E-LAB

Production Programmer System

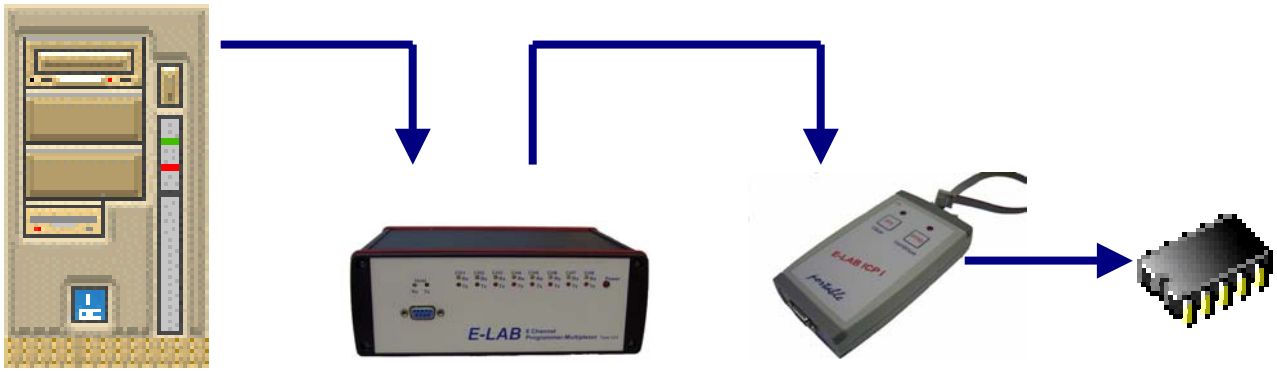
AVR *Multiplexer*

- Production Programming System for Atmel AVR CPUs
- Upto 8 simultaneous programmings
- Controlled through a WIN32 DLL
- Master control program supplied by the user
- Only one serial Interface (COMport) used
- Optional connection to USB or Ethernet
- 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 March 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 in most cases there are not enough COMports and 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-Multiplexer and it's 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 Multiplexer. The DLL and the Multiplexer 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-Multiplexer, 1 to 8 E-LAB ICP programmers and the WIN32 DLL.

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/Multiplexer

The E-LAB Multiplexer serves an interface between a test automat (or PC) and ICP-Programmers in the production of boards which are equipped with Atmel AVRs. The Multiplexer can operate upto 8 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) and the Multiplexer unit with connected ICP programmers.

The test computer communicates via the DLL with the Multiplexer. 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.

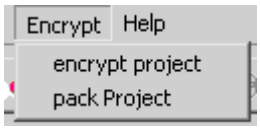
The Multiplexer provides one serial input V24/RS232 (optional USB or Ethernet) and 8 serial outputs. The main supply can handle 120V~ and 240V~ with an build-in switch.

The ACCUs of the connected ICP-programmers are continously charged by the Multiplexers so a separate charge supply is not needed in most cases. If the Multiplexer is shut down more than 1..2 weeks so the ICPs should be connected to their charge supply.

The system DLL-Multiplexer 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 (encryption only) 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).

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 **.encr**

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 **.pack**

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 procedures return as the funktion result a string (Pchar) and an integer (res) as an enumeration.

Enumeration of the procedure results:

resNone, noMux, MuxFound, 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

For example “resNone“ has the value 0 and “MuxFound“ the value 2, “ProgFound” = 4

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

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

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

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

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

procedure CheckMultiplexer(var ComPort: integer; var res : integer; resStr: PChar); stdcall;
                                external 'MuxDLLN.dll';
// res = progBusy, noMux, MuxFound
```

```
procedure CheckProgrammer(MuxPort: integer; var res : integer; resStr: PChar); stdcall;
                                    external 'MuxDLLN.dll'
// res = progBusy, noMux, noProg, errProgType, progFound

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

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

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

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

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

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

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

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

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

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

Procedure details

In the following context the term “COMport“ is used for the serial port of the test computer (1..8).
The term “MuxPort“ is used for a serial output port of the Multiplexer (0..7) and therefore defines a specific ICP programmer connected to the Multiplexer. “Control System“ and “Host“ represents the test computer where the DLL and the associated program is installed.

