E-LAB

Production Programmer System

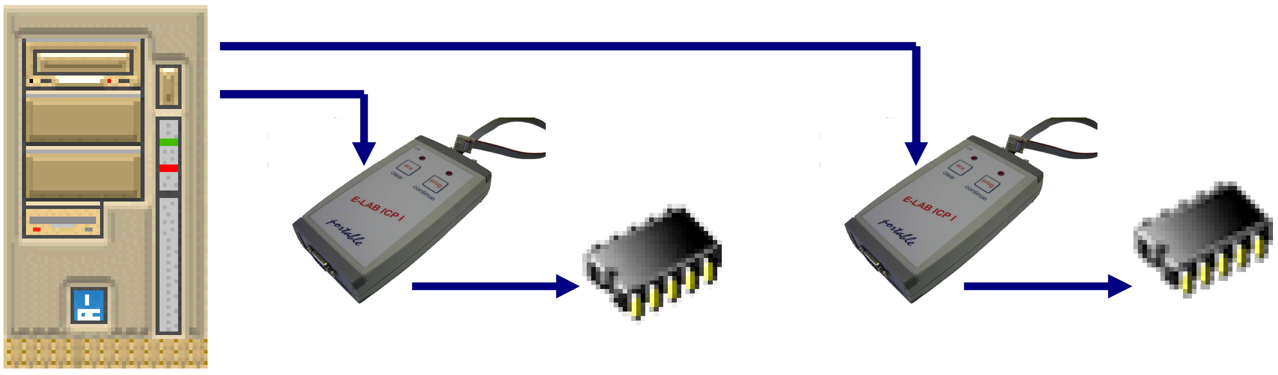
AVR *ProgDLL*

- Production Programming System for Atmel AVR CPUs
- Upto 4 simultaneous programmings
- Controlled through a WIN32 DLL
- Master control program supplied by the user
- One serial COMport used for each programmer
- Alternatively 1x USB port
- By the use of E-LAB ICP Programmer a flexible and
- fast InCircuit programming can be managed
- As an option there is device count limitation
  and password protection
- The projects are encrypted so re-engineering
  is absolutely impossible

second Edition May 2003

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

In-Circuit programming of Single Chips in mass production



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

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

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

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

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

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

A production programming system consists of the E-LAB Programmer-DLL and 1 to 4
E-LAB ICP-V24 or ICP-USB programmers (not for ICPII types).

The single channel version of the DLL is for free and is included in the ICP-Installation.
The 2-channel version of the DLL costs euro 200.- (US$200) no programmers included
The 4-channel version of the DLL costs euro 400.- (US$400) no programmers included

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

Production Programmer ProgDLL

The E-LAB programmer DLL serves an interface between a test automat (or PC) and ICP-Programmers in the production of boards which are equipped with Atmel AVRs. The DLL can operate upto 4 ICP-programmer siumultaneously and therefore it's well suited for heavy duty programming in the mass production field.

The system consists of a Windows DLL (WIN98..WIN2000/XP) and several COMports or USB ports with the connected ICP programmers.

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

The ACCUs of the connected ICP-programmers are continously charged by the COM/USB ports (if opened) so a separate charge supply is not needed in most cases. If the system is shut down more than 1..2 weeks so the ICPs should be connected to their charge supply.

The system DLL-Programmer uses packed or encrypted files which must be build with the help of the E-LAB program AVRprog.exe. These files contain all necessary informations (Flash-HEX file, EEprom –HEX file, CPU-type, Lockbits, Fusebits etc). Because of this a manipulation or faulty setup of the programmers by the production is absolutely impossible. As an option there is the feature to limit programming cycle count. Further more there is a target system dependant password protection. (see description in the ICP manual).

DLL

The DLL does the big part of the work. It is constructed in a way that the main control program in the test computer (or PC) can be concetrated to essentials : Download of programs/firmware into the connected ICP-Programmer, programming start, result analyzing. All functions return as the funktion result a string (Pchar) and an integer (res) as an enumeration.

