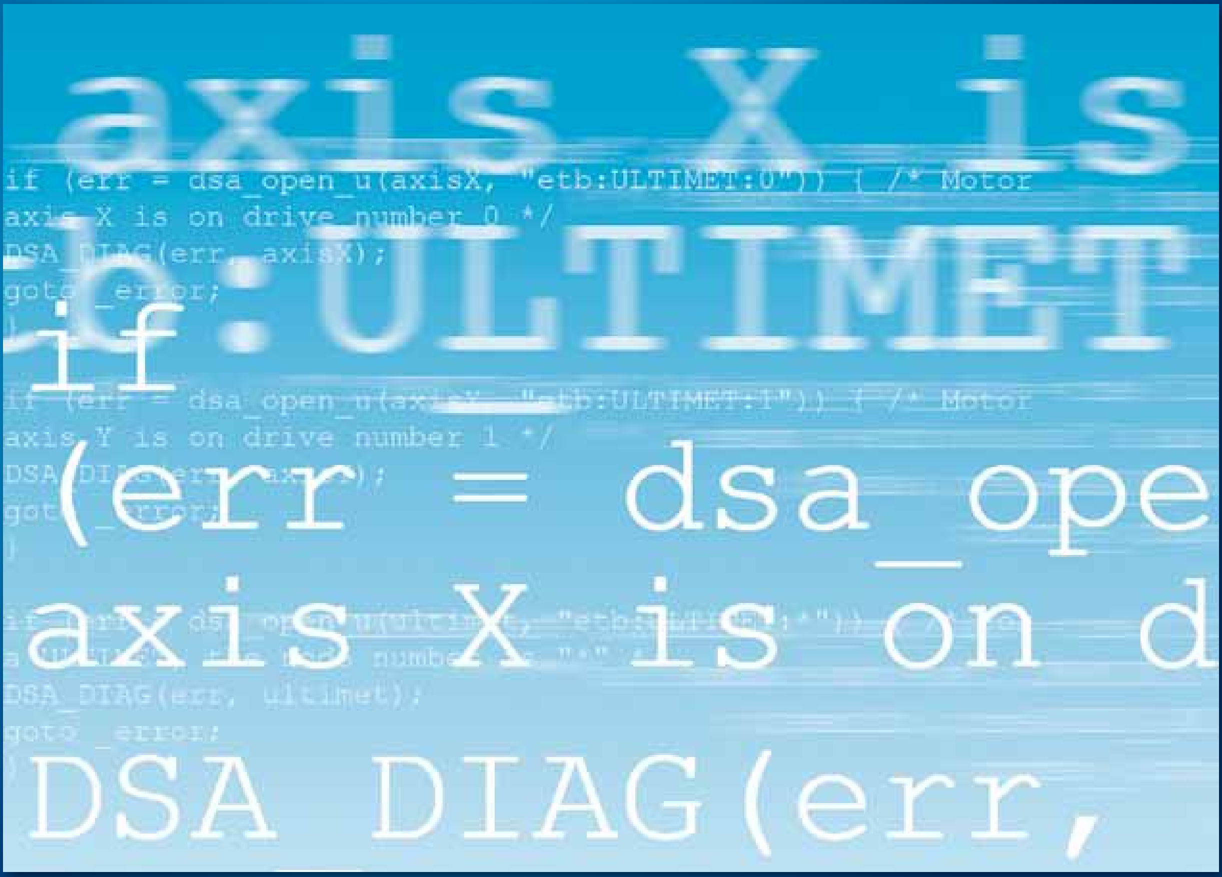

axis X is
if (err = dsa_open_u(axisX, "etb:ULTIMET:0")) { /* Motor
axis X is on drive number 0 */
DSA_DIAG(err, axisX);
goto _error;
b:ULTIMET
if
if (err = dsa_open_u(axisY, "etb:ULTIMET:1")) { /* Motor
axis Y is on drive number 1 */
(err = dsa_ope
axis X is on d
DSA_DIAG(err, ultimet);
goto _error;
DSA DIAG(err,

EDI 4
(ETEL Device Interface)

User's Manual

Version A

MOTION TECHNOLOGY

Table of contents

THIS PAGE IS INTENTIONALLY LEFT BLANK

Record of revisions concerning the EDI4 manual:

Document revisions		
Issue (x)	**Date**	**Modified**
Ver A	03.05.13	First version

Documentation concerning the EDI4:

- **EDI4-User-VerA** **(EDI4 principle & operation)**
- **HTML Reference Manual** **(File list & data structure in html.bat)**

Caution: **As HTML documentation is generated for each EDI4 release, it is the only up-to-date reference manual.**

All the functions present in the HTML Reference Manual and mentioned under ’Internal use only’, are available for the customer but neither documented nor supported by ETEL support.

1. Introduction

1.1 Acronyms

Abbreviation	Definition
API	Application Program Interface
DLL	Dynamic Link Library
EDI	ETEL Device Interface
ISO	International Standards Organization
URL	Uniform Resource Locator

Remark: The updates between two successive versions are highlighted with a modification stroke in the margin of the manual.

1.2 Glossary

The following terms are constantly used in this manual. It is essential to know these definitions before reading this manual.

AccurET-family	Devices belonging to the AccurET family and communicating through TransnET (AccurET and UltimET).
API	Application Programming Interface: a set of software functions to have access to a system.
Device	A device can be a controller, a multi-axis gateway, or a multi-axis motion controller.
DLL	A «dynamic link library». A .dll file contains one or several functions compiled, linked, and stored separately from the processes using them. The operating system maps the DLLs into the address space of the process when this process starts up or as they are called, while it is running. The process then executes the functions in the DLL.
EDI4.xx	Set of libraries supporting AccurET-family products, **but not DSC-family nor DSB-family any more.**
Increment	ETEL's devices use a large number of physical quantities (position, speed, acceleration, force, time, electric current, ...). These quantities are represented by 32 or 64 bits integers or float, with units specific to the devices (UPI, USI,...). These units are called 'increments', hence the representation by increments' term used to represent the physical quantities.
ISO unit	The international system of units used to give the physical quantities. The basic units m (meter), s (second), kg (kilogram), A (Ampere) and K (Kelvin). There are also their derivatives: m/s, m/s^2, m/s^3, N (Newton) = kg * m/s^2, W (Watt) = N * m/s, V (Volt) = W/A,... **Remark:** For EDI, the temperature is always given in °C (Celsius) and the angular positions in turns.
Multi-axis gateway	An ETEL device that is able to control several position controllers, without interpolation capabilities such as the UltimET gateway.
Multi-axis Motion controller	An ETEL device that is able to control several position controllers, with interpolation capabilities such as the UltimET light.
TransnET	Very high speed proprietary communication bus. This bus is used between to communicate between AccurETs and between AccurET and UltimET.
OS	Operating System. It can be Windows 2000, XP, Windows 7.

Object	When we talk about 'object', we refer to the definition widely used in the object-oriented programming. When we work in C, an object is nothing more than a structure containing a series of information. The classical example of C object is the FILE structure of the standard library.
Position controller	An ETEL digital controller such as AccurET family.

1.3 Presentation of this manual

1.3.1 Intended readers

It is assumed the user is familiar with software programming, in particular in C language. Software design and architecture aspects must be mastered by the developer intending to use this DLL interface. In particular integrating a DLL into a software project should not be an issue.

Also the user must be familiar with ETEL position controllers and their usage as described in the corresponding Operation and Software Manual. If the application requires a multi-axis motion controller, the developer must refer to its User’s Manual to understand how to use it.

Remark: The updates between two successive versions are highlighted with a modification stroke in the margin of the manual.

1.3.2 Contents

This document presents all the libraries of the EDI package by putting the emphasis on the DSA library. This library is the only one that is necessary to program the controllers from a PC. The other libraries are mainly used by the DSA library itself.

After the present chapter, another describes the general package features including its structure, supported operating systems and communication buses and file description.

The ensuing chapters are ordered in pretty much the same way as one would go about developing an application using ETEL controllers. They detail how to use the library by referring to an illustrating example that is the same throughout these chapters. This guiding ends up being close to what one would expect such an application to look like.

Throughout the example, the basic and most common operations will be reviewed. The complete list of operations provided by the package can be found in the **’HTML Reference Manual’** grouped by feature.

Found in the appendices, are:

- a table giving for each controller command, the corresponding DSA function when it exists
- a similar table giving for each device register, the corresponding access functions when they exist
- a description of the error output
- a description of the examples provided, other than the other analyzed throughout this manual
- a description of the changes needed to bring an application using EDI3xx to use EDI4.xx

2. Package features

2.1 Presentation

The ETEL Device Interface (EDI) is a set of libraries which enable the communication with the ETEL controllers, or multi-axis motion controllers and the access to their functionalities. ETEL began to develop EDI in 1997. Three major versions have been developed since. Here follows a list of these versions, of ETEL's products they support and which windows environment they have been developed from.

EDI version	First release	Last release	DSB family	DSC family	AccurET family	Development environment	EDI relative documentation
EDI1.xx	1.00 (1998)	1.00d (2000) Obsolete	✓			Visual studio	None
EDI2.xx	2.00A (22.02.2001)	2.28E (2010) Obsolete	✓	✓		Visual studio 97	From EDI-user-b.pdf
EDI3.xx	3.00A (17.03.2009)			✓	✓	Visual studio 2005	From EDI-user-c.pdf
EDI4.xx	4.01A (03.06.2013)				✓	Visual studio 2010	From EDI4-User-VerA.pdf

Remark: **Refer to §13.2 for more information about the changes between EDI30 and EDI4.**
There is no direct interface with the multi-axis gateways as these devices are only a communication means and they do not have any specific commands or registers to read or set. There is no access to them from the EDI package. Henceforth, unless otherwise explicit, the terms device or controller exclude this category.

All the functions contained in those libraries are often called API (Application Program Interface).

The main role of that set is to:

- Generate the communication through several existing communication buses such as the PCI bus, the TCP/IP LAN and USB.
- Give the user a group of functions allowing him to have access to all the functionalities of the ETEL products.
- Give an homogeneous interface whatever the product and the firmware version.
- Manage the conversion of the products' internal units (increments) in order to enable the user to work with conventional units (ISO).

Those libraries are totally developed in C and can be used from other common programming languages such as C++, and C# in version 4 of the .NET framework. There is a DLL to be used from C/C++ applications and a class library for .NET framework version 4. They can be used from other languages with restrictions but most of the time the interface needs to be developed.

Most of the applications calls the high level library called DSA which afterwards calls the other libraries of the EDI package. This library is able to manage all of the latest controllers and multi-axis boards.

There is a `readme.doc` file that gives the installation and compilation guidelines for the supported operation systems as well as the list of files that are used and where they are to be found.

2.2 Supported Operating Systems

EDI has been first developed for Windows systems. Along the years, some other OS have been supported (under some restrictions), especially for the EDI2.xx version. Here is a summary of supported OS.

EDI1.xx	EDI2.xx	EDI3.xx	EDI4.xx
Windows 9x / NT / 2000 / XP / Vista 32-bit	Windows 9x / NT / 2000 / XP / Vista 32-bit QNX4 (ISA, PCI, TCP/IP) QNX6 (ISA, PCI, TCP/IP) Linux (ISA, PCI, TCP/IP) and RTX (PCI)	Windows 2000 / XP / Vista 32-bit Windows 7 32-bit and 64-bit (as 32-bit application)	Windows 2000 / XP / Vista 32-bit Windows 7 32-bit and 64-bit

2.3 DLL (Dynamic Link Library)

The EDI package for Windows 2000 / XP / Vista / 7 is a set of 'dynamic link libraries' commonly known as DLL.

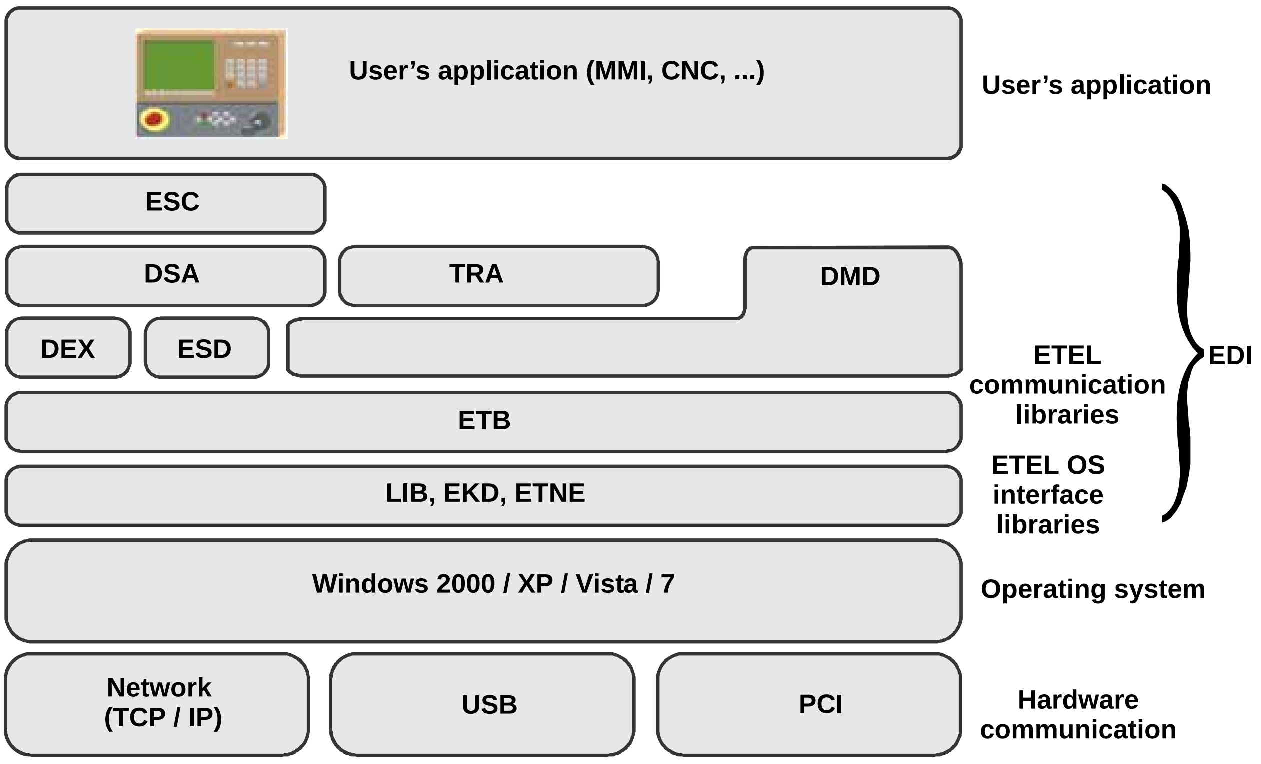
This kind of library is a standard for Windows and all the programmers have to deal with it. Actually, all the Windows' libraries are in this format and all the compilers available for Windows use this kind of libraries. The main points characterizing a DLL are:

- a Windows standard
- its format is known by all the compilers
- A DLL is dynamically linked. It allows the user to modify or to update it without compiling again the application using it.

To use the EDI package with Windows, it is important to correctly understand the functioning of a DLL. Please refer to the manuals of your compiler to know in detail how the DLLs work and how the compiler is able to manage them. Be careful to understand where the DLLs have to be stored and where the application goes to get them.

2.4 Package structure

The EDI package is in fact a collection of several libraries. But the API for programming ETEL controllers is provided entirely through the DSA library. The other libraries are only used by the top-level library DSA itself, and the developer does not need to call the others.


User's application (MMI, CNC, ...)
User's application
ESC
DSA
TRA
DMD
DEX
ESD
ETB
LIB, EKD, ETNE
ETEL communication libraries
ETEL OS interface libraries
EDI
Windows 2000 / XP / Vista / 7
Operating system
Network (TCP / IP)
USB
PCI
Hardware communication

The package is in fact made up of several DLLs but the DSA library itself constitutes the entire API necessary to the user. The DMD library offers named constants for command and register names which are interesting to use for code clarity and maintainability. Also, exceptionally the TRA library is used for downloading data files to the devices and ETB for downloading firmwares. The ESC library allows the compilation of a C-like AccurET-UltimET sequences into a special ETEL format. The ESD library extracts machine code from a compiled file and builds a file which can be downloaded into an AccurET or an UltimET. The DEX library allows the exportation and importation of Time-based (Scope), Frequency-based (Identification), Mapping and Scaling data into or from a file.

For reference, here follows the breakdown of the tasks among the libraries:

ESC library:

- Allows the compilation as well as the download and the upload of the sequence of the AccurET family. This library uses a help library called assert40c.dll

DSA library:

- Manages a single device, a device group, interpolation device group objects.
- Generic functions for using all the functionalities of the devices.
- Performs conversions between the device internal units (increments) and the ISO units.
- Transparent to communication configuration.
- Manages devices on several communication buses at the same time.
- Specific functions to set and get the main device registers.
- Specific functions to execute most commands used on the devices.
- Asynchronous and synchronous functions.

DMD library:

- Manages meta-information objects.
- Stores min / max and default values for each register.
- Stores the list of spaces / indexes and subindexes available.
- Translates error codes / status / commands and aliases.
- Directly generated from the position controller database.
- Stores information for each device and firmware version.

TRA library:

- Manages the translation between the ETEL language (terminal commands) and the ETCOM records internally used by the controllers.

ETB library:

- Manages the communication buses and logical ports.
- Handles the PCI, USB and TCP/IP transparently.
- Handles the normal and boot communications.
- Handles the status update (refer to §7.)
- Multiple logical port message queuing.
- Registering of event handlers (callback).

LIB and EKD libraries:

- Help libraries for hardware and operating system specific access.

ETNE library:

- ETNE library allows communication to ETEL ETND deamon which is a process acting as a TCP/IP server. By running this process, you allow other network user to connect to your local ETEL hardware.

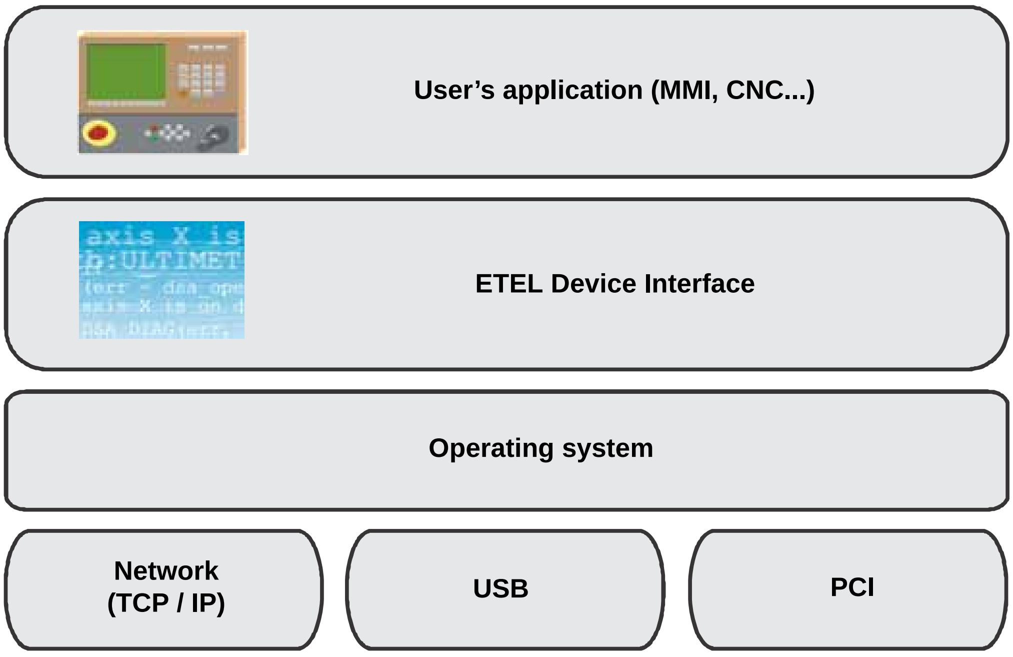
2.5 Main characteristics

The main characteristics of the software are:

- Implemented in ANSI C
- Can be used from C, C++ (Visual Studio 2010 and further), and C# int the .NET (framework 4) environment
- Common source files for multiple platforms and operating system
- C++ wrapper classes included in header files
- Thread-safe implementation
- Object-oriented architecture

2.6 Supported communication buses

2.6.1 Description



At the moment, the EDI package supports the following communication bus:

Communication bus	ETEL product
PCI	UltimET PCI
TCP/IP	AccurET and UltimET TCP/IP
USB	AccurET

With the package comes an ETEL network process call «ETND», which is a daemon running on a remote machine. This distant machine is connected to the controller system via any of the other aforementioned buses. The application can connect to this remote «ETND» on a distant machine via the a TCP/IP LAN and have access to the remotely connected controllers.

2.7 Versions numbering

All the ETEL softwares are endowed with a version number identified as follow:

<digit 1>.<digit 2><digit 3><alphabetical character 1>

For example, a version number can be: 1.00A, 3.15B, 2.00D, etc... An "X" can follow the version number if this version is under construction. A special numbering is used for the preliminary versions (alpha or beta version). Each EDI package and library have their own version number. There is no direct link between both version numbers. For the libraries, the first digit (<digit1>) representing the major version index of the version number is present, followed by a 0, in the file name of this library. To know the exact version of a library, open windows explorer and right click on the wanted file to select 'Properties'. To get the general EDI Package version the DSA library provides a function called `dsa_get_edi_version()`.

2.8 Files for Windows

For each library, there are several files necessary for the compilation and the execution of the code. For the DSA library, the following files are used:

`dsa40.h`	Header file necessary during the compilation
`dsa40c.lib`	Static library necessary during the link
`dsa40c.dll, tra40c.dll, etb40c.dll, dmd40c.dll, esd40c.dll, ekd40c.dll, (or ekd40c_64.dll if you are on Windows 7 64 bits OS),lib40c.dll, dex40c.dll`	Dynamic link library necessary during the execution
`esc40c.dll, assert40c.dll`	Dynamic link library necessary to compile
`etne40c.dll`	Dynamic link library necessary to allow TCP/IP connection through ETND daemon
`FTBUSUI.dll, FTD2XX.dll`	Dynamic link library necessary to connect through USB
`wdapi910.dll, wdapi1021.dll (or wdapi1021_32.dll if you are on Windows 7 64 bits OS) wdapi1110.dll (or wdapi1021_32.dll if you are on Windows 7 64 bits OS)`	Dynamic link library necessary to connect to UltimET PCI

The debug versions, necessary during the development, are added to these files:

`dsa40cd.lib`	Static library, debug version
`dsa40cd.dll, tra40cd.dll, etb40cd.dll, dmd40cd.dll, esd40cd.dll, ekd40cd.dll, (or ekd40cd_64.dll if you are on Windows 7 64 bits OS), lib40cd.dll, dex40cd.dll`	Dynamic link library, debug version
`esc40cd.dll, assert40cd.dll`	Dynamic link library necessary to compile
`etne40cd.dll`	Dynamic link library necessary to allow TCP/IP connection through ETND daemon
`FTBUSUI.dll, FTD2XX.dll`	Dynamic link library necessary to connect through USB
`wdapi910.dll, wdapi1021.dll (or wdapi1021_32.dll if you are on Windows 7 64 bits OS) wdapi1110.dll (or wdapi1021_32.dll if you are on Windows 7 64 bits OS)`	Dynamic link library necessary to connect to UltimET PCI

The files contain the name of the library followed by a number which reflects the first digit of the version number. One or several letters enable the differentiation of the implementation types.

The .lib files are simple interfaces between the application and the DLL. Their size is then relatively small because the code of the functions is stored in the DLL file. These files are given in the COFF format for Visual studio 2010 C++.

For the libraries other than the DSA one, the files are more or less the same. These libraries are internally used by the DSA library or for special applications. Normally, only the DSA library is directly used by the customers. ETEL does not give any support concerning the use of other libraries such as TRA, DMD, ETB, ESC, ETNE, EKD, LIB, ESD and DEX.

3. General guidelines for programming with the EDI package

3.1 Object oriented

The DSA library as all the other libraries of the EDI package, is totally implemented in C and follows the object-oriented philosophy. The work is done with objects represented by C structures. Hence, an object corresponds to each device, and all the functions concerning this device receive a pointer on this object (this structure) as parameter. The content of the structure is not visible to the user.

To illustrate that, look at the functions of the standard C library which give access files. Here also, an object (FILE structure) is created for each file the user wants to manage and all the functions doing an operation on a file receive as parameter the pointer on this object. It is the case for fprintf() which receives as first parameter the pointer on the FILE structure.

To have access to a position controller, a pointer on the DSA_DRIVE object must be created in the following way:

```
DSA_DRIVE *drive1 = NULL;
```

It is important to assign this pointer to the NULL value. To avoid wrong manipulations of the pointer, the library refuses to create an object if the pointer is not at the NULL value.

Once the pointer declared, the object must be created and the pointer affected. It is done in the following way:

```
err = dsa_create_drive(&drive1);
```

As mentioned above, most of the DSA functions return an integer which represents the error code. To detect a possible error, the creation of a drive object can be done in the following way:

```
DSA_DRIVE *drive1 = NULL;
int err;

err = dsa_create_drive(&drive1);
if (err != 0) {
    printf("cannot create drive object\n");
    exit(1);
}
```

The creation of an object, necessary for the communication with the UltimET, is done in a similar way:

```
DSA_MASTER *UltimET = NULL;
int err;

err = dsa_create_master(&UltimET);
if (err != 0) {
  printf("cannot create UltimET object\n");
    exit(1);
}
```

3.2 Groups

In the previous paragraphs, the first parameter of each DSA library function represents the device on which the user wants to do the operation. In practice, this parameter is a pointer on an object which includes all the information of the device in question.

The user often needs to send a command or to do an operation on several controllers at the same time. It is also necessary when the user wants to start synchronized movements. For that purpose, the DSA library has the possibility to create groups of devices. Afterwards, the user can perform operations on these groups instead of performing them on a simple device.

The DSA library enables the definition of the following groups:

Group	Type	Description	Product
Device group	DSA_DEVICE_GROUP	Group of several position controllers or multi-axis motion controllers of the same family.	AccurET, UltimET
Drive group	DSA_DRIVE_GROUP	Group of several position controllers of the same family.	AccurET
UltimET group	DSA_MASTER_GROUP	Group of several multi-axis motion controllers of the same family.	UltimET
Interpolation group	DSA_IPOL_GROUP	Group of several controllers with its multi-axis motion controllers which generate interpolated movements with the controllers of the same family.	AccurET, UltimET

To create a group, the process is similar to the one used to create a device. A pointer on the object corresponding to the desired group must be first created:

```
DSA_DRIVE_GROUP *group1 = NULL;
```

Like for the drive and UltimET objects, the pointer must be assigned to NULL before creating the object. Afterwards, the following function must be called:

```
err = dsa_create_drive_group(&group1, 2);
```

Compared with the creation of a drive, the creation of a group requires an extra parameter which is the number of devices that the user wants to put in the group. It is in a way the size of the group which is equal to 2 in the above example.

