EDIABAS

Electronic Diagnostic Basic System

API INTERFACE DESCRIPTION

VERSION 6d

CONTENTS

1. Update history

Version 3.0	First release
Version 4.1	Revised for EDIABAS V4.1.0
Version 5	Inclusion of new API function apiJobExt and apiJobInfo
Version 5a	Description of the new virtual job _JOBCOMMENTS
Version 6	New API function apiInitExt, additional results of the virtual job _VERSIONINFO and parallel processing with Win32
Version 6d	Revised for EDIABAS V6.4.4

2. Introduction

2.1. About this manual

This manual describes the use of constants and functions that are available in the EDIABAS-API interface. Reference [3] gives a detailed description of these functions. Linking into a development environment is described in Reference [4]. You will find general information about EDIABAS and control unit description files in Reference [2].

2.2. Conventions

The following typographical conventions are used in this manual:

Example	Description
SAMPLE.C	Upper case characters are used for filenames, registers and operating system commands.
apiJob, APIREADY	Bold type is used for key words and operators of the BEST/2 and BEST/1 languages and for API functions. In syntax descriptions these words must be written as shown.
expression	Italics designate placeholders for values to be entered by the programmer; e.g., file names.
[option]	Words enclosed in square brackets may be optionally specified.
{ **result** | **argument** }	Curvy braces and vertical strokes characterize entries from which only one must be selected, except when in square brackets.
[constant...] job...	An ellipsis (three dots) which directly follows an expression indicates that several expressions of the same type can follow.
`hallo="Test";`	This syntax designates examples, user entries, program outputs and error messages.

`while() {`	A column or a row comprising three dots
`.`	indicates that a section of an example was
`.}`	intentionally omitted.
[1]	Reference to a document in References.

2.3. Special features, definitions, acronyms

The abbreviations used in this and all other EDIABAS documents are explained in the "GLOSSARY" section of the "EDIABAS User Manual". The term API is explained in greater detail in the "General" section.

3. General

The API application interface is the EDIABAS standard program interface across which an application program sends jobs to EDIABAS to operate control unit functions and across which the application program receives the results of the job execution back from EDIABAS.

The complete system is described in [2].

4. Operating the API interface

4.1. Initializing and abort of EDIABAS

A general initialization of EDIABAS must take place before the application program can call API functions. This is done with one of the following initialization functions:

apiInit ( )

apiInitExt (*device, device connection, device application, reserved* **)**

When API is initialized with **apiInit(),** EDIABAS performs a system-dependent default setting for the diagnostic interface (*device*), *device connection and device application*.

By calling **apiInitExt(),** it is possible to determine already during API initialization the diagnostic interface (*device*) to be used, the *device connection* and *device application*. The default setting for the diagnostic interface (*device*), *device connection* and *device application* can be accepted by entering a null string. The settings performed with **apiInitExt()** are only effective until API is aborted (**apiEnd()**).

If an application program accesses EDIABAS, this is called an API session.
Calling **apiInit()** or **apiInitExt()** also runs the application locking procedure that is described below in a separate section.

As a return value, the functions **apiInit()** and **apiInitExt()** provide information about the success of the EDIABAS initializing procedure:

APITRUE	Initializing successful (API accesses are possible)
APIFALSE	Initializing failed (API accesses are not possible)

See section "Trouble shooting" for guidance on error correction.

An API session can be terminated with the **apiEnd()** function, in which case the main memory and equipment are released.

After an API session has been terminated you must initialize with **apiInit()** or **apiInitExt()** before you can access API functions again.

Calling **apiInit()** or **apiInitExt()** without an **apiEnd()** will automatically initiate an **apiEnd()** in the application program (if a session is not yet terminated).

4.2. API jobs

After initializing, control units can be accessed by sending API jobs to the EDIABAS. These jobs always have the following structure:

apiJob (*CU group/variant, job, parameters, results***)**

apiJobData (*CU group/variant, job, parameter data, parameter length, results***)**

apiJobExt (*CU group/variant, job, parameter data for standard jobs, length of parameter for standard jobs, parameter data, parameter length, results, reserved***)**

The argument **CU group/variant** establishes the link with the addressed control unit, i.e. the relevant control unit description file (SGBD). The name of the SGBD is always given without its extension, e.g.:

DMETEST.PRG -> DMETEST

ALLEDME.GRP -> ALLEDME

The argument *job* corresponds to the job name defined in the ECU description file SGBD, whereby no differentiation is made between upper- and lowercase.

Arguments *parameter* and *Parameterdata/-length* are job parameters. Only a text string in "C" format can be passed as argument to function **apiJob** (e.g., "Par1;Par2;Par3" or ""). The individual parameters are separated in the argument with a semicolon, whereby upper- and lowercase are differentiated. **apiJobData** and **apiJobExt** can be passed as both a text string and binary data. Additional parameters for standard jobs can be passed to function **apiJobExt**. These parameters are passed to all standard jobs which are automatically executed after calling **apiJobExt** either before or after the specified job. If a text string is passed as a parameter in either **apiJobData** or **apiJobExt,** the length of the text string must be passed without the final '\0 character, otherwise the number of data bytes must be passed. A typical application for binary parameter data is the TRANSPARENT-MODE of EDIABAS. Further information can be found in reference [5].

The argument *results* determines which job events are to be produced. The individual results are separated with a semicolon in the argument, whereby no difference is made between upper- and lowercase. If all results are to be processed, specify a blank string for argument *results*. The selective creation of certain results must have been implemented when creating the ECU description file, otherwise all results are produces despite specification of special results.