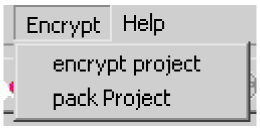
Enumeration of the function results:

resNone, noProg, progFound, MemError, progBusy, 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

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

Packed or encrypted files

The DLL can process two file types. Both file versions must be created by the E-LAB PC-program ***AVRprog.exe***



Packed File is a binary file which contains all necessary information *except* encryption, passwords or device count limitations. Because of this it *must not* be introduced to the file pool by „AddNewFile". The file extension is always **.pack**

Encrypted File: is a binary file which contains all necessary informations *inclusive* encryption, passwords and device count limitations. Because of this it *must always* be introduced to the file pool by „AddNewFile". The file extension is always **.encr**

Attention:

Because of the necessary adaption to VB and diverse Windows versions the DLL interface must be changed somewhat. Alle functions have changed to procedures. The former function result „PChar" has been changed to „resStr". All DLL functions now copy the textual result of a call into a memory block which the calling application must provide for this!

The **Interface functions** of the DLL are as follows (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); stdcall; 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 overview

Current E-LAB programmer of type ICP-V24 or ICP-USB support also the boundary scan function of the mega AVR CPUs. Boundary scan is completely separated from the programming part in the AVR CPU and also in the ICP programmer and also in this DLL. The only common function for programming and for boundary scan is the DLL function ***SetChannel***.

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

For board testing through boundary scan there are only two functions

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

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

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

Function details

In the following context the term “COMport“ is used for the serial port name or USB port of the test computer. The term “Channel“ (upto 4) is used for a serial output port of the system and therefore defines a specific ICP programmer connected to the system. With the DLL function „SetChannel“ a specific COMport is mapped to a Channel. With a single channel DLL “Channel” must be preset to 0. “Control System“ and “Host“ represents the test computer where the DLL and the associated program is installed.

Note:
With the single channel DLL “Channel“ must always contain a 0, with the 2 channel version it can be 0 or 1, with the 4 channel version “Channel“ can be 0..3
COMport can consist of „COM1“ to „COM8“ or „USB“.

Maintenance functions

These functions can be called every time regardless whether a programmer is connected to the system or not

procedure **AddNewFile**(FileName: PChar; var res : integer; resStr : Pchar);
// res = progBusy, invFile, invFName, invPassword, FileExist, resOk
The encrypted file (project) is added to the file pool. If the file is password prtected and the computer doesn’t correspond to the internal file password the file will be rejected with the errorcode “invPassword“. The filename must contain the full **path** and also the extension **.encr**
Note: this operation is **not** applicable for packed files.

procedure **DeleteFile**(FileName: PChar; var res : integer; resStr : Pchar);
// res = progBusy, invFName, notFound, errProtected, resOk
A file (project) will be removed from the file pool, but the file itself doesn’t get erased. If this file is cycle limited it can’t be deleted and the operation results in an “errProtected“.

procedure **GetFile**(index : integer; var res : integer; resStr : Pchar);
// res = progBusy, notFound, resOk
This function serves for listing of the content of the file pool. The application increments the value of “index“ until the result is different of “resOk“. If the result is resOk then “PChar“ points to a string which contains the project name which is listed with “index“ in the file pool.

procedure **GetPasswd**(var res : integer; resStr : Pchar);
// res = progBusy, resOk
This function returns as a result the password which must be used by the encrypt program part of “AVRprog.exe“ in order to generate protected files/projects which can be installed only in this target system. In most cases (internal use) a limitation or a password doesn’t make sense. With far east productions this feature can be very useful ☺.

