E-LAB

Production Programmer System

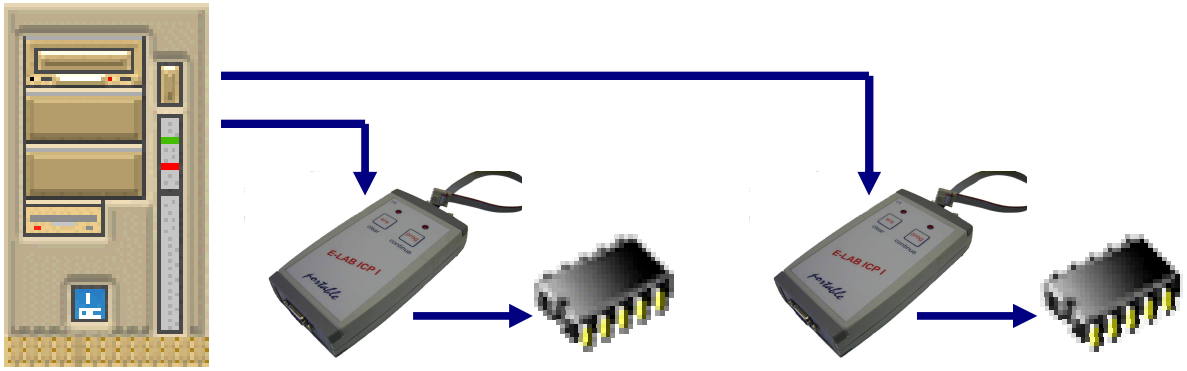
AVR *ProgDLL*

- **Produktions Programmiersystem für Atmel AVR CPUs**
- **Bis zu 4 simultane Programmierungen**
- **Steuerung über WIN32 DLL**
- **Steuerungs Programm vom Anwender selbst erstellbar**
- **Eine serielle Schnittstelle pro Programmer**
- **Alternative 1x USB Schnittstelle**
- **Durch Einsatz der E-LAB ICP Programmer flexible und schnelle InCircuit Programmierung**
- **Optionale Stückzahlen Limitierung und Passwortschutz**
- **Durch verschlüsselte Dateien ist ein re-Ingeneering ausgeschlossen**

second Edition Mai 2003

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

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 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 4 an COMports bzw. 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 4 E-LAB ICP-V24 oder ICP-USB Programmern (nicht jedoch ICPII).

Die Einkanal Version der DLL ist kostenlos und in der ICP-Installation enthalten.
Die 2-Kanal Version der DLL kostet euro 200.- ohne ICP-Programmer
Die 4-Kanal Version der DLL kostet euro 400.- ohne ICP-Programmer

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 ProgDLL

Die E-LAB Programmer DLL dient als Bindeglied zwischen Test-Automat (oder PC) und ICP-Programmern in der Produktion von Boards, die mit Atmel AVRs bestückt sind. Die DLL kann bis zu 4 ICP-Programmer gleichzeitig steuern und ist daher sehr gut für die Nutzen Fertigung bzw. Programmierung geeignet.

Das System besteht aus einer Windows DLL (WIN98..WIN2000/XP) und mehreren COMports bzw. USB mit daran angeschlossenen ICP Programmern.

Der Test Automat 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.

Die ACCUs der angeschlossen ICP-Programmer werden durch die geöffneten COM/USB Ports kontinuierlich geladen, so dass ein Ladegerät an die ICPs normalerweise nicht angeschlossen werden muss. Sollte das System länger als 1..2 Wochen abgeschaltet werden, so sollten die ICPs mit ihren Ladegeräten verbunden werden.

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-HEX file, EEprom –HEX file, CPU-Typ, Lockbits, Fusebits etc). Eine Manipulation bzw. fehlerhafte Einstellung der Programmer durch die Fertigung ist damit ausgeschlossen. Optional ist hier auch eine Stückzahlen Begrenzung und sogar ein Zielsystem abhängiger Password Schutz möglich. (ausführliche Beschreibung im ICP Handbuch).

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 ICP-Programmer, Programmieren starten, Ergebnis auswerten. Alle Funktionen liefern als Funktions Ergebnis einen Text (Pchar) und ein Integer (res) als Aufzählungstyp (Enumeration) zurück.

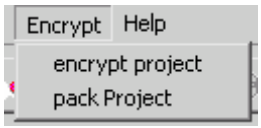
Aufzählungstyp (Enumeration) der Funktions Ergebnisse:

resNone, 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

Hierbei hat z.B. „resNone“ den Wert 0 und „progFound“ den Wert 2. ProgBusy = 4

Gepackte oder verschlüsselte Dateien

Die DLL kann zwei Datei Typen handhaben. Beide Datei Versionen müssen durch das E-LAB PC-Programm ***AVRprog.exe*** erstellt werden.



Gepackte Datei ist eine binäre Datei, die alle notwendigen Informationen enthält, jedoch *keine* Verschlüsselung, Passwörter oder Stückzahlen Begrenzung. Sie darf deshalb im Gegensatz zu der verschlüsselten Version (encrypted) *nicht* durch „AddNewFile“ im Datei Pool angemeldet werden.
Die Datei Endung ist immer **.pack**

Verschlüsselte Datei: ist eine binäre Datei, die alle notwendigen Informationen enthält, *inklusive* Verschlüsselung, Passwörter und Stückzahlen Begrenzung. Sie muss deshalb *immer* durch „AddNewFile“ im Datei Pool angemeldet werden. Die Datei Endung ist immer **.encr**

Achtung:

Durch die notwendige Anpassung an VB und diverse Windows Versionen musste das Interface geändert werden. Alle Funktionen wurden zu Prozeduren geändert. Der ehemalige Funktions Rückgabewert „PChar“ wurde in „resStr“ geändert. Alle DLL Funktionen kopieren jetzt das textuelle Ergebnis eines Aufrufs in den dafür bereitzustellenden Speicher der aufrufenden Applikation!