4.2.1. Jobs with an unknown variant

You communicate with an unknown control unit variant by defining the group name as the first argument of the job:

apiJobxxx(SG group, ...)

The control unit group description file is loaded to identify the fitted variant.

After successful identification the appropriate control unit description file is loaded and the job is run.

When identifying the variant the name of the control unit group is used as a reference for the control unit group description file, the variant name of the control units being used to allocate corresponding control unit description files.

4.2.2. Jobs with a defined variant

Communication with a defined control unit variant is achieved by defining the variant name as the first argument in the job:

apiJobxxx(SG variant, ...)

The name of the control unit variant is then copied to the standard result VARIANTE.

4.2.3. Standard jobs

Standard jobs are jobs predefined by EDIABAS, their use is precisely specified and must be stored in control unit description files.

Standard jobs are called automatically by EDIABAS.

Function **apiJobExt** can be used to transfer parameters to standard jobs.

The result of standard jobs is stored in specified standard results that are polled automatically by EDIABAS.

4.2.3.1. Standard job INITIALISIERUNG

The standard job INITIALISIERUNG of a description file is called automatically:

- following an initialization of API (**apiInit()**) or **apiInitExt()**)
- after a change of the diagnostic interface with **apiSetConfig()**
- after a device change (**apiSwitchDevice()**)
- after the simulation mode has been switched on or off with **apiSetConfig()**

- following a previous error
- for a job with a defined variant (only after changing the control unit variant)
- for a job with an unknown variant
- after a call of the BEST/2 function **doNewInit** in an SGBD.

The standard job INITIALISIERUNG must exist in every group and variant description file.

Successful initializing must be displayed with the standard result DONE (result type **APIINTEGER** or **APIWORD**) in a result set:

- Result DONE <> 0 Initializing successful
- Result DONE = 0 Initializing failed
- Result DONE not present Initializing failed

The standard job INITIALISIERUNG must not return any results except DONE.

4.2.3.2. Standard job IDENTIFIKATION

The standard job IDENTIFIKATION is called to identify the fitted control unit variant:

- for a job with an unknown variant

The standard job IDENTIFIKATION must exist in every group description file.

An identified control unit variant must be displayed with the standard result VARIANTE (string of result type **APITEXT**) in a result set:

- Result VARIANTE = string (1 character min.) Identification successful, the result is the name of the control unit variant
- Result VARIANTE = blank string Identification failed
- Result VARIANTE not present Identification failed

The standard job IDENTIFIKATION must not return any results except VARIANTE.

4.2.3.3. Standard job ENDE

The optional standard job ENDE is called when the description file is changed:

- before the description file is changed
- before a job with an unknown variant

The standard job ENDE can exist in any group and variant description file.

The standard job ENDE must not return any results.

4.2.4. Virtual jobs

Virtual jobs are jobs predefined by EDIABAS; when called they provide information about a control unit description file.

Virtual jobs cannot be edited by the EDIABAS user.

Unlike standard jobs, virtual jobs are not in the control unit description files but are simulated by EDIABAS. However virtual jobs can be overwritten by self-defined jobs within the control unit description files.

The results of virtual jobs are stored in standard results with fixed names which are polled in the same way as the results of normal jobs (see section "Identifying results").

4.2.4.1. Virtual job _JOBS

The virtual job _JOBS identifies the names of all the jobs in the control unit description file (but not the names of virtual jobs).

Each job name is stored in a separate result set as a standard result JOBNAME, so the number of result sets is the number of jobs in the control unit description file.

4.2.4.2. Virtual job _JOBCOMMENTS

The virtual job _JOBCOMMENTS determines the comment to a particular job. The name of the job to be investigated is to be specified as job parameter when calling the virtual job. Determining the results is only possible with BEST2 ECU description files, which was not subsequently processed using the tool STRIP.

The following information are stored in the first result set for each result.

JOBCOMMENT*x*	Comment lines (*x* is a consecutive number in decimal notation beginning with 0)

4.2.4.3. Virtual job _VERSIONINFO

The virtual job _VERSIONINFO determines general information about the ECU description file and stores this in the first result set:

REVISION	Versions identification
AUTHOR	Author of last revision
FROM	Date of last revision

The following additional information is determined for BEST2 SGBDs which were not post-processed by the STRIP tool :

ECU	SGBD description
ORIGIN	author of the first version
LANGUAGE	language definition

USES*x*	basic files (x is a serial number in decimal format, beginning with 0)
ECUCOMMENT*x*	SGBD comment lines (x is a serial number in decimal format, beginning with 0)

4.2.4.4. Virtual job _ARGUMENTS

The virtual job _ARGUMENTS determines all defined arguments of a job; the name of the job to be investigated is to be specified as job parameter when calling the virtual job. Determining the arguments is only possible with BEST2 ECU description files, which as not subsequently processed using the tool STRIP.

The virtual job _ARGUMENTS cannot determined the arguments actually passed at runtime, but only the arguments defined in the job header.

The following information are stored in a separate result set for each argument:

ARG	Argument name
ARGTYPE	BEST data type of the argument
ARGCOMMENT*x*	Comment lines (*x* is a consecutive number in decimal notation beginning with 0)

4.2.4.5. Virtual job _RESULTS

The virtual job_RESULTS determines job results; the name of the job to be investigated is to be specified as job parameter when calling the virtual job. Determining the results is only possible with BEST2 ECU description files, which was not subsequently processed using the tool STRIP.

