GOAL V2.20

Generic Open Abstraction Layer

Programmer´s Manual

port GmbH

Regensburger Str. 7

D-06132 Halle/Saale

Disclaimer

This manual represents the current state of the product. Please check with port.de for the latest version as the document may have a newer version since errors may be corrected or changes for a newer version of the product may be incorporated. Port.de assumes no responsibility for errors in this document. Qualified feedback is appreciated at service@port.de.

This document is the Intellectual Property of port.de and is intended to be used with the described product only. It may be forwarded and/or copied in the original and unmodified format. All rights reserved.

The product enables to use technologies such as PROFINET, EtherNet/IP and/or EtherCAT and others. These technologies are promoted by trade organizations, such as PNO (profibus.org), ODVA (odva.org) or ETG (ethercat.org). These trade organizations as well maintain the specification and care about legal issues.
We strongly recommend to become a member of these organisations. Most technologies are making use of patented or otherwise copyrighted technologies, approaches or other intellectual property. The membership usually automatically entitles the member for use of most of the technology-inherent copyrighted or otherwise protected Intellectual Property of the corresponding trade organization and most 3rd parties. Otherwise the user will need to obtain licenses for many patented technologies separately.

Further we suggest to you to subscribe to the corresponding Conformance Test Tool of these trade organizations. For instance the ODVA only accepts conformance test applications from companies who have a valid membership and have a valid subscription to the recent Conformance Test Tool. We as port are members in all corresponding organizations and are holding a subscription to these tools - however you as a customer need to have an own membership and an own subscription to the tool.

All rights reserved

The programs, boards and documentations supplied by port GmbH are created with due diligence, checked carefully and tested on several applications.
Nevertheless, port GmbH cannot guarantee and nor assume liability that the program, the hardware board or the documentation are error-free or appropriate to serve a specific customer purpose. In particular performance characteristics and technical data given in this document may not be interpreted to be guaranteed product features in any legal sense.

For consequential damages, every legal responsibility or liability is excluded.
port has the right to modify the products described or their documentation at any time without prior warning, as long as these changes are made for reasons of reliability or technical improvement.
All rights of this documentation are with port. Unless expressively granted - the transfer of rights to third parties or duplication of this document in any form, whole or in part, is subject to written approval by port. Copies of this document may however be made exclusively for the use of the user and his engineers. The user is thereby responsible that third parties do not obtain access to these copies.
The soft- and hardware designations used are mostly registered and are subject to copyright.

Regensburger Straße 7
D-06132 Halle
Tel. +49 345 - 777 55 0
Fax. +49 345 - 777 55 20
E-Mail service@port.de
www.port.de
www.port-automation.com

Contents

Table of figures

Changelog

Version	Changes
1.0	Initial release

1 Introduction

1.1 About GOAL

GOAL is a sophisticated middleware to integrate real time communication in applications for industrial networking. GOAL connects extension modules, various operating systems and GOAL core modules with applications on different hardware platforms. The modular structure simplifies the development of embedded systems and makes the exchange of single GOAL components possible, for example the communication profile can be changed by the substitution of the extension modules with the suitable communication library.

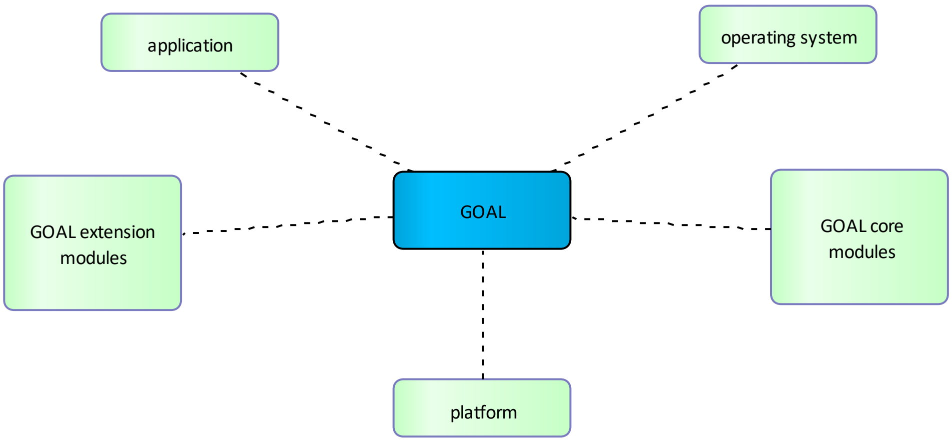


Figure 1: components of a GOAL system

The GOAL concept differentiates between hardware-dependent and hardware-independent sections in order to make the exchange of platforms possible without the rearrangement of the complete embedded system.

This manual describes the GOAL components, the GOAL structure and the usage of GOAL. Platform-specific information are documented in the GOAL Platform Manual for the specific hardware.

1.2 How to read this document

Within the document, special recommendations are given marked by two signs:

ℹ Special information giving hints to avoid common pitfalls when using the software

⚠ Special information to prevent malfunction of the software or that require special attention of the user.

2 Installation

The GOAL middleware is delivered as source code with the following directory structure:

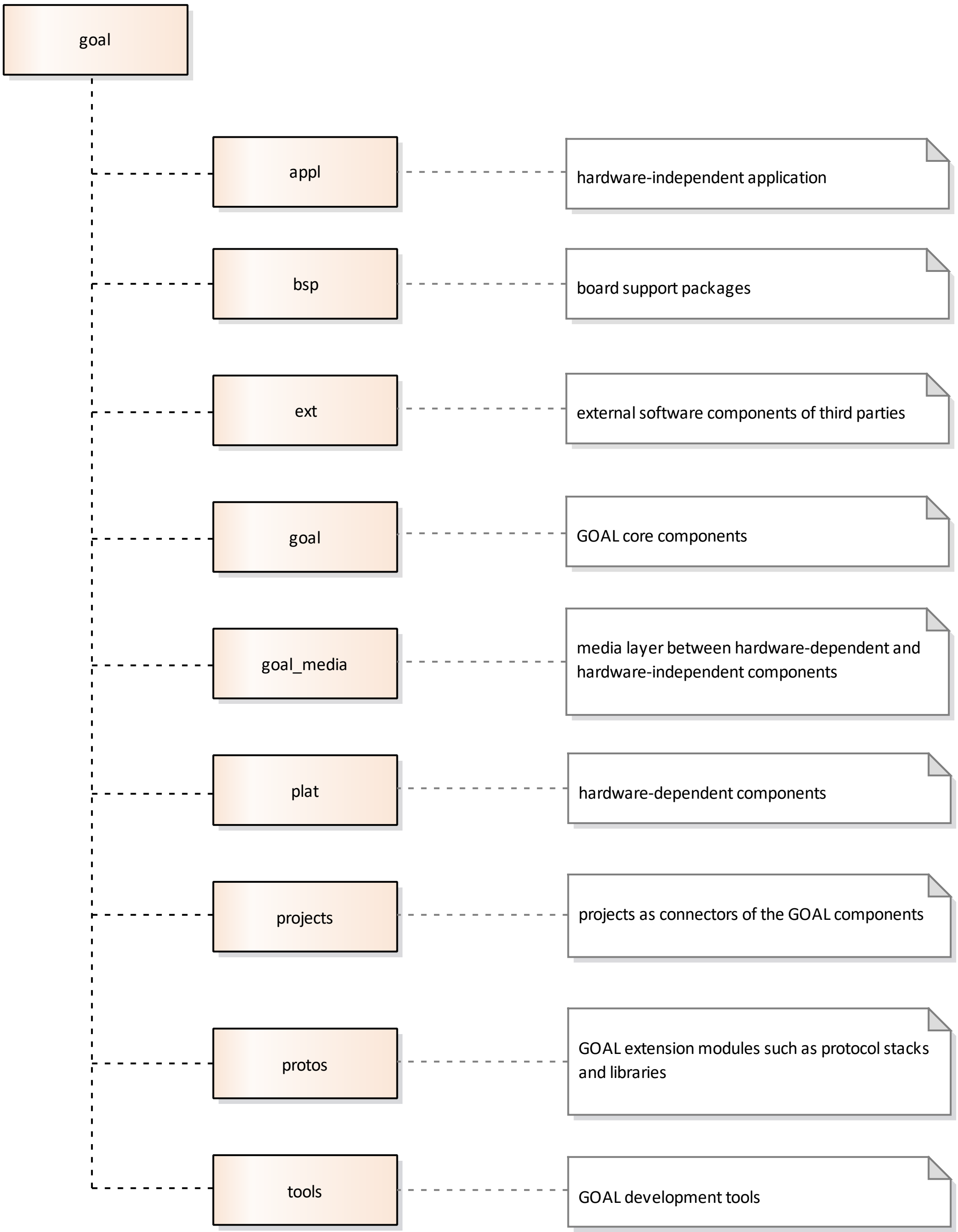


Figure 2: GOAL directory structure

All descriptions in the manuals refer to this directory structure.

2.1 Applications (appl)

The directory appl can contain various applications, see Figure 3. The user can add own application-specific files to each application.

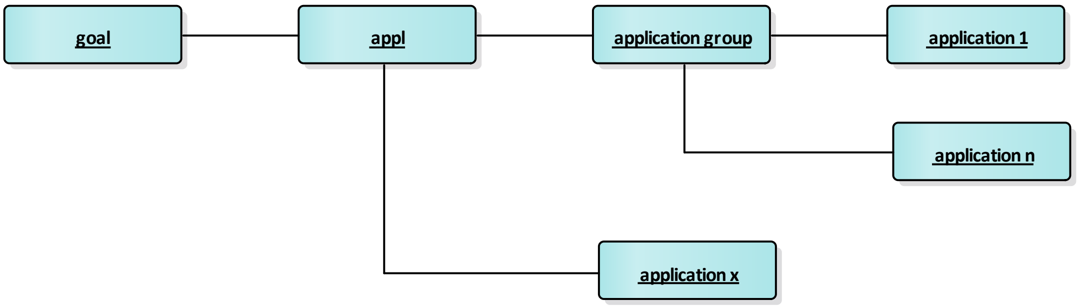


Figure 3: structure of the directory appl

The code of the applications shall be hardware-independent in order to exchange the platform without changes on the application. Application-specific functions depending on the hardware shall be connected to the hardware platform via media interfaces and media adapters. Application-specific functions can use the GOAL core modules. Public declarations and definitions of all GOAL
core modules are available by including the header file goal_includes.h.

The application …\goal\appl\00410\template belongs to the scope of the standard delivery and provides a template for own applications.

2.2 Platform (plat)

The directory plat represents the hardware platform and is divided in parts for architecture, boards and drivers according to the following structure:

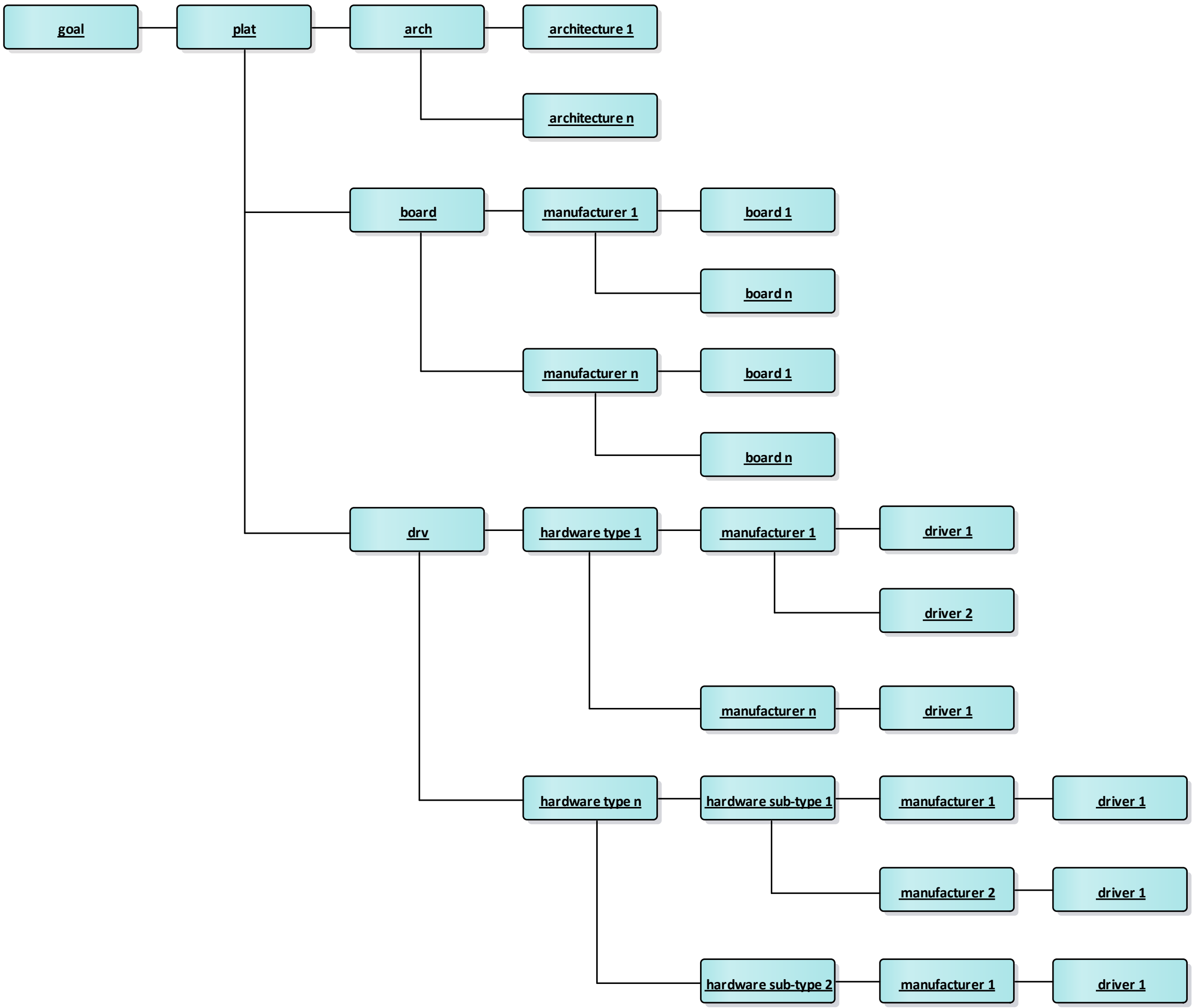


Figure 4: structure of the directory plat

Details to special platforms are documented in the GOAL Platform Manual for the specific platform.

2.3 Projects (projects)

The directory projects are designed to take in the compiler projects, which connect all necessary GOAL components including the application. The recommended structure of the directory projects is shown in Figure 5.

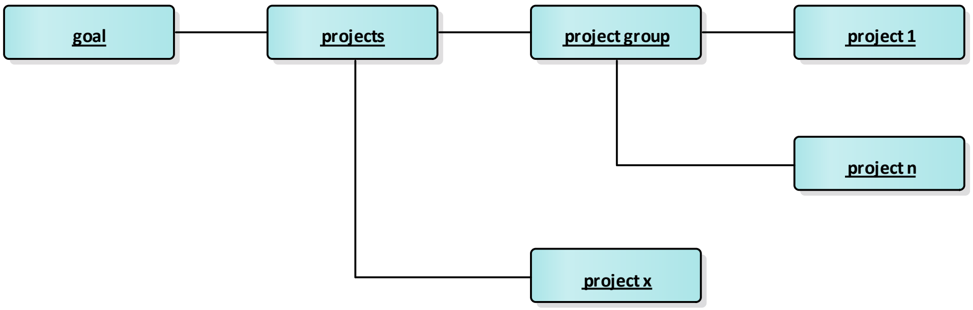


Figure 5: structure of the directory projects

3 GOAL model

GOAL is designed for the usage on

- single-core or multi-core systems
- systems with an operating system in single- or multithreaded design
- embedded systems without an operating system

Figure 6 shows the relationship between the GOAL components.

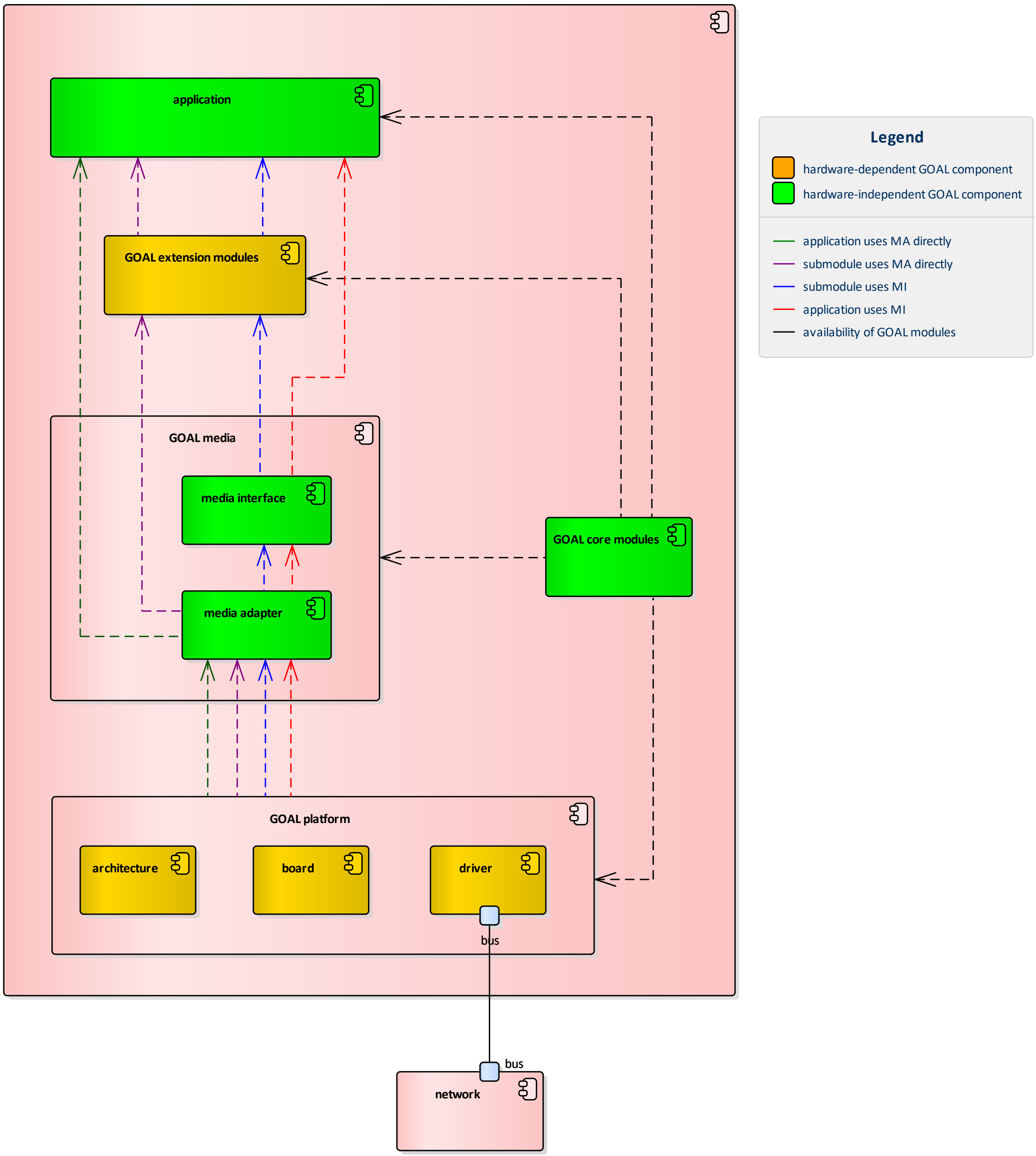


Figure 6: GOAL model

The colored arrows demonstrate different possibilities to apply GOAL components.

GOAL defines several types of components with a specific functionality:

3.1 GOAL core

The GOAL core modules provide basic middleware functionality as memory handling, timers, tasks, list etc. Those modules can be used from all GOAL components and from the application.

3.2 GOAL media adapter

Media adapter define an interface for drivers. Drivers in GOAL create a media adapter during registration. Upper layers use drivers through this unified interface, thus drivers and platforms are replaceable. Media adapters do not implement any additional logic, only provide a generic interface.

3.3 GOAL media interface

Media interfaces implement functionality based on media adapters or other media interfaces. This functionality may be a filesystem, an RPC implementation or even a communication stack. Media interfaces can be used by applications or other GOAL components.

3.4 GOAL extension modules

GOAL extension modules are additional software components, that implement application functions based on goal. These are for example:

- Communication stacks (TCP/IP, PROFINET, EtherNet/IP, EtherCAT, …)
- GOAL firewall
- GOAL log emitter
- GOAL Device Detection
- GOAL Web Server

3.5 GOAL architectures

These modules implent the architecture adaption layer between GOAL and the actual targets. There the platform specific parts of GOAL core module functionality are implemented.

3.6 GOAL boards

A board represents an actual hardware implementation of a CPU with additional peripherals and connectors, e.g. a development board. The code within this board file initializes peripherals and registers used drivers.

3.7 GOAL drivers

Drivers implement hardware access and provide the functionality through a media adapter to other layers of the stack.

4 GOAL state machine

The GOAL system provides a state machine for system and application startup and shutdown, simplified shown in Figure 7. The state machine is managed by the GOAL core module *main*.

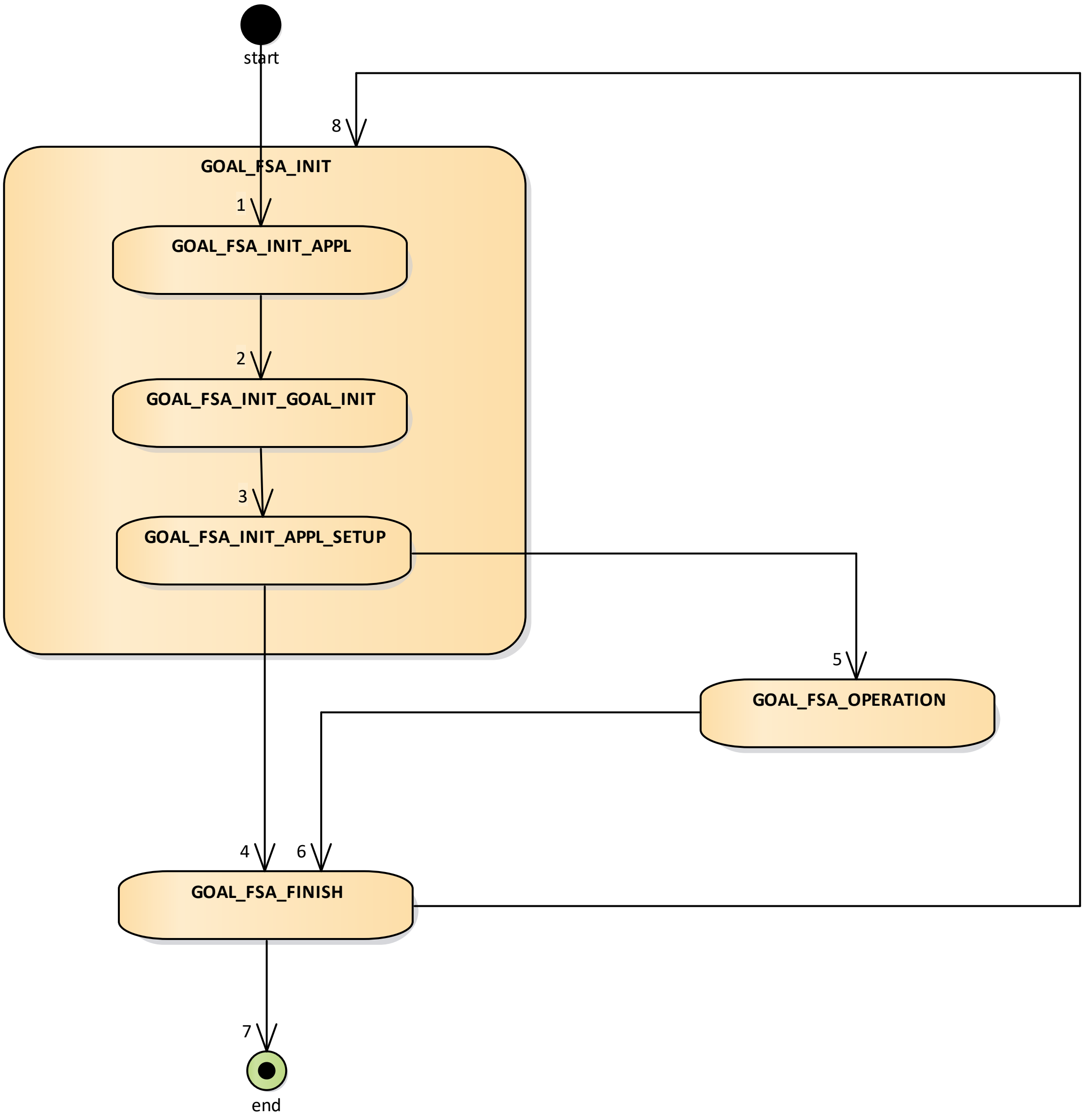


Figure 7: GOAL state machine

GOAL state	GOAL sub-state	Action(s)
GOAL_FSA_INIT	GOAL_FSA_INIT_APPL	application-specific initialization before GOAL components are initialized
	GOAL_FSA_INIT_GOAL	initialization of GOAL components

GOAL state	GOAL sub-state	Action(s)
		including the initialization of the GOAL platform
	GOAL_FSA_INIT_APPL_SETUP	application-specific initializations depending on GOAL core modules and configuration of the GOAL system
GOAL_FSA_OPERATION	--	normal operation including the execution of loop-controlled functions
GOAL_FSA_FINISH	--	halt or reset the GOAL system

Table 1: GOAL states

GOAL state transition	Event(s)
1	automatic transition after power-on or reset
2	automatic transition if application was initialized successful
3	automatic transition if all GOAL components and the application was initialized successful
4	an error occurred during initialization
5	automatic transition if GOAL system was configured
6	a severe error was occurred during normal operation
7	GOAL system is halted
8	GOAL system is reset and re-starts again

Table 2: GOAL state transitions (see Figure 7)

4.1 GOAL IDs

GOAL implements a concept of creating and identifying instances of objects by IDs. One could create two instances of a SPI driver, each operating on a different channel. Those two instances would then be identified by different IDs.

