DASYLab. To initialize the new module, the DMM_REPLACE_MODULE message will be sent to it. The lParam of the message contains a pointer to the ModuleReplaceTemplate of the original module.

Usually it is a good idea to call the CreateModule handler first to setup the module parameters. Note that no DMM_CREATE_MODULE message is sent. It should be made sure that the Create handler does not reset e.g. the number of input and output channels. However, the setup for the *Replace* case can be made completely separate from the *Create* case.

For safety, the ChannelRelation parameter of the new module's MODULE structure should be set, since it might be different in the original module.

Finally, the module should copy the relevant parameters from the MODULE_DATA_TEMPLATE structure of the original module. Note that each parameter in this structure is masked by a Boolean flag that indicates whether the entry is valid and should be copied.

If the replace operation fails for some reason, the module might return FALSE to this message. The replace operation will be cancelled in that case, reporting an error message to the user. Note that in this case, the module is responsible for doing all cleanup operations necessary to delete the new module, a separate DMM_DELETE_MODULE message will not be sent.

If the replace operation succeeded, return TRUE.

See the toolkit examples for setting up a DMM_REPLACE_MODULE handler for special types of modules.

*5.3.46.4 Processing the **DMM_GET_MODULE_TYPE** message*

If the module supports several subtypes which could replace each other (e.g. the Arithmetic module), the module should return its subtype identifier through this message.

This message will be sent to the module only in the case of an Undo of a Replace operation in order to identify whether the subtype of a module has been changed by the preceding Replace operation.

The subtype identifier can be any integer number which is a unique indication of the module's subtype. It is only checked for equality to the subtype identifier of the original module if the module class is the same.

5.3.47 Processing the **DMM_REQUEST_GLOB_VARS** message

Since version 8, DASYLab provides an overview of the usage of all global variables – it is called the *Variable Overview*. In order to get information regarding the usage of variables of a module it sends

the message DMM_REQUEST_GLOB_VARS to every module of the current worksheet. If a module does not answer this message the appropriate module will be listed as "*has to be checked manually*" because the *Variable Overview* cannot decide, if variables are used or not in this module.

Even if the module does not use any variable it has to answer this request. See the following example from the file deriv.c.

```
BOOL RequestGlobalVars_DERIVATION (MODULE *ThisModule, LPARAM pHandle)
{
   GLVO_MODULE_NOTIFY_PARAM Param;

   Param.wSize = sizeof (GLVO_MODULE_NOTIFY_PARAM);
   Param.uiVersion = GLVO_VERSION;
   Param.pHandle = pHandle;
   Param.wVarUsage = GLVO_VARUSAGE_NEVER;

   Param.wVarType = GLVO_VARTYPE_NONE;
   Param.wVarNumber     = 0;
   Param.wAccess = GLVO_ACCESS_NONE;
   Param.iChannelNumber = -1;
   Param.bAction = FALSE;
   Param.wActionNumber = 0;
   strcpy(Param.szDescription, "---");
   GlvoModuleNotifyGlobalVar (&Param);

   return TRUE;

   UNUSED (ThisModule);
}
```

The call of the function GlvoModuleNotifyGlobalVar with the properly filled structure variable Param notifies the *Variable Overview* if and in which way variables are used inside this module.

See the following example from the file generator.c where global variables are used for frequency and amplitude.

```
BOOL RequestGlobalVars_GENERATOR (MODULE *ThisModule, LPARAM pHandle)
{
   GLVO_MODULE_NOTIFY_PARAM Param;
   MODULE_GENERATOR  *ModuleInfo = ThisModule->ModuleParameters;
   UINT i;

   Param.wSize = sizeof (GLVO_MODULE_NOTIFY_PARAM);
   Param.uiVersion = GLVO_VERSION;
   Param.pHandle = pHandle;

   Param.wVarType = GLVO_VARTYPE_NUMBER;
   Param.wAccess = GLVO_ACCESS_READ;
   Param.bAction = FALSE;                  /* must be FALSE ! */
   Param.wActionNumber = 0;                /* must be 0 !     */

   for ( i=0; i<ThisModule->wNumOutChan; i++ )
   {
      /* Frequency */
      LoadString (hInst, STR_FREQUENZ, Param.szDescription,
         sizeof(Param.szDescription));
      if (ModuleInfo->nVarFrequency[i] == 0 )
      {
         /* Set proper parameters in order to show that we do not */
         /* use a variable here                                   */
         Param.wVarUsage = GLVO_VARUSAGE_NOT_YET;
         Param.wVarNumber     = 0;
         Param.iChannelNumber = -1;
         strcpy(Param.szDescription, "---");
      }
      else
      {
         Param.wVarUsage      = GLVO_VARUSAGE_USED;
         Param.wVarNumber     = (short)ModuleInfo->nVarFrequency[i];
         Param.iChannelNumber = i;
      }
      GlvoModuleNotifyGlobalVar (&Param);
```

```
      /* Amplitude */
      LoadString (hInst, STR_AMPLITUDE, Param.szDescription,
              sizeof(Param.szDescription));
      if (ModuleInfo->nVarAmplitude[i] == 0 )
      {
         /* Set proper parameters in order to show that we do not */
         /* use a variable here                                   */
         Param.wVarUsage = GLVO_VARUSAGE_NOT_YET;
         Param.wVarNumber     = 0;
         Param.iChannelNumber = -1;
         strcpy(Param.szDescription, "---");
      }
      else
      {
         Param.wVarUsage      = GLVO_VARUSAGE_USED;
         Param.wVarNumber     = (short)ModuleInfo->nVarAmplitude[i];
         Param.iChannelNumber = i;
      }
      GlvoModuleNotifyGlobalVar (&Param);
   }

   /* return true in order to show that we processed the */
   /* message DMM_REQUEST_GLOB_VARS                      */
   return TRUE;
}
```

Please note that this function has to return `TRUE`. If it does not, the module will appear in the list of unsupported modules.

The elements of `GLVO_MODULE_NOTIFY_PARAM` in detail:

- `wSize`:
  The size of the structure `GLVO_MODULE_NOTIFY_PARAM`.
- `pHandle`:
  The handle given by the request message.
- `wVarType`:
  The type of the global variable. Can be `GLVO_VARTYPE_NONE`, `GLVO_VARTYPE_NUMBER`, `GLVO_VARTYPE_STRING` or `GLVO_VARTYPE_SYSTEM`.
- `wVarNumber`:
  The Number of the numeric, string or system variable. Please note that system variables will not be shown in the Variable Overview.
- `wAccess` (kind of access):
  `GLVO_ACCESS_NONE`, `GLVO_ACCESS_READ`, `GLVO_ACCESS_WRITE`, `GLVO_ACCESS_RW`.
- `wVarUsage` (Kind of usage):
  `GLVO_VARUSAGE_NEVER`, `GLVO_VARUSAGE_NOT_YET`, `GLVO_VARUSAGE_USED`, `GLVO_VARUSAGE_USED_AS_INDEX`. Please use the constant `GLVO_VARUSAGE_NOT_YET` if the specific module parameter can be a variable but currently is not. The constant `GLVO_VARUSAGE_USED_AS_INDEX` has to be used if the content of the variable will be used as the number of another variable. So you will very likely do not need it.
- `iChannelNumber`:
  Number of the current channel. Usually in the range from 0 to 15 for standard channels but all numbers up to 255 are allowed. This can be useful for modules using a master/slave concept. If the variable is not related to a channel, you have to set -1 here.
- `bAction`:
  Always `FALSE`!
- `wActionNumber`:
  Always 0!

- szDescription:
  Please fill this element with a description of the module parameter which holds the variable. For example "Frequency" or "Amplitude". The size of the string can be up to GLVO_USAGEDESCRIPTION_MAXLEN characters.

5.3.48 Processing the DMM_REQ_MODULE_DEFAULT message

Since version 8, DASYLab provides setting user defined module defaults. When a user defined default has been set for a specific module, every time the user creates this module, it will load these settings. However, module default settings are only possible, if the creator of the specific module give permission to do that. The way to do that is to include the following lines in PerformAction_XXX:

```
case DMM_REQ_MODULE_DEFAULT:
   return TRUE;
```

The example modules in the toolkit already contain these lines. In some seldom cases, it might be necessary to refuse a permit for setting module defaults – e.g. if you providing a master/slave organization of your modules, or if the settings depend on hardware options which may change in certain circumstances. To do so just return FALSE, and the user will not be able to set a default for your module.

5.3.49 Processing the DMM_IS_DEBUG message

This message informs DASYLab whether a module originates from a release or from a debug DLL. Modules from a debug DLL have a red frame in the worksheet. The following is an example for the use of source texts within the PerformAction:

```
      case DMM_IS_DEBUG:
# ifdef _DEBUG
         return 2;
# else
         return TRUE;
# endif
```

5.3.50 Processing the DMM_GET_TIMEBASE_ID message

DASYLab 2016 introduced this new message. It is used to support the system to find out, which time bases are actual used in the worksheet.

In recent versions of DASYLab, all time bases are stored in the worksheet. When you take this worksheet to another computer with different measurement hardware, the time bases of the original hardware stay inside the worksheet, and there was no way to delete them. DASYLab was augmented with an option do delete unused time bases from the worksheet. For this purpose it needs to ask every module, which time base ID it uses. The default function returns 0 (= no time base is used) what is appropriate for all data processing modules.

But all data source modules (slider, switch, generator, hardware input modules, hardware output modules) need to return the time base ID, they are bind to, or they are synchronized to. If you do not answer to that message, the block size and the sample rate of your time base do not make it into the work sheet. Example is in Generat.c in this toolkit example:

```
case DMM_GET_TIMEBASE_ID:
   return ((MODULE_GENERATOR *)ThisModule->ModuleParameters)->uiTimeBase;
```

Hint: The time bases *DASYLab* and *Driver* are always stored into the flowchart – so nothing to do for you, when you use these time bases.

5.3.51 Processing the **ProcessData** message

We shall now discuss the probably most interesting item: the processing of the data.

Unlike all of the other messages, the `ProcessData` message is not sent by sending it to the general `PerformAction` message handler. For efficiency reasons, the DASYLab kernel calls the `ProcessData` function directly. Despite this, we will continue talking about the `ProcessData` message here.

The `ProcessData` message is sent to a module periodically while the experiment is running. The DASYLab kernel keeps a list of all modules of a worksheet (including those inside black boxes) and continuously chains through this list, sending one `ProcessData` message to every module in the list again and again until the experiment is stopped.

One important fact to know is that this process can be interrupted. No `ProcessData` messages are sent while the user drags a window or while a disk save is in progress, etc. It is also not possible for the DASYLab kernel to guarantee any minimal number of messages sent to a module in some time interval. The messages are always sent as fast as possible, and the rate depends on many parameters like the CPU power of your computer and the number and type of the modules in the worksheet.

The actions performed on the `ProcessData` message depends on the type of the module. For data processing modules like the *Derivation* example, you will typically find actions like this:

Once for each channel:

a) Check if there is enough space on the output side to hold one more data block. If so,

b) Check if data is available on the input side. If so,

c) Read one block of data from the input side, process the data, and output the processed data to the output side, and move the animation marker.

We will discuss other possibilities later when looking at more examples.

```
static int ProcessData_DERIVATION ( MODULE *ThisModule )
{
   MODULE_DERIVATION *ModuleInfo = ThisModule->ModuleParameters;
   CHANNEL_DERIVATION *ChannelInfo;
   UINT i, wFifoNr;
   FIFO_HEADER *OutFifo;
   DATA_BLOCK_HEADER *OutputBlock;
   DATA_BLOCK_HEADER *InputBlock;

   /* Walk through all the FIFO´s after each other */
   for (wFifoNr=0; wFifoNr<ThisModule->wNumOutChan; wFifoNr++)
   {
      /* Now we´re in one FIFO */
      OutFifo = ThisModule->Fifo[wFifoNr];
      ChannelInfo = ThisModule->TempChannelData[wFifoNr];

      /* Do we have Space to create a new Block ? */
      if ((OutputBlock = GetCurrentOutputBlock (OutFifo)) != NULL)
      {
         /* Are there Data Blocks at the Input ?*/
         InputBlock = GetInputBlock ( ThisModule, wFifoNr );

         if ( InputBlock != NULL &&
              /* have we received any actions */
              GotSyncActions ( ThisModule, wFifoNr, InputBlock->fStartTime ))
         {
            double faktor, y1, z;

            /* Perform actions */
            DoSyncActions(ThisModule,wFifoNr);

            /* Copy Time Information */
```

```
                OutputBlock->fStartTime     = InputBlock->fStartTime;
                OutputBlock->fSampleDistance = InputBlock->fSampleDistance;
                OutputBlock->wBlockSize   = InputBlock->wBlockSize;

                /* Do the computing... */

                switch ( ModuleInfo->Function[wFifoNr] )
                {
                   case IDD_DERIVATION_DIFF:
                   {
                      faktor = 1.0 / InputBlock->fSampleDistance;
                      y1 = ChannelInfo->y1;

                      for (i=0; i<InputBlock->wBlockSize; i++)
                      {
                         z = InputBlock->Data[i];
                         OutputBlock->Data[i] =
                            (DLAB_FLOAT) ( faktor * ( z - y1 ) );
                         y1 = z;
                      }

                      ChannelInfo->y1 = y1;
                   }
                   break;

                   case IDD_DERIVATION_INT:
                   {
                      faktor = InputBlock->fSampleDistance;
                      z = ChannelInfo->z;

                      for (i=0; i<InputBlock->wBlockSize; i++)
                      {
                         z += faktor * InputBlock->Data[i];
                         OutputBlock->Data[i] = (DLAB_FLOAT) z;
                      }

                      ChannelInfo->z = z;
                   }
                   break;

                   default:
                      /* That should not happen */
                      ImpossibleCase();
                      return FALSE;
                }

                /* Add this Data Block to the FIFO, so that a "Son" Block can */
                /* get Access to it */
                ReleaseOutputBlock (OutFifo);

                /* Release "Father" Block */
                ReleaseInputBlock (ThisModule, wFifoNr);

                /* With Restart ? */
                ChannelInfo->wCount += 1;
                if ( ChannelInfo->wCount >= ModuleInfo->wBlocks[wFifoNr] )
                {
                   if ( ModuleInfo->bRestart[wFifoNr] )
                   {
                      ChannelInfo->y1 = 0.0;
                      ChannelInfo->z  = 0.0;
                   }
                   ChannelInfo->wCount = 0;
                }

             }
          }
       }
       return TRUE;
    }
```

5.3.52 Event driven actions

We shall now discuss the appearing asynchronous or synchronous actions. As shown in the examples above there are a lot of messages which coordinate the use of the event driven actions. Here is just a short overview about the general functionality of supporting event driven actions.

5.3.52.1 Register predefined event driven action support

Insert the `DMM_QUERY_SYNC_MODULE_ACTION` for synchronous and `DMM_QUERY_MODULE_ACTION` for asynchronous actions into the `PerformAction` function. The function is called for each action (IDs defined in file `CONST.H`). Quit the supported actions with a `TRUE` return value.

Action	**ID constant in CONST.H**
Printing	`ACTION_PRINT`
Reset value	`ACTION_RESET`
Set old value to new one	`ACTION_SET`
Fade channel in	`ACTION_FADE_IN`
Fade channel out	`ACTION_FADE_OUT`
Perform next	`ACTION_NEXT`
Copy to clipboard	`ACTION_TO_CLIPBOARD`
Display all channels in one window	`ACTION_ONE_WINDOW`
Set caption string	`ACTION_SET_CAPTION`
Perform message box	`ACTION_WARN_MESSAGE`
Select windows arrangement	`ACTION_SEL_WIN_SETUP`
Load worksheet	`ACTION_LOAD_DSB`
Load and start worksheet	`ACTION_LOAD_GO_DSB`
Start backup of files	`ACTION_BACKUP`
Send layout to printer	`ACTION_PRINT_LAYOUT`
Display layout as full screen	`ACTION_FULLSCREEN_LAYOUT`
Bring layout to top	`ACTION_ACTIVATE_LAYOUT`
Change global variable	`ACTION_SET_VAR`
Change global string	`ACTION_SET_STRING`
Save global variable to INI	`ACTION_SAVE_VAR`
Save global string to INI	`ACTION_SAVE_STRING`
Get global variable from INI	`ACTION_LOAD_VAR`

Action	ID constant in CONST.H
Get global string from INI	ACTION_LOAD_STRING
Add constant to global variable	ACTION_ADD_VAR
Multiply constant with global variable	ACTION_MULT_VAR
Show layout not in full screen	ACTION_NORMALIZE_LAYOUT
Activate worksheet MDI at top	ACTION_SHOW_WORKSHEET
Repaint all layout objects	ACTION_UPDATE_LAYOUTER_OBJECTS
Enter a global string	ACTION_INPUT_GLOBAL_STRINGS
Enter a global variable	ACTION_INPUT_GLOBAL_VARS
Start an external EXE file	ACTION_START_EXTERNAL_PROGRAM
Quit DASYLab	ACTION_EXIT_DASYLAB
Quit Windows	ACTION_EXIT_WINDOWS
Quit and restart windows	ACTION_EXIT_RESTART_WINDOWS
Do a message beep	ACTION_MESSAGE_BEEP
Stop experiment	ACTION_STOP_DSB
Stop and restart experiment	ACTION_STOP_RESTART_DSB
Save global variable to INI-file	ACTION_SAVE_VAR_TO_FILE
Save global string to INI-file	ACTION_SAVE_STRING_TO_FILE
Load global variable from INI-file	ACTION_LOAD_VAR_TO_FILE
Load global string from INI-file	ACTION_LOAD _STRING_TO_FILE
Create directory	ACTION_CREATE_DIR
Copy global variable to another	ACTION_COPY_VAR
Copy global string to another	ACTION_COPY_STRING

5.3.52.2 Register user defined event driven action support

Insert the `DMM_EXPLAIN_USER_ACTION` supporting user defined event driven actions into the `PerformAction` function. The function is called for all user IDs (IDs defined in file `CONST.H`). Define here for the selected IDs the necessary variables as described under *Processing the `DMM_EXPLAIN_USER_ACTION` message*.

User Action	ID constant in CONST.H
1	ACTION_USER_0
2	ACTION_USER_1
..	..
24	ACTION_USER_24

5.3.52.3 Initialize variables for synchronous event driven action support

Insert the DMM_CLEAR_SYNC_MODULE_ACTIONS and DMM_INIT_SYNC_MODULE_ACTION into the PerformAction function. When supporting synchronous event driven actions here the needed variables are initialized as described in the example above.

