# E-LAB

## Production Programmer System

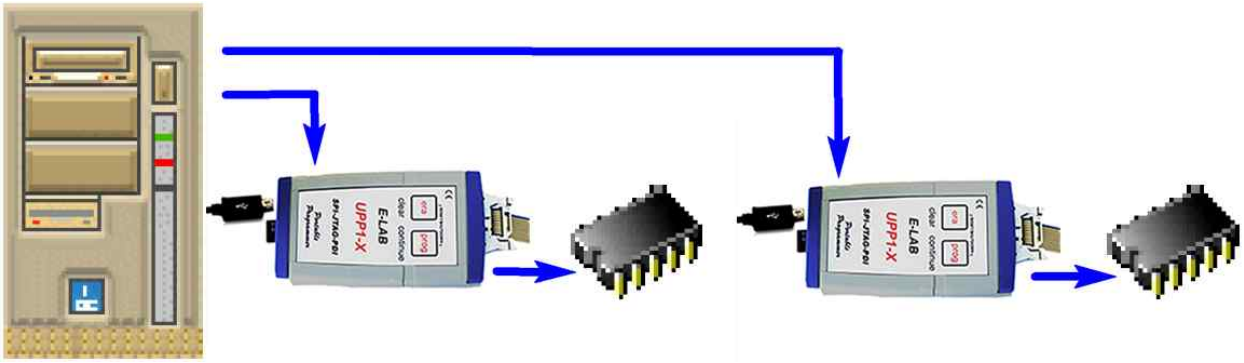
# AVR *UPP1X-DLL*


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

- **Produktions Programmiersystem für Atmel AVR CPUs**
- **Bis zu 16 simultane Programmierungen**
- **Steuerung über WIN32 USB**
- **Steuerungs Programm vom Anwender selbst erstellbar**
- **Eine USB Schnittstelle pro Programmer**
- **Durch Einsatz der E-LAB UPP1-X Programmer flexible und schnelle InCircuit Programmierung**
- **Dateien können auf einem PC gepackt oder verschlüsselt werden. Die erzeugten Dateien oder Projekte, bis zu 16 Stück, werden auf einer SD Flashkarte abgespeichert.**
- **Einfacher Programmwechsel in der Produktion durch Austausch der Flashkarte oder durch Auswahl anderer Dateien auf der Flashkarte**
- **Datei/Projekt Pflege auch direkt durch die DLL mit den Support Funktionen: List all Files, Delete File, Download File, Check File etc.**

März 2014

# In-Circuit Programmierung von Single Chips in der Produktion




In der Serien Produktion von elektronischen Baugruppen die mit Single-Chips bestückt sind, stellt sich immer wieder das Problem der In-Circuit Programmierung der CPUs innerhalb des Board Test Vorgangs. Herkömmliche ISP Programmiergeräte sind zwar preiswert und manchmal auch problemlos in der Handhabung, eignen sich jedoch nicht besonders gut für automatisierte Vorgänge.

Lässt sich der Programmierstart noch in den meisten Fällen durch einen Start Impuls noch relativ einfach steuern, so kommt es spätestens bei der Ergebnisauswertung zu einem Problem. Die zum Programmer zugehörige Steuersoftware ist in erster Line für manuellen Betrieb gedacht. Auch wenn das Protokoll für die Anbindung des Programmers an einen PC offengelegt ist, bedeutet das erheblichen Programmieraufwand bei der Erstellung eigener Treibersoftware.

Moderne Single-Chips wie der AVR benötigen z.B. eine grosse Anzahl von Fuse- und Lockbits, die das System alle handhaben muss. Auch ist es i.A. nicht Aufgabe der Produktions Verantwortlichen umfangreiche und komplexe Software zu erstellen, vor allen Dingen wenn der Produktionsstandort räumlich von der Entwicklung getrennt ist.

Fast unlösbar wird die Aufgabe, wenn im Nutzen programmiert werden soll, oder wenn mehrere CPUs sich auf dem selben Board befinden. Der Software Aufwand steigt hierbei enorm.

Wenn das Programmiersystem aber komplett durch eine sogenannte DLL (dynamic link library) gesteuert werden kann, reduziert sich der Programmaufwand in dem Testsystem im wesentlichen auf allgemeine Kommandos wie Start programming und Resultat Auswertung.

Mit der E-LAB Programmer-DLL lässt sich relativ einfach die In-Circuit Programmierung in ein vorhandenes Testsystem integrieren. Die Software kommuniziert mit ein paar wenigen DLL-Calls über die DLL mit den bis zu 16 über USB angeschlossenen Programmer. Die DLL leistet dabei die komplette Arbeit, ohne das Testsystem über Gebühr zu belasten.

Ein Produktions Programmiersystem besteht aus der E-LAB Programmer-DLL und 1 bis 16 E-LAB UPP1-X Programmern.

Die UPP1X-DLL für einen UPP1-X ist kostenlos und kann von der E-LAB homepage herunter geladen werden. Die multi-Programmer Version kostet einmalig €250.00

**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**

# Produktions Programmierer UPP1X-DLL

Die E-LAB Programmer DLL dient als Bindeglied zwischen Test-Automat (oder PC) und UPP-Programmern in der Produktion von Boards, die mit Atmel AVRs bestückt sind. Die DLL kann bis zu 16 UPP-Programmer gleichzeitig steuern und ist daher sehr gut für die Nutzen Fertigung bzw. Programmierung geeignet.

Das System besteht aus einer Windows DLL (XP, WIN7) und mehreren über USB angeschlossenen UPP1-X Programmern. Der Host (Testsystem oder PC) müssen ein funktionierendes USB Interface haben.

Der Test Automat (Applikation) kommuniziert über die DLL mit den Programmern. Die DLL muss in dem Test Automat installiert sein. Durch die Verwendung einer DLL kann das Steuerprogramm im Test Automat in fast jeder beliebigen Programmier-Sprache erstellt werden. Auch DLL-fähige Script Tools sind denkbar.

