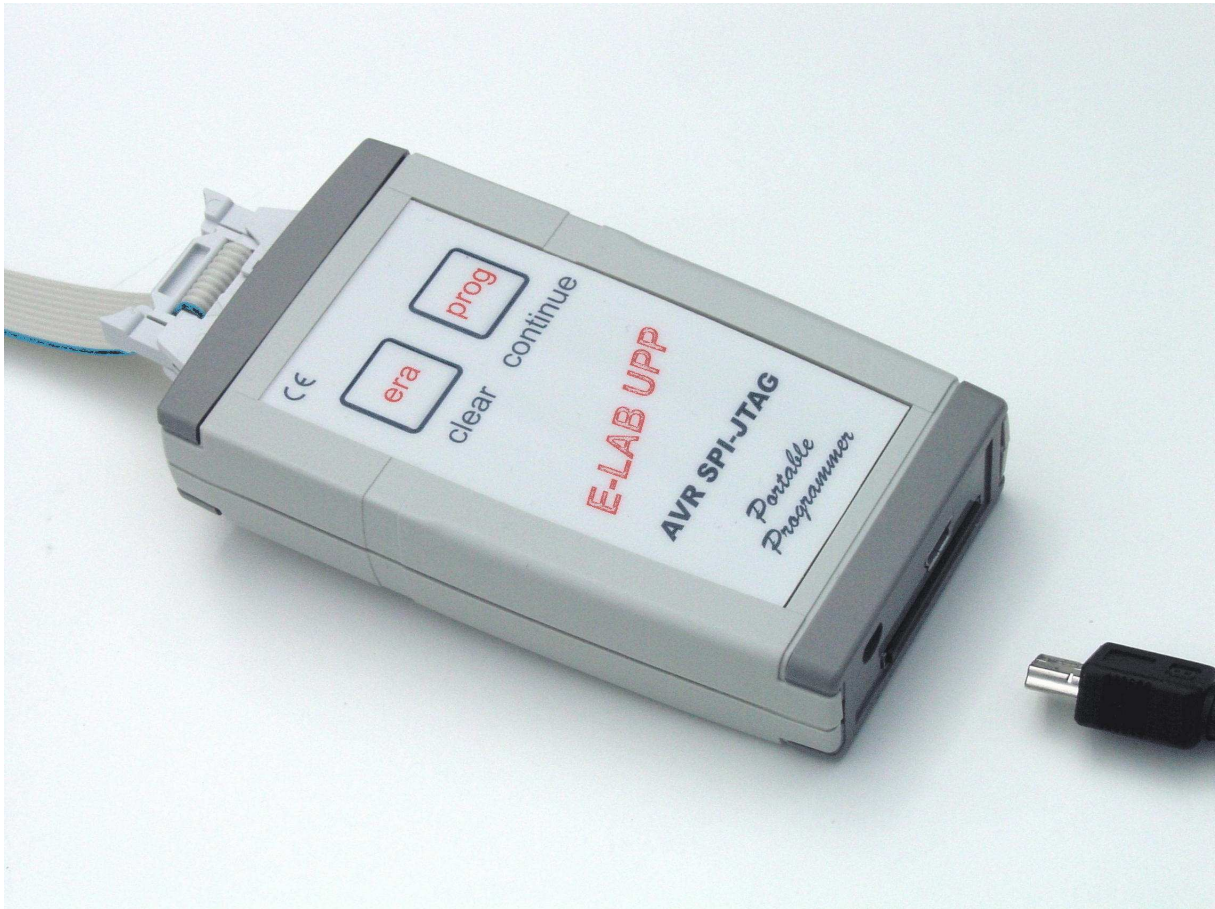


E-LAB

Production Programmer System

AVR UPP-DLL



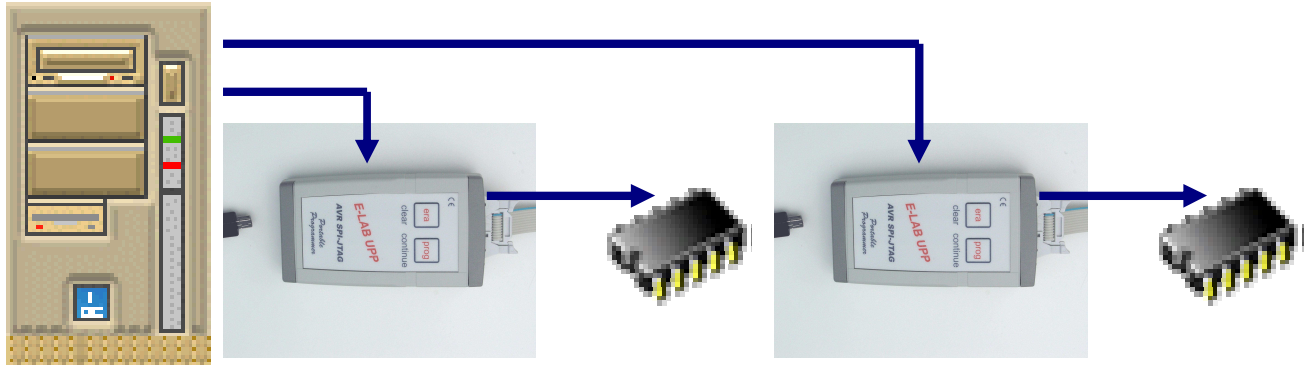


- **Production Programming System for Atmel AVR CPUs**
- **Upto 16 simultaneous programmings**
- **Controlled through a WIN32 DLL**
- **Master control program supplied by the user**
- **One USB port used for each programmer**
- **By the use of E-LAB UPP Programmer a flexible and**
- **fast InCircuit programming can be managed**
- **Projects can be packed or encrypted on a PC. The generated projects, upto 10, must be stored onto a MMC or a SD flashcard. SD cards only with version "S" UPP**
- **Simple program change in the production by exchanging durch the flashcard or by selecting another project stored on the card.**
- **File/project administration also through the DLL with the support funktions: List all Files, Delete File, Download File, Check File.**

November 2003



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 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.

Today's 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 complex software mainly if 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 its 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 up to 16 programmers connected via USB. The DLL and the programmers do the big part of the job without loading the test system in an unnecessary way.

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

The single channel version of the UPP-DLL is for free and can be downloaded from the E-LAB homepage.
The 2-channel version of the UPP-DLL costs euro 100.- (US\$100) no programmers included
The 4-channel version of the UPP-DLL costs euro 200.- (US\$200) no programmers included
The 8-channel version of the UPP-DLL costs euro 400.- (US\$400) no programmers included
The 16-channel version of the UPP-DLL costs euro 600.- (US\$600) no programmers included

E-LAB Computers
Tel. 07268/9124-0
WEB: www.e-lab.de

D74906 Bad Rappenau Germany
Fax. 07268/9124-24
mail: info@e-lab.de



Production Programmer UPP-DLL

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

The system consists of a Windows DLL (WIN98..WIN2000/XP) and several via USB connected UPP 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-HEX file, EEprom -HEX 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 UPP programmer contains a FAT16 file system and handles flash cards of the **MMC** or **SD** type. Cards can be read and written. Common infos for file downloading can be found in the UPP 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 UPP files can be created. Also the common UPP download dialog can be used for packing or encrypting files.