Initialization

These are called once at a start up. The first ones are absolutely necessary. A download must be done only if one or more programmers must be feed with new content. Note that a ICP is ACCU buffered so the content is still valid after a power-down and power-up.

procedure **SetChannel** (var Channel : integer; ComName : PChar; var res : integer; resStr : Pchar);
// res = progBusy, progFound, noProg, errProgType

This function seeks a programmer at the serial or USB port "ComName" of the Host. In „Channel" a number (0..3) must be passed, which must be used for further transactions. If this function was successful. „Channel" now is used and can only be released with „AbortAll". A scanning of ports is not implemented. The programmer himself is responsible that all desired ports exist and ICP-programmers are connected.
This function must be called once for each connected ICP-programmer

If successful (res = progFound) the parameter „Channel" returns the registered number, otherwise it returns a -1. The function forces a hardware reset in the connected programmer.

procedure **CheckProgrammer**(Channel: integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, errProgType, progFound, MemError

If an ICP-programmer was found with the function "SetChannel" this device always can be checked with this function. The parameter "Channel" defines the desired programmer. The function generates a hardware reset to the selected programmer.
If a "MemError" is reported, a download must be executed.

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

If an ICP-programmer was found with the function "SetChannel" this device can be completely cleared with this function. The parameter "Channel" defines the desired programmer. All checksums, passwords and memory contents will be cleared.

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

This function serves to interrogate the actual project stored in the ICP-programmer. The parameter "Channel" passes the number of the desired programmer. If successful the result of the function is the project name stored in the ICP. The parameter "Age" contains the date of creation of the encrypted origin file.

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

If successful this function returns the charge state of the ACCU in the parameter "ACCU" in percent.

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

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

procedure **DownloadFile**(ProjName: PChar; BurnCycles: Integer; Channel : Integer; var res : integer;
resStr : Pchar);
// res = progBusy, limitExc, errProgType, progProtected, dwnLoadErrP, dwnLoadErrF,
// dwnLoadErrE, dwnLoadErr, resOk

This function has only to be used if a project file must be downloaded into a particular programmer connected to the Host. Because the E-LAB ICP programmers are ACCU buffered this is only necessary with a project change or when the ICP reports a corrupted memory.

The error types dwnLoadErrP, dwnLoadErrF and dwnLoadErrE serve for analizing the problem (Download Parameter, Download Flash or Download EEprom). The error „limitExc" appears if the Host tries to download a programming cycle limited project which limitation has been reached or is exceeded. (Encrypted Files)

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

Encrypted files
The parameter **ProjName** is the same as the **FileName** used by **AddNewFile** without a path name but with the file extension (.encr).
The parameter **BurnCycles** is only important if the project is cycle limited. If the limit for example is 10000 and there are 4 ICP connected and each programmer has to program the same count of pieces the parameter “BurnCycles“ must be set to 10000/4 = 2500. Each programmer now can program 2500 chips. Without a cycle limit in the original project this parameter should be set to 0.

Packed files
The parameter **ProjName** must contain the path and file extension (.pack). The parameter **BurnCycles** has no meaning and should be zero. The file is directly downloaded into the programmer without any relation to the file pool.

File extensions
These ***must*** be included in the filenames so that the DLL can identify the file type.

Production

procedure **ProgDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
This function starts one programming cycle. If the result “res“ = resOk the operation started successful. At first the device is erased. Then then the Flash, EEprom, Fusebits and Lockbits are programmed. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

procedure **EraseDevice**(Channel: Integer; var res : integer; resStr : Pchar);
// res = progBusy, noProg, resOk
This function starts one chip erase cycle. If the result “res“ = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

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

procedure **ProgFlash**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
This function starts the separate programming of the Flash. If the result “res“ = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

procedure **ProgEEprom**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
This function starts the separate programming of the EEprom. If the result “res“ = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

procedure **ProgFuses**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
This function starts the separate programming of the Fusebits. If the result “res“ = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

procedure **ProgLockBits**(Channel: Integer; var res : integer; resStr : PChar); stdcall; external 'ProgDLLN.dll'
// res = progBusy, noProg, resOk
This function starts the separate programming of the Lockbits. If the result “res“ = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continosly call the function “GetProgStatus“ which polls all active ports to get the state of all startet ICPs.

