E-LAB

Production Programmer System

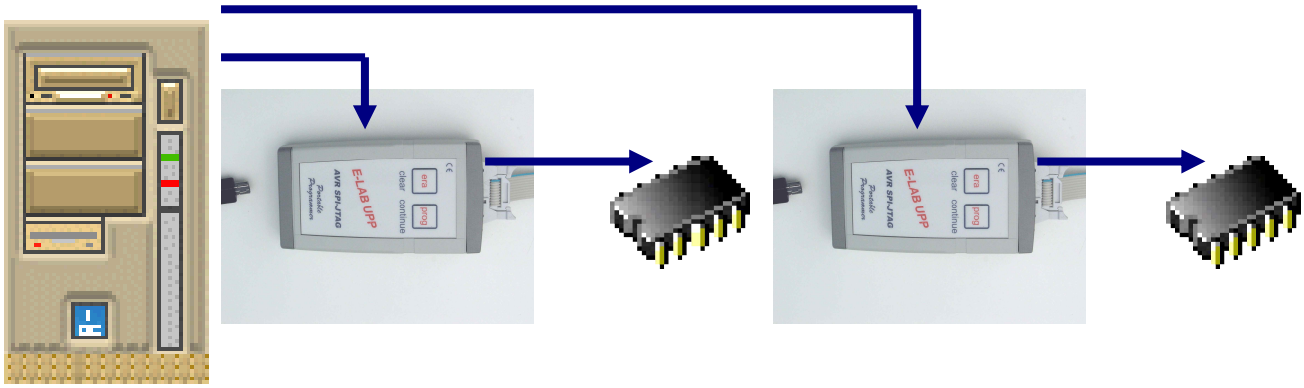
AVR *UPP1X-DLL*

E-LAB Computers D74906 Bad Rappenau www.e-lab.de

- **Production Programming System for Atmel AVR CPUs**
- **Upto 16 simultaneous programmings**
- **Controlled through a WIN32 DLL**
- **Master control program supplied by the user**
- **One USB port used for each programmer**
- **By the use of E-LAB UPP1-X Programmer a flexible and**
- **fast InCircuit programming can be managed**
- **Projects can be packed or encrypted on a PC. The generated projects, upto 16, must be stored onto a SD flashcard.**
- **Simple program change in the production by exchanging the flashcard or by selecting another project stored on the card.**
- **File/project administration also through the DLL with the support functions: List all Files, Delete File, Download File, Check File etc.**

March 2014

In-Circuit programming of Single Chips in mass production



With mass production of electronic boards build with Single-Chips there is often the problem how to program the CPUs within the automatic testing of the board. Common ISP programmers are cheap and often easy to use but they can't be easily integrated into the automatic test environment.

In some cases the program start can be remotely controlled, but at least with analyzing the result it becomes difficult or impossible to pass it to the test equipment. The supplied PC-software for the programmer is firstly intended for manual control. Also if the software interface to the PC is disclosed there is a huge amount of programming to build an own specific driver to operate the programmer.

Todays Single-Chips as the AVR need many fuse- and lockbits which the system must handle. On the other hand in most cases it isn't the job of the production managers to create complexe software mainly if the the production location is far away from the development department.

Nearly insoluble is the job if several boards must be programmed at the same time to increase system throughput or if there are more than one CPU on a board which must be programmed simultaneously. Then the software programming problems reach an unacceptable level.

If the programming system can be controlled totally by a so called DLL (dynamic link library), the software developer of the test system can concentrate on it's main job like generating commands to start programming and analyzing the results.

With the E-LAB Programmer-DLL there is a simple way to integrate the In-Circuit programming into an existing test system. The main software communicates with a few DLL-calls through the DLL with upto 16 programmers connected via USB. The DLL and the programmers do the big part of the job without loading the the test system in an unecessary way.

A production programming system consists of the E-LAB Programmer-DLL and 1 to 16 E-LAB UPP1-X programmers.

The UPP1X DLL for one UPP1-X version is free of charge and can be downloaded from the E-LAB homepage.
The UPP1X DLL for multiple programmers costs €250

E-LAB Computers **D74906 Bad Rappenau Germany**
Tel. 07268/9124-0 **Fax. 07268/9124-24**
WEB: www.e-lab.de **mail: info@e-lab.de**

Production Programmer UPP1X-DLL

The E-LAB programmer DLL serves an interface between a test automat (or PC) and UPP1-X Programmers in the production of boards which are equipped with Atmel AVRs. The DLL can operate upto 16 UPP1-X programmer siumultaneously and therefore it's well suited for heavy duty programming in the mass production field.

The system consists of a Windows DLL (XP, WIN7) and several via USB connected UPP1-X programmers. The host (testsystem or PC) must be USB enabled.

The test computer (application) communicates via the DLL with the programmers. The DLL must be installed in the test computer. By the use of a DLL the control software can be written in any programming language which can handle DLLs. Also DLL-enabled script tools are possible.

The system DLL <> Programmer uses packed or encrypted files which must be build with the help of the E-LAB program AVRprog.exe. These files then contain all necessary informations (Flash file, EEprom file, CPU-type, Lockbits, Fusebits etc). Because of this a manipulation or faulty setup of the programmers by the production staff is absolutely impossible.

Flash card

The UPP1-X programmer contains a FAT16/32 file system and handles flash cards of the **microSD** type. Cards can be read and written. Common infos for file downloading can be found in the UPP1-X manual.

File types

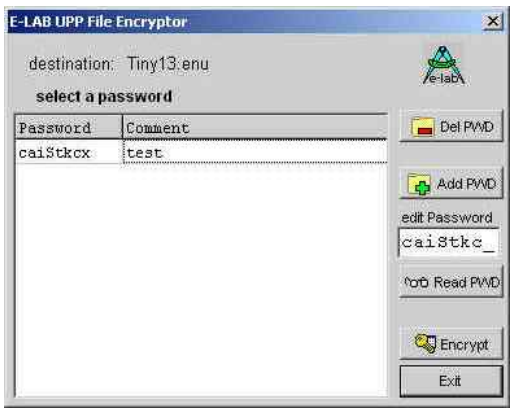
The DLL supports two file types (packed or encrypted). These files must be created by the E-LAB PC-program ***AVRprog.exe***. With the menu items below UPP1-X files can be created. Also the common UPP1-X download dialog can be used for packing or encrypting files.

The content of an encrypted file is scrambled and only readable with a unique key. This key programmed and fixed in the UPP1-X programmer. Each device has its own unique key. So files which are encrypted with this key can only be read by this device.

A packed file has a binary content which includes all necessary informations but ***none*** encryption, passwords or keys. A packed project is processable on each UPP1-X device.

The file extension is always **.pack** or **.pac**

If an additional or new key for the encryption of a UPP1-X project is needed this key/password of a connected UPP1-X programmer can be readout with the menu item above. The user of this programmer then passes this key to the project and file owner. The latter now must add this password/key in the dialog list below.