5.3.52.4 Asynchronous action handling

As shown in the ProcessData function above whenever an asynchronous event appears we get the DMM_MODULE_ACTION message. Now we have to perform the action request immediately. For example, reset a certain value. The variable ACTION->Number specifies the specific action called to process.

5.3.52.5 Synchronous action handling

We shall now discuss the appearing synchronous event driven actions. As shown in the ProcessData function above whenever a synchronous event appears we get the DMM_SYNC_MODULE_ACTION or the DMM_SYNC_MODULE_NO_ACTION message. This message is sent out from the action module after every data block, to synchronize the data and message flow. The module (here: Derivation) has to wait until it receives one of these messages until it can perform a new data block. When these messages appear, no action is performed but all necessary variables are set to the specific values so that the data flow in the ProcessData function can be continued. The ProcessData function contains two new functions which control the message flow in the synchronous mode. Here is the example of the derivation module.

5.3.52.6 Check received synchronous actions

The GotSyncActions function is called every time the ProcessData function is performed. If the number of supported synchronous actions is higher than 0, the data evaluation can be continued if the synchronous event flags for the selected channel are set. Otherwise the function returns with the Boolean value FALSE and the ProcessData function is left.

```
static BOOL GotSyncActions (MODULE *ThisModule, UINT wFifoNr, double fStartTime)
{
   VAR_DERIVATION *PrivatVars = ThisModule->TempModuleData;
   int i;

   /* Test each sychron action */
   for ( i=0; i<PrivatVars->NumSyncActions; i++ )
   {
      /* Is this channel an synch action channel */
      if ( PrivatVars->SyncAction[i].ChanMask & ( 1L << wFifoNr ) )
      {
         /* Have we received the synch action already */
         if ( ! PrivatVars->SyncAction[i].bReceived )
            return FALSE;

         /* Have we worked out this action for this channel */
         if ( PrivatVars->SyncAction[i].DoneMask & ( 1L << wFifoNr ) )
            return FALSE;

        /* Is the time of the synch action nearly equal the time of the data block */
         if ( ! IsNearlyEqual(fStartTime,PrivatVars->SyncAction[i].ActionTime,0.001) )
```

```
            {
               /* If not then stop the experiment and show a sweet messagebox */
               StopExperiment ();
               strcpy ( ShortTempString, ThisModule->ModuleName );
               LoadString (hInstDlab, STR_ILL_ACTION_TIME,
      LongTempString, sizeof(LongTempString));
               ShowWarning ( ShortTempString, LongTempString );
               return FALSE;
            }
         }
      }

      return TRUE;
   }
```

5.3.52.7 Perform synchronous action messages in the `ProcessData` function

If the test in synchronous mode has succeeded, we can perform all necessary actions and reset the variables `DoneMask` and `bReceived`, so that new event driven actions can be received and performed.

```
   static void DoSyncActions (MODULE *ThisModule, UINT wFifoNr)
   {
      VAR_DERIVATION *PrivatVars = ThisModule->TempModuleData;
      int i;

      /* Test each sychron action */
      for ( i=0; i<PrivatVars->NumSyncActions; i++ )
      {
         /* Is this channel an synch action channel */
         if ( PrivatVars->SyncAction[i].ChanMask & ( 1L << wFifoNr ) )
         {
            /* Is an action necessary */
            if ( PrivatVars->SyncAction[i].bFulFilled )
            {
               /* OK then select the action */
               switch ( PrivatVars->SyncAction[i].ActionNumber )
               {
                  /* Reset of the derivation is the only action so far */
                  case ACTION_RESET:
                  {
                     /* Channel Data */
                  CHANNEL_DERIVATION *KanalInfo = ThisModule->TempChannelData[wFifoNr];

                     /* Reset variable */
                     KanalInfo->z = 0.0;
                  }
                  break;
               }
            }

            /* Mark the channel mask that this channel is ready */
            PrivatVars->SyncAction[i].DoneMask |= ( 1L << wFifoNr );

            /* If all channel actions are done, then prepare for the next event */
            if ( PrivatVars->SyncAction[i].DoneMask == PrivatVars->SyncAction[i].ChanMask )
               PrivatVars->SyncAction[i].bReceived = FALSE;
         }
      }
   }
```

5.3.53 Window Arrangement under DASYLab

The feature of arranging window placements can also be used by user defined window modules. In the included example `LAMP.C` you can see the general functionality of saving deleting and showing window arrangements. Supporting this function is quite easy: all you have to add are the following source lines into the `PerformAction_XXX` function of your module and add the necessary structure `MODULE_WND_POS` into the module parameters.

```
   case DMM_SAVE_WINDOW_POS:
```

```
{
   MODUL_LAMPE *PrivatInfo = ThisModule->ModuleParameters;

   SaveHwndPos ( &PrivatInfo->WndPos[wParam], ThisModule->hwndWindow);
   return TRUE;
}
break;

/* Delete the actual Window position */
case DMM_DEL_WINDOW_POS:
   return TRUE;

/* Delete all actual Window positions */
case DMM_DEL_ALL_WINDOW_POS:
   return TRUE;

/* Show the window at the defined position number */
case DMM_SHOW_WINDOW_POS:
{
   MODUL_LAMPE *PrivatInfo = ThisModule->ModuleParameters;

   ShowHwndPos ( &PrivatInfo->WndPos[wParam], ThisModule->hwndWindow);
   return TRUE;
}
break;
```

5.3.54 Parameter setup dialog box handling

As mentioned before, most of the real work for processing the `DMM_PARAM_MODULE` message is done inside the `DialogBox` function.

A lot of windows functions and messages will be used in the following text. The meaning of those can be found in the Windows SDK literature, or in the online help of your compiler. We will not discuss the Windows functions or messages in this document.

Dialog boxes for modules typically consist of three parts:

1. A set of global parameters that apply to the module in general or to all channels
2. A channel selection bar to select the number of channels and to select one active channel.
3. A set of channel specific parameters. Changes here only affect the active channel.

5.3.54.1 The `DisplayChannel` and `RetrieveChannel` functions

Generally, the functions `DisplayChannel` and `RetrieveChannel` are defined inside each module class. The `DisplayChannel` function displays the parameters for the active channel in the dialog box while the `RetrieveChannel` function reads back the changed values and saves them for the active channel.

```
static void DisplayChannel (HWND hDlg, UINT wNummer)
{
   MODULE_DERIVATION *TempInfo = SingleInfo;
   char StrBuf[24];
   itoa (wNummer, StrBuf, 10);
   SetDlgItemText (hDlg, IDD_KANAL_NR, StrBuf);
   SetDlgItemText (hDlg, IDD_KANAL_NAME, TempInfo->ChannelName[wNummer]);
   /* DERIVATION-Optionen */

   CheckRadioButton (hDlg, IDD_DERIVATION_DIFF, IDD_DERIVATION_INT,
                           TempInfo->Function[wNummer]);
}

static void RetrieveChannel (HWND hDlg, UINT wNummer)
{
   MODULE_DERIVATION *TempInfo = SingleInfo;
   GetDlgItemText (hDlg, IDD_KANAL_NAME,
                   TempInfo->ChannelName[wNummer], CHANNEL_NAME_LENGTH);

   /* Pushing any Buttons, Radio Buttons or Check Boxes is handled in the */
```

```
   /* WM_COMMAND case of the Dialog Box Function */
}
```

5.3.54.2 General outline of a dialog box function

A dialog box function is similar to the `PerformAction` function discussed above, but it will operate on the window handle of the dialog box window and receive Windows messages instead of DASYLab messages. The DASYLab kernel ensures that a pointer to the module in question can be found under `CurrentModulePtr` for the entire lifetime of the dialog box.

At the start of dialog box handling, the parameters of the module are copied to intermediate storage and a pointer to the data is saved in the global variable `SingleInfo`. This can then be accessed while processing subsequent messages.

While the dialog box is open, only the temporary storage is manipulated leaving the original data intact. When the user presses `OK`, the temporary data is copied over the original data. If the user pressed `Cancel`, nothing is copied and the original data remains intact.

A typical outline of a dialog box function is as follows:

```
BOOL CALLBACK DERIVATIONProc ( HWND hDlg, UINT Message, WPARAM wParam, LPARAM lParam )
{
   MODULE *ThisModule = CurrentModulePtr;
   MODULE_DERIVATION *TempInfo = SingleInfo;
   switch (Message)
   {
      case WM_INITDIALOG:
      {
         // Initialization of the dialog box
         // Copy the module's data to a temporary storage and modify
         // only this temporary data. So the original data remain intact
         // until the OK button is pressed.
      }
      break;
      case WM_CLICK_LDOWN:
      {
         // Left mouse click on channel bar: Select new active channel
      }
      break;
      case WM_COPYFOCUS:
      {
      // F7 was pressed. Copy the active parameter of the active channel
      // to all channels
      }
      break;
      case WM_COPYALL:
      {
         // F8 was pressed. Copy all parameters of the active channel
         // to all channels
      }
      break;
      case WM_COMMAND:
      {
         UINT cmd_id = GET_WM_COMMAND_ID (wParam,lParam);

         switch (cmd_id)
         {
            case IDD_LESS:
            {
               // Minus button was pressed: remove one channel
            }
            break;
            case IDD_MORE:
            {
               // Plus button was pressed: add one channel
            }
            break;
            case IDD_MODUL_NAME:
```

```
                case IDD_MODUL_BEZEICHNUNG:
                case IDD_KANAL_NAME:
                {
                 // Found in most module classes: The user clicked on the
                 // module name or description field.
                 // No action is necessary here, we just remember the
                 // last mouse-clicked item for the context sensitive help.
                }
                break;
                case IDD_DERIVATION_DIFF:
                case IDD_DERIVATION_INT:
                {
                 // Special handling for each module class: The user clicked
                 // on some parameter field.
                 // Possibly retrieve data or do other actions here plus
                 // save the mouse click for the on-line help.
                }
                break;
                case IDHELP:
                {
               // Call the context sensitive help, based on the knowledge of
               // the last item the user mouse-clicked
                }
                break;
                case IDOK:
                {
                   // OK button was pressed. Save all parameters.
                }
                break;
                case IDCANCEL:
                {
                 // Cancel button was pressed. Ignore all changed parameters
                 // and leave the original values intact.
                }
                break;
             }
          }
       }
       return FALSE;
    }
```

The routine processes only some of the messages sent to a dialog box. Others will be handled by a default handler provided by windows. The `WM_COMMAND` message is a special case, because it is sent whenever the user clicks on an item and you can access the field ID of the clicked field from the parameters. This field ID is then used for sub-casing this message.

We will now discuss the actions done by our *Derivation* example for the individual messages.

*5.3.54.3 Actions for **WM_INITDIALOG***

In response to `WM_INITDIALOG` the module creates intermediate storage, copies the data to that storage, sets up the channel bar, selects the first channel to be active, displays the global data and the data of the active channel, defines limits for the maximum string length for all of the text input fields, and sets a focus (active item inside the dialog box).

It may (and does in this example) disable some functions while the experiment is running, if it does not want that parameters be changed during run time. You should take care to decide what parameters can, or cannot be changed during run time.

```
    case WM_INITDIALOG:
          {
             ThisModule = CurrentModulePtr;
             /* Allocate temporary Memory; if we leave the Dialog Box with    */
             /* "Cancel" we do not want to have changed values; so we create  */
             /* a Structure like the MODUL_... Structure, copy the Data from  */
             /* the original Structure, work on the temporary Structure and   */
             /* copy the Temporary Structure to the original Structure if the */
             /* User pushes the OK Button.                                    */
             SingleInfo = MemAlloc (sizeof (MODULE_DERIVATION));
             if (!SingleInfo)
```

```
            {
               EndDialog(hDlg, FALSE);
               return (TRUE);
            }
            TempInfo = SingleInfo;
            memcpy (TempInfo, ThisModule->ModuleParameters,
                    sizeof (MODULE_DERIVATION));
            /* If the Experiment is running, we cannot change the Function,  */
            /* but we like to see which is chosen. */
            if ( bExperimentIsRunning )
            {
               EnableWindow (GetDlgItem (hDlg, IDD_DERIVATION_DIFF), FALSE);
               EnableWindow (GetDlgItem (hDlg, IDD_DERIVATION_INT), FALSE);
            }
            /* Setup global Settings for the Dialog Box (More, Less Buttons) */
            wDialNumChan = ThisModule->wNumOutChan;
            wDialCurChan = 0;
            wDialMaxChan = 16;
            InitChannelBar (hDlg);
            DisplayChannel (hDlg, wDialCurChan);

            /* Set some Limits and initial Text for Edit Controls */
            Edit_LimitText (GetDlgItem (hDlg, IDD_KANAL_NAME),
                            CHANNEL_NAME_LENGTH);
            Edit_LimitText (GetDlgItem (hDlg, IDD_MODUL_NAME),
                            MODULE_NAME_LENGTH);
            SetDlgItemText (hDlg, IDD_MODUL_NAME, ThisModule->ModuleName);
            Edit_LimitText (GetDlgItem (hDlg, IDD_MODUL_BEZEICHNUNG),
                            MODULE_DESCRIPTION_LENGTH);
            SetDlgItemText (hDlg, IDD_MODUL_BEZEICHNUNG,
                            ThisModule->ModuleDescription);
            SetFocus (GetDlgItem (hDlg, IDD_MODUL_NAME));
            wDialLastFocus = 0;
            return FALSE;
         }
         break;
```

*5.3.54.4 Actions for **WM_CLICK_LDOWN***

This message is not a Windows message, but a message sent by the DASYLab kernel that looks similar to a Windows message.

In response to `WM_CLICK_LDOWN`, the module changes the active channel, that is: retrieve the possibly changed parameters of the current channel, select a new channel and then display the parameters of that channel.

```
      case WM_CLICK_LDOWN:
         {
            /* A different Channel was selected in the Channel Bar */
            if (LOWORD (lParam) == 2)
            {
               RetrieveChannel (hDlg, wDialCurChan);
               HandleChannelClick (hDlg, wParam);
               DisplayChannel (hDlg, wDialCurChan);
            }
            wDialLastFocus = IDD_KAN00;
            SetFocus (GetDlgItem (hDlg, IDD_MODUL_NAME));
            return TRUE;
         }
         break;
```

*5.3.54.5 Actions for **WM_COPYFOCUS***

This message is not a Windows message, but a message sent by the DASYLab kernel that looks similar to a Windows message. The message is send, when the user presses the *F7* key inside of a dialog.

In response to `WM_COPYFOCUS` the module should copy the focused parameter of the current channel to all other channels.

```
      case WM_COPYFOCUS:
```

```
        {
           UINT i;
           /* Handle F7, get the Field with the Focus and copy its content to all other Channels */
           RetrieveChannel (hDlg, wDialCurChan);
           DisplayChannel (hDlg, wDialCurChan);
           /* Who has the Focus */
           for (i=0; i<wDialNumChan; i++)
           {
              if (i != wDialCurChan)
              {
                 switch (wDialLastFocus)
                 {
                    case IDD_KANAL_NAME:
                       strcpy (TempInfo->ChannelName[i],
                                TempInfo->ChannelName[wDialCurChan]);
                       break;
                    case IDD_DERIVATION_DIFF:
                    case IDD_DERIVATION_INT:
                     TempInfo->Function[i] = TempInfo->Function[wDialCurChan];
                       break;
                 }
              }
           }
        }
        break;
```

*5.3.54.6 Actions for **WM_COPYALL***

This message is not a Windows message, but a message sent by the DASYLab kernel that looks similar to a Windows message. This message is send, when the user presses the *F8* key inside of a dialog.

In response to `WM_COPYALL` the module should copy all parameters of the current channel to all other channels.

```
case WM_COPYALL:
      {
         UINT i;
         /* Handle F8, copy all fields to all other Channels */
         RetrieveChannel (hDlg, wDialCurChan);
         DisplayChannel (hDlg, wDialCurChan);
         for (i=0; i<wDialNumChan; i++)
         {
            if (i != wDialCurChan)
            {
               strcpy (TempInfo->ChannelName[i],
                         TempInfo->ChannelName[wDialCurChan]);
               TempInfo->Function[i] = TempInfo->Function[wDialCurChan];
            }
         }
      }
      break;
```

*5.3.54.7 Actions for **WM_COMMAND***

The module should retrieve the field ID of the field in question and then process the subcase for that field ID.

In the literature, you will often find the use of `wParam` for this field ID which works under Windows 3.1, but not under Win32 systems. Therefore you should use the compatibility macro shown below:

```
case WM_COMMAND:
      {
         UINT cmd_id = GET_WM_COMMAND_ID (wParam,lParam);

         switch (cmd_id)
         {
             // sub-cases
         }
      }
```

5.3.54.8 Actions for *WM_COMMAND* subcase *IDD_LESS*

This removes one channel. The DASYLab kernel ensures that this message will not be sent if only one channel is active and that it will not be sent while the experiment is running.

```
case IDD_LESS:
            {
               /* One Channel is canceled */
               RetrieveChannel (hDlg, wDialCurChan);
               HandleLessMoreButton (hDlg, IDD_LESS);
               DisplayChannel (hDlg, wDialCurChan);
               wDialLastFocus = IDD_KAN00;
               SetFocus (GetDlgItem (hDlg, IDD_MODUL_NAME));
               return TRUE;
            }
            break;
```

5.3.54.9 Actions for *WM_COMMAND* subcase *IDD_MORE*

This adds one channel. The DASYLab kernel ensures that this message will not be sent if the maximum number of channels is reached and that it will not be sent while the experiment is running.

```
case IDD_MORE:
            {
               /* One Channel was added */
               RetrieveChannel (hDlg, wDialCurChan);
               HandleLessMoreButton (hDlg, IDD_MORE);
               DisplayChannel (hDlg, wDialCurChan);
               wDialLastFocus = IDD_KAN00;
               SetFocus (GetDlgItem (hDlg, IDD_MODUL_NAME));
               return TRUE;
            }
            break;
```

5.3.54.10 Actions for *WM_COMMAND* subcase *IDD_MODUL_NAME*

Several subcases of the `WM_COMMAND` message require no special processing, but only need to be recognized for the context sensitive help.

```
case IDD_MODUL_NAME:
case IDD_MODUL_BEZEICHNUNG:
case IDD_KANAL_NAME:
        {
           /* Who has the Input Focus */
          if (GET_WM_COMMAND_CMD (wParam,lParam) == EN_SETFOCUS)
             wDialLastFocus = cmd_id;
             return FALSE;
        }
        break;
```

5.3.54.11 Actions for module class specific subcases of *WM_COMMAND*

Several subcases of the `WM_COMMAND` message are specific to our DERIVATION example. These are listed below.