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

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

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

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

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

They come in the following order:

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

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

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

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

procedure **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

These four functions provide an online manipulation of the actual Flash or EEprom content in the programmer. So it's possible for example to alterate serial numbers etc. between programming cycles without doing a complete new dowload. Each download/content change is valid for the next programming cycle(s). The blocksize should not exceed 256 Bytes. The application programmer is responsible for the correct blocksize and the target adddress „dest" or the source address "source". There is no check by the system.

These 4 functions require a programmer **Typ ICP-V24** or **ICP-USB** with a firmware revision **6.0** or higher.

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

These two funktions support a read back of the actual Flash or EEprom content of the target CPU, on condition that the programmer itself and the target is not protected.

These 2 functions require a programmer **Typ ICP-V24** oder **ICP-USB** with a firmware revision **6.1** or higher.

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

The purpose of this function is to support a reprogramming of small Flash parts. This operation is only possible with flash contents where the relevant locations consist of $FF, meaning that these bytes can be reprogrammed.
For technical reasons and in order to handle all CPU types using always 256 byte blocks is mandatory. In addition this block must be aligned with a 256 byte boundary. This should not be a problem at all.
There are several possibilities to create such a block.

1. Fill a memory block of 256 bytes with $FF. Then change the desired locations of this block with new values. Call the function DownProgBlock256.
2. Fill a memory block of 256 bytes with the content of the original file. Change the desired locations of this block with new values. The origin locations to change must consist of $FF. Call the function DownProgBlock256.
3. Fill a memory block of 256 bytes by an upload from the programer's memory or direct read out of the flash of the CPU. Change the desired locations of this block with new values. The origin locations to change must consist of $FF. Call the function DownProgBlock256.

If the $FF filled block is used then only these memory locations will be reprogrammed which were changed to non $FF. If the original flash content is used all memory locations will be reprogrammed but only these bytes will be changed which are different. Attention: basically only bits with the value of "1" can be changed but never bits containing a "0".

Block is a pointer which must point to the 256 byte source/buffer. **Dest** is the target address of the 256 byte blocks in the flash memory of the CPU. It always must point to the start of a 256 byte block in the flash.

The CPU already must be programed (Flash and Fuses) and lockbits must not be activated.

These 2 functions require a programmer **Typ ICP-V24** oder **ICP-USB** with a firmware revision **6.6** or higher.

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

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

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

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

```
procedure AbortAll;
```

This procedure is necessary if the system “hangs“ or the DLL dropped out of synchronisation with the Host or one or more programmers. After the command “AbortAll“ the DLL is in a basic state and can accept new commands. All ComPorts are closed.

A complete re-init with “SetChannel “ and “CheckProgrammer“ must be done.

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

JTAG Boundary SCAN functions

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

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

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

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

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

The parameter CpuID and BoundaryLen must be copied from the datasheet of the AVR CPU.
With the mega16 is CpuID = $001E9403 and BoundaryLen = 141 (Bits).
mega128 CpuID = $001E9702 BoundaryLen = 205 (Bits)

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

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

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

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

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

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

Closes the JTAG Scan Port of the target CPU.

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

Examples and sources

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

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

Import of the DLL into Delphi

```
type
  tProgResult =   (resNone, noProg, progFound, MemError, progBusy, 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 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 of the DLL into 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)
```

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 of the DLL into 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

