SOFTING GmbH Richard-Reitzner-Allee 6
85540 Haar
Tel.: 089/45656-0 Fax: 089/45656-499

INPA

Interpreter for test procedures

User documentation

Authors:
Georg Luderböck
Michael Koch
Roland Stadtmüller

Translation
Date: 20.07.2000

1 Introduction

With the introduction of new control units or control unit versions in production a diagnostic program for this control unit, for example for post processing, must be provided. In the past this has been done by expanding the diagnostic programs developed for use in after-sales service.

To allow us to react more flexibly and above all faster to the requirements in the factory, initially a prototype, then later, in the expansion phase 1, an independent test procedure system was developed that enables a skilled executive on site to develop rework programs for the plant.

These rework programs are formulated in a test procedure description language, which, in turn, was also expanded in the current expansion phase 2 to take account of the requirements to provide it with as universal a scope as possible. The INPA (interpreter for test procedures) system described in this document is notable for the following properties:

- Easy to learn
- Immediate control of the created test program by interactive drafting
- Convenient facilities for screen output
- Convenient facilities for controlling the procedure
- Simple facility for formulating quasi-simultaneous actions
- Extensive library with standard functions, for example
  - Configuration of the screen layout (front and background colour, font size)
  - Configuration of message windows
  - Output to printer
  - Screen shots
  - File access
  - String and conversion functions
  - Standard interface (with and without result evaluation by the INPA system) for **E**lectronic **DIA**gnostic **BA**se **S**ystem (EDIABAS)

In keeping with the intended area of application of INPA, access to information about the vehicle (and of the control module that requires diagnostic) is through the electronic diagnostic base system EDIABAS and through the EDIC hardware.

1.1 Table of contents

2 Overview

This document is designed to help users to understand the interpreter for test procedures (INPA); readers of this user documentation are given an overview of the possible applications and the components of the INPA that are visible for the user in section 3.

Section 5 describes the structure of the test procedure description files, the principle procedure used for tests and the language elements (for example including the functions of the standard library) used to describe (and to execute) tests. This will enable readers to formulate and run tests in a test procedure description language. The listing of the language element in sub-section 5.3 is also intended to act as a reference.

Section 6 provides a rough overview of the software structure and the components of the INPA procedure system. This section is intended to explain the outline relationships between the modules in terms of software update action.

This INPA user documentation is aimed at the following:

- Staff in departments at BMW, who define or change test procedures in the production departments
- Staff in departments at BMW, who define laboratory tests
- Developers of application programs who include test proce3dures in the electronic diagnostic system (ELD) on the production line
- Software developers, who are responsible for INPA system support

2.1 Release status

Initial creation of version 1.0, 20.09.1993

Revision V1.1 06.10.1993:

Section 2:	Added
Section 3.3.7:	Text formatting explained in greater detail

Revision V1.2, 02.02.1994:

Section 3.3.7:	DTM functions and string array functions added Hexdump description updated
Section 3.3.2:	2nd parameter for LINE-Construct and setscreen functions described

Revision V1.3, 08.07.1994:

Section 1:	System structure and description updated
Section 2.1:	File structure updated
Section 2.2:	Description of the configuration files updated
Section 3.1.1:	New section to describe pragmas added
Section 3.1.2:	Description of the Include mechanism and script structure updated
Section 3.3.3:	Second function key level described
Section 3.3.7:	Description of the library functions updated
Section 4:	INPA localisation support described
Appendix A:	Language definition updated (program structure, pragma)
Appendix B:	Parameter assignment for analogout updated

Revision V1.4, 17.03.1995:

Section 2:	Inclusion of the new configuration
Section 3.7:	Inclusion of RK512 functions Expansion of DTM functions Replacement of sgselect by scriptselect GetBinaryDataString function printfile function calldos function removed

Revision V1.5, 16.01.1996:

Section 2.2.1:	Expansion of Setup settings
Section 3.2.3:	Fab files
Section 3.3.7:	Inclusion of the new INPAapiFsRead function and 0x40 mode Inclusion of INPAapiFsRead2 function Inclusion of ELDIStartOpenDialog function Change of togglelist function
Section 4.2.3:	Added
Section 4.2.1:	Language is ENGLISH

Revision V1.6, 03.04.1996:

Section 2.2.1:	Supplement to Setup settings
Section 5.3.7:	Functions fileread, Delay, stringtoreal, stringtoint, inttoreal, realtoint added fileopen function supplemented by ‘r’ parameter IS_LESEN special treatment included
Section 4.2.4:	‘Command line options’ section added

Revision V1.7, 28.05.1996:

	Section 5.3.7:	'Delay' function now called 'delay'
Revision V1.8, 05.06.1996:		
	Section 4.1.2:	File names supplemented
	Section 5.3.7:	'Delay' function now really called 'delay'; 'stringtoint' function adapted; limit value for Min-Real adjusted; default value for INPAapiFsMode improved.
Revision V1.9, 30.07.1996:		
	Section 5.3.7:	'getdate' and 'gettime' functions supplemented
	Section 4.1.2:	'commdlg.dll' file is no longer supplied
Revision V2.0, 17.02.1997:		
	Section 4.2.1:	'Splittingfile' configuration supplemented
	Section 5.1.3	"Imported functions" section added
	Section 5.1.5	New data types added
	Section 5.3.1	New data types added
	Section 5.3.2	"Imported functions" section added
	Section 5.3.7	Binary lines added
	Section 5.3.8	Memory manipulation routines supplemented multianalogout, scriptchange and setitem functions supplemented
Revision V2.1, 10.03.1997:		
	Section 4.1.2:	File names deleted/supplemented
	Section 5.3.8:	Supplemented to 'scriptchange' function
Revision V2.2, 27.03.1997:		
	Section 6.7:	Compiler names adjusted
	Section 5.3.8:	Supplement to 'delay' function
Revision V2.2, 11.04.1997:		
	Section 4.2.1:	Configuration setting decoder tables adjusted

3 System description

The INPA system is based on WINDOWS and therefore requires a PC 80386 (or higher) with 4 MB RAM (or more) to run properly. In addition an interface to the vehicle or to a control module, an **E**nhanced **D**iagnostic **I**nterface **C**omputer (EDIC) is provided.
Communication with the EDIC is controlled by the electronic diagnostic base system, which must be available in the form of a dynamic link library (DLL).

Typical applications for INPA include the following:

- Rework programs, in other words stand-alone programs for diagnosing a single control module
- "Fast tests ", in other words test procedures that are only used one or just a few times or which are subject to frequent changes (for example tests during development or in the laboratory)
- Special tasks such as ELDI special jobs, with the inclusion of the INPA test procedure in the ELDI procedures

Test procedure programs are developed by a trained executive on site, who is given a correspondingly flexible description language for this purpose. Using an (arbitrary) ASCII editor, test procedure description files are produced for test procedures in the above description language.

These description files are processed by interpretation during the run time of the INPA, during which the system is supported by the components shown in schematic form in Figure 1.

To reduce the loading times for the test procedure description files and to minimise memory requirements, the INPA system has been implemented in such a way that it can be operated in configurable modes, namely in the following forms:

- An integrated system for the interactive development of test procedure descriptions and to compile the created descriptions so that they can be loaded and processed during the run time of the INPA using minimum time and memory.
- INPA loader to load and process pre-compiled test procedure description files. The INPA loader configuration is used at the place where the test is used.

The INPA system allows arbitrary test procedures written in the test procedure description language (PABS) to be run; this means the following for the tests:

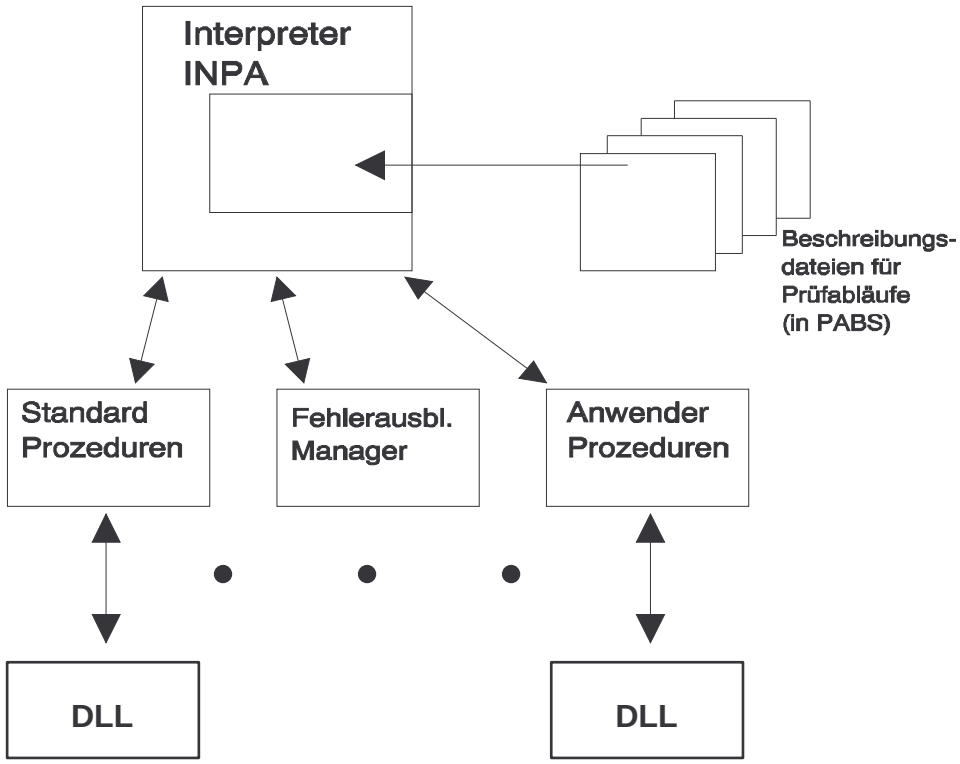
- Automatic procedures or procedures controlled by the user from the keyboard can be formulated
- Test results (or other output) can be output on the screen or in a file (protocol), or they can be returned in the form of a return value to the program that activated the system

The test procedures formulated in the test procedure description files (PABS) typically include the following tasks:

- Description of the test procedure tree by a menu controller
- Structure of the communication with a control module
- Request for data from a control module
- Evaluation of the data (results)
- Linking several results
- Display of the (linked) results on the screen

In addition PABS also provides elements to describe the screen layout (colours and fonts), control structures, such as Boolean operations, conditional commands and loops for the evaluation and display of results.

Components of the INPA:



INPA interpreter Description files for test procedures (in PABS)

Standard procedures Error output manager User procedures

DLL DLL

Figure 1

As shown in Figure 1, the interpreter for test procedures (INPA) is made up of several components and is supported by dynamic link libraries (for example EDIABAS, protocol and label managers and localisation DLL for multiple languages), which are available at the intended place of use, some of them from other diagnostic systems. For places of use at which these DLL's are not available, they can be copied unchanged.

4 Installation and configuration

4.1 Installation

4.1.1 Requirements for operating the interpreter

EDIABAS must be installed to operate the interpreter.

4.1.2 Files after installation

README	:	File containing general information on the installation diskette
README.xyz	:	Readme for INPA Vx.y.z for the specific version

BIN directory

INPA.EXE	:	Interpreter for test procedures, integrated system with compiler, loader and interpreter
INPACOMP.EXE	:	Compiler for script files
INPALOAD.EXE	:	Run time system with loader and interpreter
INPAGER.DLL	:	German foreign language support
INPAUS.DLL	:	English foreign language support
WEPEM.DLL	:	Protocol and label manager DLL
WEDTM.DLL	:	Data table manager DLL
WEFABM.DLL	:	Error mask-out DLL
ATRACE.DLL	:	Trace DLL
DEUTSCH.DLL	:	Language DLL from Wineldi
ENGLISCH.DLL	:	Language DLL from Wineldi
E3_SPR.DLL	:	Language help DLL from Wineldi
WEEDIA.DLL	:	Joint WINELDI / INPA link to EDIABAS
WINPRINT.DLL	:	Printer manager
RK512.DLL	:	RK512 communication DLL

The following are also required for operation with the WINELDI input handler:

E3X_EINH.EXE	:	WINELDI/3 input handler
DECODE.DLL	:	Decoding the order data
QSDMAN.DLL	:	Quality assurance data
WENETM.DLL	:	Network link of the input handler
EDIERROR.TXT	:	German language support for EDIABAS
EDIERROR.ENG	:	English language support for EDIABAS
USRDUMMY.DLL	:	Substitute for the WINELDI screen server
BMDUMMY.DLL	:	Substitute for the WINELDI screen server
NET.DAT	:	Settings for the network manager
WXSEMAPH.DLL	:	Flags for WINELDI and MOTEST (used by input handler)

CFGDAT directory

INPA.INI	:	Configuration file for INPA tools
INPASCR.INI	:	Script selection file for INPA and INPALOAD
STARTGER.IPS	:	Startup script for interpreter (source), German
STARTUS.IPS	:	Startup script for interpreter (source), English

STARTGER.IPO : Startup script for interpreter (object), German
STARTUS.IPO : Startup script for interpreter (object), English

HELP directory

INPAHELP.HLP : Windows help file for INPA

PRT directory

*.ini : Printer initialisation files
INPADOS.DMF : Print mask file for INPA screen shorts via PEM
INPAWIN.DMF : Print mask file for INPA screen shorts via Windows standard printer
./ENGLISH or
./GERMAN : Language-dependent versions of the:
E3ETIKET.DMF : Label print mask file (used by PEM internally)
E3PROTOK.DMF: Protocol print mask file (used by PEM internally)

TRACE directory

(contains the temporary file for INPA screenshot for operation without a printer)
("PRINTER=NO" in INPA configuration file)

FAB directory

(contains the user's error mask-out files)

SGDAT directory

INPA.H : Header file with prototype of the INPA library functions

(also contains the user's script files)

DEMO directory

(contains optional demonstration files for special INPA functions)

4.1.3 To set up the interpreter

The program is installed by running setup.exe in Windows.
During the installation the destination path must be entered (default: c:\inpa).