Die **Interface Funktionen** der DLL sind folgende (in Pascal Notation) :

```
procedure AddNewFile(FileName: PChar; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, invFile, invFName, invPassword, FileExist, resOk

procedure DeleteFile(FileName: PChar; var res : integer; resStr : PChar); stdcall;external 'ProgDLLN.dll';
// res = progBusy, invFName, notFound, errProtected, resOk

procedure GetFile(index : integer; var res : integer; resStr : PChar); external 'ProgDLLN.dll';
// res = progBusy, notFound, resOk

procedure GetPasswd(var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, resOk

procedure SetChannel (var Channel : integer; ComName : PChar; var res : integer; resStr : PChar); stdcall;
                      external 'ProgDLLN.dll'

// res = progBusy, noProg, errProgType, progFound
procedure CheckProgrammer(Channel: integer; var res : integer; resStr : PChar); stdcall;
                          external 'ProgDLLN.dll'
// res = progBusy, noProg, errProgType, progFound, MemError

procedure EraseProgrammer(Channel: integer; var res : integer; resStr : PChar); stdcall;
                          external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk

procedure GetProjName(Channel : integer; var Age, res : integer; resStr : PChar); stdcall;
                      external 'ProgDLLN.dll';
// res = progBusy, noProg, errProgType, resOk

procedure DownloadFile(ProjName: PChar; BurnCycles: Integer; Channel : Integer; var res : integer;
                       resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, limitExc, errProgType, progProtected, dwnLoadErrP, dwnLoadErrF, invFile, invFName,
//       dwnLoadErrE, dwnLoadErr, resOk

procedure GetACCUstate(Channel: integer; var ACCU, res : integer; resStr : PChar); stdcall;
                       external 'ProgDLLN.dll';
// res = progBusy, noProg, errProgType, resOk

procedure GetTargVolt(Channel: integer; var Volt, res : integer; resStr : PChar); stdcall;
                      external 'ProgDLLN.dll';
// res = progBusy, noProg, errProgType, resOk

procedure CheckDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = resNone, progBusy, noProg, errProgType, errPwrDown, errSignature,
//       errNotEmpty, errProtected

procedure ProgDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure EraseDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure VerifyDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

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

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

```
procedure ProgFuses(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure ProgLockBits(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure VerifyFlashOnly(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure VerifyEEpromOnly(Channel: Integer; var res : integer; resStr : PChar); stdcall; external
'ProgDLLN.dll';
// res = progBusy, noProg, resOk  rest of states must be done with "GetProgStatus"

procedure GetProgStatus(Channel: integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = resNone, limitExc, eraChip, prgEEp, prgFlash, progDone, errPwrDown,
//       errSignature, errProtected, errNotEmpty, errVerify

procedure DownLoadBlockF(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
                         resStr : PChar); stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErrF, resOk

procedure DownLoadBlockE(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
                         resStr : PChar);      stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErrE, resOk

procedure DownProgBlock256(Block : Pointer; dest, Channel : Integer; var res : integer; resStr : PChar);
                           stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErrE, errPwrDown, errSignature, errProtected, resOk

procedure UpLoadBlockF(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
                       resStr : PChar);       stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErr, resOk

procedure UpLoadBlockE(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
                       resStr : PChar);      stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErr, resOk

procedure ReadBackF(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
                    resStr : PChar); stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErr, resOk

procedure ReadBackE(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer; resStr : PChar);
                    stdcall;  external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErr, resOk

procedure GetFuses(Channel : integer; selfuse : char; var fuse : integer; var res : integer; resStr : PChar); stdcall;
                   external 'ProgDLLN.dll';
// res = progBusy, noProg, memErr, resOk

procedure ReleaseDevice(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk

procedure EnterProgMode (Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resNone, resOk

procedure AbortAll; stdcall; external 'ProgDLLN.dll'
```

JTAG Boundary SCAN Übersicht

Neuere E-LAB Programmer des Typs ICP-V24 und ICP-USB unterstützen auch die Boundary scan Funktion der mega AVR CPUs. Boundary Scan ist sowohl in der AVR CPU als auch im ICP Programmer als auch in dieser DLL vollkommen von den Programmierfunktionen getrennt. Die einzige gemeinsame Funktion sowohl für das Programmieren als auch für das Boundary Scan ist ***SetChannel***.

Für das Boundary Scan ist keine Encryption Datei erforderlich und damit auch kein Download. Alle notwendigen Informationen werden durch die Funktion ***JTAGBoundaryOpen*** übergeben.

Zum Board Test mittels Boundary Scan gibt es nur zwei Funktionen, nämlich

1. JTAGchainWrite, schreibt alle bits der Scan Chain
2. JTAGchainRead, liest alle bits der Scan Chain

Hiermit werden alle Bits der Chain gelesen oder geschrieben. Eine Auswertung des Ergebnisses von Read erfolgt nicht. Dies ist Sache der Applikation.

```
procedure JTAGBoundaryOpen(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;
                           var res : integer; resStr : PChar);  stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, progNoJTAG, progNoSupply, errPwrSupply, errSignature, resOk

procedure JTAGchainWrite(Channel : Integer; Block : Pointer; var res : integer; resStr : PChar);  stdcall;
                         external 'ProgDLLN.dll'
// res = progBusy, noProg, , resOk

procedure JTAGchainRead(Channel : Integer; Block : Pointer; var res : integer; resStr : PChar); stdcall;
                        external 'ProgDLLN.dll'
// res = progBusy, noProg, , resOk