The virtual job _RESULTS cannot determine the results actually stored at runtime, but the results defined in the job header of the ECU description file.

The following information are stored in a separate result set for each result:

RESULT	Result name
RESULTTYPE	BEST data type of the result
RESULTCOMMENT*x*	Comment lines (*x* is a consecutive number in decimal notation beginning with 0)

4.2.4.6. Virtual job _TABLES

The virtual job _TABLES determines all table names of the ECU description files.

Note:
Only the names of local defined tables are determined

For each table a separate result set is stored with text result TABLE:

TABLE	name of the table

The system result SAETZE determines the numer of founded tables (0 ... n).

4.2.4.7. Virtual job _TABLE

The virtual job _TABLE determines the contents of a table. The name of the table to be investigated is to be specified as job parameter when calling the virtual job.

Note:
Only the contents of a local defined table are determined

In each result set the contents of one line of the table is stored. The result set consists of the text results ‚COLUMN0' to ‚COLUMNn'.

It is started with the first line of the table:

COLUMN0	content of first column
COLUMN1	content of second column
...	...
COLUMNn	content of n+1st column

The system result SAETZE determines the numer of all lines of a tables.

4.3. API Results

4.3.1. Data management

After executing a job with **apiJob(),** EDIABAS loads the corresponding ECU description file and processed the job passed. As a job is being processed, results occur which are to be passed to the application program.

The results are stored in a data structure by EDIABAS; this structure is designated APIRESULTFIELD.

A result field consists of a number of APIRESULTFIELDS which, on the other hand, contain a number of RESULTS. The number of result sets and the results depends on the job in the ECU description file.

System result 1	System result 2	**...**	System result m_0	Result set 0
Result 1	Result 2	**...**	Result m_1	Result set 1
Result 1	Result 2	**...**	Result m_2	Result set 2
...				
Result 1	Result 2	**...**	Result m_n	Result set n

Result set 0 of a job is automatically stored by EDIABAS. This, as SYSTEMRESULTS, designates result set contains general information about the processed job.

The results of the system result set are called SYSTEMRESULTS.

Generation of the system results via EDIABAS can be disabled via the EDIABAS configuration. System results **SAETZE, VARIANTE** and **OBJECT** are excluded from this and must always be stored.

All system results as well as the possible results values are described in the following summary:

System result	Comment	Value
SAETZE	Number of returned result sets	*APIWORD*
VARIANTE	Name of the ECU	*APITEXT[]*
OBJECT	Name of the ECU description file which contains the job	*APITEXT[]*
UBATTCURRENT	State of the supply voltage	**-1** = undefined **0** = disabled **1** = enabled
UBATTHISTORY	History of the supply voltage state	**-1** = undefined **0** = was disabled **1** = was enabled
IGNITIONCURRENT	State of the ignition	**-1** = undefined **0** = disabled **1** = enabled
IGNITIONHISTORY	History of the ignition state	**-1** = undefined **0** = was disabled **1** = was enabled
JOBSTATUS	Summarized status message of the job (= job return value)	*APITEXT[]*

4.3.2. Identifying results

The results of a job must be identified or read out of the result field with the following functions:

apiResultxxx(*target address,result,result set***)**
apiResultText(*target address,result,result set,format***)**
apiResultBinary(*target address,length address,result,result set***)**

The format of the result stored in the result field is the one defined and returned by the job in the SGBD.

Irrespective of this, an application program can get a result in any desired format provided conversion is possible.

The appropriate function must be selected for each result type desired by the application program:

Result function	Result format
apiResultChar()	**APICHAR**
apiResultByte()	**APIBYTE**
apiResultInt()	**APIINTEGER**
apiResultWord()	**APIWORD**
apiResultLong()	**APILONG**
apiResultDWord()	**APIDWORD**
apiResultReal()	**APIREAL**
apiResultText()	**APITEXT**
apiResultBinary()	**APIBINARY**

The *target address* is the address of the desired result variable in the application program where API is to store the result. The name of the result to be read must be entered as the argument *result* (not case sensitive).

The result function **apiResultBinary()** is an exception because it also requires the *length address* of an **APIWORD** result variable where API stores the number of copied bytes.

The *format* argument is another special case and is only available for the target result format **APITEXT**. The *format* argument converts the original result format into the target result format **APITEXT** with a definable conversion specification (see also section "Result types and formats"). If the *format* argument is a blank string on the other hand, the result is converted to the result format **APITEXT** with the default conversion.

Jobs can generate more than one result set (bundling of multiple results, e.g. error memories), which is why the result set must always be specified (with the *result set* argument).

The return value of the **apiResultxxx()** functions indicates the existence of the desired result:

APITRUE	Result present
APIFALSE	No result present

If a result set contains more than one result with the same name, only the last result can be identified.

A result may be absent for the following reasons:

- there was an error during job execution - no results present
- the result is present but cannot be converted to the desired format
- no result present

By calling the API function **apiResultFormat()** you can identify the format in which a result is stored in the result field:

The return **APIFALSE** indicates that the result is **not** present.

See section "Trouble shooting" for guidance on error correction.

4.3.2.1. Identifying the name of the control unit variant

If control unit identification is successful when a job is sent then EDIABAS stores the name of the control unit description in the system result set (result set 0) as the result **VARIANTE**.

As well as polling with **apiResultxxx()**, simplified polling using

apiResultVar(*target address***)**

is also possible. The *target address* is the address of a variable in the application program where API stores the content of the **VARIANTE** result.

The description of the return of **apiResultxxx()** is also valid for **apiResultVar()**.