4.2 Configuration

The configuration of INPA is completed using two configuration files, which are contained in the \CFGDAT directory:

- Configuration file ..\CFGDAT ***inpa.ini*** determines the configuration for the INPA tools
- Configuration file ..\CFGDAT ***inpascr.ini*** is used as a default script selection file for INPA and INPALOAD

4.2.1 General program configuration

The general configuration of the INPA tools is completed using the configuration file inpa.ini.
This has the following structure (example):

```
[FAB]
FAB FILES= \INPA\FAB

[ENVIRON]
PRINTER=NO
NETWORK=NO
BARCODE=NO
PEM=YES
DTM=YES
NETWORK DATA=\inpa\bin\net.dat
DECODING TABLE_D_ALL=c:\WINELDI\CFGDATEN\FP_D.DET
DECODINT TABLE_E_E36=c:\WINELDI\CFGDATEN\FP_E_36.DET
SPLITTINGFILE=.. ..\CFGDAT\SPLIT.DET

LANGUAGE=GERMAN
EDITOR=DOS
SCRIPTSELECT=LIST
DEFINI=inpascr.ini
```

The configuration files have the format of Windows INI files. They contain various selections, whose names are in square brackets (for example: [FAB]), which in turn contain various entries. These have the following meaning:

Section [FAB]:
The path for the error mask-out files is shown here:
~~ACTIVE=YES~~
FAB FILES= \INPA\FAB

Section [ENVIRON]:

Printer settings for screenshots

PRINTER=WIN	-	Screenshots are output using the Windows print manager on the configured Windows default printer. NOTE: The printer manager must be started.
PRINTER=PEM	-	Screenshots are output on the connected WinEldi printer
PRINTER=YES	-	Screenshots are output on the connected WinEldi printer
PRINTER=NO	-	Screenshots are output in the file .\TRACE\p.trc and can be displayed using a DOS view program

Setting for WINELDI protocol

PEM=YES	-	activates the protocol label manager.

Settings for the data tables manager

DTM=YES	-	INPA can use the order data from the DTM; if DTM=NO all attempts to access the data in the DTM will be refused.

Multiple language settings

LANGUAGE=GERMAN	-	Program texts and messages in German
LANGUAGE=ENGLISH	-	Program texts and messages in English

Default setting of the error type

EDITOR=DOS	-	Script processing with the DOS editor as the default setting
EDITOR=WIN	-	Script processing with the Windows editor as the default setting

Settings for INPA and INPALOAD using the WINELDI input handler

NETWORK DATA=\inpa\bin\net.dat	-	Name of setting file for network access to the order data
DECODING TABLE_D_All=	-	Decoding tables for order data decoding
DECODING TABLE_E_E36=	-	Decoding tables for order data decoding
SPLITTINGFILE=	-	Splitting regulation for decoding (required)

(These entries are default entries and are only relevant if the input handler is activated by INPA or INPALOAD)

Type of script selection for INPA (not applicable for INPALOAD)

SCRIPTSELECT=LIST	-	Script selection from a selection list as per the relevant script select file
SCRIPTSELECT=DIALOG	-	Script selection using a default dialogue

Default script select file for INPA and INPALOAD:

DEFINI=inpascr.ini	-	Name of the default script select file in the .\CFGDAT directory

The other entries in this section are not currently in use and should not be changed.

4.2.2 Structured script selection

The script selection system is activated using the “scriptselect” function, which contains the name of the INI file to be used for the script selection process in the form of its function parameter. If an empty string “” is entered, the name contained in the configuration file under the entry DEFINI is used, All INI files for script selection must be in the ".\cfgdat" directory, the path for this is generated automatically. The script selection is structured by splitting into the appropriate sections in the script selection file. The section names can be selected freely, but must not contain underscores "_" since this is used as a separating character between the hierarchy levels. The order of the sections must use the top-down principle. Each section can contain the following entries:

DESCRIPTION=<Text>
contains the text that will be shown in the script selection structure

ENTRY=<Script file base name,Text>
contains the base name of the script file (in other words WITHOUT the extension) and the text that will be shown in the script selection structure. The two entries are separated by a comma.

Example:

Structure of the following tree:

Vehicles
Series 1
Control module 1
Control module 2
Series 2
Special tests
Brief test
Control module 3
Control module 4
Series 3
Control module 5
Control module 6

This requires the following entries in the script selection file:

```
[ROOT]
DESCRIPTION=Vehicles

[ROOT_SERIES1]
DESCRIPTION=Series 1
ENTRY=sg1,Control module 1
ENTRY=sg2,Control module 2

[ROOT_SERIES2]
DESCRIPTION=Series 2
ENTRY=sg3,Control module 3
ENTRY=sg4,Control module 4

[ROOT_SERIES2_SPECIAL]
DESCRIPTION=Special tests
ENTRY=brieftst,brief test

[ROOT_SERIES3]
DESCRIPTION=Series 3
ENTRY=sg5,Control module 5
ENTRY=sg6,Control module 6
```

In this example control module 1 is assigned script file "sg1.ips" for the interpreter and "sg1.ipo" for the loader.

4.2.3 Error mask-out files

4.2.3.1 General

For each SG version, for which one or more errors are to be masked out, there must be a separate error mask-out file. The path to the error mask-out files is defined in section [FAB] under the entry FABFILES= in INPA.INI.

4.2.3.2 Format and content

An error mask-out file for an SG version has the following name:

<*sgbd*>.FAB

where <*sgbd*> must be identical to the name of the relevant SG version or the name of the SG description file.

New error mask-out files have the following format and content:

; Comments are marked by ";" in the first column

; ***** FAB-Section **********
[FAB]
Error=<fort_nr>[<fort_nr> ...]

; ***** F-ORT-NR-Section **********
[<*fort_nr*>**]**

; List of mask-out conditions
*[***AB=**<*AB_name*>*[<AB_name>...]]*

; ***** AB-Section **********
[<*AB_name*>**]**
Bed1=<*AB_description* >
Bed2=<*AB_description* >
...
Bedn=<*AB_description* >

Description of the sections and their elements

FAB-Section [FAB]:

Error= Keyword to identify the entry of the error to be masked out error

<fort_nr> Error number

AB-Section [<*AB_name*>]:

Section, in which the mask out conditions are defined, which must **all** be satisfied to ensure that an error is masked out (AND).

<AB_name> Name of section

Bed1= Keyword to identify the entry for the first mask-out condition.

Bed2= Keyword to identify the entry for the second mask-out condition.
....

AB descriptions <*AB_description*>:

In the following the four methods of formulating mask-out conditions are described:

1. AAB : Order-related mask-out conditions (barcode specimen):

<AB_description> **AAB** *<AAB_Nr>[<AAB_Nr> ...]*

AAB Keyword to identify an order-related AB conditions

<AAB_Nr> Number of the order-related mask-out condition, which is defined in the file **AAB.FAB**. An order-related mask-out condition is satisfied if at least one of the AAB number conditions is satisfied (OR). See example in the appendix for details of the format of **AAB.FAB**.

The system checks whether the current order data satisfy the conditions. If there are no order data, for example during a rework standstill, the condition is deemed **not** to be satisfied.

2. FS-READ: Results of the FS_LESEN job

<AB_description>	**FS_READ** *<apiresult_typ> <apiresult_name> <operator> ...* *...* **"***<comparison value>***"**
FS_READ	keyword to identify conditions that refer to the results of the FS_READ job.
<apiresult_typ>	Type of result. Possible values: APIBINARY, APILONG, APICHAR, APIBYTE, APIDWORD, APIINTEGER, APIREAL, APITEXT and APIWORD
<apiresult_name>	Result name of the FS_READ job, for example F_UW1_WERT.
<operator>	Depending on <apiresult_typ> the following operators are possible: < : less then > : greater than == : equals != : not equal to
<comparison value>	Depending on <apiresult_typ> the characters entered her may be interpreted as numerical values or as a string. The entry must be within " ... "!

The system checks whether the value of the result *<apiresult_name>* read out of the control module by FS_READ satisfies the condition *<operator> <comparison value>*.

3. ORDER: Symbolic order-related mask-out conditions

<AB_description>	**ORDER** *<order data name> <order data value>*
ORDER	Keyword to identify an order data entry.
<order data name>	Order data name as shown in the decoding table (DET) (Order destination 'A' or 'B')
<order data value>	Nominal value as shown in the decoding table (DET) or which is generated during the order data decoding process.

The system compares whether the order date *<order data name>* has the *<order data value>*. If there are no order data, for example during a rework standstill, the condition is deemed **not** to be satisfied..

4. JOB: Results of arbitrary jobs

<AB_description>	**JOB** *<SGBD_name> <job_name>* **"***<job_parameter>***"** *...* *... <apiresult_typ> <apiresult_name> <operator> ...* *...* **"***<comparison value>***"]**
JOB	Keyword to identify a job entry.
<SGBD_name>	Name of the control module description file without the extension for example KOMBI36 or D_0010
<job_name>	Name of the job to be activated
<job_parameter>	Transfer parameter for the job to be activated. The entry must be within " ... ".
<apiresult_typ>	SEE ABOVE
<apiresult_name>	Result name of the jobs *<job_name>*
<operator>	SEE ABOVE
<comparison value>	SEE ABOVE

The system checks whether the result *<apiresult_name>* read out of control module *<SGBD_name>* using the job *<job_name>* satisfies the condition *<operator> <comparison value>*.
The job may only be activated if the corresponding control module *<SGBD_name>* is also fitted as per the order data. If this is not the case the condition is deemed not to be satisfied.
The job is only started once, in other words in the event of APIFALSE it is not repeated.

4.2.3.3 Processing in INPA

For each read error code F_ORT_NR the system checks whether there is a F-ORT-NR section of the same name in a file <sgdb>.FAB. If this is not the case the error is not masked out. If this is not the case the system checks whether there are any entries for mask-out conditions. If there are no such entries, the error is masked out.

If, on the other hand, AB sections have been defined, the system checks whether at least one (in other words OR) of the AB sections has been satisfied, in other words **all** (AND) AB individual conditions for an AB section are applicable. If this is the case the error is masked out. If none of the AB sections is satisfied, the error is not masked out.

One F-ORT-NR section and any number of AB sections can exist for one error code. Various F-ORT-NR sections may refer to the same AB section.

In INPA the error mask-out is used by the following functions:

INPAapiFsRead
INPAapiFsRead2
APIJobFsReadFAB
APIResultFsReadFAB.

4.2.3.4 Example

The following case should be assumed:

The error ***2, Idling adjuster/closing coil***, for the ***DME331*** should only be masked out if the order-related mask-out condition **10** formulated in **AAB.FAB** is satisfied, the error type ***Error saved after debouncing*** applies, the error occurred at an engine speed (environmental condition) **between 1500 and 2300 rpm**, according to the barcode specification the **passenger belt tensioner** flag is set and the telegram **Test_throttle valve** can be executed using control module **ASC2E**

OR

if the Hex sample '**02, 34, 0A, 0B, EF**' exists.

Errors **3, 5** and **7** should be masked out if the error type occurs **sporadically**.

All errors should be masked out that occur at an engine speed of **less than 650 rpm**.

An error mask-out file with the name **DME331.FAB** must exist with the following content:

```
;   FAB-Section ------------------------------------------------------------
[FAB]
Error= 2 3 4 5 6 7

; F-ORT-NR - Sections ----------------------------------------------------

; for error 2, 'Idling adjuster/Closing coil
[2]
AB=AB1 AB2 AB4

; for error 3
[3]
AB=AB3 AB4

; for error 4
[4]
AB=AB4

; for error 5
[5]
AB=AB3 AB4

; for error 6
[6]
AB=AB4

; for error 7
[7]
AB=AB3 AB4




; AB - Sections ----------------------------------------------------------
```

```
[AB1]
Con1=AAB 10
Con2=FS_READ APITEXT F_ART1_TEXT == "Error saved after debouncing"
Con3=FS_READ APIREAL F_UW1_WERT > "1500"
Con4=FS_READ APIREAL F_UW1_WERT < "2300"
Con5=ORDER BELT TENSIONER_BF == "1"
Con6=JOB ASC2E TEST_THOTTLE VALVE"" APITEXT JOB STATUS == "OKAY"

[AB2]
Con1=FS_READ APIBINARY F_HEX_CODE == "02, 34, 0A, 0B, EF"

[AB3]
Con1=FS_READ APITEXT F_ART5_TEXT == "Error sporadic"

; same for all error codes
[AB4]
Con1=FS_READ APIREAL F_UW1_VALUE < "650"
```

