ADLATUS®

SMART

SMART ADLATUS®

Application Handbook

V1.07

SMART Electronic Development GmbH
Rötestraße 17
D - 70197 Stuttgart
Tel.: ++49 (0)711 / 25521 – 0
Fax: ++49 (0)711 / 25521 – 10
http://www.smart-gmbh.de

Contents:

1 Introduction

1.1 General

To (re-)program software in the flash memory of a control unit, a component is needed within the control unit that can carry out this programming.

The SMART Electronic Development GmbH company offers, with SMART ADLATUS® (Latin for helper), such a tool based on “embedded software”. The ADLATUS® is stored in a reserved memory area of the control unit and can then be used to program the flash memory.

Typical areas of usage for the programming are:

- ❖ Programming during the development phase
- ❖ Programming of system tests during the production phase
- ❖ End of line programming during production
- ❖ Update programming in the field

To meet the requirements of the different areas of usage, the ADLATUS® software must be modified to meet individual needs. This is made possible by several configuration settings, and can also be carried out by the user.

This application handbook is meant to aid the user in setting the desired configuration and in reducing the possibility of incorrect operation. The possible settings are described for each function block.

2 Module overview

2.1 Introduction

The SMART ADLATUS® consists of multiple modules, which are in their turn grouped into logical components. All components possess standardized interfaces and can be combined to meet the desired configuration profile.

The software thereby directly utilises the hardware of the respective microcontroller. For this reason, corresponding interfaces to the hardware are implemented in ADLATUS®.

In addition, the ADLATUS® possesses interfaces for customer-specific components. It is thus possible to modify the software to have different projects on the same target hardware.

The following briefly describes all of the SMART ADLATUS® building block modules. For customer-specific configuration reasons, there is a possibility that some of the modules described in the following are not implemented in the current ADLATUS®.

2.1.1 Function libraries

One part of ADLATUS® is divided into four function libraries. The functions compiled there normally require no modification to project specifications and are therefore not integrated into the project as a C-source file.

Function libraries generally have the following naming convention:

func_comp_contr_step_prj.lib

func	Function contents, e.g. CHCK for validation functions
comp	Version including compiler
contr	Microcontroller family/derivative
step	Version/Step of microcontroller where applicable
prj	SMART internal project number/name

All validation functions are pooled in the library “...CHCK...lib". More detail can be found in Chapter 6.3, “System test after reset”.

The library "..CORE...lib" generally contains all modules that are project-independent. These form the core of ADLATUS®.

Hardware-dependent modules required for the basic functionality of the ADLATUS are pooled in "..HAL...lib".

The library "_..xxx...lib" contains the programming-specific modules such as the KWP2000 commands and their processing.

2.1.2 Directory structure

The components of the ADLATUS® are divided into directories of the structure illustrated below. The representation only shows directories contained in each project as a minimum quantity. Thus, for example, projects accompanied by the function libraries as sources have their own directory tree for each library (01_prj, 02_src....).

```
01_adlatus
    ├── 01_prj ──  Development environment and project settings
    ├── 02_src ──  Sources for configuration
    └── 0x....  ──  and/or other directories (project-specific)
03_fsw
    └── ...... ──  Example application (hex file)
05_header
    ├── ext ─────  Header files
    └── int ─────
06_hallib
    └── 0x.....──  Hardware functions (Library)
07_chcklib
    └── 0x.....──  Validation functions (Library)
08_corelib
    └── 0x.....──  Core functions (Library)
09_xxxlib
    └── 0x.....──  Programming process (Library)
10_his
    └── 0x.....──  Programming/Erase functions (hex file)
```

2.2 Files in the 01_Adlatus directory

Functionalities available as source code in the project directory (01_Adlatus\02_src):

2.2.1 ApplicationInterface

ApplicationInterface forms the active interface to the application software. As well as the jump address into the application software, RAM cells for communication between the application and ADLATUS® are administered here.

2.2.2 CheckDepend*Customer*

This module is necessary for "UDS"-conforming programming procedures in order to be able to adjust the testing of the programme dependencies in a project-specific fashion following programming.

2.2.3 ConfigProject

This module contains most elements for the project-specific application of the SMART ADLATUS®. The file is divided into configuration areas whose significance will be explained in the subsequent chapters.

2.2.4 CusInterface*Customer*

CusInterface*Customer* forms the interface to the customer-specific application data. Control unit-specific data for identification and actual software stati are stored here. The data must be located at the same location in the control unit throughout the entire product lifecycle and its size may not change.

2.2.5 DataPrepare*Customer*

The module forms an interface for the cryptographic preparation of data (compression/decompression and encrypting/decrypting).

2.2.6 EEP

The module contains the EEPROM-specific function groups for accessing a specific EEPROM. It is settled in the hierarchy between SSI and the low level EEPROM functions.

2.2.7 External Watchdog

The module contains the interface for the activation of an external watchdog for system monitoring. The functions implemented in this module run in the RAM and are also called up during deleting and programming.

2.2.8 External WatchdogFlash

This module contains the interface for the initialisation and activation or deactivation of an external watchdog for system monitoring and run in the flash memory.

2.2.9 NVMHandler

The module forms the interface for accessing an EEPROM of whatever type (NVM .. Non Volatile Memory).

2.2.10 Periphery

The “Periphery” module implements the initialisation and activation of the microcontroller I/O ports.

2.2.11 Security*Customer*

The “Security” module contains the functions for checking access rights. These must generally be modified from project to project.

2.2.12 SSIHandler

The name stands for "Serial Storage Interface" and contains functions necessary for access to the corresponding hardware of the microcontroller.

2.2.13 SPIHandler / Lowlevel Serial Storage

The assigning of a name for this module can vary, as additions of the µC manufacturer are often involved. For some projects, this file can also be entirely dispensed with and its functionality be contained in the EEP module.

2.2.14 Startup / CSTART

System initialisation after a reset is carried out in this module. This often involves assembler code and is thus not only different from processor to processor, but is also dependent upon the compiler used.
All registers, and also the stack, necessary for correct operation of the processor must be initialised in Startup with valid values.

2.2.15 Validation*Customer*

The “Validation” module contains all the ADLATUS® test functions. However, these include not only the tests carried out at system start-up, but also those that check a new application for validity after the flash procedure. This can range from a simple checksum generation by addition to asymmetric signature procedures.

2.2.16 IntVectorMapping

This module contains the interrupt vector tables for the system. If the RESET vector cannot be separated and assigned to a specific address, the entire interrupt vector table must be stored in ADLATUS® and mirrored in the application. Here too, the assigning of names may not agree with the chapter title, as additions or project-specific file(s) may be involved. For some projects, the vector tables are contained in two files, while for others they are completely absent.

2.3 Sources in the 06_hallib directory

The following files are permanent components of the "HAL...lib". If the C-sources are also available for these, one will find the following files in the corresponding directory:

2.3.1 CanHandler

The module contains all the functions necessary to access the CAN controller of the microcontroller.

2.3.2 CanHandlerRam

In the case of programming sequences during which communication continues even while deleting, the communication must take place controlled from the RAM. This module serves this purpose. Its functionality is loaded into the RAM during initialisation and executed there.

2.3.3 HISCheck

The function contained in this module checks the correctness and validity of the HIS flash container transferred via download.

2.3.4 Hal

SMART ADLATUS® needs only a few direct hardware functions besides access to communication interfaces. These are pooled in the HAL module.

2.3.5 HalRam

Access on a HW timer must be possible for triggering the watchdog and the timed sending of messages during deletion or programming. The function necessary for this may only be executed in the RAM and is found in this module.

2.4 Sources in the 07_chklib directory

The checking functions can be used following the system start for function control and following programming. More information on these functions will be provided in a subsequent chapter. The number of functions contained here is project-specific and varies. The library nonetheless always contains one file:

2.4.1 ADLATUS_ValHandler.c

The check process is carried out in the Validation Handler. This means not only the actual controlling of the process and the evaluation of the results, but also the definition of the time necessary for the checking of a number of values. On the basis of this time it is determined when the function must be interrupted in order to, for example, trigger a watchdog or maintain communication.

The other functions are not listed here. Their function is described in the corresponding chapter concerning memory tests.

2.5 Sources in the 08_corelib directory

All of the modules pooled in this library are a permanent component of the ADLATUS core.

2.5.1 ConfigAdlatus

The module contains all elements for the global configuration of the SMART ADLATUS®.

2.5.2 ConfigSystem

The module contains all elements for the system-specific application of the SMART ADLATUS®.

2.5.3 AmemCopy PMemCopy

The two memory copying functions are differentiated in that the one function transmits pointers to addresses and the other unsigned long values.

2.5.4 CollectData

The function of this module is required for microcontrollers that can only program a fixed number of bytes (Pagewrite). The function ensures this.

2.5.5 Random

The module contains the random number generation function for use with Seed&Key.

2.5.6 ServiceHandler

“Service Handler” processes the “Service Requests” received and calls up the affiliated services.

2.5.7 Session

The “Session.c” module contains the session management. This includes the checking of the command sequence and other decisions within a session.

2.5.8 System

The “System” module contains the actual operating system of ADLATUS®.

2.6 Sources in the 09_*cu*slib directory

2.6.1 EventHandler*Customer*

In the EventHandler*Customer* file, specific events, which can occur differently depending upon customer requirements (e.g. switching baud rate, triggering a reset at a certain time, etc.), are handled.

2.6.2 KWP2000*Customer*

The “KWP2000*Customer*" module contains all KWP2000 services needed for a customer-specific (re-)programming procedure as well as the corresponding responses.

2.6.3 Main

The "Main" module is executed after the “Startup” module. In it, all necessary system checks and initialisation functions are carried out and the decision is made whether to stay in ADLATUS® (e.g. because of a "flash request" or faulty application software) or whether to branch into the application.

2.6.4 Sequence*Customer*

The module contains the process sequence for the customer-specific programming procedure.

2.6.5 Setup

The functions in the “Setup” module take over all initialisation procedures necessary for operation of the Adlatus "Core" module, insofar as this has not yet been carried out in the "Main" module.

2.6.6 Timer

The “Timer” module contains the timer functionality.

2.6.7 TP

The transport layer (TP) processes messages according to the defined specifications.

2.7 Directory 10_HIS

The module “HIS” contains all functions for erasing or programming procedures and for checking for correct parameters. For projects in which the API functions are a fixed component of ADLATUS, copy functions are available to load the necessary programs into RAM.

2.8 Header modules

The header files are split into two function blocks. The definitions can be found in the files containing “cdf” (constant definition file), function prototypes and variables in “tdf” (type definition file).
Every C source has an associated “tdf” file.
Two header modules form an exception: in “ADLATUS_Types_tdf.h”, all cross-module types are defined, in the header “ADLATUS_global _cdf.h”, all cross-module "#defines" are listed.

Modifications of project specifics are usually not necessary in the header files. However, an exception is the "ADLATUS_Adr_Info.h" file, in which all addresses used to configure the software are pooled.

2.9 Naming convention

The following name assignment rules are used in ADLATUS:

Functions: FUN_CCmod_funktion_rr:

FUN	The element is a function
mod	The function can be found in the "mod" module
rr	The function has a return value of length “rr”
	rr = V (void)
	rr = UB (unsigned byte 8 Bit)
	rr = UW (unsigned word 16 Bit)
	rr = UL (unsigned long 32 Bit)

Definitions, constants and variables: x_CCmod_name_rr:

x	Type of element
	x = c (constants)
	x = d (value determined by #define directive)
	x = t (variable)
mod	The element is declared in module “mod”
rr	The element has length and type rr
	rr = UB (unsigned byte 8 Bit)
	rr = UW (unsigned word 16 Bit)
	rr = UL (unsigned long 32 Bit)
	rr = PUB (pointer to unsigned byte 8-bit)
	rr = PUW (pointer to unsigned word 16-bit)
	rr = PUL (pointer to unsigned long 32-bit)
	rr = ST (the element is a structure)
	Rr = PST (the element is a pointer to a structure)

Constants whose value is defined by a #define directive have no length information.

3 First settings

Firstly, several basic cross-system settings for SMART ADLATUS® and the development environment must be carried out. These settings are described in the following.

3.1 Software project

All components necessary for the generation of the ADLATUS® must be integrated into a software project. The corresponding file directory structure is predefined by SMART. SMART can only guarantee the desired functionality with this structure.

If the present project contains deviations from the project structure described in chapter two, this will be described in more detail in Appendix A – Software project settings, or in a "readme" file.

3.2 Compiler settings

The existing ADLATUS® software project has been translated using an exactly specified compiler with specific settings and the result has been tested by SMART. SMART only guarantees the desired functionality with this compiler and the settings described. If other compilers or other compiler settings are used, SMART does not guarantee fault-free operation of the resulting ADLATUS®.

The specifications of the compiler and the corresponding settings are described in more detail in Appendix B – Compiler Settings or in a "readme" file.

3.3 Internal system clock frequency and quartz frequency

The SMART ADLATUS needs the internal system clock frequency for several system components. This must be entered appropriately in the area.

Module:	ADLATUS_ConfigProject.c
Application area:	*Clock.01 – Internal Clock Frequency*

The constants field `c_CCconprj_ClockFrequency_UL` is described more closely below:

Format	Designation	Description
4 byte	c_CCconprj_ClockFrequency_UL	Internal system clock frequency

Entry is made in decimal format. The unit is seconds^{-1} (hertz [Hz]).

Example:

Setting the internal system clock frequency to 40 MHz.

```
const ULONG c_CCconprj_ClockFrequency_UL = \
{
  /* Clock Frequency */ 40000000  /* [Hz]   */
};
```

In the case of microcontrollers enabling the setting of the PLL, the constant c_CCconprj_PLL_Value_UL also stands as a value for the PLL directory for some projects in the application area *Clock.01*. It is thereby possible to operate the ADLATUS at a constant internal clock frequency with varying quartz frequencies.

It must thereby be ensured that the values for c_CCconprj_ClockFrequency_UL and c_CCconprj_PLL_Value_UL harmonise. This means, when, for example, a 20 MHz quartz is used, a PLL value must be selected that results in the internal system frequency entered in the constant field c_CCconprj_ClockFrequency_UL; in the example above thus 40 MHz.

With some microcontrollers, the internal clock frequency during the deletion or programming of the flash memory may not amount to the maximum permitted clock frequency (with some controllers this is only half as high). For this reason it may be necessary to operate the ADLATUS with a frequency less than that of the application.

⚠ The values for `c_CCconprj_ClockFrequency_UL` and `c_CCconprj_PLL_Value_UL` must harmonise.

Excessive internal clock frequency values can permanently damage the flash memory when deleting or programming.

The constants field `c_CCconprj_PLL_Value_UL` is described more closely below:

Format	Designation	Description
4 byte	c_CCconprj_PLL_Value_UL	Value for directory PLL_CLC

Entry is made in hexadecimal format. The value has no unit.

Example:

Setting of the internal system clock frequency to 40 MHz at a 20 MHz quartz.

```
const ULONG c_CCconprj_PLL_Value_UL = \
{
  /* PLL_CLC Value */ 0x00272926   /*   */
};
```

4 Memory configuration

4.1 General

When laying out the memory, several points must be taken into account for correct ADLATUS® operation.

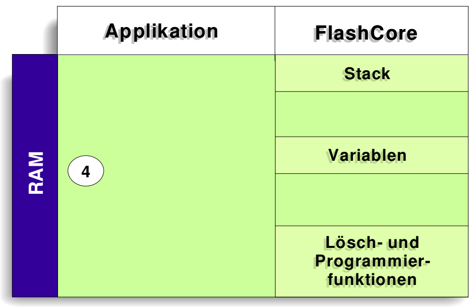
It must be possible to call up all functions used by the ADLATUS®, even when application software is missing. The ADLATUS® has been coded such that it needs a minimum of archive resources (libraries).

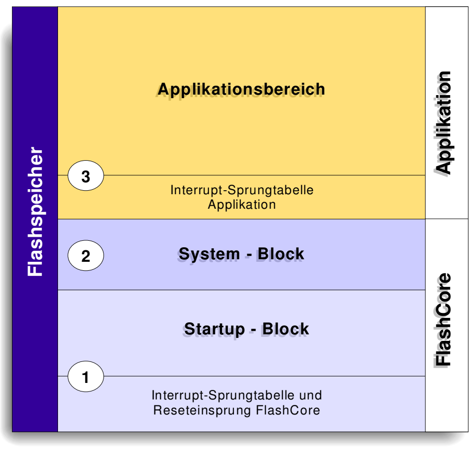
The application and the ADLATUS® should not use functions jointly.

The simplest way of avoiding the joint use of resources is a complete separation of the ADLATUS® from the application by keeping both of them in different projects. When this is not possible due to the project structure, then parts used jointly by the ADLATUS® and the application must be stored in the ADLATUS® memory area and made accessible to the application via fixed addresses. The fixed addresses are necessary since the jump-in address can shift when the ADLATUS® is reprogrammed later.

In general, the structuring of the memory is controlled by a linker command file and parts of the software are assigned to the corresponding memory sections.

The ADLATUS® is split into two sections, the start-up block and the communications/flash application block. The split is merely a logical split, and exists to be able to repeat the programming cycle when reprogramming of the ADLATUS® has failed.




Flashspeicher
Applikationsbereich
3
Interrupt-Sprungtabelle Applikation
Applikation
2
System - Block
Startup - Block
1
Interrupt-Sprungtabelle und Reseteinsprung FlashCore
FlashCore

① The ADLATUS® start-up block contains the jump-in address for the power on reset and the interrupt vector table. After execution of the start-up sequence, a quick check decides whether the program remains in ADLATUS® (checksum field of ADLATUS® contains a value not equal to 0xFF or 0x00) or whether it must be branched into the application area.

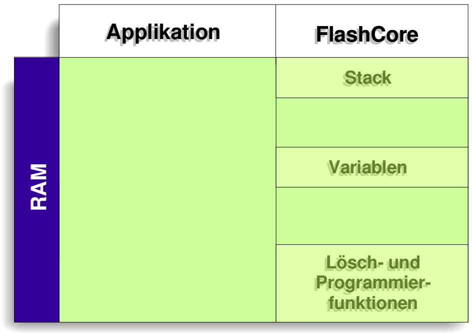
② The second part contains the actual functions of the ADLATUS®. Most of the communications functions and functions necessary for flashing can be found there.

③ The application area contains the second interrupt vector table, which points to the actual interrupt services and the jump-in function from ADLATUS® to the application.

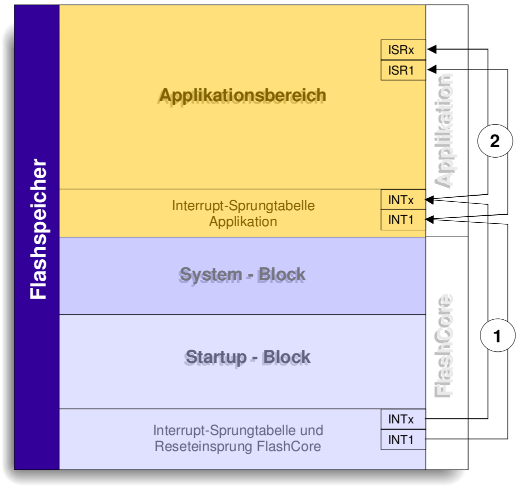
④ The RAM is used by the application and the ADLATUS® independently of one another. Since both parts only use a few memory cells as interface, the major part of the memory can be used by the application as if no ADLATUS® was present. ADLATUS® and application have independent stacks, which do not have to be at the same position.

4.2 Interrupt jump tables

① Since the interrupt jump table in most processors cannot be moved, it is always kept in ADLATUS®. It is, however, not advisable to call the interrupt service functions up directly via this table, since their addresses may have been shifted by a later reprogrammed version of the application and may therefore be inaccessible.



② For this reason a “double jump” procedure is used. The table in ADLATUS® refers to another table in the starting area of the application whose address remains unchanged through the entire product lifecycle. The onwards jump addresses to the actual ISRs are stored in this table.