Once the group created, the devices belonging to it must be assigned in the following way:

```
err = dsa_add_group_item(group1, drive1);
err = dsa_add_group_item(group1, drive2);
```

To do so, both `drive1` and `drive2` objects must have been already created. Instead of allocating a device, it is possible to assign another group. For example, the following group can be created:

```
dsa_create_drive_group(&group2, 2);
dsa_add_group_item(group2, drive3);
dsa_add_group_item(group2, group1);
```

In the above example, a group of position controllers (DSA_DRIVE_GROUP) has been created. Therefore, only DSA_DRIVE, DSA_DRIVE_GROUP, or the DSA_IPOL_GROUP objects can be assigned to it. If the user wants to include the UltimET in a group, the DSA_DEVICE_GROUP group type must be created.
A group of position controllers can include position controllers which are not on the same communication bus. On the other hand, in a DSA_IPOL_GROUP, the position controllers are only accessible via the same UltimET which means that there are connected to the same TransnET (Refer to the UltimET User's Manual for more details about the interpolation group).
In most cases, a group can be used exactly like a device. It is then possible to use a group as the first parameter of the DSA functions as in the following example, to send a «power on» command to all position controllers of group 1:

```
err = dsa_power_on_s(group1,...);
```

A DSA_IPOL_GROUP can also be used where a DSA_DEVICE_GROUP or DSA_DRIVE_GROUP can, like:

```
 DSA_IPOL_GROUP *igroup = NULL;
dsa_create_ipol_group(&ipol, 2);
 dsa_add_group_item(igroup, drive3);
 dsa_add_group_item(igroup, group1);
 err = dsa_power_on_s(igroup,...); /*send power on to drive 1, 2, and3*/
```

But a DSA_DEVICE_GROUP cannot be used everywhere a DSA_IPOL_GROUP can. The function starting the interpolation mode only accepts a DSA_IPOL_GROUP as parameter:

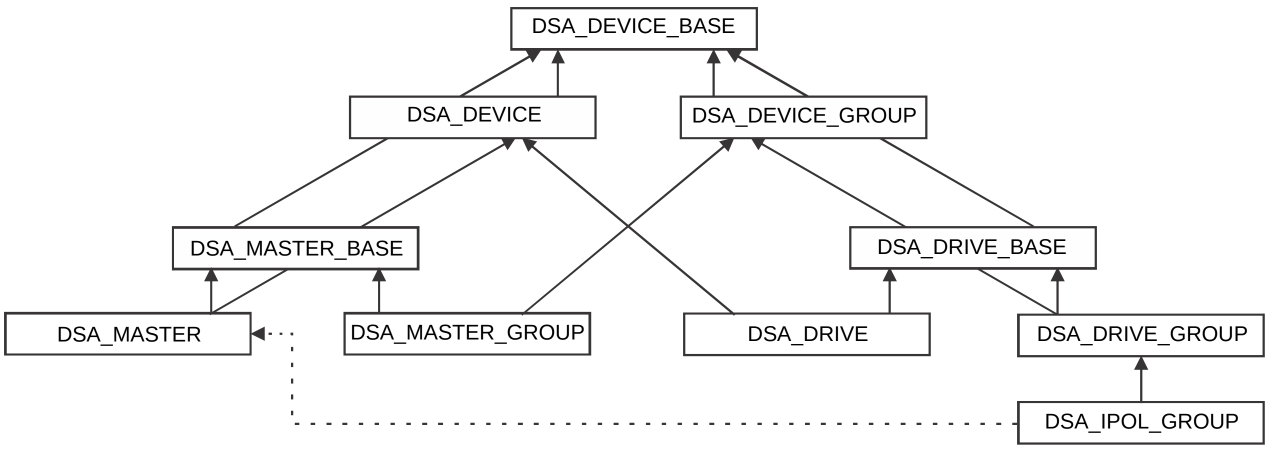
```
        dsa_ipol_begin_s(igroup,...);/*but not dsa_ipol_begin_s(group1,...)*)/
```

Also, it is not possible to read a register on a group and so the following example cannot be done:

```
         err = dsa_get_registre_s(group1,...);/* WRONG */
```

See §3.3 for more details about the hierarchy of the different objects and which type of object is accepted by each DSA library function.

3.3 Inheritance



As seen previously, the DSA library functions have always, as first parameter, an object which represents a device or a group of devices. Nevertheless, it is not possible to do any operation on any object.

For example, the power of a position controller can be switched on and off, but it is not possible to do it with a UltimET. The dsa_power_on_s() will hence accept the objects of DSA_DRIVE and DSA_DRIVE_GROUP type but not the DSA_MASTER type.

In the same way, the reading of a register can be done on only one device at the same time. The dsa_get_register_s() function will then accept the objects of DSA_DRIVE and DSA_MASTER type but not the DSA_DRIVE_GROUP type.

The objects of the DSA library are organized in a hierarchy which defines the relations of heritage commonly used in oriented-object programming. This hierarchy is illustrated below.

On the above figure, the DSA_DRIVE object inherits the DSA_DEVICE object which means that the DSA_DRIVE «is» a DSA_DEVICE or in other words that the DSA_DRIVE can be used instead of a DSA_DEVICE. If a DSA function accepts a DSA_DEVICE as parameter, it will automatically accept a DSA_DRIVE too which is the case for the dsa_get_register_s().

From these relations of heritage and the prototype of the function, it is easy to know to which object is applied.

3.4 Functions

There are two types of functions provided in the EDI package: those that return when the execution of the feature it implements is finished - called synchronous functions - and those that return immediately, without waiting for the task they have started to terminate. This last type of function is called asynchronous. Usually, for each function implementing a specific feature, their are the two types of functions. They are differentiated by appending to their names:

- **_s** for synchronous functions
- **_a** for asynchronous functions

For example, to switch the power on in the motor, we can either use `dsa_power_on_s(...)` or `dsa_power_on_a(...)` depending on wether we want the execution flow in our application to wait for the end of the «power on» or not to do other things.

The first parameter of each function is always the DSA_DRIVE or DSA_MASTER object which represents the device on which the operation will be done.

3.4.1 Synchronous functions and time-outs

These functions are called by the user and their execution ends once the desired operation is finished.

For example, when the `dsa_get_register_s()` function is called, it asks the device the value of a register and waits for the answer. Once the device has returned the value to it, the function ends by stepping down in favour of the user.

The synchronous functions stop then the execution of the program until the end of the current operation. It stops the user from doing another operation before the current one is finished. This can be bypassed by using several threads or asynchronous functions (see §3.4.2).

In the DSA library, the synchronous functions are characterized by a '**_s**' at the end of their name and by a 'timeout' parameter which is always the last one. The timeout is the maximum time allotted for the operation to finish. This time is always given in millisecond. To use the synchronous version of the «power on» function, we would write:

```
if (err = dsa_power_on_s(drive, 10000)){
     DSA_DIAG(err,drive);
     goto _error;
}
```

The call of a function ends when the operation is finished or when the time-out has elapsed. In the above example, the function ends when the motor is switched on and the control part (regulation) is activated, or when the process lasts more than 10 seconds. In that case, the function returns an error.

For the operations where the execution time does not depend on the position controller configuration or on the application, the default time-out can be used by giving the DSA_DEF_TIMEOUT (which will be explained later on).

The 'timeout' parameter allows the limitation of the maximum time that the function uses to end an operation. After this time, if the operation is not finished yet, the function ends by returning the DSA_ETIMEOUT error.

If the user does not want a timeout which means there is no time limit to do the operation, the INFINITE value can be assigned to the timeout parameter.

The functions such as `dsa_wait_movement_s()` need a time of execution which depends on the user's application. For functions like these, the use of the default timeout does not make sense.

3.4.2 Asynchronous functions and callbacks

During the development of an application, we mainly use synchronous functions. Nevertheless, in a few cases the use of synchronous functions is not ideal or requires more threads. For example it is the case when the user wants to monitor the status of a position controller at the same time than he waits for the end of a movement. In this case, the use of asynchronous functions is interesting. As mentioned above, the synchronous functions have a name ending with '_s' and a last parameter called 'timeout'. There is an asynchronous function for every synchronous function. Their name ends with an '_**a**' instead of an '_**s**' and the last parameters are a pointer to a function and a generic pointer (`void *`) instead of a timeout. For example, powering on a motor using the asynchronous version of the implementing function, we would write:

```
DSA_CALLBACK do_something_on_poweron_end(DSA_DEVICE_BASE *dev, int err,
void *param);
long[4] params;
...
if (err = dsa_power_on_a(drive,do_something_on_poweron_end,params)){
     DSA_DIAG(err,drive);
     goto _error;
}
```

The asynchronous functions start an operation without waiting for the end. Their execution is then really fast. After having started the operation, the user can execute other operations without starting another thread.

When the user calls an asynchronous function, he must give it a handler as parameter which is a pointer on a function often called 'callback'. This function is called by a thread within the library once the operation is finished.

For example, if the user calls the `dsa_get_register_a()` function, it ends before the value of the register is returned by the device. Once the value returned, the library calls the handler by giving it the value as parameter.

Besides the handler, the user has the 'param' parameter that he can use as he wishes. This parameter is not interpreted by the library and is given to the handler when it is called.

3.4.3 Generic functions

All the operations that can be done on a device boil down to three operations:

- To send a command with zero, one or several parameters
- To set the value of a register
- To read the value of a register

On top of these three basic operations, there is the status management which will be detailed in §7.
Most of the operation available for the user from the DSA library are special cases of one of these three basic operations. It means that through these three operations it is possible to have access to all the available functionalities in a device. It is for this reason that the DSA library has a set of functions - termed *generic* - for doing these three operations. There are more than three not only because there is the synchronous and asynchronous version of each but also because there are different functions depending on the number and type of the parameters of the commands, and depending on whether the value of a given register is wanted in ISO units or in controller units. Refer to the **’HTML Reference Manual’** for the complete list.

However, the generic functions can be tedious to use as they require that one knows exactly either the command’s syntax and parameters or the registers that need to be set to implement an operation. The EDI package provides functions that implement an operation without having to know which command to send or register to read and write. These are called specific functions (see next paragraph) and exist for operations routinely performed on ETEL devices.

Generic functions remain useful to have access to the functionalities for which there is no specific function. For example, the DSA library does not have any function to reset the position controller (RSD). To do so, the user has to use the generic functions.

3.4.4 Specific functions

Operations can be executed on the devices once an associated object has been created and the communication opened. Specific functions implement the most routinely used operations.

Remark: EDI4 has been cleaned and old functions which were accessing to DSB or DSC have been remove. Refer to chapter §13.1 for the list of these functions.

For example, to switch on the position controller, the following function must be called:

```
err = dsa_power_on_s(drive1, 10000);
if (err != 0) {
    printf("problems during power on\n");
    exit(1);
}
```

The first parameter of each function is always the DSA_DRIVE or DSA_MASTER object which represents the device on which the operation will be done. The last parameter is always the 'time-out' that is to say the maximum time which is authorized to effectuate the operation. This time is always given in millisecond.

The call of this function ends when the operation is finished or when the time-out has elapsed. In the above example, the function ends when the motor is switched on and the control part (regulation) is activated, or when the process lasts more than 10 seconds. In that case, the function returns an error.

For the operations where the execution time does not depend on the position controller configuration or on the application, the default time-out can be used by giving the DSA_DEF_TIMEOUT (which will be explained later on).
Some other functions need more parameters. For example, it is the case for the function which starts a movement and also needs the target point. Here is an example on how a movement can start with a target point at 0.32 meter from the origin point:

```
err = dsa_set_target_position_s(drive1, 0, 0.32, DSA_DEF_TIMEOUT);
if (err != 0) {
    printf("cannot start movement\n");
    exit(1);
}
```

In this manual’s example, the movements are specified in relation to the software position limits set on the position controller (position controller parameters KL34 & KL35 for the AccurET family). So it is necessary to read them at the beginning. This is quite a common operation so there are specific functions for this:

```
err = dsa_get_min_soft_position_limit_s(axisX,&pos_min,
                     DSA_GET_CURRENT, DSA_DEF_TIMEOUT)
err = dsa_get_max_soft_position_limit_s(axisX, &pos_max,
                     DSA_GET_CURRENT, DSA_DEF_TIMEOUT)
```

A list of the available functions is given in the appendix (see §13.). If the user is used to program sequences in the ETEL language or if he knows the commands of the ETEL terminal, he will find the correspondence between these commands and the functions of the DSA library. EDI has access to the correct register or call the correct command).

Here are some examples of specific functions:

- `dsa_reset_error_s(drive1, DSA_DEF_TIMEOUT);`
- `dsa_power_on_s(drive1, 10000);`
- `dsa_homing_start_s(drive1, 10000);`
- `dsa_wait_movement_s(drive1, 60000);`
- `dsa_set_target_position(drive1, 0, 0.5, DSA_DEF_TIMEOUT);`

3.5 Error management

In general, each function of the EDI package returns an integer (int) which represents the error code. If the operation ends with success, the function returns 0, otherwise it returns the number of the error represented by a negative number.

The possible error code are defined by a series of #define stored in dsa40.h. For the DSA library the possible error codes are as follow:

#define	Error code	Comment
DSA_EACQDEVINUSE	-337	One of the device is already doing an acquisition
DSA_EACQNOTPOSSIBLE	-336	Drives must be connected with transnet
DSA_EBADDRIVER	-328	wrong version of the installed device driver
DSA_EBADDRVVER	-325	a drive with a bad version has been detected
DSA_EBADIPOLGRP	-327	the ipol group is not correctly defined
DSA_EBADLIBRARY	-333	function of external library not found
DSA_EBADPARAM	-322	one of the parameter is not valid
DSA_EBADSEQVERSION	-338	the sequence version is not correct
DSA_EBADSTATE	-324	this operation is not allowed in this state
DSA_EBUSERROR	-313	the underlaying etel-bus is not working fine
DSA_EBUSRESET	-314	the underlaying etel-bus in performing a reset operation
DSA_ECANCEL	-319	the transaction has been canceled
DSA_ECFGCOMPFILE	-339	File has been compiled for a different axes configuration
DSA_ECONVERT	-317	a parameter exceeded the permitted range
DSA_EDRVERROR	-311	drive in error
DSA_EDRVFAILED	-323	the drive does not operate properly
DSA_EEQUATION	-340	Equation cannot be resolved
DSA_EINTERNAL	-316	some internal error in the etel software
DSA_EMAPNOTACTIVATED	-335	Mapping cannot be activated by the device
DSA_ENOACK	-312	no acknowledge from the drive
DSA_ENODRIVE	-320	the specified drive does not respond
DSA_ENOFREESLOT	-330	no free slot available
DSA_ENOLIBRARY	-332	external library not found
DSA_ENOTIMPLEMENTED	-326	the specified operation is not implemented
DSA_EOBSOLETE	-329	function is obsolete
DSA_EOPENPORT	-321	the specified port cannot be open
DSA_ERTVREADSYNCRO	-331	RTV read synchronisation error
DSA_ESYNTAX	-334	Mapping file syntax error
DSA_ESYSTEM	-315	some system resource return an error
DSA_ETIMEOUT	-310	a timeout has occured
DSA_ETRANS	-318	a transaction error has occured

It is the developer’s responsibility to test this error code and react in consequence. Good programming practice considers it is important for each error code to be tested and reacted upon.The direct benefit for the user is that during development time, it helps him identify more precisely where an error has occurred.

The DSA library offers a means of error diagnostic with its `DSA_DIAG()` macro:

```
int err;
...
if (err=dsa_power_on_s(...)) {
    DSA_DIAG(err, drv);
    goto _error;
}
```

The DSA_DIAG macro prints on the standard output an error message, the status of the device at the time of the error and the call stack trace (with line numbers) of the part of the execution performed by the EDI package up until the error occurred. This output can be used by the developer to identify the source of a problem. It also gives in depth detailed information on the context and the whereabouts of the source of error which is a very important help to ETEL when supporting customer development. Developers are therefore strongly encouraged to use this call to be able to benefit from the topmost quality support from ETEL. **You will be formally required to give this information to ETEL’s customer support for any problem that is reported to them.**

A detailed description of the meaning of the information output by DSA_DIAG() in given in appendix §14.. The output of DSA_DIAG is the standard output. If the user application does not redirect its standard output or if the standard output is not visible, it is advised to use DSA_SDIAG or even DSA_FDIAG to store the error message. Another extended version of DSA_DIAG, functions are available. Their names are DSA_EXT_DIAG, DSA_EXT_SDIAG and DSA_EXT_FDIAG. They print a more user-readable form of the normal version, first of all concerning the device status.

The user can also test the smooth operation of a DSA function in a more classical way but less efficient support-wise:

```
  int err;
  ...
  err = dsa_power_on_s(...);
  if (err == 0)
    printf ("power on done.\n");
  else if (err == DSA_ETIMEOUT)
     printf("timeout error.\n");
 else
     printf("error %d during power on.\n", err);
...
```

The DSA library also offers a function which enables the conversion of an error code into a textual error message in the form of a character string. Here is the prototype:

```
const char *dsa_translate_error(int code);
```

Here is an example of a way to process the return code using this function:

```
int err ;
...
err = dsa_power_on_s(...) ;
if (err) {
    printf("ERROR %d : %s\n", err, dsa_translate_error(err));
    exit(err);
}
printf("power on done.\n");
...
```

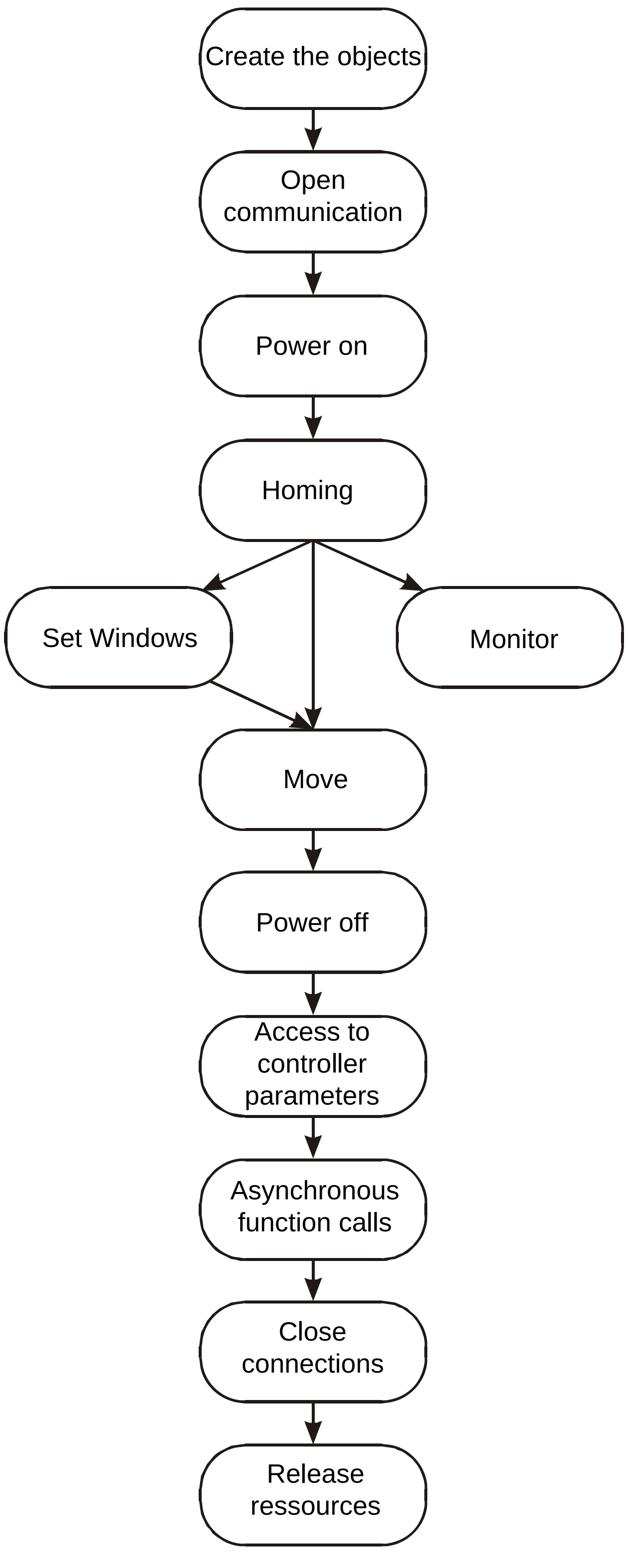
4. Application startup

4.1 Presentation of the general example

The example is given in C as it is a common general basis for programmers and other languages as well.

This example aims at using the library to control two axes concurrently via an UltimET.

EDI Application Example Synopsis



The system commissioning aspects are **outside** the scope of this example, mainly because they are beyond the scope of the DLL itself. Hence the example does **not** cover:

• downloading firmwares onto the UltimET and the position controllers (although there is an example showing how to do this)

• setting of the position controllers

• the commissioning can be done through ComET software

A first set of actions aims at getting the position controllers in an operational state which entails:

• establishing the communication with the UltimET and the position controllers connected to it (§4.5)

• powering up the position controllers (§4.6)

• performing the homing procedure on each position controller (§4.7).

The next set deals with making movements on each axis by first describing how to set precision windows (§5.2), and then actually specifying and starting the movement (§5.3 and §5.4). The movements will be controlled by the UltimET which will execute them in interpolated mode.

In a more elaborate step, one thread will be created to monitor the current position (§6.) of the motors. The monitoring thread will show the current position of each motor every 100ms. Another will loop indefinitely waiting for a user input: the space bar will immediately stop the movement and set one of the controller's digital outputs.

Additional features will be described in:
• §11. for 'Access to controller parameters'

• §12. for 'Asynchronous function calls'

Finally, it will be shown how to power off the position controllers (§4.6).

4.2 Prerequisites

The first step in using an ETEL controller is to tune its position and other regulation algorithms in accordance with the environment in which it is to operate in as described by its parameters that also have to be configured. This is done prior to programming any functional behavior via using ETEL’s commissioning tool ComET and is outside the scope of the EDI package. It is considered henceforth that these operations have been carried out successfully. They are a necessary prerequisite to starting to test a software application using the EDI package.

4.3 Application initialization steps

From a purely programming point of view, there are a couple of header files to be included at the beginning of our example’s source code:

- some standard libraries:

```
#include <stdio.h>         /* standard I/O library               */
#include <stdlib.h>        /* standard general purpose library   */
#include <ctype.h>         /* standard character conversion library*/
#include <math.h>
```

- the platform specific header for the thread management library:

```
#include <process.h>       /* standard multithreading library    */
```

- the EDI top level library header (version 4.x):

```
#include <dsa40.h>
#include <dmd40.h>
```

Also, most functions of the library return an error code or 0 if no error occurred. We need a variable to store the last error. The error codes are negative values, ranging from -399 to -300 for the dsa40 library:

```
int err=1;                 /* initialized to a value that is NOT */
                           /* returned by the library functions  */
```

4.4 Creating the objects

Create the objetcs

As an illustration of the creation of objects and for the needs of the example, we must create the objects representing the controllers and the UltimET.

We must define a pointer to a DSA_DRIVE object for each position controller. This hidden object is defined in the dsa20 library. The client does not need to access the members of this object directly, but need to pass it to various dsa library functions. It can be compared to the FILE structure defined in the standard I/O library. The pointer *must* be initialized to NULL before calling the «create drive» function, otherwise this function will fail.

```
DSA_DRIVE *axisX = NULL;
DSA_DRIVE *axisY = NULL;
```

The same goes for the object representing the UltimET:

```
DSA_MASTER *UltimET = NULL;
```

When the UltimET manages interpolated movements, the commands must be sent to what is called an interpolation group. This interpolation group can be used like a normal device group. All position controllers within this group can also be interpolated. So, we must also define a pointer to a DSA_IPOL_GROUP (like a DSA_DRIVE_GROUP) object. This hidden object is also defined in the dsa40 library, and can be considered as a sort of array containing other DSA_DRIVE objects. The client can set, change and retrieve the different devices belonging to this object through some accessor functions.

As with DSA_DRIVE, the group pointer must be initialized to NULL before calling the created group function, otherwise this function will fail.

```
DSA_IPOL_GROUP *igrp = NULL;
```

Once the pointers have been created, the actual objects themselves have to be created and initialized:

```
/* Create the drive and UltimET objects. */
if (err = dsa_create_drive(&axisX)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_create_drive(&axisY)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
if (err = dsa_create_master(&UltimET)) {
     DSA_DIAG(err, UltimET);
     goto _error;
}
```

Now, we have to create the interpolation group object. The size for the group must be given at the creation time and cannot be changed afterwards. In our case, we want to put two position controllers in this group.

```
if (err = dsa_create_ipol_group(&igrp, 2)) {
     DSA_DIAG(err, grp);
     goto _error;
}
```

After creating the group, it has to be filled with the required position controllers by calling call `dsa_add_group_item()` that needs only two arguments: the group and the position controller to put on in.

```
if (err = dsa_add_group_item(igrp, axisX)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_add_group_item(igrp, axisY)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
```

Associate the UltimET to the interpolation group. When you ask for an interpolated movement on a group of axes, the dsa40 library will use the UltimET associated with the group.

```
if (err = dsa_set_master(igrp, UltimET)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Now we are ready to open the communication channels.

4.5 Establishing the communication

Open communication

Once the device objects have been created, a physical device must be associated with it. The communication bus used to communicate with the position controller as well as the physical address of the position controller (axis number) must be indicated. This is usually done by the `dsa_open_u` function, which takes the URL of the position controller as the parameter. At this time, only ETEL-Bus's URLs are recognized. The URL identifies the protocol used, the communication bus type and its characteristics and the corresponding physical axis used. The only protocol recognized by ETEL’s EDI package is the proprietary ETB (ETEL Bus) protocol. Hence all URLs used for referencing ETEL devices take the following form:

"**etb**:<bus type>:<axis>"

As a simple example, if we wanted our position controller `axisX` object to correspond to the axis 12 which is connected to the USB port of the PC, the communication would be established using the following call:

```
err = dsa_open_u(axisX, "etb:usb:12");
```

This function will first open the communication bus on the condition that it is not already opened and then will memorize the axis number in the `axisX` object. The second parameter is the character string representing the URL.

The general syntax for ETEL URLs is described hereunder:

URL = "**etb** : <communication bus> : <axis number>"

<axis number> is either

an integer, ranging from 0 to 62 for the AccurET family, that is the axis number set by the controller’s corresponding dip switches or by the `AXI` command (refer to the corresponding **’Operation & Software Manual’** for more information);

or «*» to designate the UltimET.

<communication bus> is a string describing the bus used and depends on the bus. The table describes the syntax of <communication bus>.

Bus type	Syntax	Description
UltimET and ETND (TCP/IP)	**ETN://**<ip-address>:<port> [,T=<keep alive>]	for local or remote access via the TCP/IP protocol to an UltimET TCP/IP or ETND ip-address: the IP address of the computer running ETND, (local host or 127.0.0.1 usually represent the local computer), or that of the UltimET. port : for the UltimET: 1129, 1128 or 1127 for the AccurET: 1129 for ETND: depends on the port.properties file of the remote PC running the ETND.
UltimET (PCI)	**ULTIMET**[,<reset flags>]	reset flags: (optional) r : reset ULTIMET x: reset position controllers
USB	**USB**	Opening communication on USB will open all ETEL devices connected through USB to the PC. EDI will act as a master. But be aware that the devices will not be synchronized as they are when using a multi-axis board.

Below are some examples of commonly used URLs:

etb:ETN://172.22.7.200:1149:12 => used to open a communication with controller 12 connected to a computer whose IP address is 172.22.7.200 and that is connected to the controllers network via port 1149. The communication between the remote computer and it is controller’s network will the protocol described for port 1149 in the *port.properties* on that computer.

etb:ULTIMET,r:* => used to reset and open communication with an PCI-UltimET itself.

etb:ULTIMET:23 =>used to open communication with an AccurET controller 23, using the TRANSNET connected to an UltimET in a PCI slot.

etb:USB:2 => used to open communication with an AccurET controller 2, using USB

etb:ETN://172.22.10:1129,T=-1: ∅ => used to open a communication with controller ∅ using TransnET connected to an UltimET TCP/IP whose IP address is 172.22.10.112. The port 1129 is used. T-1 allows the user to disable keep-alive handshake between PC and UltimET. This handshake has been implemented to check the link presence between the PC and the UltimET TCP/IP. Every 10 seconds, a message is sent from the PC to the UltimET. If the UltimET does not receive this message after 3 x 10 seconds, it falls into error 1603.

In our example, we have to write the following lines to establish the communication:

```
if (err = dsa_open_u(axisX, "etb:UltimET:0")) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_open_u(axisY, "etb:UltimET:1")) {
     DSA_DIAG(err, axisY);
     goto _error;
}
if (err = dsa_open_u(UltimET, "etb:UltimET:*")) {
     DSA_DIAG(err, UltimET);
     goto _error;
}
```

4.6 Powering on (and off)

Power on

Different devices take different times to completely startup which means that in a system that includes different types of devices, such as some position controllers and a position motion controller, some devices will be ready when others are not. This can have the effect of putting in error a device that expects another to be there but that hasn’t finished starting up. This is why, when all devices have finished powering up, it is recommended to start the application by a general «reset error» command:

```
if (err = dsa_reset_error_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Now we can send commands to the position controllers. The first thing to do is to put the position controller in power on state. This is done by the `dsa_power_on_s` command. It is the equivalent of the controller’s `PWR=1` command.