Maintenance procedures

These procedures can be called every time regardless whether a Multiplexer or 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“.
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 procedure 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 procedure 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 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 **CheckMultiplexer**(var ComPort: integer; var res : integer; resStr: PChar);
// res = progBusy, noMux, MuxFound
This procedure seeks the Multiplexer at the COMports of the Host. The parameter "COMport" can select a particular port (1..8). Otherwise the parameter 0 enables searching at all ports 1..8. But this can lead to problems if other serial drivers are installed (eg. Some mice) which are not Windows conform. Because the Multiplexer is always connected to the same COMport a fixed seeking without the scan procedure is the better choice.

If successful (res = MuxFound) the parameter "COMport" contains the valid COMport. This procedure generates a hardware reset in the Multiplexer.

procedure **CheckProgrammer**(MuxPort: integer; var res : integer; resStr: PChar);
// res = progBusy, noMux, noProg, errProgType, progFound, MemError
If the Multiplexer has been found the connected programmers must be searched. If an ICP-programmer is found this device will be initialized but no download is done. The parameter "MuxPort" selects the output port and the programmer. The procedure generates a hardware reset to the selected programmer. If a "MemError" is reported, a download must be executed.

procedure **GetProjName**(MuxPort : integer; var Age, res : integer; resStr: PChar);
// res = progBusy, noMux, noProg, errProgType, resOk
This procedure serves to interrogate the actual project stored in the ICP-programmer. The parameter "MuxPort" passes the number of the desired programmer. If successful the result of the procedure is the project name stored in the ICP. The parameter "Age" contains the date of creation of the encrypted origin file.

procedure **GetACCUstate**(MuxPort: integer; var ACCU, res : integer; resStr: PChar);
// res = progBusy, noMux, noProg, errProgType, resOk
If successful this procedure returns the charge state of the ACCU in the parameter "ACCU" in percent.

procedure **GetTargVolt**(MuxPort: integer; var Volt, res : integer; resStr: PChar);
// res = progBusy, noMux, noProg, errProgType, resOk
If successful this procedure returns the target (CPU) voltage in the parameter "Volt" in 10mV steps.

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

This procedure has only to be used if a project file must be downloaded into a particular programmer connected to the Multiplexer. 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 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 diectly downloaded into the programmer without any relation to the file pool.

Production

procedure **ProgDevice**(MuxPort: Integer; var res : integer; resStr: PChar);
// res = progBusy, noMux, noProg, resOk
This procedure starts one programming cycle. If the result "res" = resOk the operation started successful. If all relevant ICP-programmer have been startet the application must continously call the procedure "GetProgStatus" which polls the Multiplexer to get the state of all startet ICPs.

procedure **GetProgStatus**(MuxPort: integer; var res : integer; resStr: PChar);
// res = resNone, limitExc, eraChip, prgEEp, prgFlash, progDone, errPwrDown,
// errSignature, errProtected, errNotEmpty, errVerify, MemError
This procedure is the main working procedure. After sending the program start command "ProgDevice" the Host must continously poll all started programmers with the procedure "GetProgStatus". The Multiplexer itself also continously receives status informations from the started programmers. These informationen are temporarily stored by the Multiplexer. This means that the Multiplexer 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.

The 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 job and can accept a new one.

procedure **CheckDevice**(MuxPort: Integer; var res : integer; resStr: PChar);
// res = resNone, progBusy, noMux, noProg, errProgType, errPwrDown, errSignature,
// errNotEmpty, errProtected
This procedure 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, MuxPort : Integer; var res : integer;
resStr: PChar); stdcall; external 'MuxDLLN.dll';
// res = progBusy, noMux, noProg, dwnLoadErrF, resOk (* Block into Flash *)

procedure **DownLoadBlockE**(Block : Pointer; BlockSize, dest, MuxPort : Integer; var res : integer;
resStr: PChar); stdcall; external 'MuxDLLN.dll';
// res = progBusy, noMux, noProg, dwnLoadErrE, resOk (* Block into EEprom *)
These two procedures 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". There is no check by the system.

procedure **AbortAll**;
This procedure is necessary if the system "hangs" or the Multiplexer dropped out of synchronisation with the Host or one or more programmers. After the command "AbortAll" the Multiplexer is in a basic state and can accept new commands.

The best way to get out of an instable condition is complete re-init with "CheckMultiplexer" and "CheckProgrammer". The first procedure forces a hardware reset onto the Multiplexer. The others are forcing a hardware reset to all connected programmers.