procedure JTAGBoundaryClose(Channel: integer; var res : integer; resStr : PChar); stdcall;
                           external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
```

Funktions Details

Im folgenden wird der Begriff „COMport“ für den seriellen Schnittstelle Namen bzw. USBport des Steuerrechners benutzt. Der Begriff „Channel“ steht für einen bestimmten seriellen Port des Systems (0..3) und damit indirekt für einen speziellen am Host angeschlossenen ICP-Programmer. Mit der DLL Funktion „SetChannel“ wird einem Channel ein bestimmtes COM/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.

Note:
Bei der Einkanal DLL muss in „Channel“ immer 0 stehen, bei der 2 Kanal Version kann 0 oder 1 übergeben werden, bei der 4 Kanal Version 0..3
COMport kann „COM1“ bis „COM8“ oder „USB“ sein.

Verwaltungs Funktionen

Diese Funktionen können immer aufgerufen werden, ohne dass ein Programmer am System angeschlossen sind.

procedure **AddNewFile**(FileName: PChar; var res : integer; resStr : Pchar);
// res = progBusy, invFile, invFName, invPassword, FileExist, resOk
Eine verschlüsselte Datei (Projekt) wird dem Datei Pool hinzugefügt. Ist die Datei z.B. Password geschützt, und der Rechner entspricht nicht dem internen Datei-Password, wird die Datei mit dem Fehlercode „invPassword“ zurückgewiesen. Der Filenamen muss auch den **Pfad** und die Extension **.encr** enthalten.
Achtung: Diese Funktion darf **nicht** für gepackte Dateien angewendet werden.

procedure **DeleteFile**(FileName: PChar; var res : integer; resStr : Pchar);
// res = progBusy, invFName, notFound, errProtected, resOk
Eine Datei/Projekt wird aus dem Pool entfernt, wobei die Datei selbst nicht gelöscht wird. Hat dieses Projekt eine Stückzahlen Limitierung, wird eine Löschung mit „errProtected“ zurückgewiesen.

procedure **GetFile**(index : integer; var res : integer; resStr : Pchar);
// res = progBusy, notFound, resOk
Diese Funktion dient zum Auslesen bzw. Auflisten der im Pool vorhandenen Projekte. Die Applikation inkrementiert solange den Übergabe Wert „index“, bis das Funktions-Ergebnis ungleich „resOk“ ist. Bei resOk zeigt „PChar“ auf einen String, der den Projekt-Namen enthält, der mit dem „index“ im Pool gelistet ist.

procedure **GetPasswd**(var res : integer; resStr : Pchar);
// res = progBusy, resOk
Diese Funktion liefert als Ergebnis das Passwort, das Packungs Programm „AVRprog.exe“ benötigt, um Passwort geschützte Dateien speziell für diesen Ziel-Rechner erstellen zu können. In den meisten Fällen (innerbetriebliche Anwendung) ist weder eine Stückzahlen Limitierung und/oder ein Passwort sinnvoll oder notwendig. Bei Fernost Produktionen ist dieses Feature u.U. sehr segensreich. ☺

Initialisierung

Diese Funktionen werden vor dem Programmier Start einmal aufgerufen. Die ersten beiden sind absolut notwendig. Der Download erfolgt nur im Bedarfsfall.

procedure **SetChannel** (var Channel : integer; ComName : PChar; var res : integer; resStr : Pchar);
// res = progBusy, progFound, noProg, errProgType
Diese Funktion sucht mit der seriellen Schnittstellen „ComName“ nach einem ICP-Programmer. In „Channel“ wird eine Zahl (0..3) vorgegeben, mit der der Programmer, falls gefunden, zukünftig selektiert werden muss. Im Erfolgsfall ist „Channel“ damit belegt und kann nur wieder mit „AbortAll“ freigegeben werden. Ein scannen der Ports erfolgt nicht. Der Programmierer ist selbst dafür verantwortlich, dass alle gewünschten Ports vorhanden sind und dass daran auch Programmer angeschlossen sind. Diese Funktion muss pro Programmer einmal aufgerufen werden.

Im Parameter „Channel“ wird im Erfolgsfall (res = progFound) die Kanal Nummer zurückgegeben, ansonsten eine -1. Die Funktion erzeugt ein Hardware Reset auf dem angeschlossenen Programmer.

procedure **CheckProgrammer**(Channel: integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, errProgType, progFound, MemError
Wurde ein ICP-Programmer mit „SetChannel“ gefunden, kann dieser jederzeit mit dieser Funktion getestet werden. Ein Download erfolgt jedoch nicht. Beim Funktionsaufruf bezeichnet der Parameter „Channel“ den zu suchenden bzw. zu prüfenden Programmer. Die Funktion erzeugt ein Hardware Reset auf dem ausgewählten Programmer. Wenn „MemError“ angezeigt wird muss ein neuer Download erfolgen.

procedure **EraseProgrammer**(Channel: integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
Wurde ein ICP-Programmer mit „SetChannel“ gefunden, kann er mit dieser Funktion gelöscht werden. Ein Download erfolgt jedoch nicht. Beim Funktionsaufruf bezeichnet der Parameter „Channel“ den zu löschenden Programmer. Diese Funktion setzt alle Daten im Programmer auf nicht initialisiert und löscht alle Checksummen und ein evtl. vorhandenes Passwort.

procedure **GetProjName**(Channel : integer; var Age, res : integer; resStr : Pchar);
// res = progBusy, noProg, errProgType, resOk
Diese Funktion dient zum Feststellen des aktuell in einem ICP-Programmer gespeicherten Projekts. Im Parameter wird die Nummer des gewünschten Programmers übergeben. Das Ergebnis der Funktion ist der im ICP eingespeicherte Projektnamen. Im Parameter „Age“ wird das Erstellungsdatum der gepackten Original Datei Datei zur weiteren Information enthalten.

procedure **GetACCUstate**(Channel: integer; var ACCU, res : integer; resStr : Pchar);
// res = progBusy, noProg, errProgType, resOk
Wenn erfolgreich, gibt die Funktion als Resultat und im Parameter „ACCU“ den ACCU Ladezustand in Prozent zurück.

procedure **GetTargVolt**(Channel: integer; var Volt, res : integer; resStr : Pchar);
// res = progBusy, noProg, errProgType, resOk
Wenn erfolgreich, gibt die Funktion als Resultat und im Parameter „Volt“ die Board Spannung (CPU) in 10mV Schritten zurück.

procedure **DownloadFile**(ProjName: PChar; BurnCycles: Integer; Channel : Integer; var res : integer;
resStr : Pchar);
// res = progBusy, limitExc, errProgType, progProtected, dwnLoadErrP, dwnLoadErrF, invFile, invFName,
// dwnLoadErrE, dwnLoadErr, 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 ICP Programmer ACCU-gepuffert sind, muss nur bei einem Projekt Wechsel für diesen Programmer neu herunter geladen werden.

Die Fehlermeldungen dwnLoadErrP, dwnLoadErrF und dwnLoadErrE dienen im Fehlerfall zur Information, wo das Problem aufgetreten ist (Download Parameter, Download Flash oder Download EEprom). Der Fehler „limitExc“ tritt auf, wenn versucht wird, eine Stückzahlen begrenzte Datei herunterzuladen, derer Begrenzung inzwischen erreicht wurde. (Encrypted Files)

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

Encrypted Files