Das System DLL <> Programmer benötigt gepackte oder verschlüsselte Arbeitsdateien, die mit dem E-LAB Programm AVRprog.exe erstellt werden müssen. Diese Dateien enthalten alle notwendigen Informationen (Flash file, EEprom file, CPU-Typ, Lockbits, Fusebits etc). Eine Manipulation bzw. fehlerhafte Einstellung der Programmer durch die Fertigung ist damit ausgeschlossen.

## *Flash Card*

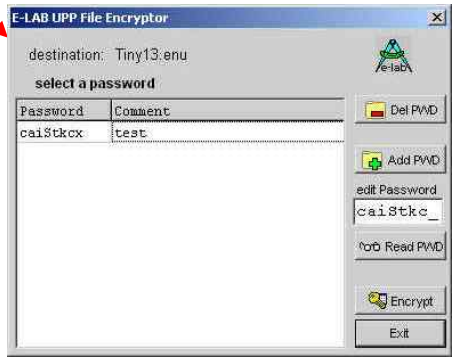
Der UPP1-X Programmer enthält ein FAT16 File System um Flash Karten des Typs **microSD** lesen und schreiben zu können. Allgemeine Infos zum Datei Download finden Sie im UPP1-X Handbuch.

**Datei Typen**

Die DLL und der UPP1-X können zwei Datei Typen handhaben (packed oder encrypted). Diese Dateien müssen durch das E-LAB PC-Programm ***AVRprog.exe*** erstellt werden. Mit untenstehenden Menu Punkten können UPP1-X Dateien erstellt werden. Diese Files können auch mit dem UPP1-X Download Dialog erstellt werden. Eine encrypted Datei ist ein File dessen Inhalt verschlusselt ist und nur mit einem bestimmten Schlüssel wieder lesbar wird. Dieser Schlüssel ist bei den UPP1-X Programmern fest im Gerät einprogrammiert. Deshalb sind diese Dateien auch nur auf diesem Gerät wieder lesbar.


Eine gepackte Datei ist eine binäre Datei, die alle notwendigen Informationen enthält, jedoch ***keine*** Verschlüsselung oder Passwörter. Ein gepacktes Projekt ist auf jedem UPP1-X verarbeitbar.
Die Datei Endung ist immer **.pack** oder **.pac**


Braucht man für das Verschlüsseln eines UPP1-X Projekts den Schlüssel bzw. Passwort eines UPP1-X Programmers, so kann dieser mit Hilfe obigen Menu Punktes herausgefunden werden. Der Butzer dieses Programmers teilt diesen Schlüssel dem Projekt bzw. Datei Ersteller mit. Letzterer muss den Schlüssel in untenstehenden Dialog eintragen.




In der Passwort Liste sind alle dem System bekannten UPP1-XSchlüssel eingetragen. Mit ***Del PWD*** kann ein Eintrag gelöscht werden und mit ***Add PWD*** kann ein neuer Schlüssel hinzugefügt werden. Dieser neue Schlüssel wird entweder durch den Benutzer des Programmers mittels obigen Dialog festgestellt und dem Projekt Ersteller mitgeteilt oder der aktuell ange-schlossene UPP1-X Programmer wird mit dem ***Read PWD*** Button abgefragt. Im ersten Fall muss das übermittelte Passwort (Key) manuell in das Editierfeld eingetragen werden. Im zweiten Fall geschieht dies automatisch durch den ***ReadPWD*** Button.

Enthält das Edit Feld das neue Passwort bzw. Schlüssel wird es mit dem ***Add PWD*** Button der Liste hinzugefügt.

Das aktuell geladene Projekt kann jetzt mit dem ***Encrypt*** Button verschlüsselt werden. Dazu muss einer der in der Liste verfügbaren Schlüssel ausgwählt werden (blauer Balken). Die erzeugte Datei kann direkt auf einer Flash Card (MMC oder SD) abgelegt werden. Alternativ kann sie auch per email an den Anwender verschickt werden, der sie selbst auf seine Flash Card kopiert. Eine so generierte Datei ist nur mit diesem dafür vorgesehenen Programmer, von dem der Schlüssel stammt, lesbar und benutzbar. Die Entschlüsselung erfolgt nur während des Programmierens. Damit wird der eigentliche Datei Inhalt auch hier niemals sichtbar.

# *DLL*

Die DLL übernimmt im wesentlichen alle anfallenden Aufgaben und ist so konzipiert, dass das Steuerprogramm im Test Automat (oder PC) sich auf das wesentliche reduzieren lässt : Download von Programmen/Firmware in die einzelnen UPP-Programmer, Programmieren starten, Ergebnis auswerten. Alle Funktionen liefern als Funktions Ergebnis einen Text (Pchar) und ein Integer (res) als Aufzählungstyp (Enumeration) zurück.

**Aufzählungstyp (Enumeration)** der Funktions Ergebnisse:

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

Hierbei hat z.B. „resNone“ den Wert 0 und „progFound“ den Wert 4. ProgBusy = 5

Die **Interface Funktionen** der DLL sind folgende (in Pascal Notation):
Zumindest beim ersten Kontakt mit der DLL muss die Funktion ***AbortAll*** aufgerufen werden!