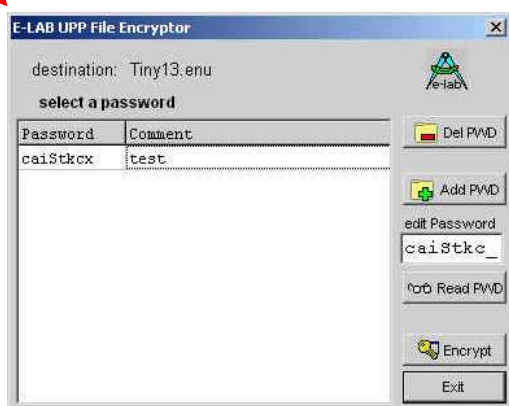
The content of an encrypted file is scrambled and only readable with a unique key. This key programmed and fixed in the UPP programmer. Each device has its own unique key. So files which are encrypted with this key can only be read by this device. The file extension is always **.enu**

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 UPP device.

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



If an additional or new key for the encryption of a UPP project is needed this key/password of a connected UPP 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 UPP 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 UPP programmer must be uploaded with the **Read PWD** button. In the first case the received key must be manually inserted into the edit field. In the second case this is done with the **Read PWD** 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 concentrated to essentials : Download of programs/firmware into the connected UPP-programmer, programming start, result analyzing. All functions return as the funktion result a string (Pchar) and an integer (res) as an enumeration.

Enumeration of the function results:

resNone, noProg, progFound, MemError, progBusy, progDefect, progIdle, progProtected, eraChip, eraEEp, eraFlash, prgEEp, prgFlash, verifyEEp, verifyFlash, errPwrDown, errSignature, errProtected, errNotEmpty, errVerify, progDone, dwnLoading, dwnLoadErrP, dwnLoadErrE, dwnLoadErrF, dwnLoadErr, invFile, invFName, FileExist, notFound, invPassword, limitExc, errProgType, resOk, progNoJTAG, progNoSupply, errPwrSupply, errParm, MMCprotected, MMCnoMedia

For example “resNone“ has the value 0 and “progFound“ the value 2, “ProgBusy” = 4

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

```
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 = progBusy, noProg, progDefect, MMCprotected, MMCnoMedia, progFound
```

```
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
```

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



JTAG Boundary SCAN overview

Current E-LAB programmer of type UPP-USB support also the boundary scan function of the mega AVR CPUs. Boundary scan is completely separated from the programming part in the AVR CPU and also in the UPP programmer and also in this DLL. The only common function for programming and for boundary scan are the DLL functions *GetProgIDs* and *InitChannel*.