Der Parameter **ProjName** ist derselbe wie **FileName** in der Funktion **AddNewFile** aber ohne Pfad. Die File Extension (.encr) muss mit angegeben werden,. Der Parameter **BurnCycles** ist nur von Bedeutung, wenn das Projekt Stückzahlen limitiert ist. Wenn die Begrenzung insgesamt bei 10000 liegt und es sind 4 ICP angeschlossen, die jeweils die gleiche Stückzahl programmieren sollen, so muss der Parameter „BurnCycles“ auf 10000/4 = 2500 gesetzt werden. Jeder Programmer kann jetzt 2500 Chips programmieren.
Ohne eine Limitierungs Vorgabe innerhalb des Projekts sollte der Parameter auf 0 gesetzt werden.

Packed files
Der Parameter **ProjName** muss auch den Pfad und File Extension einschliessen(.pack). Der Parameter **BurnCycles** ist ohne Bedeutung und sollte auf 0 gesetzt werden. Das File wird direkt in den Programmierer geladen ohne Beteiligung des File Pools.

File Extensions
Diese ***müssen*** mit angegeben werden, so dass die DLL eindeutig zwischen den beiden Datei Typen unterscheiden kann.

Produktion

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

procedure **EraseDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
Mit dieser Funktion wird ein Chip Löschvorgang gestartet. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs den Status abzuholen.

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 ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs 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 gesetzt sind.

procedure **ProgFlash**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// 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 ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs den Status abzuholen.

procedure **ProgEEprom**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// 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 ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs den Status abzuholen.

procedure **ProgFuses**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
Diese Funktion startet die Fuse-only Programmierung. Es werden nur die Fuses programmiert. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs den Status abzuholen.

procedure **ProgLockBits**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
Diese Funktion startet die LockBits-only Programmierung. Es werden nur die LockBits programmiert. Ist das Ergebnis „res“ = resOk, war der Start erfolgreich. Sind alle relevanten ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs den Status abzuholen.

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 ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs 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 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 ICP-Programmer gestartet, muss die Applikation jetzt kontinuierlich mit der Funktion „GetProgStatus“ pollen, um für alle gestarteten ICPs 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 gesetzt sind.

procedure **GetProgStatus**(Channel: integer; var res : integer; resStr : Pchar);
// res = resNone, limitExc, eraChip, prgEEp, prgFlash, progDone, errPwrDown,
// errSignature, errProtected, errNotEmpty, errVerify, MemError
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. progDone : der Vorgang inkl. Verify ist komplett abgeschlossen.

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, errProgType, 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 **DownLoadBlockF**(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
resStr : Pchar);
procedure **DownLoadBlockE**(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
resStr : Pchar);
procedure **UpLoadBlockF**(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
resStr : Pchar);
procedure **UpLoadBlockE**(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
resStr : Pchar);

// res = progBusy, noProg, dwnLoadErrE, dwnLoadErrF, dwnLoadErr, resOk
Diese vier Funktionen ermöglichen eine Manipulation des aktuellen Flash und EEprom Inhaltes im Programmer. Damit ist es möglich während der Produktion kontinuierlich z.B. Seriennummern fortzuschreiben. Jeder Download wird für den nächsten Programmiervorgang gültig. Die Blockgrösse 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 4 Funktionen setzen einen Programmer **Typ ICP-V24** oder **ICP-USB** mit einer Firmware Version **6.0** oder höher voraus.

procedure **UpLoadBlockF**(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
resStr : Pchar);
procedure **UpLoadBlockE**(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
resStr : Pchar);

// res = progBusy, noProg, dwnLoadErrE, dwnLoadErrF, dwnLoadErr, resOk

Mit diesen zwei Funktionen kann der aktuelle Flash oder EEprom Inhalt der Target CPU ausgelesen werden, vorausgesetzt, der Programmer und diese CPU sind nicht auslesegeschützt.

Diese 2 Funktionen setzen einen Programmer **Typ ICP-V24** oder **ICP-USB** mit einer Firmware Version **6.1** oder höher voraus.

procedure **DownProgBlock256**(Block : Pointer; dest, Channel : Integer; var res : integer; resStr : PChar);
stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, dwnLoadErrE, errPwrDown, errSignature, errProtected, resOk

Der Zweck dieser Funktion ist es zu ermöglichen, dass kleine Teile des Flashs nachträglich mit einem anderen Inhalt versehen werden können. Diese Überprogrammierung ist 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 mit einem 256 Byte Block gearbeitet werden. Dieser Block muss auch einer 256 Byte Grenze liegen. Das stellt in der Praxis kein Problem dar. Es gibt mehrere Möglichkeiten einen solchen Block zu erstellen.

1. Einen Speicherblock von 256 Bytes mit $FF füllen. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlock256.
2. Den Speicherblock von 256 Bytes mit dem Inhalt des Original Files füllen. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlock256.
3. Einen Speicherblock von 256 Bytes durch einen Upload des Programmer Speichers oder direkt des Flash der CPU füllen. Dann die gewünschten Bytes mit den neuen Werten überschreiben. Aufruf der Funktion DownProgBlock256.

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.

Block ist ein Pointer der auf die 256 Byte Quelle/Buffer zeigen muss. **Dest** ist die Ziel Adresse des 256 Byte Blocks im Flash Speicher der CPU. Er muss immer auf den Anfang eines 256 Byte Blocks im Flash zeigen.

Die CPU muss schon programmiert sein (Flash und Fuses) und darf nicht auslesegeschützt sein.

Diese Funktion setzt einen Programmer **Typ ICP-V24** oder **ICP-USB** mit einer Firmware Version **6.6** oder höher voraus.

procedure **GetFuses**(Channel : integer; SelFuse : char; var Fuse : integer; var res : integer; resStr : PChar);
stdcall; external 'ProgDLLN.dll';
// 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	= FuseBits low
H	= FuseBits high
E	= FuseBits extended
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, 0, 1, 2, 3) macht es natürlich nur Sinn solche Fuses auszulesen, die auch in der CPU vorhanden sind.