If only one instance of an object of specific type is created, the default ID of GOAL_ID_DEFAULT can be used. This ID can be reused for different types (e.g. for a driver and for a Media Interface), since they are directly realted to the object type.

Beside that each software component uses different IDs for identification or logging. Those IDs are defined in goal/goal_id.h. Here is an excerpt:

```
#define GOAL_ID_DEFAULT               (0)       /**< GOAL : Default ID */
#define GOAL_ID_BM                    (1)       /**< GOAL : Bitmap Handling */
#define GOAL_ID_CM                    (2)       /**< GOAL : Config Manager */
#define GOAL_ID_CTC                   (3)       /**< GOAL : CTC */
#define GOAL_ID_ETH                   (4)       /**< GOAL : Ethernet */
#define GOAL_ID_LIST                  (5)       /**< GOAL : List Management */
#define GOAL_ID_LOCK                  (6)       /**< GOAL : Lock Management */
#define GOAL_ID_LOG                   (7)       /**< GOAL : Logging */
#define GOAL_ID_MAIN                  (8)       /**< GOAL : Main */
```

```
#define GOAL_ID_MEM                       (9)       /**< GOAL : Memory Management */
#define GOAL_ID_MI                        (10)      /**< GOAL : Media Interface */
#define GOAL_ID_MA                        (11)      /**< GOAL : Media Adapter */
#define GOAL_ID_NET                       (12)      /**< GOAL : TCP/IP Networking */
#define GOAL_ID_REG                       (13)      /**< GOAL : Register Handling */
#define GOAL_ID_RPC                       (14)      /**< GOAL : RPC (CTC)  */
#define GOAL_ID_TGT                       (15)      /**< GOAL : Target */
#define GOAL_ID_DRV                       (16)      /**< GOAL : Driver */
#define GOAL_ID_TASK                      (17)      /**< GOAL : Task Management */
#define GOAL_ID_TMR                       (18)      /**< GOAL : Timer Management */
```

Code 1 goal ID list excerpt

4.2 GOAL initialization

All GOAL components are initialized in state GOAL_FSA_INIT. The initialization covers:

- application-specific initializations in state GOAL_FSA_INIT_APPL,
- the embedding of initialization functions in the GOAL initialization process by staging,
- the initialization of each GOAL component,
- the combination of GOAL components by registration in state GOAL_FSA_INIT_GOAL,
- the installation of loop-controlled processes and
- application-specific configurations in state GOAL_FSA_INIT_APPL_SETUP.

All necessary services must be created and initialized in the state GOAL_FSA_INIT, because it is only allowed to allocate memory in this state.

4.2.1 Staging

The GOAL system organizes the initialization process in stages. GOAL uses fixed stages. Each GOAL core module has own stages. Some further stages complete the range of stages. Normally there are two stages for each module:

- GOAL_STAGE_*_PRE: The initialization function shall be executed. This stage represents the start of initialization of the considered component.
- GOAL_STAGE_*: The initialization function is finished. This stage represents the end of initialization of the considered component.

Each GOAL component, also the application, can enter callback functions on every stage. The order of the stages determines the order of execution of the callback functions. It is possible to add more than one callback function to a stage. The order of execution of the callback functions within a single stage is determined by the order of registration. The callback functions are listed in the stage table, see Figure 8.

Each stage is identified by a stage-ID, defined in the enum GOAL_STAGES_T in <GOAL>/goal/goal_main.h. The callback functions with the smallest stage-ID are executed first. Platform-specific initializations assigned to the smallest stage-IDs, followed by GOAL core modules and GOAL extension modules. Table 3 lists some stages, which are most interesting from the application point of view.

Stage	Description
GOAL_STAGE_TARGET_PRE	for the initialization of the platform
GOAL_STAGE_TARGET	indicate, that the initialization of the platform is ready
GOAL_STAGE_BOARD_PRE	for additional initialization of the board
GOAL_STAGE_BOARD	indicate, that the initialization of the board is ready
GOAL_STAGE_MODULES_PRE	for the initialization of GOAL extension modules or the application
GOAL_STAGE_MODULES	indicate, that the initialization of GOAL extension modules or the application is ready
GOAL_STAGE_GOAL_PRE	for the last initialization steps in state GOAL_FSA_INIT_GOAL
GOAL_STAGE_GOAL	indicate, that the initialization of all GOAL components is ready

Table 3: some stages useful for applications

Each entry in the stage table contains a stage-ID, the direction type and a callback function. There are two direction types:

- GOAL_STAGE_INIT: These stage table entries are processed in state GOAL_FSA_INIT_GOAL.
- GOAL_STAGE_SHUTDOWN: This direction type is reserved for future use.

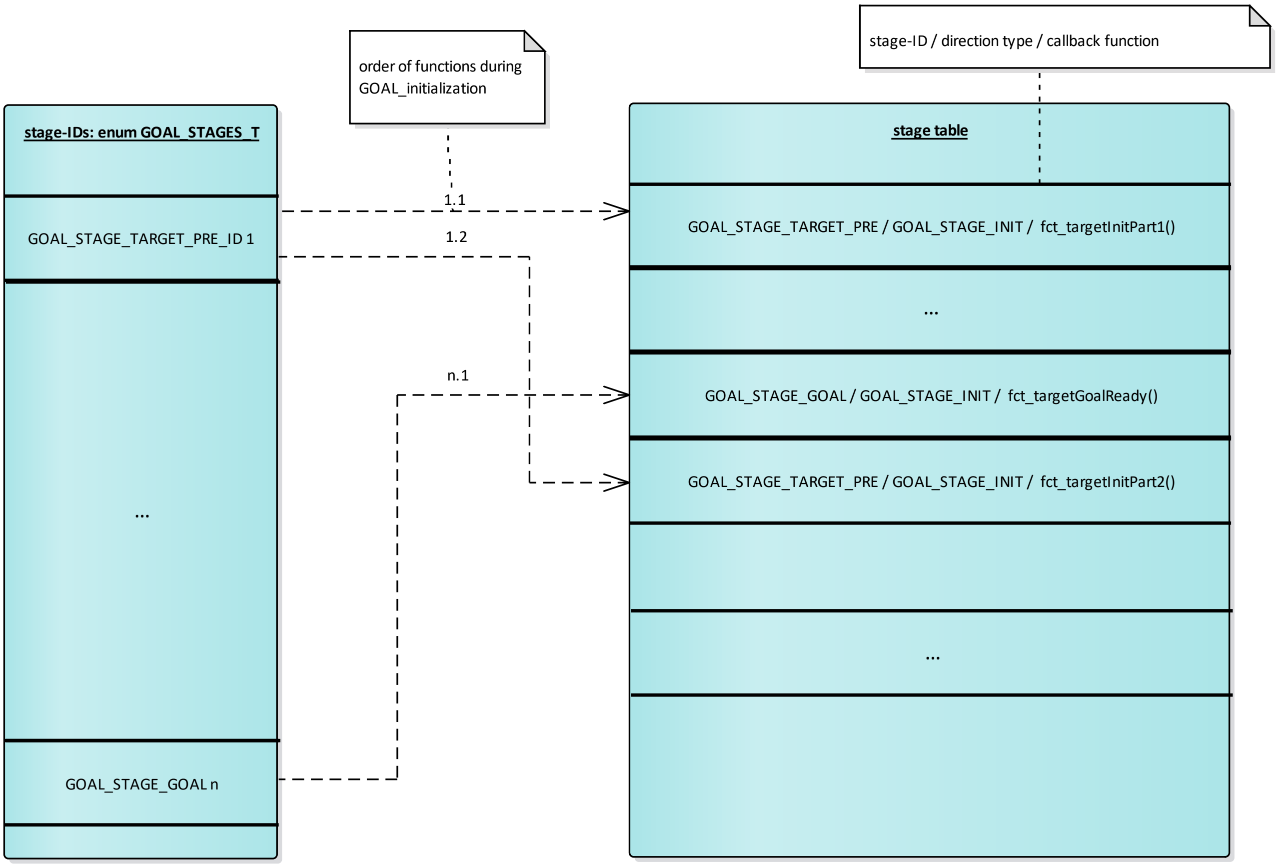


Figure 8: function order at staging

At the beginning of state GOAL_FSA_INIT_GOAL the GOAL core modules and GOAL extension modules register their callback functions. After registration the callback functions are executed.

For the example in Figure 8 the initialization functions are called in the following order:

1.1 fct_targetInitPart1()
1.2 fct_targetInitPart2()
…
n.1 fct_targetGoalReady()

4.2.2 Platform API

During initialization GOAL requires the function goal_targetInitPre() to initialize the used platform:

Prototype	`GOAL_STATUS_T goal_targetInitPre(void)`
Description	This indication function initializes the complete platform and is called in the state GOAL_FSA_INIT_GOAL in stage GOAL_STAGE_TARGET_PRE.
Parameters	None
Return values	GOAL return status, see chapter 8.3
Category	Mandatory

4.2.3 Registration of media interfaces, media adapters and drivers

GOAL allows to combine various hardware and software components with each other. The various components are connected to each other by a registration.
The platform-specific drivers are connected to platform-independent media adapters (MA). Media adapters represent a generic driver interface. Media adapters can be connected to media interfaces (MI). Media interfaces represent a generic connection interface between a media adapter and a special higher layer module.

Example 1: A GOAL device shall store parameters in the nonvolatile memory (NVM). The parameters are split in different blocks. The GOAL device uses the Synergy S7 platform. The GOAL device has to initialize and register the following GOAL components:

1. Initialize the GOAL driver for the access to the NVM for the Synergy S7 platform. The GOAL driver handles the accesses to the memory hardware.
2. The GOAL driver registers to the MA for the nonvolatile storage itself. The MA for the nonvolatile storage provides a generic interface for accesses to the memory hardware.
3. The different memory blocks are managed about various memory regions. The MI for the nonvolatile storage provides the management of memory regions and is added to the GOAL project. The MI relates to the MA for the nonvolatile storage by a registration.
4. The application specifies a memory region for each parameter block. Each region is registered to the MI for the nonvolatile storage.

The media interfaces and the media adapters can be identified by MI or MA unique ID's. The MI-IDs and MA-IDs have separate lists and are independent from each other. During registration a unique handle is created for each ID. Normally the registration is done in stage GOAL_STAGE_TARGET_PRE in state GOAL_FSA_INIT_GOAL.