The "_s" at the end of the command indicates that this is a «synchronous» function. Synchronous function wait until the end of the operation before returning. All synchronous functions have a timeout parameter as the last parameter. This parameter orders the function to return with a timeout error (DSA_ETIMEOUT) if no response comes from the position controller before the end of the specified timeout. This lack of response usually indicates an error in the application, or could result from bad position controller parameters. An appropriate timeout value heavily depends on the application and the command issued. In the «power on» case, less than 1 second could be appropriate with pulse initialization, but more than 5 seconds could be required with constant current initialization.

Here we can do this in two ways: either switch the power on in each motor individually:

```
if (err = dsa_power_on_s(axisX, 10000)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_power_on_s(axisY, 10000)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
```

or, since we have created a group - primarily for interpolation purposes, but it can also be used any where a group is expected - send the command to the group in general, the DLL taking upon itself to dispatch it to all the members of the group:

```
if (err = dsa_power_on_s(igrp, 10000)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Powering off is just as simple:

```
if (err = dsa_power_off_s(igrp, 10000)) {
     DSA_DIAG(err, igrp);
```

```
      goto _error;
}
```

4.7 Homing

Homing

Next step in readying the controllers and motors for operation is performing the homing procedure in order to find the reference for the motor’s absolute position. Once again, this can be done on each axis individually:

```
if (err = dsa_homing_start_s(axisX, 10000)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_homing_start_s(axisY, 10000)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
```

or on the group as a whole:

```
if (err = dsa_homing_start_s(igrp, 10000)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
```

`dsa_homing_start_s()` is the equivalent of the controller’s `IND` command.

This function «only» starts the homing procedure. Before we can make other movements, we must wait until it is terminated. To this end, we wait until the movement is finished, on both axis:

```
if (err = dsa_wait_movement_s(igrp, 60000)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

5. Movements

5.1 Introduction to the example’s trajectory

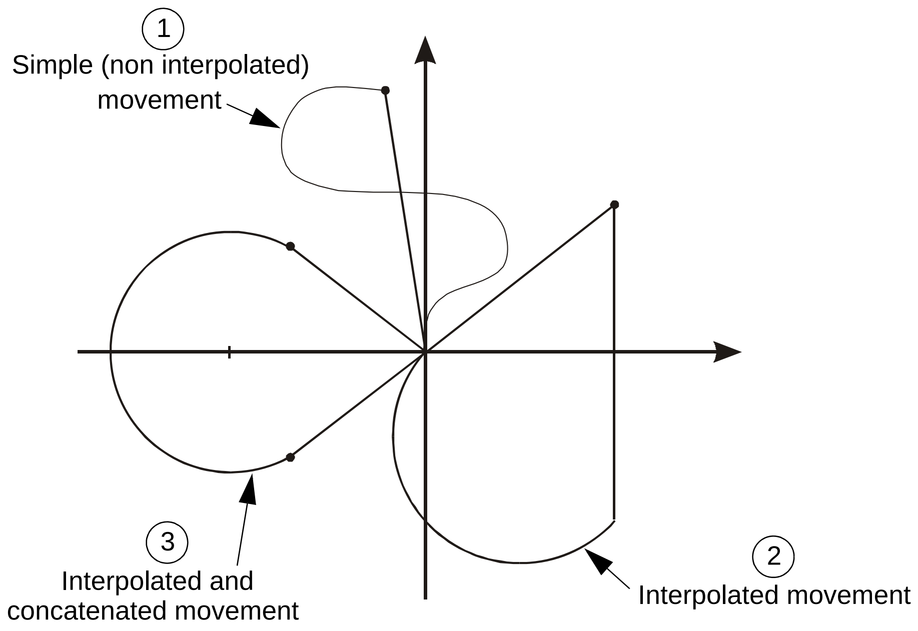
In our example, to make sure we never go beyond the movement limits and generate an error on the position controller, the coordinates of the positions are always given with reference to the minimum and maximum position limits. These are stored in the controller parameters KL34 & KL35 for the AccurET-family and are read as follows:

```
#define AXIS_NB_X 0
#define AXIS_NB_Y 1

double pos_min[2], pos_max[2];

if (err = dsa_get_min_soft_position_limit_s(axisX, &pos_min[AXIS_NB_X],
DSA_GET_CURRENT, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisX);
    goto _error;
}
if (err = dsa_get_max_soft_position_limit_s(axisX, &pos_max[AXIS_NB_X],
DSA_GET_CURRENT, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisX);
    goto _error;
}
```

We do the same for the other axis and store the available range of motion in a variable to be used later:

```
double range_of_motion[2];
range_of_motion[AXIS_NB_X] = pos_max[AXIS_NB_X] - pos_min[AXIS_NB_X];
range_of_motion[AXIS_NB_Y] = pos_max[AXIS_NB_Y] - pos_min[AXIS_NB_Y];
```

In our example the motors move along the trajectory depicted on the following figure:


1
Simple (non interpolated) movement
3
Interpolated and concatenated movement
2
Interpolated movement

5.2 Defining the position windows

Set Windows

To ensure that the movement stays within the application's requirements characteristics, there are several windows that have to be - or can be, depending on what type of requirements govern the application - defined. For example, a position driven application might want to constrain the tracking error, and/or the position settling time, and/or the precision of the target position.

The EDI package provides several special functions to meet these needs:

- `dsa_set_following_error_window_s` set the tracking error limit (position controller parameter K30)
- `dsa_set_position_window_time_s` defines the minimum time during which the position must be within given bounds for the target to be considered reached (position controller parameter K38).
- `dsa_set_position_window_s` sets the acceptable error on the real position (position controller parameter K39) compared to the target position.

Usually, these parameters are set at the position controller level when integrating the system, but some applications may need to change, for example, the tracking error detection limit depending on an operating mode. This would be done in the following way:

```
  long axisX_trackingErrorLimit =....;
  long axisY_trackingErrorLimit =....;
  if(err=dsa_set_following_error_window_s(axisX,axisX_trackingErrorLimit,
DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisX);
     goto _error
  }
  if(err=dsa_set_following_error_window_s(axisY,axisY_trackingErrorLimit,
DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisY);
     goto _error
  }
```

5.3 Simple (non interpolated) movements

Move

5.3.1 Defining the movement profile

As described in ETEL controller's manual (refer to the corresponding **'Operation & Software Manual'** for more information), the movement can be controlled in a variety of ways depending on its operating mode: force reference, speed reference, external reference, or position reference. The controller's mode (position controller parameter K61) is initialized at setting time but can be set using `dsa_set_drive_control_mode_a()`. Refer to the **'HTML Reference Manual'** for the details of this function.

When in position controlled mode, which is the default mode, the movement can be specified by means of a set of specific functions. In the other modes, the movement must be defined using the generic functions (refer to §10.2 for more information).

A position controlled movement can be defined by up to four parameters.One usually sets at least the target position, which we will see the next paragraph (§5.3.2); the speed, acceleration and jerk time describing the movement profile can also be modified.

To set the movement speed:

```
double profileSpeed = 0.1; /* m/s or t/s for rotary motors */
```

```
if (err = dsa_set_profile_velocity_s(axisX,0, profileSpeed, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
```

To set the movement acceleration:

```
double profileAcc = 10.0; /* m/s^2 or t/s^2 for rotary motors*/
if (err = dsa_set_profile_acceleration_s(axisX, 0, profileAcc,DSA_DEF_TIMEOUT))
{
     DSA_DIAG(err, axisX);
     goto _error;
}
```

To set the movement jerk time:

```
double jerkTime = 0.01; /* seconds */
if (err = dsa_set_jerk_time_s(axisX, 0, jerkTime, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
```

5.3.2 Starting

Making a single position controller simply go to a given position is quite straight forward. To move to a random position within the bounds of the motor stroke:

```
double targetPosition=pos_min[AXIS_NB_X]+(rand()*(range_of_motion[AXIS_NB_X])/
                                                                     RAND_MAX;
if (err = dsa_set_target_position_s(axisX, 0, targetPosition, DSA_DEF_TIMEOUT))
{
     DSA_DIAG(err, axisX);
     goto _error;
}
```

and then wait until it is finished:

```
if (err = dsa_wait_movement_s(axisX, 60000)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
```

The `dsa_set_target_position_s()` is the equivalent of the controller’s `POS` command. It sets the value of the target position and starts the movement.

A special note must be made for the second parameters («0» in the above example). The function takes a subindex parameter as it's second argument. If a zero is written in the subindex, like above, the movement will start immediately. A value of 1 to 3 in the subindex just prepares the movement, which will then start with the "new set point" command:

```
if (err = dsa_new_setpoint_s(axisX, preparedSubIndex, flags ,DSA_DEF_TIMEOUT))
{
     DSA_DIAG(err, axisX);
     goto _error;
}
```

Note that this function takes a group as first parameter. Indeed preparing movements at different subindices and using the `dsa_new_setpoint_s` function is meaningful when several position controllers need to start at the same time. With TransnET, as in our example, you can synchronize all position controllers together. In this case, the movements really start exactly at the same time (within one microsecond).

The third param is a mask and should contain the logical sum of all the following constants, specifying which information must be fetched from the buffer. The constants include (refer to the **’HTML Reference Manual’** for complete range of values):

- DSA_STA_POS: use the target position of the given buffer
- DSA_STA_SPD: use the profile velocity of the given buffer
- DSA_STA_ACC: use the profile acceleration of the given buffer
- DSA_STA_JRK: use the profile jerk of the given buffer

For example, if subindex 2 was used to store the new target position and movement profile speed, then, to start the movement we would write:

```
if (err = dsa_new_setpoint_s(igrp, 2, DSA_STA_POS | DSA_STA_SPD,
,DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

5.4 Interpolated movements

Move

In our example we have two axis and a UltimET, so we can also performed interpolated movements.

Interpolated movements are defined by high level commands that are sent to the UltimET which interprets them to generate the set points for each individual position controller making up the interpolation group to achieve the specified trajectory.

To start making interpolated movements you have to enter an «interpolated» mode, which is what `dsa_ipol_begin_s()` does. Once in this mode, you have to use the interpolation functions on the position controllers belonging to the interpolation group. They begin with `dsa_ipol_...()`. You can no longer set a target position using `dsa_set_target_position_s()` on a given axis. To do so, you have to explicitly have to leave the interpolation mode with `dsa_ipol_end_s()`.

Interpolated mode functions allow you to specify geometric definitions of the trajectory, such as drawing a line, a circle and so on. Refer to the **’HTML Reference Manual’** for the full list of available functions. The current position when entering the interpolation mode is used as reference point for all subsequent movement definitions. When leaving the interpolation mode, the references revert to what they were before the interpolated movements.

So, to enter interpolation mode, use:

```
if (err = dsa_ipol_begin_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Important: Entering interpolation mode changes the reference point of positions. Whereas before, all positions were relative to the encoder reference mark, once in interpolation mode, the reference point used is the position where the system was when sending the `dsa_ipol_begin_s`() command. So all coordinates given in interpolated mode are relative to the position at the start of the interpolation mode.

5.4.1 Defining the movement profile

Then, in this mode, it is the maximum tangential speed (in m/s) and accelerations (in m/s^2) that must be defined.

To set the tangential speed of the trajectory:

```
if (err = dsa_ipol_tan_velocity_s(igrp, 0.05, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

To set the tangential acceleration of the trajectory:

```
if (err = dsa_ipol_tan_acceleration_s(igrp, 0.1, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

To set the tangential deceleration of the trajectory:

```
if (err = dsa_ipol_tan_deceleration_s(igrp, 0.1, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Now we are ready to make our system move.

5.4.2 Making movements

At this point we can, for example, start the trajectory with a linear segment. The following function defines and starts the movement from the previous end of segment and stops the system at the end of the specified segment.

To specifies the coordinates of the end point of the segment:

```
double endPointX = range_of_motion[AXIS_NB_X]/4.0;
double endPointY = range_of_motion[AXIS_NB_Y]/4.0;
```

And to actually go from the end position of the previous segment to the specified end position:

```
if (err=dsa_ipol_line_2d_s(igrp,endPointX, endPointY, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

After this we can keep on specifying other lines by another call to the same function:

```
endPointX = range_of_motion[AXIS_NB_X]/4.0;
endPointY = -range_of_motion[AXIS_NB_Y]/4.0;
if (err=dsa_ipol_line_2d_s(igrp,endPointX, endPointY, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

or draw a portion of a circle with calls like:

to define the center point of the arc:

```
double centerX = range_of_motion[AXIS_NB_X]/8.0;  /* X of center of arc */
double centerY = -range_of_motion[AXIS_NB_X]/8.0; /* Y of center of arc */
```

to define end point of the arc:

```
double arcendX = 0.0; /* X coordinate of end point of arc (return to center)*/
double arcendY = 0.0; /* Y coordinate of end point of arc (return to center)*/
```

and to execute the movement:

```
if (err = dsa_ipol_circle_cw_c2d_s(igrp, arcendX, arcendY, centerX, centerY,
DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

which will draw an arc ending at the position given by the 2nd and 3rd parameters, centered on the point with the coordinates given by the 4th and 5th parameters (with respect to the point at which the system was when entering the interpolation mode, remember), and moving in the clockwise direction.

5.5 Interpolated and concatenated movements

Up until now the system will have stopped at the end of each segment. To make a continuous movement, we have to indicate that we want to start concatenation of the segments:

```
if (err = dsa_ipol_begin_concatenation_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

But you only give this command AFTER the first segment has started, to tell the trajectory generator to look ahead for the next segment.

So now, in this mode all segments are processed at constant speed, without stopping between segments. This means that when the interpolator processes the one segment, it does not decrease the speed at the end of the segment. It jumps to the next segment without speed change. Now, the following calls will generate a movement along a trajectory at a constant speed:

```
double endPointX = -range_of_motion[AXIS_NB_X]/8.0;
double endPointY = range_of_motion[AXIS_NB_Y]/8.0;

if (err = dsa_ipol_line_2d_s(igrp,
                          endPointX,   /* Segment end X */
                          endPointY,   /* Segment end Y */
                          DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
if (err = dsa_ipol_begin_concatenation_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
if (err = dsa_ipol_circle_ccw_c2d_s(igrp,
                           endPointX,                         /* Arc end X */
                           -endPointY,                        /* Arc end Y */
                           -range_of_motion[AXIS_NB_X]/4.0, /* Arc center X */
                           0.0,                               /* Arc center Y */
                           DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
if (err = dsa_ipol_line_2d_s(igrp,
                          0.0, 0.0, /* return to interpolation mode origin */
                          DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

Of course, in this mode, you have to be careful not to introduce angles in the trajectory to avoid damaging the mechanical system.

Once the concatenated segments section is terminated, you can revert to giving individual segments again, in between which the UltimET will drive the system to a stop, and a restart:

```
if (err = dsa_ipol_end_concatenation_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

When the system no longer needs to perform interpolated movements, the interpolation mode is exited by calling:

```
if (err = dsa_ipol_wait_movement_s(igrp, 60000)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

to wait for all segment to be finished, and then:

```
if (err = dsa_ipol_end_s(igrp, DSA_DEF_TIMEOUT)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

to leave the interpolation mode and come back to the previous reference system.

6. Monitoring data

Move

In this section, it will be shown how to read data in a cycle from ETEL controllers. The example will monitor the motor's position and the position controller's status, quite common monitoring requirements.

To get feedback data on a regular basis, one usually creates a specific thread to this effect. Creating a thread is system and language dependent but usually involves writing an operation that performs the tasks; it is making this operation a thread that depends on the environment. On windows and in C, we need to write:

```
if(_beginthread(display_thread, 0, igrp) <= 0) {
     err = DSA_ESYSTEM;
     printf("ERROR  in  file  %s,  at  line  %s  %s\n",  __FILE__,  __LINE__,
dsa_translate_edi_error(err));
      goto _error;
}
```

Notice here a slightly different way to handle the error code. This is because `_beginthread` is a C library call that does not return known EDI error codes, so if there is a system error creating the thread, it is translated into an EDI system error and processed for display.

`_beginthread()` is an operation made available by `<process.h>`. The tasks the thread will do are implemented by the function passed as first parameter, i.e. `display_thread()`. This operation will be called by the thread with the third parameter of `_beginthread` as its own parameter; here a pointer to our group of position controllers.

Let's now see what `display_thread()` does.

First, from a purely code point of view, we must renew access to our position controllers, which is done with the help of the drive group passed as parameter to the function:

```
DSA_DRIVE_GROUP *grp = (DSA_DRIVE_GROUP *)param; /* get group from paramter*/
DSA_DRIVE *drv[2];                                /* create local drive objects. */
for(i = 0; i < 2; i++) {
     if (err = dsa_get_group_item(grp, i, &drv[i])) {
        DSA_DIAG(err, drv[i]);
        goto _error;
     }
}
```

The task is an infinite loop that will be stopped when the system terminates the whole process after it has come to its end. Every 100ms, the loop will read the position controller status and actual position to display them. Before the position controller status can be read, as it is a DSA object, it must be initialized using the appropriate function. The position controller status is kept up-to-date in the process memory by the DSA library because it is very frequently needed. Therefore, reading it is a very efficient operation that does not degrade the overall performance of the user application.

```
for(;;) {
     double pos[2];

     DSA_STATUS status[2]= {{sizeof(DSA_STATUS)},{sizeof(DSA_STATUS)}};
     status[0].size =sizeof(DSA_STATUS);/*Initialization;see §7.2 for details*/
     status[1].size =sizeof(DSA_STATUS);
     for(i = 0; i < 2; i++) {
        int err;
        if (err = dsa_get_status(drv[i], &status[i])) {
           DSA_DIAG(err, drv[i]);
              goto _error;
           }
        }
     for(i = 0; i < 2; i++) {
         int err;
      if(err = dsa_get_position_actual_value_s (drv[i], &pos[i],
DSA_GET_CURRENT,DSA_DEF_TIMEOUT)){
            DSA_DIAG(err, drv[i]);
            goto _error;
         }
     }
     /*
      * We can now print the status string on the display.
      */
     printf("%04d: ", ++counter);
     for(i = 0; i < 2; i++) {
        printf("AXIS %d: %c%c%c/%4.4fmm", i,
           status[i].drive.moving ? 'M' : '-',
           status[i].drive.warning ? 'W' : '-',
           status[i].drive.error ? 'E' : '-',
           pos[i] * 1.0E3
           );
        printf((i == 0) ? ", " : "\r");
     }

     /* wait 100ms: system dependent call */
     { extern __stdcall Sleep(int); Sleep(100); }

}/*end endless loop*/
```

7. Status handling

7.1 Principle

Each ETEL's device has a set of information which allows the user to know what the state of the device is. This information is represented by a bit field. Each bit corresponds to a state (power on, error, warning, movement,...). The value of the bit shows if the device is or is not in this state.

Some of the states found on most ETEL devices, include:

Present	Shows if the device is present (active)
Warning	Shows if a warning appeared on the device
Error	Shows if the device is in error
Moving	Shows if the movement is in progress
In_window	Shows if the motor is in position, in the given window
Sequence	Shows if the sequence is running
Trace	Shows if the acquisition of a trace is in progress
User 0	At the user’s disposal
...	
User 16	

The main special feature of the status is its automatic update in real time. Indeed, because the status information needs to be queried often in applications, the DSA library maintains an up-to-date real-time value for the status of each position controller connected to the opened bus. Therefore the status access functions do not need any communication time over the bus and are thus not time consuming and can be called frequently without the fear of degrading the overall performance of the application.

The user has functions allowing him to wait for a device to be in a given state. Thus, the user can be informed of a status change without polling these status all the time.

Here is the list of these functions:

dsa_get_status()	Returns the status of a device
dsa_wait_status_equal_s()	Waits for a device to be in a given state
dsa_wait_status_not_equal_s()	Waits for a device to quit a given state
dsa_grp_wait_and_status_equal_s()	Waits for all the devices of a group to be in a given state
dsa_grp_wait_and_status_not_equal_s()	Waits for all the devices of a group to leave a given state
dsa_grp_wait_or_status_equal_s()	Waits for one of the devices of a group to be in a given state
dsa_grp_wait_or_status_not_equal_s()	Waits for one of devices of a group to leave a given state
dsa_cancel_status_wait()	Cancels the current waiting of the status of a device or a group of devices

7.2 Working with DSA_STATUS

The DSA library stores the status in a structure called 'DSA_STATUS'. Here is the definition:

```
typedef union DSA_STATUS {
  int size;                         /**< The size of this structure */
    struct DsaStatusSWMode {
      size_t size;                  /**< The size of the structure */
      DSA_SW1 sw1;                  /**< Drive status SW1(M60) */
      DSA_SW2 sw2;                  /**< Drive status SW2(M61) */
    } sw;                           /**< Status for M60/M61 access */
    struct DsaStatusRawMode {
      size_t size;                  /**< The size of the structure */
      dword sw1;                    /**< Drive status SW1 in dword type */
      dword sw2;                    /**< Drive status SW2 in dword type */
    } raw;                          /**< Status for raw access */
    struct DsaStatusDriveBitMode {
      size_t size;                  /**< The size of the structure */
      unsigned power_on:1;          /**< The drive is in power on */
      unsigned :2;
      unsigned present:1;           /**< The device is present */
      unsigned moving:1;            /**< The motor is moving */
      unsigned in_window:1;         /**< The motor’s position is in window */
      unsigned :2;
      unsigned sequence:1;          /**< A sequence is running */
      unsigned in_speed_window:1;
      unsigned error:1;             /**< Fatal error */
      unsigned trace:1;             /**< The acquisition of the trace is not
                                    finished */
      unsigned :3;
      unsigned sequence_thread:1;   /**< The specified thread of sequence are
                                    executing */
      unsigned :7;
      unsigned warning:1;           /**< Global warning */
      unsigned :8;
      unsigned :1;
      unsigned :1;
      unsigned save_pos:1;          /**< Position has been reached */
      unsigned :1;
      unsigned breakpoint:1;        /**< Breakpoint is reached */
      unsigned :3;
      unsigned user:16;             /**< User status */
      unsigned :8;
    } drive;                        /**< Status for drive bit access */
  struct DsaStatusDsmaxBitMode {
    size_t size;                    /**< The size of the structure */
      unsigned :3;
        unsigned present:1;         /**< The Master is present */
        unsigned moving:1;          /**<  One  of  the  interpolation  group  is
                                    moving */
        unsigned :3;
        unsigned sequence:1;        /**< A sequence is running */
        unsigned :1;
        unsigned error:1;           /**< Fatal error */
        unsigned trace:1;           /**<  The  acquisition  of  the  trace  is  not
                                    finished */
        unsigned ipol0_moving:1;    /**< Ipol group 0 is moving */
        unsigned ipol1_moving:1;    /**< Ipol group 1 is moving */
        unsigned :1;
        unsigned :1;
```

```
          unsigned :7;
          unsigned warning:1;          /**< Global warning */
          unsigned :8;
          unsigned :4;
          unsigned :4;
          unsigned user:16;            /**< User status */
          unsigned :8;
       } dsmax;                        /**<  Status  for  dsmax  bit  access  For
                                       compatibility only */
   struct DsaStatusUltimETBitMode {
       size_t size;                    /**< The size of the structure */
       unsigned :3;
          unsigned present:1;          /**< The Master is present */
          unsigned moving:1;           /**<  One  of  the  interpolation  group  is
                                       moving */
          unsigned :3;
          unsigned sequence:1;         /**< A sequence is running */
          unsigned :1;
          unsigned error:1;            /**< Fatal error */
          unsigned trace:1;            /**< The acquisition of the trace is not
                                       finished */
          unsigned ipol0_moving:1;     /**< Ipol group 0 is moving */
          unsigned ipol1_moving:1;     /**< Ipol group 1 is moving */
          unsigned :1;
          unsigned sequence_thread:1;  /**< The specified thread of sequence are
                                       executing */
          unsigned :7;
          unsigned warning:1;          /**< Global warning */
          unsigned :8;
          unsigned :1;
          unsigned :1;
          unsigned transnet_broken:1;  /**< Transnet is broken */
          unsigned :1;
          unsigned breakpoint:1;       /**< Breakpoint is reached */
          unsigned :3;
          unsigned user:16;            /**< User status */
          unsigned :8;
       } ultimet;                      /**< Status for UltimET bit access */
  struct DsaStatusUltimETBitMode {
       size_t size;                    /**< The size of the structure */
       unsigned :3;
          unsigned present:1;          /**< The Master is present */
          unsigned moving:1;           /**<  One  of  the  interpolation  group  is
                                       moving */
          unsigned :3;
          unsigned sequence:1;         /**< A sequence is running */
          unsigned :1;
          unsigned error:1;            /**< Fatal error */
          unsigned trace:1;            /**< The acquisition of the trace is not
                                       finished */
          unsigned ipol0_moving:1;     /**< Ipol group 0 is moving */
          unsigned ipol1_moving:1;     /**< Ipol group 1 is moving */
          unsigned :1;
          unsigned sequence_thread:1;  /**< The specified thread of sequence are
                                       executing */
          unsigned :7;
          unsigned warning:1;          /**< Global warning */
          unsigned :8;
          unsigned :1;
          unsigned :1;
          unsigned transnet_broken:1;  /**< Transnet is broken */
```

```
        unsigned :1;
        unsigned breakpoint:1;           /**< Breakpoint is reached */
        unsigned :3;
        unsigned user:16;                /**< User status */
        unsigned :8;
    } master;                            /**< Status for Master bit access */
} DSA_STATUS;
```

This structure has a size of 12 bytes. The first 4 bytes represent the size of the structure. The other 8 bytes represent the bit of the status. Thus, there are 64 bits available for the status.

DSA_STATUS is not a simple structure but an association of several structures. Each one of these structures is particular to a type of product. Thus, they allow the user to have access to the same 64 bits in a way specific to a type of products.

The ’dsmax’ or ’ultimet’ structure defines the 64 bits of the multi-axis motion controller. The '`drive`' structure defines the same 64 bits for a position controller. The 'raw' structure allows the user to have access to the 64 bits of the status by two words of 32 bits. The 'sw' structure allows the user to have access to the 64 bits of a status issued by reading M60 and M61. On the AccurET family, it is still possible to have access to M60 and M61 by calling the function dsa_get_status_from_drive.
There are three ways of initializing the DSA_STATUS structure:

```
  1) DSA_STATUS st = {sizeof(DSA_STATUS)};

  2) DSA_STATUS st;
     memset(st,’\0’,sizeof(DSA_STATUS)};
     st.size = sizeof(DSA_STATUS);
```

Here is a reading example of the status:

```
int err;
DSA_STATUS status= {sizeof(DSA_STATUS)};/* very important ! */

...
status.size = sizeof(DSA_STATUS);

err = dsa_get_status(drive1, &status);
if (err != 0) {
  printf("I cannot get the drive status : %s\n",
     dsa_translate_error(err));
}

if (status.drive.moving)
  printf("The drive is moving\n");
else
  printf("The drive is stoped\n");
```

In this example, the user has to set the size field of the structure. Given that the user asks for the status of a position controller, he has access to the bits of the status via the 'drive' structure.

Here is an example where the user waits for the end of a movement by using the waiting on the status:

```
DSA_STATUS mask = {sizeof(DSA_STATUS)};
DSA_STATUS ref = {sizeof(DSA_STATUS)};

...