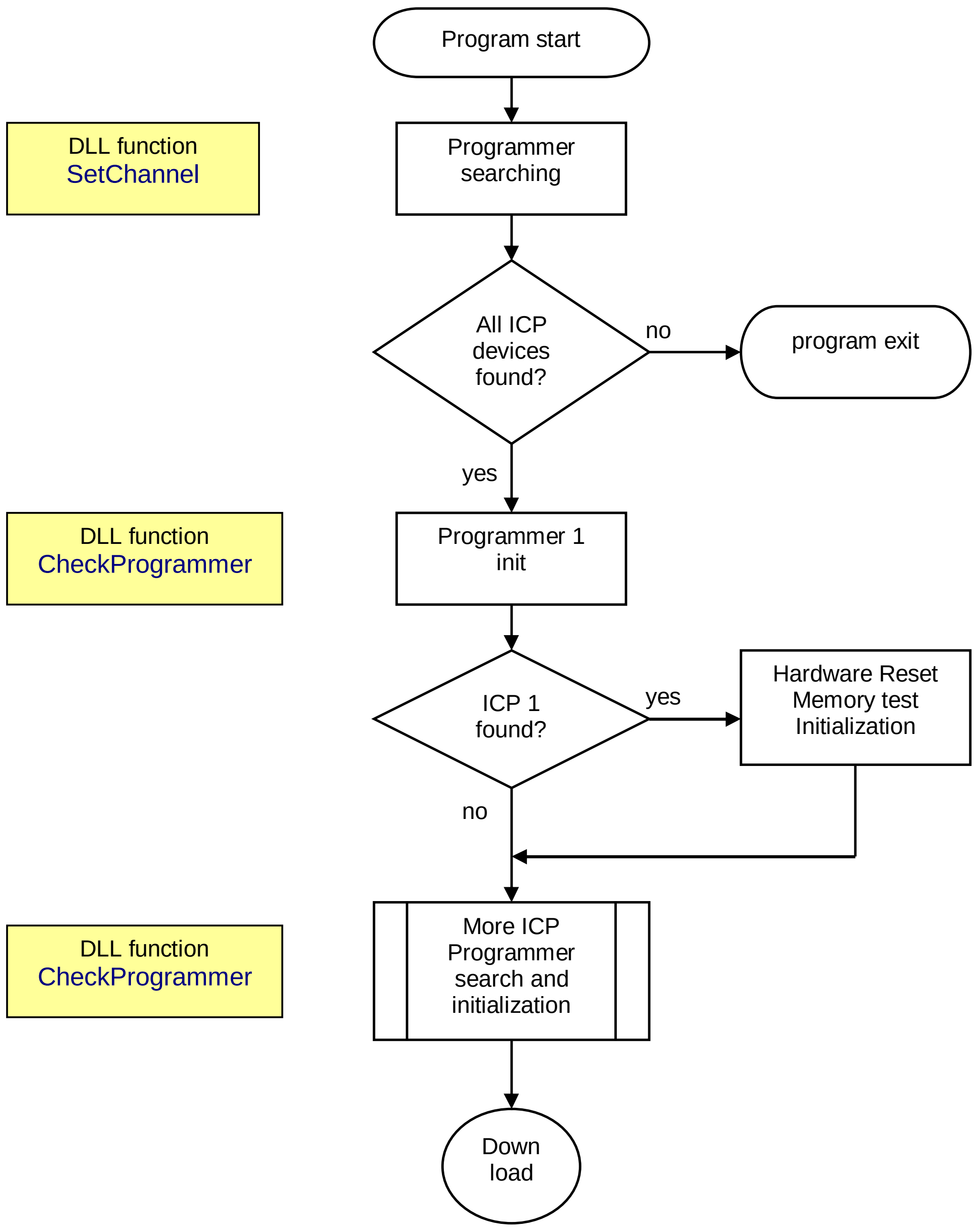
Flow diagrams

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

Initialization

The ICP-programmers must be connected to the Host (COM1..COM8 or USB). A 32bit Windows must run on this computer. (WIN98/NT/WIN200/XP). The DLL „ProgDLLN.DLL“ must reside in the same directory where the application program which controls the DLL and programmers is located.