Example 2: The registration of the media interface, media adapter and driver are shown for MCTC over SPI:

1. Define a MA-ID and a MI-ID in …\goal\plat\board\...\goal_target_board.h or use default ID GOAL_ID_DEFAULT.

2. Register the SPI driver in <GOAL>/plat/board/…/goal_target_board.c, thus creating a media adapter (MA):

```
/* register SPI driver */
res = goal_drvSpiSynReg(GOAL_ID_DEFAULT, 0);
if (GOAL_RES_ERR(res)) {
    goal_logErr("failed to register Synergy SPI driver");
    return res;
}
```

3. Register the MCTC MI with ID to the SPI MA as a parameter. The created MI will also utilize the ID GOAL_ID_DEFAULT. Since SPI MA and MCTC MI are of different type, this is ok.

```
/* register a new MCTC MI */
res = goal_miMctcSpiReg(
    GOAL_ID_DEFAULT,
    GOAL_ID_DEFAULT,
    mpMiDmRead,
    mpMiDmWrite);
if (GOAL_RES_ERR(res)) {
    goal_logInfo("Unable to reg MI SPI");
    return res;
}
```

Depending on the media type and the driver the registration functions can require different parameters.
Some drivers generate the MA-ID according to an implemented rule automatically. The rule is documented
in the suitable GOAL Platform Manual.

4.2.4 Application-specific indication function for initialization

Application-specific initializations are implemented about the indication function appl_init() located in <GOAL>/ appl/.../goal_appl.c normally:

Prototype	`GOAL_STATUS_T appl_init (void)`
Description	This indication function allows to include application-specific initialization steps before the GOAL core modules or GOAL extension modules are initialized. GOAL

	core modules must not be used. This indication function is called by GOAL automatically in state GOAL_FSA_INIT_APPL_INIT.
Parameters	None
Return values	GOAL return status, see chapter 8.3
Category	Optional If appl_init() does not exist in the application, GOAL uses an empty default function.

4.2.5 Install loop-controlled processes

Functions with low priority can be executed loop-controlled in the state GOAL_FSA_OPERATION. GOAL provides a loop mechanism, called GOAL loop. There are the following possibilities to install application-specific loop functions in the GOAL loop:

- implementation of the indication function appl_loop() or
- append the functions, which shall be called loop-controlled, to the list of loop functions.

⚠ Loop functions run in the main loop context of GOAL, thus these functions should be limited in execution time to minimize the effect on other loop functions. Longer processes should be split into multiple sequential steps.

4.2.5.1 Implementation of appl_loop()

GOAL provides the indication function appl_loop() for calling application-specific functions in the GOAL loop:

Prototype	`void appl_loop (void)`
Description	This indication function allows to execute application-specific functions loop-controlled. This indication function is called in the GOAL loop in state GOAL_FSA_OPERATION.
Parameters	None
Return values	None
Category	optional If appl_loop() does not exist in the application, GOAL uses an empty default function.

Example 3: The function applUpdate() shall be called loop-controlled. This function is implemented in the indication function appl_loop().

```
void appl_loop (void) {
    applUpdate();
}
```

Code 2 appl_loop example usage

4.2.5.2 Function list

GOAL provides a further possibility to integrate application-specific functions in the GOAL loop. GOAL manages all functions, which shall be executed in the GOAL loop in state GOAL_FSA_OPERATION about a function list. A function can be added to the loop function list by function goal_mainLoopReg(). Each loop function must have the following function prototype:

```
void loopFunction (
    void
);
```

The loop function list is created in state GOAL_FSA_INIT. At the beginning of state GOAL_FSA_INIT_APPL the loop function list is empty. The application can register loop functions. The GOAL core modules and GOAL extension modules register their loop functions in state GOAL_FSA_INIT_GOAL.

The order of execution of the loop functions depends on the order of registration. The first registered loop function is executed at first.

Example 4: The function applUserLoop() shall be executed loop-controlled. The registration is made in appl_init().

```
void applUserLoop (void) {
    …
}

GOAL_STATUS_T appl_init(void) {
    GOAL_STATUS_T res;           /* GOAL return value */

    res = goal_mainLoopReg(applUserLoop);

    return res;
}
```

Example 5: The function applDeviceLoop() shall be executed loop-controlled. The registration is made in appl_setup().

```
void applDeviceLoop (void) {
    …
}

GOAL_STATUS_T appl_setup(void) {
    GOAL_STATUS_T res;           /* GOAL return value */

    res = goal_mainLoopReg(applDeviceLoop);

    return res;
}
```

Example 6: The function applFunc() shall be executed loop-controlled. The registration is made during initialization. In Example 7 the function applActivate() is registered and called during initialization. The loop function is registered in this initialization function.

```
void applFunc (void) {
    …
}

GOAL_STATUS_T applActivate(void) {
```

```
        GOAL_STATUS_T res;            /* GOAL return value */

        res = goal_mainLoopReg(applFunc);

        return res;
    }
```

4.2.6 Application-specific indication function for configuration

After initialization the application has the possibility to configure the GOAL system. The GOAL system expects the configuration in the indication function appl_setup() located in …\goal\appl\...\goal_appl.c normally:

Prototype	`GOAL_STATUS_T appl_setup (void)`
Description	This indication function allows to configure the GOAL system after initialization. This indication function is called by GOAL automatically in state GOAL_FSA_INIT_APPL_SETUP.
Parameters	None
Return values	GOAL return status, see chapter 8.3
Category	Optional If appl_setup() does not exist in the application, GOAL uses an empty default function.

4.2.7 Integration of user functions in staging system

The stage table is created in state GOAL_FSA_INIT. At the beginning of state GOAL_FSA_INIT_APPL the stage table is empty. Its entries are registered in the indication function appl_init() or by the GOAL core modules and GOAL extension modules in state GOAL_FSA_INIT_GOAL. The registration is made by function goal_mainStageReg(). Each callback function must have the following function prototype:

```
    GOAL_STATUS_T callbackFunction (
        void
    );
```

All staged initialization functions are executed after registration in the state GOAL_FSA_INIT_GOAL.

Example 7: At the end of the initialization the application-specific function applActivate() shall be called. applActivate() is assigned to stage GOAL_STAGE_GOAL. A new entry for the stage table is created about the declaration of stageReady. This new entry is appended to the stage table by function goal_mainStageReg(). The registration is located in the indication function appl_init().

```
    GOAL_STAGE_HANDLER_T stageReady; /* create new entry for stage table */

    GOAL_STATUS_T applActivate (void) {
        …
    }

    GOAL_STATUS_T appl_init(void) {
        GOAL_STATUS_T res;            /* GOAL return value */
```

```
    res = goal_mainStageReg(GOAL_STAGE_GOAL, &stageReady, GOAL_STAGE_INIT,
          applActivate);

    return res;
}
```

GOAL evaluates the return value of staging functions. If such a function returns an error, the goal initialization will fail.

4.3 GOAL operation

In the state GOAL_FSA_OPERATION the GOAL system executes tasks, interrupt routines and loop-controlled functions. The loop-controlled functions are executed by calling the function goal_loop() in the main() function or in a task regularly without a valid cycle time. In goal_loop() the function appl_loop() and/or the listed loop functions are executed. The registration of loop functions is described in chapter 4.2.4.

4.4 GOAL finish

4.4.1 Halt

The GOAL system is stopped. The halt behavior is platform-specific and described in the suitable GOAL Platform Manual. GOAL requires the indication function goal_targetHalt() as platform API function:

Prototype	`void goal_targetHalt(void)`
Description	This indication function stops the program.
Parameters	None
Return values	None
Category	Mandatory

4.4.2 Reset

The GOAL system is reset and starts again. The reset behavior is platform-specific and described in the suitable GOAL Platform Manual. GOAL requires the indication function goal_targetReset() as platform API function:

Prototype	`void goal_targetReset(void)`
Description	This indication function resets the platform and re-starts the program.
Parameters	none
Return values	none
Category	mandatory

5 GOAL core modules (goal)

The directory goal contains the GOAL core modules. One source and one header files exist for each GOAL core module. All GOAL core modules shall be integrated in the GOAL system, i.e. all GOAL core modules are added to the compiler-project. The functions are described in detail in the GOAL Reference Manual.

The header file goal_includes.h summarizes all header files of the GOAL core modules. The application includes all public information of the GOAL core modules with this header file.

GOAL core modules are configured by compiler-defines and/or configuration variables. The interface for the configuration by variables is in <GOAL>/goal/cm.

Some GOAL core modules provide a command line interface. The extensions for the handling via the command line are saved in additional files (goal*_cli.*). This chapter only describes the supported commands. The command line interface itself represents a GOAL extension module and is documented in chapter 7.2.

5.1 Heap Memory Allocator (goal_alloc)

This GOAL core module provides functions to allocate memory. However GOAL considers the inability of embedded systems to manage memory fragmentation, thus memory allocation is limited to the initialization phase, i.e. it is not possible to allocate or free memory during normal operation. The functions of the Memory Allocator are only allowed in the state GOAL_FSA_INIT.

GOAL memory allocation is limited to startup of the application. This originates in the inability to handle memory fragmentation in embedded systems.

The memory allocator uses a statically defined HEAP, which size is configurable. The memory is allocated on base of the alignment specified by the compiler-define GOAL_TARGET_MEM_ALIGN_NET. If a special alignment is required, the Memory Allocator supplies special functions to set the desired alignment for the allocation of memory. The name of these functions have the postfix “Align”, e.g. goal_memAllocAlign().

The Memory Allocator allows to check, that only the allocated memory range is in use by a boundary checker. The boundary checker adds bytes around the allocated memory range and fills the bytes with special patterns. The application can check that the patterns are unchanged by calling the function goal_memCheck(). The boundary checker can be activated or deactivated by the compiler-define GOAL_CONFIG_DEBUG_MEM_FENCES and shall only be used during development.

This GOAL core module provides no CM-variables and no command line interface.

GOAL files:
goal_alloc.[h,c]

example:
not available

5.1.1 Configuration

The following compiler-defines are available to configure the Memory Allocator:

Using the stdlib memory allocator is a debugging features and may lead to additional code being linked to the application, thus requiring more resources.

GOAL_CONFIG_MM_EXT:
    0: use goal memory alocator (default)
    1: use stdlib alloc functionality (only for debugging)
GOAL_TARGET_MEM_ALIGN_NET:
    alignment for network transfers, see chapter 8.4

GOAL_CONFIG_HEAP_SIZE:
    size of the heap memory, see chapter 8.5

The following compiler-defines are available for debug purposes:

GOAL_CONFIG_DEBUG_MEM_FENCES:
    0: memory boundary checker is disabled (default)
    1: memory boundary checker is enabled

GOAL_CONFIG_DEBUG_HEAP_USAGE:
    0: debug feature disabled
    1: log actual heap usage per component

5.1.2 Implementation guidelines

5.1.2.1 Allocate a memory range

1. Create a handle, which is directed to the allocated memory and allocate memory

```
void *pMem = NULL;              /* memory pointer */
GOAL_STATUS_T res;              /* GOAL return value */

res = goal_memCalloc(&pMem, 2048);
```

It is important to utilize the function as shown. Creating a pointer pointer variable (void **ppMem) and passing this to the function as an argument (goal memCalloc(ppMem, …) will fail.

5.2 Bitmap handling (goal_bm)

This GOAL core module provides a function to allocate memory for a bit-field. Single bits of the bit-field can be taken from the bit-field by a function. The function can use the bit. If the bit is not more needed the function has to return the bit to the bit-field, see Figure 9. If a bit is used, another function cannot take this bit from the bit-field. The bit-field must be allocated in state GOAL_FSA_INIT.