4.3 Configuration

In order to ensure that no areas of a controller can mistakenly or intentionally be deleted or reprogrammed, the SMART ADLATUS® contains structures for the configuration of the memory map and tests the addresses or values provided by the tester against the set memory configuration at the beginning of a deletion or programming procedure.

This memory configuration can be adjusted by the user individually to the requirements.

The memory configuration is carried out centrally at the following position:

Module:	*ADLATUS_ConfigProject.c*
Application area:	`Mem.01 – Memory table for programming`

The relevant data must thereby be entered into a structure of type `DOWNLOADMAP_ST` (`ADLATUS_Types_tdf.h` → `Config.01`) . The layout of this structure is shown in the following:

Format	Designation	Description
4*1 byte	HIS header:	Indicates the HIS container to be uses
4 byte	StartAddress_UL	Start address of memory area
4 byte	EndAddress_UL	End address of memory area
2 byte	MemoryInfo_UW	Information on memory area defined above

In the case of the indexed addressing of the memory areas, as used for UDS-conforming programming procedures, the structure has two further elements:

2 byte	MemoryIndex_UW	Index for indexed addressing
1 byte	Name_UB	Name of the entry in the download table

The first element of the structure contains the four entries of the His header with the elements listed below. The values can be used for the verification of the appropriate flash container with reference to a memory area:

```
typedef struct
{
    UBYTE ubInterfaceVersion;
    UBYTE ubReserverd;
    UBYTE ubMask;
    UBYTE ubMCU;
    .......
} tHis_Header;
```

The parameter of the HIS header is entered via the following defines. These are contained in the ADLATUS_HISflash.h file.

```
#define FLASH_DRIVER_VERSION_MCUTYPE     0x...
#define FLASH_DRIVER_VERSION_MASKTYPE    0x...
#define FLASH_DRIVER_VERSION_INTERFACE   0x...
```

If, when downloading, addresses are used to identify memory areas to be deleted / programmed, these are compared with the address values and entries of the structure and, where applicable, defined as an invalid area. In the case of indexed addressing, the memory area to be processed is determined on the basis of the index. Here it can only be determined with the help of the flag register "MemoryInfo_UW" whether the area can be manipulated.

All available entries for the setting of `MemoryInfo_UW` have been defined in the module:

Module:	`ADLATUS_Global_cdf.h`
Definition area:	`Global.13 – Memory Map Defines`

The corresponding definitions are described in detail in the following

`MemoryInfo – EntryType`

#Define	Description
`d_CCglo_NormalEntry`	Identifies a normal entry in the table
`d_CCglo_LastEntry`	Identifies the last entry in the table

`MemoryInfo – MemoryType`

#Define	Description
`d_CCglo_FlashMemory`	Memory area is flash memory
`D_CCglo_RamMemory`	Memory area is RAM
`D_CCglo_EepromMemory`	Memory area is EEPROM memory

The memory type is decisive for the deletion, programming or checking functions to be used.

The programming of the EEPROM is currently only supported for UDS-conforming programming procedures.

`MemoryInfo – EraseState`

#Define	Description
`d_CCglo_Erasable`	Memory may be erased
`D_CCglo_NotErasable`	Memory may not be erased

MemoryInfo – ProgrammingState	
#Define	Description
d_CCglo_Programmable	Memory may be programmed
d_CCglo_NotProgrammable	Memory may not be programmed

Under normal conditions, it is unnecessary to differentiate between deletion and programming capability. However, it may be necessary to reprogram parts in a part of the software that cannot be deleted, insofar as the microcontroller supports this. This can thereby involve, for example, programming counters or history fields.

MemoryInfo – CheckState	
#Define	Description
d_CCglo_CheckRange	The checking of the tester addresses against the memory area addresses takes place upon congruence of the area: Start address <= Tester start address <= End address Start address <= Tester end address <= End address
d_CCglo_CheckBoundary	The checking of the tester addresses against the memory area addresses takes place upon congruence of the limiting values: Start address = Tester start address End address = Tester end address

The flag for the CheckState is only evaluated for non-indexed addressing. In the case of indexed addressing, the start address and end address must exactly contain the area to be manipulated, as the software uses both values in order to define what is to be deleted, how large the area to be programmed is and which area should be used for the checksum testing.

Corresponding examples for the respective settings are listed on the following pages:

Example 1: Non-indexed, CheckRange

The following example specifies a flash memory area from C04000$_{hex}$ to C1FFFF$_{hex}$. The memory area is erasable and can be programmed. Transferred addresses of the tester are checked to see whether they lie within the range

C04000$_{hex}$ <= Tester addresses <= C1FFFF$_{hex}$

If this is not the case, erasing or programming is refused by ADLATUS® and a corresponding request is acknowledged with a negative response.

```
const DOWNLOADMAP_ST  c_CCsysprj_DownloadMap_AST[] = \
{
  ...
  {
    /* definition for His-header */
    FLASH_DRIVER_VERSION_INTERFACE,
    0x00,
    FLASH_DRIVER_VERSION_MASKTYPE,
    FLASH_DRIVER_VERSION_MCUTYPE,
    /* ----------------------++------------------------------*/
    /* 4 Byte - Startaddress */ 0xC04000ul,
    /* ----------------------++------------------------------*/
    /* 4 Byte - EndAddress   */ 0xC1FFFFul,
    /* ----------------------++------------------------------*/
    /* 2 Byte - Memory Info  */ (
    /*  Entry Type           */ d_CCglo_NormalEntry |
    /*  Memory Type          */ d_CCglo_FlashMemory |
    /*  Erase state          */ d_CCglo_Erasable    |
    /*  Programming state    */ d_CCglo_Programable |
    /*  Check state          */ d_CCglo_CheckRange  )
    /* ----------------------++-----------------------------*/
  },
  ...
};
```

All entries for `MemoryInfo` must be linked with a bit-wise **OR**!

Example 2: Non-indexed, CheckBoundary

The following example specifies a RAM memory area from C000$_{hex}$ to C7FF$_{hex}$. The memory area is erasable and can be programmed. Transferred addresses of the tester are checked to see whether they exactly cover the range

C000$_{hex}$ = Tester start address , C7FF$_{hex}$ =Tester end address

If this is not the case, erasing or programming is refused by ADLATUS® and a corresponding request is acknowledged with a negative response.

```
const DOWNLOADMAP_ST  c_CCsysprj_DownloadMap_AST[] = \
{
  ...
  {
  /* definition for His-header */
  FLASH_DRIVER_VERSION_INTERFACE,
  0x00,
  FLASH_DRIVER_VERSION_MASKTYPE,
  FLASH_DRIVER_VERSION_MCUTYPE,
  /* ----------------------++-------------------------------*/
  /* 4 Byte - Startaddress */ (ULONG) 0x00c000ul,
  /* ----------------------++-------------------------------*/
  /* 4 Byte - EndAddress   */ (ULONG) 0x00c7FFul,
  /* ----------------------++-------------------------------*/
  /* 2 Byte - Memory Info  */ (
  /*  Entry Type           */ (UWORD) d_CCglo_NormalEntry    |
  /*  Memory Type          */ (UWORD) d_CCglo_RamMemory      |
  /*  Erase state          */ (UWORD) d_CCglo_Erasable       |
  /*  Programming state    */ (UWORD) d_CCglo_Programable    |
  /*  Check state          */ (UWORD) d_CCglo_CheckBoundary  )
  /* ----------------------++-------------------------------*/
  },
  ...
};
```

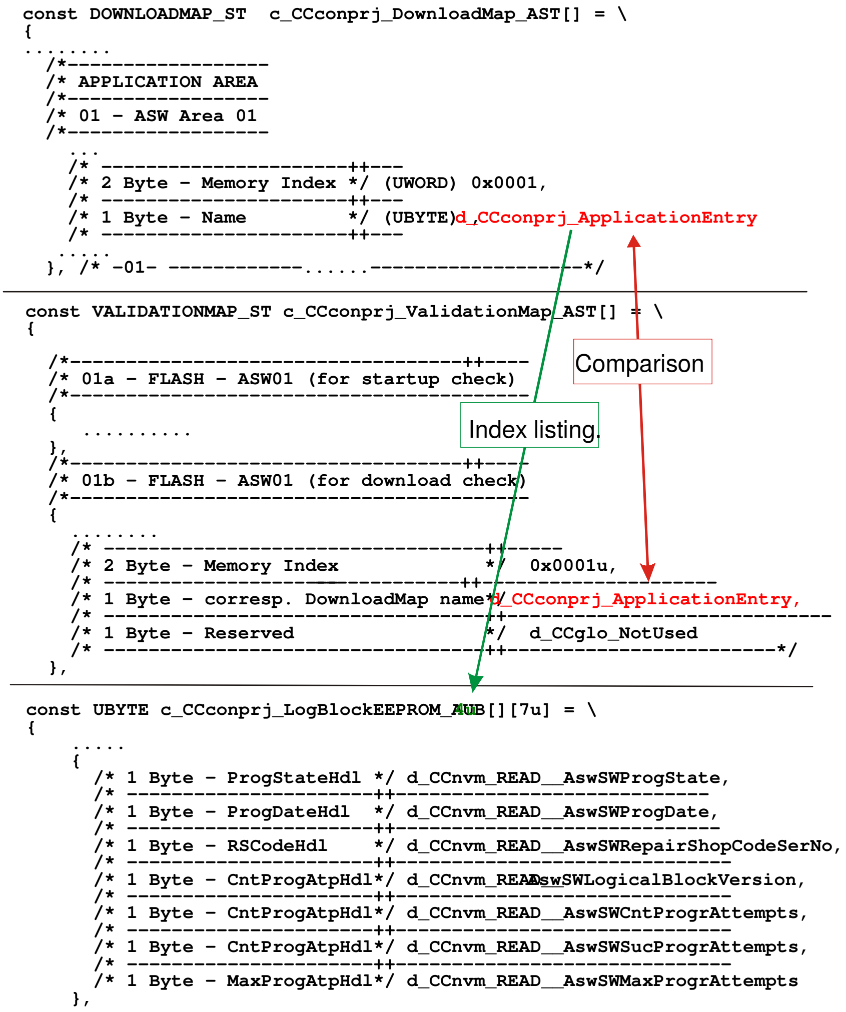
Example 3: Indexed, UDS

The following example specifies a flash memory area from A0040000hex to A0177FFFhex. The memory area is erasable and can be programmed. During the comparison, the ADLATUS recognises the validity of the area via the index of the range (0x0001u).

The entry in the 'DownloadMap' has the unique name ‘d_CCconprj_ApplicationEntry‘. This is used in order to identify the testing range in the 'ValidationMap' and as an index to identify the EEPROM entries assigned to this memory area. It is also used to create a logical link between tested data (check memory) and downloaded data (transfer data). This naming assignment is only needed for flash memory entries in the DownloadMap, since management of programming counters, RepairShopCodes, etc., is necessary here.

```
const DOWNLOADMAP_ST  c_CCconprj_DownloadMap_AST[] = \
{
  ...
  {
  /* definition for His-header */
  .....
  /* ----------------------++--------------------------------*/
  /* 4 Byte - Startaddress */ 0xA0040000ul,
  /* ----------------------++--------------------------------*/
  /* 4 Byte - EndAddress   */ 0xA0177FFFul,
  /* ----------------------++--------------------------------*/
  /* 2 Byte - Memory Info  */ (
  /*  Entry Type           */ d_CCglo_NormalEntry |
  /*  Memory Type          */ d_CCglo_FlashMemory |
  /*  Erase state          */ d_CCglo_Erasable    |
  /*  Programming state    */ d_CCglo_Programable |
  /*  Check state          */ d_CCglo_CheckRange    ),
  /* ----------------------++--------------------------------*/
  /* 2 Byte - Memory Index */ 0x0001u,
  /* ----------------------++--------------------------------*/
  /* 1 Byte - Name         */ d_CCconprj_ApplicationEntry,
  /* ----------------------++--------------------------------*/
  },
  ...
};
```

The representation below shows the connection between the "Name_UB" and the comparisons and references realised with it:



```
const DOWNLOADMAP_ST  c_CCconprj_DownloadMap_AST[] = \
{
........
  /*-------------------
  /* APPLICATION AREA
  /*-------------------
  /* 01 - ASW Area 01
  /*-------------------
    ...
    /* ----------------------++---
    /* 2 Byte - Memory Index */ (UWORD) 0x0001,
    /* ----------------------++---
    /* 1 Byte - Name         */ (UBYTE)d_CCconprj_ApplicationEntry
    /* ----------------------++---
   .....
  }, /* -01- ------------......-------------------*/

const VALIDATIONMAP_ST c_CCconprj_ValidationMap_AST[] = \
{

  /*-----------------------------------++----
  /* 01a - FLASH - ASW01 (for startup check)
  /*----------------------------------------
  {
     ..........
  },
  /*-----------------------------------++----
  /* 01b - FLASH - ASW01 (for download check)
  /*----------------------------------------
  {
    ........
    /* -----------------------------------++------
    /* 2 Byte - Memory Index               */  0x0001u,
    /* -----------------------------------++--------------------
    /* 1 Byte - corresp. DownloadMap name */d_CCconprj_ApplicationEntry,
    /* -----------------------------------++----------------------------
    /* 1 Byte - Reserved                   */  d_CCglo_NotUsed
    /* -----------------------------------++------------------------*/
  },

const UBYTE c_CCconprj_LogBlockEEPROM_AUB[][7u] = \
{
    .....
    {
      /* 1 Byte - ProgStateHdl */ d_CCnvm_READ__AswSWProgState,
      /* ----------------------++-----------------------------
      /* 1 Byte - ProgDateHdl  */ d_CCnvm_READ__AswSWProgDate,
      /* ----------------------++------------------------------
      /* 1 Byte - RSCodeHdl    */ d_CCnvm_READ__AswSWRepairShopCodeSerNo,
      /* ----------------------++------------------------------
      /* 1 Byte - CntProgAtpHdl*/ d_CCnvm_READ__AswSWLogicalBlockVersion,
      /* ----------------------++------------------------------
      /* 1 Byte - CntProgAtpHdl*/ d_CCnvm_READ__AswSWCntProgrAttempts,
      /* ----------------------++------------------------------
      /* 1 Byte - CntProgAtpHdl*/ d_CCnvm_READ__AswSWSucProgrAttempts,
      /* ----------------------++------------------------------
      /* 1 Byte - MaxProgAtpHdl*/ d_CCnvm_READ__AswSWMaxProgrAttempts
    },
```

5 System start

5.1 Introduction

Basically, the ADLATUS® knows two global system states:

No application software programmed (ADLATUS® “standalone”)

Application software programmed

In addition, the ADLATUS® functionality is divided into two phases:

Startup (generally after a RESET)

Normal operation

After system start-up, the ADLATUS® checks the entire system for one of these states (hereafter known as "quick check"). If there is no application software in the system, the ADLATUS® initialises accordingly. If application software is programmed, the next step is a check for a reprogramming request (hereafter known as “flash request”) from the application software. If this is the case, the ADLATUS® initialises accordingly. If no reprogramming request from the application software is present, the correctness of the application software is checked. If this is the case, the application is branched into. If this check recognizes corrupt application software, one generally remains in ADLATUS® to program new application software.

The ADLATUS® must be appropriately configured for the tests during the start-up phase described above. These settings are described in the following chapter in more detail.

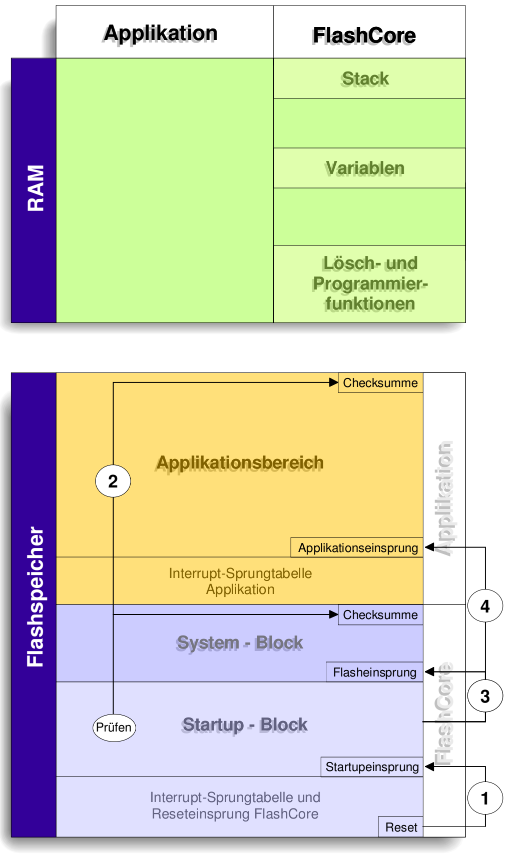
SMART

5.2 Start-up sequence

① After a reset, the start-up block is entered first. There, the system registers are initialised and the stack is set up.

② A check is carried out in the next step to determine whether valid software is programmed in the system block and the application. To this purpose, a check is merely made to determine whether a value not equal to 0xFF or 0x00 exists (described as “quick check’ in the following) at a predetermined position. If the system block is recognized as valid, the calculation (validation) of the checksum can commence.

③ The system block is jumped into when the application area does not contain valid data, the wait time for a trigger message has expired (if implemented) or a "flash request" made. Prior to this, the system is initialised and the system checks executed.



④ The application is branched into if an invalid system block is recognized during the quick check or after a timeout when waiting for further messages after a "flash request".

5.3 Flash request, general

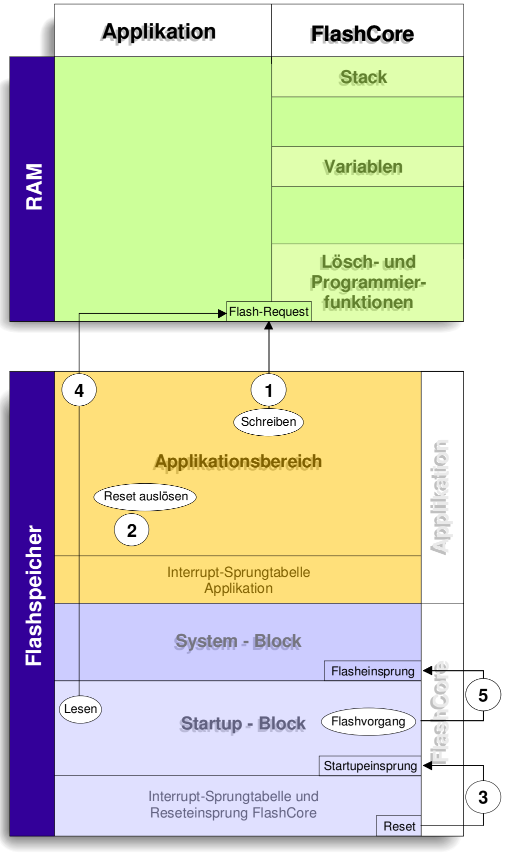
① If a valid application is programmed in the system, the first messages are responded to by the application during the programming procedure. Following a defined message, ADLATUS® handles the subsequent message traffic. In order to inform ADLATUS® that a flash request is present, the application must write a pattern in a memory cell known to both ADLATUS® and the application.

② After setting the flash request cell in the blockage of the interrupt, the application triggers a reset.

③ After the reset, the start-up block is branched into.

④ In the start-up block, the specific cell for a "flash request" is read.

⑤ If the pattern is set to “flash request”, a branch into the system block is executed and further commands are waited for there.



If the ADLATUS® is in a "standalone" state, it simulates the behaviour of the application.

5.4 I/O initialisation

During the start-up phase, the I/O ports of the microcontroller are configured in such a way that the attached peripheral elements (sensors and actuators) are in a safe operating condition.

These settings are project-specific and must therefore be carried out independently by the user. For this purpose, an interface in the form of two function call-ups is provided by the ADLATUS®. These interfaces are described in detail in the following.

5.4.1 Initialisation after start-up phase

Module:	*ADLATUS_Periphery.c*
Application area:	`SET_IO.01 – Function Definitions`

Function description

This initialises the ADLATUS® after the start-up phase. All I/O ports must be initialized in such a way that the system is in a safe state.