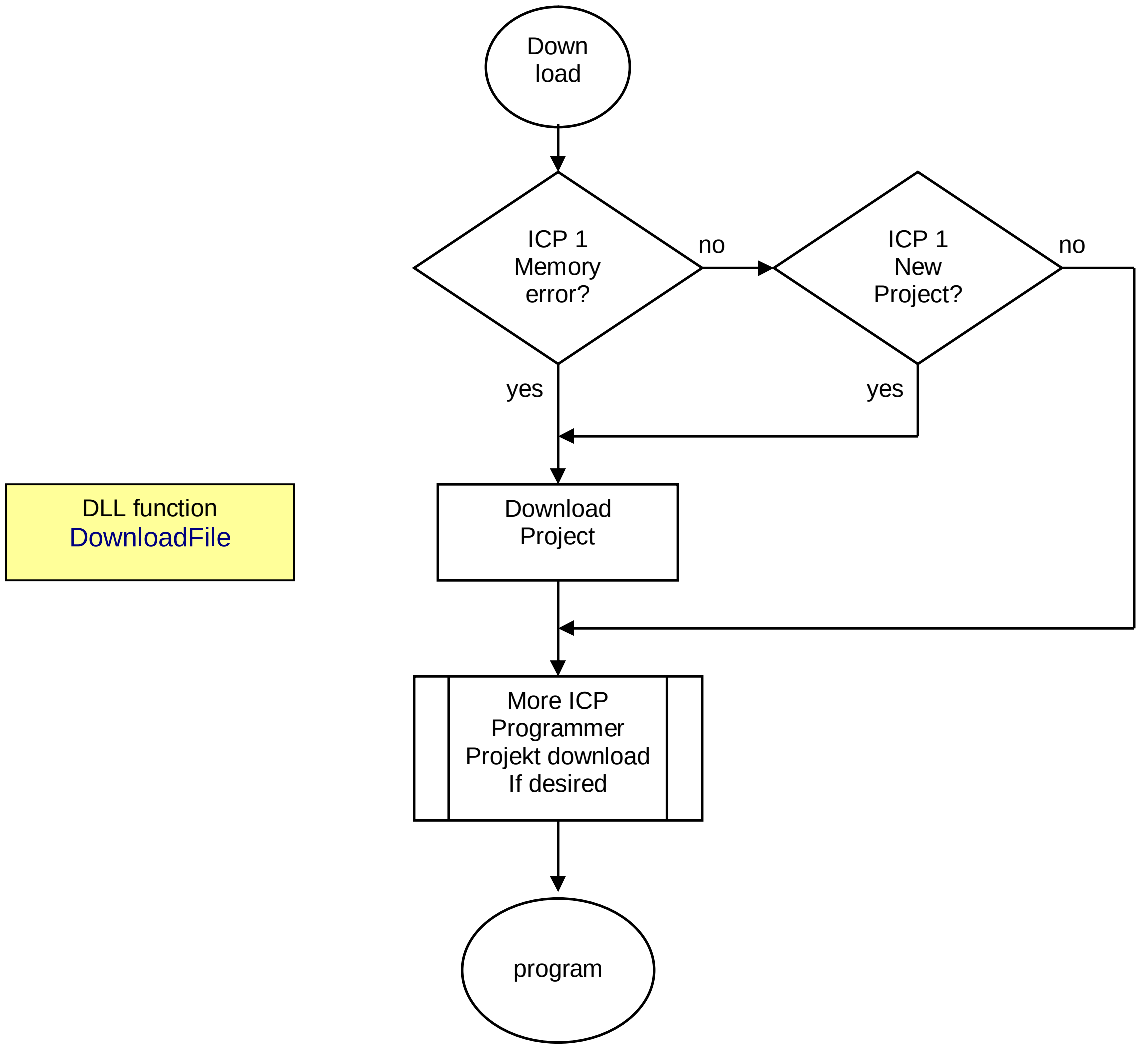
At start up of the application at first the connected programmers must be searched and mapped into “Channels”. If a ICP-programmer is found via the DLL this ICP gets also a hardware reset. The ICP can respond with a memory error via the Multiplexer to the application. Now the init sequence is complete.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



DownLoad

A project download into a connected programmer is only necessary if either within the initialization the ICP reports a memory error or the project stored in the ICP must be replaced by another or new one.

