

NC-Toolkit

for WINDOWS

Software Development Kit (Version 9.3x)

With DLL-Interface for

- **Real-Time CNC Motion Control**
- **MTASC Programming Language**
- **Suitable for most Programming Tools**
- **I/O-Access**

**Windows XP
Windows Vista
Windows 7
32 Bit**

© 2009 by

TRIMETA software GmbH

Auf der Steig 4/1
75233 Tiefenbronn
Tel. +49(0)7234 / 981733 Fax 981734

eMail: info@trimeta.de
www.trimeta.de

This Manual, along with the Software belonging to it, is protected by copyright. Except making in house backup copies any reproduction or resale is prohibited.

Copyright © 2009 by TRIMETA software GmbH, D-75233 Tiefenbronn, GERMANY

Contents

Contents of the Package.....	2
Important Files for NC-Toolkit and EdiTasc.....	3
Files with Extension ".ts":.....	3
Files with Extension ".ini":.....	3
Files with Extension ".t", ".NC", ".PLT"	3
DLL and Driver:.....	3
Other files:.....	4
The Device Types.....	4
Optional Features.....	4
Installation.....	5
The Architecture of EdiTasc / MTASC / Mtdrv.....	6
The architecture of the device driver (Mtdrv.sys).....	7
The MTASC Script Language Interface of Mtasc.dll (Win32) / MtascDN.dll (DotNet).....	7
Platform dependent Differences Win32 / DotNet.....	8
The functions exported by Mtasc.dll (MtascDN.dll).....	8
MtInit: Initialising MTASC.....	8
MtExec: Execute String as MTASC Command.....	8
MtManStep: Jogging the axes manually.....	10
MtGetData: Read values from MTASC, allow error message.....	11
MtGetDataE: Read values from MTASC, return error code.....	11
MtDataRead: Read values of complex MTASC variables.....	12
MtDataSet: Set values of complex MTASC variables.....	12
MtGetStatus: Read status data from MTASC and driver.....	13
Message Boxes and Error Handling in Mtasc.dll.....	14
The Motion and Graphic Interface (AGI).....	14
The Real-Time Device Driver Interface (DRV).....	15
The Graphic Interface (G3).....	15
The Hello Sample Programs.....	15
Initializing with Hello.ts and HelloGraph.ts.....	16
Other Programming tools.....	16
Debugging with MtTrace.....	16

Contents of the Package

NC-Toolkit contains

- **EdiTasc files:** EdiTasc uses the same DLL interface as NC-Toolkit and is a helpful and often necessary environment for testing your **MtExec** commands. It is strongly recommended to install EdiTasc first before using the Toolkit.
- **Document files:** The files in the **Docu** folder give a brief overview of the structure and the handling of the Toolkit/MTASC features. An important part of the documentation are the help files in the Docu folder of EdiTasc.
- **Runtime files:** These files are necessary for the use of Toolkit. They must be copied from the **Runfiles** folder to the folder in which any of the toolkit programs are running.
- **Driver files:** The device driver is located in the **Driver** folder. Some tools are included for registering the driver.
- **Sample files:** There are several HELLO projects including source code.

Important Files for NC-Toolkit and EdiTasc

Files with Extension ".ts":

These files are MTASC Programs that are needed at runtime (see MTASC manual Mtasc.chm).

- **System.ts** includes all MTASC definitions needed for EdiTasc. It is a good idea to start with calling it in your application too, as shown in Hello.ts. Experienced programmers may want to strip some overhead which is not used in their application.
- **EtInit_xx.ts** is usually called at startup. xx represents the first device type in the line **PdType** of the file **Mtdrv.ini**, which lists the device types used.
- **Hello.ts** is a MTASC program called by the HELLO projects.
- **ParamSet.ts**: MTASC program to read the data in **Paramset.ini**.
- **Refsrch.ts**: Searches the reference positions for all axes, usually on request of the user.

Files with Extension ".ini":

These files are initialisation files for different purposes.