Prototype:

void FUN_CCper_InitIOStartup_V (void);

Parameter:

-

Return value:

-

Example:

```
void FUN_CCper_InitIOStartup_V (void)
{
  /*... write here your code .*/..
  return;
}
```

5.4.2 Initialisation of ADLATUS prior to start of communication

Module:	*ADLATUS_Periphery.c*
Application area:	SET_IO.02 – Function Definitions

Function description

This function makes it possible, where necessary, to once again initialise the system for the ADLATUS® especially for "standalone operation" or the communication for a programming session. This may, but must not be necessary. If this functionality is not needed, a jump back is made immediately after the calling up.

Prototype:

void FUN_CCper_InitIOForAdlatus_V (void);

Parameter:

-

Return value:

-

Example:

```
void FUN_CCper_InitIOForAdlatus_V (void)
{
  /*... write here your code .*/..
  return;

}
```

6 Memory test

6.1 Overview

In operation, the SMART ADLATUS® executes memory tests at various times. These tests depend on the respective memory type and the foundational project-specific requirements. The user can therefore freely configure these tests. These are differentiated according to the following criteria:

- ❖ RAM/CANRAM memory test after a reset/power on in the course of the system test
- ❖ Memory test through checksum calculations after a reset/power on in the course of the system test
- ❖ Memory test by after a programming procedure in the course of checksum calculations

6.1.1 Basic concept of validation handler

The concept of the Validation Handler provides for strict encapsulation of the functions of the ADLATUS® core. The CheckLib functionality has a two-layer structure: the Validation Handler and the checksum functions used.

The timing behaviour of the check is defined by the Validation Handler. To this purpose, the address area to be checked is calculated in small units by the respective checksum functions. The run time for this calculation is measured and evaluated. On the basis of this time measurement, the address area that is transferred to the checksum routines is increased until a given time pattern (1ms) is maintained. After this target value is reached, no additional time measurements are carried out. If the total address area set during the initialisation has been checked, the Validation Handler calls up the checksum function with an address range of zero bytes. The checksum function then provides the result of the check.

For quick checks, the check is called up only once in order to provide the result. Calculations cannot be carried out here.

6.2 System test after reset

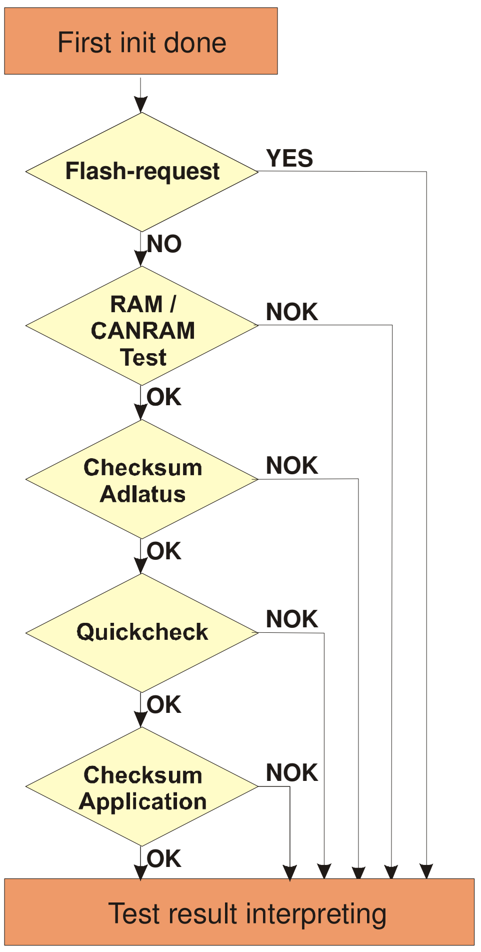
The following system tests can be carried out through corresponding settings:

- ❖ Checksum calculation of the software blocks in the flash memory
- ❖ RAM memory test
- ❖ Test of the send or receive buffer of the hardware interface (e.g. the CAN mailbox buffer)

The processing sequence of the individual test steps is displayed to the right.

If the ADLATUS recognizes a programming wish, it is presumed that the system is free of errors and that a start-up test is unnecessary.

Because the checksum procedure requires a great deal of time for your calculations, it is often dispensed with during the system start. Instead, only the so-called quick check is used to recognise a valid application.



All system tests are configured at the position designated as follows:

Module:	`ADLATUS_ConfigProject.c`
Application area:	`Val.01 – Memory table for system check`

The relevant data must thereby be entered into a structure of type `VALIDATIONMAP_ST` (`ADLATUS_Types_tdf.h` → `Type definition Validation.02`). The layout of this structure is shown in the following:

Format	Designation	Description
2 byte	CheckCondition	Configuration register for the respective check
2 byte	ErrorMessageEntry	Entry for the error message
4 byte	Startaddress	Start address of the memory area to be checked
4 byte	Stopaddress	Stop address of the memory area to be checked
4 byte	Checksumaddress	Checksum address of the memory area to be checked
4 byte	Quickcheckaddress	Quick check address of the memory area to be checked
2 byte	CheckRoutine	Pointer to the corresponding test routine for this memory area

In the case of the indexed addressing of the memory areas, as used for UDS-conforming programming procedures, the structure has two further elements:

2 byte	MemoryIndex_UW	Index for indexed addressing
1 byte	DownloadName_UW	Affiliated download – Table entry.

All available entries for the setting of `CheckCondition` have been defined in the module:

Module:	*ADLATUS_Global_cdf.h*
Definition area:	`Global.12 – Validation Map Defines`

The corresponding definitions are described in detail in the following

#Define	Description
d_CCglo_NormalEntry	Identifies a normal entry in the table
d_CCglo_LastEntry	Identifies the last entry in the table
d_CCglo_CheckDisabled	The corresponding test is deactivated
d_CCglo_CheckEnabled	The corresponding test is activated
d_CCglo_FlashMemory	Memory area is flash memory
d_CCglo_RamMemory	Memory area is RAM
d_CCglo_EepromMemory	Memory area is EEPROM
d_CCglo_CanMsgObject	Memory area is a CAN mailbox buffer
d_CCglo_AlternateMethod	Use alternative quick check method
d_CCglo_AlternateMethod2	Use alternative quick check method 2
d_CCglo_DontInitRam	RAM is not initialised after test
d_CCglo_InitRam	RAM is initialised after test with a uniform value
d_CCglo_DontDoQuickCheck	Memory area is not subject to quick check testing
d_CCglo_DoQuickCheck	Memory area is subject to quick check testing
d_CCglo_01Byte	Quick check area is 1 byte
d_CCglo_02Byte	Quick check area is 2 byte
d_CCglo_04Byte	Quick check area is 4 byte
d_CCglo_NotUsed	Entry for unnecessary information

The entry `StartAddress` defines the start address for the respective test. This is entered as a 4 byte value. The entry `StopAddress` defines the stop address for the respective test. This is entered as a 4 byte value. Start and stop address define a memory area in the application software. If this area lies within the flash memory, the entries for `ChecksumAddress` and the `QuickCheckAddress` are also relevant. The `ChecksumAddress` defines the memory position in which the checksum of the area described by `StartAddress` and `StopAddress` is stored. This checksum is recalculated and checked by the function designated in `CheckRoutine`. If the recalculated checksum does not agree with the entry at the address defined by `ChecksumAddress`, the value from the `ErrorMessageEntry` entry is entered in the global system status variable.
The `QuickCheckAddress` entry is needed to check whether application software is present. Only the content of the memory position defined by `QuickCheckAddress` is thereby checked. If this is not equal to 00_{hex} or FF_{hex} (possible value of the cell after erasure), then application software has been programmed. `ChecksumAddress` and `QuickCheck-Address` can mark the same memory position. Since quick check is the only way the ADLATUS® can check for the presence of application software and declare it valid if the checksum calculation is disabled (the corresponding field in the table is set to `d_CCglo_CheckDisabled`), the address of the memory cells to be checked must lie at the end of the code area of the application.

For a RAM test, the areas `ChecksumAddress` and `QuickCheckAddress` are not relevant. If a fault is identified during the RAM test, the entry `ErrorMessageEntry` is entered in the global system status variable.

The entry `CheckRoutine` is a pointer to the corresponding test routine.

The necessary settings for each test case in the fields described above are shown in detail in the following.

Currently, the following test routines are available:

Checksum calculation

Designation	Description
`FUN_CCval_LinAdd8Acc8Bit_UB`	Linear 8-bit upwards addition into an 8-bit checksum.
`FUN_CCval_LinAdd8Acc16Bit_UB`	Linear 8-bit upwards addition into a 16-bit checksum.
`FUN_CCval_LinAdd8Acc32Bit_UB`	Linear 8-bit upwards addition into a 32-bit checksum.
`FUN_CCval_LinAdd16Acc16Bit_UB`	Linear 16-bit upwards addition into a 16-bit checksum.
`FUN_CCval_CCITT_UB`	Calculation by means of CCITT algorithm.
`FUN_CCval_CRC32_UB`	Calculation by means of CRC32 algorithm.

Starting value for the CCITT algorithm:

In the case of the checksum calculation via the CCITT procedure, it is possible to indicate a starting value for the calculation.

Module:	`ADLATUS_ConfigProject.c`
Application area:	`Checksum.01 – Starting value of the CCITT algorithm`

const UWORD c_CCconprj_StartValue_CRC16_UW = 0x0000;

6.2.1 Checksum calculation

For calculation of a checksum across a defined flash memory area, the following entries (marked in red) are necessary:

```
/*-----------------------------------+--------------------------------*/
/* 01 - FLASH - FSW01                                                 */
/*-----------------------------------+--------------------------------*/
{
/* ----------------------------------+--------------------------------*/
/* 2 Byte - CheckState:              */ (
/*   Bit 15     - Entry type         */ (UWORD) d_CCglo_NormalEntry   |
/*   Bit 14     - En/Dis Check       */ (UWORD) d_CCglo_CheckEnabled  |
/*   Bit 13/12 - Memory Type         */ (UWORD) d_CCglo_FlashMemory   |
/*-----------------------------------+--------------------------------*/
/*   Memorytype: RAM                                                  */
/*   Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_NotUsed      |
/* ----------------------------------+--------------------------------*/
/*   Memorytype: Flash               */
/*   Bit 03     - Do QuickCheck ?    */ (UWORD) d_CCglo_DoQuickCheck |
/*   Bit 02-00 - QuickCheckLength    */ (UWORD) d_CCglo_02Byte       ),
/* ----------------------------------+--------------------------------*/
/* 2 Byte - Error Message entry      */ (UWORD) d_CCconprj_ChecksumFSWNOK,
/* ----------------------------------+--------------------------------*/
/* 4 Byte - Startaddress             */ (ULONG) 0x00004000,
/* 4 Byte - StopAddress              */ (ULONG) 0x0000F800,
/* 4 Byte - Checksum address         */ (ULONG) 0x0000FA00,
/* 4 Byte - Quick check address      */ (ULONG) 0x0000FA00,
/* ----------------------------------+--------------------------------*/
/* 2 Byte - Check Function address   */ FUN_CCval_LinAdd8Acc16Bit_UB
/* ----------------------------------+--------------------------------*/
},
```

`d_CCglo_NormalEntry` identifies a normal table entry. The test is activated by `d_CCglo_CheckEnabled` and identifies a memory area in flash through `d_CCglo_FlashMemory`. This area is described by the entries `StartAddress` and `StopAddress` (addresses 4000$_{hex}$ – F800$_{hex}$). The checksum for this area is found at the address described by `ChecksumAddress` (FA00$_{hex}$). The checksum is calculated by the routine entered in the field `CheckRoutine` (FUN_CCval_LinAdd8Acc16Bit_UB). In event of an error, the value from `ErrorMessageEntry` is written into the global system status variable.
All contents that are not needed can be configured with `d_CCglo_NotUsed`.

All entries for `CheckCondition` must be linked with a bit-wise OR!
The last entry in this table must be identified with `d_CCglo_LastEntry`.

6.2.2 Quick check

Designation	Description
`d_CCglo_DoQuickCheck`	Memory area is subject to quick check testing
`d_CCglo_AlternateMethod`	Use alternative quick check method
`d_CCglo_AlternateMethod2`	Use alternative quick check method 2

Quick check offers the possibility of determining the status of a memory area very quickly. If the value of the memory position identified by `QuickCheckAddress` is not equal to 00_{hex} or FF_{hex}, the memory cell or memory area is deemed to be programmed. If, however, `d_Ccglo_Alternate Method` is chosen while the quick check is switched on, the cells `StartAddress -2` and `StopAddress +1` are checked for congruence and content not equal to 00 $_{hex}$. If `d_CCglo_AlternateMethod2` is also selected, a comparison is only made for congruence. The value can also contain 00 $_{hex}$ or FF $_{hex}$.

`d_CCglo_NormalEntry` identifies a normal table entry. The test is activated by `d_CCglo_DoQuickCheck` and identifies a memory area in flash through `d_CCglo_FlashMemory`. The quick check data length is 2 bytes (`d_CCglo_02Byte`) and should be executed at `ChecksumAddress` ($FA00_{hex}$). `ChecksumAddress` and `QuickCheckAddress` can mark the same memory position. In event of an error, the value from `ErrorMessageEntry` is entered into the global system status variable. All unnecessary content can be configured with `d_CCglo_NotUsed`.

For execution of a quick check, the following entries (marked in red) are necessary:

```
/*-----------------------------------------------------------------------*/
/* 01 - FLASH - FSW01                                                    */
/*-----------------------------------------------------------------------*/
{
/* ----------------------------------++---------------------------------*/
/* 2 Byte - CheckState:              */ (
/*    Bit 15     - Entry type        */ (UWORD) d_CCglo_NormalEntry  |
/*    Bit 14     - En/Dis Check      */ (UWORD) d_CCglo_CheckEnabled |
/*    Bit 13/12 - Memory Type        */ (UWORD) d_CCglo_FlashMemory  |
/*-----------------------------------++---------------------------------*/
/*    Memorytype: RAM                                                    */
/*    Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_NotUsed      |
/* ----------------------------------++---------------------------------*/
/*    Softwaretype: Type entry
/*    Bit 08     - ADLATUS/ASW entry  */ (UWORD) d_CCglo_AswArea       |
/*-----------------------------------++-----------------------------------*/
/*    Quickchecktype                                                     */
/*    Bit 07     - Type of check     */ (UWORD) d_CCglo_AlternateMethod    |
/*    Bit 06     - Method 2          */ (UWORD) d_CCglo_AlternateMethod2   |
/* ----------------------------------++----------------------------------*/
/*    Memorytype: Flash              */
/*    Bit 03     - Do QuickCheck ?   */ (UWORD) d_CCglo_DoQuickCheck       |
/*    Bit 02-00 - QuickCheckLength   */ (UWORD) d_CCglo_02Byte            ),
/* ----------------------------------++---------------------------------*/
/* 2 Byte - Error Message entry      */ (UWORD) d_CCconprj_ChecksumFSWNOK,
/* ----------------------------------++---------------------------------*/
/* 4 Byte - Startaddress             */ (ULONG) 0x00004000,
/* 4 Byte - StopAddress              */ (ULONG) 0x0000F800,
/* 4 Byte - Checksum address         */ (ULONG) 0x0000FA00,
/* 4 Byte - Quick check address      */ (ULONG) 0x0000FA00,
/* ----------------------------------++---------------------------------*/
/* 2 Byte - Check Function address   */ FUN_CCval_LinAdd8Acc16Bit_UB
/* ----------------------------------++---------------------------------*/
},
```

⚠ All entries for `CheckCondition` must be linked with a bit-wise OR!

6.2.3 RAM test

Designation	Description
FUN_CCval_RamPatternTest_UB	Pattern test for RAM. Each memory cell is described with a different pattern and read out again.
FUN_CCval_CanMsgObjectTest_UB	Pattern test for CAN mailbox buffer. Each buffer cell is described with a different pattern and read out again.

For a RAM test across a defined RAM area, the following entries (marked in red) are necessary:

```
/*----------------------------------++---------------------------------*/
/* 02 - RAM - DataRam                                                  */
/*---------------------------------------------------------------------*/
{
/* ---------------------------------++---------------------------------*/
/* 2 Byte - CheckState:             */ (
/*   Bit 15     - Entry type        */ (UWORD) d_CCglo_NormalEntry   |
/*   Bit 14     - En/Dis Check      */ (UWORD) d_CCglo_CheckEnabled  |
/*   Bit 13/12  - Memory Type       */ (UWORD) d_CCglo_RAMMemory     |
/*----------------------------------++---------------------------------*/
/*   Memorytype: RAM                                                   */
/*   Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_DontInitRam   |
/* ---------------------------------++---------------------------------*/
/*   Memorytype: Flash              */
/*   Bit 03     - Do QuickCheck ?   */ (UWORD) d_CCglo_NotUsed       |
/*   Bit 02-00  - QuickCheckLength  */ (UWORD) d_CCglo_NotUsed       ),
/* ---------------------------------++---------------------------------*/
/* 2 Byte - Error Message entry     */ (UWORD) d_CCconprj_RAMCheckNOK,
/* ---------------------------------++---------------------------------*/
/* 4 Byte - Startaddress            */ (ULONG) 0x00004000,
/* 4 Byte - StopAddress             */ (ULONG) 0x000047FF,
/* 4 Byte - Checksum address        */ (ULONG) d_CCglo_NotUsed,
/* 4 Byte - Quick check address     */ (ULONG) d_CCglo_NotUsed,
/* ---------------------------------++---------------------------------*/
/* 2 Byte - Check Function address  */ FUN_CCval_RamPatternTest_UB
/* ---------------------------------++---------------------------------*/
},
```

`d_CCglo_NormalEntry` identifies a normal table entry.

The test is activated by `d_CCglo_CheckEnabled` and identifies a memory area in RAM through `d_CCglo_RamMemory`. This area is described by the entries `StartAddress` and `StopAddress` (addresses 4000_{hex} – $47FF_{hex}$). The test is executed by the routine entered in the field `CheckRoutine` (`FUN_CCval_RamPatternTest_UB`).

After the RAM cell has been tested with different patterns (typ. 55_{hex}, AA_{hex}, 00_{hex}), it is initialised according to the content of the field `CheckCondition`. If `d_CCglo_DontInitRam` has been set, the original value of the memory position is written back. For the setting `d_CCglo_InitRam`,
the memory position is initialised with the value 00_{hex}.

All unnecessary content can be configured with `d_CCglo_NotUsed`.

⚠ For reasons of homogeneity, RAM test routines and checksum routines have the same prototypes. However in RAM tests, the transfer parameter “ULONG ChecksumAddress_UL” is not evaluated.

6.2.4 CAN mailbox test

The following entries (marked in red) are necessary for a CAN mailbox test:

```
/*-----------------------------------++---------------------------------*/
/* 02 - RAM - DataRam                                                    */
/*---------------------------------------------------------------------*/
{
/* ----------------------------------++---------------------------------*/
/* 2 Byte - CheckState:               */ (
/*   Bit 15     - Entry type          */ (UWORD) d_CCglo_NormalEntry  |
/*   Bit 14     - En/Dis Check        */ (UWORD) d_CCglo_CheckEnabled |
/*   Bit 13/12 - Memory Type          */ (UWORD) d_CCglo_RamMemory    |
/*-----------------------------------++---------------------------------*/
/*   Memorytype: RAM                                                     */
/*   Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_DontInitRam  |
/* ----------------------------------++---------------------------------*/
/*   Memorytype: Flash                */
/*   Bit 03     - Do QuickCheck ?     */ (UWORD) d_CCglo_NotUsed      |
/*   Bit 02-00 - QuickCheckLength     */ (UWORD) d_CCglo_NotUsed      ),
/* ----------------------------------++---------------------------------*/
/* 2 Byte - Error Message entry       */ (UWORD) d_CCconprj_CanMsgObjCheckNOK,
/* ----------------------------------++---------------------------------*/
/* 4 Byte - Startaddress              */ (ULONG) 0x00004000,
/* 4 Byte - StopAddress               */ (ULONG) 0x000047FF,
/* 4 Byte - Checksum address          */ (ULONG) d_CCglo_NotUsed,
/* 4 Byte - Quick check address       */ (ULONG) d_CCglo_NotUsed,
/* ----------------------------------++---------------------------------*/
/* 2 Byte - Check Function address    */ FUN_CCval_RamPatternTest_UB
/* ----------------------------------++---------------------------------*/
},
```

`d_CCglo_NormalEntry` identifies a normal table entry.