4.2.3.5 Example AAB.FAB

```
; AAB.fab
;          5.11.92, 26.3.93          J.R.Romano
;
; AAB = Order-dependent mask-out conditions
;      for Eldi/3 – FS error mask-out
;
; This file has two sections:
;
; [AB]  Mask-out conditions
; 1=AEB1,AEB2,...,AEBn    AB-Nr. (as in *fab.dat)= AEB list with [symbolic] names
; 2=EML6                              (set to "AND" within the program)
;
; [AEB] Individual mask-out conditions
; AEBi=Code,Position,Length,Sample [,N]
;
;      with:   Code: FG,E1,E2,E3, D1,D2,D3 or D4  (2 chars)
;           Position: 3 <= Position <= 25          (max. 2 chars)
;           Length:  1 <= Length <= 23         (max. 2 chars)
;           Sample: String        (´Length´chars; max. 23 chars)
;           N:     Negation of AEB  (optional,  max. 1 char)
;
; Placing the AB's under an "OR" relation is implemented using multiple entries
; All entries are not case sensitive.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


[AB]
1=E31,E31_D
2=NO
; AB's 3 to 8 for on-board computer without auxiliary ventilation or heating system
3=E32,NO_SL
4=E34,NO_SL
5=E32,NO_SH
6=E34,NO_SH
7=E31,NO_SL_31
8=E31,NO_SH_31

; AB 9 for DME 316: 36 Tank ventilation
9=SA199

; AB 10 for DME 136: 29 Speed signal
10=EMLVERB

; AB 11 for ZKE1 : ADV 01 and 02
11=NO_ADV1,NO_ADV2,NO_ADV3,NO_ADV4

; AB 12 for ZKE1 : error 8 TSH
12=NO_TSH1,NO_TSH2,NO_TSH3,NO_TSH4
```

```
; AB 13 for ZKE1 : ADV 01 and 02 D barcode
13=N_ADV1_D,N_ADV2_D,N_ADV3_D,N_ADV4_D

; AB 14 for ZKE1 : Error 8 TSH D barcode
14=N_TSH1_D,N_TSH2_D,N_TSH3_D,N_TSH4_D

; AB 15/16 for BC IV : Error auxiliary heating / ventilation system
15=NO_SL_D
16=NO_SH_D

[AEB]
NO=E1,11,2=xx
YES=E1,15,1=x,N

E32=FG,6,1=G
E34=FG,6,1=H
E31=FG,6,1=E
E31_D=FP,6,1=E

NO_SL=E3,3,1=0
NO_SH=E3,3,1=2,N

NO_SL_D=D2,19,1=0
NO_SH_D=D2,19,1=3,N

NO_SL_31=E3,4,1=0
NO_SH_31=E3,4,1=2,N

SA199=E1,14,1=A

EMLVERB=E1,15,1=0,N

NO_ADV1=E3,5,1=1,N
NO_ADV2=E3,5,1=4,N
NO_ADV3=E3,5,1=5,N
NO_ADV4=E3,5,1=7,N

NO_TSH1=E3,5,1=3,N
NO_TSH2=E3,5,1=5,N
NO_TSH3=E3,5,1=6,N
NO_TSH4=E3,5,1=7,N

N_ADV1_D=D2,15,1=4,N
N_ADV2_D=D2,15,1=5,N
N_ADV3_D=D2,15,1=6,N
N_ADV4_D=D2,15,1=7,N

N_TSH1_D=D2,15,1=1,N
N_TSH2_D=D2,15,1=3,N
N_TSH3_D=D2,15,1=5,N
N_TSH4_D=D2,15,1=7,N
```

4.2.4 Command line options

As an option, all programs in the INPA package can specify an INPA script to be processed direct in the command line, for example

INPALOAD.EXE \INPA\CFGDAT\STARTGER.IPO

With the INPA compiler ‘wildcards’, in other words space holders for file names, such as ‘*’ and ‘?’ may be used. In addition it has a batch mode function that is activated using the parameter ‘-B’. The compiler then runs in the background and does not require any user entries.

Syntax:

INPACOMP.EXE <filename.ext> -B [<errors.log>] (for batch mode in the compiler only)

5 Test procedure description files

5.1 File structure

Figure 3.1 shows the schematic structure of a test procedure description file. The entries in the items "Includes", "external function declarations" and "global data definitions" are optional; the existence of a test procedure description with the functions "inpainit ()" at the start and "inpaexit ()" at the end of the test procedure (see also 3.2 procedure principle) are essential, however.

Includes
Imported functions
Global data definitions
Test procedure description

Figure 3.1

At the time of compilation a syntax check is conducted using the test procedure description file, which may cause an error message. Test procedure description files that cannot be compiled perfectly cannot be executed during the run time.

5.1.1 Pragmas

Pragmas are commands to the compiler to control the compilation process. The syntax is as follows:

```
#pragma  <identifer>
```

The following Pragmas are available to control the character conversion during the compilation process:

#pragma winedit	The script was written with a Windows editor. No OEMToAnsi conversion is to be made.
#pragma dosedit:	The script was written with a DOS editor. The OEMToAnsi conversion is to be completed.

The above Pragmas only change the editor setting defined in the configuration file for the duration of the compilation process. After the completion of the compilation, the default editor setting defined in the configuration file is activated again.

5.1.2 Includes

Using the "#include" command in the test procedure description file it is possible to include other INPA script files. It must be noted that only one Include level will be supported, in other words nested Includes are not possible. In principle, the script elements may occur in any order in the files. However, the functions and data must be defined before use, for functions this may be done by a forwards declaration externally. Therefore it is a good idea to include header files with external declarations at the start of the script and then to define all the global data. After this script Include files with standard script elements, for example, may be included before other definitions of script elements follow.

The prototypes (external declarations) of all functions of the INPA standard library are in the file "inpa.h", which is normally included in every INPA script.

Syntax:
#include "<file name>"

Example:
#include "inpa.h"

5.1.3 Imported functions

As from version 4.4.0, INPA offers the facility to include functions from libraries outside the INPA system. The functions that are to be imported must be declared in INPA using function declarations. INPA then loads the library with the appropriate function whilst the script is running and executes the function as if it were an INPA function – for the script this is indistinguishable from other functions. Since the parameters containing direction information, data type and parameter names must be specified in the declaration, the correct use of the function parameters within the INPA script can be checked by the parser; however, the consistency of it in terms of the actual activation convention cannot be checked.

Syntax:
import [”C” | pascal] lib ”DLLName[::DLLInternerFncName]” functionname(r_info: <datentyp> , r_info: <datatype> ,..., r_info: <datatype> , [returns: <datatype>]);

Example:
import pascal lib ”user.exe” MessageBox(in: int Hwnd, in: string MyText, in: string Title, in: flags, returns: int ret);

5.1.4 External function declarations

External function declarations are used to declare library functions. Within a test procedure description file they can also be used for forwards declaration, in other words an externally declared function in the file header can be used in the file before it has been defined.
Since the parameters with direction information, data type and parameter names must be defined in the external declaration, the parser can check that the function parameters are being used correctly.

Syntax:

external functionname(r_info: <datatype> par1, r_info: <datatype> par2,..., r_info: <datatype> parx);

Example:

external text_output (in: int line, in: int column, in: string outputtext);

5.1.5 Global data definitions

Variables can be agreed, which are held in a global data pool. The following data types are possible for this:

- byte (8-bit integer)
- int (16-bit integer)
- long (32-bit integer)
- real (flow point index with double accuracy)
- bool (truth valve, TRUE or FALSE)
- string (series of characters ending in zero)

The agreed variables can be initialised; variables of the same type can be separated by commas after the type declaration.

Syntax:

<data type> varname;

Example:

```
byte a = 41;
int i;
long l = -20000;
real pressure, temp;
bool valid = FALSE;
string status = "open";
```

5.1.6 test procedure description

The test procedure description consists of at least the initialisation function "inpainit()", which starts the test procedure, and the termination function "inpaexit()", which finishes the test procedure (see also 3.2 procedure principle). The initialisation function "inpainit()" is used to activate the application-specific test procedures, which may consist of screen descriptions (see also 3.3.2 Screen displays), menu control (see also 3.3.3 Menu control) and user-defined functions (see also 3.3.1 Functions).
The use of screen displays and menu control for a test procedure is optional, in other words test procedures can also be defined without displays on the screen.

5.2 Procedure principle for tests

The procedure for the test is defined by functions from the standard library or user-defined functions. Using these functions screens can be defined, which are then combined by menu definitions to produce a test procedure tree (Figure 3.2).

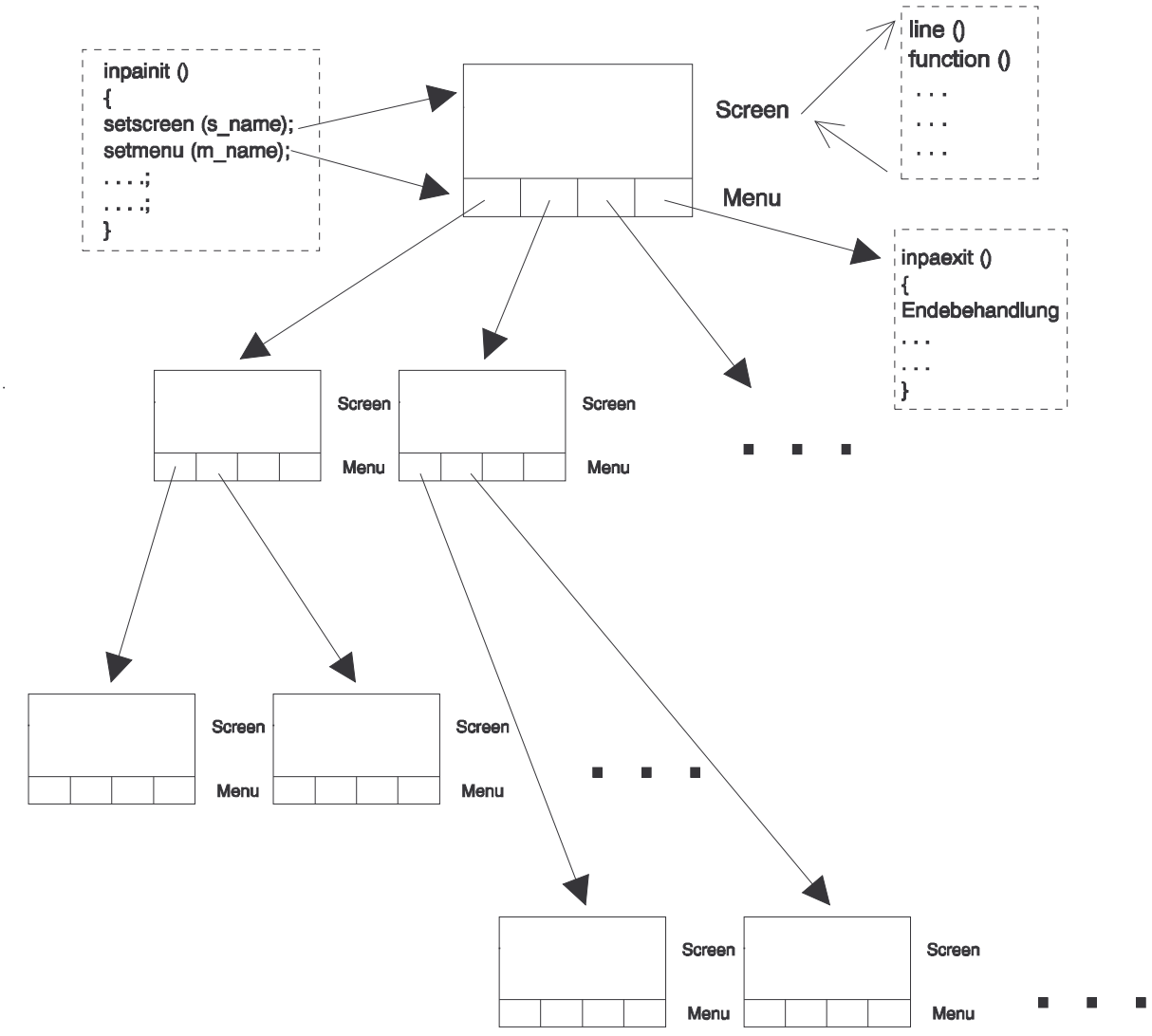


Figure 3.2

The root of the test procedure tree is determined by the function "inpainit ()", which must exist in the test procedure description file. The test procedure is terminated by activating the (standard) function "exit ()", which can be dome from any user-defined position. The function "exit ()" activates the function "inpaexit ()", in which user-specific end procedures can be defined, and then executes an end procedure for the system.

The only specification for designing a test procedure is that "inpainit ()" and "inpaexit ()" exist, although "inpaexit ()" can also be empty.

Structure of the functions "inpainit ()" and "inpaexit ()":

```
inpainit ()
{
settitle ("Title of initial screen");
setscreen (<Name of initial screen>);
setmenu (<Name of initial menu>);
}

inpaexit ()
{
/* User-specified end instructions (optional)*/
. . .
. . .
}
```

This means that test procedures can now be defined that, as shown in Figure 3.2, output displays on the screen and allow user interactions, or test procedures can be defined that run automatically without displays and action by the user, with the test results being saved in a file, for example.

One possible procedure for the production of a rework program can be described in the following steps:

- Define the screens provided in the rework program, using functions from the standard library or user-defined functions.
- Define the menus for a screen and assign the function keys on the menu bars (this defined a tree or network structure for the test procedure)
- Call up the start window and the initial menu in the function "inpainit()"
- Enter user-specific end procedure in the function "inpaexit()" (for example activate "INPAapiEnd ()")

During the run time the screen actions are executed in cycles. Pseudo-parallel to this, background procedures, which are described using state machines, are completed. This pseudo-parallel mode will be described in more detail later.

5.3 Language elements

The test procedures are defined using user-defined functions and structures. The following structures are available:

- "SCREEN" to describe the screen structure and the relevant procedures
- "MENU" to describe the possible user interactions and to link the described screens
- "STATE MACHINE" to describe procedures
- "LOGTABLE" to convert several dependent input conditions into several dependent output conditions

5.3.1 User-defined functions

User-defined functions can be activated from any point in the test procedure description. If a function is to be used before it has been declared, a forwards declaration is necessary, which is equivalent to an external function declaration.

Syntax:

```
function_name (richt_info: param_typ param_1, . . . , richt_info: param_typ param_n)
{
/* local data */
. . .
/* Instructions */
. . .
}
```

Each parameter in the declaration must be declared in full (with three coefficients). The coefficients have the following meaning:

- Direction information
  - in: only read access is available for the parameter
  - out: only write access is available for the parameter
  - inout: both read and write access is available for the parameter

- Parameter type
- byte (8-bit integer)
- int (16-bit integer)
- long (32-bit integer)
- bool (truth value, TRUE or FALSE)
- string (Series of characters ending with zero)

- Parameter name
  Can be defined by the user

5.3.2 Imported functions

As from version 4.4.0 INPA can also use functions from external libraries for execution within a script. ‘Functions from external libraries’ are functions that have been implemented outside INPA. INA supports three calls from an INPA script to a dynamic Windows library (DLL).
INPA can declare imported functions using ‘import’ instructions in the script:

Syntax:

import [<*CallConv*>] lib ”*LibName*[::LibFnc]” <*functionname*> ([<*declaration_list*>]);

Description:

Every declaration of imported functions must start with the keyword ‘import'.