Examples and sources

The Multiplexer System contains the WIN32 DLL ***MuxDLLN.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
  tMxResult   =   (resNone, noMux, MuxFound, 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);

procedure AddNewFile(FileName: PChar; var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll'
procedure DeleteFile(FileName: PChar; var res : integer; resStr: PChar); stdcall;external 'MuxDLLN.dll';
procedure GetFile(index : integer; var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll';
procedure GetPasswd(var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll'
procedure CheckMultiplexer(var ComPort: integer; var res : integer; resStr: PChar); stdcall;
                                    external 'MuxDLLN.dll';
procedure CheckProgrammer(MuxPort: integer; var res : integer; resStr: PChar); stdcall;
                                     external 'MuxDLLN.dll';
procedure GetProjName(MuxPort : integer; var Age, res : integer; resStr: PChar); stdcall;
                                  external 'MuxDLLN.dll';
procedure DownloadFile(ProjName: PChar; BurnCycles: Integer; MuxPort : Integer; var res : integer;
                                  resStr: PChar); stdcall; external 'MuxDLLN.dll';
procedure GetACCUstate(MuxPort: integer; var ACCU, res : integer; resStr: PChar);
procedure GetTargVolt(MuxPort: integer; var Volt, res : integer; resStr: PChar);
procedure ProgDevice(MuxPort: Integer; var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll';
procedure GetProgStatus(MuxPort: integer; var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll';
procedure CheckDevice(MuxPort: Integer; var res : integer; resStr: PChar); stdcall; external 'MuxDLLN.dll';
procedure DownLoadBlockF(Block : Pointer; BlockSize, dest, MuxPort : Integer; var res : integer;
                                    resStr: PChar); stdcall;  external 'MuxDLLN.dll';
procedure DownLoadBlockE(Block : Pointer; BlockSize, dest, MuxPort : Integer; var res : integer;
                                     resStr: PChar); stdcall;  external 'MuxDLLN.dll';
procedure AbortAll; stdcall; external 'MuxDLLN.dll';
```

Import of the DLL into Visual Basic

VB Interface

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

Private Declare Procedure **AbortAll** Lib "MuxDLLN.dll" ()

T.B.D.

Import of the DLL into C++

```
enum _tMxResult
{
	resNone = 0,
	noMux = 1,
	MuxFound = 2,
	noProg = 3,
	progFound = 4,
	MemError = 5,
	progBusy = 6,
	progIdle = 7,
	progProtected = 8,
	eraChip = 9,
	eraEEp = 10,
	eraFlash = 11,
	prgEEp = 12,
	prgFlash = 13,
	verifyEEp = 14,
	verifyFlash = 15,
	errPwrDown = 16,
	errSignature = 17,
	errProtected = 18,
	errNotEmpty = 19,
	errVerify = 20,
	progDone = 21,
	dwnLoading = 22,
	dwnLoadErrP = 23,
	dwnLoadErrE = 24,
	dwnLoadErrF = 25,
	dwnLoadErr = 26,
	invFile = 27,
	invFName = 28,
	FileExist = 29,
	notFound = 30,
	invPassword = 31,
	limitExc = 32,
	errProgType = 33
};

typedef   enum _tMxResult  tMxResult;