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

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,
//        dwnLoadErrE, 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
```

# Funktions Details

Im folgenden wird der Begriff „USBport“ für den USBport Namen des Steuerrechners benutzt. Der Begriff „Channel“ steht für einen bestimmten Programmer Port des Systems (0..15) und damit indirekt für einen speziellen am Host angeschlossenen UPP-Programmer. Mit der DLL Funktion „InitChannel“ wird einem Channel ein bestimmtes USB Port zugewiesen/gemappt. Mit der Einkanal Version der DLL muss „Channel“ immer 0 sein. „Steuer System“ und „Host“ stehen hierbei für das Testsystem, auf dem die DLL und das zugehörige Programm installiert ist. Zumindest beim ersten Kontakt mit der DLL muss die Funktion ***AbortAll*** aufgerufen werden!

## *Initialisierung*

Diese Funktionen werden vor dem Programmier Start einmal aufgerufen. Die ersten beiden sind absolut notwendig. Zumindest beim ersten Kontakt mit der DLL muss die Funktion ***AbortAll*** aufgerufen werden!

procedure **AbortAll**;

Diese Prozedur ist notwendig, wenn das System zum Beispiel „hängt“ oder die DLL mit dem Host oder mit einem oder mehreren Programmern ausser Synchronisation gekommen ist. Nach dem Kommando AbortAll befindet sich der DLL im Grundzustand und kann neue Befehle empfangen. Alle USBorts werden geschlossen.
Es muss jetzt eine komplette Neu-Initialisierung mit „GetProgIDs“ und „InitChannel“ erfolgen. Auch wenn Programmer abhehängt oder neu angeschlossen werden.

**AbortAll** muss auch aufgerufen werden, bevor das Steuerprogramm geschlossen wird.

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

Diese Funktion sucht nach allen angeschlossenen UPP-Programmern. In „res“ wird die Anzahl der gefundenen UPPs zurückgegeben. Der Parameter „resStr“ enthält eine entsprechende Anzahl von UPP1-X Namen, die durch CRLF getrennt sind. Beispiel:
***ECK5VMEA***<CR><LF>***ECM22DS8***<CR><LF>
Eine immer gleichbleibende Reihenfolge kann nicht gewährleistet werden. Sie wird durch Windows vorgegeben. Die Applikation braucht sich diese Namen nicht weiter zu merken, muss aber wissen, dass der erste Namen dem Kanal-0 zugeordnet ist, der zweite dem Kanal-1 etc. Im folgenden wird immer nur mit den Kanal Nummern gearbeitet. Eine physische Zuordnung zwischen Programmer und logische Nummer ist damit gegeben.

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

In „Channel“ wird eine Zahl (0..15) vorgegeben, mit der der Programmer, falls gefunden, zukünftig selektiert werden muss. Im Erfolgsfall ist „Channel“ damit belegt und kann nur wieder gezielt mit „ClosePort“ oder allgemein mit „AbortAll“ freigegeben werden. Diese Funktion muss pro Programmer einmal aufgerufen werden. Sie löst einen Hardware Reset im Programmer aus. Wenn „ProgDefect“ angezeigt wird muss ein neuer Firmware Download mit *AVRprog.exe* erfolgen.

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

Wurde ein UPP-Programmer mit „InitChannel“ gefunden und initialisiert, kann dieser jederzeit mit dieser Funktion getestet werden. Beim Funktionsaufruf bezeichnet der Parameter „Channel“ den zu suchenden bzw. zu prüfenden Programmer.

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

Nach den moisten Funktionen bleibt dieTarget CPU im Reset-mode. Diese Funktion löst den Reset Status.

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

Hiermit wird ein bestimmter UPP1-X Programmer abgemeldet und das zugehörige USB Port wird geschlossen.

# *Verwaltungs Funktionen für Dateien*

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

Diese Funktion dient zum Auflisten der im UPP1-X Programmer auf der MMC/SD gespeicherten Projekte. Bei resOk enthält ‚res" die Anzahl der Projekte und „resStr" enthält einen String, der alle Projekt- bzw. File Namen enthält. Diese sind durch CRLF getrennt. Beispiel:
***TINY13.PAC***<CR><LF>***M128_1.ENU***<CR><LF>***TEST1200.PAC***<CR><LF>
Es ist Sache der Applikation diese Namen aus dem String herauszulösen.

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

Diese Funktion dient zur textuellen Anzeige der Programmier Parameter eines in einem UPP-Programmer gespeicherten Projekts. Im Parameter „FileName" wird der Namen des gewünschten Projekts übergeben. Wenn „res = resOk" enthält der String „resStr" 12 Substrings, separiert durch CRLF. Beispiel:

| | |
|---|---|
| ***Test Tiny13***<CR><LF> | Projekt Namen |
| ***2003.Nov.08 12:15:22***<CR><LF> | Projekt Erstellung |
| ***2003.Nov.27***<CR><LF> | Projekt Download |
| ***TINY13***<CR><LF> | CPU Namen |
| ***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

Diese Funktion ist normalerweise nur aufzurufen, wenn ein neues Projekt in einen bestimmten, am Host angeschlossenen Programmer geladen werden soll. Da die E-LAB UPP1-X Programmer ein Flash File System haben, muss nur bei einer Projekt Änderung oder einem neuen Projekt dieses für diesen Programmer neu herunter geladen werden. Der Parameter **FileName** muss den Quell-Pfad enthalten, so dass die DLL das File finden kann. Die File Extension (.pac oder .enu) muss mit angegeben werden.

Es ist selbsverständlich möglich, dass jeder angeschlossene Programmer eigene Projekte erhält, so dass auch grosse Boards mit mehreren unterschiedlichen AVRs mit unterschiedlicher Firmware programmiert werden können.

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

Eine Datei/Projekt wird aus der Flashcard des Programmers entfernt. FileName darf keine Drive oder Pfad Informationen enthalten. Hat die Card einen Schreibschutz, wird eine Löschung mit „errProtected" zurückgewiesen. Das gleiche gilt für Dateien, die durch den PC mit einem Card Writer direkt auf die Karte geschrieben wurden. Diese FAT32 Files können nicht im FAT16 Mode des UPP1-X gelöscht oder geändert werden.

## *Encrypted Files*

Verschlüsselte Dateien werden mit dem PC Programm *AVRprog.exe* erstellt. Dazu muss ein ISP3-X, ein UPP1-X oder ein UPP2-X Programmer angeschlossen sein! Das File wird dann ohne Veränderung in den Programmer heruntergeladen. Dieser entschlüsselt das File zur Laufzeit. Der verwendete Schlüssel muss mit dem Schlüssel des Programmers übereinstimmen, ansonsten kommt die Fehlermeldung „invFile“ wenn das File zur Verarbeitung geöffnet wird. Es kommt ausschlieslich AES Encryption zur Anwendung!

## *Packed files*

Gepackte Dateien werden mit dem PC Programm *AVRprog.exe* erstellt und ohne Veränderung in den Programmer heruntergeladen. Dieser entpackt das File zur Laufzeit.

## *Produktion*

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

Eine Datei/Projekt wird aus der Flashcard des Programmers eröffnet und geladen. FileName darf keine Drive oder Pfad Informationen enthalten. Der UPP1-X erkennt automatisch ob dieses File gepackt oder verschlüsselt ist. Ist sie encrypted, dann wird geprüft, ob der UPP1-X interne Schlüssel zu der Datei passt. Ist das nicht der Fall, wird mit dem Ergebnis „invFile“ abgebrochen.
Eine gepackte Datei wird ebenfalls auf Integrität geprüft und ggf. mit „invFile“ abgebrochen.

Diese Funktion muss mindestens ein mal aufgerufen werden, um irgend eine der Operationen ausführen zu können, die mit der Target CPU zu tun haben. Praktisch alle im folgenden aufgeführte Funktion erwarten, dass eine Datei geöffnet wurde.

Während eine Datei/Projekt geöffnet ist dürfen keine der obigen Datei Verwaltungs Funktionen aufgerufen werden. Sollte dies aber gebraucht werden, muss mit einem „CloseAfile“ zuerst eine evtl. geöffnete Datei geschlossen werden. Nach dem Abarbeiten einer Verwaltungsfunktion muss zuerst wieder mit „OpenAfile“ dem Programmer ein Arbeits Projekt vorgegeben werden.

Im Erfolgsfall werden die Steuerparameter geladen und diverse Operationen und Checks durchgefahren:

1. Muss der UPP1-X das Target mit Spannung versorgen, wird diese erzeugt und geprüft ob diese auch richtig anliegt. Im Fehlerfall kommt ein „errPwrDown“ zurück. Muss der UPP1-X das Target nicht versorgen so wird nur die Target Spannung gemessen und bei unzureichenden Werten ebenfalls ein „errPwrDown“ zurückgegeben.
2. Der erste Zugriff auf die Target CPU erfolgt mit dem Lesen der CPU-internen ID Nummer. Stimmt diese nicht mit der Vorgabe überein, bricht die Funktion mit einem „errSignature“ ab.
3. Ist die CPU durch die Lockbits auslesegeschützt, erfolgt die Fehlermeldung „errProtected“. Diese Meldung kann ignoriert werden, wenn die CPU mit ProgDevice komplett neu programmiert werden soll, da dieses Kommando ein EraseDevice einschliesst, was wiederum die Lockbits löscht.

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