4.3.2.2. Identifying the number of result sets

API automatically stores the number of result sets in the system result set (result set 0) as the result **SAETZE** .

As well as polling with **apiResultxxx()**, simplified polling using

apiResultSets(*target address***)**

is also possible. The *target address* is the address of a APIWORD variable in the application program where API stores the content of the **SAETZE** result.

The description of the return of **apiResultxxx()** is also valid for **apiResultSets()**.

4.3.2.3. Identifying the number of results in a result set

The number of results in a result set can be identified with the function

apiResultNumber(*target address,result set***)**

The *target address* is the address of an **APIWORD** variable in the application program where API stores the number of results of the result set *result set*.

The description of the return of **apiResultxxx()** is also valid for **apiResultNumber()**.

4.3.2.4. Identifying a result name using the result index

The name of a result can be identified with the function

apiResultName(*target address,result index,result set***)**

The *target address* is the address of a string variable in the application program where API stores the result name.

The *result index* is the sequential number of a result in a result set. Valid values are 1..no., where no. is the number of results in the set *result set* identified with **apiResultNumber()**.

The description of the return of **apiResultxxx()** is also valid for **apiResultName()**.

4.3.2.5. Identifying the format of a result

The format of the result returned by the control unit description file can be identified with the function

apiResultFormat(*target address,result,result set***)**

The *target address* is the address of an **APIRESULTFORMAT** variable in the application program where API stores the result format. The argument *result* is the result you are searching for. The result set must be specified in the argument *result set*.

The description of the return of **apiResultxxx()** is also valid for **apiResultFormat()**.

4.3.3. Result types and formats

4.3.3.1. Result types

The results of a job can exist in or be converted to the following formats:

APICHAR
APIBYTE
APIINTEGER
APIWORD
APILONG
APIDWORD
APIREAL
APITEXT
APIBINARY

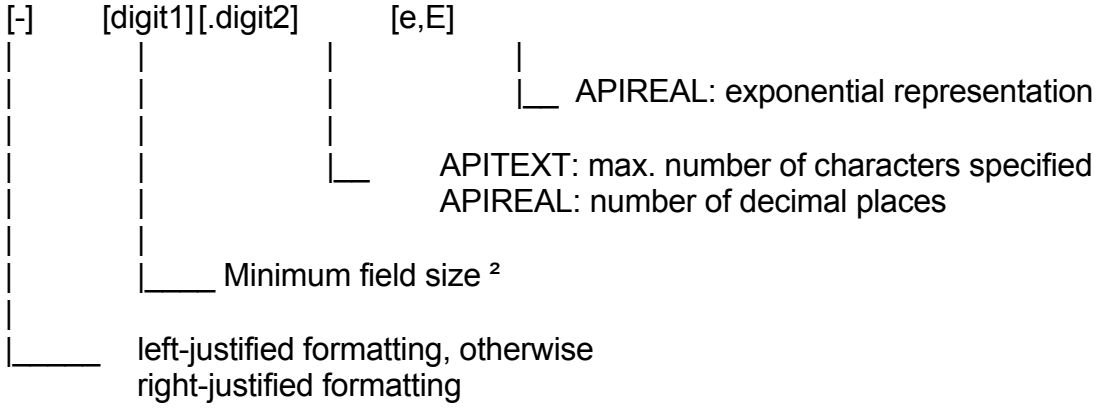
4.3.3.2. Result formats

The formatting instruction uses the **apiResultText(...,Format)** function and must be configured as follows:

[*Specifications*] *Conversion characters*

Conversion character	Conversion
C[HAR]	APICHAR --> APITEXT
B[YTE]	APIBYTE --> APITEXT
I[NTEGER]	APIINTEGER --> APITEXT
W[ORD]	APIWORD --> APITEXT
L[ONG]	APILONG --> APITEXT
D[WORD]	APIDWORD --> APITEXT
R[EAL]	APIREAL --> APITEXT

Specifications:



[2] Pad with blanks for short arguments
Expand to required number of digits for long arguments

Examples:

Formatting instruction "B":	Converts a result (if convertible) to the **APIBYTE** format, then further converting and return of a right-justified **APITEXT** string.
Formatting instruction "20T":	Converts a result (if convertible) to the **APITEXT** format, then further converting and return of a right-justified APITEXT string with a total of 20 digits.
Formatting instruction "-8.2ER":	Converts a result (if convertible) to the **APIREAL** format, then further converting and return of a left-justified APITEXT string with a total of 8 digits, with the result being stored in exponential representation with 2 decimal places.
Formatting instruction "":	Converts a result (if convertible) to the **APITEXT** format and returns the left-justified APITEXT string.

4.3.4. Application result fields

(Buffer) storage and delayed evaluation of results fields by the application program is possible using the routine described below.

CAUTION: This switch affects the result field only, the status of EDIABAS is not stored.

4.3.4.1. Creating an application result field

A duplicate of the current result field can be created within API with the function

apiResultsNew()

The return is a reference to the result field (application result field) dynamically created in the main memory. **NULL** indicates creation has failed.

4.3.4.2. Creating a reference to the application result field

By calling the function

apiResultsScope(*reference***)**

it is possible to switch to the application result field; a reference to the application result field is required as a function argument (known from **apiResultsNew()**).

The results in the application result field can then be accessed using known evaluation methods (**apiResultxxx()**).

The link to the application result field exists until new results are stored in API's own result field (initiated by calling **apiJobxxx()**).

4.3.4.3. Deleting an application result field

A memory location created with **apiResultsNew()** can be deleted with the function

apiResultsDelete(*reference***)**

for each application result field (reference as function argument).