void AbortAll();
```

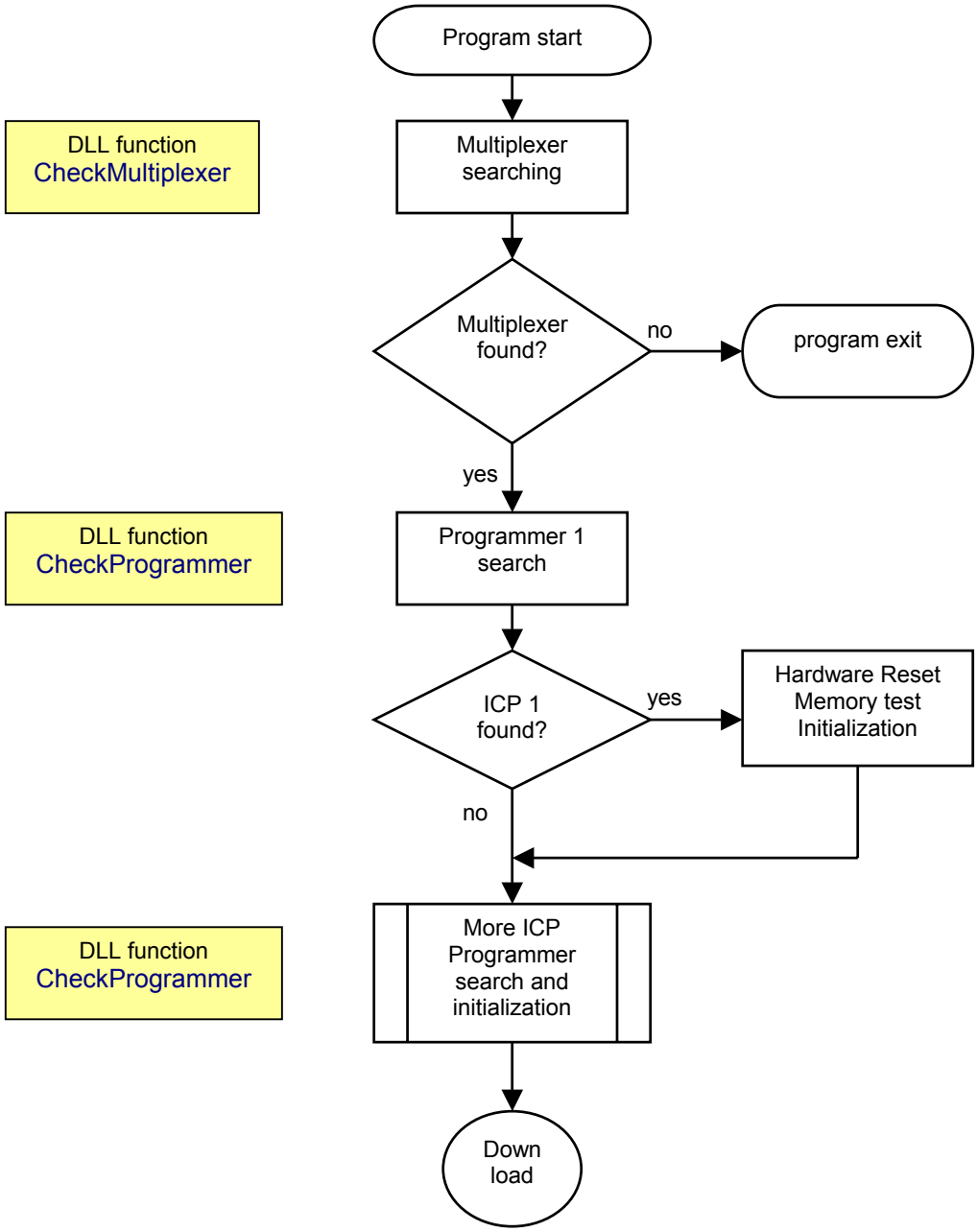
T.B.D.

Flow diagrams

For a better view onto the useage of the AVR Multiplexer in the production there a three diagrams:
1. Initialization 2. Download 3. Production.

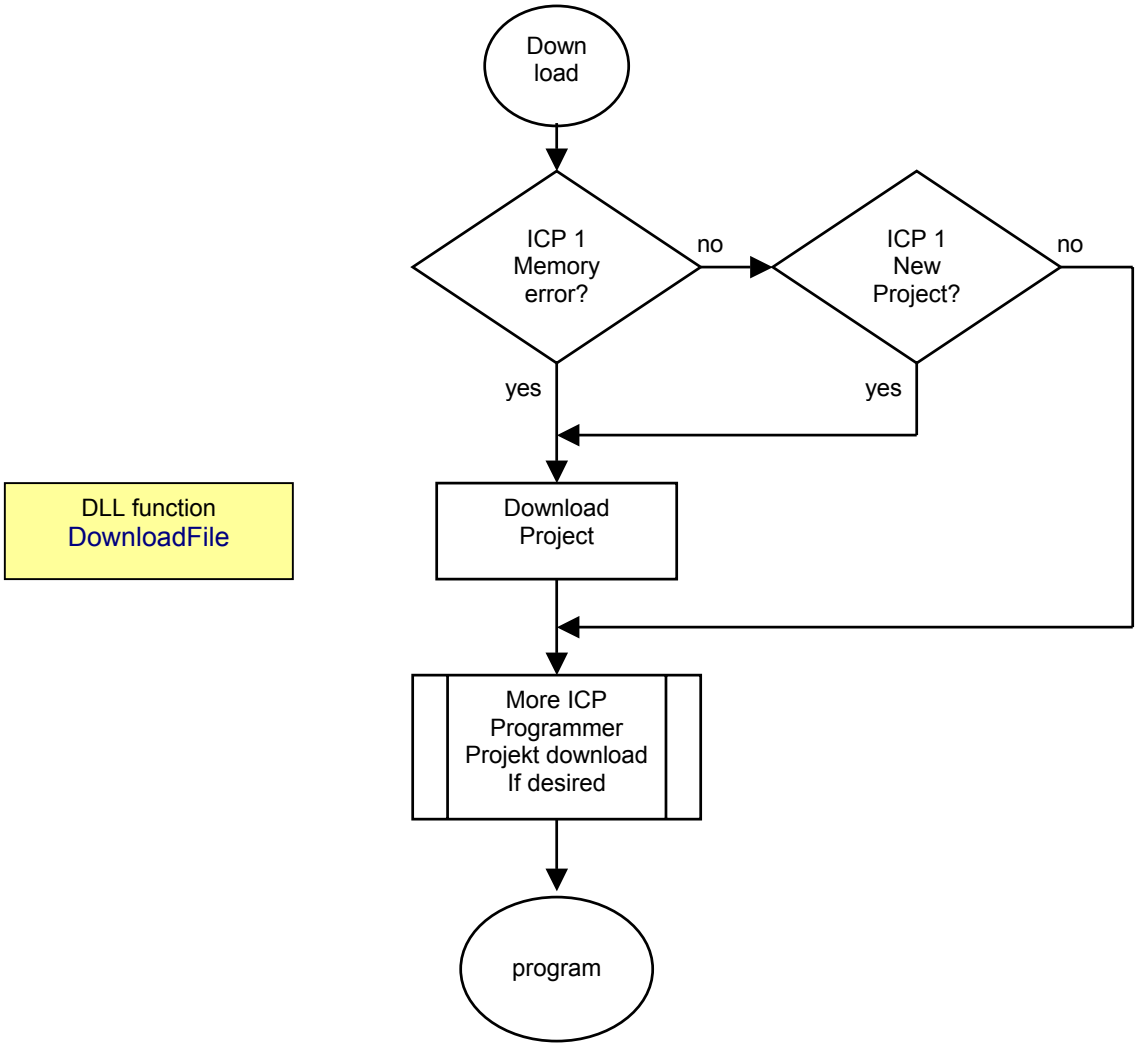
Initialization

The Multiplexer must be connected to the Host (COM1..COM8). A 32bit Windows must run on this computer. (WIN98/NT/WIN200). The DLL „MuxDLLN.DLL“ must reside in the same directory where the application program which controls the Multiplexer is located.