The password list contains all formerly introduced UPP1-X keys. With ***Del PWD*** an entry can be deleted and with ***Add PWD*** a new key/password can be inserted. This new key either must be supplied by the user of the the programmer with the help of the above dialog or the key of actual connected UPP1-X programmer must be uploaded with the ***Read PWD*** button. In the first case the received key must be manually inserted into the edit field. Iin the second case this is done with the ***ReadPWD*** button.

If the edit field contains the new password/key it must be inserted into the list with the ***Add PWD*** button.

The currently loaded project then can be encrypted with the ***Encrypt*** button. To finish this a list entry (key) must be selected with a click onto it. The selected key then becomes marked with a blue bar. The generated file can directly be stored onto a Flash Card (MMC or SD). Alternatively it can be send to the user via email who copies it then onto his Flash Card. Such an encrypted file is only useable on this programmer where the used key comes from. The decrypting is only done while programming. So the real file contents is never visible.

DLL

The DLL does the big part of the work. It is constructed in a way that the main control program in the test computer (or PC) can be concetrated to essentials : Download of programs/firmware into the connected UPP1-X programmer, programming start, result analyzing. All functions return as the funktion result a string (Pchar) and an integer (res) as an enumeration. The very first access to the DLL must be an ***AbortAll*** function call!

Enumeration of the function results:

resNone, resOk, progDone, noProg, progFound, progBusy, progDefect, progIdle, progComErr, progTimeOutErr, PwrDownErr, PwrSupplyErr, SignatureErr, invPassword, limitExc, eraChip, eraFlash, eraEEp, eraUsrRow, ProtectedErr, NotEmptyErr, prgFlash, prgEEp, prgUsrRow, verifyFlash, verifyEEp, verifyUsrRow, VerifyErr, dwnLoading, ParmErr, dwnLoadErrF, dwnLoadErrE, dwnLoadErrU, invFile, invFName, FileExist, FileNotFound, FileErr, FileDwnErr, MMCok, MMCnoMedia, MMCmissing, MMCinitFailed, MMCresetFailed, MMCcheckFailed, MMCnoProg, MMCprogBusy

For example “resNone“ has the value 0 and “progFound“ the value 4, “ProgBusy” = 5

The **Interface functions** of the DLL are as follows (in Pascal notation):

procedure **AbortAll**;

procedure **GetProgIDs** (var res : integer; resStr : PChar);
// res = count of programmers found

procedure **InitChannel** (Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, ProgDefect, progFound

procedure **DownloadFile**(Channel : Integer; FileName: PChar; var res : integer; resStr : PChar);
// res = noProg, progBusy, invFile, invFName, dwnLoadErr, FileExist, MemError, MMCprotected, resOk

procedure **DeleteAfile**(Channel : Integer; FileName: PChar; var res : integer; resStr : Pchar);
// res = noProg, progBusy, MemError, resOk

procedure **GetFileNames**(Channel : integer; var res : integer; resStr : Pchar);
// res = noProg, progBusy, MemError, resOk

procedure **CheckProgrammer**(Channel: integer; var res : integer; resStr : Pchar);
// res = mmcOk, mmcMissing, mmcInitFailed, mmcResetFailed, mmcCheckFailed, mmcNoProg, mmcBusy

procedure **CheckAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, MemError, invFile, resOk

procedure **OpenAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, notFound, invFile, errPwrDown, errSignature, dwnLoadErr, errProtected, resOk
Note: res is a special enumeration!

procedure **GetProjParams**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, MemError, invFile, dwnLoadErr, resOk

procedure **GetFileState** (Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFName, resOk

procedure **CloseAfile**(Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk

procedure **GetTargVolt**(Channel: integer; var Volt, res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk

procedure **CheckDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = resNone, progBusy, noProg, errPwrDown, errSignature, errNotEmpty, errProtected

```
procedure ProgDevice(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure EraseDevice(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk

procedure EraseDeviceX(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk

procedure EraseDeviceUsr(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk

procedure VerifyDevice(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure ProgFlash(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure ProgEEprom(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure ProgUsrRow(Channel: Integer; var res : integer; resStr : PChar);                  // XMega only
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure ProgFuses(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk

procedure ProgLockBits(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk

procedure VerifyFlashOnly(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure VerifyEEpromOnly(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure VerifyUsrRowOnly(Channel: Integer; var res : integer; resStr : PChar);    // XMega only
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure GetProgStatus(Channel: integer; var res : integer; resStr : Pchar);
// res = resNone, limitExc, eraChip, prgEEp, verifyEEp, prgFlash, verifyFlash, progDone, errPwrDown,
//         errSignature, errProtected, errNotEmpty, errVerify

procedure ClosePort(Channel: Integer; var res : integer; resStr : PChar);
// res = noProg, resNone

procedure ReadBackChipF (Channel : Integer; Block : Pointer; source : Integer; var res : integer;
                                    resStr : PChar);
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//        dwnLoadErrF, resOk

procedure ReadBackChipE(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
                                   resStr : PChar);
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//        dwnLoadErrF, resOk

procedure ReadBackChipU(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
                                    resStr : PChar);   // XMega only
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//        dwnLoadErrE, resOk
```

```
procedure DownProgBlockF(Channel : Integer; Block : Pointer; dest : Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//       resOk

procedure DownProgBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//       resOk

procedure DownProgBlockU(Channel : Integer; Block : Pointer; dest : Integer; var res : integer;
                         resStr : PChar);    // XMega only
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//       resOk

procedure DownOverBlockF(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, errParm, dwnLoadErr, resOk

procedure DownOverBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, errParm, dwnLoadErr, resOk

procedure DownOverBlockU(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);    // XMega only
// res = progBusy, noProg, errParm, dwnLoadErr, resOk

procedure DownFileBlockF(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrF, resOk

procedure DownFileBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrE, resOk

procedure DownFileBlockU(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);   // XMega only
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrF, resOk

procedure UpFileBlockF(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
                       resStr : PChar);
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrF, resOk

procedure UpFileBlockE(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
                       resStr : PChar);
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrE, resOk

procedure UpFileBlockU(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
                       resStr : PChar);   // XMega only
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrE, resOk

procedure GetFuses(Channel : integer; selfuse : char; var fuse : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, memError, resOk

procedure ReleaseTarget(Channel: integer; Var res: integer; resStr: PChar);
// res = progBusy, noProg, resOk
```

Function details

In the following context the term “USBport“ is used for the USB port name of the test computer. The term “Channel“ (0..15) is used for a programmer port of the system and therefore defines a specific UPP1-X programmer connected to the system. With the DLL function „InitChannel“ a specific USBport is mapped to a Channel. With a single channel DLL “Channel” must be preset to 0. “Control System“ and “Host“ represents the test computer where the DLL and the associated program is installed. The very first access to the DLL must be an ***AbortAll*** function call!