4.4. Device switching

Communicating with the diagnostic interface is done by accessing the interface driver.

To access the diagnostic interface (*device*) you must specify the *device connection* and the *device application:*

Necessary access information for diagnostic interface = device connection + device application

The *device connection* is the interface to be controlled; the application to be addressed on the interface is referred to as the *device application*.

When initializing API with **apiInit()**, EDIABAS performs system-dependent presets for device connection and device application.

The preset uses the device driver supplied by Softing for the diagnostic bus interface.

To switch to a different device connection or device application, call the function

apiSwitchDevice(*deviceConnection, deviceApplication***)**

Both the *device connection* and the *device application* must be entered as strings.

Switching to a different device connection or device application is only effective until a further switch with **apiSwitchDevice()** or until API is terminated (**apiEnd()**).

The preset for device connection and device application can be determined by entering a blank string.

The function **apiSwitchDevice()** returns information about the successful switch of the device connection and the device application:

APITRUE	Switch successful
APIFALSE	Switch failed

See section "Trouble shooting" for guidance on error correction.

4.5. API Sequence control

Basically, the following sequence must be observed. It can be repeated as often as desired:

```
...
apiInitxxx()
...
other API functions
...
apiEnd()
...
```

After successful API initializing (**apiInitxxx()** with the return **APITRUE**) all other API function calls can be executed until API is terminated (**apiEnd()**).

The API functions for trouble shooting are an exception to this rule - they can also be called when API initializing fails (see section "Trouble shooting").

4.5.1. Job control

4.5.1.1. Automatic job execution

The following functions initiate the execution of a job that has been started but not yet finished:

- start a new job (**apiJobxxx**)
- device change (**apiSwitchDevice**)
- identify result (**apiResultxxx**)

The unfinished job is completed before the new action.

The new action is carried out even when there is an error in the automatically executed job.

An error in the automatically executed job is cleared before the new action is carried out.

4.5.1.2. Cyclical polling

In order to carry out time-critical actions of the application program while an API job is running (e.g. keyboard polling, display configuration etc.), the processing status of the job must be polled with

apiState()

before the result is polled with **apiResultxxx()**.

The return from **apiState()** is the job status of the (last) job started:

APIBUSY : job still running
APIREADY : job is done
APIBREAK : job was aborted
APIERROR : job had an error

An ongoing job can be terminated with the function

apiBreak()

This also clears any existing results and returns you to the application program.

Unlike the other job states, **APIBREAK** can only be initiated by the application program.

To do this, either the function **apiBreak()** must be called or the call-back function must be terminated by the **APITRUE** return.

There are no results with the **APIERROR** job state.

Job states **APIBREAK** and **APIERROR** are held until a job is started with **apiJobxxx()** or API is re-initialized with **apiInitxxx()**.

4.5.1.3. Call-Back option

Instead of the EDIABAS sequence control using **apiState()**, it is possible to use a call-back routine in the application program.

During an API job EDIABAS will continuously call a user-selectable function of the application program.

Activating a Call-Back routine

EDIABAS is asked for a Call-Back routine with

apiCallBack(*function reference***)**

Syntax of a Call-Back routine.

The return of the Call-Back routine is interpreted as an abort request to EDIABAS:

Return = **APITRUE**:	Abort order immediately
Return = **APIFALSE**:	Continue order

If EDIABAS sees the request for an immediate abort after executing the Call-Back it will terminate the current API job (condition is **APIBREAK**), delete any results and return to the application program.

Run-time request

Since a Call-Back routine is activated continuously during an API job execution, interrupting EDIABAS every time, Call-Back routines must observe a relatively short run-time.

Constraints

No API functions may be called during a Call-Back routine apart from the **apiErrrorCode()** and **apiErrorText()** functions.

4.5.1.4. Job abort

The following events can abort a job in progress:

- Error during execution (indirect abort)
- API is aborted with **apiEnd()** (indirect abort)
- Direct job abort with **apiBreak()**
- Direct job abort with return **APIFALSE** in the Call-Back routine

A direct job abort initiates an error which is why all result data are lost and cannot be accessed.

An API abort with **apiEnd()** initiates an **apiBreak()** and so aborts a job in progress.

4.5.2. Application Locking

EDIABAS can only communicate with one application program at any one time, and so multitasking operating systems need a procedure to lock EDIABAS onto an application program - this is called Application Locking.

4.5.2.1. Locking an application program to EDIABAS

If API is initialized (by **apiInit()**) within an application program, the system first checks for an existing EDIABAS lock.

An existing EDIABAS lock with an active application program is displayed by an **APIFALSE** return by the API function **apiInit()**, and the application program is rejected (initializing failed).

If there is no lock or if the application program of a lock no longer exists, then this is displayed by an **APITRUE** return by the API function **apiInit()**, the application program is locked onto EDIABAS (initialized OK) and EDIABAS is now locked off for other application programs.