- **ParamSet.ini**: Contains parameter settings like standard speeds and origin settings. The values are read into the MTASC environment by executing **ParamSet.ts**.
- **Mtdrv.ini**: Configuration parameters of the driver.

Files with Extension ".t", ".NC", ".PLT"

These files contain CNC programs for or written by the user.

- **DEMO1.t** is an example for milling a closed path with line and circle commands.
- Same for other DEMO1 files with different file extension to indicate the different command types (t: MTASC, .NC: G-Code DIN 66025/ISO, .PLT: HPGL file).

DLL and Driver:

- **Mtasc.dll**: MTASC interpreter for the Win32 platform giving access to the axes (motion) and graphic interface.

The graphic functionality formerly provided by G3Graph.dll is now located in Mtasc.dll itself, so it is not needed anymore.

- **MtascDN.dll**: MTASC interpreter for the DotNet platform giving access to the axes (motion) interface. The graphic interface is not yet fully accessible.
- **Mtdrv.sys**: This is the device driver which includes the real time kernel
- **Version of MTASC:** **6.58.7.x**
Version of Driver Mtdrv: **4.58.7.x**

Other files:

- **Mtasc.lib** : Import library for Mtasc.dll
- **MtascDll.h**: The corresponding C-header file
- **MtascData.h**: Important MTASC internal variables. See also the comments there.
- **MtdrvData.h**: Important driver internal variables. See also the comments there.
- **G3Graph.h**: The corresponding C header file

Access to the graphic library is included „as is“, without guaranteed success in using it.

The Device Types

The different types of controller interfaces supported by EdiTasc / NC-Toolkit are described by a table of so called device types.

From Version 8.5x on only one driver Mtdrv.sys is needed. It supports all available device types. The line PdType (**Physical Device Type**) in Mtdrv.ini specifies the type for each axis.

The device type for each axis may be different. E.g., you can drive 3 axes with a servo PCI card and axis 4 with a stepper motor controlled by pulse/train signals from the printer port.

More specific hardware properties may be supplied by the lines **HwType (Hardware Type)** and **PdId (Physical Device Id)**.

Optional Features

Some features like the maximum number of axes supported are registered in file **ComReg.ini** (Component Registration). When calling MtInit() in Mtasc.dll, this file is checked to determine which options are to be supported.

See our price list for the most common optional features.

In the minimal base version the vector commands in MTASC like ML, MF, MC do not support "look ahead" (= smooth transition of vector commands without stopping). However, linear and circular interpolation with concurrent motion of the axes is included.

Installation

Install **EdiTasc** first by calling Setup.exe. This also installs the drivers needed by NC-Toolkit. The first installation of EdiTasc requires rebooting the PC.

Then copy the **NC-Toolkit** folder to any directory that you create (other than EdiTasc), including all subdirectories.

The sample projects require the corresponding programming tools. Their exe-files however should run as soon as EdiTasc is installed.