The test is activated by `d_CCglo_CheckEnabled` and identifies a memory area in RAM through `d_CCglo_RamMemory`. The test is executed by the routine entered in the field `CheckRoutine` (`FUN_CCval_CanMsgObject Test_UB`).

The function `FUN_CCval_CanMsgObjectTest_UB` checks all cells of all CAN message objects with an appropriate pattern (typ. 55_{hex}, AA_{hex}, 00_{hex}). When all cells have been tested, the memory cells are initialized with the value 00_{hex}.

6.3 Troubleshooting start-up

In event of an error, the value from `ErrorMessageEntry` is entered into the global system status variable t_InitCtrl_ST.SystemState_UB.

The following error bits are defined in the system status variable `t_InitCtrl_ST. SystemState_UB`.

Designation		Description
d_CCglo_RamCheckError	`0x01`	Is set when an error has occurred during a RAM test.
d_CCglo_AdlatusChecksumError	`0x02`	Is set when a checksum error is recognised at the ADLATUS itself.
d_CCglo_NoFswInSystem	`0x04`	Is set when ADLATUS recognises that no driver software (FSW) is present.
d_CCglo_FSWChecksumError	`0x08`	Is set when ADLATUS recognises that driver software (FSW) is present, but contains a checksum error.
d_CCglo_FSWFlashRequest	`0x10`	Is set when ADLATUS recognises that an exit from the FSW has taken place.

A recognised error results in the software remaining in the ADLATUS.

6.4 Checksums test after a programming procedure

Verification of a programming procedure is established by calculating a checksum across the programmed data. Both programming procedures in the flash memory and in the RAM or EEPROM (EEPROM only with UDS) are thereby checked. Standard function call-ups are available for all variants. These functions are called up at the appropriate time from the download sequence. The appropriate routing for the checksum calculation can now be accessed within these functions. For reasons of code reduction, the same checksum routines are offered as for the system check. The following describes details of the settings necessary for a checksum calculation after a programming procedure.

The checksum tests carried out after a programming procedure are configured at the same position as the system tests after a reset.

Module:	ADLATUS_ConfigProject.c
Application area:	Val.01 – Memory table for system check

6.4.1 Test after flash programming

For a check of correct programming of the flash memory, the following two functions are used:

```
FUN_CCval_InitCalcChecksumFlashDownload_UB
```

is used to initialise the checksum calculation.

```
FUN_CCval_CalcChecksumFlashDownload_UB
```

is called cyclically until the checksum calculation is finished. An interruption of the calculation is necessary to allow communication with the tester to be maintained when large code blocks have to be checked.

6.4.1.1 Initialisation of the checksum function

Function description

This function is used to fill the working structure of the checksum test values and to announce the checksum function to be used.

Prototype:

```
UBYTE  FUN_CCval_InitCalcCsumFlashDownload_UB (
          ULONG  t_StartAddress_UL,
          ULONG  t_StopAddress_UL,
          ULONG  t_ChecksumAddress_UL )
```

Parameter:

`ULONG t_StartAddress_UL`	Start address of checksum test
`ULONG t_StopAddress_UL`	Stop address of checksum test
`ULONG t_ChecksumAddress_UL`	Address of checksum

Return value:

d_CCglo_NoError	(0x00)	No error

Example:

See following page

Example:

```
UBYTE FUN_CCvalaudi_InitCalcCsumFlashDownload_UB( \
                   ULONG  t_StartAddress_UL, \
                   ULONG  t_StopAddress_UL,    \
                   ULONG  t_ChecksumAddress_UL )
{
  UBYTE t_Result_UB;

  /*---------------------------------------------------------------*/
  /* Init the calculation                                          */
  /*---------------------------------------------------------------*/
  CCValAudi_WorkingInfoInt_ST.StartAddress_PBUF =
                             (VHND_PTR_BUF_UBYTE) t_StartAddress_UL;
  CCValAudi_WorkingInfoInt_ST.EndAddress_PBUF = (
                               VHND_PTR_BUF_UBYTE) t_StopAddress_UL;
  CCValAudi_WorkingInfoInt_ST.ChecksumAddress_PBUF =
                          (VHND_PTR_BUF_UBYTE) t_ChecksumAddress_UL;

  CCValAudi_WorkingInfoInt_ST.WorkingBuffer_UL = 0x00000000ul;

  t_Result_UB = FUN_CCVhnd_InitCtrlST_UB \
         (&CCValAudi_WorkingInfoInt_ST, FUN_CCval_LinAdd8Acc16Bit_UB);

  if(d_CCglo_Ready == t_Result_UB)
 {
    t_Result_UB = d_CCglo_Busy ;
    t_FlashCheckState_UB = d_CCglo_Busy;
  }
  else
 {
    t_Result_UB = d_CCglo_Error;
    t_FlashCheckState_UB = d_CCglo_Error;
  }

  return (t_Result_UB);

} /*- END OF FUNCTION FUN_CCvalaudi_InitCalcCsumFlashDownload_UB -*/
```

The place in the programme marked red identifies the position at which the selected checksum method is made known. If another procedure is to be used, the function name will be changed.

6.4.1.2 Checksum calculation

Function description

The actual checksum formation is controlled with this function.

Prototype:

```
UBYTE  FUN_CCval_CalcCsumFlashDownload_UB (void )
```

Parameter:

```
none
```

Return value:

d_CCglo_Busy	(0x40)	The checksum calculation is not finished.
d_CCglo_Ready	(0x80)	The checksum calculation is finished, correct checksum.
d_CCglo_Error	(0x20)	The checksum calculation is finished, faulty checksum.

Example:

See following page

Example:

```
UBYTE FUN_CCvalaudi_CalcCsumFlashDownload_UB (void )
{

  ULONG              t_LoopCounter_UL;

  /*-------------------------------------------------------------*/
  /* Init the calculation                                        */
  /*-------------------------------------------------------------*/

  /* response pending time in ms */
  t_LoopCounter_UL = 0x0A;

  t_FlashCheckState_UB = d_CCglo_Busy;
  /*==============================================================*/
  /* -1- Calculate checksum                                       */
  /*     but interrupt the calculation for checking communication */
  /*==============================================================*/
  while ((t_LoopCounter_UL != \
       0x00000000ul) && (t_FlashCheckState_UB == d_CCglo_Busy))
  {
    /*-----------------------------------------------------------*/
    /* Calculate Checksum                                        */
    /*-----------------------------------------------------------*/

    /* Returns Busy as long as Calculation is not finished */
    t_FlashCheckState_UB = FUN_CCVhnd_CalcChksumChunk_UB();

    /*==========================================================*/
    /* Time to check the system clock                           */
    /*==========================================================*/

    FUN_CCtim_SystemTimerCheck_V ();
    t_LoopCounter_UL --;
  }


  return (t_FlashCheckState_UB);

} /*-+- END OF FUNCTION FUN_CCvalaudi_CalcCsumFlashDownload_UB   -*/
```

6.4.1.3 Management of software blocks

In the case of a UDS-conforming programming cycle, the testing of the checksum of a flash area in ADLATUS® is followed by the administering of the tested software block. To do so, the entry 'DownloadName' in the 'VALIDATIONMAP' is needed. This uniquely identifies the corresponding entry in the 'DOWNLOADMAP'. With the aid of this link, the programming date or the RepairShopCode of the software block are updated, for example. The individual steps executed are project-specific.

If the checked flash area is not to be assigned a 'DOWNLOADMAP' entry, it is marked as follows:

```
/* ----------------------------------++----------------------------------*/
/* 1 Byte - corresp. DownloadMap name*/ (UBYTE) d_CCglo_NoEntry,
/* ----------------------------------++----------------------------------*/
```

A separate entry in 'VALIDATIONMAP' can be used for checksum calculation after a download. This makes sense, for example, for indexed addressing if the start or stop address differ in comparison to the checksum calculation at start-up.

For a checksum calculation after a flash download, the following entries (marked in red) are necessary:

```
/*-------------------------------------------------------------------------*/
/* 01a - FLASH - FSW01 (for start-up check)                                 */
/*-------------------------------------------------------------------------*/
{
/* ----------------------------------++-----------------------------------*/
/* 2 Byte - CheckState:              */  (
/*   Bit 15     - Entry type         */  (UWORD) d_CCglo_NormalEntry       |
/*   Bit 14     - En/Dis Check       */  (UWORD) d_CCglo_CheckDisabled     |
/*   Bit 13/12 - Memory Type         */  (UWORD) d_CCglo_FlashMemory       |
/* ----------------------------------++-----------------------------------*/
/*   Memorytype: RAM                                                     */
/*   Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_NotUsed       |
/* ----------------------------------++-----------------------------------*/
/*   Memorytype: Flash               */
/*   Bit 03     - Do QuickCheck ?    */  (UWORD) d_CCglo_DontDoQuickCheck  |
/*   Bit 02-00 - QuickCheckLength    */  (UWORD) d_CCglo_02Byte        ),
/* ----------------------------------++-----------------------------------*/
/* 2 Byte - Error Message entry      */  (UWORD) d_CCconprj_ChecksumFSWNOK,
/* ----------------------------------++-----------------------------------*/
/* 4 Byte - Startaddress             */  (ULONG) 0xA0040000,
/* 4 Byte - StopAddress              */  (ULONG) 0xA0177FFF,
/* 4 Byte - Checksum address         */  (ULONG) d_CCglo_NotUsed,
/* 4 Byte - Quick check address      */  (ULONG) d_CCglo_NotUsed,
/* ----------------------------------++-----------------------------------*/
/* 2 Byte - Check Function address   */ d_CCglo_NotUsed,
/* ----------------------------------++-----------------------------------*/
/* 2 Byte - Memory Index             */  (UWORD) 0x0002,
/* ----------------------------------++-----------------------------------*/
/* 1 Byte - corresp. DownloadMap name*/  (UBYTE) d_CCconprj_ApplEntry,
/* ----------------------------------++-----------------------------------*/
/* 1 Byte - Reserved                 */  (UBYTE) d_CCglo_NotUsed
/* ----------------------------------++-----------------------------------*/
}
```

`d_CCglo_NormalEntry` identifies a normal table entry. A memory area in flash is identified by means of `d_CCglo_FlashMemory`. This area is described by the entries `StartAddress` and `StopAddress` (addresses A0040000$_{hex}$ – A0177FFF$_{hex}$). The memory area can also be addressed via `Memory Index` (0002$_{hex}$).
This entry in '`VALIDATIONMAP`' has a corresponding entry in '`DOWNLOADMAP`', identified with `d_CCconprj_ApplEntry` (`corresp. to DownloadMap name`).
All unnecessary content can be configured with `d_CCglo_NotUsed`.
An unused '`corresp. DownloadMap name`' is configured with `d_CCglo_NoEntry`.

6.4.2 Test after RAM programming

The correct programming of the RAM can be checked by calling up the function `FUN_CCval_CalcCsumRamDownload_UB`. The actual checksum routine is called up from within this function.

Function description

Prototype:

```
UBYTE  FUN_CCval_CalcCsumRamDownload_UB (
         ULONG  t_StartAddress_UL,
         ULONG  t_StopAddress_UL,
         ULONG  t_ChecksumAddress_UL )
```

Parameter:

`ULONG t_StartAddress_UL`	Start address of checksum test
`ULONG t_StopAddress_UL`	Stop address of checksum test
`ULONG t_ChecksumAddress_UL`	Address of checksum

Return value:

d_CCglo_NoError (0x00)	No error
d_CCglo_ChecksumError (0xFF)	Error

Example:

See following page

<u>Example:</u>

The checksum routine needed is called up within the function described above.

```
UBYTE FUN_CCvalaudi_CalcCsumRamDownload_UB( \
                   ULONG  t_StartAddress_UL \
                   ULONG  t_StopAddress_UL,    \
                   ULONG  t_ChecksumAddress_UL )
......
  /*-----------------------------------------------------------------*/
  /* Init the calculation                                           */
  /*-----------------------------------------------------------------*/
......

  t_Result_UB = FUN_CCVhnd_InitCtrlST_UB \
         (&CCValAudi_WorkingInfoInt_ST, FUN_CCval_LinAdd8Acc16Bit_UB);

  /*-----------------------------------------------------------------*/
  /* Call Checksum Routine                                           */
  /*-----------------------------------------------------------------*/
  if(d_CCglo_Ready == t_Result_UB)
  {
    t_Result_UB = FUN_CCVhnd_CalcChecksum_UB();

    /* Mapping of return values for compatibility */
    if ((d_CCVhnd_CheckNOK == (d_CCVhnd_CheckNOK & t_Result_UB)) ||
      (d_CCVhnd_Error == (d_CCVhnd_Error & t_Result_UB)))
    {
      t_Result_UB = d_CCglo_ChecksumError;
      t_FlashCheckState_UB = d_CCglo_Error;
    }
    else
    {
      t_Result_UB = d_CCglo_NoError ;
      t_FlashCheckState_UB = d_CCglo_Busy;
    }
  }
  else
  {
    t_Result_UB = d_CCglo_Error;
    t_FlashCheckState_UB = d_CCglo_Error;
  }
  return  t_Result_UB;

} /*-+- END OF FUNCTION FUN_CCvalAudi_CalcCsumRamDownload_UB ++-+-+-*/
```

6.4.3 Test after EEPROM programming

For some projects, the configuration of the EEPROM is possible with a programming procedure. For a check of correct programming of the EEPROM memory, the following two functions are used:

```
FUN_CCval_InitCalcCsumEpromDownload_UB
```

is used to initialise the checksum calculation. The function

```
FUN_CCvalaudi_CalcCsumEepromInternal_UB
```

is called cyclically until the checksum calculation is finished. An interruption of the calculation is necessary to allow communication with the tester to be maintained when large EEPROM blocks must be checked.

6.4.3.1 Initialisation

Function description

Prototype:

```
UBYTE FUN_CCvalaudi_InitCalcCsumEepromDownload_UB(
          ULONG  t_StartAddress_UL,
          ULONG  t_StopAddress_UL,
          ULONG  t_ChecksumAddress_UL )
```

Parameter:

`ULONG t_StartAddress_UL`	Start address of checksum test
`ULONG t_StopAddress_UL`	Stop address of checksum test
`ULONG t_ChecksumAddress_UL`	Address of checksum

Return value:

d_CCglo_NoError	(0x00)	No error
d_CCglo_Error	(0xFF)	Error

Example:

See following page

Example:

```
UBYTE FUN_CCvalaudi_InitCalcCsumEepromDownload_UB ( \
                                 ULONG  t_StartAddress_UL, \
                                 ULONG  t_StopAddress_UL,    \
                                 ULONG  t_ChecksumAddress_UL )
{
  UBYTE t_Result_UB;

  /*----------------------------------------------------------------*/
  /* Init the calculation                                           */
  /*----------------------------------------------------------------*/
.....

  /*----------------------------------------------------------------*/
  /* Init local EEPROM struct                                       */
  /*----------------------------------------------------------------*/
.....

  /*----------------------------------------------------------------*/
  /* Init EEPROM error variable                                     */
  /*----------------------------------------------------------------*/
  t_NVMResult_UB = d_CCglo_NoError;

  t_Result_UB = FUN_CCVhnd_InitCtrlST_UB \
        (&CCValAudi_WorkingInfoInt_ST,\
          FUN_CCvalaudi_CalcCsumEepromInternal_UB);


  if(d_CCglo_Ready == t_Result_UB)
  {
    t_Result_UB = d_CCglo_NoError ;
    t_FlashCheckState_UB = d_CCglo_Busy;
  }
  else
  {
    t_Result_UB = d_CCglo_Error;
    t_FlashCheckState_UB = d_CCglo_Error;
  }

  return (t_Result_UB);

} /*- END OF FUNCTION FUN_CCvalaudi_InitCalcCsumEepromDownload_UB +-*/
```

6.4.3.2 Checksum calculation

Function description

Prototype:

```
UBYTE
FUN_CCval_CalcChecksumEepromDownload_UB (void )
```

Parameter:

```
none
```

Return value:

d_CCglo_Busy	(0x40)	The checksum calculation is not finished.
d_CCglo_Ready	(0x80)	The checksum calculation is finished, correct checksum.
d_CCglo_Error	(0x20)	The checksum calculation is finished, faulty checksum.

Example:

See following page

Example:

```
UBYTE FUN_CCvalaudi_CalcCsumEepromInternal_UB ( \
               VALCONTROL_ST  *t_Parameters_PST )
{

  /* structure for the eeprom access     */
  NVMACCESS_ST t_NVMDirectAccess_ST;
...
 /*==============================================================*/
 /* -1- check result                                             */
 /*==============================================================*/
  if ( d_CCVhnd_CheckForResult == \
     (d_CCVhnd_CheckForResult & t_Parameters_PST->Control_UB) )
  {
    CCValAudi_Eeprom_ST.Control_UB |= d_CCVhnd_CheckForResult;

    t_Result_UB = FUN_CCval_CRC32_UB (&CCValAudi_Eeprom_ST);

    /*===========================================================*/
    /* -2- NVM access was not successful                         */
    /*===========================================================*/
    if (d_CCglo_NoError != t_NVMResult_UB)
    {
      /*---------------------------------------------------------*/
      /* set error as result                                     */
      /*---------------------------------------------------------*/
      t_Result_UB = (d_CCVhnd_CheckNOK | d_CCVhnd_Ready);
    }
  }
  /*=============================================================*/
  /* -1- do calculation                                          */
  /*=============================================================*/
  else
  {
......
      /*-----------------------------------------------------------*/
      /* call the eeprom access                                    */
      /*-----------------------------------------------------------*/
.....
      t_Result_UB = FUN_CCval_CRC32_UB (&CCValAudi_Eeprom_ST);

      /*----------------------------------------------------------*/
      /* Increment Address                                        */
      /*----------------------------------------------------------*/
      t_tempStartAddress_UL += (ULONG) t_BytesToRead_UW;
```

6.4.3.3 Validation map

For a checksum calculation after an EEPROM download, the following entries (marked in red) in 'VALIDATIONMAP' are necessary:

```
/*--------------------------------------------------------------------------*/
/* 05 - Eeprom - EEPROM 01                                                  */
/*--------------------------------------------------------------------------*/
{
/* ----------------------------------++----------------------------------*/
/* 2 Byte - CheckState:              */  (
/*   Bit 15     - Entry type         */  (UWORD) d_CCglo_NormalEntry       |
/*   Bit 14     - En/Dis Check       */  (UWORD) d_CCglo_CheckDisabled     |
/*   Bit 13/12 - Memory Type         */  (UWORD) d_CCglo_EepromMemory      |
/* ----------------------------------++----------------------------------*/
/*   Memorytype: RAM                                                    */
/*   Bit 11     - Init/Don't Init RAM */ (UWORD) d_CCglo_NotUsed       |
/* ----------------------------------++----------------------------------*/
/*   Memorytype: Flash               */
/*   Bit 03     - Do QuickCheck ?    */  (UWORD) d_CCglo_NotUsed       |
/*   Bit 02-00 - QuickCheckLength    */  (UWORD) d_CCglo_NotUsed       ),
/* ----------------------------------++----------------------------------*/
/* 2 Byte - Error Message entry      */  (UWORD) d_CCglo_NotUsed,
/* ----------------------------------++----------------------------------*/
/* 4 Byte - Startaddress             */  (ULONG) 0x00000100,
/* 4 Byte - StopAddress              */  (ULONG) 0x00000119,
/* 4 Byte - Checksum address         */  (ULONG) d_CCglo_NotUsed,
/* 4 Byte - Quick check address      */  (ULONG) d_CCglo_NotUsed,
/* ----------------------------------++----------------------------------*/
/* 2 Byte - Check Function address   */ d_CCglo_NotUsed,
/* ----------------------------------++----------------------------------*/
/* 2 Byte - Memory Index             */  (UWORD) 0x0004,
/* ----------------------------------++----------------------------------*/
/* 1 Byte - corresp. DownloadMap name*/  (UBYTE) d_CCglo_NoEntry,
/* ----------------------------------++----------------------------------*/
/* 1 Byte - Reserved                 */  (UBYTE) d_CCglo_NotUsed
/* ----------------------------------++----------------------------------*/
}
```

d_CCglo_NormalEntry identifies a normal table entry. A memory area in EEPROM is identified by means of d_CCglo_EepromMemory. This area is described by the entries StartAddress and StopAddress (addresses 0100_{hex} – 0119_{hex}). The memory area can also be addressed via Memory Index (0004_{hex}).