With boundary scan a file is not necessary and also no download of any file contents. All informations needed for doing the job is passed by the function *JTAGBoundaryOpen*.

For board testing through boundary scan there are only two functions

1. JTAGchainWrite, writes all bits into the scan chain
2. JTAGchainRead, reads all bits out of the scan chain

With these functions all bits of the chain are read and written. A check of the read result or content is not done. This is the job of the user's application.

```
procedure JTAGBoundaryOpen(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;  
                           var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, progNoJTAG, progNoSupply, errPwrSupply, errSignature, resOk
```

```
procedure JTAGchainWrite(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, , resOk
```

```
procedure JTAGchainRead(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, , resOk
```

```
procedure JTAGBoundaryClose(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 UPP 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.

Note:

With the single channel DLL “Channel“ must always contain a 0, with the 2 channel version it can be 0 or 1, with the 4 channel version “Channel“ can be 0..3 etc.

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 GetProgIDs (var res : integer; resStr : PChar);

// res = count of programmers found

This function scans the system for all connected UPP-programmers. The parameter “res“ returns the count of programmers found. The parameter “resStr“ contains a corresponding count of UPP 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. With most cases these names are not relevant for 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 programmer. 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 UPP-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 = progBusy, noProg, progDefect, MMCprotected, MMCnoMedia, progFound

If an UPP-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. The function generates a hardware reset to the selected programmer. If a “progDefect” is reported, a firmware download with *AVRprog.exe* must be executed.

procedure ClosePort(Channel: Integer; var res : integer; resStr : PChar);

// res = noProg, resNone

This function disconnects a specific UPP programmer and the used USB port becomes closed.

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 USBports are closed.

A complete re-init with “GetProgIDs “ and “InitChannel “ must be done.

AbortAll must also be called before the control programm is terminated.



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 UPP 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 UPP-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
no <CR><LF>	Auto Release
ISP programming <CR><LF> yes <CR><LF>	Programming mode
UPP 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 UPP programmer has complete projects in its flash card this is only necessary with a project change or the project is new or when the UPP 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 AVR's which also have different firmware to program.

Encrypted files

Encrypted files must be created with the PC program *AVRprog.exe* and 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 UPP can find out whether this file is packed or encrypted. If it is an encrypted type then the UPP 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 UPP 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 UPP 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 implicitly 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 requirements. All further following functions are support types and are only necessary for special purposes.

If all relevant UPP-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 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 UPP-programmer have been startet the application must continuously 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 UPP-programmer have been startet the application must continuously 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 UPP-programmer have been startet the application must continuously 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 UPP-programmer have been startet the application must continuously 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 UPP-programmer have been startet the application must continuously 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 continuously poll all started programmers with the function “GetProgStatus“. The DLL itself also continuously receives status informations from the started programmers. These informations are temporarily stored by the DLL. This means that the DLL always presents the latest (newest) state of the programmers.

If this 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. prgEEp : the EEprom will be programmed if enabled.
4. progDone : all done ok, inclusive verify Flash (and EEprom).

Also the special functions **ProgFlash**, **ProgEEprom**, **VerifyDevice**, **VeriFyEEpromOnly** and **VerifyFlashOnly** must be continued and finished with the function GetProgStatus. Each started programmer must be polled continuously. 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. 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);

// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
// dwnLoadErrE, resOk

With these two functions the actual Flash or EEprom 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 address. For example 0, 256, 512, etc. With the Flash 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 and EEPROM 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);
// res = progBusy, noProg, MemError, errPwrDown, errSignature, errProtected, dwnLoadErr, progProtected,
//      resOk
```

The purpose of these 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 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 \neq \$FF. If the origin file or the chip content is used all memory location become reprogrammed but only the altered 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.

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 or EEPROM 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 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);
```

// 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 function return an error if the file is encrypted. An enlargement of the Flash or EEprom 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.

SelfFuse selects the desired fuse. After the execution **Fuse** returns the value found. If successful **resStr** contains this value as a string. The char parameter **SelfFuse** can consist of these values:

L	= LockBits
F	= FuseBits low
H	= FuseBits high
E	= FuseBits extended
0	= OscCal byte 0
1	= OscCal byte 1
2	= OscCal byte 2
3	= OscCal byte 2

Although this function supports all fuse types (L, F, H, E, 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
```

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 UPP 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



JTAG Boundary SCAN functions

The Boundary Scan Chain consists of a count of bits where each single bit corresponds to an internal peripheral bit of the AVR CPU. These bits can be port pins or bits in a control register of a peripheral of the CPU. Some bits for example belong to the AD-converter. But the CPU itself and its registers and memory are not accessible through the Scan Chain.