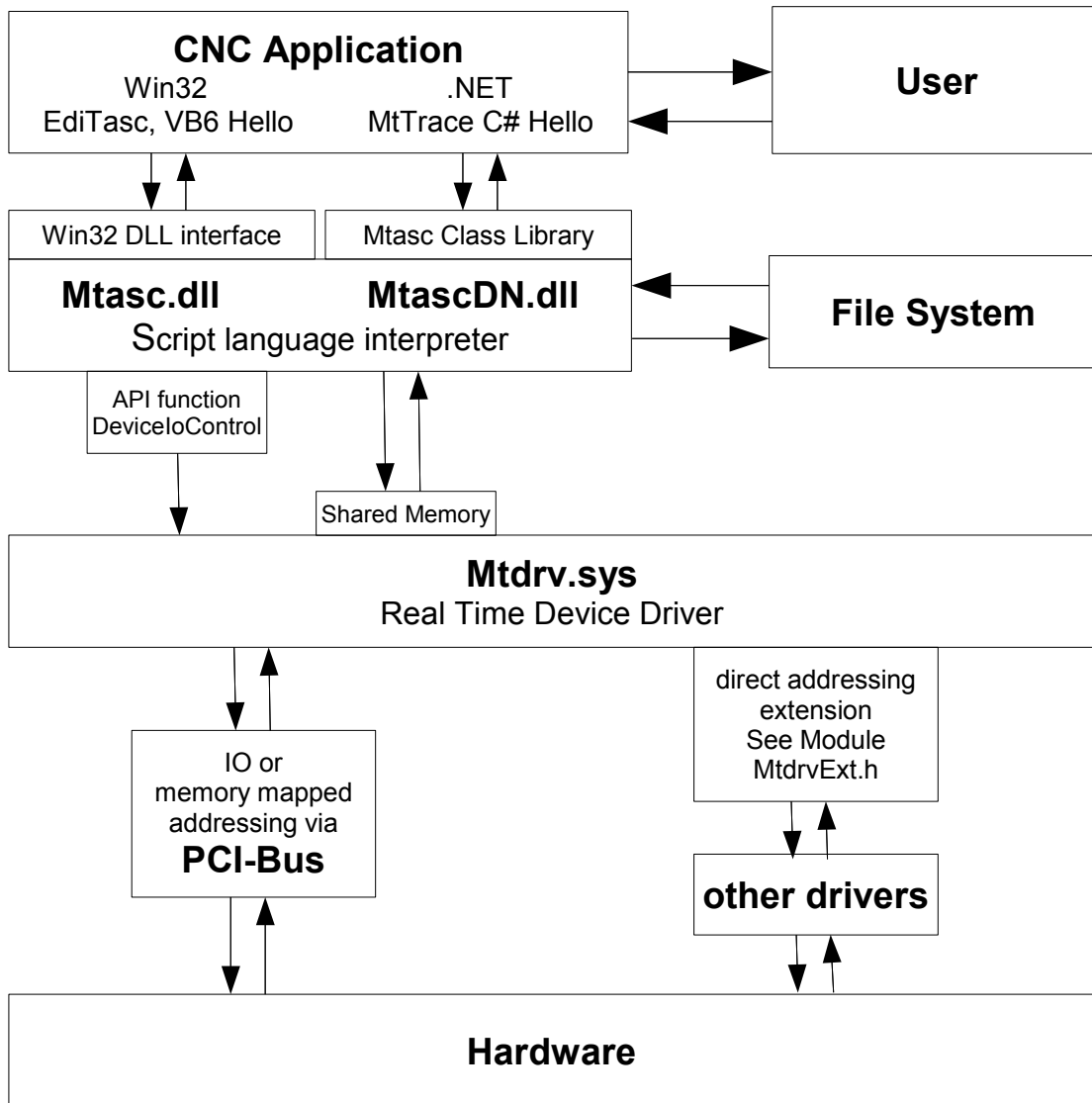
Copy the files created by you programming tool to the **RunFiles** directory or let the output path point there.

Explore EdiTasc first to get a feeling for the possibilities of the NC-Toolkit.