As an option the call convention can be declared between the keywords ‘import’ and ‘lib’:

CallConv:
pascal (default value) INPA uses the ‘Pascal’ call convention

”C” INPA uses the ‘C’ call convention

The name of the library must be declared after the keyword ‘lib’. The following entries are possible here:

”LibName”:

”C:\TMP\myLib.dll”	Declaration of the library with an absolute or relative path
”myLib.dll”	Declaration of the library without a path (the library must either already have been loaded into the Windows memory or is in one of the paths set in the DOS environment variable PATH or is in the ‘BIN’ directors of the INPA systems
”*myLib*”	Makes reference to a library already in the computer’s memory (for example”user.exe” for a kernel library or ”api” for the EDIABAS run time system).

As an option the ‘internal’ name (*”::LibFnc”*) exported from the library can be declared. This means that the name of the function used in the INPA script may differ from the name defined in the external library.

The function name must then be defined under which the script can access the external function.

The declaration of the transfer parameters for imported functions differs from the ‘external’ declaration in three respects:

- Script variables can be declared as ‘returns’ parameters.

- The transfer of structures and their validity check is possible.

- All transferred parameters are of a temporary nature.

Whilst the last point is only of interest for developers of external libraries, the first two points are extremely important for the script developer.

General syntax of an import parameter list:

import [...] fnc([<*dir*:> <*type*> <*name*> | <dir: > structure: <name> [, ...]] [, returns: <type> <name>]);

General parameter list

The general parameter list has the same format as for the declaration of ‘external’ functions:

<dir:>

Direction of the parameter

in: The parameter is transferred to the function; this cannot or may not change the value in the variables (transfer of a temporary variable)

out: The function can write to the parameter but cannot read it. In contrast to ‘externally’ declared functions, imported functions can also read the content of the variables; the value of the variables is the same as the current value of the variable ‘out:’ and therefore corresponds more to the direction indicator ‘inout:’ for imported functions.

inout: The function can both read the parameter and write to it.

<type:>

Type of parameter

byte: 8-bit integer value
int: 16-bit integer value
long: 32-bit integer value
real: 8-byte floating decimal value
string: Series of characters ending in zero (”string”)

<name> - arbitrary designator (informative if possible).

“structure:”

The declaration type ”structure” can be used for check the type or transfer on an imported function. If a parameter with the declaration type ‘structure’ is transferred when the imported function is called, INPA will check whether the parameter value actually refers to a valid structure before the function is called up. The following situations can occur in these circumstances:

- Call the function with a valid structure value
INPA calls up the function.

- Call the function with an invalid structure value:
INPA aborts the script with an error message.

- Call with a zero structure:
INPA transfers the global script memory structure to the function

- Function returns a ZERO structure:
INPA sets the value of the structure variables to ‘-1’ (invalid)

- Function returns an invalid structure:
INPA aborts the script with an error message.

“returns:”

The declaration type ‘returns:’ must be located at the end of the declaration. It defines the value that is returned by the imported function. If no value is given the INPA will assume that it is a ”Procedure” (Pascal) or a ”void” type (”C”). it is not necessary to declare the ‘returns:’ parameter of the return value from the function does not have to be evaluated.

IMPORTANT:

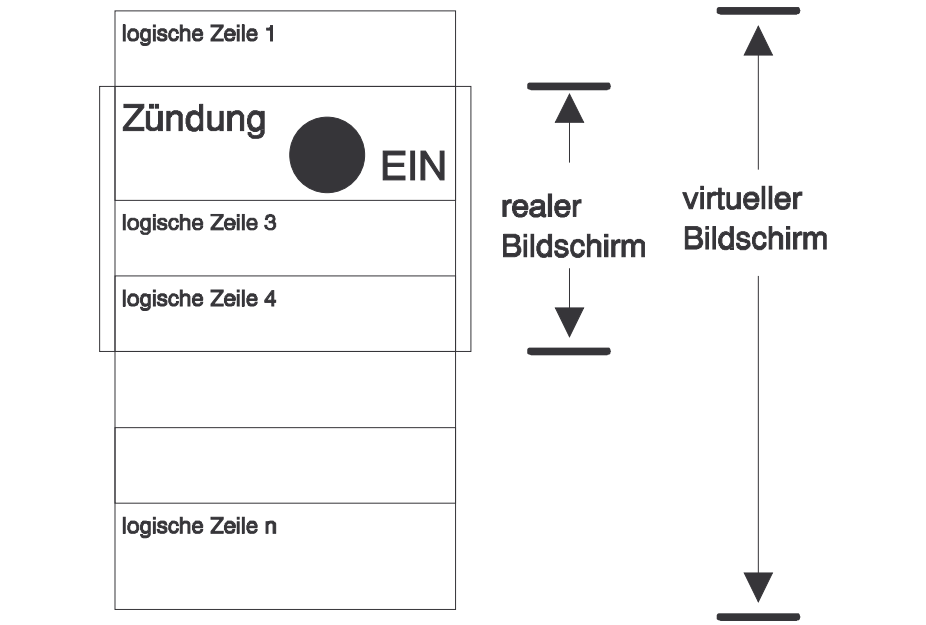
- The parameter list MUST be identical with the parameter list that is actually processed by the imported function (apart from ‘returns:’); INPA cannot identify this list itself or ‘recover’ it after an error has occurred.

- INPA does not abort the script in the event of an error immediately (as a result of the internal structure), and instead it completed the current object block in full. To prevent frequent error messages, therefore it is advisable to call only one imported function in one block .

- Each time an imported function is called, a large overhead of administration work is generated inside INPA. If imported functions are called very frequently, the processing speed of the script may suffer.

5.3.3 Screen display

The screen displays are written using SCREEN constructs. A SCREEN constitutes a named structure that is activated using a "setscreen (s_name, FrequFlag)" call. "s_name" stands for the name of the SCREEN, FrequFlag affects the cyclic processing (FrequFlag=TRUE for cyclic processing of the screen, FrequFlag=FALSE for one-off processing).
SCREENs describe virtual screens with an unlimited number of logical lines (LINE). Logical lines contain the user-defined actions and may consist of several display elements and therefore of several physical screen lines. The coordinate declarations for the display functions of a logical line are relative to the start of the corresponding logical line.
During the run time the visible area may be moved to a virtual screen.



SCREENs are processed in cycles (if FrequFlag=TRUE when they are activated using setscreen), whereby functions can be declared that belong to the SCREEN and are called up for each run. The functions that belong to the LINEs are only called if the corresponding LINE is in the visible area.

Example: LINE("LineSelectText", "API-String") { ..... }

Each logical line has a unique name "LineSelectText" and is initiated by the LINE statement. Logical lines can be selected during the run time using their name from a selection menu for increased screen display.
The API strings are used to optimise the result queries in EDIABAS mode. They contain the EDIABAS result parameter used within the relevant logical line. The API strings of all visible logical lines can be scanned in concatenated form using the library function "getapistring(...)" so that they can then be transferred during the subsequent EDIABAS result query, which produces temporal optimisation.
A logical line may contain a CONTROL block to implement control functions. The user functions bracketed by this CONTROL block are executed during the run time if requested by the user and if the logical line is in the visible area of the screen. One use for a CONTROL block, for example, may be to activate components and then output the result.

Example of a typical SCREEN:

```
SCREEN s_main ()
{
  text( 1, 0, "Identification data control module XYZ");
  text( 3, 0, "");
  INPAapiJob("XYZ","IDENT","","");            /* Called in every cycle */
  INPAapiCheckJobStatus("OKAY");

  LINE ( "Part number", "")                   /* Name of the line for enlarged display during the run time */
  {
    text( 0, 1, "Part number: ");             /* Output the text “Part number” in physical line 0, column 1*/
    INPAapiResultText(t0,"ID_NR",1,"");       /* Fetch the part number from the interface, place in t0 */
    textout( t0, 0, 35);                      /* Output the part number in physical line 0, column 35 */
    text( 2, 0, "");                          /* Empty line */
  }

  LINE ( "Datum_KW", "")                      /* Name of the line for enlarged display during the run time */
  {
    text( 0, 1, "Production date week: "); /* Output the text “Production date week” in physical line 0, column 1*/
    INPAapiResultText(t1,"ID_DATUM_KW",1,"");/* Fetch from the interface, place in t1 */
    textout( t1, 0, 35);                      /* Output t1*/
    text( 2, 0, "");                          /* Empty line */
  }

  LINE ( "Datum_JAHR", "")                    /* Name of the line for enlarged display during the run time */
  {
    text( 0, 1, "Herstelldatum Jahr: ");      /* Output the text “Production date year” in physical line 0, column 1*/
    INPAapiResultText(t3,"ID_DATUM_JAHR",1,"");/* Fetch from the interface, place in t3 */
    textout( t3, 0, 35);                      /* Output t3*/
    text( 2, 0, "");                          /* Empty line */
  }

}
```

5.3.4 Menu control

The menus (MENU) describe the actions defined by the user that are to be executed if a function key is pressed during the run time. Library functions may also be included in these actions, such as changing MENUs or SCREENs. This allows test procedures to be described as tree or network structures.
For a menu it is possible to declare INIT functions within a structure that are to be executed once when the menu is activated. For example the function "setmenutitle ("title text")" may be used to declare a title for the menu.
Up to 20 function keys, F1..F10 and <Shift>F1..F10 can be assigned using ITEM constructs in each menu. These ITEM constructs define the actions to be completed when the user presses the function key. The number and labelling of the function key is declared as nr, or "Itemtext", whereby numbers 1..10 refer to keys <F1>..<F10> and numbers 11..20 to the key combinations <SHIFT><F1>...<SHIFT><F10>. If the <SHIFT> key is pressed the function bar is automatically updated. An arbitrary number of function calls may be included in an ITEM construct.

Example:

```
MENU m_main () /* Main menu */
{
   INIT {                                  /* Initialisation is completed once when the MENU is started */
      INPAapiInit();
      setmenutitle( "Default menu");
   }

 ITEM( 2 , "Ident")  {                    /* Label function key F2 with "Ident" */
      setscreen( s_ident ,TRUE);           /*  Call the Ident SCREEN */
      setmenu( m_ident );                  /* and the Ident MENU */
   }

   ITEM( 5 , "Status")  {                 /* Label function key F5 with "Status" */
      setscreen( s_status ,TRUE);          /*  Call the Status SCREEN */
      setmenu( m_status );                 /* and the Status MENU */
   }

   ITEM( 6 , "Control")  {                /* Label function key F6 with "Control" */
      setscreen( cs_control ,TRUE);        /* Initiate the call for the SCREEN for the components */
      setmenu( m_control );                /* and the corresponding MENU */
      }

  ITEM( 20 , "End")  {                    /* Function key <SHIFT><F10>, exit INPA */
      exit();
      }
}
```

5.3.5 State machines

State machines are used for the simple formulation of interruptible background actions, which are operated in pseudo-parallel with the SCREENs and the function keys. In the state machines blocks with user code are always executed in full before a change to the parallel object (another state machine, SCREEN or function key) takes place. Either a logical line of a SCREEN or an individual status of a state machine, or the code assigned to a function key is regarded as a block.
For example the code of logical line 1 is executed, then that of status 3 and then logical line 4 and then the code of status 3 again since no change of status has taken place, etc. If a function key event takes

place in between these, its code is executed with the highest priority, in other words immediately after the termination of the code block currently being processed.

There are library functions available to activate or change state machines, to forward the status and to call "sub-state machines". The use of "sub-state machines" allows the logical division of tasks that must be interruptible in the same way used for functions (call hierarchy). This means that by calling the library function "callstate machine" it is possible to make a state machine into a new state machine in the form of a sub-state machine. The actions of this state machine can be interrupted in contrast to a pure function, in other words a change to SCREEN processing and the treatment of the keyboard events takes place. When the task of the "sub-state machine" is finished, the system returns to the code point in the superior state machine by calling the library function "returnstate machine", from where the sub-state machine was called.
The nesting depth of the state machines is not limited.

Each state machine has an initialisation status ("INIT"), which must always be available. The user code contained in this is executed after the activation of the state machine. Branching to the next status takes place in this case, too, by calling the library function "setstate". In addition to this initialisation status an unlimited number of status conditions can be included in the state machine. Each status is identified by a name that may be freely defined by the user, although it must be ensured that the status names inside the state machine are unique. Each status is assigned a block with user code. Each time the state machine is executed in cycles, the code block of the currently active status is executed. If there is no branching in the status, the same code block is executed during the next processing run.