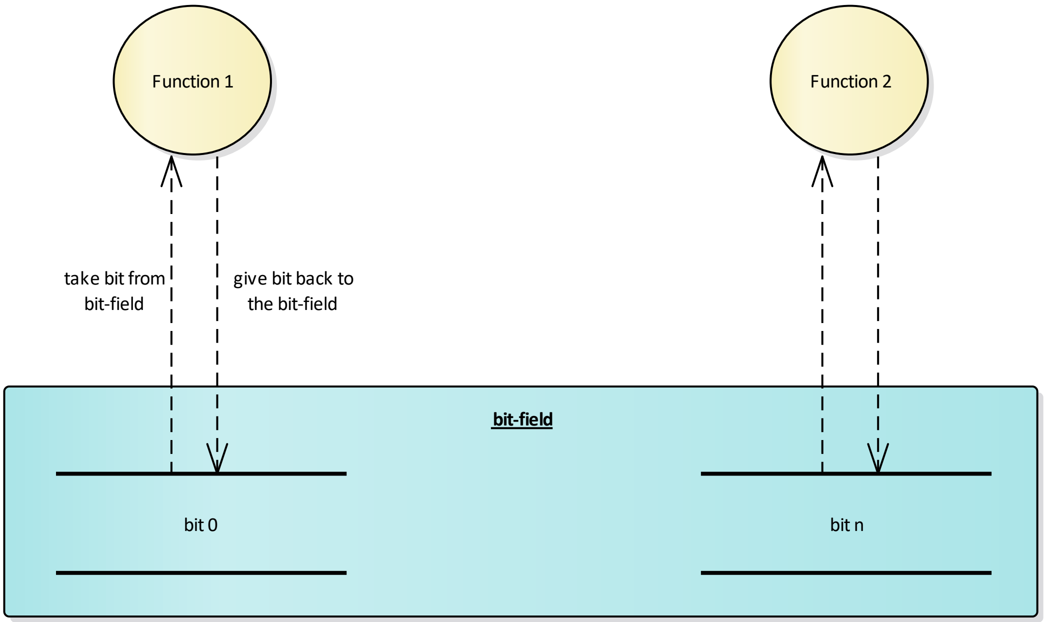


Figure 9: bitmap handling

Locking mechanisms are not implemented for the functions of this GOAL core module. If the locking of the bit-field is necessary, the locking must be done by the caller.

This GOAL core module provides no CM-variables and no command line interface.

GOAL files:

goal_bm.[h,c]

example:

not available

5.2.1 Implementation guidelines

5.2.1.1 Create a bit-field with a lock

```
/* Create a handle to the bit-field. */
GOAL_BM_T *pFlags = NULL;

/* Create a handle for the lock to the bit-field. */
GOAL_LOCK_T *pLockFlags;

/* Create a binary lock to avoid multiple accesses to the bit-field. A binary lock
has the value range [0,1]. */
GOAL_STATUS_T res;           /* GOAL return value */
res = goal_lockCreate(GOAL_LOCK_BINARY, &pLockFlags, 0, 1, GOAL_ID_BM);

/* Allocate the memory for the bit-field in state GOAL_FSA_INIT for 16 bits. */
if (GOAL_RES_OK(res)) {
    res = goal_bmAlloc(&pFlags, 16);
}
```

5.2.1.2 Take a bit from the bit-field

```
GOAL_STATUS_T res;       /* GOAL return status */
uint32_t bitNum;         /* number of the bit */

/* Set the lock. If the lock is not available, wait on the lock forever. */
res = goal_lockGet(pLockFlags, GOAL_LOCK_INFINITE);

/* Take the next available bit from the bit-field. */
if (GOAL_RES_OK(res)) {  /* lock is set successful */
    res = goal_bmBitReq(pFlags, &bitNum);

    /* Reset the lock. */
    goal_lockPut(pLockFlags);
}
```

5.2.1.3 Return a bit to the bit-field

```
GOAL_STATUS_T res;       /* GOAL return status */

/* Set the lock. If the lock is not available, wait on the lock forever. */
res = goal_lockGet(pLockFlags, GOAL_LOCK_INFINITE);

/* Return the bit to the bit-field. */
if (GOAL_RES_OK(res)) {  /* lock is set successful */
    res = goal_bmBitRel(pFlags, bitNum);

    /* Reset the lock. */
    goal_lockPut(pLockFlags);
}
```

5.3 Configuration Manager (goal_cm)

The Configuration Manager provides an interface to handle configuration variables during runtime. If a NVM is available, the configuration variables can also be managed nonvolatile. Besides providing runtime configuration data the CM also provies an interface for the Device Manger Tool/ GOAL Manager Tool.

The Configuration Manager organizes the configuration data module-wise, called CM-module. Each CM-module contains a list of configuration variables, called CM-variables, see Figure 10.

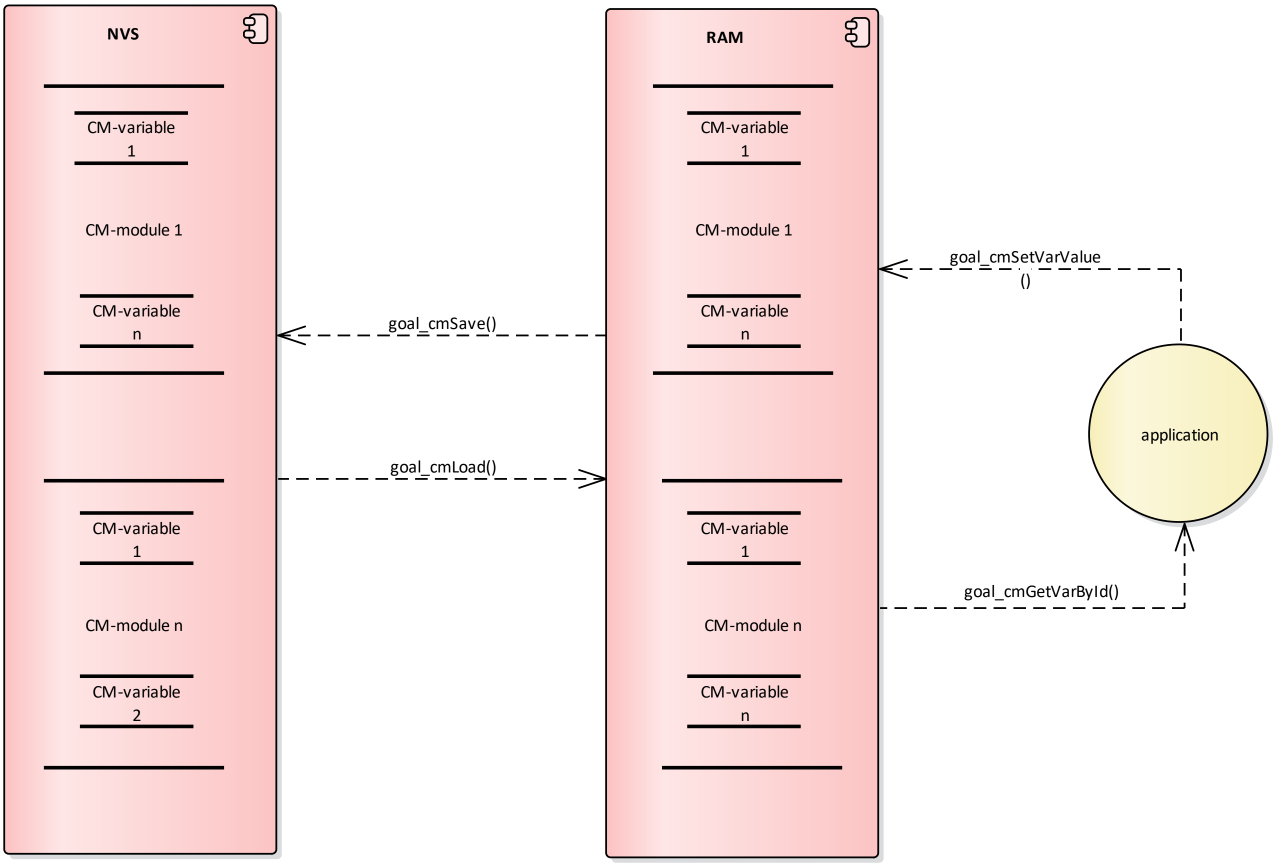


Figure 10: data structure and data flow of the Configuration Manager

Each CM-variable is uniquely identified by a CM-module-ID and a CM-variable-ID. The Configuration Manager allows to handle configuration variables of the CM-variable data types, see chapter 8.2 The CM-modules and CM-variables must be installed in state GOAL_FSA_INIT.

The configuration data in the NVM are extended by a CRC-sum to detect data errors. The applied CRC-algorithm is Fletcher-32 /Fletcher/.

The Configuration Manager differentiates between CM-variables with temporary and volatile values. CM-variables can be marked as temporary or stable by function goal_cmSetVarValue(). Temporary CM-variables can be manipulated after loading from the NVM via a callback function. Chapter 5.3.2 describes all callback functions of the Configuration Manager.

If there are changes at the interface of the Configuration Manager to the NVM or changes at the variable list, the configuration data are not loaded from NVM. Changes on interfaces are identifiable by the version number GOAL_CM_VERSION of the Configuration Manager in the file ...\goal\goal_cm.h.

It is possible to assign a name to each CM-variable. This possibility must be activated/deactivated by the compiler-define GOAL_CM_NAMES.

The Configuration Manager can be controlled via the command line interface, see chapter 5.3.5.

GOAL files:

goal_cm.[h,c], goal_cm_id.h, goal_cm_t.h, goal_cm_cli.c, cm/goal_cm_cm.[h,c]

example:

…\goal\appl\00410_goal\cfg_demo

5.3.1 Configuration

5.3.1.1 Compiler-defines

The following compiler-defines are available to configure the Configuration Manager:

GOAL_CM_NAMES:
0: CM-modules and CM-variables are identified by ID numbers (default)
1: CM-modules and CM-variables are identified by ID numbers and names

5.3.1.2 CM-variables

The following CM-variables are available for the Configuration Manager:

CM-Module-ID	GOAL_ID_CM
CM-variable-ID	0
CM-variable name	CM_CM_VAR_SAVE
Description	Each writing of any value to this CM-variable stores all CM-variables in the NVM.
CM data type	GOAL_CM_UINT8
Size	1 byte
Access	-
Default value	from NVS or 0

5.3.2 Callback functions

The Configuration Manager supports two kinds of callback functions:

- callback functions, which must be specified during implementation and
- callback functions, which can be specified during runtime

5.3.2.1 CM-variables based

During implementation callback functions

- for value validation and
- to inform the application about value changes

can be specified for each CM-variable. The specification of the callback functions is described in chapter 5.3.3. The callback functions themselves are described in the following tables. The names of the callback functions are application-specific.

Prototype	`GOAL_STATUS_T cbValidateFunc(uint32_t cmModId, uint32_t cmVarId, GOAL_CM_VAR_T *pVar, void *pNewData, uint32_t size)`	
Description	This callback function is used to validate new values for the specified CM-variable.	
Darameters	cmModId	number of the CM-module
	cmVarId	number of the CM-variable
	pVar	pointer to the entry in the CM-variable list for the CM-variable
	pNewData	new specified value for the CM-variable
	size	size of the CM-variable in byte
Return values	GOAL return status, see chapter 8.3	
Category	optional If a callback function is not available, specify NULL in the CM-variable list.	
Registration	by compilation	

Prototype	`GOAL_STATUS_T cbChangedFunc(uint32_t cmModId, uint32_t cmVarId, GOAL_CM_VAR_T *pVar)`	
Description	This callback function is used to inform other components about the changing of the value of the CM-variable.	
Parameters	cmModId	number of the CM-module
	cmVarId	number of the CM-variable
	pVar	pointer to the entry in the CM-variable list for the CM-variable
Return values	GOAL return status, see chapter 8.3	
Category	optional If a callback function is not available, specify NULL in the CM-variable list.	
Registration	by compilation	

5.3.2.2 *CM-module based*

During runtime callback functions

- for customer-specific loading of CM-variables from the NVM,
- for customer-specific saving of CM-variables to the NVM,
- to change values for temporary CM-variables after loading from the NVM by function goal_cmLoad()

can be configured for each CM-module by function goal_cmAddModule(). The callback functions are described in the following tables. The names of the callback functions are application-specific.