In this example we only have to recognize function changes for a channel.

```
case IDD_DERIVATION_DIFF:
case IDD_DERIVATION_INT:
        {
           /* Who has the Input Focus */
           TempInfo->Function[wDialCurChan] = cmd_id;
           wDialLastFocus = cmd_id;
        }
        break;
```

5.3.54.12 Actions for *WM_COMMAND* subcase *IDHELP*

When the user presses the *Help button*, the dialog box will receive a `WM_COMMAND` message with sub-code `IDHELP`. It should then call the context sensitive help.

The help file name is loaded from the DLL's own resources, so every DLL should come with its own help file providing help for its modules.

The DASYLab online Help was changed from *WinHelp* (.HLP) to *HtmlHelp* (.CHM). The `case IDHELP` examples `Deriv.c`, `dmg_trig.c`, `Generat.c`, and `Lamp.c` show the fundamental procedures for integrating the *CHM Help*.

Download the compilation libraries necessary for the *HtmlHelp*, from the Microsoft internet page and copy them into the shared folder (`htmlhelp.lib` and `htmlhelp.h`). The respective `Includes` and `Calls` are commented out so that you can compile the resource without the necessary libraries. After you have copied the libraries into the shared folder, you must delete these comment rows. Search for *htmlhelp* to find the respective code positions.

```
case IDHELP:
{
   UINT wHelpID=MODULNAME;

   LoadString ( hInstDlab, ID_EVAHILFE, ShortTempString, sizeof(ShortTempString) );
   strcpy ( LongTempString, ExeFileDir );
   strcat ( LongTempString, ShortTempString );

   // Load help file with the parameter of the last focus
   switch ( wDialLastFocus )
   {
   case IDD_MODUL_NAME:
      wHelpID = MODULNAME;
      break;
   case IDD_MODUL_BEZEICHNUNG:
      wHelpID = MODULBESCHREIB;
      break;
   case IDD_KAN00:
      wHelpID = GL_KANALANZ;
      break;
   case IDD_KANAL_NAME:
      wHelpID = KBEZEICHNUNG;
      break;
   case ID_LIST_UNIT:
      wHelpID = KEINHEIT;
      break;
   default:
      /*
      !!!! Here your help file should be used !!!!
      LoadString ( hInst, ID_YOUR_HELP, ShortTempString, sizeof(ShortTempString) );
      wHelpID = YOUR_HELP_ID;
      */
      break;
   }

   HtmlHelp (
      NULL,
      LongTempString,
      HH_HELP_CONTEXT,
      wHelpID );

   wDialLastFocus = 0;
   return TRUE;
}
break;
```

The toolkit also provides the following functions which makes calling the HTML online help easier.

```
BOOL TKCallOnlineHelp (UINT uiHelpPathInfo, UINT uHelpCommand, UINT uiLastFocus,
                       char* chHelpFileName, DWORD_PTR dwHelpIndex );
```

The functions in detail:

- uiHelpPathInfo: Valid for this parameter are:
  - HELP_PATH_IS_ALREADY_COMPLETE_DESIGNED
    The chHelpFileName parameter already contains the name of the online help including the respective path specification.
  - HELP_PATH_INCLUDE_ONLY_THE_FILE_NAME
    The chHelpFileName parameter contains the name of the online help which DASYLab is searching for on the path specified by ExeFileDir.
- uHelpCommand: Specifies the action to perform.
  This parameter corresponds to the third parameter of the API function HtmlHelp.
- uiLastFocus: Corresponds to the fourth parameter of the API function HtmlHelp.
  If the transfer is an IDD_MODUL_NAME, IDD_MODUL_BEZEICHNUNG, IDD_KANAL_NAME, IDD_KAN00, or ID_LIST_UNIT, the subject-related call automatically occurs from the DASYLab online help. Otherwise the subject-related call dwHelpIndex is realized from chHelpFileName.
- chHelpFileName: Specifies the name of the online help.
  Refer to the parameter uiHelpPathInfo
- dwHelpIndex: Corresponds to the fourth parameter of the API function HtmlHelp, refer also to the parameter uiLastFocus

5.3.54.13 Actions for WM_COMMAND subcase IDOK

When the user presses the *OK button*, the dialog box will receive a WM_COMMAND message with sub-code IDOK. It should then copy the manipulated temporary data back over the original data of the module and call the ChangeModuleSize function to handle changes in the number of selected channels. Finally, it should call EndDialog to close the dialog box.

```
case IDOK:
     {
        SetFocus ( GetDlgItem ( hDlg, IDOK ) );

        RetrieveChannel (hDlg, wDialCurChan);
        GetDlgItemText (hDlg, IDD_MODUL_NAME,
                        ThisModule->ModuleName, MODULE_NAME_LENGTH+1);
        GetDlgItemText (hDlg, IDD_MODUL_BEZEICHNUNG,
                         ThisModule->ModuleDescription,
                         MODULE_DESCRIPTION_LENGTH+1);
        /* Save the temporary Structure to the original Structure */
        memcpy (ThisModule->ModuleParameters, TempInfo,
                   sizeof (MODULE_DERIVATION));
        /* Release Memory for temporary Structure */
        MemFree (SingleInfo);
        SingleInfo = NULL;
        /* Change the Size of the Module Icon if necessary */
        ChangeModuleSize (ThisModule, wDialNumChan, wDialNumChan);
        EndDialog(hDlg, TRUE);
        return (TRUE);
     }
     break;
```

5.3.54.14 Actions for WM_COMMAND subcase IDCANCEL

When the user presses the *Cancel button*, the dialog box will receive a WM_COMMAND message with sub-code IDCANCEL. It should then call EndDialog to close the dialog box without saving the manipulated data over the original data.

```
case IDCANCEL:
            {
               SetFocus ( GetDlgItem ( hDlg, IDCANCEL ) );
               /* Release Memory for temporary Structure */
               MemFree (SingleInfo);
               SingleInfo = NULL;
               EndDialog (hDlg, FALSE);
               return (TRUE);
            }
            break;
         }
      }
   }
   return FALSE;
```

This finishes the description of the dialog box handling, as well as the description of the example module class `DERIVATION`. You have now seen the complete code for a sample DLL containing one additional module.

5.3.55 Using default directories

DASYLab supports several default directories which also can be used by the Extension Toolkit user. Please use concerning the function of your module the adequate directory listed below.

Function Group	**Variable**
Data Values (like Data Save module)	`DefDataDir`
Device Data (like IEEE module)	`DefDeviceDir`
Additional Utilities	`DefUtilityDir`

The directory selection should be used as in the following example is described. If the actual filename `ActualDataDir` is empty the default directory `DefDataDir` is used as the variable `lpstrInitialDir` in the `OpenFileName` structure. So first time a directory is selected the default directory is used. Every new selection will start from the previously defined directory.

```
memset (&ofn, 0, sizeof (OPENFILENAME));

_splitpath ( ActualDataDir, szDrive, szDir, NULL, NULL);
if ( strlen ( szDir ) == 0 )
   strcpy ( szDirName, DefDataDir );
else
{
   _makepath ( szDirName, szDrive, szDir, NULL, NULL);
}

ofn.lStructSize = sizeof (OPENFILENAME);
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = szFilter;
ofn.nFilterIndex = 1;
ofn.lpstrFile = szFile;
ofn.nMaxFile = sizeof (szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof (szFileTitle);
LoadString (hInst, STR_DATA_SAVE, szTitle, 128);
ofn.lpstrTitle = szTitle;
LoadString (hInst, STR_DATA_EXT, szFileExt, 5);
ofn.lpstrDefExt = szFileExt;
ofn.lpstrInitialDir = szDirName;
```

```
ofn.Flags = OFN_PATHMUSTEXIST | OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT | OFN_ENABLEHOOK |
ofn_ExplorerFlag;
ofn.hInstance = hInst;
ofn.lpfnHook = CommDlgHookProc;
if (GetSaveFileName (&ofn))
{
   strcpy ( ActualDataDir, ofn.lpstrFile);
}
```

Look for the description of `ofn_ExplorerFlag` under the 32-bit-extension paragraph.

5.3.56 Using country-specific settings

When displaying text, Windows uses country-specific formats for the time, the date and numbers. You can specify these settings in the Windows *Control Panel* under the *International* icon. When DASYLab displays text on the screen or writes them in a file these country-specific formats are used. So, if you have to output text on the screen or in a file use the settings defined in the control panel. The DLL receives the information from the following variables:

Function	**Variable**
Leading zero in date value (02)	`bDateLeadingZero`
Leading zero in month value (09)	`bMonthLeadingZero`
Long format of year (1999)	`bDateYearLong`
Format of the date	`wDateFormat`
Counting time from 0 to 24 hours	`bTime24h`
Leading zero in time value (09)	`bTimeLeadingZero`
Time separator	`TimeTrenn`
Decimal separator	`DezTrenn`
Leading zero in decimal value	`bDecLeadingZero`
Date format separator	`DateTrenn`

All general string conversion functions of the toolkit use this variables. So, if you use these functions you don't have to care for correct country-specific formats.

5.3.57 Printing data or graphics with DASYLab

DASYLab supports a special page layout for graphical or text printings like in the Y/t chart, or the data list used. The page format which can be selected from the main menu is also reachable from the toolkit. The following example shows the general use of the printing functions used in DASYLab. Important is the `DruckInfo` structure which includes the variables used in order to make a print.

```
void CopyDataToPrinter ( MODUL_NEW *PrivatInfo, VAR_NEW *PrivatVars,
                         MODULE * ThisModule, BOOL bAskBefore)
{
   DruckInfo DI;

   /* Init Printer */
```

```
    if ( ! InitPrinter ( &DI, bAskBefore ) )
    {
       return;
    }
    /* Supports the printer colors */
    if ( GetDeviceCaps ( DI.dc, COLORRES ) <= 2 &&
         GetDeviceCaps ( DI.dc, PLANES ) <= 2 )
       PrivatInfo->bWithColor = FALSE;
    else
       PrivatInfo->bWithColor = TRUE;
    /* Change the output mode */
    PrivatVars->uiOutputMode = OUT_PRINTER;

    /* Lock all output windows in DASYLAB */
    EnableWindow ( ThisModule->hwndWindow, FALSE );

    /* Calculate the font */
    Setup_New_Font ( PrivatInfo, PrivatVars, DI.dc );

    /* Print Page Header */
    PrintHeader ( &DI, ThisModule->ModuleDescription );

    /* Save actual DC */
    SaveDC ( DI.dc );

    /* Set the origin and mode! */
    SetMapMode ( DI.dc, MM_ISOTROPIC );

    /* Set actaul window dimensions */
    PrivatInfo->uiXsize = DI.reData.right-DI.reData.left;
    PrivatInfo->uiYsize = DI.reData.bottom-DI.reData.top;

    /* Window (Set Coordinate system to display rectangle) setzen */
    SetWindowExtEx ( DI.dc, PrivatInfo->uiXsize, PrivatInfo->uiYsize, NULL );

    /* Set the Viewport */
    SetViewportOrgEx ( DI.dc, DI.reData.left, DI.reData.top, NULL );
    SetViewportExtEx ( DI.dc, PrivatInfo->uiXsize, PrivatInfo->uiYsize, NULL );

    NOW DO ALL PRINTER OUTPUTS WITH THE DEVICE CONTEXT DI.dc

    /* Print bottom layout */
    PrintFooter ( &DI );
    RestoreDC ( DI.dc, -1 );

    /* Leave printer device */
    ExitPrinter ( &DI );

    /* restore the Output mode */
    PrivatVars->uiOutputMode = OUT_SCREEN;
    /* restore the font */
    Setup_New_Font ( PrivatInfo, PrivatVars, NULL );
    PrivatInfo->bWithColor = TRUE;

    /* Unlock window outputs */
    EnableWindow ( ThisModule->hwndWindow, TRUE );
}
```

- The function `InitPrinter` initializes the printer. Opens the device context `DI.dc`.
- The function `ExitPrinter` closes the device context.
- The function `PrintHeader` prints the header of the selected page layout.
- The function `PrintFooter` prints the bottom of the selected page layout.
- The function `NextPage` prints the bottom of the selected page layout and forces a new page.
- Important print variables of the structure `DruckInfo`:
  `HDC dc` – Device context of the printer.
  `RECT reData` – Rectangle dimension for the data output. The rectangle `reData` contains the valid rectangle for the output to the printer.

5.4 Using global strings or variables with DASYLab

DASYLab supports since version 3.0 the use of global strings or variables in processing and display modules. A global string i.e. in the *Lamp* example displays instead a normal string the string stored under the particular string number. The syntax which has to be used for a global string is `${STR_x}.x` is the number of the string (in range 1-999). As string you can define variables as `${VAR_x}`. Since version 5.0 of DASYLab system strings and variables are supported. They can be used as global variables and strings, but their values are changed by the DASYLab kernel.

There are two variable management possibilities:

- Retrieve actual variable/string at every time it is used
- Register/Unregister used variables or strings so that a change causes a message, the module can process

Example: Global Variables in the *Generator* module

Retrieve the global variable if the syntax fits to `${VAR_x}` or to the syntax of a system variable:

```
GetDlgItemText (hDlg, IDD_GEN_AMPL, ShortTempString, 24);
if ( ExpandVarNumber (ShortTempString, 24, &VarNummer) )
   TempInfo->nVarAmplitude[wNummer] = VarNummer;
else
{
   TempInfo->fAmplitude[wNummer] = atof (ShortTempString);
   TempInfo->nVarAmplitude[wNummer] = 0;

}
```

The function `ExpandVarNumber` checks the dialog box item string. If there was added a global variable the return value is larger than 0. In `VarNumber` the referring variable number from 1 to 999 is stored.

Show the global variable in the module dialog box, if used. Otherwise show the amplitude's value:

```
if ( TempInfo->nVarAmplitude[wNummer] == 0 )
   chg_float (ShortTempString, TempInfo->fAmplitude[wNummer], -4);
else
   GlVar_SetSyntax ( ShortTempString, TempInfo->nVarAmplitude[wNummer] );
SetDlgItemText (hDlg, IDD_GEN_AMPL, ShortTempString);
```

The function `GlVar_SetSyntax` displays the variable in the correct syntax.

5.4.1 Useful functions for supporting global variables

5.4.1.1 Function: `GlVar_RegisterByNumber`

Call this function to register the used variable number (1-999). Each time this variable is changed from outside, a message `DMM_GLOBAL_VAR_CHANGED` is sent to the registered module.

5.4.1.2 Function: `GlVar_UnRegisterByNumber`

Call this function to unregister the used variable number. The number of calls for register and unregister have to be equal. So pay attention in a module dialog box. Unregister the variables before the `ChangeModuleSize` function is called and register the variables again before leaving the dialog box in the `IDOK` statement. It is also useful to register all variables at the `START` function again.

5.4.1.3 Function: GlVar_Set, GlVar_Get, GlVar_PrefixGet, GlVar_VarStringGet

Here is a short description of additional functions which can be called. `GlVar_Set` and `GlVar_Get` set or retrieve the value of a particular variable. `GlVar_PrefixGet` gets the prefix string of a special variable number. `GlVar_VarStringGet` gets a variable number out of a string (dialog box string ...).

5.4.1.4 Function: GlVar_SetSyntax

This function is used for setting the correct text in a dialog box, if a global variable or a system variable is used (see example above).

5.4.1.5 Function: SetMenuForGlobalVars

The popup menu of those edit fields in a dialog box where global strings can be entered can be extended by using this function. A menu to select global variables or system variables is added. This function has to be called with the initialization of the dialog `box WM_INITDIALOG`:

```
case WM_INITDIALOG:
      {
         ...
         SetMenuForGlobalVars ( hDlg, IDD_GEN_FREQ );
         SetMenuForGlobalVars ( hDlg, IDD_GEN_AMPL );
         ...
         return (FALSE);
      }
      break;
```

5.4.2 Useful functions for supporting global strings

5.4.2.1 Function: GlStr_RegisterByText

Call this function to register a global string. You can call this function, if a global string is included in the string or not. The function checks if there is a global string is included, and registers the string number in a local stack. Each time the global string is changed from outside, a message `DMM_GLOBAL_STRING_CHANGED` is sent to the registered module.

```
static void RegisterStrVarGenerator ( MODULE * ThisModule )
{
   MODUL_GENERATOR *PrivatInfo = ThisModule->ModuleParameters;
   UINT i;

   for ( i=0; i<ThisModule->wNumOutChan; i++ )
   {
      GlStr_RegisterByText (PrivatInfo->szChannelName[i], ThisModule,
                            HandleAction_GENERATOR );
      if ( PrivatInfo->nVarFrequency[i] > 0 )
         GlVar_RegisterByNumber ( PrivatInfo->nVarFrequency[i], ThisModule,
                                  HandleAction_GENERATOR );
      if ( PrivatInfo->nVarAmplitude[i] > 0 )
         GlVar_RegisterByNumber ( PrivatInfo->nVarAmplitude[i], ThisModule,
                                  HandleAction_GENERATOR );
   }
}
```

Here all important strings are checked of existing global strings. The third parameter of the function contains the address of the `PerformAction` function which retrieves the message `DMM_GLOBAL_STRING_CHANGED`. Look also in the example where this function is called, to avoid multiple registrations without unregistering before.

5.4.2.2 Function: GlStr_UnregisterByText

Call this function to unregister the used global string. The number of calls for register and unregister have to be equal. So pay attention in a module dialog box. Unregister the string before copying the

temporary module variables into the module data. Register the variables again before leaving the dialog box in the IDOK statement. It is also useful to register all variables at the START function again.

```
static void UnregisterStrVarGenerator ( MODULE * ThisModule )
{
   MODUL_GENERATOR *PrivatInfo = ThisModule->ModuleParameters;
   UINT i;

   for ( i=0; i<ThisModule->wNumOutChan; i++ )
   {
      GlStr_UnregisterByText( PrivatInfo->szChannelName[i], ThisModule, HandleAction_GENERATOR );
      if ( PrivatInfo->nVarFrequency[i] > 0 )
         GlVar_UnregisterByNumber( PrivatInfo->nVarFrequency[i], ThisModule,HandleAction_GENERATOR );
      if ( PrivatInfo->nVarAmplitude[i] > 0 )
         GlVar_UnregisterByNumber( PrivatInfo->nVarAmplitude[i], ThisModule,HandleAction_GENERATOR );
   }
}
```

Example (GENERAT.C) dialog box treatment:

```
case IDOK:
{
      SetFocus ( GetDlgItem ( hDlg, IDOK ) );
      ...

      UnregisterStrVarGenerator ( ThisModule );

      /* Copy temporary data to the module parameters */
      memcpy (ThisModule->ModuleParameters, TempInfo, sizeof (MODUL_GENERATOR));

      ChangeModuleSize (ThisModule, 0, wDialNumChan);

      RegisterStrVarGenerator ( ThisModule );
       ...
}
break;
```

5.4.2.3 Function: ExpandString

Before the output of a string which could contain a global string, it is necessary to exchange the global string with the stored string. Use the ExpandString function for easily changing the global string holder.