Example:

```
STATE MACHINE sm_motorlaufsignal_ein()
{
        int try_repeat_counter;
        int TRY_COUNTER_MAX;
        string  job_state;
        string  job_state;
        bool    motor_lauf;

      INIT
      {

            try_repeat_counter   = 0;
            TRY_COUNTER_MAX   =   3;

            delete_screen();
            print_line("GSA_TEST", 1,10);
            print_line("Prüfung Motorlaufsignal", 4, 10);
            action_box_open("Bitte Motor einschalten");
            delay(2000);

            setstate(MOTORLAUFSIGNAL_TELEGRAMM_SENDEN);
      }


      MOTORLAUFSIGNAL_TELEGRAMM_SENDEN
      {

        if (try_repeat_counter < TRY_COUNTER_MAX)
          {
            INPAapiJob("GSA","STATUS_LESEN","","");
            INPAapiResultText(job_state, "JOB_STATUS", 1, "");
            INPAapiResultDigital(motor_lauf, "STAT_MOTOR_SIGNAL_EIN", 1);
            try_repeat_counter = try_repeat_counter + 1;
```

```
            setstate(AUSWERTEN_MOTORLAUFSIGNAL);
        }
       else
        {
            callstate machine( sm_fehlerbehandlung);
            . . .
        }
    }


    AUSWERTEN_MOTORLAUFSIGNAL
    {
        . . .
    }
}
```

5.3.6 Logic tables

Logic tables allow the formulation of Boolean expressions and therefore the conversion of linked input conditions into lined output values. This is done with the help of binary constants that are identified by a prefix "0y". The bits of the binary constants correspond to the sequence of the logic table parameters and may assume the following values:

"1" corresponds to "TRUE"
"0" corresponds to "FALSE"
"x" or "X" corresponds to "arbitrary"

In the last (and only the last) line the input definitions may assume the keyword "OTHER" in case none of the previous input definitions was accurate. If the keyword "OTHER" is not declared, and none of the input definitions applies, the output values remain unchanged.
Processing during the run time takes place by the line by line comparison of the input variables. As soon as a definition applies, the comparison is ended and the output variables are set accordingly.
The defined conditions are not checked to ensure that they are not contradictory or for overlaps.

Note on syntax:
The output parameters are separated from the input parameters in the LOGTABLE header by commas. The parameters themselves are not separated by commas but by blanks (see example).

Example:

```
LOGTABLE  testlog (out: bool out1 out2, in: bool in1 in2 in3)
        {
        0y00:    0y000; /* all input variables FALSE, then out1, out2 also both FALSE */
        0y01:    0y101; /* in1, in3 TRUE, in2 FALSE, then out1 FALSE, out2 TRUE */
        0y10:    0y010; /* . . . */
        0y11:    OTHER; /* in all other cases out1 = TRUE, out2 = TRUE */
        }


Example for calling this LOGTABLE testlog:

testfunction ()
        {
        bool out1, out2;
        . . .

        testlog (out1, out2, TRUE, FALSE, TRUE) /* leads to out1 = FALSE, out2 = TRUE */

        . . .
        }
```

5.3.7 Control structures

The control structures listed below can be used to formulate the test procedures.

- Assignment
  = (EQUALS, PRODUCES)

  See Appendix A: Language definition for syntax
  Example:
  ```
  x1 = 5;
  ```

- Comparison
  < (LESS THAN), <= (LESS THAN OR EQUAL TO), != (NOT EQUAL TO), == (EQUAL TO), >= (GREATER THAN OR EQUAL TO), > (GREATER THAN)

  See Appendix A: Language definition for syntax
  Example:
  ```
  if (x1 == 5)
  {
  . . .
  }
  ```

- Logical links for digital variables:
  ! (NOT), && (LOGICAL AND), | | (LOGICAL OR), ^ ^ (EXCLUSIVE OR)

- Binary links:
  & (BINARY AND), | (BINARY OR), ^ (EXCLUSIVE OR)
  See Appendix A: Language definition for syntax
  Example:
  ```
  x1 = ((x2 & x3) |  x4 *3;
  ```

- Loops
  WHILE (expression)
  {
  statements;
  }

  See Appendix A: Language definition for syntax
  Example:
  ```
  while (x1 > x2)
  {
  x1 = x1 - x2;
  }
  ```

- Conditional execution
  IF (expression)
  {
  statements;
  }
  ELSE
  {
  statements;
  }

See Appendix A: Language definition for syntax
Example:

```
if (x1 > 0)
{
x2 = x1;
}
else
{
x2 = 0;
}
```

5.3.8 Functions of the standard library

The standard library contains all the hard-coded INPA library functions. These comprise the following groups:

- System functions
- Conversion functions
- String functions
- Interface functions for the protocol label manager
- Input functions
- Output functions
- File access functions
- User box functions
- Interface functions for Windows
- Memory manipulation functions
- File viewer functions
- EDIABAS functions
- Interface functions for the data table manager
- String array functions
- RK512 functions
- WINELDI function(s)

The individual functions in the above groups are described briefly below. The precise function declaration is shown in the prototypes in the header file inpa.h in directory .\sgdat.

- INPA system functions

-setmenu (menu)
Activate a menu (see also 3.3.3 Menu control)

-setscreen (screen, frequ)
Activate a screen for a single (frequ = FALSE) or multiple (frequ = TRUE) run (see also 3.3.2 Screen displays)

-setmenutitle (menutitle)
Declaration of the menu title

-settitle (screentitle)
Declaration of the screen title

-setstate (state)
Set a status in a state machine

-setstate machine (statemach)
Call up a state machine

-callstate machine (statemach)
Call up a state machine as a sub-program

-returnstate machine ()
Return from a state machine as a sub-program

-settimer (timernum, time)
Set timer with number timernum with time time in ms

-testtimer (timernum, flag)
Check time with number timernum, if flag=TRUE the timer has expired

-delay(ms)
INPA stops the script processing for ‘ms’ – milliseconds (negative values correspond to (65536 - |x|) ms

-getdate (date)
Get the system date in the string 'date'. The date is filed in the format dd.mm.yyyy

-gettime (time)
Get the system time in the string 'time'. The time is filed in the format hh:mm:ss.

-setjobstatus (jobstat)
Sets the job status for WINELDI

-exit()
Quit the program

-exitwindows()
Quit Windows

-scriptselect(ScriptIniName)
Call script select using the declared script selection file

-scriptchange(ScriptFileName)
End the current script and start the script 'SkriptFileName'. The script names may contain absolute or relative path names; the suffixes *.IPO and *.IPS are added automatically. The current code list is processed in full before the specified script is started.

-select(MultSelectFlag)
Select logical lines; when MultSelectFlag=FALSE only one line can be selected, when MultSelectFlag=TRUE more than one line can be selected.
Selection combinations can be saved under a logical name (set). For this purpose, after compiling the selection enter the set name in the edit field of the dialogue box and then click on "Save". After this the set name will appear in the set list below it. Saved sets can be selected by positioning the cursor on the required set name and pressing Return. When positioning the cursor the selection display is updated so that the selection that belongs to the set is displayed immediately. The set can be saved for each screen and script in an INI file with the base name of the script and the extension "INI" in directory SGDAT.

-setitem(nr, text, enable)
Change the menu name during the run time. The text in the selection bar below the menu point 'nr' is changed to 'text'. 'Enable' shows whether the menu point can be selected or is displayed in grey.

-deselect()
Return from the enlarged display to the overall display

- control()
  Activate the control functions for the active screen

- start()
  Start the SCREEN processing

- stop()
  Stop the SCREEN processing

- getapistring(ArgNumFlag, FullScreenFlag, ApiString)
  Find the API result parameters for the visible area, if FullScreenFlag=TRUE for the full screen; preceded by the number of result parameters with a semi-colon if ArgNumFlag=TRUE, not if ArgNumFlag=FALSE.

- togglelist(MultipleSelectFlag, ArgNumFlag, ApiToggleString)
  Display a list box to select logical lines. The structure of a string with the API result parameters for the selected logical lines, preceded by the number of result parameters with a semi-colon if ArgNumFlag=TRUE, not if ArgNumFlag=FALSE.
  If MultipleSelectFlag= FALSE only one logical line can be selected; if MultipleSelectFlag= TRUE several may be selected.

- printscreen()
  Print the visible screen area using the printer settings in the configuration file

- setcolor(front, background)
  Configure the screen and text colour, colour numbers: 0=white, 1=black, 2=light grey, 3=grey, 4=bright red, 5=wine red, 6=red-violet, 7=red-lilac, 8=bright yellow, 9=olive, 10=birght green, 11=dark green, 12=light turquoise, 13=dark turquoise, 14=bright blue, 15=blue

- printfile(FileName, PrinterName, PrinterPort)
  Print the specified file on the declared Windows PRINTER

- Conversion functions

  - realtostring(r, format, s);
    Converts a real numbers into a string

  - stringtoreal(s , r);
    converts a string into a real number.
    The following rules apply

s < ”Min-Real”	->	r = Min-Real;
s > ”Max-Real”	->	r = Max-Real;
s = ”abc”	->	r = 0.0;
s = ”123 456”	->	r = 123.0;
s = ”123.456.78”	->	r = 123.456;

  - inttostring(i, s);
    converts an integer into a string

  - stringtoint (s, i)

converts a string into an integer.
The following rules apply:

s < ”Min-Integer”	->	i = Min-Integer;
s > ”Max-Integer”	->	i = Max-Integer;
s = ”abc”	->	i = 0;
s = ”123 456”	->	i = 123;
s = ”0x10”	->	i = 16 (hexadecimal; max.4 numbers);
s = ”010e2”	->	i = 100 ;
s = "0y11110101	->	i = 245 (binary, 16 numbers)
s = "0x8000"	->	i = -32768
s = "±0xffe2"	->	i = 0
s = "±0y1111"	->	i = 0

- inttoreal (i, r)
  converts an integer into a real number

- realtoint (r, i)
  converts a real number into an integer
  The following rules apply:

r > Max-Integer	->	i = Max-Integer
r < Max-Integer	->	i = Min-Integer

- bytetoint (c, i)
  converts an 8-bit integer in a 16-bit integer

- inttolong (i, l)
  converts a 16-bit integer into a 32-bit integer

- longtoreal (l, r)
  converts a 32-bit integer into a floating decimal value

- hexconvert(HexString, high, mid, low, seg);
  breaks a hex string into integer values

The following apply for all conversion functions in Windows 3.x (16-bit):

Max-Integer	=	*+32767;*
Min-integer	=	*-32768;*
Max-Real	=	*3.4E +38*
Min-Real	=	*-3.4E +38*
|Minimum value|	=	*3.4E-38*

- String functions

  - strcat(DestStr, SrcStr1, SrcStr2);
    Concatenation of SrcStr1 and SrcStr2; the result is filed in DestStr

  - strlen(length, SrcStr);
    Supplies the number of characters in the transferred string SrcStr

- midstr(DestStr, SrcStr, Index, Count);
  The number Count of characters from position Index of the input string SrcStr is filed in the destination string DestStr.

- PEM calls

  - PEMInitialisiere (Result, WinEldiVersion, Pruefstand, RechnerNr);
    Initialise the PEM with setup data

  - PEMProtokollKopf (Result, Jobtabelle, JAT_Version, Datum, Zeit, FzgTyp, FgNr);
    Protocol header to temporary file

  - PEMProtokollZeile (Result, Zeile);
    Single line empty form to temporary file

  - PEMSGZ_Kopfzeile(Result, SGVar, LogUnit, ZeilenSG);
    Title of a JAT line to temporary file

  - PEMTrennLinie(Result);
    Form separating line to temporary file

  - PEMEndLinie(Result);
    Form end line to temporary file

  - PEMLoescheTabZeilenPuffer(Result);
    If JAT line OK, delete temp. JAT lines file

  - PEMUebertrageTabZeilenPuffer(Result);
    If JAT line not OK, add temp. JAT lines file to temporary file

  - PEMProtokollAusgabe(Result);
    temp prot. file to prot PRINTER

  - PEMDruckeEtikett(Result, Datum, Zeit);
    Label with assigned data to PRINTER

  - PEMPrintFormular(Result, FormularName);
    Print form

  - PEMPrinter_ff(Result);
    Form page feed

  - PEMLoad_formular(Result, FileName, FormularName);
    Load form

  - PEMForget_formular(Result, FormularName);
    Forget loaded form

- Input functions

General: All functions for inputting numbers offer the content of the relevant output variable as a default value if this is within the defined limit values. If it is outside the limit values no default value is displayed. The HEX values are returned as a string with an initial "0x".

- getinputstate(InputState)
  Return of the internally saved input status that provides information on the type of termination of input functions. The input status may assume the following values:

  0 - Input was terminated with OK
  1 - Input was terminated with abort
  2 - No input function has been called as yet

- inputtext(Variable, Titel, Text)
  Input a text in a dialogue box with a specified title and note text in the declared text variable

- inputnum(Variable, Titel, Text, MinWert, MaxWert)
  Input of a numeric value in a dialogue box with a specified title and note text in the declared numeric variable. Plausibility check of the input (range between min and max values).

- inputint(Variable, Titel, Text, MinWert, MaxWert)
  Input of a decimal integer value in a dialogue box with a specified title and note text in the declared text variable. Plausibility check of the input (range between min and max values).

- inputhex(Variable, Titel, Text, MinWert, MaxWert)
  Input of a hexadecimal integer value in a dialogue box with a specified title and note text in the declared text variable. Plausibility check of the input (range between min and max values).

- input2text(Variable1, Variable2, Titel, Text, Text1, Text2)
  Input of two texts in a dialogue box with a specified title, note text and text field before each input field in the specified text variables

- input2hexnum(hex_Var, num_Var, Titel, Text, Text1, Text2, MinWert1, MaxWert1, MinWert2, MaxWert2)
  Input of a hexadecimal and a numeric value in a dialogue box with a specified title, note text and text field before each input field in the specified text variable (hexadecimal) and numeric variable plausibility check of the inputs (range between min value x and max value x).

- input2int(Var1, Var2, Titel, Text, Text1, Text2, MinWert1, MaxWert1, MinWert2, MaxWert2)
  Input of two decimal integer values in a dialogue box with a specified title, note text and text field before each input field in the specified text variable. Plausibility check of the inputs (range between min value x and max value x).

- input2hex(hex_Var1, hex_Var2, Titel, Text, Text1, Text2, MinWert1, MaxWert1, MinWert2, MaxWert2)

Input of two hexadecimal values in a dialogue box with a specified title, note text and text field before each input field in the specified text variable. Plausibility check of the inputs (range between min value x and max value x).