Note:
With the single channel DLL “Channel“ must always contain a 0.

Initialization

These are called once at a start up. The first ones are absolutely necessary. A download must be done only if one or more programmers must be feed with new content.

procedure **AbortAll**;

This procedure is necessary if the system “hangs“ or the DLL dropped out of synchronisation with the Host or one or more programmers. After the command “AbortAll“ the DLL is in a basic state and can accept new commands. All USBorts are closed.
A complete re-init with “GetProgIDs “ and “InitChannel “ must be done.

AbortAll must also be called before the control programm is terminated. Also if programmers are plugged or unplugged.

procedure **GetProgIDs** (var res : integer; resStr : PChar);
// res = count of programmers found

This function scans the system for all connected UPP1-X programmers. The parameter “res“ returns the count of programmers found. The parameter “resStr“ contains a corresponding count of UPP1-X names, which are separated by a *CRLF*:
ECK5VMEA<*CR*><*LF*>***ECM22DS8***<*CR*><*LF*>
It can not be guaranteed that the order of the names is always the same. The order is preset by Windows. Witrh most cases these names are not relevant for the the application. But it must know that the first name then is identical with channel–0, the second with channel-1 etc. All operations use channel numbers and not the names. But a relation between individual programmers and logical numbers is given.

procedure **InitChannel** (Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, ProgDefect, progFound

In “Channel“ a number (0..15) must be passed, which then must be used for further transactions with the related programnmer. If this function was successful. “Channel“ now is used and can only be released specific with “ClosePort” or common for all with “AbortAll“. This function must be called once for each connected and used UPP1-X programmer. The function forces a hardware reset in the connected programmer. If a “progDefect” is reported, a firmware download with *AVRprog.exe* must be executed.

procedure **CheckProgrammer**(Channel: integer; var res : integer; resStr : Pchar);
// res = mmcOk, mmcMissing, mmcInitFailed, mmcResetFailed, mmcCheckFailed, mmcNoProg, mmcBusy

If an UPP1-X programmer was found with the function “InitChannel” this device always can be checked with this function. The parameter “Channel“ defines the desired programmer which must be checked.

procedure **ReleaseTarget**(Channel: integer; Var res: integer; resStr: PChar);
// res = progBusy, noProg, resOk
After most of these functions the target CPU stays in the Reset-mode. This function removes this reset state.

procedure **ClosePort**(Channel: Integer; var res : integer; resStr : PChar);
// res = noProg, resNone

This function disconnects a specific UPP1-X programmer and the used USB port becomes closed.

Maintenance functions for files

procedure **GetFileNames** (Channel : integer; var res : integer; resStr : Pchar);
// res = noProg, progBusy, MemError, resOk

This function serves for listing of all projects/files found on the MMC/SD card of the selected UPP1-X programmer. If the result is resOk then "res" contains the count of files and "resStr" is a string which contains all the project names, separated by a <*CR*><*LF*>. Example:
TINY13.PAC<*CR*><*LF*>***M128_1.ENU***<*CR*><*LF*>***TEST1200.PAC***<*CR*><*LF*>
It is the job of the application to extract the names out of this string.

procedure **CheckAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, MemError, invFile, resOk

Diese Funktion veranlasst einen Datei Test im selektierten UPP. Die Datei „FileName" wird dabei auf die Integrität diverser Parameter getestet. Ist das File zwar vorhanden aber der Check fällt durch, ist das Resultat „invFile".

procedure **GetProjParams**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, MemError, invFile, dwnLoadErr, resOk

This function serves for a textual display of the project's programmer parameter. The project/file must exist on the storage media (MMC/SD) of the selected UPP1-X programmer. The parameter „FileName" passes the desired project or filename. If „res = resOk" the string „resStr" then contains 11 substrings, separated through a CRLF. Example:

Test Tiny13<*CR*><*LF*>	Project name
2003.Nov.08 12:15:22<*CR*><*LF*>	Project created
2003.Nov.27<*CR*><*LF*>	Project downloaded
TINY13<*CR*><*LF*>	CPU name
8 MHz<*CR*><*LF*>	CPU clock
996 bytes<*CR*><*LF*>	Flash bytes used
58 bytes<*CR*><*LF*>	EEprom bytes used
00 bytes<*CR*><*LF*>	UserRow Bytes used
no<*CR*><*LF*>	Auto Release
JTAG<*CR*><*LF*>	Program Mode SPI, JTAG, TPI, PDI
AES<*CR*><*LF*>	Encrypt Mode none, AES, AES+PWD
UPP1-X 3.30V max 100mA<*CR*><*LF*>	Intern/extern powersupply

procedure **DownloadFile**(Channel : Integer; FileName: PChar; var res : integer; resStr : PChar);
// res = noProg, progBusy, invFile, invFName, dwnLoadErr, FileExist, MemError, MMCprotected, resOk

This function has only to be used if a project file must be downloaded into a particular programmer connected to the Host. Because the E-LAB UPP1-X programmer has complete projects in its flash card this is only necessary with a project change or the project is new or when the UPP1-X reports a corrupted memory. The parameter **FileName** must contain the source path so that the DLL can find the file. The file extension (.pac or .enu) also must be included.

Of course it is possible that each connected programmer gets its own projects. So it is possible to program big boards with different AVRs which also have different firmware to program.

Encrypted files

Encrypted files must be created with the PC program *AVRprog.exe* and either an ISP3-X, an UPP1-X or an UPP2-X must be connected at encryption time. The the file must be downloaded into the programmer without any change. The programmer then decrypts this file at runtime, if it is used. The used key (on the PC) must be the same as this one which is stored in the programmer, otherwise the error „invFile" is returned if the file is opened for processing.

Packed files

Packed files must be created with the PC program *AVRprog.exe* and must be downloaded into the programmer without any change. The programmer then unpacks this file at runtime, if it is used.

procedure **DeleteAfile**(Channel : Integer; FileName: PChar; var res : integer; resStr : Pchar);
// res = noProg, progBusy, MemError, resOk

This removes a file/project from the flashcard of the programmer. The FileName must not contain any path or drive informations. If the card is write protected the delete is denied and a „errProtected“ is returned. The same is true with files which are directly written onto the card by a CardWriter. These FAT32 files can not be deleted with the FAT16 mode which is used in the UPP.

Production

procedure **OpenAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : PChar);
// res = progBusy, noProg, notFound, invFile, errPwrDown, errSignature, dwnLoadErr, errProtected, resOk

This function open and loads a file/project from the flashcard of the programmer. The parameter FileName must not contain any drive or path informations. With analyzing the file extension (.pac or .enu) the UPP1-X can find out whether this file is packed or encrypted. If it is an encrypted type then the UPP1-X internal key is checked against file internal one. If the compare fails the operation becomes aborted and the result „invFile“ is returned. A packed file is also checked for integrity and returns an “invFile“ if the check fails.