6.5 Prototype of test routine

Comprehensive test routines are available for both the checksum calculation and the RAM test. The offering of these routines can be expanded any time, but the following condition must be observed:

When the function requires less than 1ms to be executed, it must only be called up once. However, if it requires longer, the function must be capable of saving a calculated interim result and of continuing to work with the interim result when called up again. The useable structure for this purpose:

```
typedef struct _VALCONTROL_ST
{
  /* Start address of check area */
  VHND_PTR_BUF_UBYTE StartAddress_PBUF;
  /* End address of check area */
  VHND_PTR_BUF_UBYTE EndAddress_PBUF;
  /* Checksum address */
  VHND_PTR_BUF_UBYTE ChecksumAddress_PBUF;
  /* Workingbuffer: Initial Value/Working Buffer */
  ULONG WorkingBuffer_UL;
  /* For equal Validation Map entry */
  UWORD CheckCondition_UW;
  /* Flags for controlling the validation handler */
  UBYTE Control_UB;
} VALCONTROL_ST;
```

has the element "WorkingBuffer_UL" for this purpose. The function must correspond to the following prototype:

```
UBYTE NameOfFunction_UB (VALCONTROL_ST *Parameters );
```

The following values are defined for the return values:

d_CCVhnd_CheckOK	Calculated checksum equals the defined checksum.
d_CCVhnd_CheckNOK	Calculated checksum does not equal the defined checksum.

An example application:

```
UBYTE FUN_CCval_LinAdd8Acc16Bit_UB (VALCONTROL_ST  *Parameters )
{
  UBYTE          t_Result_UB;               /* Result of check */
  VHND_PTR_BUF_UBYTE t_TempAddress_PUB;     /* temp address */
  VHND_PTR_BUF_UWORD t_BufferAddress_PUW;   /* temp address */

  t_BufferAddress_PUW = \
               (VHND_PTR_BUF_UWORD)&Parameters->WorkingBuffer_UL;

  /*-----------------------------------------------------------------*/
  /* Init the calculation                                            */
  /*-----------------------------------------------------------------*/
  t_Result_UB         = d_CCVhnd_Ready;
  t_TempAddress_PUB = Parameters->StartAddress_PBUF;

  /*=================================================================*/
  /* -1- check result                                                */
  /*=================================================================*/
  if ( d_CCVhnd_CheckForResult == \
    (d_CCVhnd_CheckForResult & Parameters->Control_UB))
  {
    /* Checksum correct? */
    if(*t_BufferAddress_PUW != \
        *(VHND_PTR_BUF_UWORD)(Parameters->ChecksumAddress_PBUF))
    {
       t_Result_UB = (d_CCVhnd_CheckNOK | d_CCVhnd_Ready);
    } else {
       t_Result_UB = (d_CCVhnd_CheckOK | d_CCVhnd_Ready);
    }
  }
  else
  {
    /*=================================================================*/
    /* -1- Calculate Checksum                               <<<      */
    /*=================================================================*/
    while (t_TempAddress_PUB <= Parameters->EndAddress_PBUF)
    {
      /* Do calculation here and store value in working buffer */
      /* Increment Address                                      */
      *t_BufferAddress_PUW += (UWORD)(*t_TempAddress_PUB & 0x00FFu);

      t_TempAddress_PUB++;
    }
    t_Result_UB = (d_CCVhnd_Ready);
  }
  return (t_Result_UB);

} /*-+- END OF FUNCTION 'ADLATUS_FUN_Val_LinAdd8Acc8Bit_UB' -+-+-+-*/
```

7 Communication

7.1 Overview

A communication protocol stack is implemented in SMART ADLATUS® for communication over one of the microcontroller hardware interfaces. Its design is based on the ISO-OSI reference model. The following figure shows the relationship:

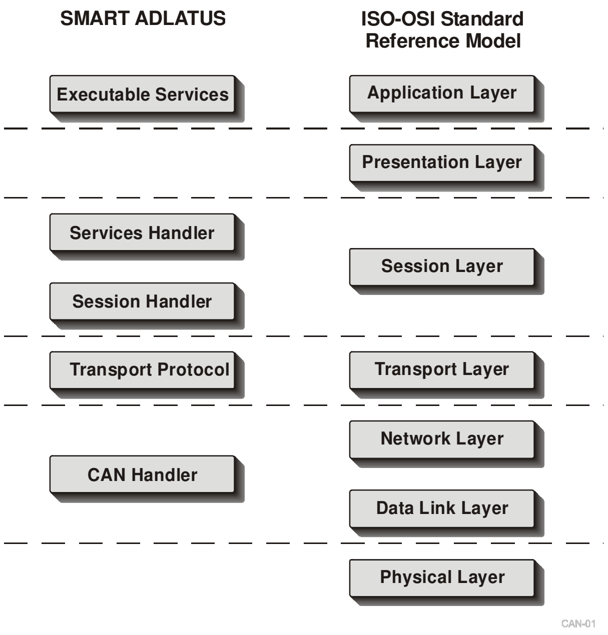


Figure 7-1 SMART communication protocol stack

The functionality of this communication protocol stack depends on the respective communications interface of the hardware on which it is based. Moreover, several parameters are absolutely necessary for communication. These are generally dependent upon the individual customer requirements. The user has the possibility of setting these communication parameters at a central position in accordance with the requirements. These settings are described in detail in the following.

7.2 CAN handler

The `ADLATUS_CANHandler.c` module implements the standardised driver for the microcontroller-specific CAN communication interface. This driver module provides the send and receive functionality as well as the corresponding initialisation routines for the hardware.

All settings for the CAN Handler are made at a central point in the module `ADLATUS_ConfigProject.c` according to the individual requirements. The following figure shows the relationship:

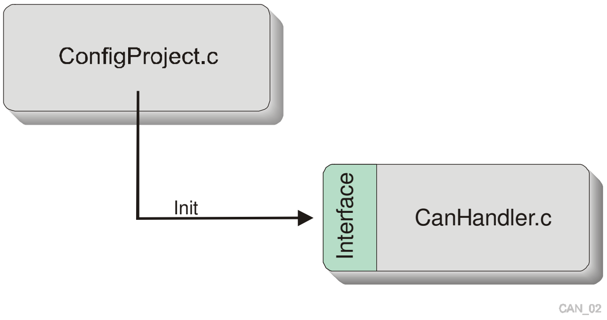


Figure 7-2 Module structure for the CAN initialisation

All data fields and their possible contents needed for the project-specific configuration are described in the following.

If another communication interface is used, the module `ADLATUS_CANHandler.c` can be replaced by the appropriate driver module.

7.2.1 CAN node selection

The selection of the CAN node to be used for communication takes place in the area

Module:	ADLATUS_ConfigProject.c
Application area:	Can.01 – CAN Controller Selection

The corresponding value must be entered in the constant `c_CCconprj_CanController_UW`.

Depending on the controller designation and the number of available CAN nodes on a controller, the following settings are possible:

#define	Comment
d_CCcan_NodeA	Selection of the designated CAN controller on the building block
d_CCcan_NodeB	Selection of the designated CAN controller on the building block
d_CCcan_NodeC	Selection of the designated CAN controller on the building block
d_CCcan_Node1	Selection of the designated CAN controller on the building block
d_CCcan_Node2	Selection of the designated CAN controller on the building block
d_CCcan_Node3	Selection of the designated CAN controller on the building block

SMART

7.2.2 Standard baud rate

The standard baud rate (default baud rate) is set in the following area:

Module:	ADLATUS_ConfigProject.c
Application area:	Can.02 – Default CAN Baudrate

The corresponding baud rate must thereby be entered in the constant field `c_CCconprj_DefaultCanBaudrate_UB`.

The following settings are possible:

#define	Comment
d_CCcan_1000000Bd	Baud rate: 1000 KBits / s
d_CCcan_500000Bd	Baud rate: 500 KBits / s
d_CCcan_333000Bd	Baud rate: 333 KBits / s
d_CCcan_250000Bd	Baud rate: 250 KBits / s
d_CCcan_200000Bd	Baud rate: 200 KBits / s
d_CCcan_125000Bd	Baud rate: 125 KBits / s
d_CCcan_100000Bd	Baud rate: 100 KBits / s
d_CCcan_50000Bd	Baud rate: 50 KBits / s

The baud rate actually used for a programming sequence ist project-specific. Thus it may occur that a rare baud rate does not appear in the list above. The values are defined in the file `ADLATUS_Can Handler_cdf.h.`

7.2.3 CAN message objects (mailbox) configuration

All CAN interface send and receive objects (mailboxes) are configured at the following designated position:

Module:	*ADLATUS_ConfigProject.c*
Application area:	Can.03 - CAN Message Object Configuration

The relevant data must thereby be entered in a structure of type `CANMSGOBJCONFIG_ST` (`ADLATUS_Types_tdf.h` → `Type definition CAN.01`). The layout of this structure is shown in the following:

Format	Designation	Description
1 byte	MsgObjectNbr	Number of the element to be configured (mailbox number)
1 byte	MsgObjState	Element (mailbox) is activated/deactivated
1 byte	MsgObjDirection	Sender or receiver
1 byte	ID Length	11-bit ID, 29-bit ID
4 byte	CanIdentifier	Physical identifier on CAN
4 byte	Acceptance Mask	Acceptance mask for the object

Table 7-1 Data structure CANMSGOBJCONFIG_ST

The following settings are available:

Designation	Possible configuration
MsgObjectNbr:	0x00 ...(Number of existing objects or mailboxes – 1)
MsgObjState:	d_CCcan_Activate d_CCcan_Deactivate
MsgObjectDirection:	d_CCcan_Receiver d_CCcan_Transmitter
ID length:	d_CCcan_11BitID d_CCcan_29BitID
CAN Identifier:	0x00000000 ... 0x1FFFFFFF Attention: Observe 11-bit / 29-bit
Acceptance Mask:	0x00000000 ... 0xFFFFFFFF Attention: Observe controller handbook

⚠ Elements that are not needed should not be entered for reasons of memory capacity.

Example:

Mailbox 0 receives all messages with the identifier 0x00000123 (29-bit ID).

```
/* ========================================================== */
/* -00-  CAN Message Object                              */
/* ========================================================== */
{
  /* MsgObjectNbr    */ 0x00,
  /* MsgObjState     */ d_CCcan_Activate,
  /* MsgObjDirection */ d_CCcan_Receive,
  /* IDLength_UB     */ d_CCcan_29BitID,
  /* CanIdentifier   */ 0x00000123,
  /* Acceptance Mask */ 0x1FFFFFFF,
}, /* - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

The last element of the structure is concluded with

```
/* ========================================================== */
/* -FF-  CAN Message Object                              */
/* ========================================================== */
{
  /* MsgObjectNbr    */ 0xFF,
  /* MsgObjState     */ d_CCcan_Deactivate,
  /* MsgObjDirection */ d_CCcan_Transmit,
  /* IDLength_UB     */ d_CCcan_29BitID,
  /* CanIdentifier   */ 0xFFFFFFFF,
  /* Acceptance Mask */ 0x1FFFFFFF,
} /* - - - - - - - - - - - - - - - - - - - - - - - - - - - */
```

This entry may not be left out! The MsgObjectNbr 0xFF may not be changed! If additional mailboxes are needed, the structure must be extended by the number of new mailboxes and these configured in accordance with the information above.

7.2.4 Channel configuration

A logical channel must be assigned to every CAN mailbox element for further internal processing of the data received. The settings are carried out at the positions designated as follows:

Module:	*ADLATUS_ConfigProject.c*
Application area:	Can.04 - CAN Message Object Channel Configuration

The relevant data must thereby be entered in a structure of type `CANMSGOBJINFO_ST` (`ADLATUS_Types_tdf.h` → `Type definition CAN.02`).

If addressing or function selection of a controller is carried out not only via the identifier, but also via an additional address byte (byte 0 of the working data), this is also entered in the structures given above. Every mailbox can be assigned multiple additional addresses.

The layout of the structure is shown in the following:

Format	Designation	Description
1 byte	MsgObjectNbr	Number of CAN element (mailbox number)
1 byte	Subset + Channel	Channel number and subset
1 byte	Additional Identifier defined	Information as to whether an additional address byte is defined instead of the first data byte.
1 byte	Additional Identifier Byte	Identifier of the additional address byte

Table 7-2 Data structure CANMSGOBJINFO_ST

The following settings are available:

Designation	Possible configuration
MsgObjectNbr:	0x00 ...(No. of objects present – 1)
Subset + Channel:	Subset: 0x00 .. 0xF0 Channel: 0x00 .. 0x0F
Add. ID Byte defined:	d_Cccon_No .. no addit. ID defined d_Cccon_Yes .. addit. ID defined
	d_CCglo_FixedMsgLgnth .. The message length is always 8 bytes.
	d_CCglo_MbSendFromRam .. The corresponding mailbox is used to send a message while erasing.
Additional ID:	0x00 .. 0xFF

Examples:

The example shows the setting for only one receive and send mailbox. No additional address identifiers are used.

```
/*-----------------+------------------+--------------------+----------------*/
/* CanMsgObjectNbr | SubSet + Channel | add ID byte defined | additional ID  */
/*-----------------+------------------+--------------------+----------------*/
{     0x00         , ( 0x00 | 0x00 ) ,   d_CCcon_No       , d_CCcon_NoEntry  },
/*-----------------+------------------+--------------------+----------------*/
{       0x01       , ( 0x01 | 0x00 ) ,    d_CCcon_No      , d_CCcon_NoEntry  },
/*-----------------+------------------+--------------------+----------------*/
/* LAST ENTRY LINE - DON'T CHANGE !!!                                      */
/*-----------------+------------------+--------------------+----------------*/
{ d_CCcon_NoEntry  , d_CCcon_NoEntry ,   d_CCcon_NoEntry  ,
d_CCcon_NoEntry   }
/*-----------------+------------------+--------------------+----------------*/
```

Example:

In the following example there are four mailboxes, two each for send and receive. All messages are marked with an additional address identifier. There are two channels for mailboxes two and three. In the example above, if a 0x01 is received as an address identifier, the sender responds with 0xF1, for 0x05 with 0xF2.

```
/*----------------+-----------------+--------------------+----------------*/
/* CanMsgObjectNbr | SubSet + Channel | add ID byte defined | additional ID  */
/*----------------+-----------------+--------------------+----------------*/
{      0x00     , ( 0x00 | 0x00 )  ,  d_CCcon_Yes      ,  0x32        },
{      0x01     , ( 0x10 | 0x00 )  ,  d_CCcon_Yes      ,  0xF1        },
/*----------------+-----------------+--------------------+----------------*/
{      0x02     , ( 0x00 | 0x01 )  ,  d_CCcon_Yes      ,  0x01        },
{      0x03     , ( 0x10 | 0x01 )  ,  d_CCcon_Yes      ,  0xF1        },
/*----------------+-----------------+--------------------+----------------*/
{      0x02     , ( 0x00 | 0x02 )  ,  d_CCcon_Yes      ,  0x05        },
{      0x03     , ( 0x10 | 0x02 )  ,  d_CCcon_Yes      ,  0xF2        },
/*----------------+-----------------+--------------------+----------------*/
/* LAST ENTRY LINE - DON'T CHANGE !!!                                       */
/*----------------+-----------------+--------------------+----------------*/
{ d_CCcon_NoEntry  , d_CCcon_NoEntry ,  d_CCcon_NoEntry  ,
d_CCcon_NoEntry   }
/*----------------+-----------------+--------------------+----------------*/
```

In each entry, the value `d_CCcon_NoEntry` must be entered in the last line of the table.

When assigning for channel number and subset, it should be noted that all even subset numbers (0x00, 0x20, .. 0xE0) identify a receive channel. All odd channel numbers (0x10, 0x30,..0xF0) identify a send channel. The respective send channel must thereby always be configured with the subset number receive channel + 0x10.

Example:

Receive channel	(
Send channel	(

7.3 Transport protocol

All settings for the transport protocol are made at a central point in the module `ADLATUS_ConfigProject.c` according to the individual requirements. The following figure shows the relationship:

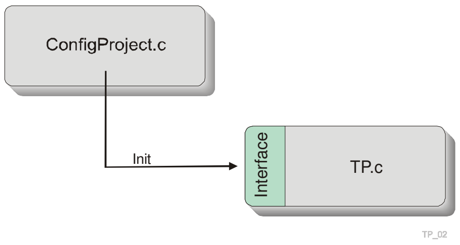


Figure 7-3 Module structure for the TPinitialisation

There are currently two different protocol interpreters available for the transport layer between the CAN and the session layer.

The standard ISO transport layer satisfies the specification:

ISO 14230 - 2

Road vehicles – Diagnostic systems – Keyword Protocol 2000

Status 15.03.99

Implemented. The TP2.0 protocol is based on:

CAN transport protocol TP2.0

Version 1.1

Status 02.02.2001

7.4 Setting times

All transport layer settings are configured at the position designated as follows:

Module:	*ADLATUS_ConfigProject.c*
Application area:	TP.01 – Transport Protocol Configuration

The relevant data must thereby be entered in a structure of type `TPINTERFACE_ST` (`ADLATUS_TP_tdf.h` → `Type definition TP.01`) . The layout of this structure is shown in the following:

The following entries can be found with a standard ISO protocol:

Format	Designation	Description
2 byte	ST min	ST_{min} – time definition Minimum time between two consecutive frames
2 byte	ST max	ST_{max} – time definition: Maximum time between two consecutive frames
2 byte	P2 min	$P2_{min}$ – Time between tester request and ECU response
2 byte	P2 max	$P2_{max}$ – Time between tester request and ECU response
2 byte	P3max	$P3_{max}$ – Time between ECU response and next tester request
1 byte	BSmax	BSmax – Blocksize parameter which is transmitted by the Adlatus in the flow control

P3max is used to recognize a communication interruption. In ADLATUS standalone operation, the system is reset after P3max has elapsed, or branches into application software if it is valid. If a value of “0” is selected for P3max, the timeout is switched off and the system can only be reset via a power-on reset. The timeout is triggered afresh every time a telegram is received.

```
const TPINTERFACE_ST c_CCcon_TpInterface_ST =
{
  /*-------------------------------------------------*/
  /* Transport Protocol Timings                     */
  /*------------++-------------------------------------*/
  /* Name       ||  Value [ms]                     */
  /*------------++-------------------------------------*/
  /* ST min     */ (UWORD)    1,  /* ms */
  /* ST max     */ (UWORD) 1000,  /* ms */
  /* P2 min     */ (UWORD)    2,  /* ms */
  /* P2 max     */ (UWORD) 1000,  /* ms */
  /* P3 max     */ (UWORD) 5000,  /* ms */
  /* BSmax      */ (UBYTE) 0x0F
}; /* -+- END OF TABLE -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*/
```

The timing for the TP2.0 transport protocol can also be configured as follows:

Format	Designation	Description
2 byte	T1	Timeout for acknowledgement telegram
2 byte	T3	Minimum time between two telegrams
2 byte	TReinit	Timeout for missing message traffic.

In standalone operation of the ADLATUS, the system is reset upon the elapsing of TReinit, or branched into valid application software. If a value of “0” is selected for TReinit, the timeout is switched off and the system can only be reset via a power-on reset. The timeout is triggered afresh every time a telegram is received.

The following example shows a possible configuration.