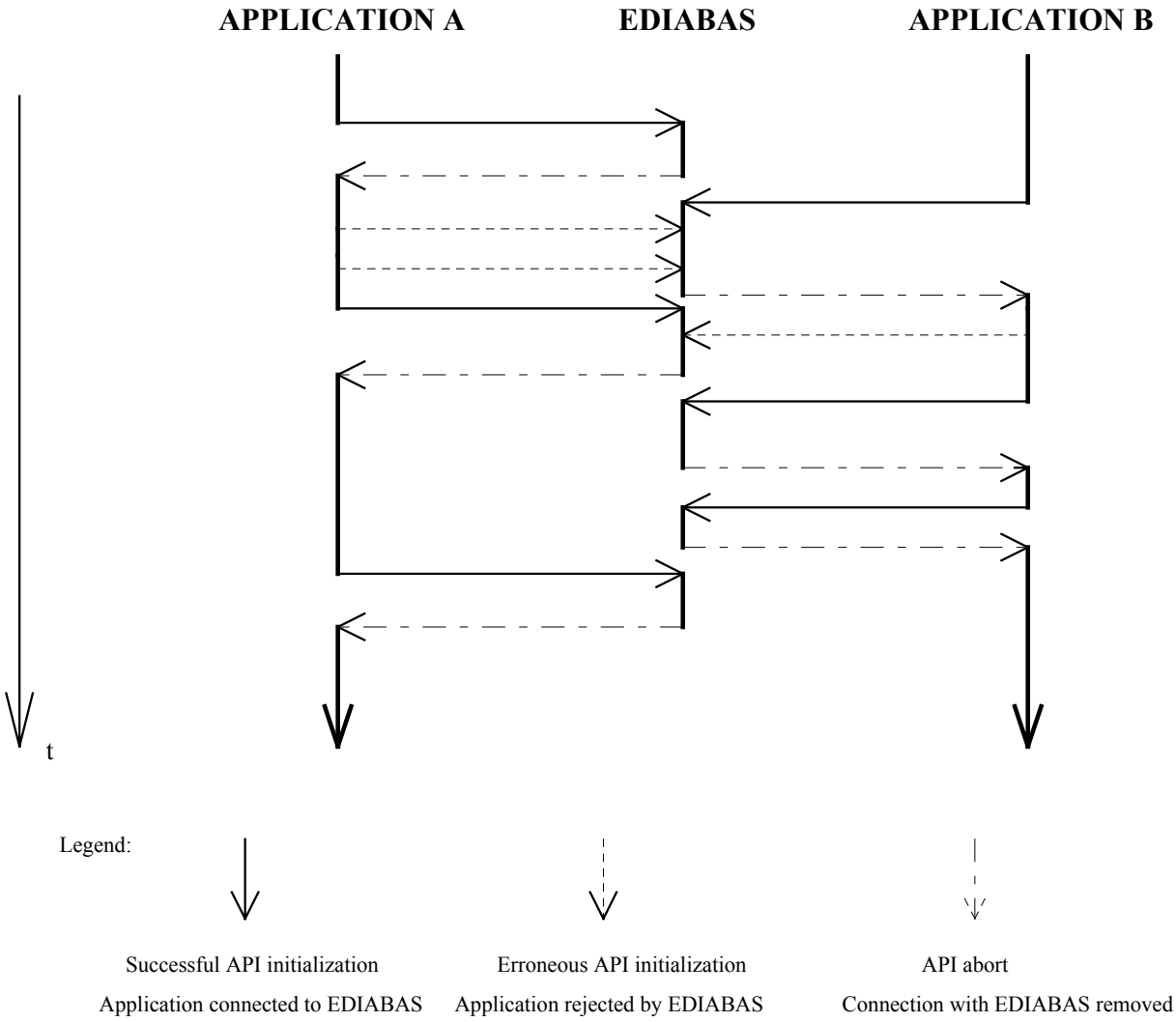
4.5.2.2. Canceling the lock with EDIABAS

The lock of an application program to EDIABAS can be canceled by calling **apiEnd()**, thereby making EDIABAS available for other processes.

4.5.2.3. Access by several application programs to EDIABAS

Quasi-parallel access to EDIABAS by a number of application programs is possible by short-time locking of EDIABAS. Ideally however there should only be one job for each lock with **apiInit()**/**apiEnd()**. When EDIABAS is locked, each application then waits for the lock to be canceled.

Example of alternating access by two application programs to EDIABAS:



4.5.3. Parallel Processing (only WIN32/CE and SCO-UNIX)

EDIABAS for WIN32/CE and SCO-UNIX permits simultaneous access to EDIABAS by several different application programs.

In contrast to the other platforms, EDIABAS for WIN32/CE is no one single software which runs all application programs consecutively. Instead, there is a separate EDIABAS for each application program, which is started and terminated automatically under WIN32/CE. Under SCO-UNIX a simultaneous access for up to 64 application programs is possible.

The interfaces available with EDIABAS may only be accessed by 1 application program at a time. Therefore, EDIABAS for WIN32/CE and SCO-UNIX requires an interface locking mechanism which ensures exclusive access of an application program to an interface (*device locking*).

Device locking takes place if the functions *apiInit()*, *apiInitExt()* or *apiSwitchDevice()* are called:

apiInit()	➔	default device "_"
apiInitExt()	➔	device according to function call, e.g. "A"
apiSwitchDevice()	➔	device according to function call, e.g. "A"

If several application programs try to access the same device, all except the first one will receive the error *IFH-0029: ACCESS DENIED*.

It must be taken into account that when *apiInit()* is called, exclusive access to the default device "_" takes place. If another application is already accessing the default device "_", the call of *apiInit()* will fail (*IFH-0029: ACCESS DENIED*).

Therefore, an application program can only access EDIABAS with apiInit() if all currently running application programs have selected an interface different from “ “ with apiInitExt() or apiSwitchDevice()!

4.5.4. Application Locking (not WIN32/CE and SCO-UNIX)

EDIABAS can only communicate with one application program at a time. With multitasking operating systems, therefore, a procedure for binding EDIABAS to one application program is necessary (application locking).