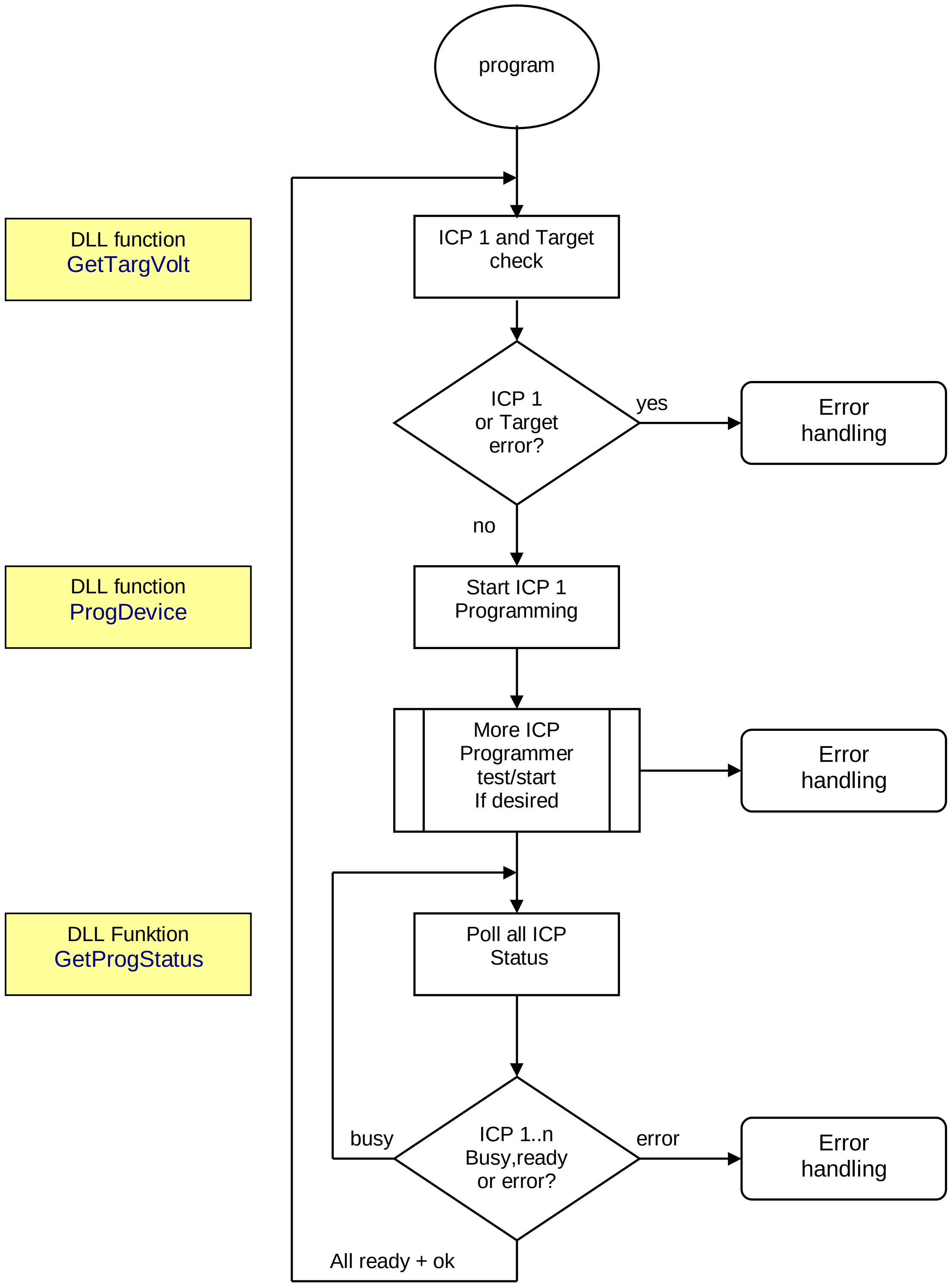
The ICP programmer are battery buffered and retain a once stored project also through a power down cycle. The memory is checked after each PowerOn or Reset against some checksums.



Production

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

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



E-LAB Multiplexer

The disadvantage of this Production System is that each programmer needs it's own serial port on the Host system.

The Production System ***E-LAB Multiplexer*** uses only one serial port at all, alternatively one USB-port. Because of this a further unloading of the Host system can be acchieved.

Infos about the ***E-LAB Multiplexer*** on our 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