/* start a movement */
dsa_set_target_position_s(drive1, 0, 1.0, DSA_DEF_TIMEOUT);

/* wait for the end of the movement
 * this code is equivalent to dsa_wait_movement_s(drive1, 20000) */
```

```
            mask.drive.moving = 1;
            dsa_wait_status_equal_s(drive1, &mask, &ref, NULL, 20000);
```

In this example, the user waits for the movement to be finished that is to say that the 'moving' bit is equal to 0. It must be specified that the user waits for the 'moving' bit to be equal to 0. This is for this reason that there are a mask ('mask' variable) and a reference ('ref' variable). In the mask, the bits on which the waiting is done are set to 1. In the example, it is the 'moving' bit. The values that the user is waiting for are set in the reference. In the example, the value is equal to 0.

In the mask, all the bits except 'moving' must be equal to 0. It is done thanks to the 'memset()' function. The 'ref' variable is also set to 0 by this function even if it is not necessary. Only the value of the 'moving' bit is interesting in this variable. The size fields of both structures must be absolutely initialized correctly.

It is better to initialize the mask and ref variables at the creation. The simplified code is:

```
DSA_STATUS mask = {sizeof(DSA_STATUS)};
DSA_STATUS ref = {sizeof(DSA_STATUS)};

...

/* start a movement */
dsa_set_target_position_s(drive1, 0, 1.0, DSA_DEF_TIMEOUT);

/* wait for the end of the movement */
mask.drive.moving = 1;
dsa_wait_status_equal_s(drive1, &mask, &ref, NULL, 20000);
```

In the following example, the user monitors the position controller to know if it is in error mode or not. To do so, an asynchronous waiting is used on the error bit of the status:

```
void DSA_CALLBACK err_handler(DSA_DEVICE *dev, int err,
    void *param, const DSA_STATUS *status)
{
    printf("ERROR ON THE DRIVE !\n");
}

int main()
{
    DSA_STATUS mask = {sizeof(DSA_STATUS)};
    DSA_STATUS ref = {sizeof(DSA_STATUS)};

    DSA_DRIVE *drive1 = NULL;

    /* create and open the drive */
    dsa_create(&drive1);
    dsa_open_u(drive1,...);

    /* wait for an error – the program do not block here */
    mask.drive.error = 1;
    ref.drive.error = 1;
    dsa_wait_status_equal_a(drive1, mask, ref, err_handler, NULL);

    /* execute movements */
    dsa_set_target_position_s(drive1,...);
    ...

    /* close and destroy */
    dsa_close(drive1);
    dsa_destroy(&drive1);
}
```

7.3 Performances

The status are the fastest way for a device to give Boolean type information to the PC. These information can be a change of state, an event taking place, or any other information. The user has a part of the status to carry the information peculiar to his application.

The routing of these status is then carefully done. The processing at the PC level is realized by interrupt. A task of the PC can be then awakened by a status change. Thus, all the waits on the status do not use the time of the CPU.

Each device has a large number of Boolean information which are often stored in the M60, M61 and M63 monitorings registers. The status do not have all these information but only the ones which need a fast routing to the PC. For example, there is a bit (number 2) of M60 which indicates if the position controller has done the homing or not. This information is really useful but it does not need a routing in real time and then it is not part of the status. If the user wants to know this information, he can simply ask the position controller the value of M60 thanks to the dsa_get_register_s() function.

The status are updated every TransnET cycle time. Hence, the multi-axis motion controller receives an update of the status at the same rate (but because some position controllers have a slower cycle time, the multi-axis motion controller will receive during several TransnET cycles the same status value for these position controllers). Each time it detects a change of status, it informs the PC. Thus, the PC is informed of a status change at the same rate as the multi-axis controller but is often limited by the performances of the Windows operating system.

If the performances of the PC do not allow it to monitor the evolution of the status, the temporary changes of these status can be invisible on the PC. For example, if a very short movement is done, the 'moving' bit of the status will be equal to 1 during a very short time. This change could be not noticed by the PC and this bit would be considered as always equal to 1.

On the other hand, the PC will be always informed of the last state available. If the user wants to wait for the end of the movement, he has to wait for the 'moving' bit to be equal to 0. It is true whatever the performances of the PC. The waiting functions on the status do not wait for a change of state (edge) but simply wait for a given state.

7.4 User status

Among the bits of the status, 16 of them are at the user’s disposal. He can use them as he wishes to carry the information specific to his application. At the PC level, these bits are managed exactly like the others. Thus, the user can read them and wait for them to be in a particular state thanks to the functions described above. At the device level, several other features such as RTIs, interpolation marks, the K177 user status parameter modify the state of these bits. See the User's Manual of the related device to know how these interact.

8. Terminating the application

Close connections

Release ressources

Application termination operations take place in the reverse order of those described in §4. which are creating the objects and opening the communication.

These two functions assign resources to the operating system. During the creation of an object or a group, memory is assigned. During the opening of the communication, the use of the USB port, the interruption line or other resources specific to the communication bus used are assigned. These resources must be given back to the system as soon as there are not used any more. It is normally done just before leaving the application.

The DSA library has the `dsa_close()` function to close the communication and the `dsa_destroy()` function to destroy an object or a group. The closing must be done before the destruction.

Firstly, the connections must be closed:

- connection to the UltimET:

```
if (err = dsa_close(UltimET)) {
     DSA_DIAG(err, UltimET);
     goto _error;
}
```

- connections to both position controllers:

```
if (err = dsa_close(axisX)) {
     DSA_DIAG(err, drv);
     goto _error;
}
if (err = dsa_close(axisY)) {
     DSA_DIAG(err, drv);
     goto _error;
}
```

Then, just like the objects were created at the beginning of the application, the memory allocated to them must be released to be made available to the system again:

- group object:

```
if (err = dsa_destroy(&igrp)) {
     DSA_DIAG(err, igrp);
     goto _error;
}
```

- UltimET object:

```
if (err = dsa_destroy(&UltimET)) {
     DSA_DIAG(err, UltimET);
     goto _error;
}
```

- and the drive objects

```
if (err = dsa_destroy(&axisX)) {
     DSA_DIAG(err, axisX);
     goto _error;
}
if (err = dsa_destroy(&axisY)) {
     DSA_DIAG(err, axisY);
     goto _error;
}
```

9. Recommended proper reaction to error detection

Code is more maintainable if all error processing is done in the same code location. This is why, in the above examples, function call error handling has always referred to a `goto` label, where all error processing is performed.

Of course, one drawback to a single common error handler is that it needs to identify where the error originated and what state the program is in. This might make more code to write, but once validated, it usually ends up making the software easier to update.

The group object of drives, if it is a valid pointer, can first be destroyed; it is no longer of any use. The group, drive, UltimET pointers can be invalid if the `dsa_create_...` functions have failed (or not been called!) or that some memory has been corrupted.

```
if(dsa_is_valid_drive_group(igrp))
        dsa_destroy(&igrp);
```

The same has to be done with the multi-axis motion controller and the position controllers. For each position controller, if the object pointer is valid, we check if we can still communicate with it to stop all movements and then to power it off and finally close the communication. Afterwards, the object itself can be destroyed:

```
if(dsa_is_valid_drive(axisX)) {

     /* Is the communication open? */
     bool open = 0;
     dsa_is_open(axisX, &open);
     if(open) {

          /* Stop movements. */
        dsa_quick_stop_s(axisX, DSA_QS_PROGRAMMED_DEC,
                    DSA_QS_BYPASS | DSA_QS_STOP_SEQUENCE, DSA_DEF_TIMEOUT);

        /* When the motor has stopped, a power off is done. */
        dsa_wait_movement_s(axisX, 60000);
        dsa_power_off_s(axisX, 60000);

        /* Close the connection. */
        dsa_close(axisX);
    }

    /* Finally, release the associated resources to the OS. */
    dsa_destroy(axisX);
}
```

The same must be done for the other axis. The UltimET also has to be shutdown:

```
if(dsa_is_valid_master(UltimET)) {

     /* Is the UltimET open ? */
     bool open = 0;
     dsa_is_open(UltimET, &open);
     if (open) {
                /* Close the connection */
        dsa_close(UltimET);
     }
     /* And finally, release all resources to the OS. */
     dsa_destroy(&UltimET);
}
```

Remark: The dsa_destroy function asks as parameter the address of the pointer on the object (`&axisX`, `&axisY` and `&UltimET`) and not the pointers themselves (`axisX`, `axisY` and `UltimET`). In this way, once the object is destroyed, its address is assigned to NULL.

10. Units and unit conversions

10.1 Specific functions

In that type of function, the parameters representing a physical quantity are always given in ISO units. For example, in `dsa_set_target_position()`, the linear position is given in meter. According to the motor type (rotary or linear) and the quantity (position, speed, time...), the value will be given with the following unit. By using specific function, EDI will automatically have access to the correct parameter depending on the family of the accessed device.

Quantity	Linear motor	Rotary motor
Position	m	Turns
Speed	m/s	Turns/s
Acceleration	m/s^2	Turns/s^2
Jerk time	s	
Time	s	
Current	A	
Temperature	°C	

10.2 Generic functions

As seen previously, the parameters of the functions representing a physical quantity are always given in ISO unit like meter, second, ampere, etc... On the other hand, the ETEL devices work with internal units often called «increments». Here is a table illustrating the differences between these two types of units:

	ISO units	Increments
Coding	Floating point 64 bits	32 or 64 bits integer or float
Reference	Normalized	Depending on the device's parameters
Names	meter [m], second [s], ampere [A],...	UPI, USI, DPI,...
Use	Application level	Device level

To set a register in increment using the generic functions, the user needs to know which register-type and register-index he wants to have access to. He must then use the corresponding function:

	DSA function
Set register containing 32-bit integer increments	dsa_set_register_int32_...
Set register containing 64-bit integer increments	dsa_set _register_int64_...
Set register containing 32-bit float increments	dsa_set_register_float32_...
Set register containing 64-bit float increments	dsa_set_register_float64_...

To set a register in iso using the generic functions, EDI will automatically detect the accessed register and convert iso double parameter into correct increment type.

	DSA function
Set register using ISO value	dsa_set_iso_register_...

When using the specific functions, the DSA library does automatically the necessary units conversions and send to the devices the quantities in increments. On the other hand, when the generic functions are used, the user can choose between the increments or the ISO units. If the ISO unit is chosen, the user must specify the name of corresponding unit in the device so that the DSA library does the appropriate conversion.

Here are some examples:

Example 1:

- The user wants to modify the K39 register of a position controller
- K39 represents the in-window position of a linear motor
- This quantity in increments (in the position controller) is given in UPI
- This quantity in ISO units is given in meters [m]

The allocation of this register in increments is done in the following way:

```
err = dsa_set_register_s(
    drive1,          /* grp: destination device */
    2,               /* typ: 2 = K register */
    39,              /* idx: register index */
    0,               /* sidx: register sub-index */
    1000,            /* value: register value in UPI */
    DSA_DEF_TIMEOUT  /* timeout: default timeout */
);
```

or

To set a register in increment using the generic functions, the user needs to know which register-type and register-index he wants to have access to. He must then use the corresponding function:

```
err = dsa_set_int32_register_s(
    drive1,            /* destination device*/
    DMD_TYP_PPK,       /* typ 2 = K register */
    39,                /* register index */
    0,                 /* register sub-index*/
    1000,              /* value: register value in UPI*/
    DSA_DEF_TIMEOUT    /* default timeout*/
);
```

The appointment of this register in ISO units is done in the following way:

```
err = dsa_set_iso_register_s(
    drive1,          /* grp: destination device */
    2,               /* typ: 2 = K register */
    39,              /* idx: register index */
    0,               /* sidx: register sub-index */
    0.0001,          /* value: register value in [m] */
    DMD_CONV_UPI,    /* conv: drive unit (UPI) */
    DSA_DEF_TIMEOUT  /* timeout: default timeout */
);
```

To set register specifying ISO value, the user must specify which conversion function number he wants to use. Here follows a list of available conversion function number:

Values (#define)	Device's unit	Description
DMD_CONV_AVI	AVI	analog voltage increment 8192 correspond to -10V -8192 correspond to +10V
DMD_CONV_AVI12BIT	AVI12BIT	analog voltage increment 2048 correspond to -10V -2048 correspond to +10V
DMD_CONV_AVI16BIT	AVI16BIT	analog voltage increment 32767 correspond to -10V -32768 correspond to +10V
DMD_CONV_AVI16BITINV	AVI16BITINV	analog voltage increment 32767 correspond to 10V -32768 correspond to -10V
DMD_CONV_BIT0	BIT0	20 = 1 correspond to 1.0
DMD_CONV_BIT10	BIT10	210 = 1024 correspond to 1.0
DMD_CONV_BIT11	BIT11	211 = 2048 correspond to 1.0
DMD_CONV_BIT11_ENCODER	BIT11_ENCODER	Analog encoder signal amplitude in volt (11 bit)
DMD_CONV_BIT11P2	BIT11P2	

Values (#define)	Device's unit	Description
DMD_CONV_BIT15	BIT15	215 = 32768 correspond to 1.0
DMD_CONV_BIT15_ENCODER	BIT15_ENCODER	Analog encoder signal amplitude in volt (15 bit)
DMD_CONV_BIT15P2	BIT15P2	
DMD_CONV_BIT24	BIT24	224 = 256*65536 correspond to 1.0
DMD_CONV_BIT31	BIT31	231 = 32768*65536 correspond to 1.0
DMD_CONV_BIT5	BIT5	25 = 32 correspond to 1.0
DMD_CONV_BIT8	BIT8	28 = 256 correspond to 1.0
DMD_CONV_BIT9	BIT9	29 = 512 correspond to 1.0
DMD_CONV_BOOL	BOOL	boolean value
DMD_CONV_C13	C13	current 13bit range
DMD_CONV_C14	C14	current 14bit range
DMD_CONV_C29	C29	current 29bit range
DMD_CONV_CLTI	CLTI	current loop time increment (cti)
DMD_CONV_CTI	CTI	current loop time increment (41us)
DMD_CONV_CTRL_CUR2	CTRL_CUR2	Controller i2t, dissipation value
DMD_CONV_CTRL_CUR2T	CTRL_CUR2T	Controller i2t, integration value
DMD_CONV_CUR	CUR	current
DMD_CONV_CUR2	CUR2	i2, dissipation value
DMD_CONV_CUR2T	CUR2T	i2t, integration value
DMD_CONV_CUR2T_V2	CUR2T_V2	i2t, integration value
DMD_CONV_DAI	DAI	drive acceleration increment
DMD_CONV_DPI	DPI	drive position increment
DMD_CONV_DPI2	DPI2	drive position increment for secondary encoder
DMD_CONV_DSI	DSI	drive speed increment
DMD_CONV_DWORD	DWORD	double word value without conversion
DMD_CONV_ENCOFF	ENCOFF	11bit with 2048 offset
DMD_CONV_EXP10	EXP10	ten power factor
DMD_CONV_FLOAT	FLOAT	float value
DMD_CONV_FREF_FCTRL	FREF_FCTRL	Force ref of force control (conv 1:1)
DMD_CONV_FTI	FTI	fast time increment (125us-166us)
DMD_CONV_HSTI	HSTI	half slow time increment
DMD_CONV_INT	INT	integer value without conversion
DMD_CONV_IP_ADDRESS	IP_ADDRESS	ip address type
DMD_CONV_K1	K1	pl prop gain, k(A/m) = k1 * Iref * dpi_factor / 229
DMD_CONV_K10	K10	1st order filter in s.
DMD_CONV_K1031	K1031	per cent unit, 3133 correspond to 1.0 or 100%
DMD_CONV_K14	K14	
DMD_CONV_K2	K2	pl speed feedback gain, k(A/(m/s)) = k2 * Iref * dsi_factor / 229
DMD_CONV_K20	K20	speed feedback, sec unit (m/(m/s)), F = k20 / 216-k50 * dsi_factor / dpi_factor
DMD_CONV_K20_DSB	K20_DSB	speed feedback, sec unit (m/(m/s)), F = k20 / 216-k50 * dsi_factor / dpi_factor
DMD_CONV_K21	K21	speed feedback, sec unit (m/(m/s2)), F = k20 / 224-k50 * dai_factor / dpi_factor
DMD_CONV_K21_DSB	K21_DSB	speed feedback, sec unit (m/(m/s2)), F = k20 / 224-k50 * dai_factor / dpi_factor
DMD_CONV_K23	K23	commutation phase advance period/(period*second/m)
DMD_CONV_K23_ACCURET	K23_ACCURET	commutation phase advance period/(Volt*second/m)
DMD_CONV_K239	K239	motor Kt factor in mN(m)/A, 1000 correspond to 1.0mN(m)/A
DMD_CONV_K4	K4	pl integrator gain, k(A/(m*s)) = k1 * Iref * dpi_factor / 229 / pl_time
DMD_CONV_K5	K5	anti-windup K[m/A]=K5*4096/(dpi_factor * Iref)
DMD_CONV_K75	K75	encoder multiple index distance, 1/1024 * encoder perion unit
DMD_CONV_K8	K8	
DMD_CONV_K80	K80	cl prop gain delta[1/A]
DMD_CONV_K80_VHP	K80_VHP	cl prop gain delta[V/A]

Values (#define)	Device's unit	Description
DMD_CONV_K81	K81	cl prop integrator delta[1/(A*s)]
DMD_CONV_K81_VHP	K81_VHP	cl prop gain delta[V/(A*s)]
DMD_CONV_K82	K82	filter time, T = [cti] * (2n-1)
DMD_CONV_K9	K9	1st order filter in pl
DMD_CONV_K94	K94	time in 2x current loop increment
DMD_CONV_K95	K95	current rate for k95
DMD_CONV_K96	K96	phase rate for k96
DMD_CONV_KF22	KF22	jerk feedforward
DMD_CONV_KF256	KF256	Kt/M for Init small movement 2
DMD_CONV_KFLOAT	KFLOAT	float value for K parameters
DMD_CONV_KIF_FCTRL	KIF_FCTRL	Integrator gain for the force loop
DMD_CONV_KPF_FCTRL	KPF_FCTRL	Proportional gain for the force loop
DMD_CONV_KT_MOTOR	KT_MOTOR	KT motor
DMD_CONV_LONG	LONG	long integer value without conversion
DMD_CONV_M16	M16	jerk value
DMD_CONV_M242	M242	quartz frequency in Hz
DMD_CONV_M29	M29	per cent unit, 100% correspond to value of M229
DMD_CONV_M82	M82	current limit in 10 mA unit, 100 correspond to 1.0A
DMD_CONV_MF89	MF89	Magnetic period for Init small movement 2
DMD_CONV_MLTI	MLTI	manager loop time increment (sti)
DMD_CONV_MSEC	MSEC	milliseconds
DMD_CONV_PER_100	PER_100	per cent unit, 100 correspond to 1.0
DMD_CONV_PER_1000	PER_1000	per thousand unit
DMD_CONV_PH11	PH11	211 = 2048 correspond to 360°
DMD_CONV_PH12	PH12	212 = 4096 correspond to 360°
DMD_CONV_PH28	PH28	228 = 65536*4096 correspond to 360°
DMD_CONV_PLTI	PLTI	position loop time increment (fti)
DMD_CONV_POLE_FREQ	POLE_FREQ	filter pole frequency in Herz
DMD_CONV_QZTIME	QZTIME	interrupt time in sec = inc / m242
DMD_CONV_SPEC2F	SPEC2F	filter time, T = [fti] * (2n-1)
DMD_CONV_STI	STI	slow time increment (500us-2ms)
DMD_CONV_STRING	STRING	packed string value
DMD_CONV_TEMP	TEMP	20 = 1 correspond to 1.0
DMD_CONV_TTI	TTI	Minimum time base TransnET (25us)
DMD_CONV_UAI	UAI	acceleration, user acceleration increment
DMD_CONV_UFAI	UFAI	user friendly acceleration increment
DMD_CONV_UFPI	UFPI	user friendly position increment
DMD_CONV_UFSI	UFSI	user friendly speed increment
DMD_CONV_UFTI	UFTI	user friendly time increment
DMD_CONV_UPI	UPI	user position increment
DMD_CONV_UPI2	UPI2	user position increment dual encodeur mode
DMD_CONV_USI	USI	user speed increment
DMD_CONV_VOLT	VOLT	20 = 1 correspond to 1.0
DMD_CONV_VOLT100	VOLT100	(20)/100 = 1 correspond to 1.0

To use the #define's values, the 'dmd40.h' file must be included at the beginning of the source file in the following way:

```
#include <dmd40.h>
```

The unit of a parameter is then represented by an integer assigned to the 'conv' parameter of the generic functions. This number represents the ID of the unit in question.

To know the unit of a register or the unit of a command's parameter, that is to say the value to give to the 'conv' parameter, three methods are available:

- Check in the 'Operation & Software manual' of the device in question, the 'units conversion' chapter. Thanks to the name of the unit given in this chapter and defines written above, the value of the 'conv' parameter can be found.
- Use the dsa_get_register() function. If the 'kind' parameter is assigned to DSA_GET_CONV_FACTOR, this function returns in the 'val' parameter the unit of the register. This method cannot be used to know the unit of a command's parameters.
- Use the DSA_REG_CONV(typ, idx, sidx) macro which is replaced by the unit of the register specified in the parameters. The DSA_CMD_CONV(typ, idx, par) macro allows the same thing for the parameters of the commands. The values given by these macros are different from the #define ones or those given by dsa_get_register(). Their use and the result are identical.

11. Access to the controller parameters

Access to controller parameters

11.1 Sending commands

Some commands don't have a corresponding specific function and it is necessary to use the generic functions to send them to the controller.

Each device is able to execute a large number of command which does all sorts of operations.

Each command has a number and 0, 1 or several parameters. These parameters can belong to one of the following categories:

- Parameters with unit: they represent physical quantities such as positions, speeds, times, etc... They can be given in increments or ISO units. In increments, they are coded in 32-bit or 64-bit integers or float. In ISO units, they are coded in double precision floating points.
- Parameters without units: they often represent non-physical quantities such as the number of the axis, the error number, the digital input/output state, etc... These quantities are coded in 32 bits integers.
- Special parameters: they represent position controller registers. This kind of parameters is not developed in this manual.

Here is an example of command that a position controller is able to execute:

Command	ETEL syntax	Command number	Parameter number	Parameters type
Power on	PWR	124	1	Without unit
Emergency stop	HLO	119	0	Without unit
Set axis number	AXI	109	2	Without unit
Waiting time	WTT	10	1	With unit (physical quantity)
Reset	RSD	88	1	Without unit

The DSA library has a set of functions which allows the user to send a command to a device. Among these functions, the user will choose one according to the number and the type of parameters of the command to send.

As commands can have several types of parameters, EDI defines only a subset of generic functions.

Here is the list of the available functions:

Function which sends the command	Parameters type of the command to send
dsa_execute_command_s()	No parameter
dsa_execute_command_d_s()	One 32-bit integer parameter
dsa_execute_command_i_s()	One ISO parameter
dsa_execute_command_dd_s()	Two 32-bit integer parameters
dsa_execute_command_id_s()	Two parameters: the first in ISO and the second in 32-bit integer
dsa_execute_command_di_s()	Two parameters: the first in 32-bit integer and the second in ISO
dsa_execute_command_ii_s()	Two parameters in ISO
dsa_execute_command_x_s()	Any number and type of parameters

If the user wants to send the command number 119 (HLO) to a position controller, he must use the `dsa_execute_command_s()` function as follows:

```
err = dsa_execute_command_s(
    axisX,              /* grp: destination device */
    119                 /* cmd: number of the command */
    FALSE,              /* fast: fast command */
    FALSE,              /* ereport: report drive errors */
    DSA_DEF_TIMEOUT     /* timeout: by default */
  );
```

In the same way, if the user wants to send the command number 109 (AXI) with the parameters 1203 and 3, he must proceed as follows:

```
  err = dsa_execute_command_dd_s(
    axisX,              /* grp: destination device */
    109,                /* cmd: number of the command */
    0,                  /* typ1: type of the first parameter */
    1203,               /* par1: first parameter */
    0,                  /* typ2: type of the second parameter */
    3,                  /* par2: second parameter */
    FALSE,              /* fast: fast command */
    FALSE,              /* ereport: report drive errors */
    DSA_DEF_TIMEOUT     /* timeout: by default */
  );
```

In the meaning of the different parameters, the first one is always the DSA_DRIVE or DSA_MASTER object which represents the device on which the operation has to be done. Among the last parameters, there is the 'timeout' with two other parameters. The remaining parameters depend on the number and the type of the command's parameters to send.

The following table shows all the parameters:

Parameter	Name	Description
First parameter	grp	DSA_DRIVE or DSA_MASTER object
Parameter dependent on the number and the type of the command's parameters to send	typ	Always equal to 0 which means that the parameter is an immediate value. During the use of commands with special parameters, this parameter can have other values.
	par	Value of the parameter. It can be a long or a double type depending on whether the value is given in increments or ISO units.
	conv	Type of conversion to do. This parameter is only present when the parameter is used with the value given in ISO unit. See §10.1.
Last parameters	fast	Equal to TRUE if the command is fast. The fast commands have priority on the others. Only a few commands can be a fast command. In the majority of cases, this parameter must be equal to FALSE.
	ereport	If this parameter is equal to TRUE and the device which the command is sent to is in error, the function returns an error. It is then possible to detect if a device is in error or not.
	timeout	Maximum time of execution.

The parameters of the dsa_execute_command_x_s() function are a little bit different. Actually, this function allows the user to send commands whose the number of parameters is variable and/or type are different. First, the user must create a DSA_COMMAND_PARAM type array whose the number of elements corresponds to the number of the command's parameters. Each element in that table must be then assigned with the type, the value, and the conversion type of the corresponding parameter. Once the array created and assigned, the dsa_execute_command_x_s() command must be called by giving it as a parameter, the pointer and the size of this array.

Here is an example where the 1025 command (*ILINE=0, 0.28, 0.12, 0, 0) is sent to the UltimET:

```
DSA_COMMAND_PARAM params[ ] = {
  {0,0,0},{0,0,0},{0,0,0},{0,0,0},{0,0,0}
};

params[1].conv = DMD_CONV_UFPI;
params[1].val.d = 0.28;
```

```
params[2].conv = DMD_CONV_UFPI;
params[2].val.d = 0.12;

dsa_execute_command_x_s(UltimET, 1025, params, 5,
   FALSE, FALSE, DSA_DEF_TIMEOUT);
```

Here is an example where command 61 (SMP command) (SMP.0=0, 12000L, 5000L, 0.05) is sent to a controller:

SMP.0 =	0,	12000L,	5000L,	0.05
	sub-index,	position in 64bits integer,	speed in 64-bit integer,	acc in ISO

DSA_COMMAND_PARAM param[4];
param[0].typ = DMD_TYP_IMMEDIATE_INT32;
param[0].conv = 0;
param[0].val.i = 0;

param[1].typ = DMD_TYP_IMMEDIATE_INT64;
param[1].conv = 0;
param[1].val.i64 = 12000;

param[2].typ = DMD_TYP_IMMEDIATE_INT64;
param[2].conv = 0;
param[2].val.i64 = 5000;

param[3].typ = DMD_TYP_IMMEDIATE_INT64;
param[3].conv = DMD_CONV_UAI;
param[1].val.d = 0.05;

dsa_execute_command_x(drive1, 61, params, 4, FALSE, FALSE, DSA_DEF_TIMEOUT);

11.2 Reading and writing of controller registers

Each device has several registers types. The 16 types usually called are:

Type	Name	Type number
32-bits integer Parameters	K0, K1, K2,…	DMD_TYP_PPK_INT32 (2) or DMD_TYP_PPK
32-bits float Parameters	KF0, KF1, KF2,…	DMD_TYP_PPK_FLOAT32 (34)
64-bits integer Parameters	KL0, KL1, KL2,…	DMD_TYP_PPK_INT64 (66)
64-bits float Parameters	KD0, KD1, KD2,…	DMD_TYP_PPK_FLOAT64 (98)
32-bits integer Monitoring	M0, M1, M2,…	DMD_TYP_MONITOR_INT32 (3) or DMD_TYP_MONITOR
32-bits float Monitoring	MF0, MF1, MF2,…	DMD_TYP_MONITOR_FLOAT32 (35)
64-bits integer Monitoring	ML0, ML1, ML2,…	DMD_TYP_MONITOR_INT64 (67)
64-bits float Monitoring	MD0, MD1, MD2,…	DMD_TYP_MONITOR_FLOAT64 (99)
32-bits integer Common	C0, C1, C2,…	DMD_TYP_COMMON_INT32 (13) or DMD_TYP_COMMON
32-bits float Common	CF0, CF1, CF2,…	DMD_TYP_COMMON_FLOAT32 (45)
64-bits integer Common	CL0, CL1, CL2,…	DMD_TYP_COMMON_INT64 (77)
64-bits float Common	CD0, CD1, CD2,…	DMD_TYP_COMMON_FLOAT64 (109)
32-bits integer User	X0, X1, X2,…	DMD_TYP_USER_INT32 (1) or DMD_TYP_USER
32-bits float User	XF0, XF1, XF2,…	DMD_TYP_USER_FLOAT32 (33)
64-bits integer User	XL0, XL1, XL2,…	DMD_TYP_USER_INT64 (65)
64-bits float User	XD0, XD1, XD2,…	DMD_TYP_USER_FLOAT64 (97)

The DSA library has a specific function to read and assign each register widely used. On top of that, it offers 12 generic functions which are able to assign and read any register:

Function	Description
dsa_get_register_s()	Allows the reading of a integer 32-bits increment value of a register of corresponding increment type
dsa_get_register_int32_s()	Allows the reading of a integer 32-bits increment value of a register of corresponding increment type
dsa_get_register_float32_s()	Allows the reading of a float 32-bits increment value of a register of corresponding increment type
dsa_get_register_int64_s()	Allows the reading of a integer 64-bits increment value of a register of corresponding increment type
dsa_get_register_float64_s()	Allows the reading of a float 64-bits increment value of a register of corresponding increment type
dsa_get_iso_register_s()	Allows the reading of an ISO unit value of any register
dsa_set_register_s()	Allows the setting of a integer 32-bits increment value of a register of corresponding increment type
dsa_set_register_int32_s()	Allows the setting of a integer 32-bits increment value of a register of corresponding increment type
dsa_set_register_float32_s()	Allows the setting of a float 32-bits increment value of a register of corresponding increment type
dsa_set_register_int64_s()	Allows the setting of a integer 64-bits increment value of a register of corresponding increment type
dsa_set_register_float64_s()	Allows the setting of a float 64-bits increment value of a register of corresponding increment type
dsa_set_iso_register_s()	Allows the setting of an ISO unit value of any register

The following example assigns the X2:0 user variable with the value read in the M64:0 monitoring register:

```
long val;
err = dsa_get_register_int32_s(
    axisX,            /* grp: destination device */
    3,                /* typ: 3 = M monitoring register */
    64,               /* idx: register index */
    0,                /* sidx: register subindex */
    &val,             /* value: register value */
    DSA_GET_CURRENT,  /* kind: actual device value */
    DSA_DEF_TIMEOUT   /* timeout: default timeout */
    );

err = dsa_set_register_int32_s(
    axisX,            /* grp: destination device */
    1,                /* typ: 1 = X user variable */
    2,                /* idx: register index */
    0,                /* sidx: register subindex */
    val,              /* value: register value */
    DSA_DEF_TIMEOUT   /* timeout: default timeout */
    );
```

The following example is similar to the previous one but with ISO quantities. It reads the speed stored in the KL211:0 parameter, multiplies it by 10 and stores it again in the KL211:0 parameter. The KL211:0 parameter is in USI unit.

```
double speed; /* in m/s */
err = dsa_get_iso_register_s(
    axisX,            /* grp: destination device */
    66,               /* typ: 66 = KL parameter */
    211,              /* idx: register index */
    0,                /* sidx: register subindex */
    &speed,           /* value: returned value */
    DMD_CONV_USI,     /* conv: drive unit (USI) */
    DSA_GET_CURRENT,  /* kind: actual device value */
    DSA_DEF_TIMEOUT   /* timeout: default timeout */
    );

err = dsa_set_iso_register_s(
    axisX,            /* grp: destination device */
    66,               /* typ: 66 = KL parameter */
    211,              /* idx: register index */
    0,                /* sidx: register subindex */
    speed * 10.0,     /* value: value to set */
    DMD_CONV_USI,     /* conv: drive unit (USI) */
    DSA_DEF_TIMEOUT   /* timeout: default timeout */
    );
```

11.3 Saving and resetting

The values of parameters that have been only modified are not stored in permanent storage on the devices which means that when they are reset the parameters revert back to the stored values. To effectively store modified values, you have to explicitly save them by calling `dsa_save_parameters_s()` which is the equivalent of the SAV command a the controller. Sending a command to a device will cause it to ’disappear’ on the ETEL device network and to reappear in error. So it is possible that the device itself does not have time to acknowledge the SAV command. So it is advised to not check the error code returned by this function.

The second parameter is one of the following constants:

- `DSA_PARAM_SAVE_ALL` save all informations in flash memory
- `DSA_PARAM_SAVE_SEQ_LKT` save sequence and user look-up tables in flash memory
- `DSA_PARAM_SAVE_X_PARAMS` save user (X) registers and parameters (K) in flash memory
- `DSA_PARAM_SAVE_K_C_E_X_PARAMS:` AccurET family: save K, KL, KF, KD, C, CL, CF, CD, EL, X, XL, XF, XD parameters in flash memory
- `DSA_PARAM_SAVE_K_PARAMS:` ONLY ON AccurET family. Save K, KL, KF, KD parameters in flash memory
- `DSA_PARAM_SAVE_C_PARAMS`: ONLY ON AccurET family. Save C, CL, CF, CD parameters in flash memory
- `DSA_PARAM_SAVE_X_PARAMS`: ONLY ON AccurET family. Save X, XL, XF, XD parameters in flash memory
- `DSA_PARAM_SAVE_L_PARAMS`: ONLY ON AccurET family. Save LD parameters in flash memory
- `DSA_PARAM_SAVE_SEQUENCES`: ONLY ON AccurET family. Save sequences in flash memory
- `DSA_PARAM_SAVE_K_E_PARAMS`: ONLY ON AccurET family. Save K, KL, KF, KD, EL parameters in flash memory
- `DSA_PARAMS_SAVE_P_PARAMS:` ONLY ON AccurET family. Save P parameters in the flash

The SAV command takes quite a while to execute hence the large timeout value for the last parameter.

Sending a SAV command to a device will cause it to «disappear» on the ETEL device network and to re-appear in error. So before you can issue any other commands, you must wait (quite some time, as the last parameter of `dsa_wait_status_equal_s()` shows) until the position controllers are present again:

```
{
    /* setup status masks */
        DSA_STATUS status_checkbits   = {sizeof(DSA_STATUS)}
        DSA_STATUS status_checkstates = {sizeof(DSA_STATUS)};
        status_checkbits.size    = sizeof(DSA_STATUS);
        status_checkstates.size = sizeof(DSA_STATUS);

    status_checkbits.drive.present   = 1;
    status_checkstates.drive.present = 1;

    /* wait that the controllers are present */
    if (err = dsa_wait_status_equal_s(axisX, &status_checkbits,
                          &status_checkstates, NULL, 20000))
    {
        DSA_DIAG(err, axisX);
        goto _error;;
        }
    }
```

For the new values to be effectively taken into account the position controller has to be reset. There is no specific function for the RSD (reset) command so you have to send it «manually»:

```
     dsa_execute_command_d_s(
              axisX,               /* grp: destination device */
              DMD_CMD_RESET_DRIVE,/* cmd: constant number of the RSD command (88) */
              DMD_TYP_IMMEDIATE,  /* typ: the parameter to the command is a value*/
              255,              /* par: the value always passed to the RSD command*/
              FALSE,               /* fast: 1 or true = fast command */
              FALSE,              /* ereport: 1 or false do not report drive errors */
              10000                /* timeout: 10 secs */
       )
```

The RSD command also takes quite a while to execute hence the large timeout value for the last parameter. Also, because the controller resets, it does not have the time to send the acknowledgement that it has received and executed this command. This is why the call to dsa_execute_command_d_s() for the RSD command nearly always returns an error indicating an absence of command acknowledge. This is the one case where it is advised NOT to test the returned error code and why the above call is a bit different from the previous ones.

You have to pay attention to the fact that if a multi-axis controller is in the group of devices that is reset, the communication and the connection to that device (and hence all controllers downstream) will be lost. In that case you must establish the communication again. The proper way to do this, is to first close the communication with all devices, and then re-open it. In our example it would require the following lines:

```
    /* Close communication.
     * You don’t need to destroy the objects; these will be used again when
     * reopening the communication
     */
    dsa_close(axisX);
    dsa_close(axisY);
    dsa_close(UltimET);

    /* Reopen communication*/
    if (err = dsa_open_u(axisX, "etb:UltimET:0")) {
         DSA_DIAG(err, axisX);
         goto _error;
    }
    if (err = dsa_open_u(axisY, "etb:UltimET:1")) {
         DSA_DIAG(err, axisY);
         goto _error;
    }
    if (err = dsa_open_u(UltimET, "etb:UltimET:*")) {
         DSA_DIAG(err, UltimET);
         goto _error;
    }
```

12. Asynchronous function calls

Asynchronous function calls

This section illustrates how to use asynchronous functions. Before we terminally close the communication and end our example, we will switch the power back on once again, but this time asynchronously, which means the program will not at this point wait for the power on to finish. This is done using the same function as at the beginning, but that ends with **_s** instead of **_a**. Also remember (see 3.4.2) that the asynchronous functions have the following form:

`operation_name_`**`a`**`(`*`device/group identifier, callback function, callback function user parameters`*`).`

So, in the present case, the call would look like this:

```
if (err = dsa_power_on_a(igrp, callback, (void*)1)) {
DSA_DIAG(err, igrp);
goto _error;
}
```

In the above call, the EDI library will start the power on (PWR.x=1) and return immediately. When the power on command finishes, the EDI library calls the function "callback" that was specified and passes it the value of 1 as user parameter.
The callback function must have the following syntax:

```
void callback(DSA_DEVICE_GROUP *grp, int err, void *param)
{
... do something when the command - here "power on" - is finished.
}
```

The first parameter is the usual device or device group identifier.
The second is the error code that was returned by the command that was executed. Here, it is the error code of the "power on" command. Usually the callback function does something different depending on whether there was an error or not. Here in our example, its prints a different message.

The third parameter is what was specified as the third parameter of the asynchronous function call: "1". Hence, in our example, the callback function looks like this:

```
void callback(DSA_DEVICE_GROUP *grp, int err, void *param)
{
   /* The asynchronous function has terminated with an error */
   if (err) {
     asyncFunctionError = err;
     printf("\n-> callback function called with error %d and parameter %d\n",
     err,(int)(param));
   }
   /* The asynchronous function has terminated without error */
   else {
     asyncFunctionError = 0;
     printf("\n->  callback  function  called  with  parameter  %d\n",
     (int)(param));
   }
   callbackCalled = 1;
}
```

Our callback sets the global variable `asyncFunctionError` as a means to transfer the command error code back to the main programme, which can then take the appropriate action. Notice also the last line, which sets a global variable to tell the main programme that the function has been called. We do this here because we want to display "*"s, pending the callback termination by writing:

```
    callbackCalled = 0;
    if (err = dsa_power_on_a(igrp, callback, (void*)1)) {
    DSA_DIAG(err, igrp);
    goto _error;
    }
    while (!callbackCalled) {
        printf("*");
        fflush(stdout);
        Sleep(100);
    }
```

Somewhere amongst the "*"s one or the other of the messages "callback function called with error XXX and parameter 1" or "callback function called with parameter 1" will be displayed, depending on how the power on terminates.
Since there is an asynchronous function for every "normal" one, it is also possible to wait for the end of a movement using an asynchronous function. In the example, this is illustrated with a homing command (IND.x). The homing command is started as usual, but waiting for the end of the associated movement is performed with an asynchronous call:

```
    if (err = dsa_homing_start_s(igrp, 10000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

    /* Initialization of main thread - callback function synchronous variable */
    callbackCalled = 0;
    if (err = dsa_wait_movement_a(igrp, callback, (void*)2)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

    /* Wait for callback function to be called and finished*/
    while (!callbackCalled) {
        printf("*");
        fflush(stdout);
        Sleep(100);
    }