4.5.4.1. Binding an Application Program to EDIABAS

If API is initialized within an application program (with **apiInit()** or **apiInitExt()**), EDIABAS is first checked for an existing binding.

An existing binding of EDIABAS to an active application program is indicated by an **APIFALSE** return value of the API functions **apiInit()** or **apiInitExt()**, respectively; the application program is rejected (incorrect initialization).

If, on the other hand, there is no binding; or if there is a binding, but the corresponding application program exists no longer, this is indicated by an **APITRUE** return value of the API functions **apiInit()** or **apiInitExt()**, respectively. The application program is therefore bound to EDIABAS (successful initialization) and EDIABAS is now locked to other application programs.

4.5.4.2. Clearing a Binding to EDIABAS

The binding of an application program to EDIABAS can be cleared by calling **apiEnd();** EDIABAS will then again be accessible by other processes.

4.5.4.3. Access of more than one Application Program to EDIABAS (not WIN32/CE and SCO-UNIX)

Practically parallel access of more than one application program to EDIABAS is possible by locking EDIABAS only for a short time. The ideal case would be an execution of only one task per binding with **apiInit**xxx**()**/**apiEnd()**. With a locked EDIABAS, every application can then wait for the clearing of the binding.

4.6. Configuration

The EDIABAS configuration consists of several information pairs, each pair of which consists of the following:

Configuration element and configuration setting

Configuration element and configuration setting each represent string with predefined contents.

All EDIABAS configuration elements can be interrogated; changing the configuration setting, however, is not possible for all configuration elements. The configuration can only be modified up to the next API abort (**apiEnd()**).

A detailed description of all configuration elements and their configuration settings can be found in reference [2].

4.6.1. Determining the configuration

The setting of a configuration element can be determined by calling the function:

apiGetConfig (*configurationelement***,** *destinationaddr* **)**

*configurationelement*represents the string value of configuration element which is to be interrogated. *destinationaddr* represents the address of the string variable where API copies the corresponding configuration setting.

The return value indicates how the configuration was determined:

Return value = **APITRUE** :	Configuration successful
Return value = **APIFALSE** :	Configuration failed

Failed configuration does not cause an EDIABAS error to be issued.

4.6.2. Modify the configuration

The setting of a configuration element can be modified with the following function, provided this is permitted by EDIABAS:

apiSetConfig (*configurationelement*, *configurationsetting***)**

configurationelement represents the string value of the configuration element to be modified; *configurationsetting* represents the string value of the new setting.

The return value indicates the success of the modified configuration:

Return value = **APITRUE** :	Configuration modified
Return value = **APIFALSE** :	Illegal configuration modification

An erroneous configuration modification does not cause an EDIABAS error to be issued.

4.7. Trouble shooting

4.7.1. Identifying the cause of an error

The error status on which an error is based can be identified by calling the function

apiErrorCode()

The default error text of the error can also be identified using the function

apiErrorText()

Reference [1] gives a description of all possible errors.

4.7.2. Using an Error Handler

For error handling, the application program can define a function without arguments that the run-time system must activate whenever an error occurs (Error Handler):

apiErrorHandler (action)

In the Error Handler, the error will be typically analyzed by polling **apiErrorCode()** or **apiErrorText()**.

4.7.3. Procedure for error analysis when initializing and identifying results

As well as the return **APIFALSE**, for functions with **APIBOOL** return an API error can be identified in one of the following two ways (see also the section on "Trouble shooting"):

a) Calling the error code with the API function **apiErrorCode()**

Return = **EDIABAS_ERR_NONE**	:	No API error present
Other return	:	API error(code) present

b) Calling the error text with the API function **apiErrorText()**

Return = **NULL**	:	No API error present
Other return	:	API error(text) present

By calling the API function **apiResultFormat()** you can check whether the desired result format even exists:

The return **APIFALSE** indicates that the result *is not* available.

4.7.4. Clearing an error

Errors remain set until a job is started by **apiJobxxx()**, the device is changed with apiSwitchDevice(), or until API is re-initialized with **apiInit**xxx**()**.

The errors API-0005 and API-0014 are special cases; these will also be reset during the next result query with apiResultXxx().