- inputdigital(Variable, Titel, Text, FalschText, Wahrtext)
  Input of a decimal value in a dialogue box with a specified title, note text and texts for TRUE and false in the specified digital variable.

- Output functions

  - text(Zeile, Spalte, Text)
    Output of a text in the specified line and column

  - textout(Variable, Zeile, Spalte)
    Output of the text contained in the specified text variable in the specified line and column

  - ftextout(text, row, col, textsize, textattr)
    Formatted text output of the transferred text in the appropriate line and column of the active screen with the specified font size and text attributes.
    Possible font sizes:
    0 - standard
    1 – medium large
    2 - large
    Possible text attributes (can be combined with a logical OR):
    0 – standard font (no text attributes)
    1 - bold
    2 - italic
    4 - underlined

  - digitalout(Variable, Zeile, Spalte, Falschtext, Wahrtext)
    Output of the text contained in the specified digital variables in the specified line and column. The output takes the form of a circle for false and a dot for true. The specified status text (true/false status) is output after the symbol.

  - analogout(Variable, Zeile, Spalte, Min, Max, MinGültig, MaxGültig, Format)
    Output of the value contained in the specified numeric variables in the specified line and column. The output takes the form of a bar with an invalid range with a red background and the valid range with a green background. The value is output next to the bar as a number value. The pre- decimal and post-decimal places can be specified using the format variables <Values>.<Post-decimal places>" (for example "4.2"), the default format is set by declaring an empty string "".

  - multianalogout(Variable, Zeile, Spalte, Min, Max, MinGültig, MaxGültig, Format, mode)
    Output of the value contained in the specified numeric variables in the same way as the function 'analogout' (mode != 1). if the parameter 'mode' == 1, instead of the bar a triangle is output as the value indicator and the zero line is displayed.

  - hexdump(StartAdresse, AnzahlZeilen, Zeile, Spalte)
    Output of a hexdump with AnzahlZeilen lines from the address StartAdresse in the specified line and column.

- ftextclear(text, row, col, textsize, textattr)
  Clear the text output with the function "ftextout" in the appropriate line and column of the active screen using the specified font size and text attributes; the screen area being cleared is calculated on the basis of the transferred parameters.

- clearrect(row, col, height, width)
  Clear the rectangular screen area with the specified coordinates

- blankscreen()
  Clear the screen

- messagebox(Titel, Text)
  Output a note text in a dialogue box with the specified title.

- infobox(Titel, Text)
  Output of an information text in a dialogue box with the specified title.

- File access functions

  - fileopen(Datei, Modus)
    Open a file in the specified mode:
    "w" = Create a new file; if the file already exists, delete the file
    "a" = Add to an existing file; if the file does not yet exist create a new file
    ”r”= Open a file for reading.

  - fileclose()
    Close file

  - filewrite(String)
    Add the specified string to the end of the file that has been opened for writing; a CRLF is inserted after each filewrite

  - fileread(String, Bool)
    Read the string from a file that has been opened for reading. One line is read up to LF: CR control sequences are not read. At the end of the file the parameter ‘Bool’ = true (EOF).

- User box functions

  - userboxopen(Box_Nummer, Zeile, Spalte, Anzahl Zeilen, Anzahl Spalten, Titel, Text)
    Open a dialogue box with the assigned Box_Number [0..11], at the specified screen position (line,column) and with the specified dimensions (number of lines, number of columns) with the specified title and text. The Box_Number and title affect the frame of the dialogue box:
    Box_Number 0 to 3, with title: frame bold
    Box_Number 0 to 3, without title: frame normal
    Box_Number 4 to 7, with and without title: frame normal
    Box_Number 8 to 11, with title: frame normal
    Box_Number 8 to 11, without title: no frame

- userboxclose(Nummer)
  Close the user box with the specified number

- userboxftextout(BoxNum, text, row, col, textsize, textattr)
  Formatted text output of the transferred text in the appropriate line and column of the user box with the number BoxNum with the specified font size and text attributes
  Possible font sizes:
  0 - standard
  1 – medium large
  2 - large
  Possible text attributes (can be combined with a logical OR):
  0 - standard font (no text attributes)
  1 - bold
  2 - italic
  4 - underlined
  8 - centred vertically, line argument is dummy
  16 - centred horizontally, column argument is dummy

  The attributes can be combined. When using centring attributes the corresponding line and column parameters are dummy values, in other words they have no function since the centring function calculates the real line/column values automatically.

- userboxclear(BoxNum)
  Clear the content of the user box with the specified number

- userboxsetcolor(BoxNum, Vordergrund, Hintergrund)
  Configure the front and background colour in a user box, colour numbers: 0=white, 1=black, 2=light grey, 3=grey, 4=bright red, 5=wine red, 6=red-violet, 7=red-lilac, 8=bright yellow, 9=olive, 10=birght green, 11=dark green, 12=light turquoise, 13=dark turquoise, 14=bright blue, 15=blue

- Interface functions to Windows

- winhelp(helpfile)
  Call the Windows help function to display the specified help file

- winhelpkey(helpfile, key)
  Call the Windows help function to display the specified help file with the transferred keyword

- callwin(filename)
  Windows call

- Memory manipulation routines

- CreateStructure(handle, length)
  Create a structure of the specified length and return the handle. 'handle' is a pointer to a structure of Windows. Peculiarity: The handle '0' is created by INPA itself and remains in forced until INPA is quit..

- SetStructureMode(ReadWrite)
  Set the access mode to the following structure functions

- StructureByte(handle, at, value)
  Read / write a byte at the position ‘at’ in the ‘handle’ structure

- StructureInt(handle, at, value)
  Read / write an integer value at the position ‘at’ in the ‘handle’ structure

- StructureLong(handle, at, value)
  Read / write a long integer at the position ‘at’ in the ‘handle’ structure

- StructureString(handle, at, length, value)
  Read / write a string with a maximum length ‘length’ at the position ‘at’ in the ‘handle’ structure

- File viewer functions

  - viewopen(datei)
    Open the specified file and display it using the file viewer
    In the file view (INPA function "viewopen") it is possible to scroll from one error to the next in protocols of the function INPAapiFsLesen using the key combination <SHIFT + LINEUP> and to the previous error using the combination <SHIFT + LINEDOWN>.

  - viewclose()
    Close the file viewer and the displayed file

- EDIABAS functions with error processing by INPA

  - INPAapiInit()
    apiInit call

  - INPAapiEnd()
    apiEnd call

  - INPAapiJob(ecu, job, para, result)
    apiJob call

  - INPAapiResultText(Textvar, abzufrag. Ergebnis, Ergebnissatz, Format zur Konvertierung)
    apiResultText call and file the result in the specified text variable

  - INPAapiResultDigital(Digitalvariablen, abzufragendes Ergebnis, Ergebnissatz)
    apiResultText call and file the result in the specified digital variable (bool)

  - INPAapiResultInt(Wert der Variablen, abzufragendes Ergebnis, Ergebnissatz)
    apiResultText call and file the result in the specified variables

  - INPAapiResultSets(ErgVar)
    apiResultSets sell; the number of result sets is supplied in ErgVar

  - INPAapiResultAnalog(Analogvariablen, abzufragendes Ergebnis, Ergebnissatz)
    apiResultReal call and file the result in the specified analogue variable (real)

- INPAapiResultBinary(Resultstring, Ergebnissatz)
  apiResultBinary call with a result string and result set; the data supplied by EDIABAS are filed in a binary data buffer that can be output using the hexdump (...) function

- INPAapiCheckJobStatus(OKAY-Text)
  apiResultText scan of "JOB_STATUS" in result set 1 and checking the result text using the transferred OKAY text. If the message box difference is active, the rear job status is declared and cyclic processing is stopped

- INPAapiFsLesen(ecu, datei)
  Error memory read for the specified control module ecu and file in the file with the specified file name.

- INPAapiFsLesen2(ecu, datei)
  Like INPAapiFsLesen, but this function has a different output format and cannot interpret FsMode = 0x40.

- INPAapiFsMode(FsMode, FsFileMode, PreInfoFile, PostInfoFile, ApiFsJobName)
  Set the mode of operation for the functions INPAapiFsLesen(...) and INPAapiFsLesen2(...). The settings are maintained during the completion of a script, when the script is changed the default values are set automatically.

  Parameters:

  FsMode – bit-coded value for the selective display of the detail information in the error protocol
  Mask 0x01 = Display the error locations if set
  Mask 0x02 = Display the environmental conditions
  Mask 0x04 = Display the error types if set
  Mask 0x08 = Display the environment value if set (combined with 0x02)
  Mask 0x10 = Display the environment unit if set (combined with 0x02)
  Mask 0x20 = Display the error frequency if set
  Mask 0x40 = Display the error hex code (only INPAapiFsLesen)
  Mask 0x80 = Display note texts if set

  Default value: 0xFF = Display everything

  Note: The masks 0x08 and 0x10 only supply results in combination with mask 0x02!!!!!

FsFileMode - Parameter to control the error protocol procedure when the function INPAapiFsLesen(...) is called:
Value "w" = Create a new error protocol when the function INPAapiFsLesen(...) is called
Value "a" = Append the error protocol to an existing one.
Default value: "w" = Always create a new error protocol

PreInfoFile - Name of the file that is inserted into the error protocol BEFORE the detail information; transfer an empty string “” if no file is to be inserted.
Default value: "" = Insert no file

PostInfoFile - Name of the file that is inserted into the error protocol AFTER the detail information; transfer an empty string “” if no file is to be inserted.
Default value: "" = Insert no file

ApiFsJobName - Name of the API job to read the error memory ApiFsJobName = "" : Default name "FS_LESEN" is used ApiFsJobName = "<name>" : The specified name is used. If "IS_LESEN” is entered, ”READ SHADOW MEMORY” is output as the protocol title, otherwise the title is ”READ ERROR MEMORY”.

- EDIABAS – Direct connection without error processing

-INP1apiInit(result)
apiInit call; result states whether the call could be completed successfully

-INP1apiEnd()
apiEnd call

- INP1apiJob(ecu, job, para, result)
apiJob call; result states whether the call could be completed successfully

- INP1apiState(Apistate)
apiState call; scan the status of the last api call to have been made

- INP1apiResultText(result, Textvar, abzufrag. Ergebnis, Ergebnissatz, Format zur Konvertierung)
apiResultText call and file the result in the specified text variable; result states whether the call could be completed successfully

- INP1apiResultInt(result, Wert der Variablen, abzufragendes Ergebnis, Ergebnissatz)
apiResultInt call and file the result in the specified variable; result states whether the call could be completed successfully

- INP1apiResultSets(result, Ergebnissatz)

apiResultSets call; the number of result sets is supplied in ErgVar; result states whether the call could be completed successfully

- INP1apiResultReal(result, Wert der Variablen, abzufragendes Ergebnis, Ergebnissatz)
  apiResultInt call and file the result in the specified variables; result states whether the call could be completed successfully

- INP1apiResultBinary(result, Ergebnisstring, Ergebnissatz)
  apiResultBinary call with a result string and result set; the data supplied by EDIABAS are filed in a binary data buffer that can be output using the function hexdump (...);result states whether the call could be completed successfully

- INP1apiErrorCode(Errorcode)
  Establishment of the current api error code

- INP1apiErrorText(Errortext)
  Establishment of the current api error text

- GetBinaryDataString(DataString, DataStringLen);
  Return of the buffer content of EDIABAS binary data in the form of a hexadecimal string. The string contains two characters for each data byte. The string and its length are returned.

- DTM calls (only available when INPA is called from WinEldi):

  - DTMFindLogUnit (out: bool rc, in: string LogUnit)
    Find the transferred LogUnit in the data table and return the result rc

  - DTMGetSGVar (out: string SGVar, in: string SGArt)
    Establish the SGVariant for the transferred SGType

  - DTMGetSGArt (out: string SGArt, in: string SGVar)
    Establish the SGType for the transferred SGVariant

  - DTMGetVarWert (out: string VarWert, in: string VarName)
    Return the value of the transferred variable

  - DTMSetupGetVarWert (out: string VarWert, in: string VarName)
    Return the value of the transferred setup variables

  - DTMSetupGetStartPosition ()
    Set the search position at the start of the data table with the setup variables

  - DTMSetupGetNextAssoc (out: bool rc, inout: string VarName, inout: string VarWert)
    Return the data from the current table entry, in other words name and value of the setup variables and position on the next table entry. When you reach the end of the table and the current table entry therefore does not contain any data, the result FALSE is returned.

  - DTMLogUnitEintragen (in: string LogUnit);
    Enters a logical unit into the table

- DTMSGEintragen (in: string SGArt, in: string SGVar);
  Enters a control module into the table

- DTMLoescheAuftrag ();
  Deletes a job from the table

- DTMVariableEintragen (in: string VarName, in: string VarWert);
  Enters a variable into the table

- DTMVariableLoeschen (out: bool rc, in: string VarName);
  Deletes a variable from the table and returns the result

- DTMLoescheAlleVariablen ();
  Deletes all variables from the table

- DTMSetupVariableEintragen (in: string VarName, in: string VarWert);
  Enters a variable into the table

- DTMSetupVariableLoeschen (out: bool rc, in: string VarName);
  Deletes a variable from the table and returns the result

- Dynamic string arrays:

  - StrArrayCreate(out: bool rc, out: int hStrArray)
    Create a new empty string array and result the handle for it

  - StrArrayDestroy(in: int hStrArray)
    Release the string array with the transferred handle. The string array no longer exists after the function call.

  - StrArrayWrite(in: int hStrArray, in: int index, in: string str)
    Enter the transferred string in the string array with the transferred handle under the specified index. The array is dynamically changed in size to comply with the index.

  - StrArrayRead(in: int hStrArray, in: int index, out: string str)
    Read the string with the transferred index from the string array with the transferred handle. If you read non-existent elements an error message appears on the screen.
    Note: If you read in program loops the current size of the array must be established by the user before the read process using the function StrArrayGetElementCount(...).