    if (asyncFunctionError) {
        DSA_DIAG(asyncFunctionError, igrp);
        goto _error;
    }
```

Here amongst the "*"s, one or the other of the messages "callback function called with error XXX and parameter 2" or "callback function called with parameter 2" will be displayed, depending on how the homing terminates. Notice "2" is the difference with the previous message seen for the power on.

Notice also how the error code of the command can be processed as usual error codes using a global variable - `asyncFunctionError`, here - and the `DSA_DIAG` function.

13. Appendixes

13.1 DSA library functions

13.1.1 Functions to send commands

ETEL syntax	Command number	EDI function
ASG	235	
ASM	230	
ASP	234	
ASR	231	dsa_assign_slot_to_register_s dsa_assign_slot_to_register_a dsa_unassign_slot_to_register_s dsa_unassign_slot_to_register_a
ASS	252	
ASW	232	dsa_assign_slot_to_register_s dsa_assign_slot_to_register_a dsa_unassign_slot_to_register_s dsa_unassign_slot_to_register_a
AUT	150	
AXI	109	
BRK	70	dsa_quick_stop_s dsa_quick_stop_a
CAL	68	
CEC	58	
CLX	17	
CSH	21	
CTFOFF	301	
CTFON	300	
CTFRTV	302	
CWS	522	
DFCLD	210	
DIS	82	
DWN	42	
ECAT_ASW	229	
ECAT_ITP	117	
EIOMAXURST	755	dsa_externalIO_reset_max_update_time_s
EIOSTA	750	dsa_externalIO_set_enable_cyclic_update_s
EIOWDEN	720	dsa_externalIO_enable_watchdog_s dsa_externalIO_disable_watchdog_s
EIOWDSTP	721	dsa_externalIO_stop_watchdog_s
EIOWDTIME	722	dsa_externalIO_set_watchdog_time_s
END	0	dsa_stop_sequence_s dsa_stop_sequence_a dsa_stop_sequence_in_thread_s dsa_stop_sequence_in_thread_a
ERR	80	
FGA	313	
FLT	222	
FORCE_LIMIT	118	
FREESLOT	531	
FWD	520	
GETDINS	738	dsa_localIO_get_digital_input_state_s
GETDOUT	735	dsa_localIO_get_digital_output_s
GETEAINCS	746	dsa_externalIO_get_analog_input_converted_data_state_s
GETEAINRS	745	dsa_externalIO_get_analog_input_raw_data_state_s
GETEAOUTC	741	dsa_externalIO_get_analog_output_converted_data_s
GETEAOUTCS	743	dsa_externalIO_get_analog_output_converted_data_state_s
GETEAOUTR	740	dsa_externalIO_get_analog_output_raw_data_s
GETEAOUTRS	742	dsa_externalIO_get_analog_output_raw_data_state_s
GETEDINS	765	dsa_externalIO_get_digital_input_state_s

ETEL syntax	Command number	EDI function
GETEDOUT	760	dsa_externalIO_get_digital_output_s
GETEDOUTS	764	dsa_externalIO_get_digital_output_state_s
GETEREG	726	dsa_externalIO_get_modbus_register_s
GETMDINS	734	dsa_localIO_get_masked_digital_input_state_s
GETMDOUT	732	dsa_localIO_get_masked_digital_output_s
GETMEDINS	758	dsa_externalIO_get_masked_digital_input_state_s
GETMEDOUT	757	dsa_externalIO_get_masked_digital_output_s
GETMEDOUTS	766	dsa_externalIO_get_masked_digital_output_state_s
GETSLOT	530	
GGA	312	
HLB	121	dsa_quick_stop_s dsa_quick_stop_a dsa_ipol_quick_stop_s dsa_ipol_quick_stop_a
HLO	119	dsa_quick_stop_s dsa_quick_stop_a
HLT	120	dsa_quick_stop_s dsa_quick_stop_a dsa_ipol_quick_stop_s dsa_ipol_quick_stop_a
IABSCOORDS	556	dsa_ipol_abs_coords_s dsa_ipol_abs_coords_a
IABSMODE	555	dsa_ipol_set_abs_mode_s dsa_ipol_set_abs_mode_a
IBEGIN	553	dsa_ipol_begin_s dsa_ipol_begin_a
IBRK	653	dsa_ipol_quick_stop_s dsa_ipol_quick_stop_a
ICCW	1041	dsa_ipol_circle_ccw_c2d_s dsa_ipol_circle_ccw_c2d_a
ICCWR	1027	dsa_ipol_circle_ccw_r2d_s dsa_ipol_circle_ccw_r2d_a
ICLRB	657	dsa_ipol_clear_buffer_s dsa_ipol_clear_buffer_a
ICONC	1030	dsa_ipol_begin_concatenation_s dsa_ipol_begin_concatenation_a
ICONT	654	dsa_ipol_continue_s dsa_ipol_continue_a
ICW	1040	dsa_ipol_circle_cw_c2d_s dsa_ipol_circle_cw_c2d_a
ICWR	1026	dsa_ipol_circle_cw_r2d_s dsa_ipol_circle_cw_r2d_a
IEND	554	dsa_ipol_end_s
IEQ	151	
IGE	156	
IGT	154	
ILE	155	
ILINE	1025	dsa_ipol_line_s dsa_ipol_line_a dsa_ipol_line_2d_s dsa_ipol_line_2d_a
ILKT	1032	dsa_ipol_lkt_s dsa_ipol_lkt_a
ILOCK	1044	dsa_ipol_lock_s dsa_ipol_lock_a
ILT	153	
IMARK	1039	dsa_ipol_mark_s dsa_ipol_mark_a dsa_ipol_mark_2param_s dsa_ipol_mark_2param_a
IMRES	1063	
IMROT	1056	dsa_ipol_rotate_matrix_s dsa_ipol_rotate_matrix_a

ETEL syntax	Command number	EDI function
IMSCALE	1055	dsa_ipol_scale_matrix_s dsa_ipol_scale_matrix_a dsa_ipol_scale_matrix_2d_s dsa_ipol_scale_matrix_2d_a
IMSHEAR	1057	dsa_ipol_shear_matrix_s dsa_ipol_shear_matrix_a
IMTRANS	1054	dsa_ipol_translate_matrix_s dsa_ipol_translate_matrix_a dsa_ipol_translate_matrix_2d_s dsa_ipol_translate_matrix_2d_a
INCONC	1031	dsa_ipol_end_concatenation_s dsa_ipol_end_concatenation_
IND	45	dsa_homing_start_s dsa_homing_start_a
INE	152	
INI	44	
INS	41	
IPT	1045	dsa_ipol_pt_s dsa_ipol_pt_a
IPVT	1028	dsa_ipol_pvt_s dsa_ipol_pvt_a dsa_ipol_pvt_reg_typ_s dsa_ipol_pvt_reg_typ_a
IPVTUPDATE	662	dsa_ipol_pvt_update_s dsa_ipol_pvt_update_a
ISET	552	dsa_ipol_begin_s dsa_ipol_begin_a
ISTP	656	dsa_ipol_quick_stop_s dsa_ipol_quick_stop_a
ITACC	1036	dsa_ipol_tan_acceleration_s dsa_ipol_tan_acceleration_a
ITDEC	1037	dsa_ipol_tan_deceleration_s dsa_ipol_tan_deceleration_a
ITJRT	1038	dsa_ipol_tan_jerk_time_s dsa_ipol_tan_jerk_time_a
ITP	116	
ITRIG	1042	
ITSPD	1035	dsa_ipol_tan_velocity_s dsa_ipol_tan_velocity_a
IULINE	1033	dsa_ipol_uline_s dsa_ipol_uline_a dsa_ipol_uline_time_s dsa_ipol_uline_time_a
IUNLOCK	655	dsa_ipol_unlock_s dsa_ipol_unlock_a
IUNOCONC	1052	dsa_ipol_disable_uconcatenation_s dsa_ipol_disable_uconcatenation_a
IURELATIVE	1051	dsa_ipol_set_urelative_mode_s dsa_ipol_set_urelative_mode_a
IUSPDMASK	1053	dsa_ipol_uspeed_axis_mask_s dsa_ipol_uspeed_axis_mask_a
IUSPEED	1049	dsa_ipol_uspeed_s dsa_ipol_uspeed_a
IUTIME	1050	dsa_ipol_utime_s dsa_ipol_utime_a
IWTT	1029	
JBC	37	
JBS	36	
JEQ	137	
JGT	138	
JLT	136	
JMP	26	dsa_execute_sequence_s dsa_execute_sequence_a dsa_execute_sequence_in_thread_s dsa_execute_sequence_in_thread_a

ETEL syntax	Command number	EDI function
JNE	139	
MAM	199	
MCS	200	
MCT	193	
MDA	195	
MDT	192	
MMO	194	
MSI	190	
MSR	196	
MSV	191	
MTP	197	
MTU	198	
MVE	60	dsa_set_target_position_s dsa_set_target_position_a dsa_start_profiled_movement_s dsa_start_profiled_movement_a
NEW	78	dsa_default_parameters_s dsa_default_parameters_a
NOG	149	
PBK	221	
PCT	752	
POP	34	
PTS	76	
PWR	124	dsa_quick_stop_s dsa_quick_stop_a dsa_power_on_s dsa_power_on_a
PWR_MUL	83	
RCT	754	dsa_externalIO_reset_client_communication_s
RES	49	dsa_load_parameters_s dsa_load_parameters_a
RET	69	
RIC	753	dsa_externalIO_reset_io_cycle_count_s
RMVE	62	dsa_start_relative_profiled_movement_s dsa_start_relative_profiled_movement_a
RSD	88	
RSH	600	
RST	79	dsa_reset_error_s dsa_reset_error_a
RSTDOUT	737	dsa_localIO_reset_digital_output_s
RSTEDOUT	762	dsa_externalIO_reset_digital_output_s
RSU	601	
SAV	48	dsa_save_parameters_s dsa_save_parameters_a
SCI	102	
SDF	240	
SDP	248	
SDS	245	
SEQBKPALL	142	dsa_debug_sequence_enable_breakpoint_everywhere_s dsa_debug_sequence_enable_breakpoint_everywhere_a
SEQBRKTHR	143	
SEQCONT	140	dsa_debug_sequence_continue_s dsa_debug_sequence_continue_a
SEQSETENBP	141	dsa_debug_sequence_enable_breakpoint_at_s dsa_debug_sequence_enable_breakpoint_at_a
SET	22	
SETDOUT	736	dsa_localIO_set_digital_output_s
SETEAOUTC	731	dsa_externalIO_set_analog_output_converted_data_s
SETEAOUTR	730	dsa_externalIO_set_analog_output_raw_data_s
SETEDOUT	761	dsa_externalIO_set_digital_output_s
SETEREG	725	dsa_externalIO_set_modbus_register_s
SETMDOUT	733	dsa_localIO_apply_mask_digital_output_s

ETEL syntax	Command number	EDI function
SETMEDOUT	763	dsa_externalIO_apply_mask_digital_output_s
SLS	46	
SMP	61	
SPG	105	
SPI2	220	
STA	25	dsa_new_setpoint_s dsa_new_setpoint_a dsa_change_setpoint_s dsa_change_setpoint_a
STE_ABS	129	dsa_step_motion_s dsa_step_motion_a
STE_ADD	114	
STE_SUB	115	
STI	33	
STP	18	dsa_quick_stop_s dsa_quick_stop_a
STV	20	
SUF	242	
SUMAG	239	
SUP	249	dsa_start_upload_register_s
SUS	246	dsa_start_upload_sequence_s
SUT	247	dsa_start_upload_trace_s
TCPERR	693	
TMA	236	
TNS	77	
TRANCRC	694	
TRANSYNC	695	
TRE	104	
TRESET	540	
TRF	233	
TRM	103	
TRR	106	
TST	74	
UDPERR	692	
UST	30	
WAB	13	
WBC	54	dsa_wait_bit_clear_s dsa_wait_bit_clear_a
WBS	55	dsa_wait_bit_set_s dsa_wait_bit_set_a
WPG	53	
WPL	52	
WSBC	515	
WSBS	514	
WSG	57	dsa_wait_sgn_register_greater_s dsa_wait_sgn_register_greater_a
WSL	56	dsa_wait_sgn_register_lower_s dsa_wait_sgn_register_lower_a
WTK	513	dsa_ipol_wait_mark_s dsa_ipol_wait_mark_a
WTM	8	dsa_wait_movement_s dsa_wait_movement_a
WTP	9	dsa_wait_position_s dsa_wait_position_a
WTS	12	
WTT	10	dsa_wait_time_s dsa_wait_time_a
WTW	11	dsa_wait_window_s dsa_wait_window_a

ETEL syntax	Command number	EDI function
XYY_ABS	123	dsa_set_register_s dsa_set_register_a dsa_set_register_int32_s dsa_set_register_int32_a
XYY_ADD	91	
XYY_AND	95	
XYY_AND_NOT	97	
XYY_CONV	122	
XYY_DIV	94	
XYY_MODULO	101	
XYY_MUL	93	
XYY_NOT	174	
XYY_ORL	96	
XYY_ORL_NOT	98	
XYY_SET_MULTI	125	
XYY_SHL	173	
XYY_SHR	172	
XYY_SUB	92	
XYY_XOR	99	
XYY_XOR_NOT	100	
YLD	205	
ZFT	203	dsa_acquisition_acquire_s dsa_acquisition_acquire_a dsa_sync_trace_force_trigger_s dsa_sync_trace_force_trigger_a
ZTE	204	dsa_acquisition_acquire_s dsa_acquisition_acquire_a dsa_sync_trace_enable_s dsa_sync_trace_enable_a

13.1.2 Functions for the reading and the writing of the registers

Registers	Alias	EDI read function	EDI write function	Remark
C6	XDOUT	dsa_get_x_digital_output_s	dsa_set_x_digital_output_s	
C7:0	XAOUT	dsa_get_x_analog_output_1_s	dsa_set_x_analog_output_1_s	
C7:1	XAOUT	dsa_get_x_analog_output_2_s	dsa_set_x_analog_output_2_s	
C7:2	XAOUT	dsa_get_x_analog_output_3_s	dsa_set_x_analog_output_3_s	
C7:3	XAOUT	dsa_get_x_analog_output_4_s	dsa_set_x_analog_output_4_s	
C30	XSRT	dsa_get_mon_source_type_s	dsa_set_mon_source_type_s	
C31	XSRI	dsa_get_mon_source_index_s	dsa_set_mon_source_index_s	
C107		dsa_get_analog_output_s	dsa_set_analog_output_s	For AccurET Firmware < 2.05A
CF107	AOUT	dsa_get_analog_output_s	dsa_set_analog_output_s	For AccurET Firmware >= 2.05A
KF1	KPP	dsa_get_pl_proportional_gain_s	dsa_set_pl_proportional_gain_s	
KF2	KDP	dsa_get_pl_speed_feedback_gain_s	dsa_set_pl_speed_feedback_gain_s	
KF4	KIP	dsa_get_pl_integrator_gain_s	dsa_set_pl_integrator_gain_s	
KF5	KAWP	dsa_get_pl_anti_windup_gain_s	dsa_set_pl_anti_windup_gain_s	
KF6		dsa_get_pl_integrator_limitation_s	dsa_set_pl_integrator_limitation_s	
K7		dsa_get_pl_integrator_mode_s	dsa_set_pl_integrator_mode_s	
K11		dsa_get_ttl_speed_filter_s	dsa_set_ttl_speed_filter_s	
KF21	KAFFP	dsa_get_pl_acc_feedforward_gain_s	dsa_set_pl_acc_feedforward_gain_s	
KL27		dsa_get_max_position_range_limit_s	dsa_set_max_position_range_limit_s	
K30		dsa_get_following_error_window_s	dsa_set_following_error_window_s	
K31		dsa_get_velocity_error_limit_s	dsa_set_velocity_error_limit_s	
K32		dsa_get_switch_limit_mode_s	dsa_set_switch_limit_mode_s	
K33		dsa_get_enable_input_mode_s	dsa_set_enable_input_mode_s	
KL34	MINSL	dsa_get_min_soft_position_limit_s	dsa_set_min_soft_position_limit_s	
KL35	MAXSL	dsa_get_max_soft_position_limit_s	dsa_set_max_soft_position_limit_s	
K36	MODESL	dsa_get_profile_limit_mode_s	dsa_set_profile_limit_mode_s	
K38	TIMEW	dsa_get_position_window_time_s	dsa_set_position_window_time_s	
K39	POSW	dsa_get_position_window_s	dsa_set_position_window_s	
K40	HMODE	dsa_get_homing_method_s	dsa_set_homing_method_s	
KL41	HSPD	dsa_get_homing_zero_speed_s	dsa_set_homing_zero_speed_s	
KL42	HACC	dsa_get_homing_acceleration_s	dsa_set_homing_acceleration_s	
KL43		dsa_get_homing_following_limit_s	dsa_set_homing_following_limit_s	
KF44		dsa_get_homing_current_limit_s	dsa_set_homing_current_limit_s	
KL45	HOFFS	dsa_get_home_offset_s	dsa_set_home_offset_s	
KL46		dsa_get_homing_fixed_mvt_s	dsa_set_homing_fixed_mvt_s	
KL47		dsa_get_homing_switch_mvt_s	dsa_set_homing_switch_mvt_s	
KL48		dsa_get_homing_index_mvt_s	dsa_set_homing_index_mvt_s	
K52		dsa_get_homing_fine_tuning_mode_s	dsa_set_homing_fine_tuning_mode_s	
K53		dsa_get_homing_fine_tuning_value_s	dsa_set_homing_fine_tuning_value_s	
K56		dsa_get_motor_phase_correction_s	dsa_set_motor_phase_correction_s	
KF60	IPEAK	dsa_get_software_current_limit_s	dsa_set_software_current_limit_s	
K61		dsa_get_drive_control_mode_s	dsa_set_drive_control_mode_s	
K66		dsa_get_display_mode_s	dsa_set_display_mode_s	
K68		dsa_get_encoder_inversion_s	dsa_set_encoder_inversion_s	
K70		dsa_get_encoder_phase_1_offset_s	dsa_set_encoder_phase_1_offset_s	
K71		dsa_get_encoder_phase_2_offset_s	dsa_set_encoder_phase_2_offset_s	
KF72		dsa_get_encoder_phase_1_factor_s	dsa_set_encoder_phase_1_factor_s	
KF73		dsa_get_encoder_phase_2_factor_s	dsa_set_encoder_phase_2_factor_s	
K75		dsa_get_encoder_index_distance_s	dsa_set_encoder_index_distance_s	
KF80	KPC	dsa_get_cl_proportional_gain_s	dsa_set_cl_proportional_gain_s	
KF81	KIC	dsa_get_cl_integrator_gain_s	dsa_set_cl_integrator_gain_s	
KF83		dsa_get_cl_current_limit_s	dsa_set_cl_current_limit_s	
KF84		dsa_get_cl_i2t_current_limit_s	dsa_set_cl_i2t_current_limit_s	
KF85		dsa_get_cl_i2t_time_limit_s	dsa_set_cl_i2t_time_limit_s	
K90		dsa_get_init_mode_s	dsa_set_init_mode_s	
KF91		dsa_get_init_pulse_level_s	dsa_set_init_pulse_level_s	
KF92		dsa_get_init_max_current_s	dsa_set_init_max_current_s	

Registers	Alias	EDI read function	EDI write function	Remark
K93		dsa_get_init_final_phase_s	dsa_set_init_final_phase_s	
K94		dsa_get_init_time_s	dsa_set_init_time_s	
K97		dsa_get_init_initial_phase_s	dsa_set_init_initial_phase_s	
K120			see dsa_acquisition functions	
K121			see dsa_acquisition functions	
K122			see dsa_acquisition functions	
K123			see dsa_acquisition functions	
K124			see dsa_acquisition functions	
K125			see dsa_acquisition functions	
K126			see dsa_acquisition functions	
K127			see dsa_acquisition functions	
KD127			see dsa_acquisition functions	
KF127			see dsa_acquisition functions	
KL127			see dsa_acquisition functions	
K128			see dsa_acquisition functions	
K129			see dsa_acquisition functions	
K164		dsa_get_syncro_start_timeout_s	dsa_set_syncro_start_timeout_s	
K171	DOUT	dsa_get_digital_output_s	dsa_set_digital_output_s	
K198		dsa_get_indirect_register_idx_s	dsa_set_indirect_register_idx_s	
K201	MMC	dsa_get_concatenated_mvt_s	dsa_set_concatenated_mvt_s	
K202	MMD	dsa_get_profile_type_s	dsa_set_profile_type_s	
K203	LTN	dsa_get_mvt_lkt_number_s	dsa_set_mvt_lkt_number_s	
K204	LTI	dsa_get_mvt_lkt_time_s	dsa_set_mvt_lkt_time_s	
KF205	CAM	dsa_get_came_value_s	dsa_set_came_value_s	
KL206	BRKDEC	dsa_get_brake_deceleration_s	dsa_set_brake_deceleration_s	
K210:1 .. K210:3		dsa_get_target_position_s	dsa_set_target_position_s	
KL210:1 .. KL210:3	TARGET	dsa_get_target_position_s	dsa_set_target_position_s	
KL211	SPD	dsa_get_profile_velocity_s	dsa_set_profile_velocity_s	
KL212	ACC	dsa_get_profile_acceleration_s	dsa_set_profile_acceleration_s	
K213	JRT	dsa_get_jerk_time_s	dsa_set_jerk_time_s	
K220		dsa_get_ctrl_source_type_s	dsa_set_ctrl_source_type_s	
K221		dsa_get_ctrl_source_index_s	dsa_set_ctrl_source_index_s	
K223		dsa_get_ctrl_offset_s	dsa_set_ctrl_offset_s	
KF224		dsa_get_ctrl_gain_s	dsa_set_ctrl_gain_s	
K239		dsa_get_motor_kt_factor_s	dsa_set_motor_kt_factor_s	
K530	ISPDRATE	dsa_get_ipol_velocity_rate_s	dsa_set_ipol_velocity_rate_s	
M2		dsa_get_position_ctrl_error_s		
M3		dsa_get_position_max_error_s		
ML6		dsa_get_position_demand_value_s		
ML7		dsa_get_position_actual_value_s		
M10		dsa_get_velocity_demand_value_s		
M11		dsa_get_velocity_actual_value_s		
M14		dsa_get_acc_demand_value_s		
MF20	RCUR1	dsa_get_cl_current_phase_1_s		
M21		dsa_get_		
MF21	RCUR2	dsa_get_cl_current_phase_2_s		
MF22	RCUR3	dsa_get_cl_current_phase_3_s		
M25		dsa_get_cl_lkt_phase_1_s		
M25:1		dsa_get_cl_lkt_phase_2_s		
M25:2		dsa_get_cl_lkt_phase_3_s		
MF30	TIQ	dsa_get_cl_demand_value_s		
MF31	RIQ	dsa_get_cl_actual_value_s		
M40	SINENC	dsa_get_encoder_sine_signal_s		
M41	COSENC	dsa_get_encoder_cosine_signal_s		
M48		dsa_get_encoder_hall_dig_signal_s		
M50	DIN	dsa_get_digital_input_s		
MF51	AIN	dsa_get_analog_input_s		
M55	XDIN	dsa_get_x_digital_input_s		

Registers	Alias	EDI read function	EDI write function	Remark
M56:0	XAIN	dsa_get_x_analog_input_1_s		
M56:1	XAIN	dsa_get_x_analog_input_2_s		
M56:2	XAIN	dsa_get_x_analog_input_3_s		
M56:3	XAIN	dsa_get_x_analog_input_4_s		
M60	SD1	dsa_get_drive_status_1_s		
M61	SD2	dsa_get_drive_status_2_s		
M64	ERRCD	dsa_get_error_code_s		
M66	WARCD	dsa_get_warning_code_s		
M67	SEQNBP	dsa_debug_sequence_get_nb_breakpoints_s		
MF67		dsa_get_cl_i2t_value_s		
M77	SEQTHRBKD	dsa_debug_sequence_get_break_thread_nb_s		
M87		dsa_get_axis_number_s		
M90	CTEMP	dsa_get_drive_temperature_s		
M95		dsa_get_drive_display_s		
M96		dsa_get_drive_sequence_line_s		
M140		dsa_get_drive_fuse_status_s		
M282		dsa_get_parameters_version_s		
M475		dsa_get_sequence_code_usage_s		
M476		dsa_get_sequence_data_usage_s		
M477		dsa_get_sequence_source_usage_s		

13.2 Main programming changes when passing from EDI30 to EDI40

13.2.1 All files including EDI header files must include new version of header files

EDI30	EDI40
#include<etne20.h>	#include <etne40.h>
#include<tra20.h>	#include <tra40.h>
#include<dsa30.h>	#include <dsa40.h>
#include<etb20.h>	#include <etb40.h>
#include<dmd20.h>	#include <dmd40.h>
#include<ekd20.h>	#include <ekd40.h>
#include<lib20.h>	#include <lib40.h>
#include<esc10.h>	#include<esc40.h>

13.2.2 The application must be linked with new version of library files

EDI30	EDI40
-L etne20c.lib	-L etne40c.lib
-L tra20c.lib	-L tra40c.lib
-L dsa30c.lib	-L dsa40c.lib
-L etb20c.lib	-L etb40c.lib
-L dmd20c.lib	-L dmd40c.lib
-L ekd20c.lib	-L ekd40c.lib
-L lib20c.lib	-L lib40c.lib
-L esd10c.lib	-L esd40c.lib
-L esc10c.lib	-L esc40c.lib (available for 32-bits application only)

13.2.3 EDI obsolete functions

13.2.3.1 Register and functionality remove

EDI, especially DSA library, provides a huge amount of specific functions. EDI40 does not support old DSA, DSB and DSC devices any more. Some specific functions were designed to access DSA/DSB/DSC specific registers. These functions are no more available:

Obsolete functions due to register or functionality remove
dsa_set_trace_mode_mvt_s/a
dsa_set_trace_mode_pos_s/a
dsa_set_trace_mode_dev_s/a
dsa_set_trace_mode_iso_s /a
dsa_set_trace_mode_immediate_s/a
dsa_trace_acquisition_s/a
dsa_ipol_reset_s/a
dsa_wait_window_user_channel_s/a
dsa_wait_movement_user_channel_s/a
dsa_wait_time_user_channel_s/a
dsa_wait_position_user_channel_s/a
dsa_wait_sgn_register_greater_user_channel_s/a
dsa_wait_sgn_register_lower_user_channel_s/a
dsa_wait_bit_set_user_channel_s/a
dsa_wait_bit_clear_user_channel_s/a
dsa_get_rtm_mon
dsa_init_rtm_fct
dsa_start_rtm
dsa_stop_rtm
dsa_edit_sequence_s/a

Obsolete functions due to register or functionality remove
dsa_exit_sequence_s/a
dsa_can_command_1_s/a
dsa_can_command_2_s/a
dsa_get_ebl_baudrate_s/a
dsa_get_drive_fuse_checking_s/a
dsa_get_motor_temp_checking_s/a
dsa_get_interrupt_mask_1_s/a
dsa_get_interrupt_mask_2_s/a
dsa_get_indirect_axis_number_s/a
dsa_get_indirect_register_sidx_s/a
dsa_get_daisy_chain_number_s/a
dsa_get_drive_mask_value_s/a
dsa_get_irq_drive_status_1_s/a
dsa_get_irq_drive_status_2_s/a
dsa_get_ack_drive_status_1_s/a
dsa_get_ack_drive_status_2_s/a
dsa_get_irq_pending_axis_mask_s/a
dsa_get_encoder_phase_3_factor_s/a
dsa_get_encoder_phase_3_offset_s/a
dsa_get_encoder_index_signal_s/a
dsa_get_encoder_hall_1_signal_s/a
dsa_get_encoder_hall_2_signal_s/a
dsa_get_encoder_hall_3_signal_s/a
dsa_get_apr_input_filter_s/a
dsa_get_cl_regen_mode_s/a
dsa_get_pdr_step_value_s/a
dsa_get_ctrl_shift_factor_s/a
dsa_get_ref_demand_value_s/a
dsa_get_drive_control_mask_s/a
dsa_get_cl_output_filter_s/a
dsa_get_cl_input_filter_s/a
dsa_get_cl_phase_advance_factor_s/a
dsa_get_cl_phase_advance_shift_s/a
dsa_get_pl_speed_feedfwd_gain_s/a
dsa_get_pl_force_feedback_gain_1_s/a
dsa_get_pl_force_feedback_gain_2_s/a
dsa_get_pl_output_filter_s/a
dsa_get_pl_speed_filter_s/a
dsa_get_ttl_special_filter_s/a
dsa_get_init_current_rate_s/a
dsa_get_init_phase_rate_s/a
dsa_get_end_velocity_s/a
dsa_get_profile_deceleration_s/a
dsa_get_min_position_range_limit_s/a
dsa_get_max_profile_velocity_s/a
dsa_get_max_acceleration_s/a
dsa_get_acc_actual_value_s/a
dsa_get_io_error_event_mask_s/a
dsa_get_syncro_input_mask_s/a
dsa_get_syncro_input_value_s/a
dsa_get_syncro_output_mask_s/a
dsa_get_syncro_output_value_s/a
dsa_get_x_analog_gain_s/a
dsa_get_x_analog_offset_s/a
dsa_get_can_feedback_1_s/a
dsa_get_can_feedback_2_s/a
dsa_get_mon_dest_index_s/a
dsa_get_mon_gain_s/a
dsa_get_mon_offset_s/a

Obsolete functions due to register or functionality remove
dsa_get_trigger_map_offset_s/a
dsa_get_trigger_map_size_s/a
dsa_get_trigger_io_mask_s/a
dsa_get_trigger_irq_mask_s/a
dsa_get_realtime_enabled_global_s/a
dsa_get_realtime_valid_mask_s/a
dsa_get_realtime_enabled_mask_s/a
dsa_get_realtime_pending_mask_s/a
dsa_set_ebl_baudrate_s/a
dsa_set_drive_fuse_checking_s/a
dsa_set_motor_temp_checking_s/a
dsa_set_interrupt_mask_1_s/a
dsa_set_interrupt_mask_2_s/a
dsa_set_indirect_axis_number_s/a
dsa_set_indirect_register_sidx_s/a
dsa_set_encoder_phase_3_factor_s/a
dsa_set_encoder_phase_3_offset_s/a
dsa_set_apr_input_filter_s/a
dsa_set_cl_regen_mode_s/a
dsa_set_pdr_step_value_s/a
dsa_set_ctrl_shift_factor_s/a
dsa_set_cl_output_filter_s/a
dsa_set_cl_input_filter_s/a
dsa_set_cl_phase_advance_factor_s/a
dsa_set_cl_phase_advance_shift_s/a
dsa_set_pl_speed_feedfwd_gain_s/a
dsa_set_pl_force_feedback_gain_1_s/a
dsa_set_pl_force_feedback_gain_2_s/a
dsa_set_pl_output_filter_s/a
dsa_set_pl_speed_filter_s/a
dsa_set_ttl_special_filter_s/a
dsa_set_init_current_rate_s/a
dsa_set_init_phase_rate_s/a
dsa_set_end_velocity_s/a
dsa_set_profile_deceleration_s/a
dsa_set_min_position_range_limit_s/a
dsa_set_max_profile_velocity_s/a
dsa_set_max_acceleration_s/a
dsa_set_io_error_event_mask_s/a
dsa_set_syncro_input_mask_s/a
dsa_set_syncro_input_value_s/a
dsa_set_syncro_output_mask_s/a
dsa_set_syncro_output_value_s/a
dsa_set_x_analog_gain_s/a
dsa_set_x_analog_offset_s/a
dsa_set_mon_dest_index_s/a
dsa_set_mon_gain_s/a
dsa_set_mon_offset_s/a
dsa_set_trigger_map_offset_s/a
dsa_set_trigger_map_size_s/a
dsa_set_trigger_io_mask_s/a
dsa_set_trigger_irq_mask_s/a
dsa_set_realtime_enabled_global_s/a
dsa_set_realtime_valid_mask_s/a
dsa_set_realtime_pending_mask_s/a
etb_get_baudrate
etb_multi_send
etb_start_rtm
etb_stop_rtm

Obsolete functions due to register or functionality remove
etb_init_rtm_fct
etb_get_rtm_mon
etb_link_error
etb_irq_watchdog
etb_get_bus_counters
etb_add_rt_handler
etb_remove_rt_handler
etb_auto_number
etb_activate_download
etb_start_download_file
etb_start_upload_file
etb_etcom_multi_send
etb_etcom_start_download_file
etb_etcom_start_upload_file

13.2.3.2 Axis addressing

DSC and AccurET families differ concerning axis addressing:

- DSC family allows 32 axes, where DSMAX has number 31.
- AccurET family allows 64 axes, where UltimET has number 63.

EDI functions with parameters allowing axis addressing have been doubled in EDI3 (Example: dsa_get_etb_axis was dedicated to DSC family and returned an axis number 0 and 31, while dsa_etcom_get_etb_axis was dedicated to AccurET family and returned an axis number 0 and 63). As EDI4 does not support DSC any more, the DSC dedicated functions addressing axis have been removed:

Obsolete functions due to DSC family axis addressing
dsa_open_e
dsa_open_ef
dsa_get_etb_axis
dsa_create_auto_e
etb_get_baudrate
etb_multi_send
etb_get_drv_present
etb_get_drv_status
etb_get_drv_info
etb_get_ext_info
etb_diag
etb_sdiag
etb_fdiag
etb_putm
etb_putr
etb_getm
etb_getr
etb_start_download
etb_start_upload
etb_download_firmware
tra_upload_register_stream_e
tra_download_register_stream_e
tra_download_register_stream_e2
tra_upload_limited_register_stream_e
tra_send_direct_stream_e
tra_get_axis_mask_e
tra_set_axis_mask_e
tra_set_preference_axis_mask
tra_get_preference_axis_mask

13.2.3.3 *Record accessing*

DSC and AccurET families differ concerning the record used on the communication protocol:

- DSC family uses ETBREC fixed size record.
- AccurET family uses ETCOM variable length record.

As EDI3.xx supports both DSC and ETEL products, some functions allowing management of records have been implemented. The functions for the old ETBREC record have been removed:

Obsolete functions due to DSC-family ETBREC record
tra_translate_cmd_to_ascii
tra_translate_cmd_to_ascii_ex
tra_translate_cmd_from_ascii
tra_translate_cmd_from_ascii_ex
tra_translate_rqs_to_ascii
tra_translate_rqs_to_ascii_ex
tra_get_iso_converter
tra_set_iso_converter
etb_etcom_to_recs
etb_recs_to_etcom

13.2.3.4 *NON ANSI functions*

Some non ANSI functions have been removed. These are the functions returning a whole structure content. These are especially the functions allowing the initialization of a structure:

Obsolete non ANSI functions
dsa_init_status
dsa_init_info
dsa_init_x_info
dsa_init_vector
dsa_init_vector_typ
dsa_init_rtm
etb_init_drv_info
etb_init_timeouts
etb_init_ext_info
etb_init_drv_status
etb_init_rec_param
etb_init_counters
etb_init_bus_status
etb_init_svr_info
etb_activate_status
etb_init_master_info

The call to these functions can be replaced by:

Example:

DSA_STATUS status = dsa_init_status();
replaced by
DSA_STATUS status = {sizeof(DSA_STATUS)};

13.2.3.5 *Functions managing special ETEL devices*

A special ETEL device called ’gp_module’ was developed. This device was a special ’TEB’ device. All functions accessing this device have been removed:

Obsolete ’GP_MODULE’ special device functions
dsa_create_gp_module
dsa_create_gp_module_group
dsa_is_valid_gp_module
dsa_is_valid_gp_module_group
dsa_is_valid_gp_module_base

13.2.3.6 Special functions accessing ’DSMAX’

Some functions accessing DSMAX must be used in EDI3 to access UltimET as well. These functions have been renamed into ’...MASTER...’, providing a more generic name. To allow portability between EDI3 and EDI4, the old functions name still exists, but will be removed in further EDI version:

Obsolete ’DSMAX’ functions	
dsa_create_dsmax	dsa_create_master
dsa_create_ dsmax _group	dsa_create_ master _group
dsa_is_valid_ dsmax	dsa_is_valid_master
dsa_is_valid_dsmax _group	dsa_is_valid_master _group
dsa_is_valid_dsmax_base	dsa_is_valid_master_base
dsa_set_dsmax	dsa_set_master
dsa_get_dsmax	dsa_get_master

Obsolete ’DSMAX’ objects	
DSA_DSMAX	DSA_MASTER
DSA_DSMAX_GROUP	DSA_MASTER_GROUP
DSA_DSMAX_BASE	DSA_MASTER_BASE

13.2.3.7 Function allowing old ETEL sequence translation/download/upload

The ETEL sequences on DSC family were interpreted by the controller. These sequences were saved into S registers of the controller. The functions, allowing translation, download and upload of such sequences into S registers, have been removed:

Obsolete ETEL sequence translation/download/upload
tra_is_valid_sequence_traductor
tra_create_sequence_traductor_o
tra_create_sequence_traductor_o2
tra_clear_sequence_drive_map
tra_setup_sequence_drive_map
tra_etcom_setup_sequence_drive_map
tra_is_valid_sequence_traductor_e
tra_create_sequence_traductor_e
tra_download_sequence_stream_e
tra_upload_sequence_stream_e
tra_get_sequence_line_e

13.2.4 EDI 64-bit unavailable functionality

Inside EDI, a dll allowing new ETEL sequence compilation was provided on EDI3.xx. This dll is no more available on EDI 64-bit native dll. Compilation of ETEL sequence must be done with a 32-bit application as ComET.

13.3 Diagnostic (DSA_DIAG()) output description

`DSA_DIAG()` outputs a mixture of valuable information meant in part for the developer of the application, and in part for the ETEL DLL customer support should that be necessary. The output of DSA_DIAG is the standard output. If the user application does not redirect its standard output or if the standard output is not visible, it is advised to use DSA_SDIAG or even DSA_FDIAG to store the error message.

The layout of the output comes in three parts:

- the first line is very important for **you**, the application developer, because it gives you a report on the state of the situation: communication status and drive status. With this line, and your knowledge of the application and the system, you should be able to identify the source of the problem in a large majority of the cases.
- the following lines constitutes the calling stack trace and is primarily meant for ETEL. You will notice that it gives precise information on the execution flow (file name, function called and line number) that led to the error. It is for this piece of information that you are strongly encouraged to systematically call `DSA_DIAG()` after each operation used from the EDI package. **You will be formally required to give this information to ETEL's customer support for any problem that is reported to them.**
- the last line is a summarizing error message

Remark: All DSA_DIAG have their more readable version, called DSA_EXT_DIAG, DSA_EXT_SDIAG, DSA_EXT_FDIAG. These versions will display an extended form of the error.

13.3.1 Example 1

Programming example:

```
DSA_DRIVE *drv1;
if (err = dsa_create_drive(&drv1)) {
     DSA_DIAG(err, drv1);
     goto _error;
}
```

Output:

```
.\tests.c (2193): ERROR -322: bad parameter (EDI package 0x3088080)
     Bad device pointer (magic 0xc9850e8b)
     .\dsaopn.c (dsa_create_drive: 839)
     ERROR -322: bad parameter
```

In this case, the user forgot to initialize the drv1 pointer. By calling dsa_create_drive function, EDI checks that the pointer has been initialized to NULL. If it is not the case, it returns a DSA_EBADPARAM error. DSA_DIAG is called by passing the pointer as parameter. DSA_DIAG prints out that the pointer is not valid.

13.3.2 Example 2

Programming example:

```
DSA_DRIVE *drv1;
if (err = dsa_create_drive(&drv1)) {
     DSA_DIAG(err, NULL);
     goto _error;
}
```

Output:

```
.\tests.c (2193): ERROR -322 : bad parameter (EDI package 0x3088080)
     Null device pointer
     .\dsaopn.c (dsa_create_drive: 839)
     ERROR -322: bad parameter
```

In this case again, the user forgot to initialize the drv1 pointer. By calling dsa_create_drive function, EDI checks that the pointer has been initialized to NULL. If it is not the case, it returns a DSA_EBADPARAM error. DSA_DIAG is called by passing NULL as the pointer. DSA_DIAG prints out that the pointer is NULL.

13.3.3 Example 3

Programming example:

```
DSA_DRIVE *drv1 = NULL;
if (err = dsa_create_drive(&drv1)) {
     DSA_DIAG(err, NULL);
     goto _error;
}
if (err = dsa_open_u(drv1, "etb:ULTIMET:2")) {
     DSA_DIAG(err, drv1);
     goto _error;
}
```

Output:

```
.\tests.c (2215): ERROR -320: no drive response (EDI package 0x3088080)
     on device 2 (status 0x0 0x0)
     .\dsaopn.c (dsa_open_u: 1434)
     .\dsaopn.c (_dsa_open_etb: 423)
     .\dsaopn.c (dsa_reset: 1700)
     ERROR -320: no drive response
```

In this case, the user try to open a not present axis. The call to dsa_open_u function returns an error. DSA_DIAG prints out that the axis 2 does not respond. The user can also see this by analysing the printed out drive status (0x0 0x0).

13.3.4 Example 4

Programming example:

```
DSA_DRIVE *drv1 = NULL;
DSA_DRIVE *drv2 = NULL;
DSA_DRIVE_GROUP *grp = NULL;
int err = 0;

if (err = dsa_create_drive(&drv1)) {
     DSA_DIAG(err, NULL);
     goto _error;
}
if (err = dsa_create_drive(&drv2)) {
     DSA_DIAG(err, drv2);
     goto _error;
}
if (err = dsa_create_drive_group(&grp, 2)) {
     DSA_DIAG(err, grp);
     goto _error;
}
if (err = dsa_set_group_item(grp, 0, drv1)) {
     DSA_DIAG(err, drv1);
     goto _error;
}
if (err = dsa_set_group_item(grp, 1, drv2)) {
     DSA_DIAG(err, drv2);
     goto _error;
```