```
const TP20INTERFACE_ST c_CCconprj_Tp20Interface_ST = \
{
  /*---------------------------------------------------*/
  /* Transport Protocol Timings                        */
  /*------------++-------------------------------------*/
  /* Name       ||  Value [ms]                         */
  /*------------++-------------------------------------*/
  /* T1         */ (UWORD)  100,  /* ms */
  /* T3         */ (UWORD)    2,  /* ms */
  /* TReinit    */ (UWORD) 5200,  /* ms */
}; /* -+- END OF TABLE -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-*/
```

8 External watchdog

8.1 Overview

The operation of the external watchdog is implemented in two files. The division is necessary because a part of the watchdog operation takes place in RAM and some compilers/linkers do not enable a division into RAM and flash memory functions by way of "pragma" or other directives within a file.

The functions working in the flash memory are found in the `ADLATUS_ExternalWatchdogFlash.c` module. These are generally functions for the initialisation and copying or for the activation and deactivation of the watchdog. The `ADLATUS_ExternalWatchdog.c` file contains the functions that are copied into RAM and carried out there prior to being called up.

The actual calling up of the external watchdog takes place in the function `FUN_CCwtdog_TriggerExtWatchdog_V`. Depending upon the system status, the function is called up from various sources. The calling up of the function takes place after the elapsing of a 1ms time slice. The triggering point in time is defined by a software timer.

⚠

The functions carried out in RAM may not access the flash memory. This applies for both the usage of constants and functions. Accessing the flash memory can lead to unexpected results when deleting or programming the memory and to permanent damage of the memory. Because access is not always immediately recognised during the critical phases, it may be necessary to check whether unauthorised access has taken place through a review of the assembler code.

If the compiler uses code elements from a library for certain functionalities, and if these elements are used in the watchdog functions, the library function must either be loaded into the RAM and carried out there or its being called up avoided through a different code implementation. Familiar library functions include, for example, mathematical functions, the calling up of a function through a pointer, but also "switch ..case" instructions.

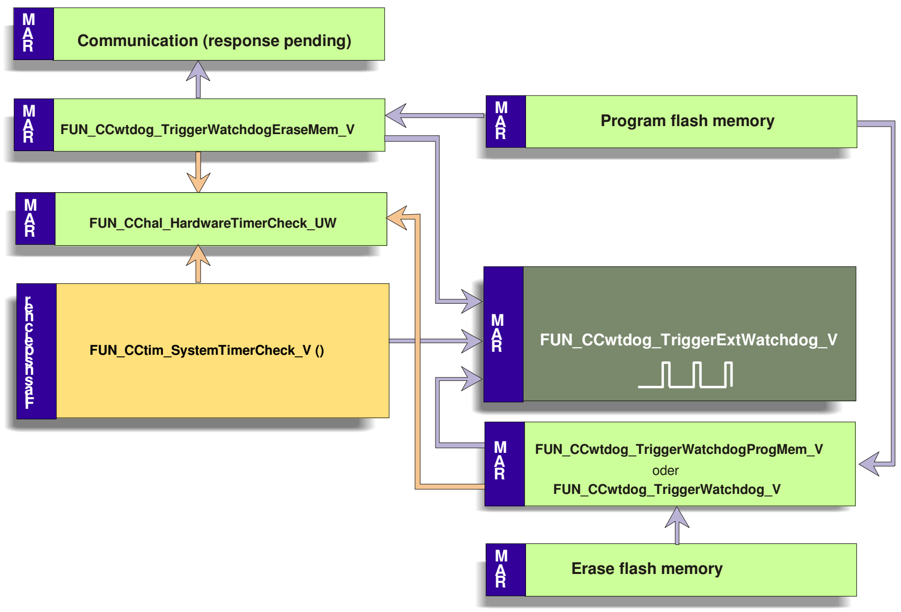

MAR
Communication (response pending)
MAR
FUN_CCwtdog_TriggerWatchdogEraseMem_V
MAR
Program flash memory
MAR
FUN_CChal_HardwareTimerCheck_UW
FUN_CCtim_SystemTimerCheck_V ()
MAR
FUN_CCwtdog_TriggerExtWatchdog_V
MAR
FUN_CCwtdog_TriggerWatchdogProgMem_V
oder
FUN_CCwtdog_TriggerWatchdog_V
MAR
Erase flash memory

Fig. 8-1 Calling up the external watchdog

The illustration above shows which functions may be necessary for calling up the external watchdog. In the case of programming sequences that do not provide for the maintenance of communication while erasing, the communication function and `FUN_CCwtdog_TriggerWatchdogEraseMem_V` are dispensed with. The `FUN_CCwtdog_TriggerWatchdogProgMem_V` function is then called `FUN_CCwtdog_TriggerWatchdog_V` and is called up during erasing and programming.

The `FUN_CChal_HardwareTimerCheck_UW` function provides the time basis for the call-up. Depending upon the system status, the trigger function is addressed either by the `FUN_CCtim_SystemTimerCheck_V` function or from the RAM.

The basic conditions for using the watchdog are very project-specific and the variance of the watchdog concepts enormous. It can therefore happen that some of the functions are dispensed with. However, for a standard project, the ADLATUS requires at least the following functions:

```
FUN_CCwtdog_InitExtWatchdog_V
FUN_CCwtdog_EnableExtWatchdog_V
FUN_CCwtdog_DisableExtWatchdog_V
FUN_CCwtdog_TriggerExtWatchdog_V
```

Other functions are either

```
FUN_CCwtdog_TriggerWatchdog_V
```

or

```
FUN_CCwtdog_InitExtWatchdogIntern_V
FUN_CCwtdog_TriggerWatchdogEraseMem_V
FUN_CCwtdog_TriggerWatchdogProgMem_V
```

The calling up of the functions during the initialisation of the system takes place as follows:

```
1. - FUN_CCwtdog_InitExtWatchdog_V();
2. - FUN_CCwtdog_EnableExtWatchdog_V();
3. - FUN_CCwtdog_DisableExtWatchdog_V();
```

With this call-up sequence, the following conditions can be satisfied with a corresponding implementation:

External watchdog is present / is not present

External watchdog can be switched off / External watchdog cannot be switched off

8.2 Control structure

The control of the watchdog handler takes place through a control structure that has been defined in the `ADLATUS_ExternalWatchdog_tdf.h` file. The elements contained in the structure are not used by any other function of the ADLATUS and can be easily adapted to the conditions of the respective project.

The following table shows the layout of this control structure in its simplest variant:

Format	Designation	Description
4 byte	TXDelayCounter_UL	Software timer for maintaining communication
1 byte	WatchdogState_UB	Internal status register
2 byte	WatchdogTimer_T16	16-bit timer variable

All control variables are described in detail in the following.

8.2.1 TXDelayCounter_UL

The TXDelayCounter_UL variable serves as a software timer for programming sequences for which communication must be maintained during erasing. If the variable reaches the value `d_CCwtdog_TXDelayCounter`, this results in the calling up of the `FUN_CCcan_TxDataFromRam_V` function. The function transmits a message, generally "response pending", via CAN. In the case of programming sequences not prescribing communication while erasing, the variable can be removed from the structure.

8.2.2 Status register

Control of the watchdog handler is carried out via an internal flag register. These flags are correspondingly modified or read out and interpreted by the above functions. The following table shows the layout of the flag register:

Bit	Description
0	**Enable watchdog**
	0 .. Watchdog should not be enabled.
	1 .. Watchdog should be enabled.
1	**Disable Watchdog**
	0 .. Watchdog should not be disabled.
	1 .. Watchdog should be disabled.
2	**Trigger Watchdog**
	0 .. Watchdog must not be triggered.
	1 .. Watchdog must be triggered.
3, 4, 5, 6	**Not used**
	0 .. No effect
	1 .. No effect
7	**Watchdog enabled**
	0 .. Watchdog is not enabled
	1 .. Watchdog is enabled

8.2.3 Watchdog timer

The watchdog timer is a 16-bit variable provided for time window-controlled triggering of the watchdog. The timer is queried and restarted, where applicable, within the routine

```
FUN_CCwtdog_TriggerExtWatchdog_V
```

In the area below,

Type definition:	*ADLATUS_ExternalWatchdogFlash.c*
Type definition:	*ADLATUS_ExternalWatchdog.c*
Application area:	`WDT.01 – Watchdog trigger time`

the value for

```
#define t_WtdogTimerTicks_UW (10) /* ms */
```

must be correspondingly set. This value must be given in 10^{-3} seconds [ms]. In the example above, the watchdog operates according to a 10ms pattern.

8.3 Initialisation

The function initialises the status register and the control structure timer. The initialisation value for the timer is to be set in the same module in the area

Type definition:	*ADLATUS_ExternalWatchdogFlash.c*
Type definition:	*ADLATUS_ExternalWatchdog.c*
Application area:	`WDT.01 – Watchdog trigger time`

(See 8.2.3).

Function description

The function initialises the required structure elements and copies the watchdog functions into the RAM.

Prototype:

```
        void FUN_CCwtdog_InitExtWatchdog_V (void)
```

Parameter:

–

Return value:

–

8.4 Enabling watchdog

Function description

The function sets bit 0 in the status register of the control structure. This signals to the function `FUN_CCwtdog_TriggerExtWatchdog_V` when called up, that the watchdog should be enabled.

Prototype:

```
Void FUN_CCwtdog_EnableExtWatchdog_V (void)
```

Parameter:

–

Return value:

–

Example:

```
void FUN_CCwtdog_EnableExtWatchdog_V (void)
{
  /*------------------------------------------------*/
  /*  Enable the system watchdog                     */
  /*------------------------------------------------*/
  t_CCper_WatchdogCtrl_ST.WatchdogState_UB |= (UBYTE) \
          d_CCglo_FlagEnableExternalWatchdog;
  t_CCper_WatchdogCtrl_ST.WatchdogState_UB &= (UBYTE) \
          ~d_CCglo_FlagDisableExternalWatchdog;

  return;

} /* END OF FUNCTION FUN_CCwtdog_EnableExtWatchdog_V */
```

8.5 Disabling watchdog

Function description

The function sets bit 1 to one and resets bit 0 in the status register of the control structure. In this way the function `FUN_CCwtdog_TriggerExtWatchdog_V` is signalled when called up that it should be deactivated as a watchdog.

Prototype:

```
void FUN_CCwtdog_DisableExtWatchdog_V (void)
```

Parameter:

–

Return value:

–

Example:

```
void FUN_CCwtdog_DisableExtWatchdog_V (void)
{
  /*-------------------------------------------------*/
  /*  Disable the system watchdog                    */
  /*-------------------------------------------------*/
  t_CCper_WatchdogCtrl_ST.WatchdogState_UB |= (UBYTE) \
                        d_CCglo_FlagDisableExternalWatchdog;
  t_CCper_WatchdogCtrl_ST.WatchdogState_UB &= (UBYTE) \
                        ~d_CCglo_FlagEnableExternalWatchdog;
  t_CCper_WatchdogCtrl_ST.WatchdogState_UB &= (UBYTE) \
                       ~d_CCglo_FlagTriggerExternalWatchdog;

  return;

} /* END OF FUN 'FUN_CCwtdog_DisableExtWatchdog_V ' +-+-*/
```

8.6 Querying watchdog status

Function description

The status of the watchdog (enabled / disabled → Bit7) is read out by the following defined function.

Prototype:

UBYTE FUN_CCwtdog_GetExtWatchdogState_UB (void)

Parameter:

–

Return value:

`1Byte`	`WatchdogState_UB`	Contents of status register

Example:

```
UBYTE FUN_CCwtdog_GetExtWatchdogState_UB (void)
{
  return (t_CCper_WatchdogCtrl_ST.WatchdogState_UB & \
                                d_CCglo_StateExternalWatchdogEnabled);
} /*-+- END OF 'FUN_CCwtdog_GetExternalWatchdogState_UB' +-+*/
```

8.7 Triggering watchdog

Function description

The function triggers the external watchdog building block. This function is called up periodically to this purpose. The routine first decrements the timer in the control structure and then checks whether this has now elapsed. If this is not the case, the routine is terminated. If the timer has elapsed, then the respective action (enable, disable, trigger) is executed according to the contents of the status register.

Prototype:

void FUN_CCwtdog_TriggerExtWatchdog_V (
    UWORD t_WtdogTimerTicks_UW)

Parameter:

`UWORD t_WtdogTimerTicks_UW`	Timer ticks [ms] since last call-up.

Return value:

–	-

Example:

```
void FUN_CCwtdog_TriggerExtWatchdog_V (void)
{
  /* -1- Watchdogtimer not elapsed                          */
  if (t_CCper_WatchdogCtrl_ST.WatchdogTimer_T16 != 0x0000)
  {
    /* Decrement Watchdog Timer                             */
    t_CCper_WatchdogCtrl_ST.WatchdogTimer_T16 --;
    /* -2- Watchdogtimer elapsed                            */
    if (t_CCper_WatchdogCtrl_ST.WatchdogTimer_T16 == 0x0000)
    {
      t_CCper_WatchdogCtrl_ST.WatchdogTimer_T16 = \
                               d_CCwtdog_InitExternalWatchdogTime;
```

```
        /* -3- Select activity                                  */
        switch (t_CCper_WatchdogCtrl_ST.WatchdogState_UB & \
                                   d_CCglo_ExternalWatchdogFlagMask)
        {
            /* ENABLE WATCHDOG                                  */
            case d_CCglo_FlagEnableExternalWatchdog:
             ... fill in your code ...
             /* Set watchdog state information                  */
             t_CCper_WatchdogCtrl_ST.WatchdogState_UB |= \
                  (UBYTE) d_CCglo_StateExternalWatchdogEnabled;
             break;

           /* TRIGGER WATCHDOG                                  */
           case d_CCglo_FlagTriggerExternalWatchdog:
              ...fill in your code ...
              break;

           /* DISABLE WATCHDOG                                  */
           case d_CCglo_FlagDisableExternalWatchdog:
...           fill in your code ...
              /* Reset watchdog state information               */
              t_CCper_WatchdogCtrl_ST.WatchdogState_UB &= \
                  (UBYTE) ~d_CCglo_StateExternalWatchdogEnabled;
              break;
            /* DEFAULT                                         */
            default:
              break;
        } /* ---3- END OF if(..) -------------------------*/
      } /* -----2- END OF if(..) -------------------------*/
    } /* -------1- END OF if(..) -------------------------*/

    return;

  } /*-+- END OF 'FUN_CCwtdog_TriggerExtWatchdog_V' ++-+-+-+*/
```

⚠ If no timer is used, the timer must be initialised with 1. Timer value = 0 deactivates the entire watchdog functionality.

8.8 Calling up watchdog from RAM

Function description

This is an example for a function without communication during erasing. The function checks the elapsed time and, if appropriate, calls up the watchdog function. It runs in RAM and is called up during erasing or programming.

Prototype:

void FUN_CCwtdog_TriggerWatchdog_V (void)

Parameter:

–

Return value:

–

Example:

```
void FUN_CCwtdog_TriggerWatchdogProgMem_V (void )
{
  UWORD t_Elapsedmsec_UW;
  /*--------------------------------------------------------*/
  /* read out the elapsed 1ms hardware timer ticks          */
  /*--------------------------------------------------------*/
  t_Elapsedmsec_UW = FUN_CChal_HardwareTimerCheck_UW();
  /*=========================================================*/
  /* -1- 1ms is elapsed                                      */
  /*=========================================================*/
  if ( 0x0000u != t_Elapsedmsec_UW )
  {
    /*------------------------------------------------------*/
    /* trigger the watchdog                                 */
    /*------------------------------------------------------*/
    FUN_CCwtdog_TriggerExtWatchdog_V ( t_Elapsedmsec_UW );
  }
  return;

} /*- END OF FUNCTION 'FUN_CCwtdog_TriggerWatchdogProgMem_V'*/
```

9 Application SW Jump

9.1 Overview

At a certain time, the ADLATUS® decides whether the programmed application software will be executed or to remain in ADLATUS®. If all criteria for a call-up have been fulfilled, the system branches into the application. The jump address of the application must be known to ADLATUS®. This jump address can be configured by the user according to the project requirements. This procedure is described in the following section.

9.2 Jump address

The jump address in the application software is established in the module

Module:	ADLATUS_Adr_Info.h
Application area:	ASW.01 - ASW entry address

A `#define` d_CCadrinfo_AswEntryAddress is implemented here.

Example:

Jump into application at address 10000_{hex}.

```
/*--------------------------------------------------*/
/*      Application - INTERFACE                     */
/*      -----------------------------------         */
/*      FSW entry address                           */
/*--------------------------------------------------*/
#define d_CCfswint_FswEntryAddress 0x010000
```

However, this procedure cannot always be used. For some projects, the compiler or the microcontroller architecture demands the direct encoding of the address within an assembler sequence. For such projects, one finds the appropriate sequence in the file:

Module:	ADLATUS_ApplicationInterface.c
Application area:	ASW.01 - ASW entry address

Example:

Jump into the application via the address found in the second interrupt vector table (that of the application) in the address `FFBFFC`$_{hex}$.

```
void FUN_CCapplint_JumpToFSW_V (void)
{
  /*-------------------------------------------------------------*/
  /* Jump to the address pointered by the second int-vect-table    */
  /*-------------------------------------------------------------*/

  asm("MOV.W #0x00,A0;");         /* offset = 0*/
  asm("JMPI.A 0xFFBFFC:24[A0]");  /* jump to FSW, indirect jump */

} /*-+- END OF FUNCTION 'FUN_CCapplint_JumpToFSW_V'   -+-+-+-+-+-+-*/
```

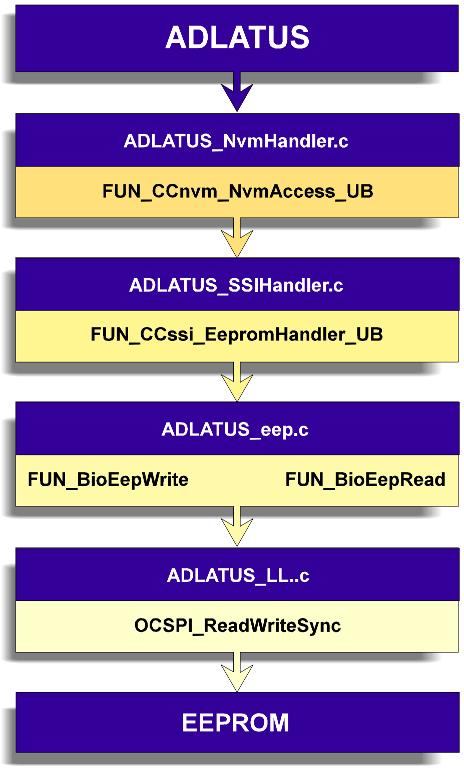
10 NVM handler

10.1 Overview

Access to the EEPROM takes place over several levels. Each level has its own task, is encapsulated in itself and only calls up the level located directly beneath it. The functions in the ADLATUS_NVMHandler.c file serve as the call-up interface with the other ADLATUS functions. The translation of the write or read queries for the subordinate SSI_Handler (SSI stands for "Serial Storage Interface") takes place there. Beneath this is the level that processes EEPROM-specific properties such as the address length, or the composition of the commands. The lowest level is the module that directly accesses the NVM, which contains all hardware-dependent low level functions. For reasons of memory capacity, for some projects the two lowest levels are compiled in one file. The name assignment for the lowest level need not be the same in all projects.

In the levels beneath the ADLATUS_NvmHandler, it may be necessary to carry out project-specific modifications.

Figure 10-1 Level model for NVM handler



10.2 Configuration of the NVM entries

The information for access to an external EEPROM is configured at the following position.

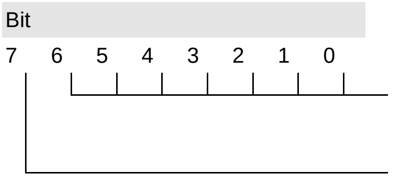
Type definition:	**ADLATUS_ConfigProject.c**
Application area:	**NVM.01 – Address Table**

The elements of the data structure have the following meaning:

Format	Designation	Description
1 byte	Name_UB	Designation of stored data
1 byte	StorageType_UB	Storage format of data
2 byte	DataLength_UW	User data length
2 byte	TotalLength_UW	Actual data length (incl. all extra information such as checksum, etc.)
2 byte	Address_UW	Address of data in EEPROM

Each data record defined for the NVM has a "handle" composed of the access mode (read/write) and an ordinal number. The structure element for this purpose is "Name_UB".

Layout of the transfer parameter t_Handle_UB:



Description

Data Name

x .. Name ofthe data as a binary number.

Access

0 .. Read

1 .. Write

The ADLATUS is preconfigured for several NVM storage structures. The layout of the storage structure is defined in the element "StorageType_UB":

StorageType_UB	
#define	Description
d_CCglo_SingleBuffering	The user data is stored once in EEPROM.
d_CCglo_DoubleBuffering	The user data is stored twice in EEPROM.
d_CCglo_TribbleBuffering	The data is stored three times in EEPROM.

d_CCglo_WithCheckSum	The user data is stored in EEPROM with a checksum.

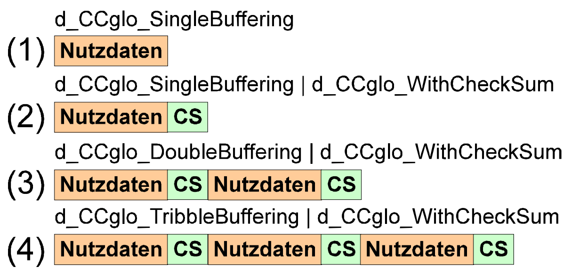

d_CCglo_SingleBuffering
(1) Nutzdaten
d_CCglo_SingleBuffering | d_CCglo_WithCheckSum
(2) Nutzdaten CS
d_CCglo_DoubleBuffering | d_CCglo_WithCheckSum
(3) Nutzdaten CS Nutzdaten CS
d_CCglo_TribbleBuffering | d_CCglo_WithCheckSum
(4) Nutzdaten CS Nutzdaten CS Nutzdaten CS

Figure 10-2 Layout of several storage formats

Only the first procedure is implemented in ADLATUS. In order to implement a different procedure, the function must be recoded in the ADLATUS_SSHandler.c file.

A possible procedure for writing data with procedure three would be the following:

Erase checksum one – Write the first data record – Write checksum one – Erase checksum two – Write data record two – Write checksum two.

Example:

The following example shows the layout of an entry using a fictitious hardware number at address 200$_{hex}$ in the EEPROM as an example. The data length is 5 bytes, the storage format is double with checksum. The total data length in EEPROM is 12 bytes (2 x (5 + 1).