Example (LAMP.C):

```
ExpandString (PrivatInfo->chTextOff[wChannelNr], sizeof(PrivatInfo->chTextOff[wChannelNr]),
              cBuf, sizeof(cBuf));
Textout( hDC, xpos, ypos, cBuf, strlen (cBuf));
```

Here the text of the *OFF* state is checked for a global string and stored into the string cBuf. The maximum string length of each string has to be included, so that an overwriting behind the end of the string can be avoided.

5.4.2.4 Functions: ExpandVarNumber, GlStr_RegisterByNumber, GlStr_UnregisterByNumber, GlStr_Set, GlStr_Get, GlStr_PrefixGet

The general function is nearly the same as of the global variable functions.
GlStr_RegisterByNumber and GlStr_UnregisterByNumber registers or unregisters the string directly by the string number. GlStr_Set manipulates a global string, GlStr_Get allows to read an actual string by its number. GlStr_PrefixGet calls the prefix text which is displayed in the global string configuration dialog box.

5.4.2.5 Function: GlStr_SetSyntax

This function is used for setting the correct text in a dialog box if a global string or a system string is used (see example above).

5.4.2.6 Function: **SetMenuForGlobalStrings**

The popup menu of those edit fields in a dialog box where global strings can be entered can be extended by using this function. A menu to select global strings or system strings is added. This function has to be called with the initialization of the dialog box WM_INITDIALOG:

```
case WM_INITDIALOG:
{
   ...
   SetMenuForGlobalStrings ( hDlg, IDD_KANAL_NAME );
   ...
   return (FALSE);
}
break;
```

5.5 The GDI Stack

DASYLab provides several GDI register and unregister functions. Normally, a device is allocated as often as it is requested. Although it could be the same type, i.e. a red brush. So because the Windows operating system offers a limited device count multiple requests of same devices should be avoided. Therefore DASYLab offers a GDI management. DASYLab looks if i.e. a red brush is allocated. If not the handle is requested from Windows, otherwise a counter is increased. The user has to pay attention, to release as often the device handle as it was requested.

5.5.1 Function: CreateStackedPen

Call this function to receive a pen handle by the DASYLab GDI stack management.

5.5.2 Function: CreateStackedPenIndirect

Call this function to receive a pen handle by the DASYLab GDI stack management. As parameter the LPLOGPEN structure is used. It is defined as

```
typedef struct tagLOGPEN {  /* lgpn */
    UINT     lopnStyle;
    POINT    lopnWidth;
    COLORREF lopnColor;
} LOGPEN;
```

5.5.3 Function: CreateStackedFontIndirect

Call this function to receive a font handle by the DASYLab GDI stack management. As parameter the LPLOGFONT structure is used. It is defined as:

```
typedef struct tagLOGFONT {     /* lf */
    int   lfHeight;
    int   lfWidth;
    int   lfEscapement;
    int   lfOrientation;
    int   lfWeight;
    BYTE  lfItalic;
    BYTE  lfUnderline;
    BYTE  lfStrikeOut;
    BYTE  lfCharSet;
    BYTE  lfOutPrecision;
    BYTE  lfClipPrecision;
    BYTE  lfQuality;
    BYTE  lfPitchAndFamily;
    BYTE  lfFaceName[LF_FACESIZE];
} LOGFONT;
```

5.5.4 Function: CreateStackedSolidBrush

Call this function to receive a solid brush handle by the DASYLab GDI stack management. As parameter the COLORREF structure is used.

5.5.5 Function: CreateStackedBrushIndirect

Call this function to receive any brush handle by the DASYLab GDI stack management. As parameter the `LPLOGBRUSH` structure is used. It is defined as:

```
typedef struct tagLOGBRUSH {    /* lb */
    UINT     lbStyle;
    COLORREF lbColor;
    int      lbHatch;
} LOGBRUSH;
```

5.5.6 Function: DeleteStackedObject

After using a device handle it has to be unregistered by the DASYLab GDI stack. So call this function before requesting i.e. a changed pen color or at the `DELETE` function, when the module is destroyed.

5.5.7 New Extra Data Transport API

For a later DASYLab release (after DASYLab 2016), we provide a new API for data transport at start time (we call this static) and data transport with the data blocks at process time (we call that dynamic). A first version of this API is included in the toolkit, but all functions (except one) are subject to change. To provide compatibility with later DASYLab releases (as far as we can look into the future) add a call of `EmemBlock_PROCESS_MsgCopyPlain` before the call of `ReleaseOutputBlock` in your `ProcessData` function:

```
// TK2016: For future compatibility: Copy extra memory (per data block) from the "Father" block
//         to the block we filled with data above
//   - This will have no effect in DASYLab 2016 but your module/dll will support a new feature in
//     DASYLab 2017 without recompiling
//   - The cpu cost for DASYLab 14 is near zero - the call of this funtion is optional for DASYLab 2016
//     and mandatory for the following DASYLab versions.
//
// If the InFifo parameter is NULL, the function assumes, that the maximum blocksize of the input fifo
// is equal to the maximum blocksize of the output fifo (what is "the normal behaviour") .
// If you are unsure (what you shouldn't be because you wrote/modified the
// SetupFifo_xxx routine) or the maximum blocksize of OutFifo and InFifo is not equal, then provide
// the InFifo as parameter.

EmemBlock_PROCESS_MsgCopyPlain (OutFifo, OutputBlock, NULL, InputBlock);
// Add this Data Block to the FIFO, so that a "Son" FIFO can get Access to it
ReleaseOutputBlock (OutFifo);
```

6 32 Bit and Versions

6.1 File dialogs and file names

6.1.1 File Names

DASYLab *2016* uses long file names. So we recommend a string length of at least 256 characters.

6.2 Worksheets in ASCII format

6.2.1 Background

Since DASYLab was also available as 32-bit version it was impossible to load 16-bit worksheets with the 32-bit DASYLab version and vice versa. Also loading a worksheet with an older version of DASYLab is not possible. The solution for this problem is to save and load worksheets as ASCII text files. As described in chapter 5.3.1.1 the structure `PARAMETER_INFO` is necessary for that purpose.

Saving worksheets in a text format can be used for documentation purposes also.

6.2.2 Structure PARAMETER_INFO

6.2.2.1 General Description

The `PARAMETER_INFO` structure is used to describe the module's parameters in such a manner that each variable, its type and its location in the corresponding structure are specified. The `PARAMETER_INFO` structure is defined as follows:

```
typedef struct
{
   char     *szDescription;
   char     *szType;
   size_t   nOffset;
   size_t   nStructOffset;
} PARAMETER_INFO;
```

In the following description of the structure, elements enclosed in *{curly brackets}* are *optional*:

`szDescription`: Text to describe the parameter. It can have the following contents:
`{*}Parameter_Description_Text{[0..X1]}`
`X1`: Maximum number of channels of the module.
A leading asterisk (`*`) means, that this parameter cannot be set in the dialog box. Parameters marked with `*` will not be saved in a text documentation worksheet.

`szType`: Text that describes the type of the parameter:
`Type{[0..X2]}{:Y1=DESCRIPTION1, Y2=DESCRIPTION2, ..., Yn=DESCRIPTIONn}`
`X2`: Maximum number of this parameter.
`Y1...Yn`: Values this parameter can have (not allowed for *decimal*, *string* or *struct* parameters).
`DESCRIPTION1...DESCRIPTIONn`: Description of the values this parameter can have (not allowed for *decimal*, *string* or *struct* parameters).
See chapter 6.2.2.3 for supported types.

`nOffset`: Offset of the starting address of this parameter in bytes within the parameter structure. Use the macro `offsetof (STRUCTURE, PARAMETER)` defined in header file `STDDEF.H` to calculate this value.

`nStructOffset`: Sometimes an array of another structure is used within the parameter structure. The value `nStructOffset` is the size of this structure. Normally this value would be 0.

At the end of the initialization all elements have to be set to `NULL` or `0`.

6.2.2.2 Example

Now we shall discuss an example of the `PARAMETER_INFO` structure for a better understanding:

```
...
#include <stddef.h>                                        /* Header for the offsetof macro */
...
#define MAX_STR_LEN 24
...
// Structures used in the module's parameter structure
typedef struct
{
   int nValue;
   double fValue;
} INT_DOUBLE;

typedef struct
{
   double fValue[MAX_CHANNEL];
   int nValue[MAX_CHANNEL];
} DOUBLE_INT;
...
typedef struct
{
   // Channel Name
   char  szChannelName[MAX_CHANNEL][MAX_STR_LEN];
   UINT  wFunction[MAX_CHANNEL];                           // Function: 0=FUCNTION1, 1=FUNCTION2, 2=FUNCTION3
   char  szUnit[MAX_CHANNEL][MAX_UNIT_LEN];                // Unit
   BOOL  bCopyChannelName[MAX_CHANNEL];                    // Copy channel names to the output
   WORD  wNewArray[9];                                     // 9 element array of values type WORD
   char  szNewArray[9][124];                               // 9 element array of strings with 124 characters
   INT_DOUBLE  IntDouble[MAX_CHANNEL];                     // Structure with MAX_CHANNEL elements
   DOUBLE_INT  DoubleInt;                                  // Structure with MAX_CHANNEL elements
} MODULE_EXAMPLE;

static PARAMETER_INFO FAR ParameterExample[] =
{
   // Using strings
   { "Channel_Name[0..15]", "string[24]", offsetof ( MODULE_EXAMPLE, szChannelName[0][0] ), 0 },

   // Using descriptions
   { "Function[0..15]", "UINT:0=FUNCTION1,1=FUNCTION2,2=FUNCTION3", offsetof ( MODULE_EXAMPLE, wFunction[0] ), 0 },
   { "Unit[0..15]", "string[64]", offsetof ( MODULE_EXAMPLE, szUnit[0][0] ), 0 },
   { "Copy_Channel_Name[0..15]", "BOOL", offsetof ( MODULE_EXAMPLE, bCopyChannelName[0] ), 0 },

   // Using an array which does not correspond to the number of channels
   { "Word_Array", "WORD[0..8]", offsetof ( MODULE_EXAMPLE, wNewArray[0] ), 0 },

   // Array of strings which does not correspond to the number of channels
   { "String_Array", "string[124][0..8]", offsetof ( MODULE_EXAMPLE, szNewArray[0][0] ), 0 },
   // Using a struct where we have to use nStructOffset
   { "Int_Double_nValue[0..15]", "int", offsetof ( MODULE_EXAMPLE, IntDouble[0].nValue ), sizeof ( INT_DOUBLE ) },
   { "Int_Double_fValue[0..15]", "double", offsetof ( MODULE_EXAMPLE, IntDouble[0].fValue ), sizeof ( INT_DOUBLE ) },

   // Using a struct where we don’t have to use nStructOffset
   { "Double_Int_nValue[0..15]", "int", offsetof ( MODULE_EXAMPLE, DoubleInt.nValue[0] ), 0 },
   { "Double_Int_fValue[0..15]", "double", offsetof ( MODULE_EXAMPLE, DoubleInt.fValue[0] ), 0 },

   // Never forget this line !!!
   { NULL, NULL, 0, 0 }
};
```

This example shows the different cases which can occur in a module's data structure:

1. The variable `wFunction` can have the values 0, 1 or 2. Each value specifies a specific function. These functions are named `FUNCTION1`, `FUNCTION2` and `FUNCTION3`. Look in `GENERATOR.C` for an example.

2. The variables `wNewArray` and `szNewArray` show how to handle arrays which do not correspond to the maximum number of channels of the module. The maximum array index has to be set in the `szType` parameter.

3. The variables `IntDouble` and `DoubleInt` explain the use of the `nStructOffset` parameter: The addresses of the elements of `DoubleInt` can be accessed directly, because each element is stored after the other, but the elements of `IntDouble` have an offset of the structure's size.

6.2.2.3 Supported Types

The `szType` parameter in the `PARAMETER_INFO` structure describes the type of parameter. The following table shows which types can be used:

Type	C/C++/MFC equivalent
"string"	Char[x] (array)
"lpstring"	Char*
"char"	Char
"BOOL"	BOOL
"UINT"	UINT
"UINT32"	UINT32
"WORD"	WORD
"DWORD"	DWORD
"DWORD32"	DWORD32
"unsigned long"	Unsigned long
"unsigned long int"	Unsigned long int
"long"	Long
"long int"	Long int
"signed long int"	Signed long int
"unsigned short"	Unsigned short
"unsigned short int"	Unsigned short int
"short"	Short
"short int"	Short int
"signed short int"	Signed short int
"int"	Int
"signed"	Signed
"signed int"	Signed int

Type	C/C++/MFC equivalent
"char"	Char
"signed char"	Signed char
"unsigned char"	Unsigned char
"COLORREF"	COLORREF
"time_t"	Time_t
"DLAB_FLOAT"	see types.h
"float"	Float
"double"	Double
"long double"	Long double
"lpstring"	Char*
"LOGFONT"	LOGFONT
"POINT"	POINT
"RECT"	RECT
"MODULE_WND_POS"	see types.h

6.3 Multi-threading

If the experiment is running, the calling of the `ProcessData` function for each module is running in its own thread. This has some effects: e.g. handles which are received in the `START` function may be invalid in the `ProcessData` function.

For more information see the multi-threading programming tips published by Microsoft.

7 Module Independent Dialog Box

It is possible to generate a module independent dialog box. In this dialog box the user could make some general settings, which do not belong to a special module but for example to the connected hardware. The box could be opened to setup, e.g. address and interrupt. The necessary extensions are described in following parts.

7.1 Toolkit Menu

Add to your existing menu in the RC file the new entry for the setup box.

```
UX1_MENU MENU DISCARDABLE
BEGIN
    POPUP "&New"
    BEGIN
        MENUITEM "&Global Setup",              MN_UX_SETUP
```

The menu ID `MN_UX_SETUP` has to be defined in particular ranges. The definitions are shown in the file `CONST.H` and have to be between 4990 and 4999. So at least 10 menu entries are possible for each DLL. Every DLL has to use these IDs. DASYLab remaps the IDs to distinguish between every DLL call.

```
[CONST.H]
...
#define MN_FIRST_EXT_MENU_ID       4990
#define MN_LAST_EXT_MENU_ID        4999
...

[DLAB_UX1.H]
...
#define MN_UX_SETUP                4990
...
```

7.2 Register Menu Callback Function

The menu entries are prepared in such a way that DASYLab has to know which function should be called if the user selects the particular menu ID. Therefore you have to register a callback function during the initialization. So include the registration function in the `INIT_DLL` function. We have added it in the `ExpandModuleBar` function:

```
static void ExpandModuleBar ( void )
{
   DASYLAB_INSERT_MENU DlabMenuStruct;
   ...

   DlabMenuStruct.uiNewMenuID = MN_UX_SETUP;
   if ( ExpandDASYLabMenu ( &DlabMenuStruct ) == TRUE )
   {
      // Register the inserted menu item as it is not a module. (Modules are registered
      // calling: RegisterModulClass(&mc)).
      RegisterMenuEntry ( DlabMenuStruct.uiNewMenuID , &SetupProc );
   }
}
```

The function `SetupProc` has to be a declared as: `void SetupProc(void)`.

For the structure `DASYLAB_INSERT_MENU` refer to chapter 5.2.

8 Layout / VI Tool Connections

One of the most new features is the new “DASYLab Layouter” called *VITool*. The user can build his special print layout or working desk. Therefore he has to generate connections between the display modules and the layout window. Look for detailed description and functionality of the *VITool* in the DASYLab manual.

The *VITool* connection is managed with the help of the DASYLab message concept. The display module has to register the support of a layout connection. All functions for adding a new connection, updating the complete drawing rectangle and deleting the connection are implemented as message events inside the `Perform_Action` message loop.

8.1 Layout example: LAMP.C

8.1.1 Processing the **DMM_QUERY_PANEL** message

DASYLab asks after creating a new module which panel connections are supported by this module. Therefore the module has to return with an OR-operation added panel flags:

- `PT_METAFILE`
  The layout object is send as a metafile. This is a slow drawing method, which should be used for drawings which do not often change. There are also a good solution for exporting the layout to other drawing programs.

- `PT_PAINT`
  This is a drawing mode, where the drawing happens on a virtual coordinate system and is stretched into the layout rectangle. So, it could happen that, for example, some drawing lines are hidden.

- `PT_PAINT_SCALED`
  In this drawing mode the display module get the drawing rectangle in its original size. This mode is fine for detailed drawings like the Y/t Chart module, or the Chart Recorder module where i.e. scrolling has to be done.

- `PT_TEXT`
  This drawing mode is only for the output of single data values. The *VITool* handles the graphical output itself and gets only the value to be shown.

- `PT_WINDOW`
  This drawing mode is used to include control windows as a Windows slider or control button. The *VITool* creates therefore a window inside the link rectangle where the control can be placed on.