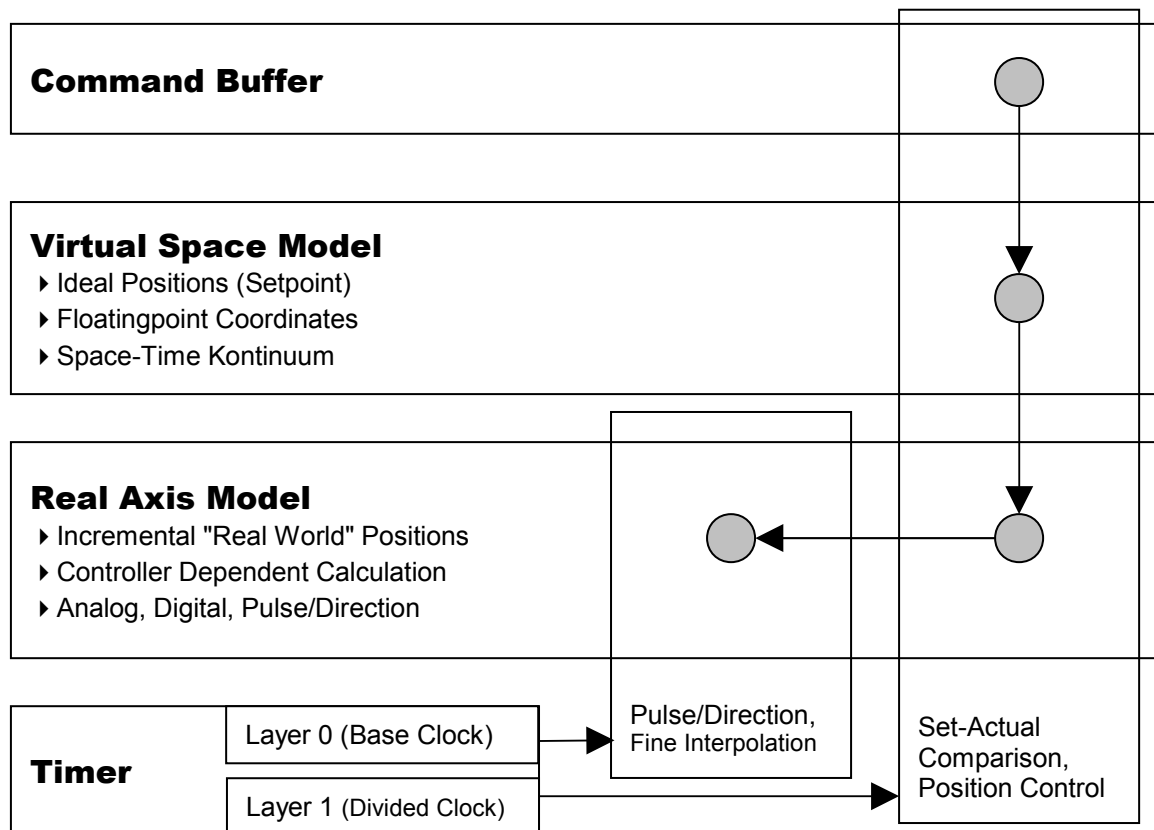
Setup copies the Device driver **Mtdrv.sys** into the **System32\drivers** folder and adds entries in the registry. This can also be done by the batch files **CopyDrv.bat** and **RegDrv.bat** in the **Driver** folder of EdiTasc or Toolkit . In case you want to repeat this process, just execute the appropriate batch file.

If you are working with different driver types or versions, you can switch the currently loaded driver calling **DrvStopStart.bat** in the appropriate driver folder. This will copy the new driver to the system directory, stop the current driver and start the new one. If these steps are successful, it is not necessary to reboot the PC.

The Architecture of EdiTasc / MTASC / Mtdrv



The architecture of the device driver (Mtdrv.sys)



The MTASC Script Language Interface of Mtasc.dll (Win32) / MtascDN.dll (DotNet)

Most commands are passed via the script language **MTASC** as string. The MTASC language is similar to C and Basic and is described in the MTASC manual.

The Hello projects show how to get all stuff initialized with minimum MTASC script environment.

If you want a graphic window to show the linear movement of X and Y, use the MTASC system variables **_ncMode** and **_grMode** to enable NC mode (moving the axes) or graphic mode (showing the movement) or both.

See the **MtExec** function below along with the MTASC manual for write access to these data.

See the **MtGet...** functions for read access.

Platform dependent Differences Win32 / DotNet

For Win32 (see samples in Visual Basic 6) the declarations of all exported functions are in the C header file MTASCDLL.H.

For DotNet (see samples in C# for Visual Studio 2005) you find the MTASC class library declaration in MtascDN.xml.

The functions exported by Mtasc.dll (MtascDN.dll)

The following functions are described for the Win32 platform only.

The interface for DotNet is very similar and easier to use because the context sensitive autocompletion (IntelliSense) of Visual Studio helps to find the suitable function or data name of the MTASC class.

Note: File MtascDN.xml is required for this autocompletion to work.

In the DotNet function names the preceding Mt... is omitted.