At start up of the application at first the Multiplexer and then the connected programmers must be searched. If the Multiplexer is found an automatic hardware reset to the Multiplexer is initiated. If a ICP-programmer is found via the Multiplexer 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-ACCUs can be requested.



DownLoad

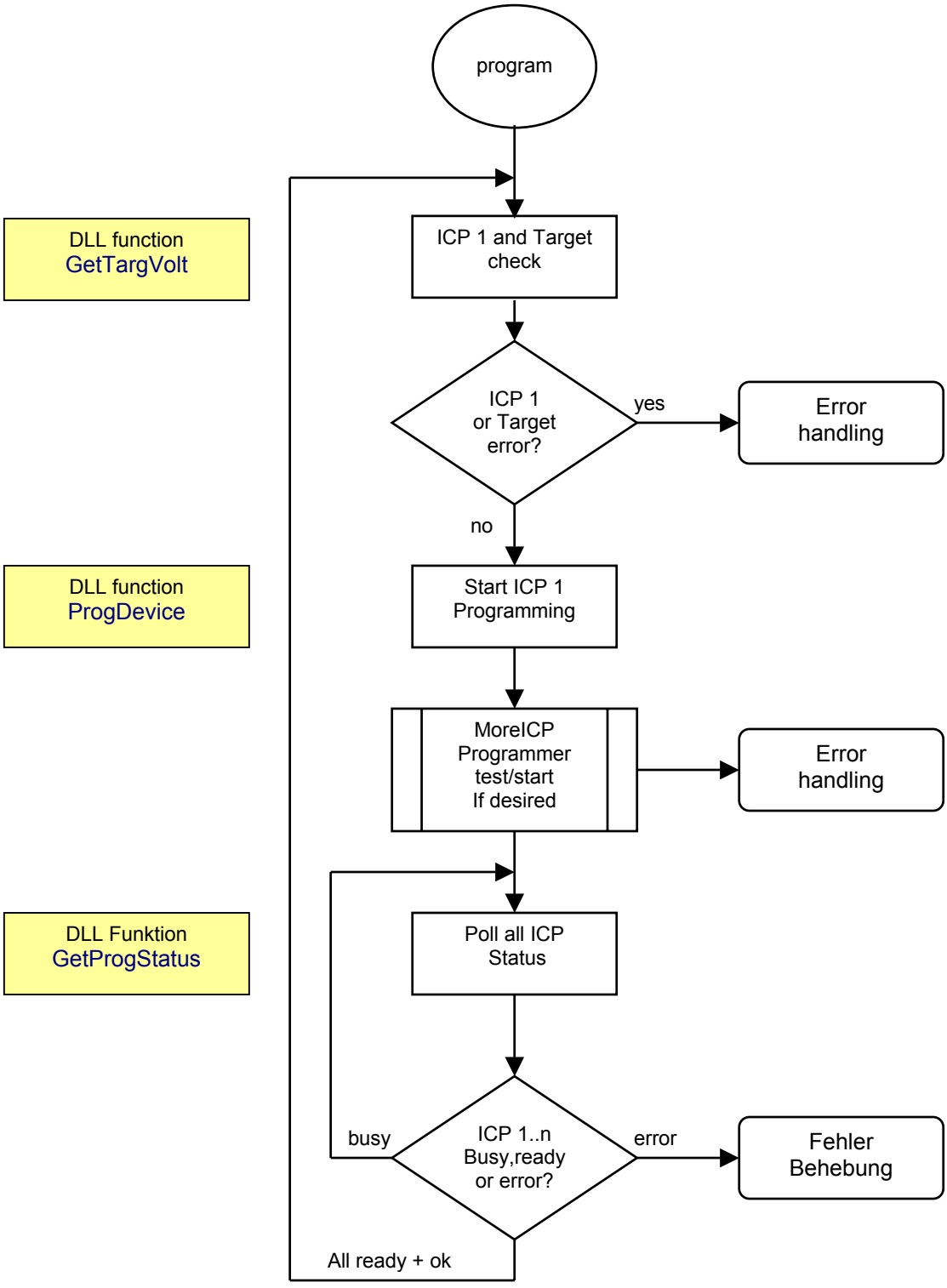
A project download into a connected programmer is only necessary if either within the initialization reports a memory error or the project stored in the ICP must be replaced by another or new one.