Diese Funktion stellt fest ob eine Datei/Projekt geöffnet ist oder nicht. Ist kein Projekt geöffnet wird ein „invFName“ zurückgegeben. Ist ein Projekt offen, kommt ein „resOk“.

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

Diese Funktion schliesst ein eventuell geöffnetes File/Projekt.

procedure **ProgDevice**(Channel: Integer; var res : integer; resStr : Pchar);

// res = progBusy, noProg, resOk

Mit dieser Funktion wird ein Programmiervorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Zuerst wird ein Erase durchgeführt. Dann folgen die Programmierung von Flash, EEprom, Fusebits und Lockbits. Welche Teile des Targets und wie diese programmiert werden, muss durch den Dialog „Programmer Options“ im Programm *AVRprog.exe* for dem erstellen des Pack oder Encryption Files festgelegt werden.

Diese Funktion ist die Haupt Operation der DLL. Normalerweise genügt diese allen Anforderungen. Alle weiteren folgenden Funktionen sind Support und werden nur in speziellen Fällen gebraucht.

Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen.

## *Erweiterte Programmierung*

procedure **EraseDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
Mit dieser Funktion wird ein Chip Löschvorgang gestartet. Ist das Ergebnis „res“ = resOk, war die Operation erfolgreich.

procedure **EraseDeviceX**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
Nur für XMegas. EraseDevice löscht beim XMega die UserRow nicht. Mit dieser Funktion wird auch die UserRow gelöscht.

procedure **EraseDeviceUsr**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, invFile, errPwrDown, errNotEmpty, resOk
Nur für XMegas. Nur die UserRow wird gelöscht.

procedure **ProgFlash**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk
Diese Funktion startet die Flash-only Programmierung. Es wird nur das Flash selbst programmiert.
Es ist wird kein Erase vorher durchgeführt. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen.
***Nicht*** für encrypted Files anwendbar!

procedure **ProgEEprom**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, resOk
Diese Funktion startet die EEprom-only Programmierung. Es wird nur das EEprom selbst programmiert. Es ist wird kein Erase vorher durchgeführt. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen.
***Nicht*** für encrypted Files anwendbar!

procedure **ProgUsrRow**(Channel: Integer; var res : integer; resStr : PChar); // XMega only
// res = progBusy, noProg, resOk rest of states must be done with "GetProgStatus"
Diese Funktion startet die UserRow-only Programmierung. Es wird nur das UserRow selbst programmiert.
Es ist wird kein Erase vorher durchgeführt. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen.
***Nicht*** für encrypted Files anwendbar!

procedure **ProgFuses**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk
Diese Funktion startet die Fuse-only Programmierung. Es werden nur die Fuses programmiert. Ist das Ergebnis „res“ = resOk, war die Operation erfolgreich.
***Nicht*** für encrypted Files anwendbar!

procedure **ProgLockBits**(Channel: Integer; var res : integer; resStr : PChar);
// res = progBusy, noProg, dwnLoadErr, errPwrDown, errSignature, errProtected, resOk
Diese Funktion startet die LockBits-only Programmierung. Es werden nur die LockBits programmiert. Ist das Ergebnis „res“ = resOk, war die Operation erfolgreich.
***Nicht*** für encrypted Files anwendbar!

procedure **VerifyDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
Mit dieser Funktion wird ein Chip Verify Vorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen. Dies ist nur eine Support Funktion, sie ist normalerweise nicht sinnvoll, da jeder Programmiervorgang auch ein Verify mit einschliesst. Macht auch nur Sinn, wenn keine Lockbits im Chip gesetzt sind.

procedure **VerifyFlashOnly**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
Mit dieser Funktion wird ein Flash Verify Vorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen. Dies ist nur eine Support Funktion, sie ist normalerweise nicht sinnvoll, da jeder Programmiervorgang auch ein Verify mit einschliesst. Macht auch nur Sinn, wenn keine Lockbits im Chip gesetzt sind.

procedure **VerifyEEpromOnly**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
Mit dieser Funktion wird ein EEprom Verify Vorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen. Dies ist nur eine Support Funktion, sie ist normalerweise nicht sinnvoll, da jeder Programmiervorgang auch ein Verify mit einschliesst. Macht auch nur Sinn, wenn keine Lockbits im Chip gesetzt sind.

procedure **VerifyUsrRowOnly**(Channel: Integer; var res : integer; resStr : Pchar); // XMega only
// res = progBusy, noProg, resOk
Mit dieser Funktion wird ein UserRow Verify Vorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten UPP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten UPPs den Status abzuholen. Dies ist nur eine Support Funktion, sie ist normalerweise nicht sinnvoll, da jeder Programmiervorgang auch ein Verify mit einschliesst. Macht auch nur Sinn, wenn keine Lockbits im Chip gesetzt sind.

## *Support Funktionen*

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

Diese Funktion ist die eigentliche Arbeitsfunktion. Nach dem Programmier-Start Befehl „ProgDevice“ durch das Steuer System muss mit der Funktion „GetProgStatus“ kontinuierlich alle gestarteten Programmer gepollt werden. Die DLL selbst erhält fortwährend Status Informationen von den gestarteten Programmern. Diese Informationen werden in der DLL temporär gespeichert. Das bedeutet, dass die DLL immer die neueste Information jedes einzelnen Programmers bereit hält.

Werden diese Informationen vom Host nicht rechtzeitig abgeholt, so werden diese durch die neueste Status Information der Programmer ersetzt. Das bedeutet jedoch keinen wichtigen Informations Verlust, da in der Regel nur der letzte, abschliessende Status von Interesse ist.
Die Infos kommen in folgender Reihenfolge:

1. eraChip : die CPU wird gelöscht.
2. prgFlash : das Flash wird programmiert
3. prgEEp : das EEprom wird programmiert, falls so vorgegeben.
4. prgUsr : das UserRow wird programmiert, falls so vorgegeben. // XMega only
5. progDone : der Vorgang inkl. Verify ist komplett abgeschlossen.

Die Spezial Funktionen ***ProgFlash, ProgEEprom, VerifyDevice, VerifyEEpromOnly*** und ***VerifyFlashOnly*** müssen auch mit der Funktion GetProgStatus beendet werden. Auch hier werden kontinuierlich alle gestarteten Programmer gepollt. Dabei sind folgende Stati möglich:

1. prgFlash : das Flash wird programmiert
2. prgEEp : das EEprom wird programmiert
3. verifyFlash : das Flash wird verifiziert
4. verifyEEp : das EEprom wird verifiziert
5. verifyUsr : das UserRow wird verifiziert // XMega only
6. progDone or resOk

Da an jeder Stelle ein Fehler auftreten kann, ist in diesem Fall der Fehlercode der letzte Status, der empfangen wird. Anderst ausgedrückt heisst das dass jeder Programmer so lange gepollt werden muss, bis entweder ein „progDone“ oder ein Fehlercode auftritt. Erst dann ist der Programmer fertig und bereit eine neue Aktion auszuführen.

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

Diese Funktion kann nach z.B. nach dem erfolgreichen Programmiervorgang aufgerufen werden, um festzustellen, ob das Chip auslesegeschützt ist. Im Normalfall ist das jedoch nicht notwendig.

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

Wenn erfolgreich, gibt die Funktion als Resultat im Parameter „Volt“ die Board Spannung (CPU) in 10mV Schritten zurück.

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,
// dwnLoadErrF, dwnLoadErrE, resOk

Mit diesen drei Funktionen kann der aktuelle Flash, EEprom oder UserRow Inhalt der Target CPU ausgelesen werden, vorausgesetzt, das Projekt File ist nicht encrypted und die CPU ist nicht auslesegeschützt. ***Block*** zeigt dabei auf einen mindestens 256byte grossen Speicher Bereich in der Applikation. ***Source*** ist die Auslese Adresse im Chip, wobei hier immer eine 256byte Block Addresse abgegeben werden muss. Also z.B. 0, 256, 512, 1024 etc. Beim EEprom werden „count“ Bytes gelesen. Beim Flash und UserRow werden immer 256 bytes übertragen. Ist Source z.B. 1024 dann werden ab der Speicherstelle 1024 ein Block von 256bytes transferiert.
***Nicht*** für encrypted Files anwendbar!

# *Chip und File Manipulation*

Die DLL bietet diverse Möglichkeiten eine Datei/Projekt im UPP1-X zu manipulieren bzw. zu verändern. Ähnliches trifft auch für den Flash und EEprom Bereich der angeschlossenen CPU zu.

Ist das zugehörige Projekt File encrypted ist eine Manipulation des Files oder des Chips ausgeschlossen
Chips deren LockBits schon aktiviert sind, können natürlich unter keinen Umständen manipuliert werden.