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 ComPorts werden geschlossen.

Es muss jetzt eine komplette Neu-Initialisierung mit „SetChannel“ und „CheckProgrammer“ erfolgen.

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

JTAG Boundary SCAN Funktionen

Die Boundary Scan Chain besteht aus einer Anzahl von Bits, wobei jedes einzelne Bit einem internen Peripherie Bit der AVR CPU zugeordnet ist. Diese Bits können entweder Port Pins sein oder Bits in Steuerregister der Peripheri Einheiten der CPU, wobei nicht jedes Bit direkt eine Verbindung zu einem Pin haben muss. So sind manche Bits z.B. zuständig für den AD-Wandler. Die CPU selbst, d.h. Register und Memory sind nicht im Zugriff der Scan Chain.

Die Applikation muss 2 Buffer zur Verfügung stellen, einen Read und einen Write Buffer. Die Grösse dieser Buffer sind abhängig von der Länge (Anzahl Bits) der Scan Chain. Am besten ist es man definiert jeweils ein Array von 0..255 of byte. Das ist ausreichend für alle CPU Typen.

Jedes Bit in diesen Buffern entspricht jetzt exakt einem Bit in der Scan Chain. Die Applikation kann jetzt einzelne Bits im Download Buffer manipulieren, einen Download starten und dann alle Bits mit einem Upload in den Upload Buffer zurücklesen. Hierbei ist zu beachten, dass zwar einzelne Bits im Download Buffer manipuliert werden können, aber der Download immer alle Bits der Chain betrifft. Eine gezielte direkte Änderung von einzelnen Bits in der Chain ist nicht direkt möglich, da immer alle Bits der Chain gleichzeitig beschrieben werden müssen.

Die in der DLL implementierten Scan Funktionen können die Chain komplett lesen und Schreiben, eine Auswertung kann nicht erfolgen. Das ist Sache der Applikation. Der Boundary Scan Teil in der DLL, im Programmer und auch in der CPU ist komplett getrennt vom Programmier Teil. Beide haben nichts miteinander gemeinsam. In der DLL ist nur die Funktion ***SetChannel*** für beide gemeinsam. Ein Encryption File Load etc. ist für das Scannen nicht erforderlich.

procedure **JTAGBoundaryOpen**(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;
var res : integer; resStr : Pchar);
// res = progBusy, noProg, progNoJTAG, progNoSupply, errPwrSupply, errSignature, resOk

Die Parameter CpuID und BoundaryLen sind dem Datenblatt der AVR CPU zu entnehmen. Beim mega16 z.B. ist CpuID = $001E9403 und BoundaryLen = 141 (Bits).
mega128 CpuID = $001E9702 BoundaryLen = 205 (Bits)

Die Parameter SupplyVolt und SupplyCurr bestimmen zusammen die Versorgung der externen Target CPU. Hat das Target eine eigene Versorgung, so sind beide Parameter auf 0 zu setzen. Soll der Programmer das Target versorgen, so ist in SupplyVolt die gewünschte Spannung anzugeben, z.B. 500 für 5Volt. In SupplyCurr wird der max. Strom übergeben, 30 oder 100 (mA). Ist die Target Versorgung durch den Programmer aktiviert, so muss der Programmer selbst mit seinem Ladegerät versorgt werden, oder es müssen 6..10V= an seiner Ladebuchse anliegen.

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

Download des lokalen Write-Buffers in die Scan Chain. Der Pointer ***Block*** muss in den Download Buffer zeigen.

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

Auslesen der Scan Chain aus dem Target in den lokalen Read-Buffer. Der Pointer ***Block*** muss in den Upload Buffer zeigen.

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

Schliesst das JTAG Scan Port der Target CPU.

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

Beispiele und Sourcen

Zum Lieferumfang dieses Systems gehören die WIN32 DLL ***ProgDLLN.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, 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);

procedure AddNewFile(FileName: PChar; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll'
procedure DeleteFile(FileName: PChar; var res : integer; resStr : Pchar); stdcall;external 'ProgDLLN.dll';
procedure GetFile(index : integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure GetPasswd(var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll'
procedure SetChannel (var Channel : integer; ComName : PChar; var res : integer; resStr : Pchar);
                        external 'ProgDLLN.dll'
procedure CheckProgrammer(Channel: integer; var res : integer; resStr : Pchar); stdcall;
                                external 'ProgDLLN.dll';
procedure EraseProgrammer(Channel: integer; var res : integer; resStr : Pchar); stdcall;
                                external 'ProgDLLN.dll';
procedure GetProjName(Channel : integer; var Age, res : integer; resStr : Pchar); stdcall;
                                external 'ProgDLLN.dll';
procedure DownloadFile(ProjName: PChar; BurnCycles: Integer; Channel : Integer; var res : integer;
                                resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure GetACCUstate(Channel: integer; var ACCU, res : intege; resStr : Pchar); stdcall;
                                external 'ProgDLLN.dll';
procedure GetTargVolt(Channel: integer; var Volt, res :integer; resStr :Pchar); stdcall; external 'ProgDLLN.dll'.

procedure ProgDevice(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure ProgFlash(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure ProgEEprom(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure ProgFuses(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure ProgLockBits(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';

procedure VerifyDevice(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure VerifyFlashOnly(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure VerifyEEpromOnly(Channel: Integer; var res : integer; resStr : Pchar); stdcall; external
'ProgDLLN.dll';

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

procedure DownProgBlock256(Block : Pointer; dest, Channel : Integer; var res : integer; resStr : PChar);
                                stdcall; external 'ProgDLLN.dll';
procedure DownLoadBlockF(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
                                resStr : Pchar); stdcall;  external 'ProgDLLN.dll';
procedure DownLoadBlockE(Block : Pointer; BlockSize, dest, Channel : Integer; var res : integer;
                                resStr : Pchar); stdcall;  external 'ProgDLLN.dll';
procedure UpLoadBlockF(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
                                resStr : Pchar); stdcall;  external 'ProgDLLN.dll';
procedure UpLoadBlockE(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer;
                                resStr : Pchar);  stdcall;  external 'ProgDLLN.dll';
procedure ReadBackF(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer; resStr : Pchar);
                                stdcall;  external 'ProgDLLN.dll';
```