This function must be called once so that the UPP1-X is able to handle any operation concerning the target CPU. Nearly all the functions below expect that a file is already opened.

While a file/project is open none of the above file maintenance functions must be called. If this is necessary a possible opened file must be closed first with “CloseAfile“. After processing a maintenance function it is mandatory to use “OpenAfile“ in order to initialize a working project.

If succesful, all control parameters are loaded and some operations and checks are started:

1. If the UPP1-X must supply the target (DUT) this voltage is generated and checked for the correct value. In case of an error the result “errPwrDown“ is returned. If the target supplies itself the target voltage is measured and in case of invalid values also an “errPwrDown“ is returned.
2. The first access to the target CPU is reading the chip-internal ID number. If this is not the same as given in the project’s parameters the funvtion aborts with an “errSignature“.
3. If the CPU is protected by its Lockbits the result “errProtected“ is returned. This result can be ignored if the CPU must be completely reprogrammed with “ProgDevice”, because this command implicitely includes an “EraseDevice” which then includes a Lockbits erase.

procedure **GetFileState** (Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFName, resOk

This function checks whether a file /project is opened or not. If no project is opened an “invFName“ is returned. If a project is open a “resOk“ is returned.

procedure **CloseAfile**(Channel : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk

This function close an eventually open file.

procedure **ProgDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk

This function starts one programming cycle. If the result “res“ = resOk the operation started successful. At first the device is erased. Then then the Flash, EEprom, Fusebits and Lockbits are programmed. Which parts of the target (DUT) and how they have to be programmed must be defined by the dialog “Programmer Options“ in the program *AVRprog.exe* before the pack or encryption file is created.

This function is the main operation of the DLL. Basically it is sufficient for all rquirements. All further following functions are support types and are only necessary for special purposes.

If all relevant UPP1-X programmer have been startet the application must continously call the function “GetProgStatus“ which polls all active ports to get the state of all startet UPPs.

Enhanced programming

procedure **EraseDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
This function starts one chip erase cycle. If the result "res" = resOk the operation is successful.

procedure **EraseDeviceX**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
For XMegas only. With EraseDevice the UserRow is not deleted with XMegas. With this function also UserRow becomes erased.

procedure **EraseDeviceUsr**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
For XMegas only. Only the UserRow becomes erased.

procedure **ProgFlash**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk
This function starts the separate programming of the Flash. If the result "res" = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function "GetProgStatus" which polls all active ports to get the state of all startet UPPs.
Not applicable with encrypted files!

procedure **ProgEEprom**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk
This function starts the separate programming of the EEprom. If the result "res" = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function "GetProgStatus" which polls all active ports to get the state of all startet UPPs.
Not applicable with encrypted files!

procedure **ProgUsrRow**(Channel: Integer; var res : integer; resStr : PChar); // XMega only
// res = progBusy, noProg, resOk rest of states must be done with "GetProgStatus"
This function starts the separate programming of the UserRow. No erase is done. If the result "res" = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function "GetProgStatus" which polls all active ports to get the state of all startet UPPs.
Not applicable with encrypted files!

procedure **ProgFuses**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk
This function starts the separate programming of the Fusebits. If the result "res" = resOk the operation was successful.
Not applicable with encrypted files!

procedure **ProgLockBits**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk
This function starts the separate programming of the Lockbits. If the result "res" = resOk the operation was successful.
Not applicable with encrypted files!

procedure **VerifyDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
This function starts a chip verify cycle. If the result "res" = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function "GetProgStatus" which polls all active ports to get the state of all startet UPPs. This is only a support function which is not necessary after programming a chip because a programming cycle always includes a verify. It's not applicable if lockbits are activated in the chip.

procedure **VerifyFlashOnly**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
This function starts a flash only verify cycle. If the result “res“ = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function “GetProgStatus“ which polls all active ports to get the state of all startet UPPs. This is only a support function which is not necessary after programming a chip because a programming cycle always includes a verify. It’s only applicable if no lockbits were set.

procedure **VerifyEEpromOnly**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
This function starts a EEprom only verify cycle. If the result “res“ = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function “GetProgStatus“ which polls all active ports to get the state of all startet UPPs. This is only a support function which is not necessary after programming a chip because a programming cycle always includes a verify. It’s only applicable if no lockbits were set.

procedure **VerifyUsrRowOnly**(Channel: Integer; var res : integer; resStr : Pchar); // XMega only
// res = progBusy, noProg, resOk
This function starts a UserRow only verify cycle. If the result “res“ = resOk the operation started successful. If all relevant UPP1-X programmer have been startet the application must continously call the function “GetProgStatus“ which polls all active ports to get the state of all startet UPPs. This is only a support function which is not necessary after programming a chip because a programming cycle always includes a verify. It’s only applicable if no lockbits were set.

Support functions

procedure **GetProgStatus**(Channel: integer; var res : integer; resStr : Pchar);
// res = resNone, limitExc, eraChip, prgEEp, verifyEEp, prgFlash, verifyFlash, progDone, errPwrDown,
// errSignature, errProtected, errNotEmpty, errVerify

This function is the main working function. After sending the program start command "ProgDevice" the Host must continously poll all started programmers with the function "GetProgStatus". The DLL itself also continously receives status informations from the started programmers. These informationen are temporarily stored by the DLL. This means that the DLL always presents the latest (newest) state of the programmers.

If these informations are not read by the Host it will be replaced by new incoming infos from the programmers. This is no essential lost of informations because in most cases the last and final state of a programmer is important.

They come in the following order:

1. eraChip : the CPU will be erased
2. prgFlash : the Flash will be programmed
3. prgUsr : the UserRow will be programmed if enabled. // XMega only
4. prgEEp : the EEprom will be programmed if enabled.
5. progDone : all done ok, inclusive verify Flash (and EEprom).

Also the special functions ***ProgFlash, ProgEEprom, VerifyDevice, VeriFyEEpromOnly*** and ***VerifyFlashOnly*** must be contiunued and finished with the function GetProgStatus. Each started programmer must be polled continously. These results can be expected:

1. prgFlash : currently programming the Flash
2. prgEEp : currently programming the EEprom
3. verifyFlash : currently verifying the Flash
4. verifyEEp : currently verifying the EEprom
5. verifyUsr : currently verifying the UserRow // XMega only
6. progDone oder resOk

Because an error can appear at each point the error state will be the last state which can be read. In other words: each active/started programmer must be polled until either a "progDone" or an error code appears. Only then all programmers had finished their jobs and can accept a new one.

procedure **CheckDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = resNone, progBusy, noProg, errPwrDown, errSignature, errNotEmpty, errProtected

This function can be called after a successful programming to check whether the Chip is locked (protected against further read out). But this is not necessary in most cases.

procedure **GetTargVolt**(Channel: integer; var Volt, res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk

If successful this function returns the target (CPU) voltage in the parameter "Volt" in 10mV steps.

procedure **ReadBackChipF**(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar);
procedure **ReadBackChipE**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : PChar);
procedure **ReadBackChipU**(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar); // XMega only
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
// dwnLoadErrE, resOk

With these three functions the actual Flash, EEprom or UserRow content of the Target CPU can be read back, on condition that the project file is not encrypted and the CPU is not protected.
Block points to a 256byte sized memory block in the application. ***Source*** is the readback address in the chip, where "source" must always point to a 256byte boundary adddress. For example 0, 256, 512, etc. With the Flash and UserRow always 256 bytes are transfered. For example if "source" is 1024 then 256 bytes are uploaded beginning with the memory location 1024.