The `Lamp.c` example supports only the metafile and paint mode.

```
case DMM_QUERY_PANEL:
      /* Set supported drawing methods */
      wParam &= ( PT_METAFILE | PT_PAINT );
      return wParam;
```

8.1.2 Processing the **DMM_PANEL_CONNECT** message

This message is received if a user wants to insert a *VITool* link. Now the module has to register the connection and setup the connection variables and has to check if a new link is possible, or all possible copies are in use.

```
case DMM_PANEL_CONNECT:
   /* active connection */
   return AddPanelConnection ( ThisModule, wParam, (HPANEL) lParam );
[...]
static int AddPanelConnection ( MODULE *ThisModule, int iType, HPANEL hPanel )
{
   VAR_LAMPE *PrivatVars = ThisModule->TempModuleData;
   int i;

   for ( i=0; i<MAX_PANEL_CONNECTIONS; i++ )
   {
      /* Is there a new connection possible */
      if ( PrivatVars->PanelInfo[i].hPanel == 0 )
      {
         /* Then store it here */
         PrivatVars->PanelInfo[i].hPanel = hPanel;
         PrivatVars->PanelInfo[i].iType = iType;
         return i;
      }
   }

   return -1;   /* No space, no connection, what a pitty */
}
```

8.1.3 Processing the **DMM_PANEL_SET_SIZE** message

After installing the *VITool* connection, the module gets the `DMM_PANEL_SET_SIZE` message. This message occurs also after every change in the size of the rectangle in the *VITool*. Store the size of the output rectangle into the temporary module data, to calculate the display dimensions before drawing.

```
case DMM_PANEL_SET_SIZE:
         /* Size of layout rectangle has been changed */
         SetPanelSize ( ThisModule, wParam, (SIZE FAR *) lParam );
         return TRUE;
[...]
static void SetPanelSize ( MODULE *ThisModule, int PanelIndex, SIZE *lpsize )
{
   VAR_LAMPE *PrivatVars = ThisModule->TempModuleData;

   if ( PanelIndex < 0 || PanelIndex >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[PanelIndex].hPanel == 0 )
      return;

   /* Store the size into the private data */
   PrivatVars->PanelInfo[PanelIndex].sz.cx = lpsize->cx;
   PrivatVars->PanelInfo[PanelIndex].sz.cy = lpsize->cy;
}
```

8.1.4 Processing the **DMM_PANEL_REQUEST_METAFILE** message

If the link is build up for metafile (slow) method, the `DMM_PANEL_REQUEST` is sent out to update the metafile in the *VITool*.

```
case DMM_PANEL_REQUEST_METAFILE:
         /* send new meta file to layout */
         return (unsigned long) (UINT) GetPanelMetaFile ( ThisModule, wParam,
                                          (SIZE FAR *) lParam );
```

Now the module has to send back the metafile handle. Look for detailed descriptions of creating a metafile to the `Lamp.c` example and in any Windows programming manual.

8.1.5 Processing the **DMM_PANEL_PERFORM_DRAW** message

After every redraw of the *VITool* each layout connection has to redraw its contents. So the DMM_PANEL_PERFORM_DRAW message is sent to every module with a PT_PAINT or PT_PAINT_SCALED mode connection. The lParam contains the necessary device context handle (HDC).

```
case DMM_PANEL_PERFORM_DRAW:
        /* completly redraw */
        DrawPanelComplete ( ThisModule, wParam, (HDC)(UINT) lParam );
        return TRUE;
```

The DrawPanelCommplete function should operate as implemented in the Lamp.c example:

- Store window dimensions in temporary data
- Check the output mode (Metafile, Screen, Plotter,…) to do some special setups
- Resize all output dimensions concerning to the *VITool* rectangle
- Recalculate all used font sizes and request new font handles
- Make the output into the *VITool* device context
- Restore the old dimensions regarding the display window
- Restore old font handles

8.1.6 Processing the **DMM_PANEL_DRAW_NEW_DATA** message

This message occurs only by the display module itself. If the output display (here the state of the lamp) has changed the module has to inform the *VITool*, that a new update of the rectangle has to be done. To avoid a completely redraw, it is better to redraw only the parts that have to change (Lamp module: state, Y/t Chart: new curves,...). So, send after a window update the message DMM_PANEL_GOT_NEW_DATA.

```
for ( i=0; i<MAX_PANEL_CONNECTIONS; i++ )
{
   if ( PrivatVars->PanelInfo[i].hPanel != 0 )
            SendPanelMessage ( PrivatVars->PanelInfo[i].hPanel, DPM_PANEL_GOT_NEW_DATA, TRUE, 1 );
}
```

For parameter #3 and #4 as shown for synchronous layout connection, use FALSE and 0 for asynchronous connection!

Now the layout calls back with the DMM_PANEL_DRAW_NEW_DATA. Now, redraw only the changed parts of the *VITool* rectangle, to avoid flickering effects (see also the example Lamp.c).

8.1.7 Processing the **DMM_PANEL_DISCONNECT/DMM_PANEL_DISCONNECT_ALL** message

When a module connection is deleted, and when the worksheet gets destroyed, all connections have to be unregistered first. So, the *VITool* sends out the DMM_PANEL_DISCONNECT message to delete a single connection or the DMM_PANEL_DISCONNECT_ALL message to delete all registered connections.

```
case DMM_PANEL_DISCONNECT:
   /* end of single layout connection, so disconnect it please */
   RemovePanelConnection ( ThisModule, wParam );
   return TRUE;

case DMM_PANEL_DISCONNECT_ALL:
   /* end of ALL layout connection, so disconnect them please */
   {
      int i;

      for ( i=0; i<MAX_PANEL_CONNECTIONS; i++ )
         RemovePanelConnection ( ThisModule, i );
   }
   return TRUE;
```

```
static void RemovePanelConnection ( MODULE *ThisModule, int PanelIndex )
{
   VAR_LAMPE *PrivatVars = ThisModule->TempModuleData;
   HPANEL hPanelSave;

   if ( PanelIndex < 0 || PanelIndex >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[PanelIndex].hPanel == 0 )
      return;

   hPanelSave = PrivatVars->PanelInfo[PanelIndex].hPanel;
   memset ( &PrivatVars->PanelInfo[PanelIndex], 0,
            sizeof(PrivatVars->PanelInfo[PanelIndex]) );

   SendPanelMessage ( hPanelSave, DPM_PANEL_DISCONNECT, 0, 0 );
}
```

8.1.8 Processing the **DMM_PANEL_WM_XXXX** mouse messages

Mouse messages can be operated from a *VITool* object. Therefore the *VITool* sends all mouse events to the corresponding modules. The messages which can be processed are:

- DMM_PANEL_WM_LBUTTONDOWN
- DMM_PANEL_WM_LBUTTONUP
- DMM_PANEL_WM_LBUTTONDBLCLK
- DMM_PANEL_WM_RBUTTONDOWN
- DMM_PANEL_WM_RBUTTONUP
- DMM_PANEL_WM_RBUTTONDBLCLK
- DMM_PANEL_WM_MOUSEMOVE

8.1.9 Drawing method *PT_PAINT_SCALED*

In addition to the PT_PAINT mode there are only few differences in treatments. The size function gets a different parameter structure.

```
case DMM_PANEL_SET_SIZE:
         SetPanelSize ( ThisModule, wParam, (TWO_SIZES FAR *) lParam );
         return TRUE;
```

The TWO_SIZES structure includes an absolute pixel size and a scaled size. Use the absolute size for all dimension calculations and output operations.

```
static void SetPanelSize ( MODULE *ThisModule, int PanelIndex, TWO_SIZES *lpsize )
{
   VAR_ABC *PrivatVars = ThisModule->TempModuleData;

   if ( PanelIndex < 0 || PanelIndex >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[PanelIndex].hPanel == 0 )
      return;

   PrivatVars->PanelInfo[PanelIndex].szAbsolute.cx = lpsize->absolute.cx;
   PrivatVars->PanelInfo[PanelIndex].szAbsolute.cy = lpsize->absolute.cy;
   PrivatVars->PanelInfo[PanelIndex].szScaled.cx = lpsize->scaled.cx;
   PrivatVars->PanelInfo[PanelIndex].szScaled.cy = lpsize->scaled.cy;
}
```

8.1.10 Drawing method *PT_TEXT*

In contrast to making the display output by the corresponding module, the PT_TEXT mode offers the output control to the *VITool*. Be aware of the fact, that only strings can be displayed! The message the module has to work on is DMM_PANEL_REQUEST_STRING.

```
case DMM_PANEL_REQUEST_STRING:
         /* Aktuellen Wert zurückliefern */
         return (unsigned long) GetPanelString ( ThisModule, wParam );
```

The GetPanelString functions builds up the string to be displayed. Use a static string which should be displayed by the *VITool*.

```
static char *GetPanelString ( MODULE *ThisModule, int PanelIndex )
{
   MODUL_TEST *PrivatInfo = ThisModule->ModuleParameters;
   VAR_TEST *PrivatVars = ThisModule->TempModuleData;
   static char FAR text[40];

   if ( PanelIndex < 0 || PanelIndex >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[PanelIndex].hPanel == 0 )
      return NULL;

   float2str ( PrivatInfo->wCharoBeDisplayed, PrivatInfo->wPoints,
               PrivatVars->fDataValue[0], ID_KONV_NRM, text);

   return text;
}
```

8.1.11 Drawing method *PT_WINDOW*

To include Windows controls use the `PT_WINDOW` mode, when registering the supported layout modes.

```
/* Communication to VITool */
case DMM_QUERY_PANEL:
   {
      wParam &= (PT_WINDOW);
   }
   return wParam;
```

When adding a new layout link a windows handle created by the *VITool*. On the message `DMM_PANEL_SET_WINDOW` the new windows handle is send to the user extension module. Use this window to add on your control.

```
case DMM_PANEL_SET_WINDOW:
   SetPanelWindow ( ThisModule, wParam, (HWND) (UINT)lParam );
   return TRUE;
```

The function `SetPanelWindow` members the *VITool* window handle. The `SetWindowLong` function places a long value at the specified offset into the extra window memory of the given window. Extra window memory is reserved by specifying a nonzero value in the `cbWndExtra` member of the `WNDCLASS` structure used with the `RegisterClass` function. The placed long value has to be interpreted as the module pointer and the layout link number. With the help of this variables the layout can send the messages to the correct module and link ID.

```
void SetPanelWindow(MODULE *ThisModule, int PanelIndex, HWND hParentWnd)
{
   VAR_CONTROL *PrivatVars = ThisModule->TempModuleData;

   if ( PanelIndex < 0 || PanelIndex >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[PanelIndex].hPanel == 0 )
      return;

   /* Parent window to place on the control */
   PrivatVars->PanelInfo[PanelIndex].hParentWnd = hParentWnd;

   /* Add the module pointer and index to have the link to the correct module */
   SetWindowLong (hParentWnd, 0, (long)ThisModule);
   SetWindowLong (hParentWnd, 4, (long)PanelIndex);
   return;
}
```

Add now the controls to the parent window:

```
void AddPanelControls( HWND hParent, MODULE *ThisModule )
{
   MODUL_CONTROL *PrivatInfo = ThisModule->ModuleParameters;
   VAR_CONTROL *PrivatVars = ThisModule->TempModuleData;
   int         wPanelId = (int)GetWindowLong (hParent, 4);
   UINT        i;

   if (wPanelId < 0 || wPanelId >= MAX_PANEL_CONNECTIONS ||
        PrivatVars->PanelInfo[wPanelId].hPanel == 0 )
      return;

   /* Add controls */
   for (i=0; i<ThisModule->wNumOutChan; i++)
      {
         PrivatVars->PanelInfo[wPanelId].hCombo[i] = CreateWindow("COMBOBOX",
            NULL,
            WS_CHILD | WS_VISIBLE | CBS_DROPDOWNLIST | WS_VSCROLL | WS_TABSTOP,
            PrivatVars->PanelInfo[wPanelId].reChan[i].left,
            PrivatVars->PanelInfo[wPanelId].reChan[i].top,
            PrivatVars->PanelInfo[wPanelId].reChan[i].right,
            PrivatVars->PanelInfo[wPanelId].reChan[i].bottom,
            14,
            hParent,
            (HMENU)(ID_COMBO + i), hInst, NULL);
      }
   }
   return;
}
```

9 Multiple Time Bases in DASYLab

9.1 Background

Over the years, many device drivers for DASYLab have been developed that do not use the standard driver interface but a special DLL. Especially, these drivers can define their own sample rate and block size settings, so that multiple data streams with different timing settings can exist within a DASYLab worksheet.

In DASYLab, a central administration tool for all these timing settings has been created. Drivers that create their own timing, can register a time base structure within DASYLab. DASYLab offers a central tabbed dialog box where the user can set up all time base information. When this information changes, the driver is informed by a callback function.

Also means have been provided for other data generating modules to synchronize to any of the time bases thus defined. Up to now, this was only possible by using a synchronization input in those modules. This method still exists for compatibility reasons, but is now necessary only in rare circumstances.

9.2 Time base identifiers

Each time base in the system is identified by a unique identifier, which is a natural number in the range of 1 to 999. Note that IDs 1 to 499, and 800 to 849, are reserved by *National Instruments* for internal use.

To maintain uniqueness also among distributors of different drivers, new IDs will be assigned by *measX GmbH & Co. KG*. If you plan to use a time base of your own, please contact

measX GmbH & Co. KG
Trompeterallee110
D–41189 Mönchengladbach, Germany

to receive an unused ID.

9.3 Using a time base from a driver's view

9.3.1 Registering and unregistering

A driver that wants to use a time base of its own, must first register this time base with the `RegisterTimeBase` function. With this registration following information will be made known to DASYLab:

- The unique time base identifier.
- A time base name. This will appear on the tab in the tabbed time base dialog.
- A time base description. This will appear in the information field of the time base dialog.
- A callback function which informs the driver when the time base settings have been changed by the user.

At the end of its life time, the time base should be unregistered with `UnregisterTimeBase`. Note that all registered time bases will be unregistered when DASYLab ends. However, it does not harm if the driver does it itself.

9.3.2 Setting the time base information

After registration, the driver should initialize the time base data using the `SetTimeBase` function. This function uses an `EXT_TIMEBASE` data structure to communicate with the time base administration. The driver can use this function whenever the information for the user interface has to be updated.

On the contrary, if the user changes the time base settings in the dialog, the callback function provided in the registration is called to inform the driver about the changes. This function also uses an `EXT_TIMEBASE` structure for transfer.

Note that the central time base administration will automatically handle global variable settings and automatic block size. At the start of the experiment, these are evaluated for each time base and the driver is informed about the actual settings of block size and sample distance by the callback function. Normally the driver has no action to take to evaluate this information itself.

9.3.3 Updating the time base time

When the experiment is running, the driver has to periodically update the time information for its time base according to the data released by its hardware. This information is retrieved by other modules which synchronize to this time base in order to release their output blocks at certain times.

At experiment start, the time is automatically reset to zero for all time bases. To update the time, two functions are available: `SetTimeBaseTime` will set the time to the value given, and `IncTimeBaseTime` will increment the time by the amount given. The driver can choose either method for updating.

9.3.4 Calling the time base dialog

It might be interesting for the driver to provide a means to set up its time base by the user. Instead of creating a dialog of its own, it can call the central DASYLab time base dialog with the tab of its own activated. Use the `TimeBaseDialog` function for this purpose.

9.4 Using a time base from a module's view

9.4.1 Synchronizing to an existing time base

A module that wants to synchronize to a time base can select from the list of all time bases available. It can use a Windows *Combo Box* to do this selection. The *Combo Box* must be of type *Dropdown list* with the *Sort* option switched off. Two functions are available to support the selection mechanism: `FillTimeBaseCombo` will fill the *Combo Box* with the list of all available time bases and highlight the item currently selected. `GetTimeBaseComboID` can be used to retrieve the time base ID from the current *Combo Box* selection index.

9.4.2 Setting up a module's output parameters

Once a time base is selected, the `GetTimeBaseSampleDistance` and `GetTimeBaseBlockSize` functions can be used to retrieve the basic information about the time base. This information can be used to set up the module's output parameters like FIFO settings. In former applications, this information (for the only time base existing: the global one) was available through the global variables `uiGlobalBlockSize` and `fGlobalMilliSecondsPerSample`. Note that new applications should always use the time base functions instead.

9.4.3 Retrieving the actual time

During an experiment, the module can retrieve the time of the time base with `GetTimeBaseTime`. This information replaces the one formerly retrieved by `CurrentExperimentTime(1)`.

10 More examples

For each base type of module we have an example included:

Data source module:	`GENERAT.C`
contents:	- generation of data - using global variables and strings - handling of asynchronous actions - use of a selectable time base
Data processing module:	`DERIV.C`
contents:	- processing and calculating the data - handling of synchronous actions
Data processing module:	`DMD_TRIG.C`
contents:	- processing and calculating the data
Data sink module:	`LAMP.C`
contents:	- displaying data - using global strings - processing window messages - *VITool* functionality - handling of synchronous actions

To create your own modules have a look to these examples.

11 DASYLab's Data Structures

The following describes the constants and data structures found in the included headers which may be referred to in user created module classes.

Several constants, variables, types, and structure elements are to be used by the DASYLab kernel only, and are therefore not documented here. We reserve the right to change these any time without prior notice. The only constants, variables, types, and structuere elements that are allowed to be used are the ones described in this document.

11.1 General constants

Maximum values for several purposes:

`MAX_CHANNEL`	The maximum number of input or output channels a module can have.
`CHANNEL_NAME_LENGTH`	The maximum number of characters for a channel name.
`MODULE_NAME_LENGTH`	The maximum number of characters for a module name.
`MODULE_STATUS_LENGTH`	The maximum number of characters for the status line description text for a module class.
`MODULE_DESCRIPTION_LENGTH`	The maximum number of characters for a modules description.
`MAX_PATH`	The maximum number of characters for a path name.
`MAX_FILENAME`	The maximum number of characters for a file name.
`MAX_SYNC_ACTIONS`	The maximum number of synchronous actions for each module.
`MAX_PANEL_CONNECTIONS`	The maximum number of panel connections for each module.
`MAX_UNIT_LEN`	The maximum number of characters for units.

11.2 Internal representation of data

`DLAB_FLOAT`	Floating point type used internally inside DASYLab. Since DASYLab 2016, we use `double`, before we used `float`. Do not use the `float` type in your module classes!
`DLAB_FLOAT_MAX`	Largest number which can be represented in the `DLAB_FLOAT` type.
`DLAB_FLOAT_MIN`	Smallest positive number which can be represented in the `DLAB_FLOAT` type.
`DLAB_FLOAT_EPSILON`	Smallest number greater than 1.0 which can be represented in the `DLAB_FLOAT` type minus 1.0.
`ALMOST_ZERO`	A positive number smaller than `DLAB_FLOAT_MIN` and thus equal to zero in the `DLAB_FLOAT` representation.

11.3 The `MODCLASS` Type

The `MODCLASS` type describes one module class. It is used in the `init` function of a module class only and will not be needed later on. `MODCLASS` is a structure containing the following elements:

`hInst`	Instance handle of the DLL containing this module class.
`Name`	Internal name of the module class. You must follow the naming convention found on page 108. This is a language-independent name. If you prepare different versions of your DLL for different countries, make sure that this string is the same for all languages.