  - StrArrayGetElementCount(in: int hStrArray, out: int ElementCount)
    Return the number of elements contained in the string array with the transferred handle

  - StrArrayDelete(in: int hStrArray)
    Delete the string array with the transferred handle. The string array will still exist after the function call but all the elements in it will have been deleted and the memory occupied by it freed.

- RK512 functions

- SPSEnd (out: int komstatus)
  End the programmable controller

- SPSInit (out: int komstatus, in: string device, in: string file, in: int count, in: int msgbox);
  Initialise the programmable controller

- SPSLeseVakWerte (out: int komstatus, out: int check, out: real vakuum1, out: real vakuum2);
  Read the vacuum values

- SPSLeseVonSPS (out: int komstatus, out: int spsstatus, out: int check, out: real vakuum1, out: real vakuum2);
  Read the programmable controller and the vacuum values and evaluate them

- SPSSendeAnSPS (out: int komstatus, in: int fehler, in : int sg);
  Send the error byte and the control module

- ApiJobFsLesenFAB (out: int rc, in: string sgvar, out: int edifehler, out: string jobstatus,out: int fehler, out: int saetze);
  Read the error memory with errors masked out

- ApiResultFsLesenFAB (out: int rc, out: int ausgeblendet, in: int satz);
  Specifies for each error whether it has been masked out

- WINELDI function(s)

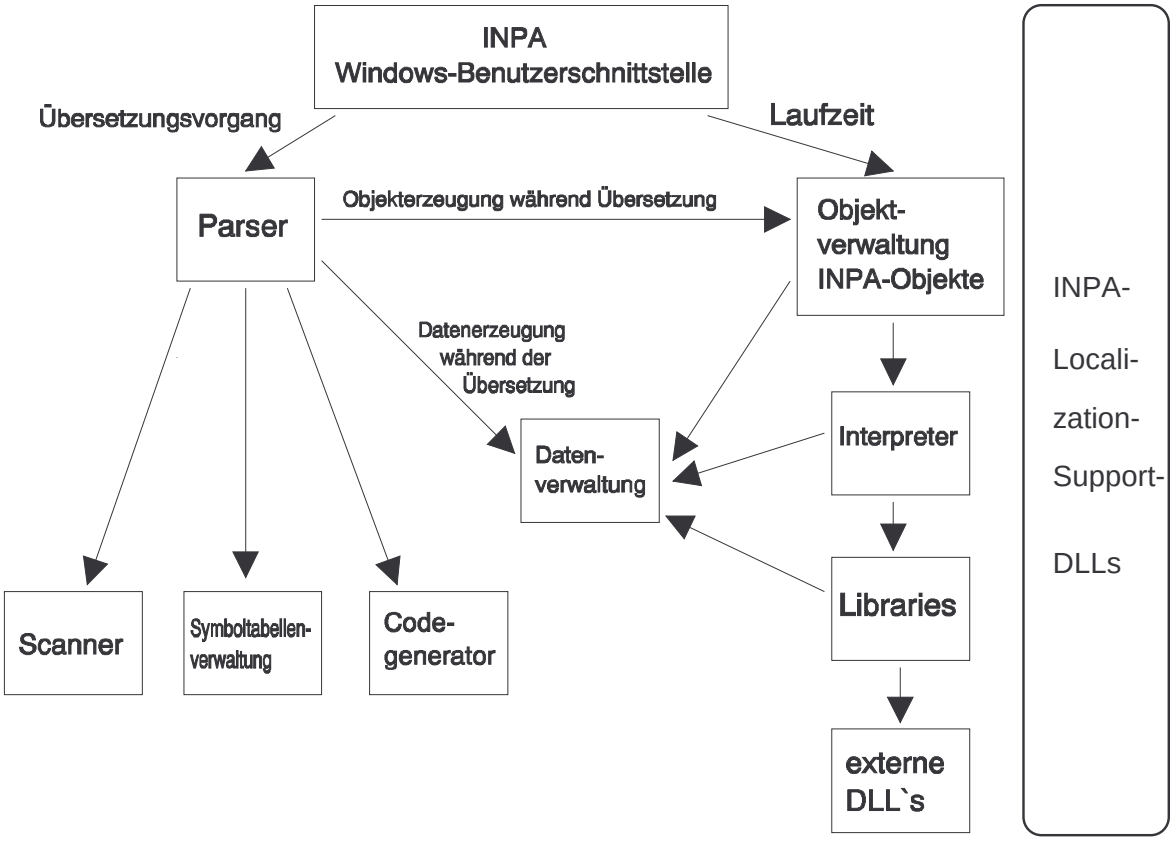
  - ELDIOpenStartDialog (in: string CommandParameter, out: int ResultCode)
    Start the ELDI/3 input handler dialogue.

    WARNING
    Before calling the input handler, all the job data in DTM are deleted; particularly when running in WINELDI this may lead to incorrect tests and test aborts.

6 Software structure

The software for the interpreter for test procedures is implemented in several layers. The Windows user interface acts as the top layer. It supplies the main window and some child windows for INPA and distributes incoming Windows messages (mouse events, keyboard events, etc.) to the lower software layers. In addition this layer controls the system startup, the global procedure control during the run time and the proper termination of the program.
Test procedure description files are processed during the run time using the following pattern:

- (1) Start the system by compiling and interpreting a startup script; the test procedure description files that are available for selection are configured in the startup script
- (2) Select a test procedure description file to be interpreted
- (3) Compile and interpret the selected file
- (4) End the test procedure and return to the startup script
- (5) with (2) start a new test procedure or quit the system



INPA Windows interface

Compilation process — Run time
Parser — Object generation during compilation — Object administration INPA Objects
Data generation during compilation
Data administration — Interpreter
Scanner — Symbol table administration — Code generator — Libraries
External DLL's

Figure 4 Software structure of INPA

The structure of INPA can be split roughly into two parts:

1. The compiler that compiles the INPA file into an operable code and compiles the data and objects required for the run time
2. The run time system that interprets the generated code and access the generated data and objects

The object and data administration system, used to manage the generated code, the data and the objects, acts as an interface between the two parts.
The jobs for the various software parts are described in more detail in the following.

6.1 Parser

The parser performs the syntactic and semantic language analysis of the INPA file to be compiled on the basis of the INPA language definition. The lexicon analysis of the scrip file with tokenisation and masking out of the comments is completed with the aid of the scanner. During the compilation process the parser generates the appropriate symbols with the aid of the symbol table administration system. This supplies functions to the parser for generating, deleting, searching, etc. global and local symbols and completes the actual symbol data maintenance (symbols including symbol-specific information). The symbols are no longer required after the completion of the compilation process and are therefore deleted.

INPA does not generate an exclusively linear program code but works on the basis of the object. The following INPA object types exist:

- SCREEN
- MENU
- STATE MACHINE
- LOG TABLE

The data structure of the objects is based on the problem and is covered by the form of the SCREEN, MENU, STATE MACHINE and LOG TABLE constructs described in the language definition. In addition to pure data, the objects contain administration functions and methods, in other words functions that are based on the special requirements of the objects (for example enlarged screen display and scrolling for SCREEN objects) INPA objects are implemented in the form of C++ classes.

During the compilation process the objects are dynamically generated by the parser using the object administration system (for example a MENU object is generated for each user MENU). The object-related code of the user (for example actions for each function key) are filed in the objects (for example a code list for each assigned function key) in the form of linear program code (code list). Widespread use of the principle of "information hiding" is made (for example the object administration system does not know the structure of the code lists).
The code for the user functions is saved in one code list, which is managed direct by the object administration system.

The code for the code lists filed in the objects is generated dynamically by the code generator during the compilation process. the code generator suppliers the parser with functions to general new empty code lists for this purpose as well as for the generation of code and for access to the code lists.
The generated code is code for a simple stack machine.

The data required during the run time are generated by the parser during the compilation process with the aid of the data administration system

6.2 Data administration

The data administration system completes the data maintenance of all global, constant and local data. The local data are administrated separately for each object type so that it is possible to switch between object types in the process during the run time (for example pressing a key whilst processing a SCREEN).
The data administration system supplies some external interface functions for data access:

- Functions to generate and delete global, constant and local data
- Elementary data operations during the run time (PUSH, POP, ...)
- Access functions for function parameters for library functions

6.3 Object administration

The object administration system completes the data maintenance of the dynamically generated INPA objects (see above) and the process control and coordination of the object actions during the run time. It supplies external functions with the following tasks for this purpose:

1. Generation and deletion of INPA objects
2. Setting object parameters (initialisation)
3. Activation/deactivation of objects
4. Activation/deactivation of code lists for the objects
5. Scheduling the object run during the run time
6. Administration of an object stack for the nested activation of sub-objects (used by state machines)

The functions for generating objects (1.) and initialisation (2.) are used during the compilation phase, the other functions during the run time.

The object management system's scheduler is triggered continuously during the run time and takes care of the coordination of the active INPA objects and the interpretation of their code lists. It has the following tasks:

- Round robin scheduling of SCREEN and STATE MACHINE objects
- Priority treatment of function key actions
- Transfer of code lists for objects to the interpreter
- Activation of code list interpretation

6.4 Interpreter

The code lists used by INPA contain program code for a simple stack machine with facilities for data manipulation (PUSH, POP, ALU operations, etc.) and program branching (JUMP, JUMPNZ, CALL, etc.). the interpreter implements this machine and is responsible for the sequential interpretation of the transferred code lists. During the execution of the code that required interpretation it accesses data administration functions for stack manipulations (PUSH, POP, etc.). Program branching within the same code list (JUMP) and ALU functions are processed direct by the interpreter.

For function calls a distinction is made between user and library functions: user function consist of a single code list; the call of a user function is implemented by a change of code list. Calls for library functions are made using pointers.

6.5 Library functions and external DLL's

the library functions are an integral part of INPA. They are hard-coded C++ functions that can be called from INPA script files with possible input and output parameters.
The library functions can be split into the following groups:

- Normal library functions (for example text output)
- System-related library functions, some of which influence the process (for example timers)
- Interface to external DLL's (for example EDIABAS). These links either provide a 1:1 function link or contain additional mechanisms (for example error processing and messages) to make the work of the user easier

6.6 INPA localisation support-DLL's

The support for multi-lingual operation is provided by string tables, which are integrated in one INPA localisation support DLL for each language. The language is selected by an entry in the configuration file (see above). After the program start this entry is evaluated and the corresponding DLL is loaded dynamically. All the texts that depend on the language are loaded from the DLL using access functions and buffered temporarily.
INPA V3.4.x supports the German and English languages.

6.7 Development tools used

- MSVC V1.50 (C++)
- MFC classes 2.0 from MSVC
- MKS LEX/YACC with C++ adjustment
- Windows SDK