```
/*--------------------------------------------------------*/
/* 00 - Hardware Number (for 0x1A 0x91)                  */
/*--------------++---------------------------------------*/
{
/* Name:          */ (UBYTE) d_CCnvmXXX_HWNumber,
/* StorageType:   */ (UBYTE) ( d_CCglo_doubleBuffering |
                               d_CCglo_WithCheckSum    ),
/* Data Length:   */ (UWORD) 0x05, /*  5 Byte */
/* Real Size      */ (UWORD) 0x0C, /* 12 Byte */
/* Address:       */ (UWORD) 0x200
}, /*-----------++---------------------------------------*/
/*-------------------------------------------------------*/
/* 01 – Last Entry (DON’T CHANGE)                        */
/*--------------++---------------------------------------*/
{
/* Name:          */ (UBYTE) (d_CCglo_NoEntry & 0x7F),
/* StorageType:   */ (UBYTE) (d_CCglo_NoEntry & 0x7F),
/* Data Length:   */ (UWORD) (d_CCglo_NoEntry & 0x7F),
/* Real Size      */ (UWORD) (d_CCglo_NoEntry & 0x7F),
/* Address:       */ (UWORD) (d_CCglo_NoEntry & 0x7F)
}, /*-----------++---------------------------------------*/
```

⚠ **The last entry must be described with the values `(d_CCglo_NoEntry & 0x7F).`**

10.3 Interface to EEPROM handler

The interface between the NVM handler and an EEPROM handler for a special EEPROM consists of the function call-up of the routine:

FUN_CCnvm_NvmAccess_UB in the file ADLATUS_NvmHandler.c

This function contains the downstream processing of the EEPROM access. All relevant data for the access is stored dynamically in a runtime structure and thus made available to the routine.

Function description

Prototype:

```
        UBYTE FUN_CCnvm_NvmAccess_UB (
                                UBYTE           t_Handle_UB,
                                PTR_BUFFER_UB   t_Buffer_PBUF   )
```

Parameter:

`UBYTE t_Handle_UB`	Handle for read/write access and ordinal number
`PTR_BUFFER_UB t_Buffer_PBUF`	Pointer to the data buffer into which the data is to written or from which the data is to be read

Return value:

`UBYTE t_Result_UB`	`0x00` .. No error `0xFF` .. Error

The function `FUN_CCnvm_NvmAccess_UB` writes the structure RunTimeEntry_ST with the currently relevant data for the EEPROM access at runtime. The function `FUN_CCssi_EepromHandler_UB` is then called up in the file `ADLATUS_SSIHandler.c`. The following figure shows the relationship:

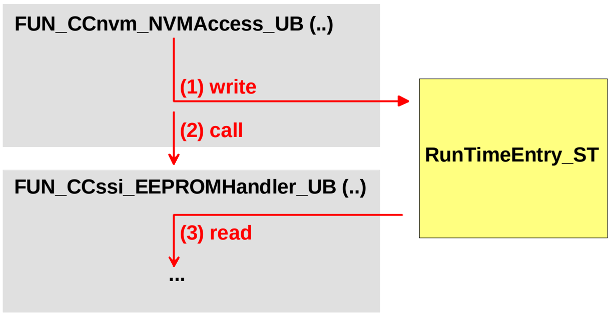


Figure 10-3 Structure of the interface to the EEPROM handler

The RunTimeEntry structure thereby has the following layout:

Format	Designation	Description
1 byte	**Access_UB**	**Designation of stored data**
1 byte	**StorageType_UB**	**Storage format of data**
2 byte	**DataLength_UW**	**User data length**
2 byte	**TotalLength_UW**	**Actual data length (incl. all extra information such as checksum, etc.)**
2 byte	**Address_UW**	**Address of data in EEPROM**
2/3/4 byte[1]	**Buffer_PBUF**	**Pointer to the write/read buffer**

1) depending upon the address spectrum of the microcontroller

A pointer to the structure described above is transferred to the corresponding EEPROM routine as a parameter.

Example:

```
/*---------------------------------------------------------------*/
/* Function FUN_CCnvm_NvmAccess_UB                               */
/*---------------------------------------------------------------*/
UBYTE FUN_CCnvm_NvmAccess_UB ( UBYTE           t_Handle_UB,  \
                               PTR_BUFFER_UB    t_Buffer_PBUF  )
{
  UBYTE t_Result_UB = d_CCglo_NoError;

  /*-------------------------------------------------------------*/
  /* Scan the NVM memory map for a correct entry                 */
  /*-------------------------------------------------------------*/
  t_Index_UB = FUN_CCnvm_ScanNvmMemoryTable_UB (t_Handle_UB);

  .......

  /*=============================================================*/
  /* -1- Correct entry found                                      */
  /*=============================================================*/
  else
  {
    /*-------------------------------------------------------------*/
    /* copy all access data to runtime entry structure            */
    /*-------------------------------------------------------------*/
    .....
    /*--------------------------------------------------------------*/
    /* Call the specific EEPROM Handler                             */
    /*--------------------------------------------------------------*/
    ...

    t_Result_UB = FUN_CCssi_EepromHandler_UB ( \
                 (NVMRUNTIMEENTRY_ST*) &t_CCNvm_Runtimeentry_ST );
  return (t_Result_UB);

} /*-+- END OF FUNCTION FUN_CCnvm_NVMAccess_UB +-+-+-+-+-+-+-+-+-+*/
```

The user need not normally carry out project-specific modifications in the software in the file `ADLATUS_NvmHandler.c`.

10.4 EERPOM initialisation

In the function FUN_CCnvm_InitNvmHandler_V of the file ADLATUS_ NvmHandler.c it is possible to use a "#Defines" to activate the basic initialisation of the EEPROM. This functionality is exclusively conceived of for a quick start for Development and should be removed from the source code prior to series production.

Attention:

Some flash procedures assume the presence of an initialised EEPROM! Flash interruptions result from a non-initialised EEPROM.

10.5 Low-level functions

Because the layout of the low-level functions for NVM access depends upon many factors (project, microcontroller, EEPROM, etc) and can therefore differ significantly, references to the sources can only be made here. However, a few points still need to be mentioned:

- The watchdogs operate during NVM access. The functions in the `ADLATUS_Eep.c` file periodically call up the function `FUN_CCtim_SystemTimer Check_V`, whose task it is to count down the software timer and, when necessary, to call up the watchdog functions.
- The "page mode" of some EEPROMs is not supported.

10.6 Troubleshooting

No troubleshooting has been implemented in the lower levels of the NVM functions. However, the functions of the upper levels are capable of transferring errors back to the application level. In the event of a reading error, the data is rejected there and alternative data records are accessed. Access to the alternative data can be configured. If a valid application can be accessed, its values are used. If, due to a failed programming attempt, the ADLATUS is unable to access the data, the values of the `ADLATUS_ CusInterfaceAudi.c` file are used. Writing errors are not processed and must be corrected in the low-level functions.

11 Data preparation

11.1 Overview

The `DataPrepareXXX.c` file contains the interface for the preparation of the date to be transferred during a "transfer data". This can involve the decrypting but also the decompression of the data records. However, decompression with the present function cannot always be carried out free of problems. The cause of this involves flash memory-specific data quantities that must be written as the smallest unit per programming cycle. With some memory this can involve 32 bit values, but with others also 512 bytes as the smallest unit.

The functionality implemented in the delivered version of the file is a simple copying function, which is meant to serve as an implementation aid.

If a procedure is selected that requires a time of more than one millisecond per data block for the reorganisation of the values, it may be necessary to operate the watchdog during the data conversion. This can be solved with the calling up of the `FUN_CCtim_SystemTimerCheck_V()` function.

11.2 Interface description

The routine is given a pointer to an input buffer, a pointer to an output buffer and an information byte. The information byte describes the format of the data in the input buffer and is normally the value delivered with a "Request Download". The value is passed on unchanged to the function. In the following example it is called `t_MemoryInfo_UB`.

The example function reads the data from the input buffer and writes the result in the output buffer.

Function description

Prototype:

```
UBYTE FUN_CCdp_DataPrepare_UB (
            PTR_BUFFER_UB t_DecryptBuffer_PBUF,
            PTR_BUFFER_UB t_EncryptBuffer_PBUF,
            UBYTE         t_MemoryInfo_UB )
```

Parameter:

`PTR_BUFFER_UB t_DecryptBuffer_PBUF`	Pointer to buffer with input data
`PTR_BUFFER_UB t_EncryptBuffer_PBUF`	Pointer to buffer with output data
`UBYTE t_MemoryInfo_UB`	Format of data in input buffer

Return value:

d_CCglo_Ready	(0x80)	No error

Example:

```
UBYTE  FUN_CCdp_DataPrepareXXX_UB ( \
           PTR_BUFFER_UB t_EncryptBuffer_PBUF, \
           PTR_BUFFER_UB t_DecryptBuffer_PBUF, \
           UBYTE         t_MemoryInfo_UB )
{
  UBYTE t_Result_UB;
  UBYTE t_Index_UB;

  /*-------------------------------------------------------*/
  /* Init internal variables                               */
  /*-------------------------------------------------------*/
  t_Result_UB = d_CCglo_Ready;
  t_MemoryInfo_UB |= 0;
  /*=======================================================*/
  /* -1- Data decryption                                   */
  /*=======================================================*/
  /* --> No algorithm defined.                              */
  /* Here is only a data copy implemented                  */
  for (t_Index_UB = 0; \
       t_Index_UB < t_EncryptBuffer_PBUF[d_CCglo_DLC]; \
       t_Index_UB ++    )
  {
    /* copy the data ..                                           */
    t_DecryptBuffer_PBUF[d_CCglo_StartOfData + t_Index_UB] = \
    t_EncryptBuffer_PBUF[d_CCglo_StartOfData + t_Index_UB];
    /* .. and clear up the buffer                                 */
    t_EncryptBuffer_PBUF[d_CCglo_StartOfData + t_Index_UB] = \
              (UBYTE) 0x00u;

  } /* -1- END OF for(..) ---------------------------------*/

  /*-------------------------------------------------------*/
  /* Set the decrypt buffer information                    */
  /*-------------------------------------------------------*/
  t_DecryptBuffer_PBUF[d_CCglo_BufferState] |= \
  d_CCglo_BufferValid;

  t_DecryptBuffer_PBUF[d_CCglo_ChannelNbr] = \
  t_EncryptBuffer_PBUF[d_CCglo_ChannelNbr];

  t_DecryptBuffer_PBUF[d_CCglo_MaxBufferLength] = \
  d_CCconsys_RxWorkBufferLength00;

  t_DecryptBuffer_PBUF[d_CCglo_DLC] = \
  t_EncryptBuffer_PBUF[d_CCglo_DLC];

  /*--------------------------------------------------------*/
  /* Reset the encrypt buffer information                   */
  /*--------------------------------------------------------*/
  t_EncryptBuffer_PBUF[d_CCglo_BufferState] &= \
  ~d_CCglo_BufferValid;
  t_EncryptBuffer_PBUF[d_CCglo_DLC] = 0x00;

  return  t_Result_UB;

} /*-+- END OF FUNCTION 'FUN_CCdp_DataPrepareXXX_UB' ++-+-*/
```

12 Access protection/Authorisation

12.1 Overview

The authorisation (security access) prior to programming and the testing of validity (Checksum/Signature) after programming should prevent the possibility of unauthorised manipulation of the software. A corresponding interface exists for the implementation of the authorisation (security access), which makes it possible to position desired procedures in the system. Because there are countless procedures for authorisation, the `ADLATUS_SecurityXXX.c` file contains for the least projects an immediately useable algorithm, but instead only a programme shell.

If a procedure is selected that requires a time of more than one millisecond for the calculation of the values, it may be necessary to operate the watchdog during the calculation. This can be solved by calling up the `FUN_CCtim_SystemTimerCheck_V()` function.

Interrupt capability is not planned in the currently available function. Interrupt capability means that the function is exited after a previously determined period of time, in order to then ensure that a communication timeout does no occur before the function is in turn called up again for further calculation. This has the consequence that the function run time may not exceed the timeout time of the communication.

12.2 Random number generation

The ADLATUS® contains a routine for the generation of a random number. After being called up, this routine returns a 4-byte long number.

Function description

Prototype:

```
ULONG  FUN_CCglo_GetRandomValue_UL ( void)
```

Parameter:

```
void
```

Return value:

ULONG	4-byte random number

Example:

```
/*----------------------------------------------*/
/* Create a seed                                */
/*----------------------------------------------*/
t_CCseccas_SecurityCtrl.Seed_U.ULong_ST.ULong0_UL = \
                       FUN_CCglo_GetRandomValue_UL ();
```

12.3 Seed & Key

The routines for seed or key calculation always have the same layout. The "CreateSeed" routine is passed to a buffer into which the seed is to be written. The "CheckKey" routine is passed to a buffer in which the received key is stored for checking.

Function description

Prototype:

```
void FUN_CCsec_CreateSeed_V ( \
                    PTR_BUFFER_UB t_SeedBuffer_PBUF )
```

Parameter:

`PTR_BUFFER_UB`	Pointer to buffer

Return value:

void	-

Example:

```
Void  FUN_CCsec_CreateSeed_V(PTR_BUFFER_UB t_SeedBuffer_PBUF)
{
  UNION4 Seed_U4
  /*-------------------------------------------------------*/
  /* Create a seed                                 */
  /*-------------------------------------------------------*/
  Seed_U4.ULong_ST.ULong0_UL = FUN_CCglo_GetRandomValue_UL ();
  /*-------------------------------------------------------*/
  /* Write the random value into given buffer              */
  /*-------------------------------------------------------*/
  t_SeedBuffer_PBUF[0x00] = (UBYTE) \
                           Seed_U4.UByte_ST.Ubyte0_UB;
  t_SeedBuffer_PBUF[0x01] = (UBYTE) \
                           Seed_U4.UByte_ST.Ubyte1_UB;
  t_SeedBuffer_PBUF[0x02] = (UBYTE) \
                           Seed_U4.UByte_ST.Ubyte2_UB;
  t_SeedBuffer_PBUF[0x03] = (UBYTE) \
                           Seed_U4.UByte_ST.Ubyte3_UB;
    return;
} /*-+- END OF FUNCTION 'FUN_CCsec_CreateSeed_V' -+-+-+-+-+*/
```

Function description

Prototype:

UBYTE FUN_CCsec_CheckKey_V (\
PTR_BUFFER_UB t_KeyBuffer_PBUF)

Parameter:

PTR_BUFFER_UB	Pointer to buffer

Return value:

d_CCseccas_AccessGranted	Access authorised
d_CCseccas_AccessDenied	Access denied

Example:

```
UBYTE  FUN_CCsec_CheckKey_V(PTR_BUFFER_UB t_KeyBuffer_PBUF)
{
  UBYTE t_Result_UB;
  UNION4 Key_U4

  /* Get the key from given buffer                            */
  Key_U4.UByte_ST.Ubyte0_UB = t_KeyBuffer_PBUF[0x00];
  Key_U4.UByte_ST.Ubyte1_UB = t_KeyBuffer_PBUF[0x01];
  Key_U4.UByte_ST.Ubyte2_UB = t_KeyBuffer_PBUF[0x02];
  Key_U4.UByte_ST.Ubyte3_UB = t_KeyBuffer_PBUF[0x03];

  /* Calculate the key from seed too                          */
  //→ .. fill in the correct algorithm!!!

  /* -1- Key is correct                                       */
  if (Key_U4.ULong_ST.ULong0_UL == 0xabcd )
  {
    t_Result_UB = d_CCseccas_AccessGranted;
  } /* -1- END OF if(..) ----------------------------------*/

  /* -1- Key is not correct                                   */
  else
  {
    t_Result_UB = d_CCseccas_AccessDenied;
  } /* -1- END OF else (if(..)) ---------------------------*/

  return t_Result_UB;

} /*-+- END OF FUNCTION 'FUN_CCsec_CreateSeed_V' -+-+-+-+-+*/
```

13 Cryptography

13.1 Integration of the kryptolib

For integrating the kryptolib into the bootloader, the following steps are necessary:

- Download of the kryptolib (version 2.1) from https://www.s2ds.net
- Copying the headerfiles from /lib/inc/ into the folder R:\05_header\crypto\.
- Copying the c-files from /lib/src/ into the folder R:\11_cryptolib\02_src\. The empty c-files within this folder will be overwritten thereby.

13.2 Configuration of the kryptolib-sources

For using the kryptolib in the ADLATUS, some settings have to be adjusted in the kryptolib code: enabling the watchdog support, disabling some debugging code and choosing security class and hash algorithm.

In order to automatically adjust the kryptolib a headerfile named ADLATUS_CryptoTypesConfig_cdf.h could be used. For applying, it has to be included into R:\05_header\crypto\cryptolib.h and R:\05_header\crypto\SecM_config.h as shown below:

```
#ifndef _ESC_..._H_
#define _ESC_..._H_

#include "ADLATUS_CryptoTypesConfig_cdf.h"

/*******************************************
* 1. INCLUDES                              *
*******************************************/
```

13.3 Redeclarations

It is necessary to include some kryptolib- headerfiles into the bootloader to allow access to datatypes and function prototypes. In order to prevent redeclarations of data types, a preprocessor instruction has to be inserted into R:\05_header\crypto\cryptolib.h. The combination of #ifndef… and #endif has to be applied as follows:

```
#ifndef d_CCcrypto_PreventReclaration

/** Type can only be TRUE (Non-zero) or FALSE (Zero).
Boolean operations deliver the expected results. */
typedef UINT8 BOOL;

/** BOOL value true. */
#define TRUE        ((BOOL)1)

/** BOOL value false. */
#define FALSE       ((BOOL)0)

#endif
```

13.4 Configuration of the functionality

To be able to use the download signature, the 1024bit RSA-Key - composed of modulus and exponent - has to be configured.