```
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
```

Der Zweck dieser Funktionen ist es zu ermöglichen, dass kleine Teile des Flash oder EEprom nachträglich mit einem anderen Inhalt versehen werden können. Im Falle des Flashs ist diese Überprogrammierung natürlich nur sinnvoll in Bereichen des Flashs, der in den relevanten Stellen $FF, also unprogrammierte Bytes stehen hat.
Aus technischen Gründen und um alle CPU Typen damit erreichen zu können, muss beim Flash oder UserRow mit einem 256 Byte Block gearbeitet werden. Dieser Block muss auf einer 256 Byte Grenze liegen. Das stellt in der Praxis kein Problem dar. Es gibt mehrere Möglichkeiten einen solchen neuen ***Flash***-Block zu erstellen.

1. Einen Speicherblock von 256 Bytes mit $FF füllen. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlockF
2. Den Speicherblock von 256 Bytes mit dem Inhalt des Original Files füllen durch “UpFileBlockF”. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlockF
3. Einen Speicherblock von 256 Bytes durch einen Upload direkt aus dem Flash der CPU mit “ReadBackChipF” füllen. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlockF.

Wird mit $FF gefülltem Block gearbeitet, dann werden nur diese Speicherstellen neu programmiert, deren Inhalt jetzt <> $FF werden müssen. Wird mit dem Original Inhalt des Flashs gearbeitet, werden zwar alle Speicherstellen neu programmiert, geändert werden aber nur die neu besetzten bzw. geänderten. Achtung: es können natürlich nur Bits auf 0 programmiert werden, aber niemals auf 1.

Im Fall dass das ***EEprom*** überschrieben werden soll, gelten obige Einschränkungen natürlich nicht, da EEprom Zellen immer mit jedem neuen Inhalt überschrieben werden können. Aber Achtung: auch das EEprom hat pages. Deshalb muss die Ziel-Adresse immer ein vielfaches der EEprom page size sein, (8..32, abhängig vom AVR Typ).

Der Parameter **Block** ist ein Pointer der auf die 256 Byte grosse Quelle/Buffer in der Applikation zeigen muss. **Dest** ist die Ziel Adresse des 256 Byte Blocks im Flash, EEprom oder UserRow Speicher der CPU. Er muss immer auf den Anfang eines 256 Byte Blocks (Boundary) zeigen.

Die CPU muss schon programmiert sein (Flash und Fuses) und darf nicht auslesegeschützt sein. Das zugehörige Projekt File darf nicht verschlüsselt sein.
***Nicht*** für encrypted Files anwendbar!

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

Diese vier Funktionen ermöglichen das Lesen und die Manipulation des aktuellen Flash und EEprom Inhaltes eines Projekt Files. Damit ist es z.B. möglich während der Produktion Teile der Firmware oder des EEprom Teils zu ändern. Die Blockgrösse „count“ sollte 256 Bytes nicht überschreiten. Für die korrekte Blockgrösse und die Zieladresse „dest“ bzw. Quelladresse „source“ ist der Programmierer selbst verantwortlich. Eine Überprüfung findet nicht statt. Diese Funktionen sind bei verschlüsselten Files nicht anwendbar. Eine Vergrösserung des Flash, EEprom oder UserRow Teils eines Files darf nicht vorgenommen werden, ansonsten wird die Datei korrumpiert!
***Nicht*** für encrypted Files anwendbar!

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

Diese Prozedur liest die LockBits, FuseBits oder Calibration Bytes aus der angeschlossenen CPU aus. Mit **SelFuse** wird die gewünschte Fuse angegeben. In **Fuse** wird der gefundene Wert zurückgegeben. Im Erfolgsfall enthält **resStr** den Wert als String. Der char Parameter **SelFuse** kann folgende Werte haben:

| | |
|---|---|
| **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 |

Obwohl diese Prozedur alle Selektions Zeichen akzeptiert (L, F, H, E, I, J, K, 0, 1, 2, 3) macht es natürlich nur Sinn solche Fuses auszulesen, die auch in der CPU vorhanden sind.

# *Serien Nummer Support*

Obenstehende Chip Manipulationen wie „DownProgBlockF“ können nur nach dem Programmieren der Fusebits, des Flashs und des EEproms und vor dem Programmieren der LockBits vorgenommen werden. Deshalb muss eine Chip Programmierung Schritt für Schritt von der Applikation durchgefahren werden was natürlich länger dauert als die „ProgDevice“ Funktion, die alles in einem Durchgang erledigt. Weiterhin können im Flash nur Bits auf „0“ programmiert werden und niemals auf „1“. Ist das Projekt File encrypted sind obige Funktionen garnicht mehr anwendbar.

Abhilfe schaffen hier die folgenden Funktionen. Die damit bereitgestellten Daten werden während des „ProgDevice“ Vorgangs benutzt und ersetzen hierbei den Original Datei Inhalt. Um „trojanische Pferde“ hier auszuschliessen, ist der betreffende Speicherbereich und der Download auf 32bytes beschränkt.