```
    }
    if (err = dsa_open_u(drv1, "etb:ULTIMET:0")) {
         DSA_DIAG(err, drv1);
         goto _error;
    }
    if (err = dsa_open_u(drv2, "etb:ULTIMET:1")) {
         DSA_DIAG(err, drv2);
         goto _error;
    }
    if (err = dsa_wait_time_s(grp, 5.0, 10)) {
         DSA_DIAG(err, grp);
         goto _error;
    }
```

Output:

```
.\tests.c (2231): ERROR -310: timeout error (EDI package 0x3088080)
     on device 0 (status 0x28 0x0)
     on device 1 (status 0x28 0x0)
     .\dsawai.c (dsa_wait_time_s: 848)
     .\dsatrn.c (dsa_commit_sync_trans: 202)
     .\dsamsg.c (_dsa_wait_sync_event: 733)
     ERROR -310: timeout error
```

In this case, the user call WTT function with a EDI timeout as 10 ms. This follows to a DSA_ETIMEOUT error. DSA_DIAG is called passing the device group as parameter. DSA_DIAG prints out the error, and the status of each drive present in the group.

13.3.5 Example 5

```
DSA_DRIVE *drv1 = NULL;
DSA_DRIVE *drv2 = NULL;
int err = 0;

if (err = dsa_create_drive(&drv1)) {
     DSA_DIAG(err, NULL);
     goto _error;
}
if (err = dsa_create_drive(&drv2)) {
     DSA_DIAG(err, drv2);
     goto _error;
}
if (err = dsa_open_u(drv1, "etb:ULTIMET:0")) {
     DSA_DIAG(err, drv1);
     goto _error;
}
if (err = dsa_open_u(drv2, "etb:ULTIMET:1")) {
     DSA_DIAG(err, drv2);
     goto _error;
}
if (err = dsa_power_on_s(drv1, 1000)) {
     DSA_DIAG(err, drv1);
     goto _error;
}
if (err = dsa_homing_start_s(drv1, 1000)) {
     DSA_DIAG(err, drv1);
     goto _error;
}
```

Output:

```
  .\tests.c (2231): ERROR -311: drive in error (EDI package 0x4018000)
         on device 0 (status 0x2428 0x0) (device-error 67: Movement not possible
         (drive not in power on))
         u:\pc\edi\sw_dsa\v30\c\dsafct.c (dsa_homing_start_s: 472)
         u:\pc\edi\sw_dsa\v30\c\dsasyn.c (_dsa_sync_cmd_1c: 422)
         u:\pc\edi\sw_dsa\v30\c\dsasyn.c (_dsa_sync_cmd_1cn: 596)
         u:\pc\edi\sw_dsa\v30\c\dsasyn.c (_dsa_sync_cmd_ncn: 571) (AxisMask 0x1
         Rec 0x20, Cmd 0x2d 1000

         u:\pc\edi\sw_dsa\v30\c\dsamsg.c (_dsa_wait_sync_event: 729)
         ERROR -311: drive in error
```

In this case, the user calls dsa_homing_start on a drive which is not powered on. This follows to a DSA_EDRVERR error. DSA_DIAG prints out that the axis 0 cannot be homed because it is not powered on.

13.4 Listing of the manual’s example

```
/*
 * This is the example that is referred to in the EDI package User's Manual.
 *
 * This examples aims at using the library to control  two axes concurrently via a ULTIMET.
 * The ULTIMET is a ULTIMET-PCI and the controllers are one AccurET 400.
 *
 * A first set of actions aims at getting the drives in an operational state which entails:
 * - establishing the communication with the ULTIMET and the drives connected to it
 * - powering up the drives
 * - performing the homing procedure on each drive.
 *
 * The next set deals with making movements on each axis by first describing how to set the limits and precision
* windows , and then actually specifiying and starting the movement. The movements will be controlled by the
* ULTIMET which will execute them in interpolated mode. In a more elaborate step, one thread will be created
* to monitor the current position of the motors. The monitoring thread will show the current position of each
* motor every 100ms. Another will loop indefinitely waiting for a user input: the space bar will immediately stop
* the movement and set one of the controller’s digital outputs.
 *
 * Finally, it will be shown how to power off the drives.
 *
 * The last portion of the program covers error handling.
 *
 * To run this example without modification, you must have:
 * - two linear motors and two ETEL position loop controllers set and tuned accordingly.
 * - a ULTIMET on a PCI slot of your PC
 * - a TransnET connection between the ULTIMET and the drives.
 * - the drives must have the axis numbers 0 and 1.
 * - the software position limits (KL34 and KL35) must have been set when setting the controllers otherwise
 *    these will be defaulted to zero and nothing will move.
 */

/*** Initialization steps ***/

/*** standard libraries ***/

#include <stdio.h> /* the standard I/O library */
#include <stdlib.h> /* standard general purpose library */
#include <ctype.h> /* standard character conversion library */
#include <process.h>/* standard multithreading library */
#include <math.h> /* standard math interface; used for random numbers here */
#include <conio.h> /* standard library for getch() */

/*** ETEL libraries ***/

#include <dsa40.h> /* EDI high level library version 3.0x */
#include <dmd40.h> /* EDI constants definitions */

extern __stdcall Sleep(int);

/*** prototypes ***/

/*
 * The following function define the threads which will run independently. The exact parameter type of theses
* functions as well as the return value are platform specific. This will be the thread that displays the drive status
* and the current position of the motor.
 */
static void display_thread(void *param);
```

```
/*
 * Synchronisation between main thread and displaying thread:
 * To tell the displaying thread whether to fetch and display the data or not.
 */
int doDisplay = 1; /* 0: don't; 1: do*/

/*
 * The following function is the function which is called back when an asynchronous function call terminates.
 */
static void callback(DSA_DEVICE_GROUP *grp, int err, void *param);
/*
 * Returned error by callback of asynchronous call. 0 if no error
 */
int asyncFunctionError;

/*
 * Synchronisation between main thread and asynchronous call function
 * True when callback has been called
 */
int callbackCalled;

/*
 *** Main entry point function.***
 */
int main(int argc, char *argv[])
{
  /*** Initialization steps ***/

    /*
     * We are going to be making some loops, so we need a variable to count them.
     */
    int loops=0;
    /*
     * Most functions of the library return an error code or 0 if no error occurred. We need a variable to store the
     * last error. The error codes are negative values, ranging from -399 to -300 for the dsa30 library.
     */
    int err;            /* initialized to a value that is NOT*/
                        /* returned by the library functions    */
    /*
     * We need two variables to hold the minimum and maximum position of the motor. We are going to make
     * movements within these bounds. All parameters which represent physical dimensions are usually defined
      * in ISO units and use double precision floating point variables.
     */
    double pos_min[2], pos_max[2];
    /*
    * Moving distances will also be expressed in relation to the total range of motion defined by the above limits.
     */
    double range_of_motion[2];

  /*** Creating the objects ***/

    /*
     * We must also define a pointer to a DSA_DRIVE object for each drive. This 'hidden' object is defined in the
* dsa30 library. The client does not need to access the members of this object directly, but need to pass it
* to various dsa library functions. It can be compared to the FILE structure defined in the standard I/O library.
   * The pointers *must* be initialized to NULL before calling the created drive function, otherwise this function
* will fail.
     */
    DSA_DRIVE *axisX = NULL;
    DSA_DRIVE *axisY = NULL;
```

```
/*
 * For programming purposes we define here the constants for the axis numbers
 */
#define AXIS_NB_X 0
#define AXIS_NB_Y 1

/*
 * The same goes for the object representing the ULTIMET
 */
DSA_MASTER *ultimet = NULL;

/* When the ULTIMET manages interpolated movements, the commands must be sent to what is called an
 * interpolation group. This interpolation group can be used like a normal device or drive group. All drives
 * within this group can also be interpolated.
 * So, we must also define a pointer to a DSA_IPOL_GROUP (like a DSA_DRIVE_GROUP) object. This
 * hidden object is also defined in the dsa30 library, and can be considered as a sort of array containing other
 * DSA_DRIVE objects. The client can set, change and retrieve the different drives belonging to this object
 * through some accessor functions. But you might also simply want to create a group (a
 * DSA_DRIVE_GROUP) just to send one single same command to several drives or ULTIMET. See §
 * Groups of the User's Manual for more details on groups. As with DSA_DRIVE, the grp pointer must be
 * initialized to NULL before calling the created group function, otherwise this function will fail.
 */
DSA_IPOL_GROUP *igrp = NULL;

/* Once the pointers have been created, the actual objects themselves have to be created and initialized:
 */

if (err = dsa_create_drive(&axisX)) {
    DSA_DIAG(err, axisX);
    goto _error;
}
if (err = dsa_create_drive(&axisY)) {
    DSA_DIAG(err, axisY);
    goto _error;
}
if (err = dsa_create_master(&ultimet)) {
    DSA_DIAG(err, ultimet);
    goto _error;
}

/* Now, we also have to create the interpolation group object. The size for the group must be given at the
* creation time. In our case, we want to put two drives in this group.
 */
if (err = dsa_create_ipol_group(&igrp, 2)) {
    DSA_DIAG(err, igrp);
    goto _error;
}
/* After creating the group, it has to be filled with the required drives.
 */
if (err = dsa_add_group_item(igrp, axisX)) {
    DSA_DIAG(err, axisX);
    goto _error;
}
if (err = dsa_add_group_item(igrp, axisY)) {
    DSA_DIAG(err, axisY);
    goto _error;
}

/* Now associate the ULTIMET to the interpolation group. When you ask for an interpolated movement on a
* group of axes, the dsa30 library will use the ULTIMET associated with the group.
 */
```

```
    if (err = dsa_set_master(igrp, ultimet)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

  /*** Opening the communication ***/

    /* Now we are ready to open the communication channels.
     * In our example, we have to write the following lines to establish the communication. See § on Establishing
* the communication in the Users's Manual for more details on the syntax involved with opening the
    * communication.
     */

    if (err = dsa_open_u(axisX, "etb:ULTIMET:0")) { /* Motor axis X is on drive number 0   */
        DSA_DIAG(err, axisX);
        goto _error;
    }
    if (err = dsa_open_u(axisY, "etb:ULTIMET:1")) { /* Motor axis Y is on drive number 1   */
        DSA_DIAG(err, axisY);
        goto _error;
    }
    if (err = dsa_open_u(ultimet, "etb:ULTIMET:*")) { /* For a ULTIMET, the node number is "*" */
        DSA_DIAG(err, ultimet);
        goto _error;
    }

  /*** Powering up the motors ***/
    /* Different devices take different times to completely startup which means that in a system that includes
     * different types of devices, such as some position controllers and a position motion controller, some
     *devices will be ready when others are not. This can have the effect of putting in error a device that expects
     * another to be there but that hasn’t finished starting up. This is why, when all devices have finished
* powering up, it is recommended to start the application by a general «reset error» command
     */
    if (err = dsa_reset_error_s(igrp, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    if (err = dsa_reset_error_s(ultimet, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, ultimet);
        goto _error;
    }

/* Now we can send commands to the drive. The first thing to do is to put the drive in power on state. This is
* done by the dsa_power_on_s command. The "_s" at the end of the command indicates that this is a
* synchronous» function. Synchronous function wait until the end of the operation before returning. All
* synchronous functions have a timeout parameter as the last parameter. This parameter orders the function to
* return with a timeout error (DSA_ETIMEOUT) if no response comes from the drive before the end of the
* specified timeout. This lack of response usually indicates an error in the application, or could result from bad
* drive parameters. An appropriate timeout value heavily depends on the application and the command issued.
* In the «power on» case, less than 1 second could be appropriate with pulse initialization, but more than 5
* seconds could be required with constant current initialization.
* Here we can do this in two ways: either switch the power on in each motor individually:
*/
    if (err = dsa_power_on_s(axisX, 10000)) {
          DSA_DIAG(err, axisX);
          goto _error;
    }
    if (err = dsa_power_on_s(axisY, 10000)) {
          DSA_DIAG(err, axisY);
          goto _error;
    }
```

```
     /* or, since we have created a group - primarily for interpolation purposes, but it can also be used any where
     * a group is expected - send the command to the group in general, the DLL taking upon itself to dispatch it
     * to all the members of the group:
     */
    /*
    if (err = dsa_power_on_s(igrp, 10000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }*/

  /*** Homing ***/

   /* Next step in readying the controllers and motors for operation is performing the homing procedure in order
* to find the reference for the motor's absolute position.
     * Once again, this can be done on each axis individually:
     */
    /*
    if (err = dsa_homing_start_s(axisX, 10000)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
    if (err = dsa_homing_start_s(axisY, 10000)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }
    */
    /* or on the group as a whole:
     */
    if (err = dsa_homing_start_s(igrp, 10000)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }

    /*
     * During the indexation procedure, we can read the minimum and maximum position allowed for the motor.
* This is done by calling the function 'get min/max soft position limit'. This function writes the data at the
     * address provided in the second argument. The third argument indicates to the function what kind of value
     * must be returned: the current value, the default value, the maximum or minimum value,... Some of these
     * operations (getting min, max or default values) do not involve any communication.
     * As usual, a timeout must be provided, but for request operation, which are always executed in one cycle
* in the drive, it is recommended to ask for the default timeout. This special timeout value is automatically
     * adjusted to the communication channel used.
     */
          if (err = dsa_get_min_soft_position_limit_s(axisX, &pos_min[AXIS_NB_X], DSA_GET_CURRENT,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
         if (err = dsa_get_max_soft_position_limit_s(axisX, &pos_max[AXIS_NB_X], DSA_GET_CURRENT,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
    range_of_motion[AXIS_NB_X] = pos_max[AXIS_NB_X] - pos_min[AXIS_NB_X];

          if (err = dsa_get_min_soft_position_limit_s(axisY, &pos_min[AXIS_NB_Y], DSA_GET_CURRENT,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }
         if (err = dsa_get_max_soft_position_limit_s(axisY, &pos_max[AXIS_NB_Y], DSA_GET_CURRENT,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }
    range_of_motion[AXIS_NB_Y] = pos_max[AXIS_NB_Y] - pos_min[AXIS_NB_Y];

    /* Now, before we can make other movements, we must wait until it is terminated. To this end, we wait until
* the movement is finished, on both axis:
     */
    if (err = dsa_wait_movement_s(igrp, 60000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

  /*** Monitoring data ***/
    /*
    * We will start the thread once the controllers are homed. The mechanism for starting the threads is platform
* dependant. Notice here a slightly different way to handle the error code. This is because _beginthread is
     * a C library call that does not return known EDI error codes, so if there is a system error creating the thread,
* it is translated into an EDI system error and processed for display.
     */
    if(_beginthread(display_thread, 0, igrp) <= 0) {
        err = DSA_ESYSTEM;
        printf("ERROR at file %s, line %d %s\n", __FILE__, __LINE__, dsa_translate_edi_error(err));
        goto _error;
    }

 /*** Movements ***/

  /*** -- Defining and making simple movements ***/

    /* A position controlled movement can be defined by up to four parameters. The speed, acceleration and jerk
* time describing the movement profile can be specified. The movement is started when setting the target
     * position
     */
    {
        double profileSpeed = 0.5;  /* m/s (or turns/secs) for rotary motors */
        double profileAcc   = 10.; /* m/s^2 or rad/s^2 */
        double jerkTime     = 0.01; /* seconds */

        if (err = dsa_set_profile_velocity_s(axisX, 0, profileSpeed, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, axisX);
            goto _error;
        }

        if (err = dsa_set_profile_acceleration_s(axisX, 0, profileAcc, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, axisX);
            goto _error;
        }

        if (err = dsa_set_jerk_time_s(axisX, 0, jerkTime, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, axisX);
            goto _error;
        }

        /* If the values aren't specified, the values set when tuning the controller are used, (or the default ones if
* they were not changed at that time).
         */
    }

    /* Making motors simply go to a given position can be as straight forward as calling
    * dsa_set_target_position_s() which is the equivalent of the controller's POS command : it sets the value
    * of the target position and starts the movement.
    */

   {
           double targetPositionX = pos_min[AXIS_NB_X] + (rand() * (range_of_motion[AXIS_NB_X])) /
RAND_MAX; /* meters */
           double targetPositionY = pos_min[AXIS_NB_Y] + (rand() * (range_of_motion[AXIS_NB_Y])) /
RAND_MAX; /* meters */

      if (err = dsa_set_target_position_s(axisX, 0, targetPositionX, DSA_DEF_TIMEOUT)) {
         DSA_DIAG(err, axisX);
         goto _error;
      }
      if (err = dsa_set_target_position_s(axisY, 0, targetPositionY, DSA_DEF_TIMEOUT)) {
         DSA_DIAG(err, axisY);
         goto _error;
      }
      /*
      * The above calls will just *start* the movement, which means that the operation will just need one or two
* milliseconds (on the ETEL-Bus-Lite, much less on the TEB) to be executed by the drive, and then control
      * will be handed back immediately to the application. In that case, the timeout value that need to be
      * supplied must only take the communication time from the pc to the drive into account. In this situation,
      * a special value can be used, DSA_DEF_TIMEOUT, which ask the function to use a default timeout value
      * adjusted to the communication channel used.
       * After the command however, we will again wait for the end of the motion. The timeout value has been
* fixed to 1 minute for each of the following movement waiting functions; this obviously depends on the
       * requested move.
       */
      if (err = dsa_wait_movement_s(igrp, 60000)) {
         DSA_DIAG(err, igrp);
         goto _error;
      }
   }
   /* In the above example, the two motors start as soon as they receive the command (index «0» indicated by
* the second parameter) and not synchronously. If we wanted to make the drive start simultaneously, we
    * would send the target position at index 1 in the controllers, for instance , and then indicated for the whole
    * group we have a new setpoint to be executed with dsa_new_setpoint_s():
    */
   {
           double targetPositionX = pos_min[AXIS_NB_X] + (rand() * (range_of_motion[AXIS_NB_X])) /
RAND_MAX; /* meters */
           double targetPositionY = pos_min[AXIS_NB_Y] + (rand() * (range_of_motion[AXIS_NB_Y])) /
RAND_MAX; /* meters */

      if (err = dsa_set_target_position_s(axisX, 1, targetPositionX, DSA_DEF_TIMEOUT)) {
         DSA_DIAG(err, axisX);
         goto _error;
      }
      if (err = dsa_set_target_position_s(axisY, 1, targetPositionY, DSA_DEF_TIMEOUT)) {
         DSA_DIAG(err, axisY);
         goto _error;
      }
      if (err = dsa_new_setpoint_s(igrp, 1, DSA_STA_POS ,DSA_DEF_TIMEOUT)) {
         DSA_DIAG(err, axisX);
         goto _error;
      }
      /* The third parameter tells the controller that it is the position that has changed. One can also changed
      * the speed and accelerations and jerk time  and then ask the controller to take them into account with the
      * dsa_new_setpoint_s() by using the
       * constants DSA_STA_SPD, DSA_STA_ACC, DSA_STA_JRK combined in an OR expression.
```

```
         */
        if (err = dsa_wait_movement_s(igrp, 60000)) {
           DSA_DIAG(err, igrp);
           goto _error;
        }
    }

  /*** Making interpolated movements ***/

  /*** -- Defining the movements ***/
    /*
     * Now we enter in interpolated mode. In this mode the position of the motors is controlled by the multi-axis
* board. This applies only for drives that are specified in the interpolation group (igrp). When you are in this
     * mode, you cannot use the dsa_set_target_position_x() functions on the drives specified in the igrp. You
     * must use the dsa_ipol_xxx() functions instead to define the interpolated path. The interpolator (the
     * ULTIMET) will drive the motors along this path. When you enter into interpolated mode, the actual position
     * becomes the reference postition for the interpolation. For the interpolation functions (dsa_ipol_xxx()), all
     * coordinates are given in absolute from this reference point. Practically, after the call to dsa_ipol_begin_s()
     * you are at the point with coordinates: x = 0.0, y = 0.0
     * When you leave the interpolated mode, you come back to the previous reference system.
    * At present, the X-Y system is in a random absolute (encoder related) position (in meters). For the sake of
* simplicity, we will first move to the center of the motion ranges of both motors, and start interpolation mode
     * from there: it will be our (0,0) reference for interpolated movements.
     */

    /*
     * Move to center of both motor motion ranges
     */
        if (err = dsa_set_target_position_s(axisX, 1, (pos_min[AXIS_NB_X]+pos_max[AXIS_NB_X])/2.0,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
        if (err = dsa_set_target_position_s(axisY, 1, (pos_min[AXIS_NB_Y]+pos_max[AXIS_NB_Y])/2.0,
DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }
    if (err = dsa_new_setpoint_s(igrp, 1, DSA_STA_POS ,DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
    if (err = dsa_wait_movement_s(igrp, 60000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    /*
     * Now start interpolation mode
     */
    if (err = dsa_ipol_begin_s(igrp, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

   /* In this mode, it is the maximum tangential speed (in m/s) and accelerations (in m/s^2) that must be defined:
     * Here, the tangential speed is set to 0.05 m/s; and the tangential acceleration and deceleration are set to
* 0.1 m/(s*s).
     */
    if (err = dsa_ipol_tan_velocity_s(igrp, 0.5, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    if (err = dsa_ipol_tan_acceleration_s(igrp, 1.0, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    if (err = dsa_ipol_tan_deceleration_s(igrp, 1.0, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

  /*** Making interpolated movements ***/

    /* At this point we can, for example, start the trajectory with a linear segment. The following function defines
 * and starts the movement from the previous end of segment and stops the system at the end of the
     * specified segment.
     */
    {
        double endPointX = range_of_motion[AXIS_NB_X]/4.0;
        double endPointY = range_of_motion[AXIS_NB_Y]/4.0;

        if (err=dsa_ipol_line_2d_s(igrp,endPointX, endPointY, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        /* After this we can keep on specifying other lines by another call to the same function:
         */
        endPointX = range_of_motion[AXIS_NB_X]/4.0;
        endPointY = -range_of_motion[AXIS_NB_Y]/4.0;
        if (err=dsa_ipol_line_2d_s(igrp,endPointX, endPointY, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        /* or draw a portion of a circle with a call like:*/
        if (err = dsa_ipol_circle_cw_c2d_s(    igrp,
                                     0.0,                        /* arc end X coordinate    */
                                     0.0,                        /* arc end Y coordinate    */
                                     range_of_motion[AXIS_NB_X]/8.0,    /* arc center X coordinate */
                                     -range_of_motion[AXIS_NB_Y]/8.0,/* arc center Y coordinate */
                                     DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        /* which will draw an arc ending at the position given by the 2nd and 3rd parameters, (back to center of
        * reference frame, in our case) centered on the point with the coordinates given by the 4th and 5th
        * parameters (centerX=range_of_motion[AXIS_NB_X]/8.0; centerY=range_of_motion[AXIS_NB_Y]/8.0)
         *(with respect to the point at which the system was when entering the interpolation mode, remember).
         */
    }

    /* Up until now the system will have stopped at the end of each segment. To make a continuous movement,
 * we have to indicate that we want to start concatenation of the segments: BUT YOU ONLY SPECIFY THIS
     * AFTER THE FIRST SEGMENT HAS BEEN STARTED
     */

    /* Now, in this mode all segments are processed at constant speed, without stopping between segments.
     * This means that when the interpolator processes the one segment, it does not decrease the speed at the
     * end of the segment. It jumps to the next segment without speed change. Now, the following calls will
     * generate a movement along a trajectory at a constant speed:
     */
    {
        double endPointX = -range_of_motion[AXIS_NB_X]/8.0;
        double endPointY = range_of_motion[AXIS_NB_Y]/8.0;

        if (err = dsa_ipol_line_2d_s(igrp, endPointX, endPointY, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        /*
         * Now specifiy that next segment has to be concatenated.
         */
        if (err = dsa_ipol_begin_concatenation_s(igrp, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        /*
         * A now the circle and last segment back to the center:
         */
        if (err = dsa_ipol_circle_ccw_c2d_s(igrp,
                                endPointX,              /* arc end X coordinate    */
                                -endPointY,             /* arc end Y coordinate    */
                                -range_of_motion[AXIS_NB_X]/4.0,/* arc center X coordinate */
                                0.0,                    /* arc center Y coordinate */
                                DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
        if (err = dsa_ipol_line_2d_s(igrp, 0.0, 0.0, DSA_DEF_TIMEOUT)) {
            DSA_DIAG(err, igrp);
            goto _error;
        }
    }

   /* Once the concatenated segments section is terminated, you can revert to giving individual segments again,
     * in between which the ULTIMET will drive the system to a stop, and a restart:
     */
    if (err = dsa_ipol_end_concatenation_s(igrp, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

    /** add more interpolated individual segments here **/

    /* When the system no longer needs to perform interpolated movements, the interpolation mode is exited by
 * calling first :
     */
    if (err = dsa_ipol_wait_movement_s(igrp, 60000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    /* to wait for all segments to be finished, and then:*/
    if (err = dsa_ipol_end_s(igrp, DSA_DEF_TIMEOUT)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    /*to leave the interpolation mode and come back to the previous reference system.*/


    /*
     * End of movements: power motors off:
     */
    if (err = dsa_power_off_s(igrp, 10000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

   /*** Reading and writing of registers (generic functions) ***/

    /*
     * Stop display thread while we are going to save and reset, otherwise we will get errors.
     */
    doDisplay = 0;
    /*
    * Here we read a user register (type X) and increment its value, to be saved and read again for the examples
  * purpose in the next two portions of code.
     */
    {
        long val = 0;

        if (err = dsa_get_register_s(axisX,                /* grp: destination device */
                            DMD_TYP_USER,        /* typ: DMD constant 1:X user register */
                            2,                   /* idx: register index */
                            0,                   /* sidx: register subindex */
                            &val,                /* value: register value */
                            DSA_GET_CURRENT,     /* kind: actual device value */
                            DSA_DEF_TIMEOUT      /* timeout: default timeout */
                           )) {
            DSA_DIAG(err, axisX);
            goto _error;
        }
        printf("\nUser register read from controller X2 = %d\n",val);
        val++;

        if (err = dsa_set_register_s(axisX,                /* grp: destination device */
                            DMD_TYP_USER,        /* typ: DMD constant 1:X user register */
                            2,                   /* idx: register index */
                            0,                   /* sidx: register subindex */
                            val,                /* value: register value */
                            DSA_DEF_TIMEOUT      /* timeout: default timeout */
                           )) {
            DSA_DIAG(err, axisX);
            goto _error;
        }
        printf("\nUser register before sav/reset X2 = %d\n",val);
    }

   /*** Sending commands : Saving and resetting ***/

   /* The values of parameters that have been only modified are not stored in permanent storage on the devices
  * which means that when they are reset the parameters revert back to the stored values. To effectively store
     * modified values, you have to explicitly save them by calling dsa_save_parameters_s() which is the
     * equivalent of the SAV command a the controller. Sending a SAV command to a device will cause it to
     * «disappear» on the ETEL device network and to re-appear in error. So it is possible that the device itself
     * does not have time to acknowledge the SAV command. So it is advised to not check the error code
     * returned by this function...
     */
     dsa_save_parameters_s(axisX,DSA_PARAM_SAVE_X_PARAMS,10000);

    /* The SAV command takes quite a while to execute hence the large timeout value for the last parameter.
     */

    /* Sending a SAV command to a device will cause it to «disappear» on the ETEL device network and to
     * re-appear in error. So before you can issue any other commands, you must wait (quite some time, as the
     * last parameter of dsa_wait_status_equal_s() shows) until the drives are present again:
```

```
 */
{
    /* setup status masks */
    DSA_STATUS status_checkbits = {sizeof(DSA_STATUS)};
    DSA_STATUS status_checkstates = {sizeof(DSA_STATUS)};
    status_checkbits.drive.present = 1;
    status_checkstates.drive.present = 1;