MtInit: Initialising MTASC

Syntax:

```
Long Return = MtInit (LPCSTR appPath, short lang);
```

Parameters:

appPath: Path to the current main program (EXE file) and the drivers INI file.

In MTASC script code, this path is available in system variable `_path`

lang: The **language for error messages**. Ascii-Code for 'G': German, 'E': English, 'C' for Chinese. In MTASC script code, the language is available in system variable `_language`.

Return value:

0 if successful, otherwise 1.

MtExec: Execute String as MTASC Command

This is the most important MTASC function. It expects a string as argument, which may contain any legal MTASC command that is to be executed. This is much like executing a text line from within the text editor of EdiTasc by pressing Ctrl+Enter. See EdiTasc manual for details.

Syntax:

```
MtExec (LPCSTR Command);
```

Command: A command string to be executed by the MTASC interpreter.

Note: A call to **MtExec** may not return immediately in one of these cases:

- The command string contains a **Wait** or **Sleep** statement. The call will return after the time has elapsed.
- More than **100 vector commands** are being sent to the driver. E.g. by MTASC function **fcall**, that passes execution to a file containing MTASC code, meaning that the driver's command buffer (a FiFo type buffer) gets filled up.
- There is an endless loop in the code. In this situation you can cause a break by issuing another **MtExec** call with a **MRESET** command or by changing variables that are responsible for the loop. For **MRESET** see below.

As long as an **MtExec** call is pending, the program should still react to user commands, since the MTASC interpreter allows reentrance. There is a reentrance limit which is controlled by the MTASC system variable **_SpinLimit**. It should not be 0 because endless loops could then let your program hang.

You can cancel any pending **MtExec** call by

```
MtExec "MRESET 1;" or
```

```
MtExec "MRESET ""Your Message"";" (string as argument of MRESET in Visual Basic).
```

If the argument string is empty, you avoid a message when aborting execution. Otherwise, the filename and line number will be displayed in case it was a file (called by **FCALL**).

Sometimes it may be necessary to pass short commands that **may not be interrupted** by other events like key or mouse messages. In this case place '@' at the beginning of the string. This is advisable for **MtExec** calls in background routines triggered by a timer.

MtManStep: Jogging the axes manually

Recommended for jogging by the keyboard.

Syntax:

Long **MtManStep** (short **iAxis**, short **dir**);

Parameters:

iAxis: Index of the axis to be moved. 0 = X-axis, 1 = Y etc.

dir: Direction. +1 = positive, -1 = negative.

Return value:

1 if successful, otherwise 0.

When called once the function moves the indicated axis at steps defined in Mtdrv.ini, entry **manVmin** (see EdiTasc manual, Appendix).

Repeated calls (about at auto repeat rate of the keyboard) increase the moving speed. The maximum is defined by **manVmax**. EdiTasc uses this jogging method with the AltGr + Cursor keys.

If many MvManStep commands have been sent due to pressing a key permanently, stop the movement of the axes by calling **MtManStep(-1, 0)** in the KeyUp event. This is not recommended if the key was only pressed once or twice. See **MvManCnt** in the Visual Basic project EdiTasc.

If limit watching is active and any axis exceeds the software limits (see MTASC commands **MLimit** and **MtGetStat**), MtManStep allows jogging back into the limit cube. This is the only way to move the machine when limits are hit. The ML, MF functions do not work then, unless you disable limit watching by executing an **MLimit '0'**; command.

MtGetData: **Read values from MTASC, allow error message**

MtGetDataE: **Read values from MTASC, return error code**

This function expects – like MtExec – an MTASC command as String. The string may contain any number of expressions. The result of the last expression is returned. Both functions do the same, except in case of an error, where MtGetData creates an error message while MtGetDataE sets a nonzero error code. The pointer “err” itself must not be 0.