```
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
```

Der Parameter ***Block*** muss auf die Quelle in der Applikation zeigen. ***Dest*** bestimmt die Ziel Adresse dieses Blocks in dem Flash oder EEprom des Chips. ***Count*** gibt die Blockgrösse an (max. 32). Bedingung ist, dass der Block keine 256 Byte Grenze in der CPU überschreitet. Ungülig wäre Dest = 250, count = 20. Der Block überschreitet damit eine 256 Byte Grenze.

Die Funktionen kopieren einen Block von bis zu 32bytes in den UPP1-X Programmer. Die Standard Programmier Funktion ***ProgDevice*** tauscht bei ihrer Durchführung die Original File Daten durch diese Blöcke aus wenn sie an der Adresse Dest ankommt. Das gilt für das Flash als auch für das EEprom. Es ist damit möglich nach jedem Programmiervorgang einen solchen Download durchzuführen und damit Serien Nummern, die durch die Applikation gebildet werden, fortzuschreiben.

Der Download bleibt solange erhalten und aktiv, bis ein ***OpenAfile*** durchgeführt wird. Dies löscht die Blöcke. Deshalb muss folgendermassen vorgegangen werden:

1. Öffnen des gewünschten Files/Projekts
2. download des Flash oder EEprom Blocks
3. Programmierung des Chips
4. ersetzen des Chips durch ein neues
5. weiter mit Punkt 2 oder Punkt 1

# Beispiele und Sourcen

Zum Lieferumfang dieses Systems gehören die WIN32 DLL ***UPP1X_DLL.DLL***, ein ausführliches Delphi/Pascal Programm, das als EXE mitgeliefert wird sowie ein Visual Basic und ein C++ Programm. Die Quelltexte aller Programme sind ebenfalls enthalten.

Dem geübten Windows Programmierer dürfte es deshalb nicht allzu schwer fallen, ein eigenes, den Verhältnissen angepasstes Programm zu erstellen.

## *Import der DLL in 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);
```

Zumindest beim ersten Kontakt mit der DLL muss die Funktion ***AbortAll*** aufgerufen werden!

```
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);
                       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 der DLL in 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
```

***Bitte beachten: alle integer sind 32bit Typen***

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 der DLL in C++*

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

#define DLLImport   __declspec( DLLimport )

enum ProgDLL_Result
        {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};
```

***Achtung: alle integer sind 32bit Typen***

Die folgende List ist nicht komplett…

```
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 );