Not applicable with encrypted files!

Chip and File Manipulation

The DLL supports several ways to manipulate or change parts of a project/file in an UPP. Similar is true for the Flash, EEprom and UserRow area of the connected CPU.

If the concerning project file is encrypted a manipulation of the file or the chip is disabled.
Chips with activated LockBits of course can not be manipulated.

```
procedure DownProgBlockF(Channel : Integer; Block : Pointer; dest : Integer; var res : integer; resStr : PChar);
procedure DownProgBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
procedure DownProgBlockU(Channel : Integer; Block : Pointer; dest : Integer; var res : integer;
                         resStr : PChar);   // XMega only
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//       resOk
```

The purpose of thse functions is to provide an online manipulation of the actual Flash or EEprom content of the connected chip. In case of the flash a reprogramming only makes sense in areas where the relevant memory locations contain $FF, which means unprogrammed bytes.
For technical reasons and also to be able to handle all CPU types always a 256 byte Flash/UserRow block must be used. In addition this block must start on a 256 byte boundary. But this is not really a problem.
There are several ways to build such a ***Flash*** block.

1. The application fills a memory block of 256 bytes with $FF. Then overwrite the desired locations with the desired values. Then call the function DownProgBlockF
2. Load the memory block of 256 bytes with the origin content out of the project file with "UpFileBlockF". Then overwrite the desired locations with the desired values. Then call the function DownProgBlockF.
3. Load the memory block of 256 bytes with the origin content out of the Flash of the CPU with "ReadBackChipF". Then overwrite the desired locations with the desired values. Then call the function DownProgBlockF.

If a $FF filled block is used then only these memory location become changed which content now are <> $FF. If the origin file or the chip content is used all memory location become reprogrammed but only the altereated are changed. Attention! It must be clear that in all cases bits can only be programmed to zero "0" but never to "1".

If the ***EEprom*** content must be changed there are no limitations because EEprom cells always can be changed and overwritten with any values. But consider that there are also pages. So the destination address must always be a multiple of the EEprom page size (8..32, dependend of the AVR type).

The parameter **Block** is a pointer which must point to a 256 byte sized source buffer in the application.
Dest is the target address of a 256 byte block in the Flash, EEprom or UserRow memory of the CPU. It always must start at a 256 byte page (boundary).

The CPU must be already programmed (Flash and Fuses) and must not be protected. The concerning project file must not be encrypted.

Not applicable with encrypted files!

procedure **DownFileBlockF**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar);
procedure **DownFileBlockE**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar);
procedure **DownFileBlockU**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar);
procedure **UpFileBlockF**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : PChar);
procedure **UpFileBlockE**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : PChar);
procedure **UpFileBlockU**(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar); // XMega only
// res = progBusy, noProg, MemError, progProtected, dwnLoadErrEF, dwnLoadErrE, resOk

These four functions support the upload and the manipulation of the actual Flash and EEprom content of a project file. So it is possible to change parts of the firmware or EEprom part of a file. The block size "count" should not exceed 256 bytes. The application is responsible for the correct blocksize and the target address "dest" or source address "source". A check is not done. These functions return an error if the file is encrypted. An enlargement of the Flash, EEprom or UserRow part of the file must not happen otherwise the file becomes corrupted!

Not applicable with encrypted files!

procedure **GetFuses**(Channel : integer; selfuse : char; var fuse : integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, memErr, resOk

This function reads back the LockBits, FuseBits or Calibration bytes out of the connected CPU. **SelFuse** selects the desired fuse. After the excution **Fuse** returns the value found. If successful **resStr** contains this value as a string. The char parameter **SelFuse** can consist of these values:

L	= LockBits
F	= Fuse0
H	= Fuse1
E	= Fuse2
I	= Fuse3
J	= Fuse4
K	= Fuse5
0	= OscCal byte 0
1	= OscCal byte 1
2	= OscCal byte 2
3	= OscCal byte 2

Akthough this function supports all fuse types (L, F, H, E, I, J, K, 0, 1, 2, 3) it makes no sense to upload fuses which are not present in the actual CPU.

Serial Number Support

The chip manipulations described above like “DownProgBlockF“ can only be used after the programming of the Fusebits, the Flash and the EEprom and before programming the LockBits. Because of this a chip programming must be done step by step controlled by the application. It must be clear that this lasts longer as the “ProgDevice“ function which does all the job in a single action. Furthermore the Flash bits can only be programmed to “0“ but never to “1“. If the project file is encrypted the above functions can not be used.

And here is the purpose of the following functions. The data downloaded by these functions is used within the “ProgDevice“ function to replace the origin data loaded from the file. In order to avoid “trojan horses“ the memory areas and the download count is limited to 32bytes.

```
procedure DownOverBlockF(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, errParm, dwnLoadErr, resOk

procedure DownOverBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);
// res = progBusy, noProg, errParm, dwnLoadErr, resOk

procedure DownOverBlockU(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
                         resStr : PChar);    // XMega only
// res = progBusy, noProg, errParm, dwnLoadErr, resOk
```

The parameter ***Block*** must point to the source in the application. ***Dest*** defines the target address of this block in the Flash or EEprom memory of the chip. ***Count*** defines the block size (max. 32). The block must not cross a 256 byte boundary. Invalid values are Dest = 250, count = 20. The block crosses a 256 byte boundary.

The functions load a block with upto 32bytes into the UPP1-X programmer. The standard programmer function ***ProgDevice*** then replaces the original file data with this block if the address Dest is reached. This is true for both the Flash and the Eeprom if a corresponding block exists. It is also possible to make a new download after each succesful programming operation. So the application can build a unique serial number for each device.

The download stays valid and active until a ***OpenAfile*** is executed which deletes the blocks.
So the standard procedure is:

1. open the desired file/project
2. download the Flash or EEprom block
3. program the device
4. replace the device by a new one
5. continue with item 2 or item 1

Examples and sources

This system contains the WIN32 DLL ***UPP1X_DLL.dll***, a comprehensive Delphi/Pascal program and a Visual Basic and a C++ program. All sources of these programs are also included.