DataSize	Size of the data ThisModule->ModuleParameters points to. This structure will be saved en-bloc with the worksheet and should contain only parameters of the module, but no intermediate data, pointers, handles and the like.
VarSize	Size of the data ThisModule->TempModuleData points to. This structure will not be saved with the worksheet and should contain module-specific intermediate data, pointers, handles and the like.
ChannelSize	Size of the data ThisModule->TempChannelData[chan] points to. This structure will not be saved with the worksheet and should contain channel-specific intermediate data, pointers, handles and the like.
MenuId	ID in the *SubMenu* that this DLL installs. Must be in the range from 2950 to 2974. DASYLab will map this to a different code, so there are no conflicts when different DLL's use the same *MenuId*.
IdString	Pointer to a string describing the module's default name. This is a language-specific name. You will have different names here if you prepare versions of your DLL for different languages.
StatusString	Pointer to a string describing the module's purpose. Currently this is not used, but DASYLab may use this string in the future to display a message in the status bar while the module is active in the menu bar. This is a language-specific string. You will have different texts here if you prepare versions of your DLL for different languages.
HelpId	*HelpId* in your help file that denotes the entry point for the help page for this module class. Currently not used by DASYLab, but DASYLab may use this constant someday to call the help system for your module.
HelpFileName	Name of the help file that contains the help page for this module class. Use a plain name without a path name here. Currently not used by DASYLab, but in the future DASYLab may use this constant to call the help system for your module.
BBoxId	The ID of the black boxes that may contain this module. Currently the only possible value here is BB_UNIVERSAL.
ModIcon	Icon for use inside the module bar, the bar left to the worksheet desk.
BlkIcon	Icon to use when displaying the module inside a worksheet.
PerformAction	Function to process the messages sent to a module. See the example above for a detailed description.
ProcessData	Function to do the data processing during the experiment. See the example above for a detailed description.
reserved[30]	Reserved space for future extensions. Must be set to zero.

11.4 The MODULE Type

The MODULE type describes one instance of a module class. You will often see pointers like ThisModule pointing to a MODULE type. MODULE is a structure containing elements for use by the DASYLab kernel only as well as elements for general use by module classes. The following elements can be accessed from within module classes:

hwndModule	Window handle of the representation of the module inside the worksheet. You will normally not want to look at this.
hwndWindow	Window handle of an additional window. Modules like the Y/t Chart that open an additional window, must save the window handle of that window here. A module can open only one additional window, but that additional window may have sub-windows.

wModuleNum	Order number of this module relative to the black box it is contained in. You normally don't need to look at this.
iGlobalModuleClass	Order number of this module class this module belongs to. You normally don't need to look at this.
bModuleIsConnected	Flag that is TRUE, if the module has at least one connection to other modules.
hBlockBmp	Bitmap handle copied from the module class. You normally don't need to look at this.
wBmpWidth	Width of the block. You normally will not need to look at this.
wBmpHeight	Height of the block. You normally will not need to look at this.
wXpos	X-Position of the block inside the worksheet. You normally don't need to look at this.
wYpos	Y-Position of the block inside the worksheet. You normally don't need to look at this.
ModuleName	Name of this module. Maximum length is 12 characters plus one trailing zero.
ModuleDescription	Description text for this module. Maximum length is 40 characters plus one trailing zero.
wNumInpChan	The number of input channels for this module.
wNumOutChan	The number of output channels for this module.
ChannelRelation	Describes how the input channels are related to the output channels. Must be set on Creation and Loading of a module. The only legal values are the KZ_* constants listed below.
Fifo[chan]	Pointer to a FIFO_HEADER structure for each output channel.
ModuleParameters	Pointer to a structure containing the module's parameters. This structure will be saved en bloc with the worksheet and should contain only parameters of the module, but no intermediate data, pointers, handles and the like.
TempModuleData	Pointer to a structure containing the module's temporary data. This structure will not be saved with the worksheet and should contain module specific intermediate data, pointers, handles and the like.
TempChannelData[chan]	Pointer to a structure containing each channel's temporary data. This structure will not be saved with the worksheet and should contain channel-specific intermediate data, pointers, handles and the like.

The only legal values for the ChannelRelation field are the following constants:

KZ_NORMAL	'Standard' channel relation. The module can be one of the following three types: a) The module has no outputs, only inputs. b) The module has no inputs, only outputs. c) The module has the same number of inputs and outputs and the first input is related to the first output, the second input is related to the second output and so on.
KZ_2_1	The module has twice as many input channels as output channels. The first two input channels are related to the first output channel, the next two input channels are related to the second output channel and so on.
KZ_1_2	The module has twice as many output channels as input channels. The first two output channels are related to the first input channel, the next two output channels are related to the second input channel and so on.

KZ_2_2	The module has the same number of input and output channels. They belong pair-wise to each other. The first two output channels are related to the first two input channels, the next two output channels are related to the next two input channels and so on.
KZ_1_ALL	The module has any number of input and output channels. All output channels are related to the first input channel.
KZ_ALL_1	The module has any number of input and output channels. All output channels are related to all input channels.
KZ_1_LESS	The module has one more input than output channels. The first output is related to the second input, the second output is related to the third input and so on.
KZ_ALL_ALL	The module has any number of inputs and outputs. The input and output channels don't have a special relationship to each other.

You are not allowed to use the following constants in your modules:

KZ_DELAY	Special. To be used by time delay modules only.
KZ_BBOX	Special. To be used by black box modules only.
KZ_BBOXIO	Special. To be used by black box I/O modules only.
KZ_BBOXSAT	Special. To be used by black box satellite modules only.
KZ_2_1_FIRST	Special. Internal use only.

11.5 The **FIFO_HEADER** Type

The FIFO_HEADER type describes one output FIFO buffer of a module's output channel. FIFO_HEADER is a structure containing elements for use by the DASYLab kernel only as well as elements for general use by module classes. The following elements can be accessed from within module classes:

uiMaxBlockSize	The maximum size a block in this FIFO can have. A data block contained in this FIFO may be shorter that the maximum, but not longer.
fSampleDistance	The (minimum) time in seconds between two samples on this channel. Samples contained inside one data block are always exactly fSampleDistance apart from each other, but it is possible to have holes (or gaps) between the individual data blocks. See the description of the KF_HOLES flag below.
ChannelType	The type of data contained in this channel. Legal values are the KT_* constants listed below.
ChannelFlags	In addition to a data type, some flags may be present. The value may be an OR-ed combination of the KF_* constants listed below.
X_Min	Used for Histogram data only: First range of the data.
X_Max	Used for Histogram data only: Last range of the data.
Unit	The unit of this channel.
ModuleName	Module name.

The only legal values for the ChannelType field are the following constants:

KT_NORMAL	'Standard' data like sampled analog input, digital input, counter input, temperature data etc. Use for all channels that display a time axis in the Y/t Chart.
KT_SPEC	Spectral (FFT) data of full length. These data will display a frequency axis in the Y/t chart.
KT_SPEC2	Spectral (FFT) data of half length. These data will display a frequency axis in the Y/t chart.

KT_SPEC4	Spectral (FFT) data of full length. These data will display a symmetrical frequency axis in the Y/t Chart from -X_Min to +X_Max.
KT_CLASS2	Histogram data. These data will display a range axis in the Y/t Chart.

The constants KT_BINARY, KT_SPEC3, and KT_CLASS, are obsolete and no longer in use.

The constants KT_DIG_WORD, KT_MUX_CHAN, KT_FMUX_CHAN, KT_TRIG, KT_COUNT, KT_UCOUNT, KT_THERMO, and KT_VAR, are used internally by the *DAP DLL* only. You will not see them inside DASYLab*'s* modules and should not use them in your modules.

The only legal values for the ChannelFlags field are OR-ed combinations of the following constants:

KF_NORMAL	No special flags. This value is zero (contains no flags) so it cannot be used for testing. Test with KF_HOLES etc. instead.
KF_HOLES	There may be time gaps (holes) between the individual data blocks on this channel. The data inside one data block however may never contain time gaps.
KF_SHORT_BLK	The data inside one data block can contain the declared maximum block size but can also contain less values than maximum block size. This can occur on triggered data, where the trigger event does not appear for a constant time.

The constant KF_ASYNC may be introduced in the near future as well as others. So when checking flags you should not test equality, but check by AND-ing constants instead.

The constants KF_TRIG_INFO, KF_LONG, and KF_FLOAT are used internally by the *DAP DLL* only. You will not see them inside DASYLab*'s* modules and should not use them in your modules.

The constants KF_COPY_PART, KF_READY, and KF_DELAY are used internally by the DASYLab kernel only.

11.6 The **DATA_BLOCK_HEADER** Type

The DATA_BLOCK_HEADER type describes one block of data contained in some FIFO. DATA_BLOCK_HEADER is a structure containing elements for use by the DASYLab kernel only as well as elements for general use by module classes. The following elements can be accessed from within module classes:

fStartTime	The starting time of this block of data counted in seconds after the experiment has started.
fSampleDistance	The time in seconds between any two samples in this data block. Must exactly match the value given in the FIFO_HEADER.
uiBlockSize	The length (in samples) of this block of data. Must not exceed the value given in the FIFO_HEADER.
Data[]	The data contained in this block. Valid data are in the range from Data[0] to Data[wBlockSize-1].

11.7 The **PARAMETER_INFO** Type

The PARAMETER_INFO type describes the private variables of a module in ASCII-format to store a worksheet as ASCII-file. This structure is returned with the DMM_GET_PARAMETERS_INFO message before loading or saving the module from or to an ASCII-file.

Its elements are as follows:

szDescription	Static description of the variable.
szType	Type of the variable, i.e. int.
nOffset	Offset in the private data structure of the module. Use the macro offsetof to calculate this value.
nStructOffset	Size of a structure in the private data structure of the module.

See chapter *6.2 Worksheets in ASCII format* for detailed information.

11.8 The **EXT_TIMEBASE** Type

The EXT_TIMEBASE type serves as a communication medium between DASYLab's time base administration and the driver that defines the time base. Elements:

uiID	unique identifier of the time base
bAutoBlockSize	Indicates that the block size is computed automatically from the sample rate.
uiBlockSize	Block size used by this time base.
nFreqFormat	Indicator for the display format of the sample rate. Values can be:
ID_TIMEBASE_MHZ	Sample rate in MHz
ID_TIMEBASE_KHZ	Sample rate in kHz.
ID_TIMEBASE_HZ	Sample rate in Hz.
ID_TIMEBASE_MS	Sample distance in msec.
ID_TIMEBASE_SEK	Sample distance in sec.
ID_TIMEBASE_MIN	Sample distance in min.
fSampleDistance	Sample distance in sec used by the time base.
wVarBlockSize	Global variable number for the block size. 0 if not used.
wVarSampleRate	Global variable number for the sample rate. 0 if not used. Note that the variable must contain the sample rate in Hz and not the sample distance in sec.

See chapter "9 Multiple Time Bases in DASYLab" for detailed information.

11.9 Variables

The following variables may be used by user created module classes:

hInst	The instance handle of the DLL that contains the module class. Use this for loading strings from your resource etc.
hInstDlab	The instance handle of the DASYLab program. You will normally not need this.
hBlackPen	A black pen for general use. Try to avoid creating dozens of standard objects over and over again because that would consume lots of resources. Use the standard objects instead.
hWhitePen	A white pen for general use.
hDkGrayPen	A dark gray pen for general use.
hPipePenRed	A red pen for general use.
hPipePenGreen	A green pen for general use.
hLightRedBrush	A light red brush for general use.
hGreenBrush	A green brush for general use.
hDkGrayBrush	A dark gray brush for general use.

wDialCurChan	Dialog box handling: The channel number currently selected.
wDialNumChan	Dialog box handling: The number of activated channels.
wDialMaxChan	Dialog box handling: The maximum number of channels this module can have.
wDialLastFocus	Dialog box handling: The last focused parameter.
SingleInfo	Dialog box handling: Pointer to temporary data.
CurrentModulePtr	Dialog box handling: Pointer to the actual module.
ShortTempString	A short (100 char) string space for general use.
LongTempString	A long (400 char) string space for general use.
fGlobalMilliSecondsPerSample	Sample distance in milliseconds, as specified in the global experiment setup dialog box. Modules like the Generator or Timer use this rate to produce their data.
uiGlobalBlockSize	Data block size, as specified in the global experiment setup dialog box. Modules like the Generator or Timer use this block size to produce their data.
dwExperimentStartTimeTicks	This is the value of `GetTickCount()` that windows reported when the experiment was started. Modules like the Generator or Timer use this as a time base for real time release of the data blocks.
ExperimentStartTime	Date and time when the experiment was started. The time is given in seconds since *Jan 1, 1970, UTC*.
bAnimation	`TRUE`, if the user activated the animation.
bExperimentIsRunning	`TRUE`, while the experiment is running.
bExperimentIsPaused	`TRUE`, while the experiment is paused.
hwndMain	Window handle of the DASYLab main window.
hwndDesk	Window handle of the DASYLab desk window; this is the area where the worksheet gets displayed.
MainMenu	Handle of DASYLab's main menu bar.
MENU_EXPAND_POS	This is the point where user DLLs are allowed to expand DASYLab's menu bar.
wNumberOfModules	Number of modules in the currently active black box.
uiHighModuleNumber	Last module number in use for the currently active black box plus one.
NameOfFlowChart	Plain name (without path) of the worksheet currently loaded.
FullNameOfFlowChart	Full name (including path) of the worksheet currently loaded.
InitialStartupDir	The DASYLab directory.
wxScreen	The screen width in effect.
wyScreen	The screen height in effect.
GlobalFlowChartInfo	The information the user entered in the info dialog box.
bDateDayLeadingZero	Date day format with leading zero
bDateMonthLeadingZero	Month day format with leading zero
bDateYearLong	Date year format is long (1999)
wDateFormat	Date format
bTime24h	Time format counts from 0 to 24 hours
bTimeLeadingZero	Time format has a leading zero
TimeTrenn	Time separator
DezTrenn	Decimal separator
bDecLeadingZero	Decimal format has a leading zero
DateTrenn	Time separator
DefFlowchartDir	The default directory for storing worksheets.

`DefStreamingDir`	The default directory for storing streaming files.
`DefDataDir`	The default directory for storing data values.
`DefDeviceDir`	The default directory for storing device data.
`DefUtilityDir`	The default directory for storing utility and tool data.

12 Functions provided by DASYLab

12.1 Memory Management

DASYLab supplies a set of memory management functions as a replacement for the standard windows functions. Besides ease of use, the DASYLab functions provide additional checks to detect bad code that writes over the memory bounds etc.

`void *MemAlloc ( DWORD nbytes )`

Purpose: Allocates `nbytes` Bytes of zero-filled memory.

Parameters: `nbytes` = number of bytes to allocate

Return value: Pointer to memory or `NULL` in case of error.

Remarks: Before returning `NULL`, `MemAlloc` displays a dialog box informing the user of the memory shortage. So the calling function need not display an additional 'out of memory' dialog box

`void MemFree ( void *ptr )`

Purpose: Free memory block.

Parameters: `ptr` = a memory block pointer that was allocated using `MemAlloc` before.

Return value: None.

Remarks: `MemFree` is implemented as a macro. In addition to releasing the memory block, it also sets `ptr = NULL`.

`void *MemReAlloc ( void *ptr, DWORD nbytes )`

Purpose: Increase or decrease the size of a memory block keeping the old contents intact.

Parameters: `ptr` = pointer to old memory block
`nbytes` = number of bytes to allocate

Return value: Pointer to new memory block or `NULL` in case of error.

Remarks: Before returning `NULL`, `MemReAlloc` displays a dialog box informing the user of the memory shortage. So the calling function need not display an additional 'out of memory' dialog box.

12.2 FIFO Buffer handling

FIFO buffers are used for data transport between the modules. Each output channel of a module has its own FIFO buffer. An output FIFO may be connected to the input side of more than one module in a worksheet.

The following functions handle output of data blocks to an output FIFO:

`DATA_BLOCK_HEADER *GetCurrentOutputBlock ( FIFO_HEADER *fifo )`

Purpose: Retrieves a pointer to a data block to store new data.

Parameters: `fifo` = pointer to a FIFO header.

Return value: Pointer to data block or `NULL` if the FIFO is full.

Remarks: When calling this function often, you will always get the same pointer again until you call the `ReleaseOutputBlock` function for the FIFO. After that you will get a new data block.

`BOOL ReleaseOutputBlock ( FIFO_HEADER FAR *fifo )`

Purpose: Outputs one data block to the FIFO.

Parameters: `fifo` = pointer to a FIFO header.

Return value: `TRUE` on success, `FALSE` in case of error. There is no need to look at the return code of this function.

Remarks: You should have filled up the block with data and have set the time information first before calling this function.

The following functions handle input of data blocks from the FIFO connected to the input side of a module:

`FIFO_HEADER *GetInputFifo ( MODULE *ThisModule, UINT wChan )`

Purpose: Retrieves a pointer to the FIFO connected to a module's input side at a specific channel.

Parameters: `ThisModule` = the module in question
`wChan` = input channel number

Return value: Pointer to FIFO header or `NULL` if input side is not connected.

Remarks: Typically this function will be called on experiment start time to check if the input channel data type etc. matches the need of the module.

We do not allow open input channels at the moment, but we may allow it in the future, so you'd better check for the `NULL` before accessing the data.

`DATA_BLOCK_HEADER *GetInputBlock ( MODULE *ThisModule, UINT wChan )`

Purpose: Retrieves a pointer to an input block for the selected channel.

Parameters: `ThisModule` = The module in question.
`wChan` = input channel number.

Return value: Pointer to data block or `NULL` if either the input side is not connected or no block is waiting there.

Remarks: When calling this function often, you will always get the same pointer again until you call the `ReleaseInputBlock` function for the channel. After that you will get a new data block.

`void ReleaseInputBlock ( MODULE *ThisModule, UINT wChan )`

Purpose: Release an input block.

Parameters: `ThisModule` = The module in question.
`wChan` = input channel number.

Return value: none.

Remarks: All modules that are connected to a FIFO must call the `ReleaseInputBlock` function before they all can access the next data block.

`void TOOLAPI CopyBlock ( MODULE *ThisModule, UINT wChan )`

Purpose: Copy an input block without change to the output.

Parameters: `ThisModule` = The module in question.
`wChan` = input channel number

Return value: none.

Remarks: Used in the `Lamp` example. The function is used here to copy input channels to output channels for a better worksheet overview.