// .....
  }
```

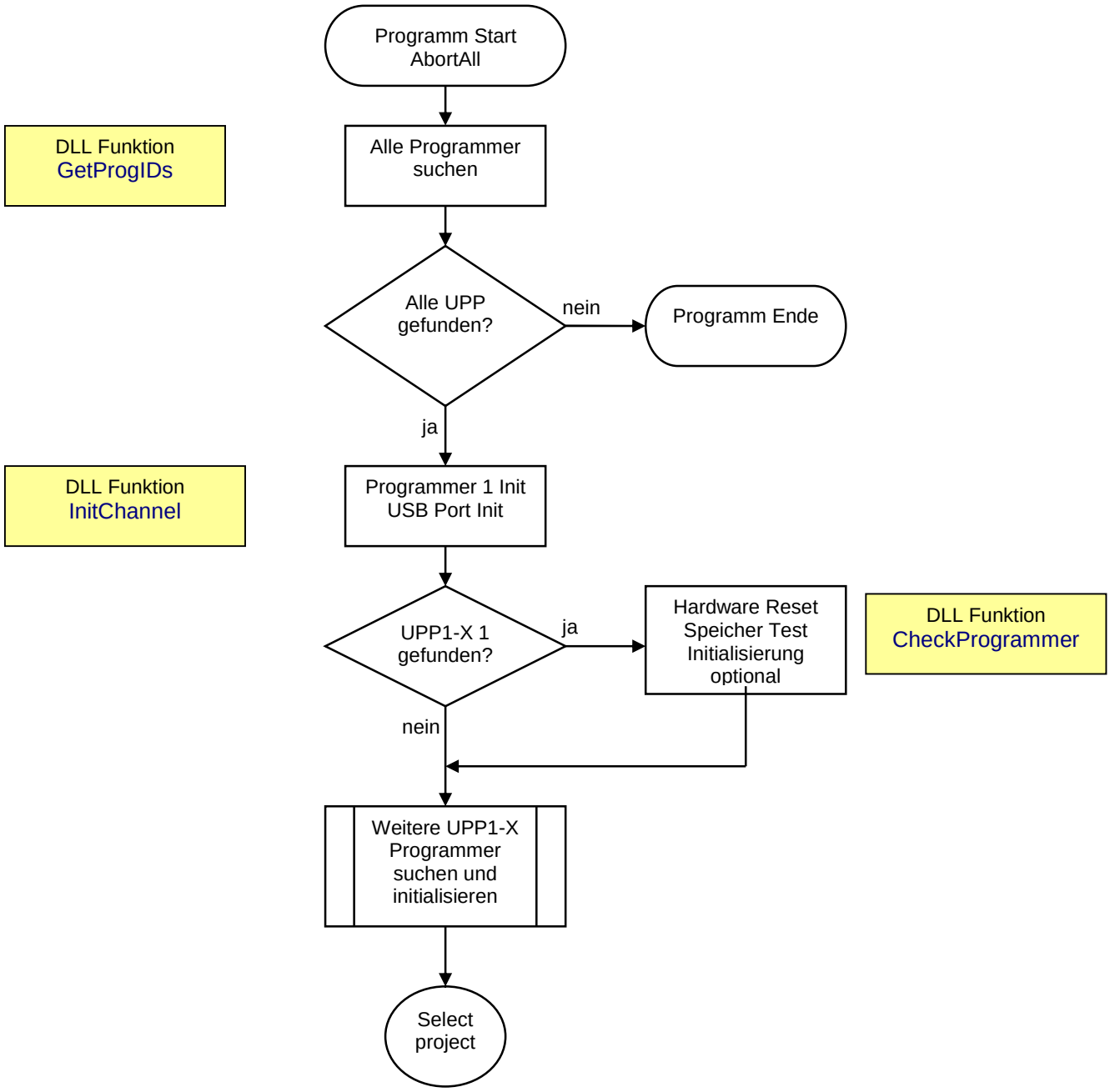
# Fluss Diagramme

Zum besseren Verständniss der Vorgehensweise beim Einsatz der E-LAB UPP1X DLL in der Produktion sind drei Diagramme vorhanden: 1. Initialisierung 2. Projekt Auswahl 3. Produktion.

## *Initialisierung*

Die UPP-Programmer müssen an dem Testrechner angeschlossen sein (USB Port). Auf dem Testrechner muss ein 32bit USB-fähiges Windows laufen (XP/WIN7/WIN8). Die DLL „UPP1X_DLL.DLL“ muss sich in dem gleichen Verzeichnis befinden, indem sich auch das Anwender Steuerprogramm für die UPPs befindet. Zumindest beim ersten Kontakt mit der DLL muss die Funktion ***AbortAll*** aufgerufen werden!

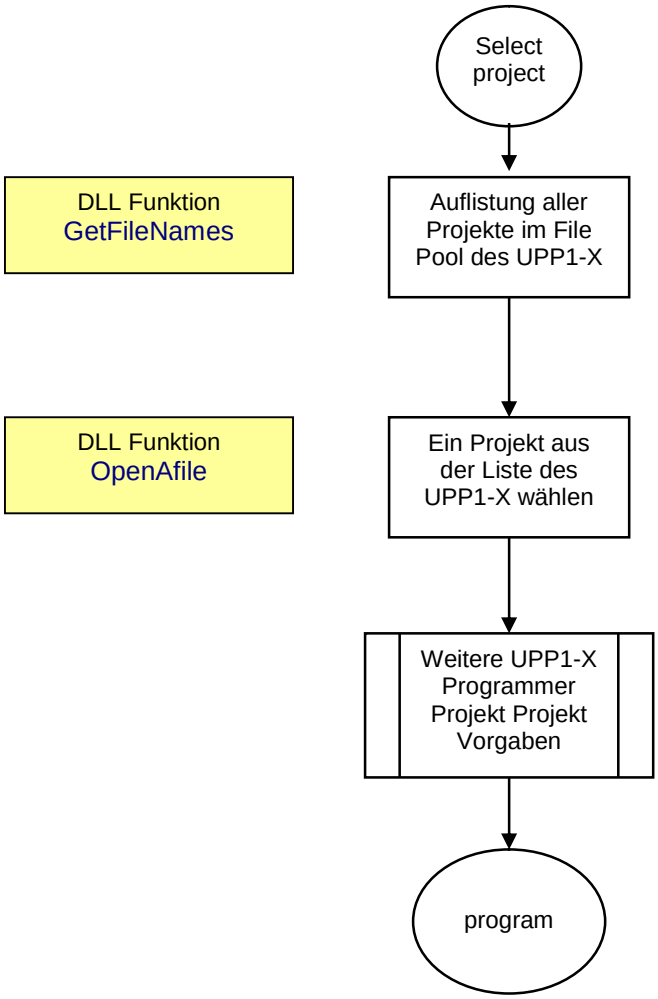
Beim Starten des Steuerprogramms muss dann nach den angeschlossenen Programmern gesucht werden und diese in Channels gemappt werden. Wird ein UPP-Programmer durch die DLL gefunden, erfolgt ein Hardware Reset auf diesen UPP. Der UPP1-X meldet einen eventuellen Speicherfehler über die DLL an das Steuerprogramm. Damit ist die Initialisierung abgeschlossen.