The application must provide 2 buffers, a read and a write buffer. The size of these buffers depends on the length (bit count) of the Scan Chain. Simply define two arrays of 0..255 of byte. This is sufficient for all CPU types.

Now each bit in these buffers corresponds to a bit in the Scan Chain. The application now can manipulate bits in the download buffer, start a download and then read back all bits with an upload into the upload buffer. Please note that a single bit can be changed in the download buffer but the download always includes all bits of the chain. A direct single bit change in the chain is not possible because a download rewrites always all bits of the chain.

The scan functions implemented in the DLL simply read or write the entire chain, an analysis is not done. This is the job of the application. The Boundary Scan part of the DLL, in the programmer and also in the CPU is completely separated from the programming part. Both have nothing in common. In the DLL only the functions *GetProgIDs* and *InitChannel* are common for both parts. An Encryption file load etc. is not necessary for the scanning.

```
procedure JTAGBoundaryOpen(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;  
                           var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, progNoJTAG, progNoSupply, errPwrSupply, errSignature, resOk
```

The parameter CpuID and BoundaryLen must be copied from the datasheet of the AVR CPU.

With the mega16 is CpuID = \$001E9403 and BoundaryLen = 141 (Bits).

mega128 CpuID = \$001E9702 BoundaryLen = 205 (Bits)

The parameter SupplyVolt and SupplyCurr together define the supply of the external target CPU. If the target has its own powersupply so both parameter must be zero. If the programmer must supply the target so SupplyVolt must contain the desired voltage (eg. 500 for 5Volt). Then SupplyCurr must contain the max. current, 30 or 100 (mA). If the programmer must supply the target so the programmer must have its charge unit connected or a powersupply with 6..10V= must be connected to its charge jack.

```
procedure JTAGchainWrite(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, , resOk
```

Download the local write-buffers into the Scan Chain. The pointer **Block** must point to this download buffer.

```
procedure JTAGchainRead(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, , resOk
```

Read-back the Scan Chain out of the target into the local read-buffer. The pointer **Block** must point to this upload buffer.

```
procedure JTAGBoundaryClose(Channel: integer; var res : integer; resStr : Pchar);
```

```
// res = progBusy, noProg, resOk
```

Closes the JTAG Scan Port of the target CPU.

AbortAll must also be called before the control program is terminated.



Examples and sources

This system contains the WIN32 DLL **UPP_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, noProg, progFound, MemError, progBusy, progDefect, progIdle, progProtected,
eraChip, eraEEp, eraFlash, prgEEp, prgFlash, verifyEEp, verifyFlash, errPwrDown,
errSignature, errProtected, errNotEmpty, errVerify, progDone, dwnLoading, dwnLoadErrP,
dwnLoadErrE, dwnLoadErrF, dwnLoadErr, invFile, invFName, FileExist, notFound,
invPassword, limitExc, errProgType, resOk, progNoJTAG, progNoSupply, errPwrSupply,
errParm, MMCprotected, MMCnoMedia);
```

```
procedure GetProgIDs(var res : integer; resStr : PChar); external 'UPP_DLL.dll'
procedure InitChannel(Channel : integer; var res : integer; resStr : PChar); external 'UPP_DLL.dll'
procedure DeleteAfile(Channel: Integer; FileName: PChar; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure GetFileNames(Channel : integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure CheckProgrammer(Channel: integer; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure CheckAfile(Channel : integer; FileName: PChar; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure OpenAfile(Channel : integer; FileName: PChar; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure GetProjParams(Channel : integer; FileName: PChar; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure GetFileState(Channel : integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure CloseAfile(Channel : integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure DownloadFile(Channel : Integer; FileName: PChar; var res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure GetTargVolt(Channel: integer; var Volt, res : integer; resStr : PChar); stdcall;
external 'UPP_DLL.dll';
```

```
procedure ProgDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure EraseDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure ProgFlash(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure ProgEEProm(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure ProgFuses(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure ProgLockBits(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure VerifyDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure VerifyFlashOnly(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure VerifyEEPromOnly(Channel: Integer; var res : integer; resStr : PChar); stdcall; external
'UPP_DLL.dll';
```

```
procedure GetProgStatus(Channel: integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure ClosePort(Channel: integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
procedure CheckDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure ReadBackChipF(Channel : Integer; Block : Pointer; source : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure ReadBackChipE(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP_DLL.dll';
```