12.3 Module class handling

The following two functions are described in detail in the above example.

`BOOL RegisterModuleClass ( MODCLASS *mc )`

Purpose: Register a module class with DASYLab.

Parameters: `mc` = description of the new module class.

Return value: `TRUE` on success, `FALSE` in case of error. There is no need to look at the return code of this function.

Remarks: `mc` may well point to a local variable, since the area is no longer needed after the `RegisterModuleClass` call.

`unsigned long PerformDefaultAction ( MODULE *ThisModule, int wMsg, int wParam, long lParam )`

Purpose: Default module message handler.

Remarks: A module must call this function for every message it does not process on its own.

12.4 General utility functions

void DsbHasChanged (void)

Purpose: Tell DASYLab that a worksheet has changed. Use this call i.e. at the `IDOK` of a dialog box so that the user is asked to save the worksheet before leaving the DASYLab session.

Parameters: none.

Return value: none.

void ShowWarning (char *title, char *text)

Purpose: Show a dialog box. Mainly used for warning messages.

Parameters: `title` = dialog box title.
`text` = dialog box text.

Return value: none.

void EnableAllWindows (BOOL bEnable)

Purpose: Enable/Disable all module windows.

Parameters: `bEnable` = `TRUE` for enable, `FALSE` for disable.

Return value: none.

void StopExperiment (void)

Purpose: Stop the experiment.

Parameters: none.

Return value: none.

Remarks: Call this function **before** calling a dialog box from inside the `ProcessData` function – otherwise you will get a series of dialog boxes until the stack overflows.

void PauseExperiment (void)

Purpose: Pause the experiment; can then be resumed later on.

Parameters: none.

Return value: none.

void CopyChannelName (FIFO_HEADER * Fifo, LPSTR *chBezeichnung)

Purpose: Copy the channel name to the connected module(s).

Parameters: `Fifo` = pointer to the channel's FIFO.
`chBezeichnung` = pointer to the channel name.

Return value: none.

`void FillUnitList ( HWND hDlg, UINT nListId, LPSTR SetString )`

Purpose: Fills a combo box with the default units.

Parameters: `hDlg` = handle to the dialog box.
`nListId` = ID of the combo box.
`SetString` = string to be set in the edit field of the combo box.

Return value: none.

`void ExpandUnitString ( char *Destination, char *UnitFormula,`
`MODULE *ThisModule, UINT wStartChannel );`

Purpose: Simplify a unit given in `UnitFormula`.

Parameters: `Destination` = string to receive the simplified unit.
`ThisModule` = pointer to the actual module.
`wStartChannel` = channel number this unit belongs to.

Return value: none.

`void ModalChooseColor ( void *colptr )`

Purpose: Replacement for the Windows `ChooseColor` dialog that disables all module windows first.

Parameters: `colptr` = pointer to a `CHOOSECOLOR` structure.

Return value: none.

`void ModalChooseFont ( void *fntptr )`

Purpose: Replacement for the Windows `ChooseFont` dialog that disables all module windows first.

Parameters: `fntptr` = pointer to a `CHOOSEFONT` structure.

Return value: none.

`BOOL LoadInterpolationTable ( void FAR *Table, char FAR *FileName,`
`int FAR *ErrorCode, char FAR *TableText )`

Purpose: Load DASYLab Interpolation Table

Parameters: `Table` = pointer to structure declared as

```
typedef struct
{
   UINT Columns;
```

```
   DLAB_FLOAT * xVal;
   DLAB_FLOAT * yVal;
} INTERPOLATION;
```

`FileName` = Table Filename with path description

`ErrorCode` = if return value == `FALSE`, check `ErrorCode` as String-ID in DASYLab Instance to get an Error description string

`TableText` = if the table is loaded successfully, the table description string is set to the `TableText`.

Return value: `TRUE` on success, `FALSE` in case of error.

`BOOL LoadTableFromFile ( void FAR *Table, char FAR *FileName,`
`char FAR *FirstLine, int nColumns,`
`int FAR *ErrorCode, char FAR *TabellenText )`

Purpose: Load DASYLab Interpolation Table from a file

Parameters: `Table` = pointer to structure declared as

```
typedef struct
{
   UINT Columns;
   DLAB_FLOAT * xVal;
   DLAB_FLOAT * yVal;
} INTERPOLATION;
```

`FileName` = Table Filename with path description
`ErrorCode` = if return value == `FALSE`, check `ErrorCode` as String-ID in DASYLab Instance to get an Error description string
`TableText` = if the table is loaded successfully, the table description string is set to the `TableText`.

Return value: `TRUE` on success, `FALSE` in case of error.

`void SetupInputBitmap ( MODULE *ThisModule, UINT wChannel, UINT wIndexNr )`

Purpose: Show a custom defined bitmap instead of the default bitmaps in the input channel

Parameters: `ThisModule` = selected module address
`wChannel` = channel number to change bitmap
`wIndexNr` = Index for bitmap or bitmap handle

Return value: none.

Remark: the bitmap handle has to destroyed by the user DLL

`void SetupOuputBitmap ( MODULE *ThisModule, UINT wChannel, UINT wIndexNr )`

Purpose: Show a custom defined bitmap instead of the default bitmaps in the output channel

Parameters: `ThisModule` = selected module address
`wChannel` = channel number to change bitmap
`wIndexNr` = Index for bitmap or bitmap handle

Return value: none.

Remark: the bitmap handle has to destroyed by the user DLL

12.5 String utility functions

`void chg_int ( char *buf, int val )`

Purpose: Converts an integer value into a string.

Parameters: `buf` = space to hold the result.
`val` = value to convert.

Return value: Conversion result is stored in `buf`.

Remarks: `buf` must be large enough to hold the result string plus one trailing zero.

`void chg_float ( char *buf, double val, int decimals )`

Purpose: Converts a floating point value into a string.

Parameters: `buf` = space to hold the result.
`val` = value to convert.
`decimals` = number of decimal places

Return value: Conversion result is stored in `buf`.

Remarks: `buf` must be large enough to hold the result string plus one trailing zero.

`double str2float ( char *s )`

Purpose: Replacement for `atof()`: Converts a string into a floating point value.

Parameters: `s` = string to convert.

Return value: Converted value.

Remarks: Within DASYLab we allow both comma and dot (period) to be used as a decimal separator. Therefore you must use `str2float()` instead of `atof()` when converting values read back from a dialog box into floating point again.

Currently we also re-define `atof()` to be identical to `str2float()` to be compatible with old code that used `atof()`. We may eventually remove this re-definition someday.

`char *strins ( char *dest, char *src )`

Purpose: Inserts the string `src` before `dest`.

Parameters: `dest` = pointer to destination string.
`src` = pointer to source string.

Return value: new string.

Remarks: This function is missing from the MSC libraries.

`void sprintf_year ( LPSTR cYear, int year, int mon, int day )`

Purpose: Format the date string in the way of the country-specific settings.

Parameters: `cYear` = space to hold the result.
`year` = actual year to be converted
`mon` = actual month to be converted
`day` = actual day to be converted

Return value: none.

Remarks: Get the actual date with the help of the `localtime` function of windows. Now convert to string.

Example:
```
struct tm *today;
today = localtime (&ltime);
sprintf_year ( szString, today->tm_year, today->tm_mon+1,
               today->tm_mday );
```

`void sprintf_clock ( LPSTR cTime, int hour, int min, int sec )`

Purpose: Format the time string in the way of the country-specific settings.

Parameters: `cTime` = space to hold the result.
`hour` = actual hour to be converted
`min` = actual minute to be converted
`sec` = actual second to be converted

Return value: none.

Remarks: Get the actual time values with the help of the `localtime` function of windows. Now convert to string.

Example:
```
struct tm *today;
today = localtime (&ltime);
sprintf_clock ( szString, today->tm_hour, today->tm_min, today->tm_sec );
```

12.6 Math utility functions

`BOOL IsNearlyEqual ( double a, double b, double tol )`

Purpose: Test, if `a` and `b` are nearly equal within a given tolerance.

Parameters: `a, b` = the two numbers to compare
`tol` = tolerance, i.e. `0.01` = 1 percent deviation

Return value: `TRUE`, if `a` and `b` are nearly equal within the tolerance, else `FALSE`.

Remarks: When comparing floating point numbers, it is generally not wise to do exact equality or inequality comparisons. Use this function instead.

void do_fft (DLAB_FLOAT *datar, DLAB_FLOAT *datai, int n, int Direction)

Purpose: Basic FFT calculation.

Parameters: `datar` = real part of the data
`datai` = imaginary part of the data
`n` = number of data; must be a power of 2
`Direction` = 1 for FFT or = -1 for inverse FFT

Return value: The transformed data replaces the original data in the `datar` and `datai` arrays.

Remarks: This is the working function behind module classes like FFT or Correlation. Note that it always works complex and always works on the `DLAB_FLOAT` basic type.

12.7 Dialog Box Handling

void InitChannelBar (HWND hDlg)

Purpose: Initializes the channel bar for a module dialog box and change all necessary dialog box texts to small fonts.

Parameters: `hDlg` = handle of dialog box window.

Return value: none.

Remarks: You must set up the proper values for `wDialNumChan`, `wDialCurChan` and `wDialMaxChan` first before calling this function.

void HandleChannelClick (HWND hDlg, UINT but)

Purpose: Change the active (green) channel in the channel bar.

Parameters: `hDlg` = handle of dialog box window.

`but` = Button ID of the clicked button

Return value: This function manipulates the global variable `wDialCurChan`.

void HandleLessMoreButton (HWND hDlg, UINT but)

Purpose: Add/remove one channel from the channel bar.

Parameters: `hDlg` = handle of dialog box window.
`but` = `ID_LESS` to remove a channel
`but` = `ID_MORE` to add a channel

Return value: none.

void HandleLessMoreCountButtons (HWND hDlg, UINT but , UINT wCount)

Purpose: Add/remove wCount channels from the channel bar.

Parameters: hDlg = handle of dialog box window.

but = ID_LESS to remove channels

but = ID_MORE to add channels

wCount = number of added/removed channels

Return value: none.

BOOL ChangeModuleSize (MODULE *ThisModule, UINT NumIn, UINT NumOut)

Purpose: Change the number of input and output channels for a module.

Parameters: ThisModule = the module in question.
NumIn = new number of input channels.
NumOut = new number of output channels.

Return value: TRUE on success, FALSE in case of error.

Remarks: This function needs to be called from inside the IDOK case of the WM_COMMAND message handling of the module's parameter dialog box whenever the number of input or output channels may have changed.

BOOL ChangeModuleName (MODULE *ThisModule, LPSTR szNewName)

Purpose: Change the name of a module and make the name change known to all instances within DASYLab that might use it (Action modules, Layout objects).

Parameters: ThisModule = the module in question.

szNewName = new name of the module.

Return value: TRUE on success, FALSE in case of error.

Remarks: This function needs to be called from inside the IDOK case of the WM_COMMAND message handling of the module's parameter dialog box whenever the module name may have changed. An error check is made that the new module name is not yet in use, and information about the name change is passed to all instances that make use of it. Note that the module name should no longer be changed directly.

BOOL CenterDialog (HWND hwnd, HWND hwndOwner)

Purpose: Set the dialog box in the center of the referring window and change all necessary dialog box texts to small fonts.

Parameters: hwnd = window handle of the window which is to be positioned.
hwndOwner =window handle of the „father“ window.

Return value: TRUE on success, FALSE in case of error.

Remarks: This function needs not to be called in dialog box with channel bars. The function InitChannelBar calls the CenterDialog function. All other dialog boxes have to call the CenterDialog function in the WM_INITDIALOG switch of the dialog box function.

12.8 Print utility functions

BOOL InitPrinter (DruckInfo FAR *di, BOOL bAskBefore)

Purpose: Set all initialization variables before printing.

Parameters: di = print structure.

bAskBefore = Quit a message box before printing or print at once.

Return value: TRUE on success, FALSE in case of error.

Remarks: Before any printing actions this function is the first one to be called. If the return value is false leave the printing routine without any printing action.

BOOL PrintHeader (DruckInfo FAR *di, DruckInfo FAR *di,
char FAR *chModuleName)

Purpose: Print the header defined in the printer layout.

Parameters: di = print structure.

chModuleName = Name of the printing module

Return value: TRUE on success, FALSE in case of error.

BOOL PrintFooter (DruckInfo FAR *di)

Purpose: Print the bottom defined in the printer layout.

Parameters: di = print structure.

Return value: TRUE on success, FALSE in case of error.

BOOL NextPage (DruckInfo FAR *di)

Purpose: Send the complete page to the printer.

Parameters: di = print structure.

Return value: TRUE on success, FALSE in case of error.

12.9 Functions for global variables and global strings

BOOL ExpandVarNumber (char *Source, int MaxSourceSize, short *Number)

Purpose: Receive the corresponding number to a placeholder string {$VAR_x}.

Parameters: `Source` = pointer to the placeholder string
`MaxSourceSize` = number of characters in the placeholder string
`Number` = address of the variable to receive the corresponding number

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlVar_RegisterByNumber ( short Number, void *ObjectPtr,
     unsigned long (*UpdateFunction) ( void *ObjectPtr, int wMsg,
                                       int wParam, long lParam ) )
```

Purpose: Register the update function which has to be called if the global variable indicated by `Number` has changed.

Parameters: `Number` = number of the global variable.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlVar_UnregisterByNumber ( short Number, void *ObjectPtr,
     unsigned long (*UpdateFunction) ( void *ObjectPtr, int wMsg,
                                       int wParam, long lParam ) )
```

Purpose: Unregister the update function which has to be called if the global variable indicated by `Number` has changed.

Parameters: `Number` = number of the global variable.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlVar_Set ( short Number, double GlValue )
```

Purpose: Assign a value to the global variable indicated by `Number`.

Parameters: `Number` = number of the global variable.
`GlValue` = new value.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlVar_Get ( short Number, double *GlValue )
```

Purpose: Receive the value of the global variable indicated by `Number`.

Parameters: `Number` = number of the global variable.
`GlValue` = address of the variable to receive the value.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlVar_PrefixGet ( short Number, char *GlText, int MaxSize )
```

Purpose: Receive the prefix string of the global variable indicated by `Number`.

Parameters: `Number` = number of the global variable.
`GlText` = address of the string to receive the prefix string.
`MaxSize` = max. size of `GlText`.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlVar_VarStringGet (short Number, char *GlText)

Purpose: Receive the value of the global variable indicated by `Number` as string.

Parameters: `Number` = number of the global variable.
`GlText` = address of the string to receive the value as string.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlVar_SetSyntax (char* szDest, short nNumber)

Purpose: This function is used to display placeholders for global variables in the correct syntax. (Often used in dialog boxes.)

Parameters: `Number` = number of the global variable.
`GlText` = address of the string to receive the placeholder string.

Return value: `TRUE` on success, `FALSE` in case of error.

void SetMenuForGlobalVars (HWND hDlg, UINT wID)

Purpose: Use this function to expand the popup menu of edit fields in dialog boxes. The appended menu item allows to select global variables via dialog.

Parameters: `hDlg` = handle of the dialog box.
`wID` = ID of the edit field.

Return value: none.

BOOL ExpandString (char *Source, int MaxSourceSize, char *Destination, int MaxDestSize)

Purpose: Receive the corresponding string to a placeholder string `{$STR_x}`.

Parameters: `Source` = pointer to the placeholder string.
`MaxSourceSize` = number of characters in the placeholder string.
`Destination` = pointer to the destination string.
`MaxDestSize` = number of characters in the placeholder string.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlStr_RegisterByNumber (short Number, void *ObjectPtr,
unsigned long (*UpdateFunction) (void *ObjectPtr, int wMsg,
int wParam, long lParam))

Purpose: Register the `update` function which has to be called if the global string indicated by `Number` has changed.

Parameters: `Number` = number of the global string.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlStr_RegisterByText ( char *theText, void *ObjectPtr,
     unsigned long (*UpdateFunction) ( void *ObjectPtr, int wMsg,
                                       int wParam, long lParam ) )
```

Purpose: Register the update function which has to be called if the global string indicated by the placeholder `theText` has changed.

Parameters: `theText` = placeholder of the global string.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlStr_UnregisterByNumber ( short Number, void *ObjectPtr,
     unsigned long (*UpdateFunction) ( void *ObjectPtr, int wMsg,
                                       int wParam, long lParam ) )
```

Purpose: Unregister the update function which has to be called if the global string indicated by `Number` has changed.

Parameters: `Number` = number of the global string.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlStr_UnregisterByText ( char *theText, void *ObjectPtr,
     unsigned long (*UpdateFunction) ( void *ObjectPtr, int wMsg,
                                       int wParam, long lParam ) )
```

Purpose: Unregister the update function which has to be called if the global string indicated by the placeholder `theText` has changed.

Parameters: `theText` = placeholder of the global string.
`ObjectPtr` = pointer to the current module.
`UpdateFunction` = address of the `PerformAction` function of this module.

Return value: `TRUE` on success, `FALSE` in case of error.

```
BOOL GlStr_Set ( short Number, char *GlText, int MaxSiz e)
```

Purpose: Assign a string to the global string indicated by `Number`.

Parameters: `Number` = number of the global variable.
`GlText` = new string.
`MaxSize` = length of new string.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlStr_Get (short Number, char *GlText, int MaxSize)

Purpose: Receive the string of the global string indicated by `Number`.

Parameters: `Number` = number of the global string.
`GlText` = address of the string to receive the string.
`MaxSize` = max. length of `GlText`.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlStr_PrefixGet (short Number, char *GlText, int MaxSize)

Purpose: receive the prefix string of the global string indicated by `Number`.

Parameters: `Number` = number of the global string.
`GlText` = address of the string to receive the prefix string.
`MaxSize` = max. length of `GlText`.

Return value: `TRUE` on success, `FALSE` in case of error.

BOOL GlStr_SetSyntax (char* szDest, short nNumber)

Purpose: This function is used to display placeholders for global strings in the correct syntax. (Often used in dialog boxes.)

Parameters: `Number` = number of the global string.
`szDest` = address of the string to receive the placeholder string.

Return value: `TRUE` on success, `FALSE` in case of error.

void SetMenuForGlobalStrings (HWND hDlg, UINT wID)

Purpose: Use this function to expand the popup menu of edit fields in dialog boxes. The appended menu item allows to select global strings via dialog.

Parameters: `hDlg` = handle of the dialog box.
`wID` = ID of the edit field.

Return value: none.