Prototype	`GOAL_STATUS_T cbLoadFunc(uint32_t cmModId, uint32_t cmVarId, GOAL_CM_VAR_T *pVar, uint32_t *pSize)`	
Description	This callback function is used to load a CM-variables from NVM customer-specific.	
Parameters	cmModId	number of the CM-module
	cmVarId	number of the CM-variable
	pVar	pointer to the entry in the CM-variable list for the CM-variable
	pSize	returns the current size of the CM-variable in byte
Return values	GOAL return status, see chapter 8.3	
Category	optional If not available, specify NULL in the call of goal_cmAddModule().	
Registration	during runtime about function goal_cmAddModule()	

Prototype	`GOAL_STATUS_T cbSaveFunc(uint32_t cmModId, uint32_t cmVarId, GOAL_CM_VAR_T *pVar)`	
Description	This callback function is used to save a CM-variables in the NVM customer-specific.	
Parameters	cmModId	number of the CM-module
	cmVarId	number of the CM-variable
	pVar	pointer to the entry in the CM-variable list for the CM-variable
	pSize	returns the current size of the CM-variable in byte
Return values	GOAL return status, see chapter 8.3	
Category	optional If not available, specify NULL in the call of goal_cmAddModule().	
Registration	during runtime about function goal_cmAddModule()	

Prototype	`GOAL_STATUS_T cbTmpsetFunc(uint32_t cmModId, uint32_t cmVarId, GOAL_CM_VAR_T *pVar, uint32_t *pNewSize)`
Description	This callback function allows to overwrite the value of the temporary CM-variable

	after loading from the NVM. If no callback function is specified, GOAL uses the default function goal_cmTmpSet() and clears the value to 0.	
Parameters	cmModId	number of the CM-module
	cmVarId	number of the CM-variable
	pVar	pointer to the entry in the CM-variable list for the CM-variable
	pNewSize	returns the current size of the CM-variable in byte, goal_cmTmpSet() returns 0
Return values	GOAL return status, see chapter 8.3	
Category	optional If not available, specify NULL in the call of goal_cmAddModule().	
Registration	during runtime about function goal_cmAddModule()	

5.3.3 Creating a CM-module and a variable list

The Configuration Manager provides a scheme for the creation of a CM-module and a list of CM-variables. It is recommended to use this scheme for application-specific CM-modules too.

1. For each CM-module a unique number is necessary.

   Example:
   ```
   #define APPL_CM_MOD_ID 0x00EE0000
   ```

2. The CM-variables, which shall be available via the Configuration Manager, must be specified and assigned to a CM-variable-ID. Because the CM-variable-ID is also used as list index, the counting has to start with 0 and must be consecutively. Create a enum for the CM-variable-IDs to access the configuration variable by a symbolic name.
   Example:

   ```
   typedef enum {
       APPL_CM_VAR_1,
       APPL_CM_VAR_2
   } APPL_CM_VARS_ID_T;
   ```

3. The CM-variables are listed with the following properties:
   - CM-variable-ID,
   - CM-variable data types of the CM-variable,
   - maximal size of the CM-variable in byte,
   - a callback function for the validation of the written value,
   - a callback function to inform the application about the change of the variable's value and
   - the name of the CM-variable, if naming is switched on by the compiler-define GOAL_CM_NAMES.

   Create a table with the properties for all CM-variables assigned to the CM-module. Each line of the table represents one CM-variable according to the structure GOAL_CM_VARENTRY_T. This structure contains the properties of the CM-variable and pointer references for the internal handling. Please set the internal pointer references to NULL. If no callback functions are available for validation and/or change reports, set the references also to NULL.

Example 8: for GOAL_CM_NAMES = 0 with callback functions

```
static GOAL_CM_VARENTRY_T applCmVars[] = { \
    {APPL_CM_VAR_1, GOAL_CM_UINT8, 1, NULL, applValidateFct, applChangeFct, NULL,
     NULL
    },
    {APPL_CM_VAR_2, GOAL_CM_UINT32, 4, NULL, applValidateFct, applChangeFct, NULL,
     NULL
    }
}
```

Example 9: for GOAL_CM_NAMES = 0 without callback functions

```
static GOAL_CM_VARENTRY_T applCmVars[] = { \
    {APPL_CM_VAR_1, GOAL_CM_UINT8, 1, NULL, NULL, NULL, NULL, NULL},
    {APPL_CM_VAR_2, GOAL_CM_UINT32, 4, NULL, NULL, NULL, NULL, NULL}
}
```

4. Now the created CM-module can be integrated in the code as described in chapter 5.3.6.1.

5.3.4 Virtual Variables

GOAL CM supports virtual variables, which only are stored in memory and not written to the non volatile storage.
Virtual variables are created in stage GOAL_STAGE_CM_MOD_ADD using the function goal_cmRegVarVirtual.

Prototype	`GOAL_STATUS_T goal_cmRegVarVirtual(uint32_t modId, unt32_t varId, GOAL_CM_DATATYPE_T type, uint32_t sizeMax, goal_cm_validate validate, goal_cm_changed changed);`	
Description	Register a virtual cm variable	
Parameters	modId	Module ID
	varId	Variable ID
	type	CM datatype
	sizeMax	Maximum size of variable
	goal_cm_validate	Validation callback or NULL
	goal_cm_changed	Modification callback or NULL
Return values	GOAL return status, see chapter 8.3	
Category	Optional	
Condition	-	

```
/* add virtual variables */
```

```
if (GOAL_RES_OK(res)) {
    res = goal_cmRegVarVirtual(
        2,                           /* module Id */
        CM_CM_VAR_SAVE,              /* variable Id */
        GOAL_CM_UINT8,               /* type */
        1,                           /* size */
        NULL,                              /* validation callback */
        goal_cmCmSave                /* modification callback*/
    );
}
```

Code 3 create virtual cm variable

5.3.5 Command line interface

Command	`cm set <modId> <varId> <newVal>`	
Description	Sets the value of an existing variable identified by the CM-module-ID and CM-variable-ID in the Configuration Manager.	
Parameter	`<modId>`	number of the CM-module
	`<varId>`	number of the CM-variable within the CM-module, value range 00000001h – FFFFFFFFh
	`<newVal>`	new value Integer values are entered with an optional sign. String values begin and end with “-character.

Command	`cm show [<modId> <varId>]`	
Description	Shows the value of the variable identified by the CM-module-ID and CM-variable-id or all CM-variables. If no IDs are given all CM-variables of all CM-modules are printed out to the command line interface.	
Parameter	`<modId>`	number of the CM-module, value range 00000001h – FFFFFFFFh
	`<varId>`	number of the CM-variable within the CM-module, value range 00000001h – FFFFFFFFh

5.3.6 Implementation guidelines

5.3.6.1 Creating a new CM-module

1. Specify a unique CM-module-ID number, see chapter 5.3.3.
2. Specify the list of CM-variables, see chapter 5.3.3.

3. Create a variable for the CM-module-ID.

```
GOAL_CM_MODDEF_T cmMod;
cmMod.modId = APPL_CM_MOD_ID;
```

```
Register the CM-variables by function goal_cmRegModule() in the state
GOAL_FSA_INIT_APPL, stage GOAL_STAGE_CM_MOD_REG.

GOAL_STATUS_T res;        /* GOAL return status */
res = goal_cmRegModule(applCmVars);
```

4. In stage GOAL_STAGE_CM_MOD_ADD add the CM-variable list to the CM-module by function goal_cmAddModule() in the state GOAL_FSA_INIT_APPL and do not specify callback functions for customer-specific nonvolatile load and save and the modification of temporary CM-variables after loading from NVM.

```
if (GOAL_RES_OK(res)) {
    res = goal_cmAddModule(&cmMod, applCmVars, NULL, NULL, NULL);
}
```

Write a value to a CM-variable by function goal_cmSetVarValue().

```
uint32_t val = 0x11223344;
if (GOAL_RES_OK(res)) {
    res = goal_cmSetVarValue(APPL_CM_MOD_ID, APPL_CM_VAR_2,
        (void *)&val, 4, GOAL_FALSE, NULL);
}
```

5. Read the value of a CM-variable about function goal_cmGetVarById().

```
GOAL_CM_VAR_T *pEntry;
if (GOAL_RES_OK(res)) {
    res = goal_cmGetVarById(APPL_CM_MOD_ID, APPL_CM_VAR_2, &pEntry);
    if (GOAL_RES_OK(res)) {
        val = GOAL_CM_VAR_UINT32(pEntry);
    }
}
```

5.3.6.2 Add a new CM-variable to a CM-module

1. Add the CM-variable to the variable list, see chapter 5.3.3.

2. Create a variable for the CM-module-ID.

```
GOAL_CM_MODDEF_T cmMod;
cmMod.modId = APPL_CM_MOD_ID;
```

3. Register the CM-module by function goal_cmRegModule() in the state GOAL_FSA_INIT_APPL.

```
GOAL_STATUS_T res;        /* GOAL return status */
res = goal_cmRegModule(applCmVars);
```

4. Add the CM-variable list to the CM-module by function goal_cmAddModule() in the state GOAL_FSA_INIT_APPL and do not specify callback functions for customer-specific nonvolatile load and save and the modification of temporary CM-variables after loading from NVM.

```
if (GOAL_RES_OK(res)) {
```

```
    res = goal_cmAddModule(&cmMod, applCmVars, NULL, NULL, NULL);
}
```

5. Write a value to a CM-variable by the function goal_cmSetVarValue().

```
uint8_t val = 0xA5;
if (GOAL_RES_OK(res)) {
    res = goal_cmSetVarValue(APPL_CM_MOD_ID, APPL_CM_VAR_1,
           (void *)&val, 1, GOAL_FALSE, NULL);
}
```

6. Read the value of a CM-variable by the function goal_cmGetVarById().

```
GOAL_CM_VAR_T *pEntry;
if (GOAL_RES_OK(res)) {
    res = goal_cmGetVarById(APPL_CM_MOD_ID, APPL_CM_VAR_1, &pEntry);
    if (GOAL_RES_OK(res)) {
        val = GOAL_CM_VAR_UINT8(pEntry);
    }
}
```

5.3.6.3 Load and save CM-variables nonvolatile

1. Create a variable for the CM-module-ID.

```
GOAL_CM_MODDEF_T cmMod;
cmMod.modId = APPL_CM_MOD_ID;
```

2. Register the CM-module by function goal_cmRegModule() in the state GOAL_FSA_INIT_APPL.

```
GOAL_STATUS_T res;        /* GOAL return status */
res = goal_cmRegModule(applCmVars);
```

3. Add the CM-variable list to the CM-module by function goal_cmAddModule() in the state GOAL_FSA_INIT_APPL and do not specify callback functions for customer-specific nonvolatile load and save and the modification of temporary CM-variables after loading from NVM.

```
if (GOAL_RES_OK(res)) {
    res = goal_cmAddModule(&cmMod, applCmVars, NULL, NULL, NULL);
}
```

4. Load all CM-variables from NVM by function goal_cmLoad().

```
if (GOAL_RES_OK(res)) {
    res = goal_cmLoad();
}
```

5. Write a value to a CM-variable by the function goal_cmSetVarValue().

```
uint8_t val = 0xA5;
if (GOAL_RES_OK(res)) {
    res = goal_cmSetVarValue(APPL_CM_MOD_ID, APPL_CM_VAR_1,
        (void *)&val, 1, GOAL_FALSE, NULL);
}
```

6. Save all CM-variables nonvolatile by function goal_cmSave().

```
if (GOAL_RES_OK(res)) {
    res = goal_cmSave();
}
```

5.4 Generic Ethernet Frame Handler (goal_eth)

This GOAL core module provides functions to send and receive Ethernet frames, see Figure 11.

The Ethernet Frame Handler receives all Ethernet frames. The frame processing load can be reduced by activation of the MAC address filtering by the compiler-define GOAL_CONFIG_MAC_ADDR_FILTER. Then only all broadcast/multicast and the own unicast ethernet frames pass the MAC filter and are received. This is only a software filter, which drop packets not directed to the device.

The Ethernet Frame Handler identifies received ethernet frames on base of the

- MAC address
- the Ether Type

The values for the Ether Type are standardized in IEEE 802.3. GOAL supports the following Ether Types:

- 0800h: IP Internet Protocol, version 4 (IPv4)
- 0806h: Address Resolution Protocol (ARP)
- 8100h: VLAN Tag

Other Ether Types are registeres by additional software components, such as the PNIO communication stack,

The Ethernet Frame Handler accepts all ethernet frames if the Ether Type is set to GOAL_ETH_ETHERTYPE_ANY.
The kind of the identification is configured by function goal_ethProtoAdd() or goal_ethProtoAddPos().