```
procedure DownProgBlockF(Channel : Integer; Block : Pointer; dest : Integer; var res : integer; resStr : PChar);
stdcall; external 'UPP_DLL.dll';
```

```
procedure DownProgBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;
resStr : PChar); stdcall; external 'UPP_DLL.dll';
```



```
procedure DownOverBlockF(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure DownOverBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure DownFileBlockF(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure DownFileBlockE(Channel : Integer; Block : Pointer; dest, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure UpFileBlockF(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure UpFileBlockE(Channel : Integer; Block : Pointer; source, count : Integer; var res : integer;  
    resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure GetFuses(Channel : integer; SelfFuse : char; var Fuse : integer; var res : integer; resStr : PChar);  
    stdcall; external 'UPP_DLL.dll';  
procedure AbortAll; stdcall; external 'UPP_DLL.dll';  
  
procedure JTAGBoundaryOpen(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;  
    var res : integer; resStr : PChar); stdcall; external 'UPP_DLL.dll';  
procedure JTAGchainWrite(Channel : Integer; Block : Pointer; var res : integer; resStr : PChar); stdcall;  
    external 'UPP_DLL.dll';  
procedure JTAGchainRead(Channel : Integer; Block : Pointer; var res : integer; resStr : PChar); stdcall;  
    external 'UPP_DLL.dll';  
procedure JTAGBoundaryClose(Channel: integer; var res : integer; resStr : PChar); stdcall;  
    external 'UPP_DLL.dll';
```



Import of the DLL into Visual Basic

VB Interface

```
tProgResult = (resNone, noProg, progFound, MemError, progBusy, progDefect, progIdle, progProtected,  
eraChip, eraEEp, eraFlash, prgEEp, prgFlash, verifyEEp, verifyFlash, errPwrDown,  
errSignature, errProtected, errNotEmpty, errVerify, progDone, dwnLoading, dwnLoadErrP,  
dwnLoadErrE, dwnLoadErrF, dwnLoadErr, invFile, invFName, FileExist, notFound,  
invPassword, limitExc, errProgType, resOk, progNoJTAG, progNoSupply, errPwrSupply,  
errParm, MMCprotected, MMCnoMedia);
```

```
Public Const resNone = 0  
Public Const noProg = 1  
Public Const progFound = 2  
Public Const MemError = 3  
Public Const progBusy = 4  
Public Const progIdle = 5  
Public Const progProtected = 6  
Public Const eraChip = 7  
Public Const eraEEp = 8  
Public Const eraFlash = 9  
Public Const prgEEp = 10  
Public Const prgFlash = 11  
Public Const verifyEEp = 12  
Public Const verifyFlash = 13  
Public Const errPwrDown = 14  
Public Const errSignature = 15  
Public Const errProtected = 16  
Public Const errNotEmpty = 17  
Public Const errVerify = 18  
Public Const progDone = 19  
Public Const dwnLoading = 20  
Public Const dwnLoadErrP = 21  
Public Const dwnLoadErrE = 22  
Public Const dwnLoadErrF = 23  
Public Const dwnLoadErr = 24  
Public Const invFile = 25  
Public Const invFName = 26  
Public Const FileExist = 27  
Public Const notFound = 28  
Public Const invPassword = 29  
Public Const limitExc = 30  
Public Const errProgType = 31  
Public Const resOk = 32  
Public Const progNoJTAG = 33  
Public Const progNoSupply = 34  
Public Const errPwrSupply = 35  
Public Const errParm = 36  
Public Const MMCprotected = 37  
Public Const MMCnoMedia = 38
```

Please note: all integer are 32bit types



```
Public Declare Sub GetProgIDs Lib "UPP_DLL.dll" (ByRef res As Integer, ByVal strRes As String)

Public Declare Sub InitChannel Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub GetProgStatus Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub ClosePort Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub DeleteAfile Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal FileName As String,
ByRef res As Integer, ByVal strRes As String)

Public Declare Sub GetFileNames Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub CheckProgrammer Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub CheckAfile Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal FileName As String,
ByRef res As Integer, ByVal strRes As String)

Public Declare Sub OpenAfile Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal FileName As String,
ByRef res As Integer, ByVal strRes As String)

Public Declare Sub GetProjParams Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal FileName As String,
ByRef res As Integer, ByVal strRes As String)

Public Declare Sub GetFileState Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub CloseAfile Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)

Public Declare Sub GetTargVolt Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef Volt As Integer, ByRef
res As Integer, ByVal strRes As String)

Public Declare Sub DownloadFile Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal FileName As String,
ByRef res As Integer, ByVal strRes As String)

Public Declare Sub AbortAll Lib "UPP_DLL.dll" ()

Public Declare Sub ProgDevice Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub EraseDevice Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub ProgFlash Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub ProgEEProm Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub ProgFuses Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal
strRes As String)