BOOL ChangeNameInString (LPSTR szString, UINT uiStringLength,
LPSTR szFrom, LPSTR szTo)

Purpose: This function is used to exchange placeholders for global strings or variables in strings after a name change.

Parameters: `szString` = the string in which placeholders are to be exchanged.
`uiStringLength` = maximum length of this string.
`szFrom` = old name of the global variable/string.
`szTo` = new name of the global variable/string.

Return value: `TRUE` on success, `FALSE` in case of error. An error can occur if the new name is longer than the old one and the new string does not fit into `uiStringLength` characters.

Remarks: This function usually is called upon receiving a `DMM_CHANGE_VAR_NAME` message in a module, when the module contains strings that may contain global variables (say file names).

12.10 Internal Error Handling

We strongly suggest introducing some validity checks in the code to look for suspicious situations. You can use the following macros. All of them show the `InternalError` dialog box on the screen displaying the source file and line where the error occurred and the error message. They also save the current worksheet and error message under the name `dlab_err.dsb` and `dlab_err.log`.

`void InternalError ( char *msg )`

Purpose: Show an internal error message

Parameters: `msg` = message to be displayed

Return value: none.

`void ImpossibleCase ( void )`

Purpose: Same as `InternalError("Impossible case");`

Parameters: none.

Return value: none.

Remarks: Use this one in all default cases of switch statements that '*do not have a default case*'.

`void ImpossibleCaseNum ( long n )`

Purpose: Similar to `ImpossibleCase()` but in addition displays the number of the wrong case also.

Parameters: `n` = number of the wrong case.

Return value: none.

12.11 Console Output

Starting with DASYLab 11.0, two commands are available to display warnings and texts.

`void PASCAL ConsoleOutf ( char *szText )`

Purpose: Outputs a text with certain formatting.

Parameters: `szText` = text to be output in the console.

Return value: None

Remarks: Formatting instructions can be integrated in the text. The instructions always refer to the complete text. All instructions must be at the beginning of the text.
The following instructions are possible:
`\b+` Text is output in bold.
`\i+` Text is output in italics.
`\cRRGGBB` Text is output in color (a letter pair must be replaced by a hexadecimal number which specifies the respective color intensity, for example, `\cFF0000` for red or `\c000000` for black. You also can set a semicolon behind the color code.

Example: `ConsoleOutf ("\\b+\\i+\\cFFFFFF;Example");`

`void PASCAL ConsoleOutWarning (LPCTSTR szTitle, LPCTSTR szMsg)`

Purpose: Outputs a warning in the console as „`Title: Msg`".

Parameters: `szTitle` = Title, e.g. module name to be displayed in the console
`szMsg` = Message to be displayed in the console

Return value: None

The display of either function cannot be suppressed in the DASYLab warning options. However, you can specify in the warning options whether the messages get a time stamp.

12.12 Module independent memory registration

The function `RegisterExtraMemory` registers exactly one additional memory range. After that it is necessary to call `AddExtMemInstance` if one wants to save the registered memory in a worksheet (in DSB- or ASCII-format) and loading the worksheet reads this memory later.

The pointer to this memory range and its size have to be passed as arguments of the function `RegisterExtraMemory`. The memory range consists of an array of structures; it is possible that there is only one element of the array. The fifth argument determines the count of array-elements for each identifier. It is also possible to call the function `RegisterExtraMemory` several times to register some memory ranges inside of a DLL. The identifier must be unique in order to distinguish the memory ranges. This identifier should not have too few characters as it must be unique inside of DASYLab. If the identifier is not unique an error message will appear. Also a pointer to `PARAMETER_INFO` for saving in ASCII-format of the memory description should be passed.

All structure elements, which are saved as additionally memory, are described in the memory description similarly to the description of module parameters. If `pParamInfo` is `NULL` the structures

of the additionally memory is not readable in the ASCII-worksheet file. Trying to save this file in text format it will result in a warning message.

BOOL TOOLAPI RegisterExtraMemory (char *Identifier, void *pExtMemory, int Size, PARAMETER_INFO *pParamInfo, int NumStructures)

Purpose: The function registers exactly one additional memory range.

Parameters: `Identifier` = unique inside of DASYLab.
`pExtMemory` = pointer to the additional memory.
`Size` = size of the additional memory.
`pParamInfo` = pointer to the memory description.
`NumStructures` = count of array-elements.

Return value: `TRUE` on success, `FALSE` in case of error.

Remarks: You can register a maximum of 32 memory ranges. The length of the identifier can be 19 characters and the memory range must not be bigger than 65536 bytes.

BOOL TOOLAPI AddExtMemInstance (char *Identifier)

Purpose: This function call increments the instance counter for the named identifier by one.

Parameters: `Identifier` = unique inside of DASYLab.

Return value: `TRUE` on success, `FALSE` in case of error.

Remarks: Use the same identifier as in the call to the function `RegisterExtraMemory`.

BOOL TOOLAPI RemoveExtMemInstance (char *Identifier)

Purpose: This function call decrements the instance counter for the named identifier by one, if there is at least one instance of the identifier. Otherwise the counter of instance is set to zero.

Parameters: `Identifier` = unique inside of DASYLab.

Return value: `TRUE` on success, `FALSE` in case of error.

Remarks: Use the same identifier as in the call to the function `AddExtMemInstance`.

Example:

```
char *ExtraMemID = "IDENTIFICATION";
int size = NumStructures*sizeof(EXTRA_MEMORY);
pExtraMemory = MemAlloc (size);
if (pExtraMemory == NULL)
      return FALSE;
if (!RegisterExtraMemory (ExtraMemID, (void*) pExtraMemory, size, ParameterInfo, NumStructures))
      return FALSE;
//Call once AddExtMemInstance to make it possible to save registered memory
//independent of modules
if (!AddExtMemInstance (ExtraMemID))
    return FALSE;
```

12.13 Multiple Time Base Usage

These functions provide access to the multiple time bases of DASYLab. For a full description of the methods, read *Chapter 9 Multiple Time Bases in DASYLab*.

For an example, refer to the source module `GENERAT.C`.

`void TOOLAPI RegisterTimeBase ( LPSTR szName, UINT uiID, LPSTR szDescription, void (CALLBACK *SetDriver) (EXT_TIMEBASE *pExtBase), BOOL bTemporary )`

Purpose: The function registers a time base structure for use within DASYLab.

Parameters: `szName` = name of the time base. Appears on the tab in the time bases dialog box.
`uiID` = unique identifier of the time base.
`szDescription` = long description of the time base.
`SetDriver` = pointer to the callback function that is called by DASYLab when the time base information is changed by the user.
`bTemporary` = set this always to `FALSE`.

Return value: none.

`void TOOLAPI UnregisterTimeBase ( UINT uiID )`

Purpose: The function unregisters a time base structure from DASYLab.

Parameters: `uiID` = unique identifier of the time base.

Return value: none.

Remarks: This function needs not to be called at the end of the program since DASYLab properly unregisters all time bases.

`BOOL TOOLAPI SetTimeBase ( EXT_TIMEBASE *pExtBase )`

Purpose: The function updates the time base settings within DASYLab.

Parameters: `pExtBase` = pointer to an `EXT_TIMEBASE` structure describing the new settings of the time base.

Return value: `TRUE`, if the time base was updated successfully, `FALSE` otherwise.

Remarks: This function must be called if the driver wants to update the time base information (e.g. if a settings request cannot be fulfilled by the hardware).

`void TOOLAPI SetTimeBaseTime ( UINT uiID, double fTime )`

Purpose: The function updates the time of the time base.

Parameters: `uiID` = unique identifier of the time base.
`fTime` = new time for this time base in seconds.

Return value: none.

Remarks: The driver should call this function whenever the hardware delivers a new block of data so that the time information of the time base is synchronous to the hardware clock.

void TOOLAPI IncTimeBaseTime (UINT uiID, double fAddTime)

Purpose: The function increments the time of the time base.

Parameters: `uiID` = unique identifier of the time base.
`fAddTime` = time increment for this time base in seconds.

Return value: none.

Remarks: This function is an alternative to the previous one, so the remarks there hold here also.

UINT TOOLAPI GetTimeBaseBlockSize (UINT uiID)

Purpose: The function retrieves the actual block size of the time base.

Parameters: `uiID` = unique identifier of the time base.

Return value: The block size of the time base.

double TOOLAPI GetTimeBaseSampleDistance (UINT uiID)

Purpose: The function retrieves the actual sample distance of the time base.

Parameters: `uiID` = unique identifier of the time base.

Return value: The sample distance of the time base in seconds.

double TOOLAPI GetTimeBaseTime (UINT uiID)

Purpose: The function retrieves the actual time of the time base.

Parameters: `uiID` = unique identifier of the time base.

Return value: The time of the time base in seconds.

Remarks: This function is a replacement for `CurrentExperimentTime` which retrieves the time of the DASYLab default driver.

void TOOLAPI FillTimeBaseCombo (HWND hDlg, UINT uiIdCombo, UINT uiIDSel)

Purpose: The function sets up a combo box in a dialog for selection of a time base.

Parameters: `hDlg` = window handle of the dialog
`uiIdCombo` = resource ID of the combo box

uiIDSel = time base ID of the current time base. This item will be selected in the combo box.

Return value: none

Remarks: Call this function in the INITDIALOG part of a dialog box function.

UINT TOOLAPI GetTimeBaseComboID (UINT uiSel)

Purpose: The function returns the time base ID of the selected entry of a combo box created with the previous function.

Parameters: uiSel = selection index of the combo box.

Return value: The unique identifier of the time base selected.

BOOL TOOLAPI TimeBaseDialog (HWND hwndParent, UINT uiID)

Purpose: The function opens the DASYLab time base dialog.

Parameters: hwndParent = window handle of the actual window. The dialog will be displayed as a sub-window of this one. It can be NULL.
uiID = the time base identifier of the tab that is brought to front in the dialog.

Return value: TRUE if the new settings are valid (user has clicked *OK*), FALSE if settings are invalid (user has clicked *Cancel*).

Remarks: Note that the driver is informed about the new settings by a call to the SetDriver callback function (see RegisterTimeBase).

12.14 Extra Memory Block handling across data connections

For versions auf DASYLab released after DASYLab 2016, we provide a new API for data transport at start time (we call this *static*) and data transport with the data blocks at process time (we call that *dynamic*). A first version of this API is included in the *Extension Toolkit for DASYLab 2016*, but all functions (except one) are subject to change. To provide compatibility with future versions of DASYLab (as far as we can look into the future) add a call of EmemBlock_PROCESS_MsgCopyPlain to your code before the call of ReleaseOutputBlock.

```
BOOL TOOLAPI EmemBlock_PROCESS_MsgCopyPlain (
     FIFO_HEADER* OutFifo, DATA_BLOCK_HEADER* OutBlock,
     FIFO_HEADER* InFifo, DATA_BLOCK_HEADER* InBlock );
```

Purpose: The function copies the dynamic extra memory block from the input to the output of a module.

Parameters: OutFifo = Pointer the output FIFO.
OutBlock = Pointer to the actual output block.
InFifo = Pointer to the FIFO on the input of a channel, NULL if the same FIFO setting are used for input and output (block size and sample rate).
InBlock = Pointer to the actual Input Block.

Return value: `TRUE` if the operation was successful.

Example (from `deriv.c`):

```
// TK2016, for future compatibility: Copy extra memory (per data block) from the "Father" block
//                                   to the block we filled with data above
// - This will have no effect in DASYLab 2016 but your module/dll will support a new feature in
//   later versions of DASYLab without recompiling
// - The cpu cost for DASYLab 14 is near zero - the call of this funtion is optional for DASYLab 2016
//   and mandatory for the following DASYLab versions.
// - If the InFifo parameter is NULL, the function assumes
//   that the maximum blocksize of the input fifo is equal to the maximum blocksize of
//   the output fifo (what is "the normal behaviour").
//
// If you are unsure (what you shouldn't be because you wrote/modified the SetupFifo_xxx routine)
// or the maximum blocksize of OutFifo and InFifo is not equal, then provide the InFifo as parameter.

EmemBlock_PROCESS_MsgCopyPlain (OutFifo, OutputBlock, NULL, InputBlock);
// Add this Data Block to the FIFO, so that a "Son" FIFO can get Access to it
ReleaseOutputBlock (OutFifo);
```

13 General Conventions

New module classes should match the general look and feel of DASYLab. In the ideal case the end user cannot (or very hardly) decide what modules are original DASYLab ones and what modules were provided from different sources.

To achieve this it is usually best to use the supplied examples as a starting point for own projects. To avoid conflicts with DASYLab modules and between different DLL extension each newly created module class also has to follow certain naming conventions etc.

13.1 Creating new module classes

Any module class used by DASYLab must have a unique name set up in the `mc.Name` field of the `MODCLASS` structure before calling the `RegisterModuleClass` function. DASYLab will use this name for identification when loading saved worksheets.

The name string should consist of the following components:

- two or three characters to identify the supplier of the module class. We suggest using the initials of the company or the programmer there.
- a colon
- the internal name of the module class.

All names starting with *`DAP:`* or *`DL:`* or *`EVA:`* and all names containing no colon (*`:`*) are reserved for use by *National Instruments Ireland Resources Limited*.

As mentioned before, the Menu ID in the `mc.MenuId` field must lie within the range of 2950 to 2974 for all module classes defined inside a DLL extension. The ID must be unique within one DLL extension but you may use the same ID codes in different DLLs since DASYLab remaps all ID codes of a DLL to an internal code.

No module class should try to extend any of DASYLab's internal structures, use reserved fields, send messages to modules, or define new messages other than the ones listed above. No module class should assume any special behavior of DASYLab 's function that is not documented here.

13.2 Creating new project files / makefiles

When creating new project files/makefiles for projects containing DASYLab extensions, you have to follow these rules for the project setup:

a) First, select **Project Defaults** for making a Windows DLL. Then check/change these default settings like listed below.

b) Set the general function calling convention to **C**, or **C/C++**.

c) Change the **Structure Member Alignment** to 1 Byte. This ensures structures are packed tight.

d) Change the **Memory Model** to Large (default for code and data pointers is use FAR pointers) and set up for SS not equal to DS.

e) Add STRICT to the predefined constants.

f) Select generating **Windows Prolog/Epilog** code for protected mode DLL functions.

g) Select automatic generation of Windows Prolog/Epilog code for all `FAR` functions.

h) Set up environment variables (include path), etc. to allow the compiler find the toolkit files: Add **`..\inc`**, **`..\bmp`** and **`..\rc`** to the include path.

Make sure that all of these settings apply to both: Generating code for debug or for release versions.

13.3 Dialog box style guide

Below are suggestions for dialog box design that should be followed when designing dialog boxes for DASYLab extensions.

Generally speaking, DASYLab extension dialog boxes should exactly match the style of DASYLab's internal (new style) dialog boxes. When designing new dialog boxes, it is usually best to copy an existing dialog box first and make modifications to it, rather than starting from scratch each time. All dialog boxes should be tested (and must look good) in at least the following screen resolutions:

- 640x480, small fonts
- 800x600, small fonts
- 1024x768, small fonts
- 1024x768, large fonts

We suggest that you use 800x600 to create dialog boxes because dialog boxes created while using large fonts often look bad in other resolutions.
Text fields will generally vary in size when viewed in different resolutions. You should leave space after each text field and avoid lengthy text fields where possible.

Below are suggestions for how the individual elements of a dialog box should look.

13.3.1 Global dialog box

- Use the font `MS Sans Serif` in `8pt` size and `FAT` style.
- Combine controls that belong to a common group into one *group box*.
- Use a *grid setting* of 5 in both directions and create/modify your dialog box with grid turned on.
- All items except group boxes are *10 units vertically* and *5 units horizontally* apart from the border of the dialog box.

13.3.2 Group boxes

- The edges of group boxes should lie exactly on a grid point. The items inside a group box are positioned relatively to that group box.
- Group boxes are *5 units vertically* and *10 units horizontally* apart from each other.
- If the group box contains a title (recommended), the title must have a blank as first and last characters.
- Group boxes are *5 units vertically* and *5 units horizontally* apart from the border of the dialog box.

13.3.3 Static and edit controls

- Edit controls are preceded (or surrounded) by a static text that explains the input field.
- This static text is *15 units* apart from the top of the group box and *5 units* apart from the left border of the dialog box. The edit controls are vertically *2 units* higher than the static text describing it.
- Static text fields are *12 units* high.
- Text and edit controls are vertically *15 units* apart from each other.

13.3.4 Radio and check buttons

- Any group of radio or check buttons should normally be surrounded by a separate titled group box.
- Radio and check buttons are *5 units* apart from the left border of the dialog box.
- The first radio/check button is *12 units* apart from the top of the group box.
- Radio and check buttons are vertically *13 units* apart from each other.

13.3.5 Push Buttons

- Push buttons (including the default push button) are always *50 units wide* and *14 units high*.
- Push buttons are normally the rightmost items of the dialog box.
- Push buttons are horizontally *10 units* apart from other items or from the border of the dialog box.
- Push buttons are vertically *20 units* apart from each other. This may be increase to *30 units* where you want to group push buttons.
- The push buttons included in every dialog box are (from top to bottom): *OK*, *Cancel*, *Help*.
- If a dialog box gets too wide, the push buttons may be placed at the lower border of the dialog box.

14 Version History of the Changes

DASYLab 7.0

- DQM_CHECK_REPLACE, DMM_CREATE_REPLACE, and DMM_REPLACE_MODULE
- ChangeModuleName

DASYLab 8.0

- DMM_REQUEST_GLOB_VARS

DASYLab 9.0

- Procedure for including modules in DASYLab menus simplified
- Changing the color density of the symbols to 24 bit

DASYLab 10.0

- Dialog to display Module Times

DASYLab 11.0

- Procedure for calling the help with TKCallOnlineHelp
- Console Output
- DMM_PREPARE_STOP message, DMM_IS_DEBUG message
- DMM_PREPARE_START_MODULE
- DMM_NORMALIZE_ALL_WINDOWS, DMM_HIDE_ALL_WINDOWS, DMM_SHOW_ALL_WINDOWS, DMM_MINIMIZE_ALL_WINDOWS message

DASYLab 2016

- DLAB_FLOAT changed from float to double
- MAX_BLOCKSIZE is now 1048576 (2^20)
- Sysinfo changed to SerialOpt
- Removed old functions that are no longer used.
- New module message DMM_GET_TIMEBASE_ID
- New functions for handling of *Extra Memory Blocks* across the data connections in a worksheet.