    /* wait that the controllers are present */
    if (err = dsa_wait_status_equal_s(axisX, &status_checkbits,
                                  &status_checkstates, NULL, 30000)){
        DSA_DIAG(err, axisX);
        goto _error;;
    }
}
/* For the new values to be effectively taken into account the drive has to be reset.
 * There is no specific function for the RSD (reset) command so you have to send it «manually»:
 */
dsa_execute_command_d_s(igrp,                          /* grp: destination device */
                        DMD_CMD_RESET_DRIVE,       /* cmd: constant number of the RSD command (88) */
                        DMD_TYP_IMMEDIATE,           /* typ: the parameter to the command is a value*/
                        255,                       /* par : the value always passed to the RSD command*/
                        0,                         /* fast: 1 : true = fast command */
                        0,                          /* ereport: 0 false = don't report drive errors */
                        DSA_DEF_TIMEOUT               /* timeout: 10 secs */
                       );
/* The RSD command also takes quite a while to execute.
 * Also, like SAV command, because the controller resets, it does not have the time to send the
* acknowledgement that it has received and executed this command. This is why the call to
* dsa_execute_command_d_s() for the RSD command nearly always returns an error indicating an absence
* of command acknowledge. This is the one case where it is advised NOT to test the returned error code
* and why the above call is a bit different from the previous ones. Also this is why you don't have to put a big
* timeout value. However, you do have to wait a while for the system to come up again:
 */
 Sleep(60000);

/* Note: here we have reset all the drives in the network. Only the drive on which the SAV was done really
 * needs to be reset. Here it is done because the group includes the ULTIMET and that means that the
 * communication will be lost and the following lines example shows how to reestablish it. The proper way to
 * do this, is to first close the communication with all devices, and then re-open it:

/* --Close communication.
 * You don't need to destroy the objects; these will be used again when reopening the communication
 */
dsa_close(axisX);
dsa_close(axisY);
dsa_close(ultimet);

/* --Reopen communication*/
if (err = dsa_open_u(axisX, "etb:ULTIMET:0")) {
    DSA_DIAG(err, axisX);
    goto _error;
}
if (err = dsa_open_u(axisY, "etb:ULTIMET:1")) {
    DSA_DIAG(err, axisY);
    goto _error;
}
if (err = dsa_open_u(ultimet, "etb:ULTIMET:*")) {
    DSA_DIAG(err, ultimet);
    goto _error;
}
```

```
/*** Reading and writing of registers ***/

  /* We will do this here just to show that we have
   *   a) regained access to the drive
   *   b) actually changed and saved the register
   */
  {
      long val;

      if (err = dsa_get_register_s(axisX,                /* grp: destination device */
                                   DMD_TYP_USER,        /* typ: DMD constant 1:X user register */
                                   2,                   /* idx: register index */
                                   0,                   /* sidx: register subindex */
                                   &val,                /* value: register value */
                                   DSA_GET_CURRENT,     /* kind: actual device value */
                                   DSA_DEF_TIMEOUT      /* timeout: default timeout */
                                  )) {
          DSA_DIAG(err, axisX);
          goto _error;
      }

      printf("User register after sav/reset X2 = %d\n",val);
  }


/*** Asynchronous function ***/

  /* Reset errors before power on */
  if (err = dsa_reset_error_s(igrp, DSA_DEF_TIMEOUT)) {
      DSA_DIAG(err, igrp);
      goto _error;
  }
  if (err = dsa_reset_error_s(ultimet, DSA_DEF_TIMEOUT)) {
      DSA_DIAG(err, ultimet);
      goto _error;
  }

  /* Do an asynchronous power on Handler will be called once power on function is finished. Meanwhile, the
* application can do something, in this case displaying "*" to indicate power on process in-progress status */

  /* Initialization of main thread - callback function synchronous variable */
  callbackCalled = 0;

  /* asynchronous power on */
  /* This function will return BEFORE the controllers terminate the PWR=1 function */
  /* call function will be called once the controllers has terminated PWR=1 function */
  printf("\nPower on in progress ");
  fflush(stdout);
  if (err = dsa_power_on_a(igrp, callback, (void*)1)) {
      DSA_DIAG(err, igrp);
      goto _error;
  }

  /* Wait that callback function is called */
  while (!callbackCalled) {
      printf("*");
      fflush(stdout);
      Sleep(100);
  }
```

```
    if (asyncFunctionError) {
        DSA_DIAG(asyncFunctionError, igrp);
        goto _error;
    }

   /* Do a homing followed by an asynchronous wait on end of movement. Because even synchronous homing
* comes back immediately, it is no sense to do an asynchronous homing. On the other hand, it is advised
    * to do an asynchronous dsa_wait_movement and meanwhile displaying the homing process in-progress
    * status to do an asynchronous wait Handler will be called once power on function is finished. Meanwhile,
    * the application can do something, in this case displaying "*" */

    /* asynchronous power on */
    /* This function will return BEFORE the controllers terminate the PWR=1 function */
    /* call function will be called once the controllers has terminated PWR=1 function */
    printf("\nHoming in progress ");
    fflush(stdout);
    if (err = dsa_homing_start_s(igrp, 10000)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

    /* Initialisation of main thread - callback function synchronous variable */
    callbackCalled = 0;
    if (err = dsa_wait_movement_a(igrp, callback, (void*)2)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }

    /* Wait that callback function is called */
    while (!callbackCalled) {
        printf("*");
        fflush(stdout);
        Sleep(100);
    }

    if (asyncFunctionError) {
        DSA_DIAG(asyncFunctionError, igrp);
        goto _error;
    }

  /*** Application termination ***/

    /*
     * When we have arrived at the end of our application, we have to go through a couple of termination steps
* for a proper closeout of our program.
     */

    /* Application termination operations take place in the reverse order of those described in at startup which
     * are creating the objects and opening the communication.
     * These two functions assign resources to the operating system. During the creation of an object or a group,
     * memory is assigned. During the opening of the communication, the use of the serial port, the interruption
     * line or other resources specific to the communication bus used are assigned. These resources must be
     * given back to the system as soon as there are not used any more. It is normally done just before leaving
     * the application.
     * The DSA library has the dsa_close() function to close the communication and the dsa_destroy() function
* to destroy an object or a group. The closing must be done before the destruction.
     * So, first, the connections must be closed:
     * -connection to the ULTIMET:
     */
    if (err = dsa_close(ultimet)) {
        DSA_DIAG(err, ultimet);
        goto _error;
    }
     /* - connections to both drives:*/
    if (err = dsa_close(axisX)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
    if (err = dsa_close(axisY)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }
    /* Then, just like the objects were created at the beginning of the application, the memory allocated to them
* must be released to be made available to the system again:
     * -group object:
     */
    if (err = dsa_destroy(&igrp)) {
        DSA_DIAG(err, igrp);
        goto _error;
    }
    /* - ultimet object: */
    if (err = dsa_destroy(&ultimet)) {
        DSA_DIAG(err, ultimet);
        goto _error;
    }
    /* - and the drive objects */
    if (err = dsa_destroy(&axisX)) {
        DSA_DIAG(err, axisX);
        goto _error;
    }
    if (err = dsa_destroy(&axisY)) {
        DSA_DIAG(err, axisY);
        goto _error;
    }

    printf("Application correctly ended. Press a key to continue\n");
    _getch();
    /* All is well that ends well.*/
    return 0;

    /**************************
     * Now the error handler...
     **************************/
_error:
    /*
     * Stop access to the drive while stopping, otherwise we will get errors.
     */
    doDisplay = 0;

    /* Destroy the group if not already done. */
    if(dsa_is_valid_drive_group(igrp)) {
        /* Leave the interpolation mode. */
        if (dsa_is_ipol_in_progress(igrp))
            dsa_ipol_end_s(igrp, DSA_DEF_TIMEOUT);
        dsa_destroy(&igrp);
    }

    /* The same has to be done with the ultimet and the drives. For each drive, if the object pointer is valid, we
* check if we can still communicate with it to stop all movements and then to power it off and finally close
     * the communication. Afterwards, the object itself can be destroyed
     */
```

```
/* axis X */
if(dsa_is_valid_drive(axisX)) {

    /* Is the communication open ? */
    bool open = 0;
    dsa_is_open(axisX, &open);
    if(open) {

          /* Stop movements. */
        dsa_quick_stop_s(axisX, DSA_QS_PROGRAMMED_DEC,
                         DSA_QS_BYPASS | DSA_QS_STOP_SEQUENCE, DSA_DEF_TIMEOUT);

        /* When the motor has stopped, a power off is done. */
        dsa_wait_movement_s(axisX, 60000);
        dsa_power_off_s(axisX, 60000);

        /* Close the connection. */
        dsa_close(axisX);
    }

    /* Finally, release the associated resources to the OS. */
    dsa_destroy(&axisX);
}
/* axis Y */
if(dsa_is_valid_drive(axisY)) {

    /* Is the communication open ? */
    bool open = 0;
    dsa_is_open(axisY, &open);
    if(open) {

         /* Stop movements. */
        dsa_quick_stop_s(axisY, DSA_QS_PROGRAMMED_DEC,
                         DSA_QS_BYPASS | DSA_QS_STOP_SEQUENCE, DSA_DEF_TIMEOUT);

        /* When the motor has stopped, a power off is done. */
        dsa_wait_movement_s(axisY, 60000);
        dsa_power_off_s(axisY, 60000);

        /* Close the connection. */
        dsa_close(axisY);
    }

    /* Finally, release the associated resources to the OS. */
    dsa_destroy(&axisY);
}

/* The ULTIMET also has to be shutdown:*/
if(dsa_is_valid_master(ultimet)) {

    /* Is the drive open ? */
        bool open = 0;
         dsa_is_open(ultimet, &open);
         if (open) {
            /* Close the connection */
            dsa_close(ultimet);
         }
         /* And finally, release all resources to the OS. */
        dsa_destroy(&ultimet);
}
```

```
    printf("Application ended with error. Press a key to continue\n");
    _getch();
    /* Exit indicating an error */
    return -1;
}/* end main() */

/*** Monitoring data ***/

/*
 * Display thread function.
 */
static void display_thread(void *param)
{
    /* the common variables must be defined */
    int counter = 0;
    int err = 0;
    int i;

    /*
     * Casting of parameter to a group of pointer.
     */
    DSA_IPOL_GROUP *grp = (DSA_IPOL_GROUP *)param;

    /* Get individual drive objects. */
    DSA_DRIVE *drv[2];
    DSA_MASTER *ultimet;
    for(i = 0; i < 2; i++) {
        if (err = dsa_get_group_item(grp, i, &drv[i])) {
            DSA_DIAG(err, NULL);
            goto _error;
        }
    }
    if (err = dsa_get_master(grp, &ultimet)) {
        DSA_DIAG(err, NULL);
        goto _error;
    }

    /*
     * Start an infinite loop. This function will be stopped by the OS only when the whole process will end.
     */
    for(;;) {

        /*
         * Each 100 ms, we will display the status and the position of each drive. As usually, the status structure
          * must be initialized before
         * using it.
         */
        DSA_STATUS status[] = {sizeof(DSA_STATUS), sizeof(DSA_STATUS)};
        DSA_STATUS ultimet_status = {sizeof(DSA_STATUS)};
        double pos[2];

        pos[0] = pos[1] = 0.0;

        /* stop display thread */
        if(!doDisplay) {
            break;
        }

        /*
         * Get the drive status. Theses functions don't generate any activity on the communication channel. The
* status of each drive is always kept up-to-date in the process memory, which means that theses function
```

```
         * are very effective and could be often called without degrading performances.
         */
        if (err = dsa_get_status(ultimet, &ultimet_status)) {
            DSA_DIAG(err, ultimet);
            goto _error;
        }
        for(i = 0; i < 2; i++) {
            if (err = dsa_get_status(drv[i], &status[i])) {
                DSA_DIAG(err, drv[i]);
                goto _error;
            }
            /* Get the position of each axis. */
          if (err = dsa_get_position_actual_value_s(drv[i], &pos[i], DSA_GET_CURRENT, DSA_DEF_TIMEOUT))
{
                DSA_DIAG(err, drv[i]);
                goto _error;
            }
        }

        /*
         * We can now print the status string on the display.
         */
        printf("%04d: ", ++counter);
        printf("ULTIMET: %c%c%c, ",
            ultimet_status.master.moving ? 'M' : '-',
            ultimet_status.master.warning ? 'W' : '-',
            ultimet_status.master.error ? 'E' : '-');

        for(i = 0; i < 2; i++) {
            printf("AXIS %d: %c%c%c [%12.4fmm]", i,
                status[i].drive.moving ? 'M' : '-',
                status[i].drive.warning ? 'W' : '-',
                status[i].drive.error ? 'E' : '-',
                pos[i]  * 1000.0
                );
            printf((i == 0) ? ", " : "\r");
        }

        /*
         * We can now stop for 100ms - the following call is platform dependant.
         */
        Sleep(100);
    }

    printf("\ndisplay thread ended\n");
    return;

    /* Error handler. */
_error:

    /* Print the first error that occured. */
    printf("error in display (%d it)\n", counter);
}

/*
 * callback function.
 * This function will be called once asynchronous call is terminated
 */
void callback(DSA_DEVICE_GROUP *grp, int err, void *param)
{
    /* The asynchronous function has terminated with an error */
    if (err) {
        asyncFunctionError = err;
        printf("\n-> callback function called with error %d and parameter %d\n", err, (int)(param));
    }
    /* The asynchronous function has terminated without error */
    else {
        asyncFunctionError = 0;
        printf("\n-> callback function called with parameter %d\n", (int)(param));
    }
    callbackCalled = 1;
}
```

Exemple 20: io1.c

```
/*
 * This sample program show how to set and get io from UltimET.
 */

/*
 * To run this demo without modification, you must have:
 *  - an UltimET PCI board plugged into your PC or
 *  - an UltimET TCP/IP connected to the network
 */
```

Exemple 21: StreamingRegisterTransfer.c

```
 /*
 * This sample program show how to make "stream" transfer between PC and device. Stream transfer allows
* to write or read a large amount of data into/from the device. This type of transfer is mainly used when
 * uploading traces or registers or when downloading firmwares.
 * This sample allows the user to choose the communication bus and the connected devices. No tests are done
* concerning the validity of the registers'typ. For example, if is not possible to write M registers, but this
* example doen no validity check
 */
```

13.5 Description of the other examples in C

Example1 : acquisition\acquisition.c

```
/*
 * This simple demo program shows how to make acquisition, using ETEL EDI library set. This example will do
* an acquisition of one trace on 2 Accuret controllers connected to an UltimET
 */

/*
 * To run this demo without modification, you must have:
 * - 2 linear motor and an AccurET properly configured.
 * - an UltimET plugged in the PC
 * - AccurET and UltimET connected through transnET
 */
```

Example2 : compiler\compiler.c

```
/*
 * This simple demo program shows how to :
 * - compile and download a sequence into an AccurET.
 * - upload a sequence from an AccurET
 */

/*
 * To run this demo without modification, you don't need to have a motor connected. You must have one or
 * many AccurET connected through USB or UltimET. The example allows to choose between several
 * configurations
 */
```

Example3 : compiler\offline_compiler.c

```
/*
 * This simple demo program shows how to :
 * - compile a sequence using an offline compiler.
 * - download the compiled sequence into a device
 */

/*
 * To run this demo without modification, you don't need to have a motor connected. You must have one or
 * many AccurET connected through USB or UltimET. The example allows to choose between several
 * configurations
 */
```

Example4 : generic\sample_1.c

```
 /*
 * This simple demo program show how to make basic drive operations (power on, indexation, movement) on
* a single drive, using ETEL EDI library set.
 * The program will send a power on and an indexation command, then move the motor near the two limits of
* the available range, go to the zero position again and power off the drive.
 */

/*
 * To run this demo without modification, you must have:
 * - a linear motor and an ETEL position loop controller properly configured.
 * - a USB connection between the drive and the PC.
 * - the drive must have the axis number 0.
 * - KL45 must be set to insure that position 0 is inside the valid range.
 * - KL34 and KL35 must be set properly.
 * If KL34 and KL35 aren't set properly, you can modify this code to set pos_min and pos_max manually.
 */
```

Example5 : generic\sample_2.c

```
/* This second sample program extends sample_1.c and shows how to use the library to control multiple axes
* in once. It will move a x/y table to random positions.
 */

/*
 * To run this demo without modification, you must have:
 * - two linear motor with one AccurET controller configured.
 * - a USB connection between the drive and the PC.
 * - the drives must have the axes number 0 and 1.
 * - KL45 must be set to insure that position 0 is inside the valid range.
 * - KL34 and KL35 must be set properly.
 */
```

Example6 : generic\sample_3.c

```
/*
 * This third sample program extends sample_2.c and shows how to use multithreading with ETEL's dlls. In this
* program, we will add a thread to monitor the current position of the drive and a thread for emergency stop.
 * The monitoring thread will show the current position of both axes every 100ms, and the second thread will
 * wait indefinitely for a command from the user: the space key will immediately stop the sequence, which will
* run indefinitely otherwise.
 */

/*
 * To run this demo without modification, you must have:
 * - two linear motor with one AccurET controller configured.
 * - a USB connection between the drive and the PC.
 * - the drives must have the axes number 0 and 1.
 * - KL45 must be set to insure that position 0 is inside the valid range.
 * - KL34 and KL35 must be set properly.
 */
```

Example7 : generic\sample_4.c

```
/*
    C example for upload / download of parameters in a ULTIMET.
    The functions defined in this example are also valid for AccurET drives
    ETEL SA, 30.05.02
*/
```

Example8 : generic\sample_5.c

```
/*
    C example for download firmware into a DRIVE or a ULTIMET
    ETEL SA, 2012
*/
```

Example9 : io\io1.c

```
/*
 * This sample program show how to set and get io from UltimET.
 */

/*
 * To run this demo without modification, you must have:
 * - an UltimET PCI board plugged into your PC or
 * - an UltimET TCP/IP connected to the network
 */
```

Example10 : mapping\mapping1.c

```
/*
 * This sample program show how to :
 *    -    download mapping file into a group of drives
 *   -   upload data stored in a group of device into a mapping file
 *   -   activate mapping of a group of drive
 *   -   deactivate mapping of a group of drive
 *   -   check mapping activity of a group of drive
 */

/*
 * To run this demo without modification, you must have:
 * - a PCI-UltimET board plugged into your PC
 * - one or many AccurET connected to UltimET through TransnET
 * If you want to run this example without UltimET, you must change the URL It is advised to download first
 * mapping files provided with the example before uploading data from drives. There are 2 mapping files
 * provided with the example:
 * - map2D.txt which is a correct mapping file downloadable into 2 AccurET number 0 and 1.
 * - map3D.txt which is a correct mapping file downloadable into 3 AccurET number 0, 1 and 2.
 */
```

Example11 : mapping\scaling1.c

```
/*
 * This sample program show how to
 *   -   download scaling file into a drive
 *   -   activate scaling of a drive
 *   -   deactivate scaling of a drive
 *   -   check scaling activity of a drive
 */
/*
 * To run this demo without modification, you must have:
 * - an AccurET connected to your PC, either by USB, TCP/IP or through an UltimET
 * Just change the URL to fit the connection
 * There is 1 scaling file provided with the example:
 * - scaling.txt which is a correct scaling file downloadable into an AccurET number 0.
 */
```

Example12 : rtv\rtv1.c

```
/*
 * This sample program show how to read real-time value on TransnET without irq synchronisation, this means
* that the software will configure UltimET to put:
 *       M50 : Digital input
 *       ML7 : real position real-time value
 *       MF31 : Real force Iq measured of the AccurET into Real time slot on TransnET.
 * The example will then read these value asynchronously.
 */

/*
 * To run this demo without modification, you must have:
 * - a PCI-UltimET board plugged into your PC
 * - an AccurET device 0 connected to UltimET through TransnET
 * - Eventually a connected and set motor
 */
```

Example13 : rtv\rtv2.c

```
/*
 * This sample program show how to read real-time value on TransnET with irq synchronisation, this means
that the software will configure UltimET to put:
 *       M50 : Digital input
 *       ML7 : real position real-time value
 *       MF31 : Real force Iq measured of the AccurET into Real-time slot on TransnET.
 * The example will then read these value synchronously when DSA_RTV_HANDLER will be called
 */

/*
```

```
 * To run this demo without modification, you must have:
 * - a PCI-UltimET board plugged into your PC
 * - an AccurET device 0 connected to UltimET through TransnET
 * - Eventually a connected and set motor
 */
```

Example14 : rtv\rtv3.c

```
 /*
 * This sample program show how to setup slave-to-slave communication using RTV.
 * - Configure Axis 0 to put ML1 value into a slot
 * - Configure Axis 1 to read slot value into ML450
 * - Move Drive 0 and monitor ML450 of drive 1 during move
 * This is an example. Slave to slave communication can be made without RTV if the two axes are part of the
* same AccurET
 */

/*
 * To run this demo without modification, you must have:
 * - a PCI-UltimET board plugged into your PC
 * - an AccurET axis 0 connected to UltimET through TransnET with a connected and set motor
 * - an AccurET axis 1 connected to UltimET through TransnET
 */
```

Example15 : stream\stream.c

```
/*
 * This sample program show how to make "stream" transfer between PC and device. Stream transfer allows
* to write or read a large amount of data into/from the device. This type of transfer is mainly used when
 * uploading traces or registers or when downloading firmwares.
 * This sample allows the user to choose the communication bus and the connected devices. No tests are done
* concerning the validity of the registers' typ. For example, if is not possible to write M registers, but this
 * example does no validity check
 */
```

Example16 : ultimet\ultimet_1.c

```
/*
 * This simple demo program shows how to make basic interpolated movements on a X-Y axes system using
* ETEL's EDI libraries and the ULTIMET multi-axis board. The program will switch the motors on, send an
 * indexation command, make a simple movement, switch to the interpolation mode and interpolate a G-code
* movement.
 */

/*
 * To run this demo without modification, you must have:
 * - two linear motors with one AccurET controllers configured.
 * - one ULTIMET multi-axis board on the PCI bus, configured like this
 * - a TransnET connection between the drives and the ULTIMET.
 * - the drives must have the axis number 0 and 1.
 * - KL45 must be set to insure that position 0 is inside the valid range.
 * - KL34 and KL35 must be set properly.
 */
```

Example17 : ultimet\ultimet_2.c

```
 /*
 * This is a simple program that use the ULTIMET multi-axis board to perform an interpolated movement on a
* X-Y axes system with PVT commands. PVT means Position - Velocity - Time. A movement is defined by the
 * position of the destination point (in x,y,z and theta coordinates), the velocity at the destination point, and the
* time to do the displacement.
 */

/*
```

```
* To run this demo without modification, you must have:
* - two linear motors with one AccurET controllers configured.
* - one ULTIMET multi-axis board on the PCI bus, configured like this
* - a TransnET connection between the drives and the ULTIMET.
* - the drives must have the axis number 0 and 1.
* - KL45 must be set to insure that position 0 is inside the valid range.
* - KL34 and KL35 must be set properly.
*/
```

14. Service and support

For any inquiry regarding technical, commercial and service information relating to ETEL products, please contact your ETEL representative:

HEADQUARTER / SWITZERLAND	AMERICAS	AUSTRIA (Representative)
ETEL S.A. **Zone industrielle** **CH-2112 Môtiers** **Phone: +41 (0)32 862 01 00** **Fax: +41 (0)32 862 01 01** **E-mail: etel@etel.ch** **http://www.etel.ch**	**ETEL Inc.** 333 E. State Parkway US - Schaumburg IL USA 60173-5337 Phone: +1 847 519 3380 Fax: +1 847 490 0151 E-mail: info@etelusa.com	**I+L ELEKTRONIK GmbH** Vibrütteweg 9 AT-6840 Götzis Phone: +43 55 23 645 42 Fax: +43 55 23 645 424 E-mail: b.hoerburger@iul-elektronik.at
CHINA	**FRANCE**	**GERMANY**
DR. JOHANNES HEIDENHAIN (CHINA) Co., Ltd No. 6, Tian Wei San Jie, Area A, Beijing Tianzhu Airport Industrial Zone Shunyi District, Beijing 101312 Tel. +86 10 8042-0000	**ETEL S.A.** **Carré Haussmann** 4 allée du Trait d'Union FR-77127 Lieusaint Phone: +33 (0)1 60 60 20 92 Fax: +33 (0)1 60 02 26 38 E-mail: etel@etel.fr	**ETEL GmbH** Schillgasse 14 DE-78661 Dietingen Phone: +49 (0)741 17453-0 Fax: +49 (0)741 17453-99 E-mail: etel@etelgmbh.de
GREAT-BRITAIN	**HONG-KONG**	**ISRAEL (Representative)**
HEIDENHAIN (GB) Ltd. 200 London Road, GB - Burgess Hill, West Sussex RH 15 9RD Phone +44 (0)1444 247711 Fax +44 (0)1444 870024 E-mail: sales@heidenhain.co.uk	**HEIDENHAIN Limited** Unit 08-10, 20/F, Apec Plaza, 49 Hoi Yuen Road, Kwun Tong, Kowloon, Hong Kong Tel : +852 2759 1920 Fax : +852 2759 1961 E-mail: sales@heidenhain.com.hk	**MEDITAL COMOTECH Ltd.** 7 Leshem St. - 3rd floor PO Box 7772 - Ramat Siv IL-49170 Petach Tikva Phone: +972 3 923 3323 Fax: +972 3 923 1666 E-mail: comotech@medital.co.il
ITALY	**KOREA**	**SINGAPORE**
ETEL S.A. Piazza della Repubblica 11 IT-28050 Pombia Phone: +39 0321 958 965 Fax: +39 0321 957 651 E-mail: etel@etelsa.it	**HEIDENHAIN KOREA Ltd.** 2F Namsung Plaza (9th Ace Techno Tower), 345-30, Gasan-Dong, Geumcheon-Ku, Seoul, Korea 153-782 Phone + 82 2 2028-7430 Fax +82 2 2028-7431 E-mail: info@heidenhain.co.kr	**ETEL MOTION TECHNOLOGY (SINGAPORE) PTE. LTD.** 51 Ubi Crescent, Singapore 408593 Phone : +65 6742 6672 Fax : +65 6749 3922 E-mail: info@etel.sg
SPAIN (Representative)	**SWITZERLAND**	**TAIWAN**
Farresa Electronica, S.A. C/ Les Corts, 36 bajos ES-08028 Barcelona Phone: +34 93 409 24 93 Fax: +34 93 339 51 17 E-mail: farresa@farresa.es	**ETEL S.A.** Zone industrielle CH-2112 Môtiers Phone: +41 (0)32 862 01 33 Fax: +41 (0)32 862 04 12 E-mail: sales@etelsa.ch	**HEIDENHAIN CO., LTD.** No. 29, 33rd road, Taichung Industrial Park Taichung 40768, Taiwan, R.O.C. Phone : +886 4 2358 8977 Fax : +886 4 2358 8978 E-mail: info@heidenhain.tw
THE NETHERLANDS		
ETEL B.V. Copernicuslaan 34 NL-6716 BM Ede Phone: +31 (0)318 495 200 Fax: +31 (0)318 495 210 E-mail: etel@etelbv.nl		

The technical hotline, based in ETEL's headquarters, can be reached:

- by phone: +41 (0)32 862 01 12
- by fax: +41 (0)32 862 01 01
- by e-mail: support@etel.ch

All marketing and technical documentation as well as firmware and software can be downloaded from ETEL's web site: www.etel.ch. ETEL organizes training courses for customers on request, including theoretical presentations of our products and practical demonstrations at our facilities.