Public Declare Sub ProgLockBits Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer,
ByVal strRes As String)
```



Public Declare Sub VerifyFlashOnly Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub VerifyEEPromOnly Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub CheckDevice Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub ReadBackChipE Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal source As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub ReadBackChipF Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal source As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownProgBlockF Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal dest As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownProgBlockE Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal dest As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownOverBlockF Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal dest As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownOverBlockE Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal dest As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownFileBlockF Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal count As Integer, ByVal dest As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub DownFileBlockE Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal count As Integer, ByVal dest As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub UpFileBlockF Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal source As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)

Public Declare Sub UpFileBlockE Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByVal source As Integer, ByVal count As Integer, ByVal res As Integer, ByVal strRes As String)



JTAG Boundary Scan

Private Declare Sub JTAGBoundaryOpen Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal CpuID As Integer, ByVal BoundaryLen As Integer, ByVal SupplyVolt As Integer, ByVal SupplyCurr As Integer, ByRef res As Integer, ByVal strRes As String)

Private Declare Sub JTAGchainWrite Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByRef res As Integer, ByVal strRes As String)

Private Declare Sub JTAGchainRead Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByVal Block As Pointer, ByRef res As Integer, ByVal strRes As String)

Private Declare Sub JTAGBoundaryClose Lib "UPP_DLL.dll" (ByVal Channel As Integer, ByRef res As Integer, ByVal strRes As String)



Import of the DLL into C++

```
////////////////////////////////////
//
// UPP Programmer DLL Header
//
//
////////////////////////////////////

#define DLLImport __declspec( dllimport )

////////////////////////////////////
// function call results
resNone, noProg, progFound, MemError, progBusy, progDefect, progIdle, progProtected, eraChip, eraEEp,
eraFlash, prgEEp, prgFlash, verifyEEp, verifyFlash, errPwrDown, errSignature, errProtected, errNotEmpty,
errVerify, progDone, dwnLoading, dwnLoadErrP, dwnLoadErrE, dwnLoadErrF, dwnLoadErr, invFile,
invFName, FileExist, notFound, invPassword, limitExc, errProgType, resOk, progNoJTAG, progNoSupply,
errPwrSupply, errParm, MMCprotected, MMCnoMedia