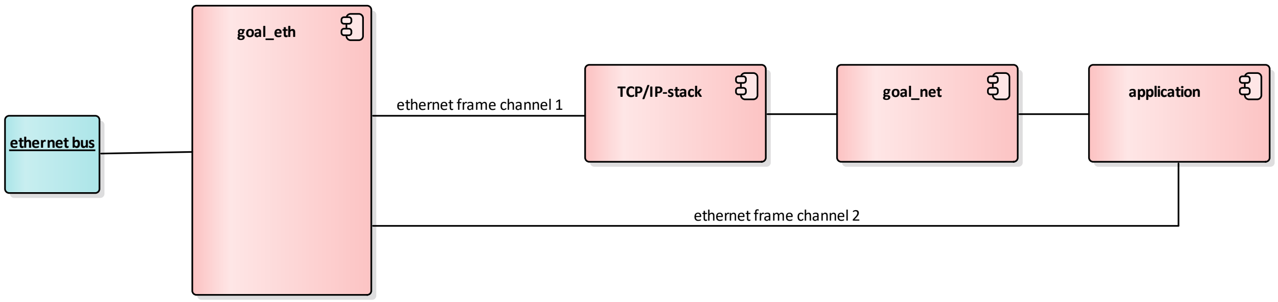


Figure 11: ethernet frame handler as part of the GOAL system

The ethernet frames can be divided in frames with low and high priority. The priority is also specified by function goal_ethProtoAdd() or goal_ethProtoAddPos(). The type of identification and the priority determine the handling of received frames, see Figure 12.

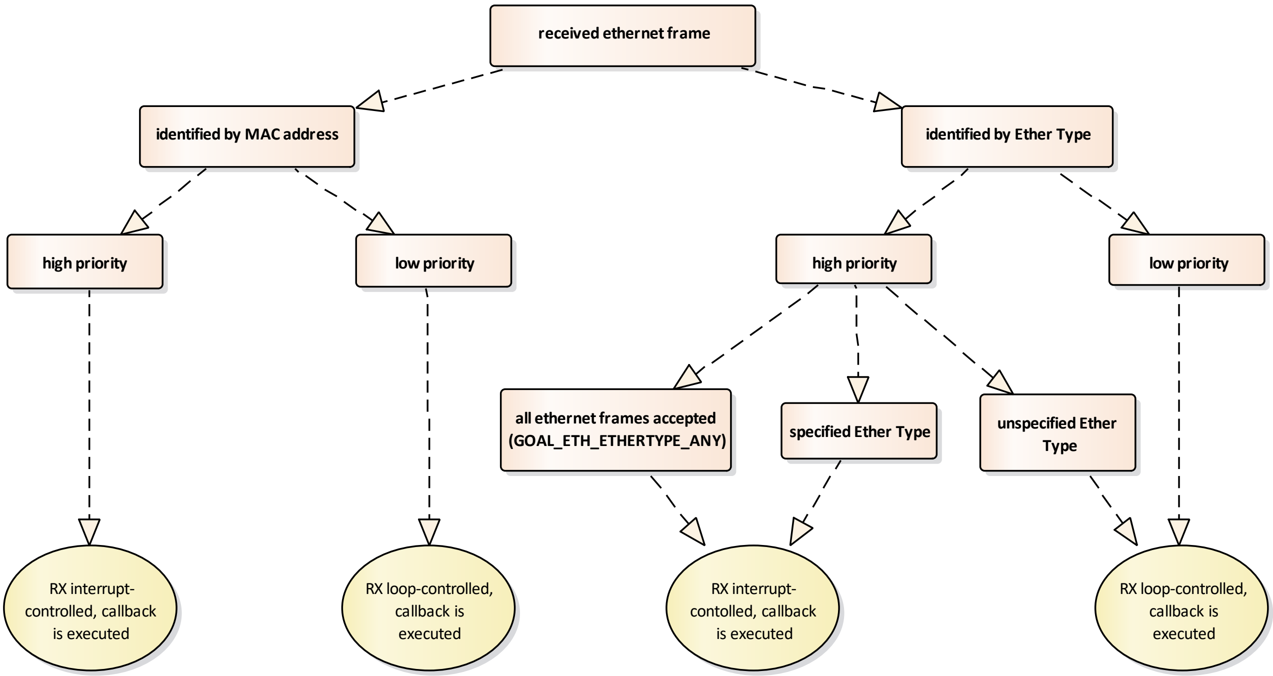


Figure 12: RX ethernet frame handling

During the interrupt-controlled receipt the callback function specified by function goal_ethProtoAdd() or goal_ethProtoAddPos() is called immediately.
For the loop-controlled handling the received message is stored internal and the callback function is called in the GOAL loop. The Ethernet Frame handler registers the function goal_ethLoop() for this purpose.

Ethernet controllers provide more or less possibilities to analyze the ethernet communication by counting of events represented as statistics, see chapter 5.4.9.

The generic Ethernet Frame Handler can be controlled via the command line interface, see chapter 5.4.9.

GOAL files:

goal_eth.[h,c]

example:

not available

5.4.1 Configuration

5.4.1.1 Compiler-defines

The following compiler-defines are available to configure the generic Ethernet Frame Handler:

GOAL_CONFIG_ETHERNET:
0: generic Ethernet Frame Handler is disabled (default)
1: generic Ethernet Frame Handler is enabled

GOAL_TARGET_ETH_PORT_COUNT:
number of external ports (default: platform-specific)

GOAL_CONFIG_MAC_ADDR_FILTER:
0: MAC address filtering disabled (default)
1: MAC address filtering enabled

GOAL_ETH_NAMES:
0: names for ethernet commands are not available (default)
1: names for ethernet command available

GOAL_CONFIG_ETH_STATS:
0: support of ethernet statistics is disabled (default)
1: support of ethernet statistics is enabled

GOAL_CONFIG_ETH_STATS_NAMES:
0: short description of ethernet statistic is not available (default)
1: short description of ethernet statistics is available

GOAL_CONFIG_TDMA:
0: time division multiple access disabled (default)
1: time division multiple access enabled

The following compiler-defines are available for debug purposes:

GOAL_CONFIG_LOGGING_TARGET_SYSLOG:
0: no output of ethernet frames (default)
1: output of ethernet frames

GOAL_CONFIG_LOGGING:
0: output of warnings disabled (default)
1: output of warnings enabled

5.4.1.2 CM-variables

The following CM-variables are available to configure the Configuration Manager:

CM-Module-ID	GOAL_ID_ETH
CM-variable-ID	0
CM-variable name	ETH_CM_VAR_MAC
Description	MAC address
CM data type	GOAL_CM_GENERIC
Size	6 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_ETH
CM-variable-ID	1
CM-variable name	ETH_CM_VAR_LINK
Description	mask for the link state of the ethernet port
CM data type	GOAL_CM_UINT32
Size	4 bytes
Default vaue	from NVS or 0

CM-Module-ID	GOAL_ID_ETH
CM-variable-ID	2
CM-variable name	ETH_CM_VAR_SPEED
Description	mask for the speed of the ethernet port
CM data type	GOAL_CM_UINT32
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_ETH
CM-variable-ID	3
CM-variable name	ETH_CM_VAR_DUPLEX
Description	mask for duplex property of the ethernet port
CM data type	GOAL_CM_UINT32
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_ETH
CM-variable-ID	4

CM-variable name	ETH_CM_VAR_PORTCNT
Description	number of ethernet ports
CM data type	GOAL_CM_UINT32
Size	4 bytes
Default value	from NVS or 0

The generic Ethernet Frame Handler uses GOAL queues internally. The size of these queues can be optimized for the current platform. The configuration is described in chapter 5.12.4.

5.4.2 Callback functions

The ethernet frame handler supports a callback function:

- for the receipt of ethernet frames and
- to inform the application about the changed state of the ethernet port.

The names of the callback functions are application-specific.

Prototype	`GOAL_STATUS_T cbEthFrameReceivedFunc(GOAL_BUFFER_T **ppBuf)`	
Description	This callback function is used to deal with the received ethernet frame.	
Parameters	ppBuf	pointer at the buffer containing the received ethernet frame
Return values	GOAL return status, see chapter 8.3	
Category	mandatory, if the Ethernet Frame Handler is used	
Registration	during runtime via function goal_ethProtoAdd()	

Prototype	`void cbEthPortChangedFunc(GOAL_ETH_PORT_T id, uint32_t maskChg, struct GOAL_ETH_PORT_STATE_T *pState)`	
Description	This callback function is called to inform the application about the changed state of the ethernet port.	
Parameters	id	number of the ethernet port
	maskChg	mask for the changed state bits
	pState	new state of the ethernet port
Return values	None	
Category	Optional	
Registration	during runtime via function goal_ethPortStateCbReg()	

5.4.3 Platform API

GOAL requires the following indication function to communicate via ethernet:

Prototype	`GOAL_STATUS_T goal_targetEthInit(void)`
Description	This indication function initializes the ethernet interface for the selected platform. This function is called in the state GOAL_FSA_INIT_GOAL in stage GOAL_STAGE_TARGET_PRE.
Parameters	none
Return values	GOAL return status, see chapter 8.3
Category	mandatory, if the Ethernet Frame Handler is used
Condition	compiler-define GOAL_CONFIG_ETHERNET must be set to 1

Prototype	`GOAL_STATUS_T goal_targetEthCmd(GOAL_ETH_CMD_T cmd, GOAL_BOOL_T wrFlag, uint32_t port, void *pArg)`	
Description	This indication function executes an ethernet command.	
Parameters	cmd	ethernet command
	wrFlag	access direction • GOAL_TRUE: execute the set option of the ethernet command • GOAL_FALSE: execute the read option of the ethernet command
	port	number of ethernet port
	pArg	argument to the ethernet command
Return values	GOAL return status, see chapter 8.3	
Category	mandatory, if the Ethernet Frame Handler is used	
Condition	compiler-define GOAL_CONFIG_ETHERNET must be set to 1	

Prototype	`GOAL_STATUS_T goal_targetGetMacAddr(GOAL_ETH_PORT_T portIdx, char *pMacAddr)`	
Description	This indication function returns the MAC address of the ethernet interface of the specified board.	
Parameters	portIdx	number of the ethernet port
	pMacAddr	buffer to return the MAC address
Return values	GOAL return status, see chapter 8.3	
Category	optional	
Condition	compiler-define GOAL_CONFIG_ETHERNET must be set to 1	

Prototype	`void goal_targetEthSend(void)`
Description	This indication function transmits an ethernet frame from the internal transmit GOAL queue.

Parameters	none
Return values	none
Category	mandatory, if the Ethernet Frame Handler is used
Condition	compiler-define GOAL_CONFIG_ETHERNET must be set to 1

5.4.4 Ethernet interface

GOAL makes a general interface available to configure the ethernet interface and to get state information about the ethernet interface, e.g. the switch or PHY. There are two possibilities to access to the configuration setting or the state information:

- via special functions of the Ethernet Frame Handler or
- via the ethernet command and the function goal_ethCmd().

The implementation and the support of the ethernet commands depend on the platform. The platform-specific details are described in the GOAL Platform Manual.

GOAL provides the following commands for the configuration of the ethernet interface:

Ethernet command	GOAL_ETH_CMD_AUTONEG_PROGRESS
Description	get the state of the auto-negotiation process: • GOAL_ETH_AUTONEG_INPROGRESS, • GOAL_ETH_AUTONEG_FAIL_ALL, • GOAL_ETH_AUTONEG_FAIL_DUPLEX, • GOAL_ETH_AUTONEG_DONE, • GOAL_ETH_AUTONEG_SKIPPED
Special set function	--
Special get function	goal_ethAutonegProgressGet()

Ethernet command	GOAL_ETH_CMD_AUTONEG
Description	set or get the behavior for the auto-negotiation: • GOAL_ETH_AUTONEG_ON, • GOAL_ETH_AUTONEG_OFF
Special set function	goal_ethAutonegSet()
Special get function	goal_ethAutonegGet()

Ethernet command	GOAL_ETH_CMD_AUTONEG_RESTART
Description	restart the auto-negotiation process
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_DUPLEX
Description	set or get the transfer mode: • GOAL_ETH_DUPLEX_HALF, • GOAL_ETH_DUPLEX_FULL
Special set function	goal_ethLinkDuplexSet()
Special get function	goal_ethLinkDuplexGet()

Ethernet command	GOAL_ETH_CMD_HW_FAULT
Description	get an indicator for the last hardware fault
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_SPEED
Description	set or get the rate of transfer: GOAL_ETH_SPEED_10, GOAL_ETH_SPEED_100 or GOAL_ETH_SPEED_1000 Mbit/s
Special set function	goal_ethLinkSpeedSet()
Special get function	goal_ethLinkSpeedGet()

Ethernet command	GOAL_ETH_CMD_SPEED_MAX
Description	get the maximal allowed rate of transfer:

	- GOAL_ETH_SPEED_10 Mbit/s, - GOAL_ETH_SPEED_100 Mbit/s, - GOAL_ETH_SPEED_1000 Mbit/s
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_LINK_STATE
Description	get the current state of the ethernet connection: - GOAL_ETH_STATE_UP, - GOAL_ETH_STATE_DOWN
Special set function	--
Special get function	goal_ethLinkStateGet()