5. Application example

```
/*************************************************************
***
***  apidemo.c
***
***  Copyright (c) 2001 Softing AG
***  All rights reserved
***
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include "api.h"


#define LINESIZE  33

void printResultField(FILE *);


/*****************************************
 *  Output a binary result
 *
 *
 *  Parameters:
 *    - fp   File Pointer of output file
 *    - yb   Address of APIBINARY buffer
 *    - yn   Number of bytes to be output
 *****************************************/
void printBinary(FILE *fp,APIBINARY *yb,APIWORD yn)
{
   /* line format:                          */
   /* line   0    25                        */
   /* XXX:  XX XX XX XX XX XX XX XX XXXXXXXX */

   static char line[LINESIZE+1];
   static char addr[4];
   unsigned n=0,z,hexz;

   while (n<yn) {
      for (z=0;z<LINESIZE;z++) line[z]=' ';
      line[LINESIZE]=addr[0]='\0';
      z=hexz=0;

      while (n<yn && z<8) {
         if (z==0)
            sprintf(addr,"%03X: ",n);
         else
            hexz++;
         sprintf(&line[hexz],"%02X",yb[n]);
         hexz+=2;
         line[hexz]=' ';
         line[LINESIZE-8+z]=isprint((int)yb[n]) ? yb[n]:'.';
         n++;
         z++;
      }
            /*123456789012345678901234 5*/
      fprintf(fp,"\n                          %s%s",addr,line);
   }
}
```

```
/*****************************************
 *  Output the result field
 *
 *  If the displayed File Pointer shows the value NULL
 *  all results are polled but not output
 *
 *  Parameters:
 *    - fp
 *      File Pointer of output file
 *
 *  Return is APITRUE if correctly output,
 *  otherwise APIFALSE
 *
 *****************************************/
void printResultField(FILE *fp)
{
   static APITEXT t[APIMAXTEXT];
   static APITEXT text[APIMAXNAME];
   static APIBINARY yb[APIMAXBINARY];
   APIWORD sets,set,results,result;
   APIWORD yn,w;
   APICHAR c;
   APIBYTE b;
   APIINTEGER n;
   APILONG l;
   APIDWORD d;
   APIREAL r;
   APIRESULTFORMAT format;

   if (fp!=NULL)
      fprintf(fp,"RESULTFIELD:\n\n");

   /* Identify the number of results */
   if (apiResultSets (&sets)) {

      /* For all result sets */
      for (set=0;(set <= sets) && (apiErrorCode()==EDIABAS_ERR_NONE);set++) {

         if (fp!=NULL)
            fprintf(fp,"SET %3u:",set);

         /* Number of results in current result set */
         if (apiResultNumber(&results,set)) {

            /* For all results */
            for(result=1;(result<=results) && (apiErrorCode()==EDIABAS_ERR_NONE);result++) {

               /* Identify result name */
               if (apiResultName(text,result,set)) {

                  if (fp!=NULL)
                     fprintf(fp,"  %sRESULT %3u: ",(result==1)?"":"        ",result);

                  /* Identify result format */
                  if (apiResultFormat(&format,text,set)) {

                     /*Download results by format and output */
                     switch (format) {

                        case APIFORMAT_CHAR:
                           if (apiResultChar(&c,text,set))
                              if (fp!=NULL)
                                 fprintf(fp,"[ CHAR    ] %s = %+d",text,c);
                        break;

                        case APIFORMAT_BYTE:
                           if (apiResultByte(&b,text,set))
                              if (fp!=NULL)
                                 fprintf(fp,"[ BYTE    ] %s = %u",text,b);
                        break;
```

```
                        case APIFORMAT_INTEGER:
                           if (apiResultInt(&n,text,set))
                              if (fp!=NULL)
                                 fprintf(fp,"[ INTEGER ] %s = %+d",text,n);
                           break;

                           case APIFORMAT_WORD:
                              if (apiResultWord(&w,text,set))
                                 if (fp!=NULL)
                                    fprintf(fp,"[ WORD    ] %s = %u",text,w);
                           break;

                           case APIFORMAT_LONG:
                              if (apiResultLong(&l,text,set))
                                 if (fp!=NULL)
                                    fprintf(fp,"[ LONG    ] %s = %ld",text,l);
                           break;

                           case APIFORMAT_DWORD:
                              if (apiResultDWord(&d,text,set))
                                 if (fp!=NULL)
                                    fprintf(fp,"[ DWORD   ] %s = %lu",text,d);
                           break;

                           case APIFORMAT_REAL:
                              if (apiResultReal(&r,text,set))
                                 if (fp!=NULL)
                                    fprintf(fp,"[ REAL    ] %s = %E",text,r);
                           break;

                           case APIFORMAT_TEXT:
                              if (apiResultText(t,text,set,""))
                                 if (fp!=NULL)
                                    fprintf(fp,"[ TEXT    ] %s = %c%s%c",text,'"',t,'"');
                           break;

                           case APIFORMAT_BINARY:
                              if (apiResultBinary(yb,&yn,text,set)) {
                                 if (fp!=NULL)
                                    fprintf(fp,"[ BINARY  ] %s = %u bytes,",text,yn);
                                 printBinary(fp,yb,yn);
                              }
                           break;
                        }

                        if (fp!=NULL)
                           fprintf(fp,"\n");
                     }
                  }
               }
            }
         }
      }
   }
}

/*****************************************
 *
 *  program start
 *
 *****************************************/
int main (void)
{
   static char ecu[APIMAXNAME];
   static char job[APIMAXNAME];
   static char para[APIMAXPARA];
   static char result[APIMAXRESULT];

   printf("********** APIDEMO V1.0 **********\n");
   printf("* Copyright (c) 1998 Softing GmbH *\n");
   printf("* All rights reserved             *\n");
   printf("***********************************\n\n");
```

```
   /* API-Job upload parameters */
   printf("control unit : "); gets(ecu);
   printf("Job          : "); gets(job);
   printf("Job-Parameter: "); gets(para);
   printf("Job-Results  : "); gets(result);

   printf("\nPress ENTER to start processing\n\n");
   getchar();

   if (!apiInit()) {
      printf("\nerror on apiInit(); %s\n",apiErrorText());
      goto end;
   }

   apiJob(ecu,job,para,result);

   while (apiState()==APIBUSY); /* processing */;

   if (apiState()==APIERROR) {
      printf("\nEDIABAS error %d:%s\n",apiErrorCode(),apiErrorText());
      goto end;
   }

   printResultField(stdout);

end:

   apiEnd();

   printf("\nPress ENTER to exit program\n");
   getchar();

   return 0;
}
```

A. List of references

[1] EDIABAS: Error Primer

[2] EDIABAS: User Manual

[3] EDIABAS: API Function Primer

[4] EDIABAS: API User Manual

[5] EDIABAS: Transparent Mode