enum ProgDLL_Result
{
    resNone,
    noProg,
    progFound,
    MemError,
    progBusy,
    progIdle,
    progProtected,
    eraChip,
    eraEEp,
    eraFlash,
    prgEEp,
    prgFlash,
    verifyEEp,
    verifyFlash,
    errPwrDown,
    errSignature,
    errProtected,
    errNotEmpty,
    errVerify,
    progDone,
    dwnLoading,
    dwnLoadErrP,
    dwnLoadErrE,
    dwnLoadErrF,
    dwnLoadErr,
    invFile,
    invFName,
    FileExist,
    notFound,
    invPassword,
    limitExc,
    errProgType,
    resOk,
    progNoJTAG,
    progNoSupply,
    errPwrSupply,
    errParm,
    MMCprotected,
    MMCnoMedia
};
```



*// Please note: all **integer** are **32bit** types*

////////////////////////////////////

// functions

TBD.

JTAG Boundary Scan

TBD.



Flow diagrams

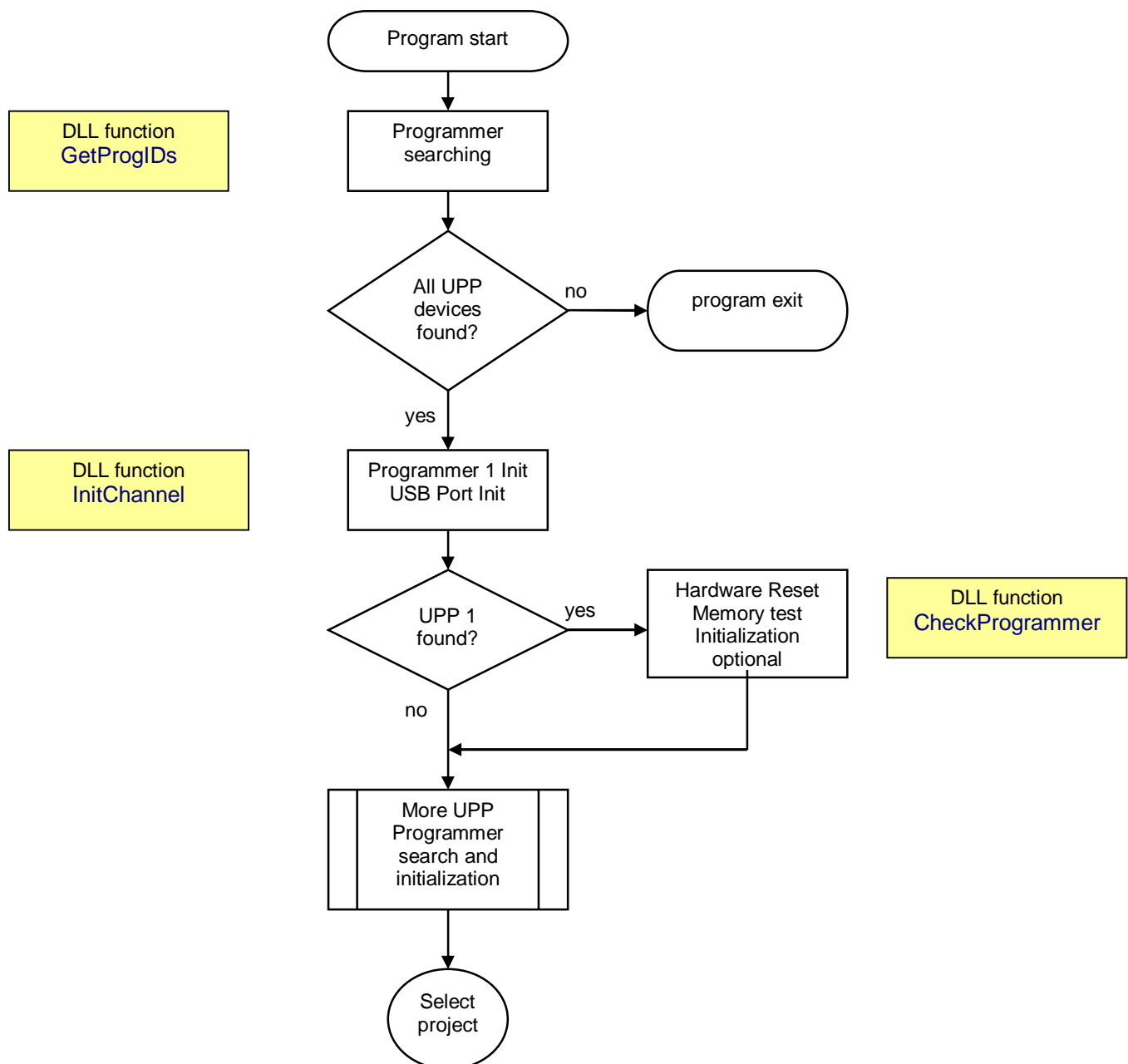
For a better view onto the usage of the E-LAB UPP-DLL in the production there are three diagrams:

1. Initialization 2. Project select 3. Production.

Initialization

The UPP-programmers must be connected to the Host via a USB port. An USB-enabled 32bit Windows must run on this computer. (WIN98/WIN2000/XP). The DLL „UPP_DLL.dll“ must reside in the same directory where the application program which controls the DLL and programmers is located.