```
in R:\05_header\crypto\SecM_keys.h:

  #define SEC_MODULUS

  #define SEC_EXPONENT
```

Before using the encrypted data transfer, the 128Bit AES-Key – composed of the key and an initialisation vector – must be set up.

```
in R:\01_adlatus\02_src\ADLATUS_DataPrepareAudi.c:
  const UBYTE g_DecryptKey_AUB[]
  const UBYTE g_DecryptIv_AUB[]
```

The required functionality can be enabled with the following compiler switches.

```
in R:\05_header\ext\ADLATUS_CompilerSwitches_cdf.h:
  #define cs_Signatur_Aktiv
  #define cs_Decryption_Aktiv
```

13.5 Characteristics at Freescale HCS12X

If using the ADLATUS for the Freescale HCS12X, there are some further changes necessary to get cryptography working. Due to the special pointer layout (16bit address + content of page register), casting an address from unsigned long to a pointer may fail in some cases. Therefore, the way a buffers address is passed to the cryptolib has to be changed.

The type of sigResultBuffer in SecM_SignatureType has to be changed from UINT32 to UINT8*.

```
in R:\05_header\crypto\SecM_verifySignature.h,
 line 83:
 old:  UINT32 sigResultBuffer;
 new:  UINT8* sigResultBuffer;
```

The cast when assigning an address to this variable has to be changed.

```
in R:\11_cryptolib\02_src\SecM_CCC.c\SecM_CCC.c,
 line 129:
 old:  param->currentHash.sigResultBuffer =
                          (UINT32) verifyData->sig_buf;
 new:  param->currentHash.sigResultBuffer =
                          (UINT8*) verifyData->sig_buf;
```

A. Software Project

All components necessary to generate an ADLATUS® are filed in a software project in an appropriately structured form. This project must be integrated into the corresponding development environment.

Further information:	see CD: Readme.txt

B. Compiler

The ADLATUS® software project concerned has been translated with the following specified compiler.

Manufacturer:	See CD: Readme.txt
Compiler designation:	See CD: Readme.txt

The ADLATUS® software project concerned has been translated with the compiler settings described in the following:

See CD: Readme.txt

C VW-Specific Modifications

C.1 VW-specific time constants

Several time constants have been implemented in ADLATUS for the VW-specific programming procedure.

The settings are configured at the position designated as follows:

Module:	ADLATUS_ConfigProject.c
Application area:	Timer.01

There, amongst others, the following values are listed:

Format	Designation	Description
2 byte	c_CCconprj_StopCommResetTime_T16	Timeout following a successful flash procedure (in ms)
2 byte	c_CCconprj_Timelock_T16	Time lock after unsuccessful “Security Access” (in **sec**)
2 byte	c_CCconprj_EraseResetTime_T16	Timeout during the deleting of the flash (in ms)

The value of `c_CCconprj_StopCommResetTime_T16` determines how long it must be waited following a successful programming procedure before resetting and branching into the DSW (driver software). If a value of "0" is used here, there is no reset. The controller can only be reset via KL15.

The value of `c_CCconprj_Timelock_T16` determines the wait time after three unsuccessful access attempts. No further accesses are permitted before this time has elapsed. If a value of "0" is used here, this means an endless wait time!

The value of `c_CCconprj_EraseResetTime_T16` determines the value for the communications timeout when a flash area is to be deleted (following start routine erase memory). The ADLATUS® is reinitialised when the timeout occurs. If the value “0” is used here, the communications timeout is disabled.

C.2 Time lock function for security access

A time lock function has been implemented in the ADLATUS®. Following three unsuccessful access attempts at the Seed&Key, the time lock is activated, meaning that no more access attempts are permitted. The time lock is again deactivated after the parametered time has elapsed.
In order that the time can elapse both in the ADLATUS® and in the application, the procedure is described in the following.

A 1 byte memory position is used in the EEPROM for the time lock. This is addressed with the name 'd_CCnvm_READ__TIMELOCK'.

Module:	ADLATUS_ConfigProject.c
Application area:	Nvm.01

The number of unsuccessful access attempts is saved in this memory position.

Format	Designation	Description
1 byte	d_CCnvm_READ__TIMELOCK	Number of unsuccessful access attempts

When the number of failed attempts exceeds three, a timer for the time lock is activated in the ADLATUS®. When the parametered time has expired at the timer, the EEPROM cell is set to zero and the time lock function thereby deactivated.

In order to handle the time lock correctly in the application, the EEPROM cell can be selected. When this has a value equal to or greater than three, it can be set to zero after the wait time has expired.

C.3 ReadDataByIdentifier

With the aid of the ReadDataByIdentifier Services (0x22), data addressed via a RecordDataIdentifier can be read out from the control unit.
These can be stored either in EEPROM or in a flash area.

Configuration of the data to be read out takes place at the following position:

Type definition:	`ADLATUS_ConfigProject.c`
Application area:	`Nvm.02 – ReadDataByIdentifier (Service 0x22)`

The relevant data must thereby be entered into a structure of type `RDBIDENTRY_ST` (`ADLATUS_Types_tdf.h` → `Type definition Nvm.04`). The layout of this structure is shown in the following:

Format	Designation	Description
2 byte	`Identifier_UW`	RecordDataIdentifier
1 byte	`NvmEntryOrLenth_UB`	Associated NVM entry or number of bytes to be read
1 byte	`Flags_UB`	Flags
4 byte	`Address_UL`	Alternative address for incorrect EEPROM access, or address for flash access
4 byte	`AltAddress_UL`	Alternative address for incorrect EEPROM access, or address for flash access

All data fields are described in detail in the following.

Identifier_UW
Description

Contains the RecordDataIdentifier that uniquely identifies the data to be read out.

Flags_UB	
#define	Description
d_CCconprj_UseAltAddress	If the EEPROM access was unsuccessful, a default value shall be read out. This is read at the addresses Address_UL or AltAddress_UL, depending on whether the corresponding software block is valid.
d_CCconprj_ReadFromFlash	The data should be read from the flash and not from the EEPROM.

NvmEntryOrLenth_UB
Description

If the data is to be read out from the EEPROM, the entry NvmEntryOrLenth_UB describes the NVM entry associated with the RecordDataIdentifier. The entry given must be contained in the NVM address table (Application area: NVM.01 – Address Table, Byte 1 'Name_UB') (see Chapter 10.2).

If the data is to be read from the flash memory, NvmEntryOrLenth_UB gives the number of data bytes to be read out.

`Address_UL`
Description

If data is to be read out from the EEPROM:

When an EEPROM access was unsuccessful, default data is read out from this address (if the flag ‘`d_CCcusint_UseAltAddress`’ is set). This address is normally found in the driver software area. This only takes place, however, when the software block in which the address lies is recognized as being valid.

Data is to be read out from flash memory:

When the data is to be read out of flash memory, '`Address_UL`' is used as the source address of the data (normally in the driver software area). This only takes place, however, when the software block in which the address lies is recognized as being valid.

`AltAddress_UL`
Description

If, when reading data, the software block, in which the address '`Address_UL`' lies is recognized as invalid, the default data is read out from address '`AltAddress_UL`'. This normally points to a constant in the ADLATUS® area.

Example 1:

```
{
/*-----------------------------------------------------------------*/
/*      StateOfFlash (for 0x22 0x0405)                             */
/*--------------++-------------------------------------------------*/
/* Identifier:   */   (UWORD) 0x0405,
/* NvmEntryOrLen:*/   (UBYTE) d_CCnvm_READ__StateOfFlash,
/* Flags:        */   (UBYTE) d_CCglo_InitToZero,
/* Address:      */   (ULONG) d_CCglo_InitToZero,
/* AltAddress:   */   (ULONG) d_CCglo_InitToZero
}, /*-----------++-------------------------------------------------*/
```

An EEPROM entry that is uniquely identified with 'd_CCnvm_READ__StateOfFlash' is read out via RecordDataIdentifier 0x0405. If the EEPROM access is unsuccessful, no default data is used.

Example 2:

```
{
/*-----------------------------------------------------------------*/
/*      AppSoftwareNbr (for 0x22 0xF188)                           */
/*--------------++-------------------------------------------------*/
/* Identifier:   */   (UWORD) 0xF188,
/* NvmEntryOrLen:*/   (UBYTE) d_CCnvm_READ__AppSoftwareNbr,
/* Flags:        */   (UBYTE) d_CCconprj_UseAltAddress,
/* Address:      */   (ULONG) 0xA0043000,
/* AltAddress:   */   (ULONG)&C_CCcusint_AppIdent_ST.SoftwareNbr_AUB[0]
}, /*-----------++-------------------------------------------------*/
```

An EEPROM entry that is uniquely identified with 'd_CCnvm_READ__AppSoftwareNbr' is read out via RecordDataIdentifier 0xF188. If the EEPROM access is unsuccessful, default data from address 0xA0043000$_{hex}$ is read. If the address lies in an invalid software block, the data is read from the constant 'C_CCcusint_AppIdent_ST.SoftwareNbr_AUB'.

Example 3:

```
{
/*-----------------------------------------------------------------*/
/*      AppSoftwareVersionNbr (for 0x22 0xF189)                   */
/*--------------++--------------------------------------------------*/
/* Identifier:   */  (UWORD) 0xF189
/* NvmEntryOrLen:*/  (UBYTE) 0x04
/* Flags:        */  (UBYTE) d_CCconprj_ReadFromFlash,
/* Address:      */  (ULONG) 0xA0043050,
/* AltAddress:   */  (ULONG)&C_CCcusint_AppIdent_ST.SWVersion_AUB[0]
}, /*-----------++--------------------------------------------------*/
```

4 bytes are to be read out from flash memory from address 0xA0043050$_{hex}$ on, via RecordDataIdentifier 0xF189. If the address lies in an invalid software block, the data is read from the constant '`C_CCcusint_AppIdent_ST.SWVersion_AUB`'.

The end of the structure must always finish with the following entry:

```
{
/*-----------------------------------------------------------------*/
/* Last Entry - Don't change !!!                                  */
/*--------------++--------------------------------------------------*/
/* Identifier:   */  (UWORD) d_CCglo_NotUsed,
/* NvmEntryOrLen:*/  (UBYTE) (d_CCglo_NoEntry),
/* Flags:        */  (UBYTE) d_CCglo_NotUsed,
/* Address:      */  (ULONG) d_CCglo_NotUsed,
/* AltAddress:   */  (ULONG) d_CCglo_NotUsed
} /*------------++--------------------------------------------------*/
```

⚠ The last entry must be described with the values `d_CCglo_NotUsed` or `d_CCglo_NoEntry` as shown in the example.

D Project-Specific Modifications

D.1 Source code files

D.1.1 ADLATUS_ConfigProject.c

All settings needed for the project are activated in this module.

D.1.2 ADLATUS_CustomerInterfaceAudi.c

The following chapter applies for the ADLATUS® with UDS access:

Version recognition of ADLATUS® has been implemented, that can be read out via an identification service.

The version can be configured at the position designated as follows:

Module:	ADLATUS_CusInterfaceAudi.c
Application area:	CusInt.01 – Adlatus Version

The relevant data must thereby be entered into a 4-byte array. The significance of the version number is derived from the corresponding VW document.

When the ADLATUS® is initialized, the version recognition configured in '`ADLATUS_CusInterfaceAudi.c`' is compared with the version stored in the EEPROM. If these differ, the version contained in the EEPROM is updated.

Example:

```
/*-----------+----------------+---------+-----------------*/
/* Byte No   | Comment        | coding  | Value           */
/*-----------+----------------+---------+-----------------*/
/* 00 (high) | MSB            | hex     */ 0x55,
/*-----------+----------------+---------+-----------------*/
/* 01        |                | hex     */ 0x00,
/*-----------+----------------+---------+-----------------*/
/* 02        |                | hex     */ 0x01,
/*-----------+----------------+---------+-----------------*/
/* 03 (low)  | LSB            | hex     */ 0x00
/*-----------+----------------+---------+-----------------*/
```

D.1.3 ADLATUS_Adr_Info.h

All address information used in ADLATUS® is pooled in this module. This simplifies application work, since address changes can be centrally made at one point.

D.1.4 ADLATUS_ApplicationInterface.c

Depending upon the type of flash access, one finds several functions in this module.

'`FUN_CCapplint_JumpToFSW_V (void)`' is called up by ADLATUS® to access the driver software. The jump-in address is defined with the aid of '`d_CCcusint_FswEntryAddress`'.

'`FUN_CCapplint_SetFlashRequest_V (void)`' sets the flash request pattern. The driver software must place this pattern in a defined RAM cell in order to report the start of a reprogramming procedure to ADLATUS®.

'`FUN_CCapplint_SetResetRequest_V (void)`' sets the ECU reset request pattern. ADLATUS® writes this pattern into a defined RAM cell in order to inform the driver software that the ADLATUS® has received the 'ECUReset' command. Subsequently, the driver software must send the terminating positive answer to the request. If the driver software is not present, ADLAUTS® sends the terminating positive answer. [UDS-specific]

'`FUN_CCapplint_CheckFlashRequest_UB (void)`' checks whether a flash request pattern or another reset request pattern has been entered at the defined RAM position. [flash request pattern is universally valid; other patterns are UDS-specific]

'`FUN_CCapplint_ResetFlashRequest_V (void)`' erases any potentially existing pattern from the defined RAM cell.

The location of the RAM cell can be configured in the 'ADLATUS_Adr_Info.h' module: In addition, depending upon the compiler, modifications can be made in the linker locater file.

Designation	Description
d_CCadrinfo_FlashRequestAddress	Address where a flash request or other request pattern has been stored in RAM.

The values of the pattern can be configured at the position designated in the following:

Module:	ADLATUS_CusInterfaceAudi_cdf.h

The following patterns can be configured:

Designation	Description
d_CCcusint_FlashRequestAudi	Pattern with which the driver software signals a flash request to the ADLATUS®.
d_CCcusint_EcuResetRequestAudi [UDS-specific]	Pattern with which the ADLATUS® signals a successful reset via ECUReset (Service 0x11 01) to the application.
d_CCcusint_DefSessionRequestAudi [UDS-specific]	Pattern with which the ADLATUS® signals a successful reset via SessionControl-DefaultSession (Service 0x10 01) to the application.

Example:

```
/*-----------------------------------------------------------*/
/* Flash Request Pattern                                     */
/*-----------------------------------------------------------*/
#define d_CCcusint_FlashRequestAudi              0x55AAF00FUL
#define d_CCcusint_EcuResetRequestAudi           0x55AA0FF0UL
#define d_CCcusint_DefSessionRequestAudi         0x55AA0AA0UL
```

In the case of the UDS flash access, the procedure for ECUReset and SessionControlDefaultSession is defined in such a way that the ADLATUS® must reply to the service with response pending. The reset is subsequently triggered. The concluding response of the service will be sent either by the application or by ADLATUS®.

⚠	Both the application and the ADLATUS® can determine, on the basis of the value of the RAM cell, whether a positive reply to the ECUReset, a positive reply to the SessionControl or no reply at all (normal startup) must be sent.

In the case of TP2.0 projects with the 0x1A service for identification, one finds, in addition to these functions, which form the interface between the ASW and Adlatus, the following constants in the 'ADLATUS_ApplicationInterface.c' file:

```
C_CCapplint_ASWDefaultPartnumber1a9b_UL
C_CCapplint_ASWDefaultProgramStatus1a9b_UL
C_CCapplint_ASWDefaultDataStatus1a9b_UL
```

With the help of these constants, addresses of default data can be parameterised in the application software. This data is read out at the 1A 9B service, when access to the EEPROM did not function and the application software is valid. The objective here is that a controller with non-functional EEPROM can still be flashed. The part numbers, the programme status and the data status are parameterised.

D.1.5 ADLATUS_ExternalWatchdog.c

(or ADLATUS_ExternalWatchdogFlash.c)

Functions for the initialisation of an external watchdog. During the initialisation, the watchdog functions are copied from the flash into the RAM, from where they are executed.

D.1.6 ADLATUS_ExternalWatchdogRAM.c

(or ADLATUS_ExternalWatchdog.c)

This file contains the function for triggering the external watchdog. When adapting to the project-specific watchdog, it must be ensured that the function can be executed from the RAM. This means that it should not be possible to accessthe flash in the function (e.g. access to constants)

D.1.7 ADLATUS_NvmHandler.c

Interface between Adlatus/Reference and an EEPROM.

D.1.8 ADLATUS_Periphery.c

Interface to target hardware configuration. The functions contained therein are called up by Adlatus during the startup phase to place the control unit in a customer-defined condition.

D.1.9 ADLATUS_SSIHandler.c

Interface functions between NVM handler and EEPROM. The necessary modifications to the EEPROM must be made in this module.

D.2 Selection of the identification service

In the case of TP2.0 projects, the ADLATUS® can be configured as follows:

The ADLATUS® can use either the service $1A (ReadEcuIdentification) or the service $22 (ReadDataByIdentifier) as an indentification service. In order to make the corresponding selection, a compiler switch is provided in the "ADLATUS_CompilerSwitches_cdf.h" file.

If the service $22 is used, the following define must be active:
"#define cs_CCkwp_Use0x22"

If, instead, service $1A is used, it must contain the line
"#undef cs_CCkwp_Use0x22".

Simultaneous usage of both services is not provided for.

A separate project environment is contained in the "01_adlatus/01_prj" folder for each of the two configurations, as various source files are used:

The names of the project files vary depending upon the processor and compiler used.
Example:
"ADLATUS_Tasking_TC1766_VW_0x1A.pjt"
"ADLATUS_Tasking_TC1766_VW_0x22.pjt"

D.3 Access protection

For access protection (Seed & Key), an example implementation has been made (Key = Seed + 1). The necessary header and source code for the security module are contained in the directory “01_adlatus” or “05_header”. The desired algorithm must be integrated into the module '`ADLATUS_SecurityAudi.c`' in the functions '`FUN_CCsecaudi_CreateSeed_V`' and '`FUN_CCsecaudi_CheckKey_UB`'.

D.4 Initialising EEPROM

Depending upon the flash procedure, an EEPROM is used that is used both by the driver software and ADLATUS®. On one hand, the ADLATUS® accesses driver software values that are needed during a flash procedure. On the other hand, ADLAUTS® uses a few EEPROM entries internally to administer the flash procedure. Both the ADLATUS® EEPROM entries and several driver software entries must be initialized in order to allow error-free operation of ADLATUS. Non-initialised EEPROM can lead to cancellation of the flash procedure.

The following EEPROM entries must be initialised before ADLATUS® is used. The individual entries are configured at the location given below, as described in Chapter 10:

Type definition:	ADLATUS_ConfigProject.c
Application area:	NVM.01 – Address Table

All EEPROM entries described in the corresponding AUDI/VW documents must be initialised to a sensible value.

In addition to these EEPROM entries, there are still a few internal entries that are used by ADLATUS® for the administration/controlling of the flash procedure.

These entries must be initialised as follows:

Name	Flash access	Size in bytes	Description	Initial value
d_CCnvm_ READ__ TIMELOCK	TP20/ UDS	1	Counter for unsuccessful Seed & Key accesses	0x00 (no unsuccessful attempts)
d_CCnvm_ READ__INTIN CONSISTENCE	TP20	1	Validity of the application	0x80 (application is invalid) or 0x00 (application is valid)
d_CCnvm_ READ__Sys Programmed	UDS	1	Shows whether the application has been reprogrammed or whether the programming was successful	0x00 (application has not been reprogrammed or is not invalid) or 0x02 (application has been reprogrammed and CheckDependencies has run successfully)
d_CCnvm_READ__BootSW ProgrammingState	UDS	1	Programming status of the boot loader	0x01 (the boot loader SW block is valid)
d_CCnvm_READ__App ProgrammingState	UDS	1	Programming status of the application	0x00 (the application SW block is invalid) or 0x01 (SW block is valid)
d_CCnvm_READ__xxx ProgrammingState	UDS	1	Programming-status of other blocks	0x00 (the SW block is invalid) or 0x01 (block is valid)