The experienced programmer should have not much difficulties to build an own specialized program which fulfills the particular needs.

Import of the DLL into Delphi

```
type
  tProgResult  =  (resNone, resOk, progDone, noProg, progFound, progBusy, progDefect, progIdle,
                   progComErr, progTimeOutErr,  PwrDownErr, PwrSupplyErr, SignatureErr,
                   invPassword, limitExc, eraChip, eraFlash, eraEEp, eraUsrRow, ProtectedErr, NotEmptyErr,
                   prgFlash, prgEEp, prgUsrRow, verifyFlash, verifyEEp, verifyUsrRow, VerifyErr,
                   dwnLoading, ParmErr, dwnLoadErrF, dwnLoadErrE, dwnLoadErrU, invFile, invFName,
                   FileExist, FileNotFound, FileErr, FileDwnErr, MMCok, MMCnoMedia, MMCmissing,
                   MMCinitFailed, MMCresetFailed, MMCcheckFailed, MMCnoProg, MMCprogBusy);
```

procedure **AbortAll**; stdcall; external ‘UPP1X_DLL.dll';

procedure **GetProgIDs** (var res : integer; resStr : Pchar); external ‘UPP1X_DLL.dll'

procedure **InitChannel** (Channel : integer; var res : integer; resStr : Pchar); external ‘UPP1X_DLL.dll'

procedure **DeleteAfile**(Channel: Integer; FileName: PChar; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **GetFileNames**(Channel : integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **CheckProgrammer**(Channel: integer; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **CheckAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **OpenAfile**(Channel : integer; FileName: PChar; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **GetProjParams**(Channel : integer; FileName: PChar; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **GetFileState**(Channel : integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **CloseAfile**(Channel : integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **DownloadFile**(Channel : Integer; FileName: PChar; var res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';

procedure **GetTargVolt**(Channel: integer; var Volt, res : integer; resStr : Pchar); stdcall;
external ‘UPP1X_DLL.dll';
procedure **ProgDevice**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **EraseDevice**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **EraseDeviceX**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **EraseDeviceUsr**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **ProgFlash**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external ‘UPP1X_DLL.dll';

procedure **ProgEEprom**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **ProgUsrRow**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.DLL';

procedure **ProgFuses**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **ProgLockBits**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **VerifyDevice**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **VerifyFlashOnly**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **VerifyEEpromOnly**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **VerifyUsrRowOnly**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.DLL';

procedure **GetProgStatus**(Channel: integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **ReleaseTarget**(Channel: integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **ClosePort**(Channel: integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **CheckDevice**(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';

procedure **ReadBackChipF**(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.dll';
procedure **ReadBackChipE**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.dll';
procedure **ReadBackChipU**(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.DLL';

procedure **DownProgBlockF**(Channel : Integer; Block : Pointer; dest : Integer; var res : integer; resStr : PChar);
stdcall; external 'UPP1X_DLL.dll';
procedure **DownProgBlockE**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.dll';
procedure **DownProgBlockU**(Channel : Integer; Block : Pointer; dest : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.DLL';

procedure **DownOverBlockF**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.dll';
procedure **DownOverBlockE**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.dll';
procedure **DownOverBlockU**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP1X_DLL.DLL';

procedure **DownFileBlockF**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';
procedure **DownFileBlockE**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';
procedure **DownFileBlockU**(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.DLL';

procedure **UpFileBlockF**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';
procedure **UpFileBlockE**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.dll';
procedure **UpFileBlockU**(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : Pchar); stdcall; external 'UPP1X_DLL.DLL';

procedure **GetFuses**(Channel : integer; SelFuse : char; var Fuse : integer; var res : integer; resStr : PChar);
stdcall; external 'UPP1X_DLL.dll';

Import of the DLL into Visual Basic

VB Interface

```
Enum tprgResult

    resNone = 0
    resOk = 1
    progDone = 2
    noProg = 3
    progFound = 4
    progBusy = 5
    progDefect = 6
    progIdle = 7
    progComErr = 8
    progTimeOutErr = 9
    PwrDownErr = 10
    PwrSupplyErr = 11
    SignatureErr = 12
    invPassword = 13
    limitExc = 14
    eraChip = 15
    eraFlash = 16
    eraEEp = 17
    eraUsrRow = 18
    ProtectedErr = 19
    NotEmptyErr = 20
    prgFlash = 21
    prgEEp = 22
    prgUsrRow = 23
    verifyFlash = 24
    verifyEEp = 25
    verifyUsrRow = 26
    VerifyErr = 27
    dwnLoading = 28
    ParmErr = 29
    dwnLoadErrF = 30
    dwnLoadErrE = 31
    dwnLoadErrU = 32
    invFile = 33
    invFName = 34
    FileExist = 35
    FileNotFound = 36
    FileErr = 37
    FileDwnErr = 38
    MMCok = 39
    MMCprotected = 40
    MMCnoMedia = 41
    MMCmissing = 42
    MMCinitFailed = 43
    MMCresetFailed = 44
    MMCcheckFailed = 45
    MMCnoProg = 46
    MMCprogBusy = 47
    dummy = Int32.MaxValue
End Enum
```

Please note: all integer are 32bit types

```
Public Declare Sub AbortAll Lib "UPP1X_DLL.dll" ()

Public Declare Sub GetProgIDs Lib "UPP1X_DLL.dll" (ByRef result As tprgResult, ByVal resString As String)

Public Declare Sub InitChannel Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub GetFileNames Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub OpenAfile Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByVal FileName As String,
        ByRef result As tprgResult, ByVal resString As String)

Public Declare Sub CloseAfile Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub GetTargVolt Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef Volt As Integer, ByRef
        result As tprgResult, ByVal resString As String)

Public Declare Sub EraseDevice Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub GetProgStatus Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub ProgDevice Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

Public Declare Sub ReleaseTarget Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
        ByVal resString As String)

   Public Declare Sub VerifyDevice Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByRef result As tprgResult,
ByVal resString As String)

   Public Declare Sub GetFuses Lib "UPP1X_DLL.dll" (ByVal channel As Integer, ByVal selfuse As Char, ByRef
fuse As Integer, ByRef result As tprgResult, ByVal resString As String)

……
```