At start up of the application at first the connected programmers must be searched and mapped into “Channels”. If a UPP-programmer is found via the DLL this UPP gets also a hardware reset. The UPP can respond with a memory error to the application. Now the init sequence is complete.

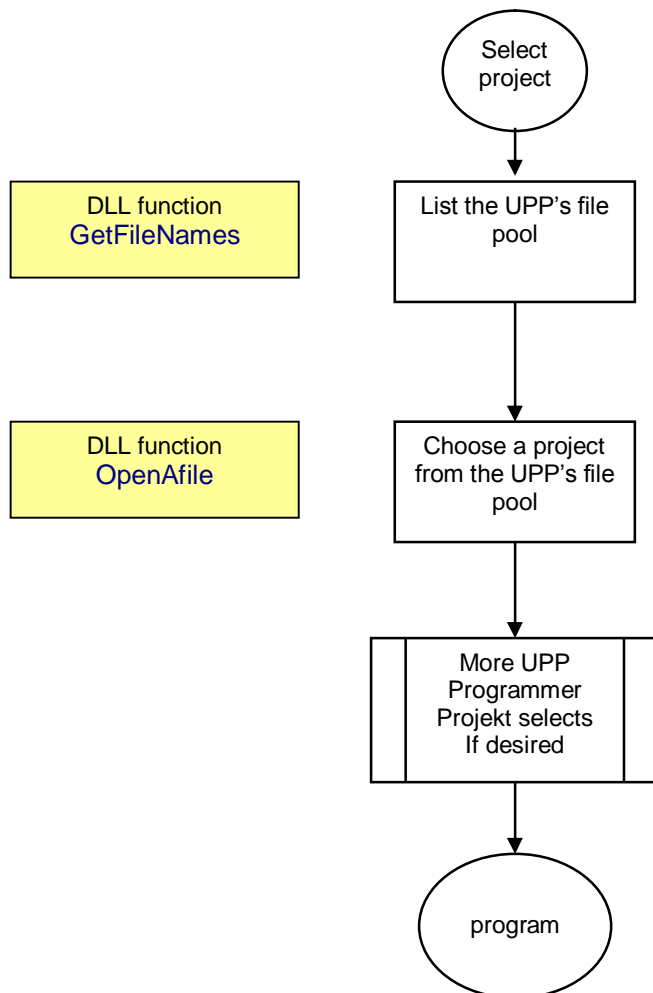




Select Project

The UPP programmer memory is flash based and retain all once stored project also through a power down cycle. The selected file is checked against some checksums.

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

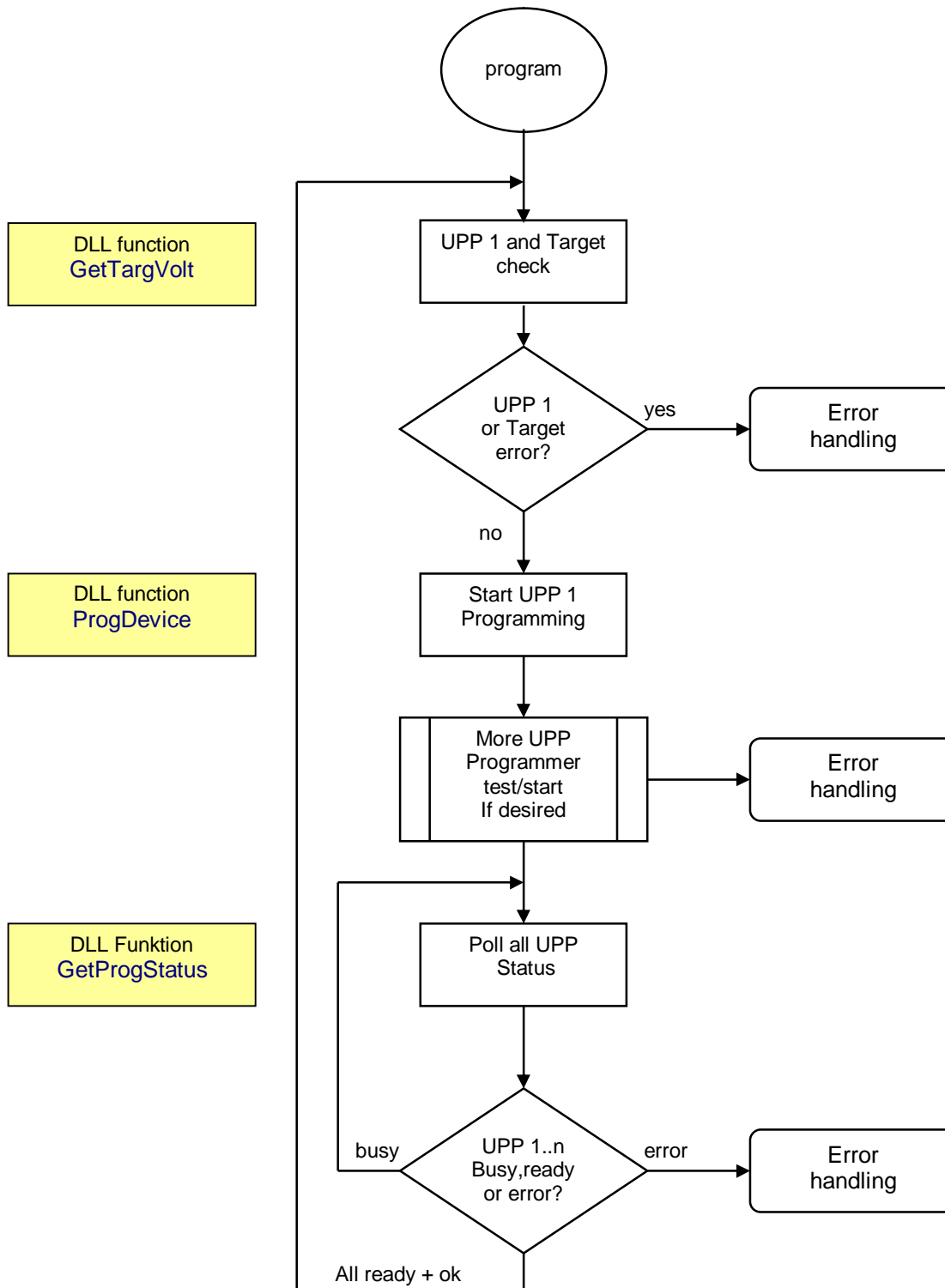




Production

The programming cycle must be started separately for each connected UPP programmer. The start function itself results in an OK or FAIL. The reason for a fail can be a disconnected or defective UPP 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 be 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 Tel. 07268/9124-0 WEB: www.e-lab.de	D74906 Bad Rappenau Germany Fax. 07268/9124-24 mail: info@e-lab.de
--	--