# *Select Project*

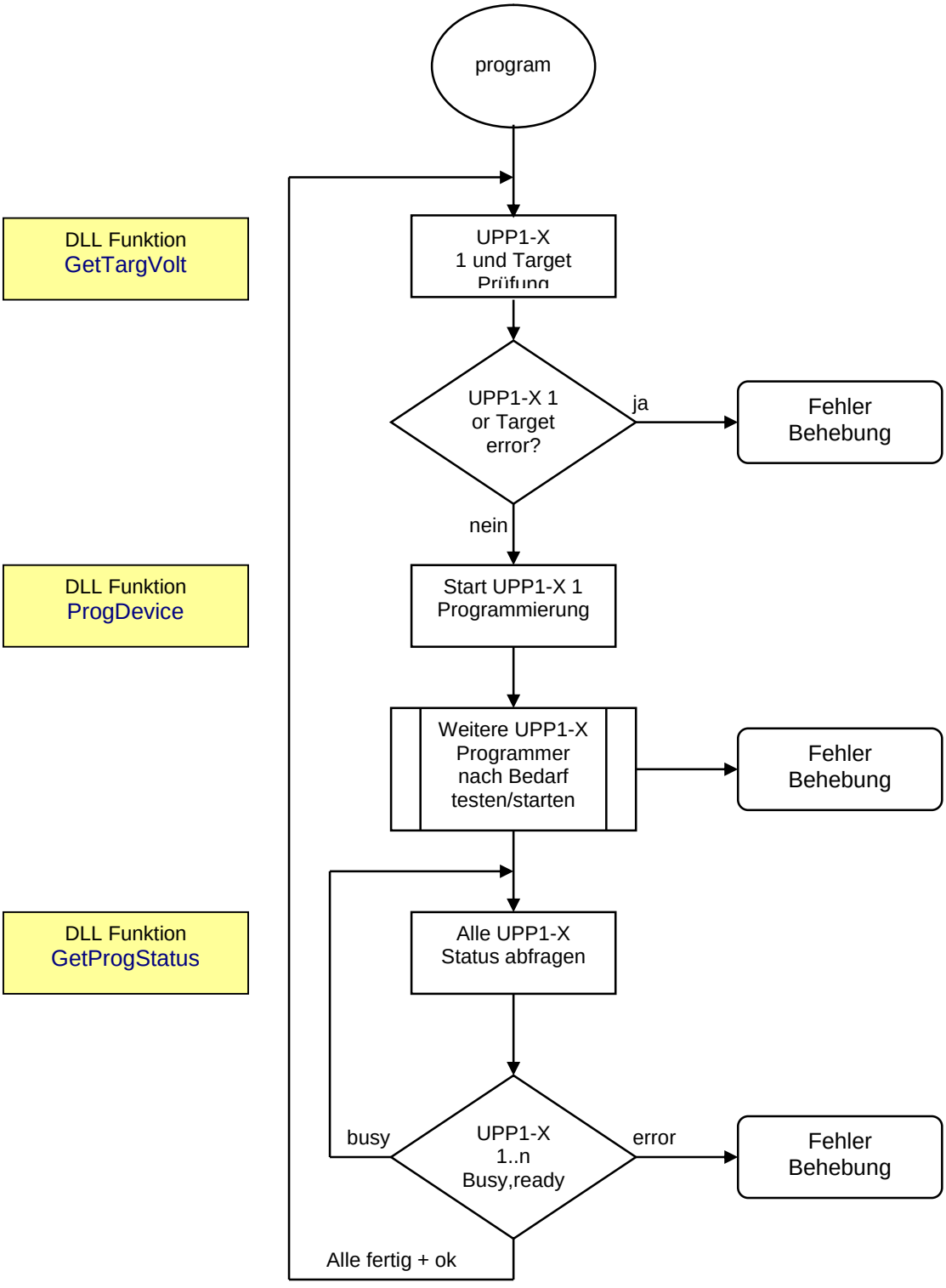
Die UPP1-X Programmer haben ein Flash File System und behalten einmal geladene Projekte auch nach Abschaltung in ihrem Flash Speicher.

Um ein Programmiervorgang starten zu können, muss ein Projekt ausgewählt werden.



Produktion
Der Programmiervorgang muss für jeden einzelnen angeschlossenen UPP-Programmer auch einzeln gestartet werden. Dabei gibt die Startfunktion als Ergebnis schon ein generelles OK oder Ausfall zurück. Ein Ausfall kann hierbei ein nicht vorhandener oder defekter Programmer sein. Unmittelbar vor dem Programmierzyklus sollten deshalb alle relevanten Programmer mit der Funktion „GetTargVolt“ auf Funktion und die Ziel-CPU auf vorhandene Betriebsspannung geprüft werden.

Der eigentliche Programmierstatus wird nach dem Start aller Programmer durch eine generelle Poll Funktion abgeholt und analysiert. Wenn alle aktiven Programmer entweder ein „progDone“ oder eine Fehlermeldung abgegeben haben, ist der Programmierzyklus zu Ende und es kann ein neuer gestartet werden.



**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**