procedure **ReadBackE**(Block : Pointer; BlockSize, source, Channel : Integer; var res : integer; resStr : Pchar);
stdcall; external 'ProgDLLN.dll';

procedure **ReleaseDevice**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resOk

procedure **EnterProgMode** (Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll';
// res = progBusy, noProg, resNone, resOk

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

procedure **AbortAll**; stdcall; external 'ProgDLLN.dll';

procedure **JTAGBoundaryOpen**(Channel: integer; CpuID, BoundaryLen, SupplyVolt, SupplyCurr : integer;
var res : integer; resStr : Pchar); stdcall; external 'ProgDLLN.dll';
procedure **JTAGchainWrite**(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar); stdcall;
external 'ProgDLLN.dll';
procedure **JTAGchainRead**(Channel : Integer; Block : Pointer; var res : integer; resStr : Pchar); stdcall;
external 'ProgDLLN.dll';
procedure **JTAGBoundaryClose**(Channel: integer; var res : integer; resStr : Pchar); stdcall;
external 'ProgDLLN.dll';

Import der DLL in Visual Basic

VB Interface

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

Public Declare Sub GetPasswd Lib "ProgDLLN.dll" (ByRef res As Integer, ByVal strRes As String)

Public Declare Sub AddNewFile Lib "ProgDLLN.dll" (ByVal FileName As String, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub DeleteFile Lib "ProgDLLN.dll" (ByVal FileName As String, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub GetFile Lib "ProgDLLN.dll" (ByVal Index As Integer, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub SetChannel Lib "ProgDLLN.dll" (ByRef Channel As Integer, ByVal ComName As String, ByRef res As Integer, ByVal strRes As String)

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

Public Declare Sub GetProjName Lib "ProgDLLN.dll" (ByVal Channel As Integer, ByRef Age As Integer, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub GetACCUstate Lib "ProgDLLN.dll" (ByVal Channel As Integer, ByRef ACCU As Integer, ByRef res As Integer, ByVal strRes As String)

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

Public Declare Sub DownloadFile Lib "ProgDLLN.dll" (ByVal ProjName As String, ByVal BurnCycles As Integer, ByVal Channel As Integer, ByRef res As Integer, ByVal strRes As String)

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

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

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

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

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

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

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

Public Declare Sub DownLoadBlockF Lib "ProgDLLN.dll" (ByVal Block As Pointer, ByVal BlockSize As Integer, ByVal dest As Integer, ByVal Channel As Integer, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub DownLoadBlockE Lib "ProgDLLN.dll" (ByVal Block As Pointer, ByVal BlockSize As Integer, ByVal dest As Integer, ByVal Channel As Integer, ByRef res As Integer, ByVal strRes As String)

Public Declare Sub DownProgBlock256 Lib "ProgDLLN.dll" (ByVal Block As Pointer, ByVal dest As Integer, ByVal Channel As Integer, ByRef res As Integer, ByVal strRes As String)

JTAG Boundary Scan

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

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

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

Import der DLL in C++

```
//////////////////////////////////////////////////////////////////////
//
// ICP Programmer DLL Header
//
// Author:          Uwe Mayer (Contronix)
// Date:  2003-03-12
//
//////////////////////////////////////////////////////////////////////


#define DllImport   __declspec( dllimport )


//////////////////////////////////////////////////////////////////////
// function call results

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
};
```

```
//////////////////////////////////////////////////////////////////////
// functions

typedef void (CALLBACK *RINT_STR_RINT_STR)(int&, char*, int&, char*);
typedef void (CALLBACK *RINT_STR)(int&, char*);
typedef void (CALLBACK *INT_RINT_RINT_STR)(int, int&, int&, char*);
typedef void (CALLBACK *STR_INT_INT_RINT_STR)(char*, int, int, int&, char*);
typedef void (CALLBACK *STR_RINT_STR)(char*, int&, char*);
typedef void (CALLBACK *INT_RINT_STR)(int, int&, char*);
typedef void (CALLBACK *NOTHING)();


STR_RINT_STR AddNewFile;
STR_RINT_STR DeleteFileB;
INT_RINT_STR GetFile;
RINT_STR GetPasswd;
RINT_STR_RINT_STR SetChannel;
INT_RINT_STR CheckProgrammer;
INT_RINT_STR EraseProgrammer;
INT_RINT_RINT_STR GetProjName;
STR_INT_INT_RINT_STR DownloadFile;
INT_RINT_RINT_STR GetACCUstate;
INT_RINT_RINT_STR GetTargVolt;
INT_RINT_STR CheckDevice;
INT_RINT_STR ProgDevice;
INT_RINT_STR EraseDevice;
INT_RINT_STR VerifyDevice;
INT_RINT_STR ProgFlash;
INT_RINT_STR ProgEEprom;
INT_RINT_STR ProgFuses;
INT_RINT_STR ProgLockBits;
INT_RINT_STR VerifyFlashOnly;
INT_RINT_STR VerifyEEpromOnly;
INT_RINT_STR GetProgStatus;

NOTHING AbortAll;
```