Import of the DLL into C++

```
//////////////////////////////////////////////////////////////////////////
// UPP1-X Programmer DLL Header
//////////////////////////////////////////////////////////////////////////

// function call results

 enum prgResult
     {
       resNone, resOk, progDone, noProg, progFound, progBusy, progDefect, progIdle, progComErr,
       progTimeOutErr, PwrDownErr, PwrSupplyErr, SignatureErr,
       invPassword, limitExc,
       eraChip, eraFlash, eraEEp, eraUsrRow, ProtectedErr, NotEmptyErr,
       prgFlash, prgEEp, prgUsrRow,
       verifyFlash, verifyEEp, verifyUsrRow, VerifyErr,
       dwnLoading, ParmErr, dwnLoadErrF, dwnLoadErrE, dwnLoadErrU,
       invFile, invFName, FileExist, FileNotFound, FileErr, FileDwnErr,
       MMCok, MMCprotected, MMCnoMedia, MMCmissing, MMCinitFailed, MMCresetFailed,
       MMCcheckFailed, MMCnoProg, MMCprogBusy
     };
```

note: all integer are 32bit types

The following list is **not** complete…

```
extern "C"
  {
 void __stdcall AbortAll          ( void );

 void __stdcall InitChannel       ( int channel, int &res, char *resStr );

 void __stdcall GetProgStatus     ( int channel, int &res, char *resStr );

 void __stdcall ClosePort         ( int channel, int &res, char *resStr );

 void __stdcall ReleaseTarget     ( int channel, int &res, char *resStr );

 void __stdcall OpenAfile         ( int channel, char *fileName, int &res, char *resStr );

 void __stdcall GetFileState      ( int channel, int &res, char *resStr );

 void __stdcall CloseAfile        ( int channel, int &res, char *resStr );

 void __stdcall DeleteAfile       ( int channel, char *fileName, int &res, char *resStr );

 void __stdcall GetFileNames      ( int channel, int &res, char *resStr );

 void __stdcall CheckAfile        ( int channel, char *projName, int &res, char *resStr );

 void __stdcall GetProjParams     ( int channel, char *projName, int &res, char *resStr );

 void __stdcall GetProgIDs        ( int &res, char *resStr );

 void __stdcall CheckProgrammer   ( int channel, int &res, char *resStr );

 void __stdcall DownloadFile      ( int channel, char *projname, int &res, char *resStr );

 void __stdcall GetTargVolt       ( int channel, int &volt, int &res, char *resStr );

 void __stdcall CheckDevice       ( int channel, int &res, char *resStr );
```

```
  void __stdcall ProgDevice        ( int channel, int &res, char *resStr );

  void __stdcall EraseDevice       ( int channel, int &res, char *resStr );

  void __stdcall VerifyDevice      ( int channel, int &res, char *resStr );

  void __stdcall GetFuses          ( int channel, char selFuse, int &fuse, int &res, char *resStr );

// ........
   }
```

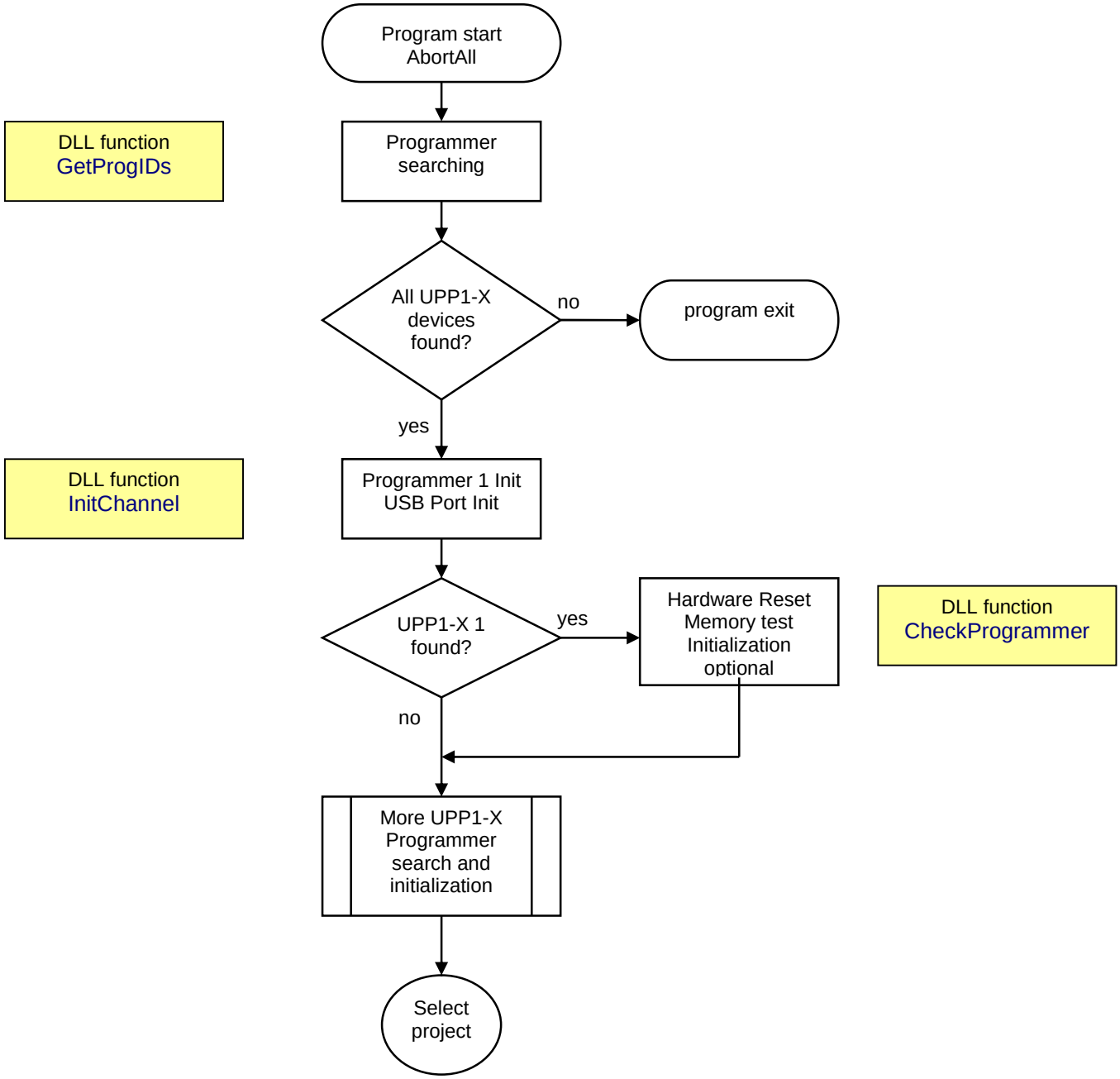
Flow diagrams

For a better view onto the useage of the E-LAB UPP1-X DLL in the production there a three diagrams:
1. Initialization 2. Project select 3. Production.

Initialization

The UPP1-X programmers must be connected to the Host via an USB port. An USB-enabled 32bit Windows must run on this computer. (XP/WIN7/WIN8). The DLL „UPP_DLL.dll“ must reside in the same directory where the application program which controls the DLL and programmers is located.