7 Appendix A Language definition

```
program:
        { pragma }
        | { include }
        | { ext_func_declaration }
        | { ext_libcall_decl }
        | { g-data-definition }
        | { definition }

pragma:
        "#pragma"  identifier

include:
        "#include" string_constant

ext_func_declaration:
        "extern" func_dec ";"

ext_libcall_decl:
        "import" "pascal" ext_libcall_heading ";"
        "import" "\"C\""    ext_libcall_heading ";"
        "import" ext_libcall_heading ";"

ext_libcall_heading:
        ext_lib_spec func_dec

ext_lib_spec:
        "lib" string-constant

definition:
        {function-definition}
        | { menu-definition }
        | { screen-definition }
        | { logictable-definition }
        | { state machine-definition }

g-data-definition:
        type-specifier  init-dec-list  ";"

function-definition:
        func-dec
        func-block

menu-definition:
        menu-dec
        menu-block

screen-definition:
        screen-dec
        screen-block

logictable-definition:
```

```
        logictable-dec
        logictable-block

state machine-definition:
        state machine-dec
        state machine-block

type-specifier:
        "byte"
        | "int"
        | "long"
        | "real"
        | "bool"
        | "string"

object_type_specifier:
        "MENU"
        | "SCREEN"
        | "STATE MACHINE"
        | "STATE"

param-dir-specifier:
        "in"
        | "out"
        | "inout"
        | "returns"
        | "structure"

func-dec:
        identifier  "(" [func-param-list] ")"

func-param-list:
        func-param-equal-type { ","  func-param-equal-type }

func-param-equal-type:
        param-dir-specifier ":"  type-specifier  identifier
        |  param-dir-specifier ":"  object_type-specifier  identifier

func-block:
        "{"
        { l-data-definition}
        { statement }
        "}"

l-data-definition:
        type-specifier  init-dec-list  ";"

menu-dec:
        "MENU"  identifier  "(" ")"

menu-block:
        "{"
        { l-data-definition }
```

```
        "INIT"  "{"  {statement}  "}"
        { menu-item-block }
        "}"

menu-item-block:
        "ITEM"  "("  decimal-constant  ","  string-constant  ")"  "{"  {statement}   "}"

screen-dec:
        "SCREEN" identifier "(" ")"

screen-block:
        "{"
        { l-data-definition }
        { statement }
        { screen-line-block }
        "}"

screen-line-block:
        "LINE"  "("  string-constant , string-constant ")"
        "{"  {statement}
              [ control-line-block ]
        "}"

control-line-block:
        "CONTROL"  "{"  {statement}   "}"

logictable-dec:
        "LOGTABLE"  identifier  "("  logictable-out-par-list  ","  logictable-in-par-list ")"

logictable-out-par-list:
        "out"  ":"  "bool"  identifier  { " "  identifier }

logictable-in-par-list:
        "in"  ":"  "bool"  identifier  { " "  identifier }

logictable-block:
        "{"
        { logictable-line }
        [ logictable-other ]
        "}"

logictable-line:
        binary_constant ":" binary_mask ";"

logictable-other:
        binary_constant ":"  "OTHER"  ";"

state machine-dec:
        "STATE MACHINE" identifier "(" ")"

state machine-block:
        "{"
        { l-data-definition }
```

```
        "INIT"  "{"             { statement }   "}"
        { state machine-state-block }
        "}"

state machine-state-block:
        identifier "{"   {statement}  "}"

initializer:
        binary

init_dec_list:
        init_dec_list_line { "," init_dec_list_line }

init-dec-list_line:
        identifier
        | identifier "=" initializer

statement:
        binary  ";"
        | assignment ";"
        | func_call ";"
        | ";"
        | if-construct
        | while-construct

func_call:
        identifier "(" [argument_list] ")"

assignment:
        identifier "=" binary

binary:
        identifier
        | constant
        | "(" binary ")"
        |  "!" binary
        |  "-" binary
        |  binary "+" binary
        |  binary "-" binary
        |  binary "*" binary
        |  binary "/" binary
        |  binary ">" binary
        |  binary "<" binary
        |  binary ">=" binary
        |  binary "<=" binary
        |  binary "==" binary
        |  binary "!=" binary
        |  binary "||" binary
        |  binary "&&" binary
        |  binary "^^" binary
        |  binary "|" binary
        |  binary "&" binary
        |  binary "^" binary
```

```
argument-list:
        binary  { ","  binary }

statement_statements:
        statement
        | "{"  { statement } "}"

while-construct:
        "while" "("  binary  ")" statement_statements

if-construct:
        "if"  "("  binary  ")"  statement_statements  [ "else"  statement_statements ]

constant:
        int-constant
        | real-constant
        | bool-constant
        | string-constant

int-constant:
        decimal-constant
        | hexadecimal-constant

real-constant:
        ( (decimal-constant  [ "." decimal-constant] ) | ( "." decimal-constant) )    [exponent-constant]

exponent-constant:
        ("e" | "E")  ["+" | "-"]  decimal-constant

bool-constant:
        "FALSE"
        | "TRUE"

string-constant:
        """ {char-lit} """

decimal-constant:
        decimal-digit  { decimal digit}

binary_prefix:
        "0y"
        | "0Y"

binary_constant:
        binary_prefix  binary         _digit  { binary_digit }

binary_mask:
        binary_prefix  binary_mask_digit  { binary_mask_digit }

binary_mask_digit:
        binary_digit
        | "x"
```

```
        | "X"

binary-digit:
        "0"
        | "1"

decimal-digit:
        "0"
        | "1"
          ...
        | "9"

hexadecimal_prefix:
        "0x"
        | "0X"

hexadecimal-constant:
        hexadecimal_prefix  hexadecimal-digit  { hexadecimal digit}

hexadecimal-digit:
        "0"
        | "1"
        . . .
        | "9"
        | "a"
        . . .
        | "f"
        | "A"
        . . .
        | "F"

char-lit:
        printing-char
        | "\b"
        | "\n"
        | "\r"
        | "\t"
        | "\f"
        | "\\"
        | "\'"
        | "\""

identifier:
        (letter | underscore) { letter | decimal-digit | underscore }

letter:
        upper-case | lower-case

upper-case:
        "A"
        | "B"
        ...
        | "Z"
```

```
lower-case:
        "a"
        | "b"
        ...
        | "z"

underscore:
        "_"
         –

printing-char:      (ASCII-CharSet)
        " "
        | "!"
        ...
        | "~"

comment:
        "/*"   { printing-char }  "*/"
        | "//" { printing-char } "\n"
```

8 Appendix B Example of a test procedure description

```
// ****************************************************
// *** Includes
// ****************************************************

#include "inpa.h"

// ****************************************************
// *** Globals
// ****************************************************

string t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t98,t99;
real real0,real1,real2,real3,real4,real5,real6,real7,real8,real9,real10;
bool bool0,bool1,bool2,bool3,bool4,bool5,bool6,bool7,bool8,bool9;
bool bool10,bool11,bool12,bool13,bool14,bool15,bool16,bool17,bool18,bool19;
bool bool20,bool21,bool22,bool23,bool24,bool25,bool26,bool27,bool28,bool29;
bool bool30,bool31,bool32,bool33,bool34,bool35,bool36,bool37,bool38,bool39;

// ****************************************************
// *** Rework description file for SPM redesign
// ****************************************************
// *** Basic level
// ****************************************************

 SCREEN s_main ()
 {
   userboxclose(0);
   text( 1, 0, "Identification data mirror memory redesign ");
   text( 3, 0, "");
   INPAapiJob("sm_rd","IDENT","","");
   INPAapiCheckJobStatus("OKAY");
   LINE ( "BMW_partnumber", "")
   {
     text( 0, 1, "BMW-part number: ");
     INPAapiResultText(t0,"ID_BMW_NR",1,"");
     textout( t0, 0, 35);
     text( 2, 0, "");
   }

   LINE ( "Date_CW", "")
   {
     text( 0, 1, "Production date week: ");
     INPAapiResultText(t1,"ID_DATUM_KW",1,"");
     textout( t1, 0, 35);
     text( 2, 0, "");
   }
```

```
    LINE ( "Date_YEAR", "")
    {
      text( 0, 1, "Production date year: ");
      INPAapiResultText(t3,"ID_DATUM_JAHR",1,"");
      textout( t3, 0, 35);
      text( 2, 0, "");
    }
}


 MENU m_main ()
{
    INIT {
      INPAapiInit();
      setmenutitle( "Default menu");
    }
    ITEM( 1 , "Info")  {
      infobox("Information","Rework program for SPM redesign, Version 1.1. If you have any questions
please contact …");
    }
    ITEM( 2 , "Ident")  {
      setscreen( s_ident ,TRUE);
      setmenu( m_ident );
    }
    ITEM( 4 , "Error")  {
      setscreen( s_error,TRUE);
      setmenu( m_error );
    }
    ITEM( 5 , "Status")  {
      setscreen( s_status ,TRUE);
      setmenu( m_status );
    }
    ITEM( 10 , "End")  {
      INPAapiJob("sm_rd","diagnostic_end","","");
      INPAapiCheckJobStatus("OKAY");
      exit();
      }
}
```

```
// ******************************************************
// *** IDENT – control module identification
// ******************************************************
 SCREEN s_ident ()
 {
   text( 1, 0, "Identification");
   text( 3, 0, "");
   INPAapiJob("sm_rd","ident","","");
   INPAapiCheckJobStatus("OKAY");
   LINE ( "BMW_part number", "")
   {
     text( 0, 1, "BMW part number: ");
     INPAapiResultText(t0,"ID_BMW_NR",1,"");
     textout( t0, 0, 35);
     text( 1, 0, "");
   }

   LINE ( "BMW_hardware number", "")
   {
     text( 0, 1, "BMW hardware number: ");
     INPAapiResultText(t8,"ID_HW_NR",1,"");
     textout( t8, 0, 35);
     text( 1, 0, "");
   }

   LINE ( "software number", "")
   {
     text( 0, 1, "software number: ");
     INPAapiResultText(t1,"ID_SW_NR",1,"");
     textout( t1, 0, 35);
     text( 1, 0,"");
   }

   LINE ( "Supplier", "")
   {
     text( 0, 1, "Supplier number: ");
     INPAapiResultText(t2,"ID_LIEF_NR",1,"");
     textout( t2, 0, 35);
     text( 1, 0,"" );
   }

   LINE ( "Coding_Index", "")
   {
     text( 0, 1, "Coding Index: ");
     INPAapiResultText(t3,"ID_COD_INDEX",1,"");
     textout( t3, 0, 35);
     text( 1, 0,"");
   }
```

```
 LINE ( "Diagnostic Index", "")
   {
      text( 0, 1, "Diagnostic Index: ");
      INPAapiResultText(t4,"ID_DIAG_INDEX",1,"");
      textout( t4, 0, 35);
      text( 1, 0, "");
   }

   LINE ( "Bus_Index", "")
   {
      text( 0, 1, "Bus Index: ");
      INPAapiResultText(t5,"ID_BUS_INDEX",1,"");
      textout( t5, 0, 35);
      text( 1, 0, "");
   }

   LINE ( "Date_CW", "")
   {
      text( 0, 1, "Production date week: ");
      INPAapiResultText(t6,"ID_DATUM_KW",1,"");
      textout( t6, 0, 35);
      text( 1, 0, "");
   }

   LINE ( "Date_YEAR", "")
   {
      text( 0, 1, "Production date year: ");
      INPAapiResultText(t7,"ID_DATUM_JAHR",1,"");
      textout( t7, 0, 35);
   }
}

 MENU m_ident ()
{
   INIT {
      setmenutitle( " /Ident");
   }
ITEM( 10 , "Back")  {
      setscreen( s_main ,TRUE);
      setmenu( m_main );
   }
}
```



```
// ******************************************************
// *** Read error memory
// ******************************************************
 MENU m_error ()
 {
   INIT {
     userboxopen(0,10,20,6,50,"Read error memory", "Reading error memory !          ... Please wait ");
     INPAapiFsLesen("sm_rd","na_fs.tmp");
     userboxclose(0);
     viewopen("na_fs.tmp","Read error memory");
     setmenutitle( " /error/read error memory");
   }
   ITEM( 10 , "back")  {
     viewclose();
     setscreen( s_error ,TRUE);
     setmenu( m_error );
   }
 }

 SCREEN s_error ()
 {
   LINE ( "", "")
   {
   }
 }

// ******************************************************
// *** Read status
// ******************************************************
 SCREEN s_status ()
 {
   LINE ( "Text message", "")
   {
     text( 5, 10, "Display the selected status values");
     text( 8, 10, "Key <F4>: Analogue status (potentiometer values)");
   }
 }
```

```
MENU m_status ()
 {
   INIT {
     setmenutitle( " /Status");
   }
   ITEM( 4 , "Analogue")  {
     setscreen( s_status_analog ,TRUE);
     setmenu( m_status_analog );
   }
   ITEM( 10 , "back")  {
     blankscreen();
     setscreen( s_main ,TRUE);
     setmenu( m_main );
   }
 }

// ************************************
// Analogue status values
// ************************************

 SCREEN s_status_analog ()
 {
   INPAapiInit();
   text( 1, 0, "Rear analgoue status ");
   text( 2, 0, "Output potentiometer voltages ");
   INPAapiJob("sm_rd","read_status_analogue ","","");
   INPAapiCheckJobStatus("OKAY");
   LINE ( "Headrest", "")
   {
     text( 0, 1, "Headrest [Volt]: ");
     INPAapiResultAnalog(real0,"STAT_HEADREST_VALUE",1);
     analogout( real0, 1, 30, 0.0,5.0,0.0,5.0, "");
   }

   LINE ( "Seat height", "")
   {
     text( 0, 1, "Seat height [Volt]: ");
     INPAapiResultAnalog(real1,"STAT_SEAT HEIGHT_VALUE",1);
     analogout( real1, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Backrest", "")
   {
     text( 0, 1, "Backrest [Volt]: ");
     INPAapiResultAnalog(real2,"STAT_BACKREST_VALUE",1);
     analogout( real2, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }
```

```
LINE ( "Thigh support", "")
   {
     text( 0, 1, "Thigh support [Volt]: ");
     INPAapiResultAnalog(real3,"STAT_THIGH_VALUE",1);
     analogout( real3, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Seat angle", "")
   {
     text( 0, 1, "Seat angle[Volt]: ");
     INPAapiResultAnalog(real4,"STAT_SEAT ANGLE_VALUE",1);
     analogout( real4, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Seat slide", "")
   {
     text( 0, 1, "Seat slide [Volt]: ");
     INPAapiResultAnalog(real5,"STAT_SEAT SLIDE_VALUE",1);
     analogout( real5, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Passenger side mirror_horizontal", "")
   {
     text( 0, 1, " Passenger side mirror horizontal [Volt]: ");
     INPAapiResultAnalog(real6,"STAT_PASS_MIRROR_HOR_VALUE",1);
     analogout( real6, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Passenger side mirror_vertical", "")
   {
     text( 0, 1, " Passenger side mirror vertical [Volt]: ");
     INPAapiResultAnalog(real7,"STAT_APSS_MIRROR_VER_VALUE",1);
     analogout( real7, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Driver side mirror_horizontal", "")
   {
     text( 0, 1, " Driver side mirror horizontal [Volt]: ");
     INPAapiResultAnalog(real8,"STAT_DRIV_MIRROR_HOR_VALUE",1);
     analogout( real8, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }

   LINE ( "Driver side mirror_vertical", "")
   {
     text( 0, 1, " Driver side mirror vertical [Volt]: ");
     INPAapiResultAnalog(real9,"STAT_DRIV_MIRROR_VER_VALUE",1);
     analogout( real9, 1, 30, 0.0, 5.0, 0.0, 5.0, "");
   }
```



```
LINE ( "Battery voltage", "")
   {
      text( 0, 1, " Battery voltage [Volt]: ");
      INPAapiResultAnalog(real10,"STAT_U_BATT_VALUE",1);
      analogout( real10, 1, 30, 10.0 ,18.0,11.0,14.5, "");
   }
}


 MENU m_status_analog ()
{
   INIT {
      setmenutitle( " /Status/Analogue");
   }
   ITEM( 2 , "Start")  {
      start();
   }
   ITEM( 3 , "Stop")  {
      stop();
   }
   ITEM( 4 , "Select")  {
      select(TRUE);
   }
   ITEM( 5 , "Total")  {
      deselect();
   }
   ITEM( 10 , "Back")  {
      blankscreen();
      setscreen( s_status ,TRUE);
      setmenu( m_status );
   }

}


inpainit()
{
   INPAapiInit();
   settitle( " SPM - redesign   ");
   setmenu(m_main);
   setscreen(s_main,TRUE);
}


inpaexit()
{
   INPAapiEnd();
}
```