JTAG Boundary Scan

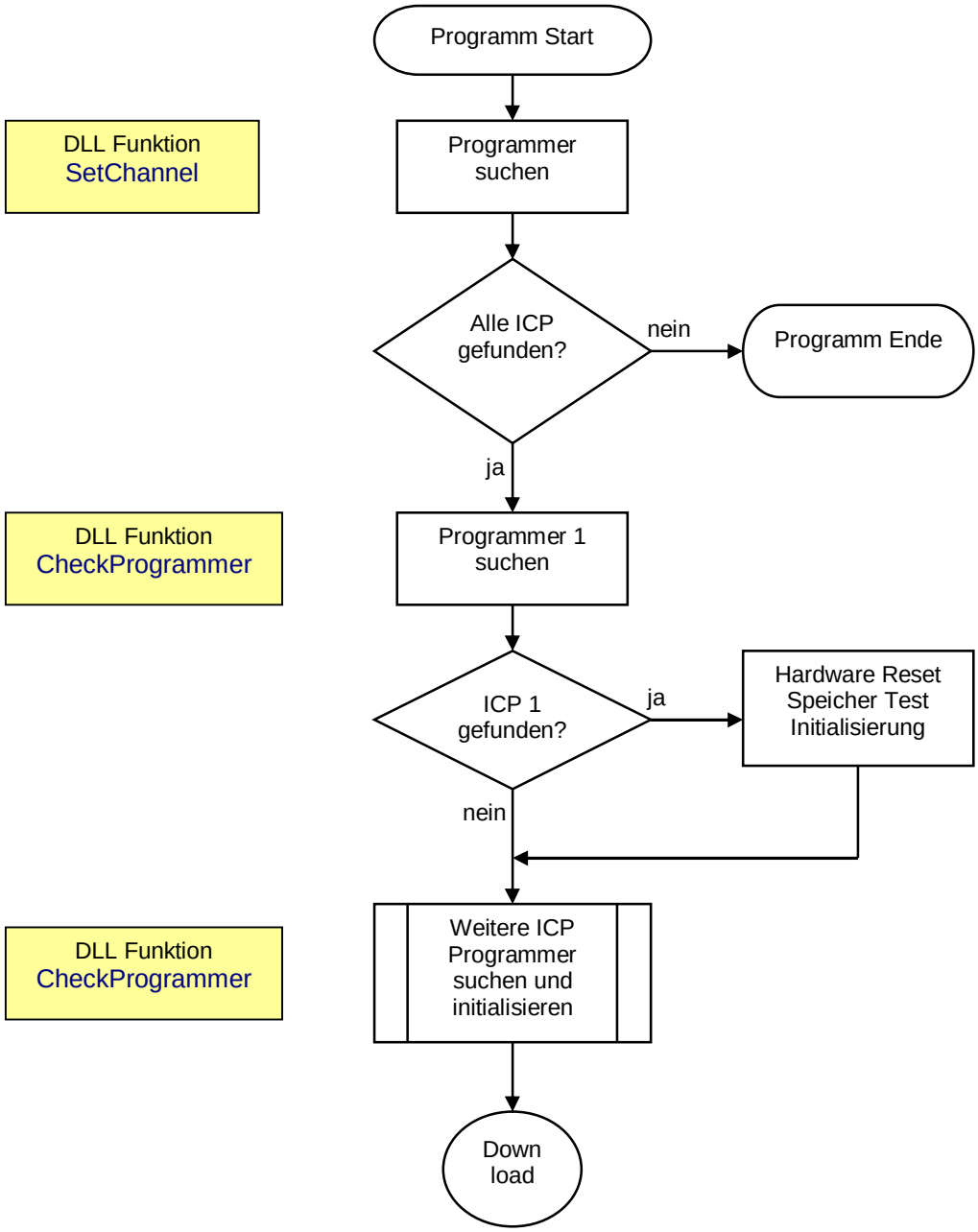
TBD

Fluss Diagramme

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

Initialisierung

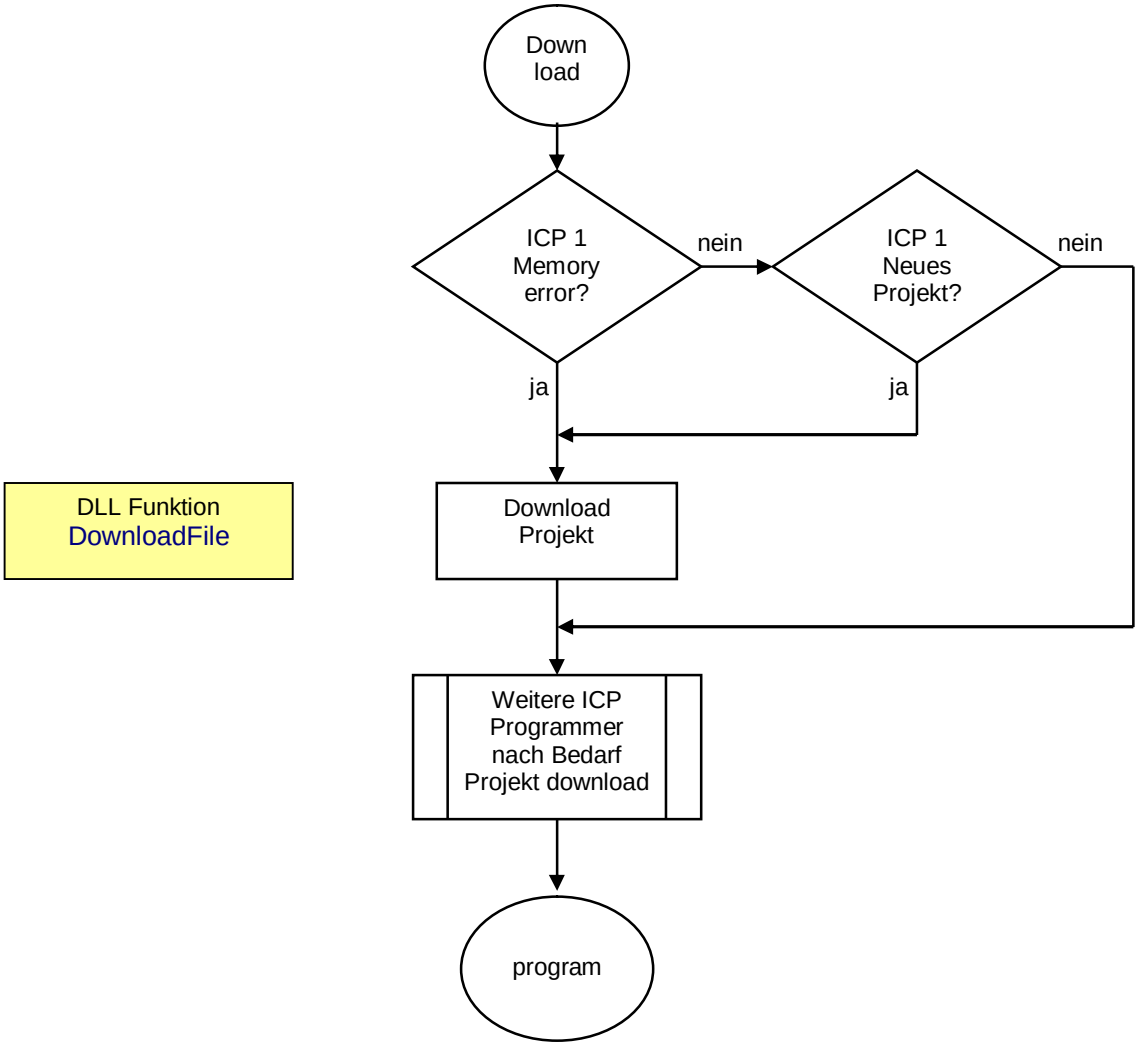
Die ICP-Programmer müssen an dem Testrechner angeschlossen sein (COM1..COM8). Auf dem Testrechner muss ein 32bit Windows laufen (WIN98/NT/WIN200). Die DLL „ProgDLLN.DLL“ muss sich in dem gleichen Verzeichnis befinden, indem sich auch das Anwender Steuerprogramm für die ICPs befindet.
Beim Starten des Steuerprogramms muss zuerst nach den angeschlossenen Programmern gesucht werden und diese in Channels gemappt werden. Wird ein ICP-Programmer durch die DLL gefunden, erfolgt ein Hardware Reset auf diesen ICP. Der ICP meldet einen eventuellen Speicherfehler über die DLL an das Steuerprogramm. Damit ist die Initialisierung abgeschlossen. Optional kann jetzt auch noch der Ladezustand des ICP-ACCUs abgefragt werden.