The ICP programmer are ACCU buffered and retain a once stored project also through a power down cycle. The memory is checked after each PowerOn or Reset against some checksums.

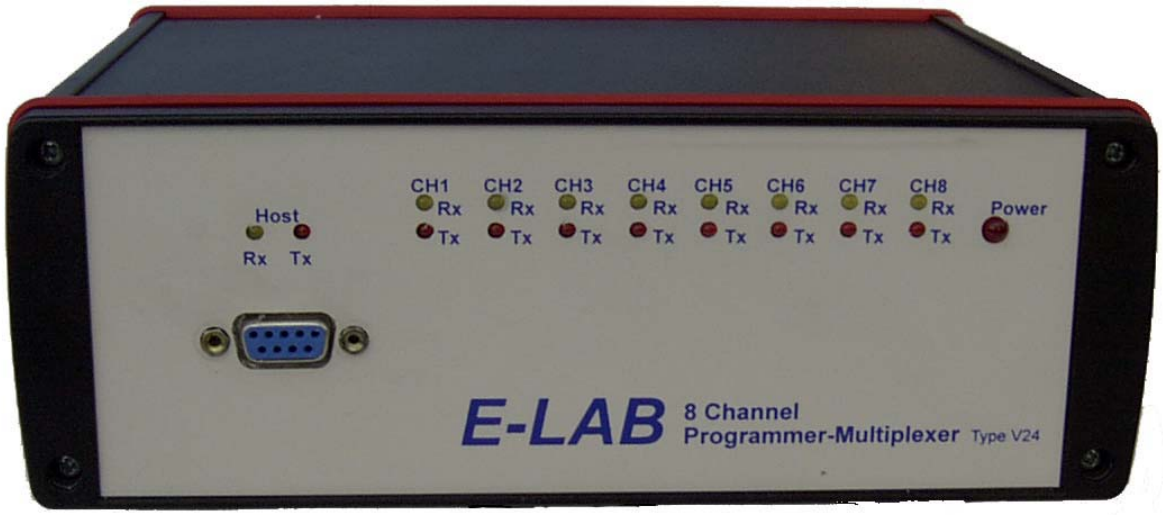


Production

The programming cycle must be started separately for each connected ICP programmer. The start procedure 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 procedure „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 procedure. If all active programmers have send either a „progDone" or an error message, the programming cycle is finished and a new one can be started.





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