Syntax:

```
Long Return = MtGetData (LPCSTR cmd, short* lpi, long* lpl, double* lpdbl,
                        LPCSTR str);
```

```
Long Return = MtGetDataE (LPCSTR cmd, short* lpi, long* lpl, double* lpdbl,
                        LPCSTR str, short* err);
```

Parameters:

cmd: String with MTASC command. The result of the last expression is returned in the corresponding pointer lpi, lpl, lpdbl or str. The data type determines which container is used.

lpi: A pointer to data type T_short.

lpl: A pointer to data type T_long.

lpdbl: A pointer to data type T_double.

str: A pointer to data type T_char or T_char-Array (STRING). A T_char is placed at the beginning of str. The length of str is defined by its terminating 0. On entry str must not contain any other 0. Be careful when using a static string as container, since the return string is copied including its terminating 0.

err: A pointer which points to the error code.

Return value:

An Integer telling you the data type:

0: Data type not supported (undefined or Arrays other than T_char)

1: T_short

2: T_long

3: T_double

4: T_char

5: T_char-Array (STRING)

Example for C:

Read text "HIGH" when input signal is 1, text "LOW" when input signal is 0

```
short  valS;
long   valL;
double valDbL;
char   valStr[] = "    ";
```

```
Ret = MtGetData
    ("if(inp 0x378 1){return \"HIGH\";}
     else          {return \"LOW\";}",
     &valS, &valL, &valDbl, valStr); // pointers to return data
```

Returns 5, indicating that ValStr contains the resulting String with text "LOW" or "HIGH" depending on the lowest bit of the printer port.

In the Visual Basic project EdiTasc you find a useful function called **MtGet** which allows convenient reading of all data types. It returns the data as data type Variant.

In C# you may create similar functions like **MtGetStr** or **MtGetDbl** etc.

MtDataRead: **Read values of complex MTASC variables**

MtDataSet: **Set values of complex MTASC variables**

These functions can be used to read ARRAYs or class objects with a single call. (Since Mtasc.dll 620.x data classes or structures are supported).

Syntax:

```
Long Bytes = MtDataRead (LPCSTR cmd, void* buf, long maxBytes,
                        MtDataType* type);
Long Bytes = MtDataSet  (LPCSTR cmd, void* buf, long maxBytes,
                        MtDataType* type);
```

Parameters:

Cmd: String with MTASC command. The result of the last expression is returned. It depends on the return value.

Buf: Pointer to data buffer to receive the data.

MaxBytes: Maximum size (in bytes) of the data buffer.

type: Points to an identifier to receive the data type. Type must have a valid value. See Mtascdll.h for details. *type is only used as return buffer. Its original value is ignored.

Return value:

The number of bytes written to buf.

Example:

Define a data structure to receive the most important status data from MTASC and the driver:

```
MtStatusX mtStat;
MtDataType type;

MtDataRead("MtsRead", &mtStat, sizeof(mtStat), &type);
```

MtGetStatus: **Read status data from MTASC and driver.**

This function replaces the obsolete `MtGetStat()`, which is no longer supported.

Syntax:

```
MtGetStatus (MtascStatus* mStat, MtdrvStatus* dStat);
```

Parameters:

mStat: Pointer to an instance of `MtascStatus` or NULL if not needed. See file **MtascData.h** for the commented declaration.

dStat: Pointer to an instance of `MtdrvStatus` or NULL if not needed. See file **MtdrvData.h** for the commented declaration.

See the **Watch sample project** for an illustration how to use `MtGetStatus()` and how to access the `MtdrvStatus` data.

Example Limit Watching

Call `MtGetStatus()` repeatedly for general status watching:

```
MtascStatus mStat;
MtdrvStatus dStat;
MtGetStatus(&mStat, &dStat);
```

The status of the software limit appears here:

`dStat.wFlags[i].dirNeg` for the negative directions and
`dStat.wFlags[i].dirPos` for the positive directions.

Message Boxes and Error Handling in Mtasc.dll

By default, a message box will be displayed in case of an error. After that, the interpreter is aborted.

If you want these and other message boxes to be modal to some window of your application, assign its window handle to the MTASC variable `_hwnd`.