DownLoad

Ein Project Download in einen angeschlossenen Programmer ist nur notwendig, wenn entweder der Programmer während der Initialisierung einen Speichertest Fehler gemeldet hat, oder das geladene Projekt durch ein neueres ersetzt werden soll, oder ein anderes Projekt ab sofort vom Programmer verarbeitet werden soll.

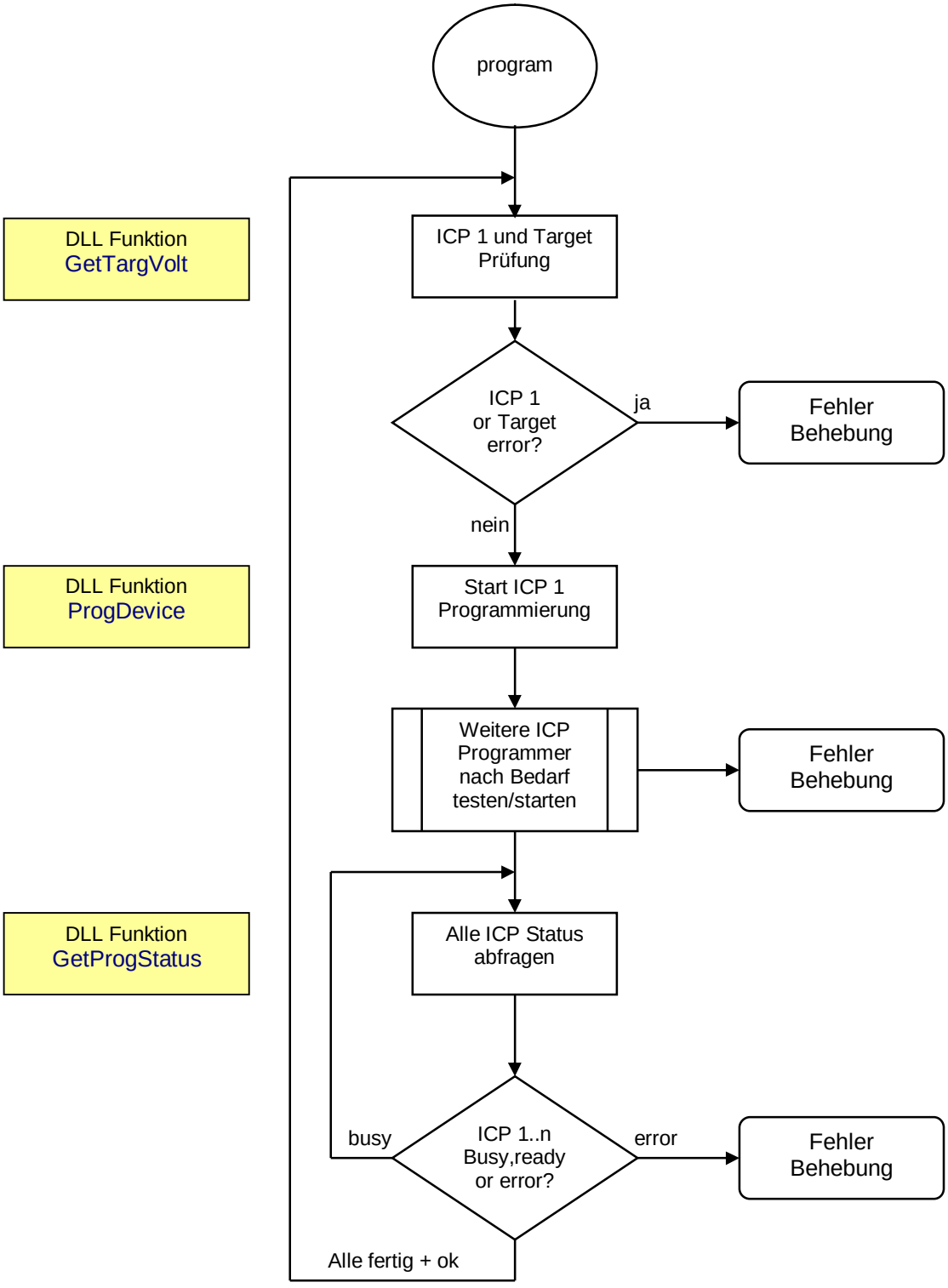
Die ICP Programmer sind ACCU gepuffert und behalten ein einmal geladenes Projekt auch nach Abschaltung in ihrem Speicher. Dieser Speicher wird nach jedem PowerOn bzw. Reset mittels Checksummen geprüft.



Produktion

Der Programmiervorgang muss für jeden einzelnen angeschlossenen ICP-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 Multiplexer

Der Nachteil des hier vorgestellten Produktions Systems ist, dass pro Programmer eine serielle Schnittstelle gebraucht wird.

Das Produktions System ***E-LAB Multiplexer*** benötigt nur eine serielle Schnittstelle bzw. ein USB-Port. Hiermit erfolgt eine weitere Entlastung des Test-Systems.

Infos über den ***E-LAB Multiplexer*** auf unserer Homepage.

www.e-lab.de

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**