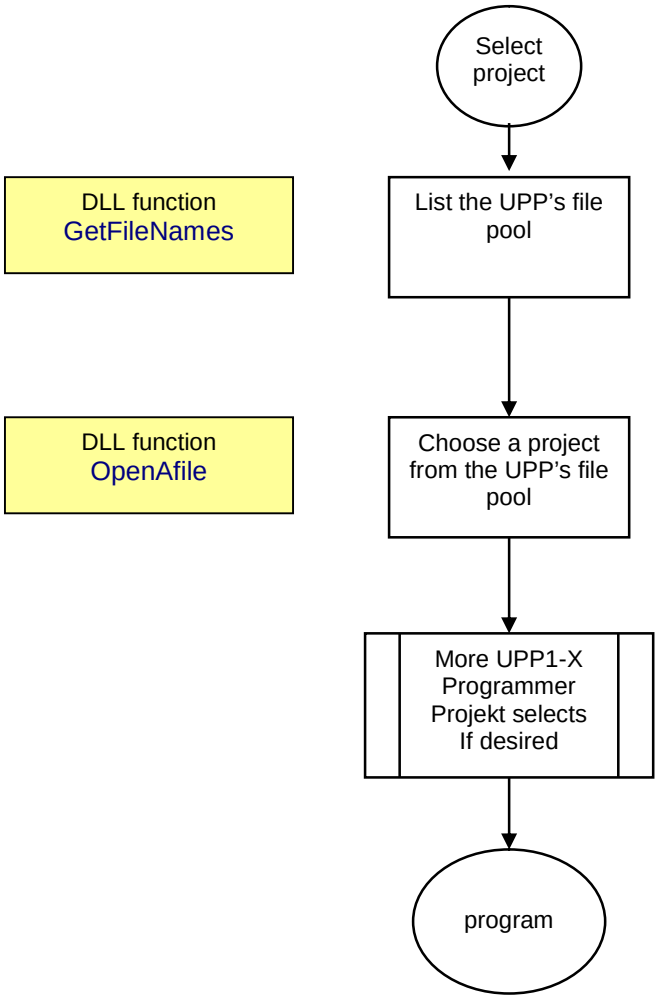
The very first access to the DLL must be an ***AbortAll*** function call!
At start up of the application and AbortAll the connected programmers must be searched and mapped into “Channels”. If a UPP1-X programmer is found via the DLL this UPP1-X gets also a hardware reset. The UPP1-X can respond with a memory error to the application. Now the init sequence is complete.



Select Project

The UPP1-X programmer memory is flash based and retain all once stored project also through a power down cycle.

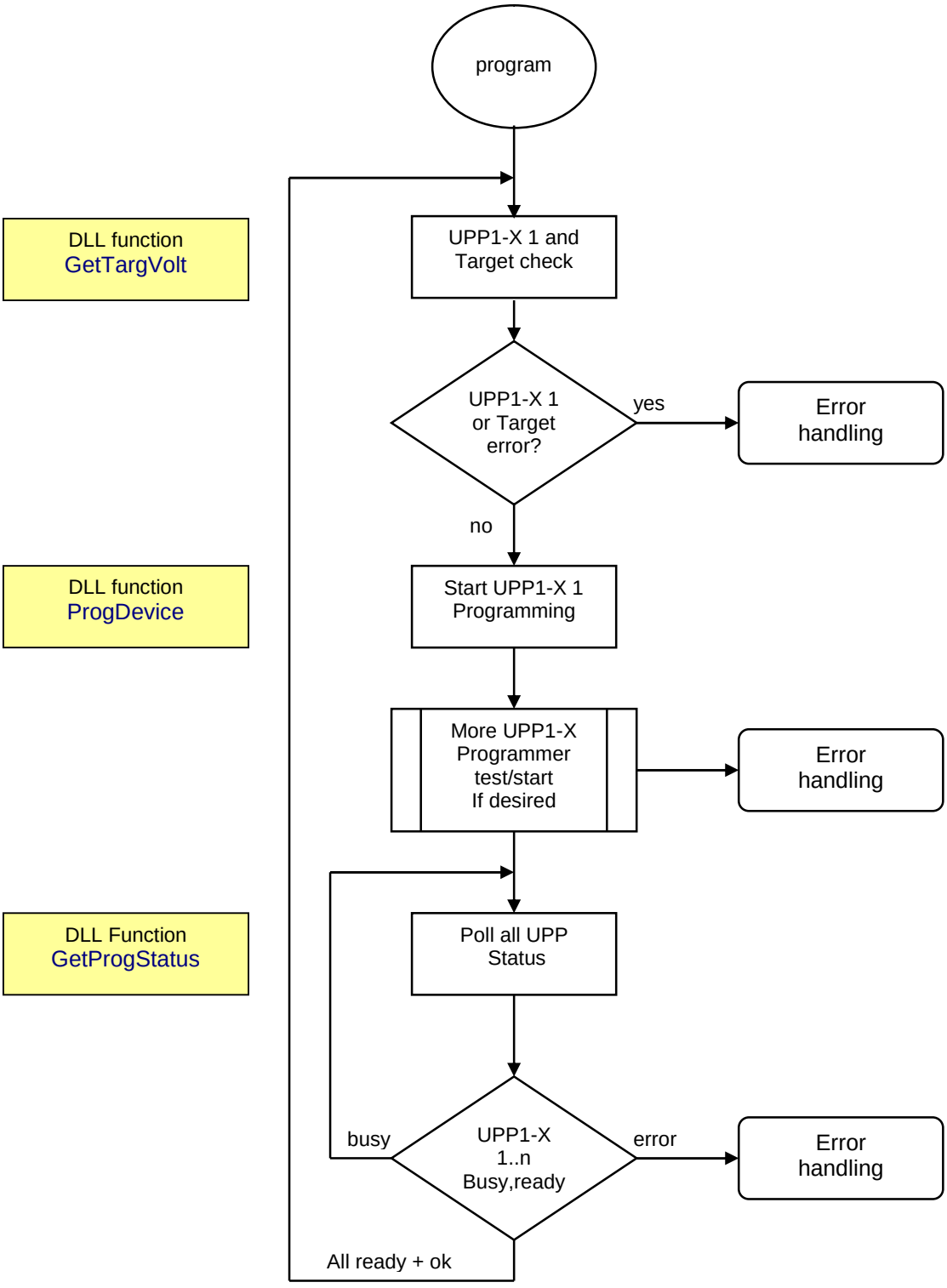
In order to start a programming operation a project must be selected.



Production

The programming cycle must be started separately for each connected UPP1-X programmer. The start function itself results in an OK or FAIL. The reason for a fail can be a disconnected or defective UPP1-X programmer. So it's a good idea to check all involved programmers with the function „GetTargVolt" for presence and idle and also the target CPU for power-good etc.

The main programming state after the start of all programmers must polled by a general poll function. If all active programmers have send either a „progDone" or an error messag, the programming cycle is finished and a new one can be started.



E-LAB Computers D74906 Bad Rappenau Germany
Tel. 07268/9124-0 Fax. 07268/9124-24
WEB: www.e-lab.de mail: info@e-lab.de