When an error in any MTASC script occurs, the following error handler is called. Its default definition is:

```
(MacroDef "_ErrorHandler") =
    "MsgBox _sysMsg.errText \"MTASC message\" MB_OK;";
```

You can define your own error handler by overwriting this macro.

E.g., the following version defined in *System.ts* adds the writing to a log file.

```
(MacroDef "_ErrorHandler") =
"MsgBox _sysMsg.errText \"\" & (GetIniTxt "System" "ErrMsgCapt") & "\"
MB_OK;
  LogWrite \"MsgBox: \" & _sysMsg.errText;
";
```

The Motion and Graphic Interface (AGI)

Access to the motion commands in the **coordinate vector space** is provided by a special class called **CMtAGI** (MTASC Axes and Graphic Interface). We just will call each instance of this class an "AGI".

The properties are defined by member functions and data of the AGI. The application can define more than one. For **multi channel CNC** applications there will be defined one for each channel.

In *System.ts* there is a container `mtag` for up to **4 channels** or **axis groups**, which means the same. The **maximum number of channels** currently supported is 4 (= `mtagMax` in *System.ts*).

Usually, the thread that addresses an AGI is attached to it by MTASC function `Thread_SetAgi`. After it is attached the thread can use the standard motion commands like `mL`, `mf` etc.

The AGI provides a routing to

- an actual physical motion via the real-time device driver `Mtdrv.sys`

and / or

- graphic display into one of the applications windows

The AGI's are created in *System.ts* by this line:

```
mtag[i].agi = &(gcnew CMtAgi (drv.fag i).vDim (drv.fag i));
```

The Real-Time Device Driver Interface (DRV)

Access to **real-time motion** is provided by an MTASC object of type **CMtdrv**.

In *System.ts* a global object named **drv** is created by:

```
drv = gcnw CMtdrv 1 n;
```

where *gcnw* is a MTASC function to create objects via a class name (here **CMtdrv**) followed by arguments for the constructor.

Argument 1 means that this process wants an active driver access mode, 0 would be passive.

n is the number of axes to be used.

Important members of *drv* are *drv.fag* and *drv.laa*, which represent 2 levels within the driver.

drv.newFag is a member function to create up to 4 axis groups.

drv.fag gives access to the axis groups.

drv.laa is a class for position control close to the hardware of each axis.

See the MTASC manual for further details.

The Graphic Interface (G3)

Access to graphic is also provided by the MTASC class **CMtAgi**.

Initializing and setting the properties is done by member functions beginning with *g3 . . .*,

This command in *System.ts* initializes the G3 interface of an AGI:

```
(*mtag[i].agi).g3open;
```

In the current version there is an additional DLL function required (*g3SetWc2Dc*) to define the transformation from Object or "World Coordinates" to "Device Coordinates". This function is only available for Win32 (*Mtasc.dll*) and not yet for the DotNet Platform (*MtascDN.dll*).

See Visual Basic 6 sample project *HelloGraph*.

The Hello Sample Programs

We deliver 3 Samples:

2 for Win32 written in Visual Basic 6: **Hello** and **HelloGraph**.

1 for DotNet written in C# under Visual Studio 2005 Express: **Hello**.

Initializing with Hello.ts and HelloGraph.ts

The programs, that do not use Graphic call MTASC script file **Hello.ts**, the one other one (HelloGraph) call **HelloGraph.ts**. The only difference is the initial setting of `_AppFlags`, `_NcMode` and `_grMode`. They call *System.ts* for the generic stuff and *EtInit_xx.ts* for the application specific settings.

`_AppFlags` is a global long integer that tells *System.ts* how to initialize the environment.

Other Programming tools

all languages capable of calling functions in DLLs can be used.

In future more samples in C# will be available.

Debugging with MtTrace

MtTrace is an important application designed for debugging other applications. It can be used to debug MTASC code in Win32 as well as DotNet applications.

It is written in C#, so it needs *MtascDN.dll* in the same folder. *MtTrace.exe* should always be started from the same folder as the application to be debugged, because it calls *System.ts* as well.

MtTrace connects to the application via shared memory, so it can actually see all variables and thread of the debuggee.