Ethernet command	GOAL_ETH_CMD_PORT_STATE
Description	switch on/off the ethernet port or get the current state of the ethernet port: - GOAL_ETH_STATE_UP, - GOAL_ETH_STATE_DOWN
Special set function	goal_ethPortStateSet()
Special get function	goal_ethPortStateGet()

Ethernet command	GOAL_ETH_CMD_LINK_CAPABILITIES
Description	get the supported transfer mode and transfer rate The return value has data type uint32_t and is bit-coded. The bits have the following meaning for bit value 1: - bit 0: 10 Mbit/s in half-duplex mode is supported - bit 1: 10 Mbit/s in full-duplex mode is supported - bit 2: 100 Mbit/s in half-duplex mode is supported - bit 3: 100 Mbit/s in full-duplex mode is supported - bit 4: 1000 Mbit/s in half-duplex mode is supported - bit 5: 1000 Mbit/s in full-duplex mode is supported
Special set	--

function	
Special get function	--

Ethernet command	GOAL_ETH_CMD_AUTONEG_ADVERTISMENT
Description	set or get the list for transfer rate and transfer mode for the auto-negotiation process The value has the data type uint32_t and is bit-coded. The bits have the following meaning for bit value 1: • bit 0: 10 Mbit/s in half-duplex mode is used • bit 1: 10 Mbit/s in full-duplex mode is used • bit 2: 100 Mbit/s in half-duplex mode is used • bit 3: 100 Mbit/s in full-duplex mode is used • bit 4: 1000 Mbit/s in half-duplex mode is used • bit 5: 1000 Mbit/s in full-duplex mode is used
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_PORT_ADMIN_STATE
Description	get the current state of the ethernet port: • GOAL_ETH_STATE_UP, • GOAL_ETH_STATE_DOWN
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_LED_LINK
Description	set or get the PHY link LED state
Special set function	--
Special get function	--

Ethernet command	GOAL_ETH_CMD_PORT_COUNT
Description	get the number of installed ethernet ports
Special set function	--
Special get function	--

5.4.5 VLAN

GOAL makes a general interface available to configure the VLAN capabilities of the underlying switch. The access to the configuration setting or the state information is realized via ethernet commands and the function goal_ethCmd(). The implementation and the support of the ethernet commands depend on the platform.

GOAL provides the following ethernet commands for the VLAN capabilities:

Ethernet command	Description
GOAL_ETH_CMD_VLAN_MODE_IN	set or get the input mode of the VLAN processing
GOAL_ETH_CMD_VLAN_MODE_OUT	set or get the output mode of the VLAN processing
GOAL_ETH_CMD_VLAN_DEF	set or get the default VLAN-ID and priority for a port
GOAL_ETH_CMD_VLAN_PORT_ADD	adds a port as a member of the given VLAN-ID
GOAL_ETH_CMD_VLAN_PORT_REM	removes a port as a member from the given VLAN-ID
GOAL_ETH_CMD_VLAN_TABLE_CNT	get the VLAN table entry count
GOAL_ETH_CMD_VLAN_TABLE_GET	shows the entries of the VLAN table
GOAL_ETH_CMD_VLAN_VERIFY	enables/disables the VLAN domain verification for the given port
GOAL_ETH_CMD_VLAN_DISCUNK	enabled/disables the discarding of frames with unknown VLAN-IDs

5.4.6 MAC table

The MAC table subgroup provides an interface to the MAC table settings and allows to access to specific MAC table entries.

GOAL provides the following ethernet commands for the handling of AC table settings:

Ethernet command	Description
GOAL_ETH_CMD_MACTAB_CONF	enables/disables the given feature of the MAC table: • `learning`: Automatic MAC address learning • `ageing`: MAC address ageing for dynamic entries

Ethernet command	Description
	• `migration`: Allows the migration of MAC addresses between ports • `discunknown`: Discard frames with unknown destination address • `pervlan`: Learn MAC addresses per VLAN allowing the same MAC address in different VLANs
GOAL_ETH_CMD_MACTAB_SET	set an entry in the MAC table
GOAL_ETH_CMD_MACTAB_GET	get an entry from the MAC table
GOAL_ETH_CMD_MACTAB_CLR	clear MAC table

5.4.7 Port settings

Ethernet command	Description
GOAL_ETH_CMD_PORT_FWD_ADD	add port to forward table
GOAL_ETH_CMD_PORT_FWD_DEL	delete port from forward table
GOAL_ETH_CMD_PORT_AUTH	set/get port authorization
GOAL_ETH_CMD_PORT_CTRL_DIR	set/get port controlled directions
GOAL_ETH_CMD_PORT_EAPOL_ENABLE	set/get port EAPOL frame reception mode

5.4.8 QoS settings

Ethernet command	Description
GOAL_ETH_CMD_QOS_MODE	set/get QoS mapping type
GOAL_ETH_CMD_QOS_PRIO_VLAN	set/get QoS VLAN priority
GOAL_ETH_CMD_QOS_PRIO_IP	set/get QoS IP priority
GOAL_ETH_CMD_QOS_PRIO_TYPE	set/get QoS Ethertype priority

5.4.9 Implementation guidelines

5.4.9.1 Configure speed rate by special command

1. Set the transfer rate to 100 Mbit/s for the ethernet port with the number portNum:

```
GOAL_STATUS_T res;          /* GOAL return status */
res = goal_ethLinkSpeedSet(portNum, GOAL_ETH_SPEED_100);
```

5.4.9.2 Restart the autonegotiation with goal_ethCmd()

1. Reset the autonegotiation for the ethernet port with the number portNum:

```
GOAL_STATUS_T res;          /* GOAL return status */
res = goal_ethCmd(GOAL_ETH_CMD_AUTONEG_RESTART, GOAL_TRUE, portNum, NULL);
```

5.4.9.3 Send and receive ethernet frames

Ethernet frames shall be send from the application directly, see Figure 11 ethernet channel2:

1. Create a callback function to handle received ethernet frames application-specific:

```
GOAL_STATUS_T cbEthFrameReceivedFunc(GOAL_BUFFER_T **ppBuf) {
    …
}
```

2. Register the callback function for the receipt of IPv4 ethernet frames with a high priority:

```
GOAL_STATUS_T res;          /* GOAL return status */
res = goal_ethProtoAdd(GOAL_TRUE, GOAL_ETH_ETHERTYPE_IPV4, NULL,
    cbEthFrameReceivedFunc);
```

3. If an ethernet frame was received, the callback function cbEthFrameReceivedFunc() is called and the application can handle the ethernet frame.

4. Send an ethernet frame:

```
GOAL_BUFFER_T *pBuf = NULL;

/* get buffer */
goal_ethGetNetBuf(&pBuf);

/*  build frame by writing to pBuf-> ptrData */
GOAL_MEMCPY(pBuf->ptrData, buf, len);

/* write frame length to pBuf->dataLen  */
pBuf->dataLen = len;

/* specify egress port */
pBuf->netPort = GOAL_ETH_PORT_HOST;

goal_ethSend(&pBuf, GOAL_NET_TX_LOW);

/* pBuf was released by goal_ethSend */
```

5.5 Command line interface

5.5.1 Naming and parameter conventions

5.5.2 Actions

Every command executes a so-called action describing the functionality of the command. The following table provides an overview of actions that may occur:

Action	Function	Example
set	Set parameter values	`eth vlan verify set 1 on`
show	Show parameter values. The action may accept one or more optional parameters	`rstp port show`
help	Show a help string for specific (sub)command	`rstp port help`
add	Adding a value to a set of values e.g. adding a port to a port map.	`eth mactab mac add 00:11:22:33:44:55 1`
rem	Removing a value from a set of values e.g. removing a port from a port map.	`eth mactab mac rem 00:11:22:33:44:55 1`

Not all commands implement all actions.

5.5.3 Command parameter conventions

5.5.3.1 Integer values

Integer values are currently only accepted with a base of 10 and may optionally contain a sign. As an example, the following command sets the port membership of port 1 to VLAN 1024:

```
$ eth vlan port add 1 1024
```

5.5.3.2 Strings

Strings are started and ended with a “-character. As an example, the following command sets the value of config variable 0-1 to value “example”

```
$ cm set 0 1 “example”
```

5.5.3.3 Ports

Ports are entered as integer values starting with 0 up to max. port number + 1. Max. port number +1 represents the management port. A 5 port switch provides ports 0 – 3 (external ports) and port 4 as management port.
For example, the following commands set the default VLAN tag for port 1 to 1024 with prio 7:

```
$ eth vlan default set 1 1024 7
```

5.5.3.4 MAC addresses

MAC addresses are given in the format `xx:xx:xx:xx:xx:xx` where `xx` stands for a two char hex number. For example, the following command adds port 3 to MAC address 00:11:22:33:44:55

```
$ eth mactab mac add 00:11:22:33:44:55 3
```

5.5.3.5 IP addresses

IP addresses are given in the format `xxx.xxx.xxx.xxx` where `xxx` stands for a one- to three-digit decimal number. For example, the following command sets the IP address, netmask and gateway for the TCP/IP stack:

```
$ net ip set 192.168.1.133 255.255.255.0 0.0.0.0
```

5.5.4 Ethernet Interface

The eth command provides an interface to Ethernet interface including access to VLAN configuration, Ethernet statistics aso.

5.5.5 VLAN

The VLAN subgroup provides an interface for configuring the VLAN capabilities of the underlying switch.

Command	`eth vlan mode in set <port> <ptrover|replace|tag|disable>`	
Description	Sets the input mode of the VLAN processing.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<ptrover|replace|tag|disable>`	The VLAN input processing mode to set: • `ptrover`: Passthrough/Overwrite • `replace`: If untagged, add the tag, if single tagged, overwrite the tag. • `tag`: Insert a tag always • `disable`: Disable input processing

Command	`eth vlan mode in show [port]`
Description	Shows the input of the given port or all ports if no port is given

Parameter	`[port]`	The optional port where the input mode shall be shown.

Command	`eth vlan mode out set <port> <tagthr|domain|strip|disable>`	
Description	Sets the output mode of the VLAN processing.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<tagthr|domain|strip|disable>`	The VLAN input processing mode to set: • `tagthr`: Tag thru • `domain`: Transparent mode • `strip`: Strip (outer) tag • `disable`: Disable output processing

Command	`eth vlan mode out show [port]`	
Description	Shows the output processing mode of the given port or all ports if no port is given	
Parameter	`[port]`	The optional port where the output mode shall be shown.

Command	`eth vlan port add <port> <vlanid>`	
Description	Adds a port as a member of the given VLAN id.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<vlanid>`	The VLAN id where the port shall become a member.

Command	`eth vlan port rem <port> <vlanid>`	
Description	Removes a port as a member from the given VLAN id.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<vlanid>`	The VLAN id where the port shall be removed from.

Command	`eth vlan table show`
Description	Shows the entries of the VLAN table.
Parameter	`None`

Command	`eth vlan default set <port> <vlanid> <prio>`	
Description	Sets the default VLAN id and priority for a port.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<vlanid>`	The default VLAN id for the port.
	`<prio>`	The priority ranging from 0 – 7.

Command	`eth vlan default show [port]`	
Description	Shows the default VLAN settings of the given port or all ports if no port is given	
Parameter	`[port]`	The optional port where the default VLAN settings shall be shown.

Command	`eth vlan verify set <port> <on|off>`	
Description	Enables/disables the VLAN domain verification for the given port.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<on|off>`	• `on` – enable verification • `off` – disable verification

Command	`eth vlan verify show [port]`	
Description	Shows the VLAN verification settings of the given port or all ports if no port is given	
Parameter	`[port]`	The optional port where the VLAN verification settings shall be shown.

Command	`eth vlan discunknown set <port> <on|off>`	
Description	Enabled/disables the discarding of frames with unknown VLAN ids.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<on|off>`	• `on` – enable discarding • `off` – disable discarding

Command	`eth vlan discunknown show [port]`	
Description	Shows the unknown VLAN discarding settings of the given port or all ports if no port is given	
Parameter	`[port]`	The optional port where the VLAN discarding settings shall be shown.

5.5.6 MAC table

The MAC table subgroup provides an interface to the MAC table settings and allows to access specific MAC table entries.

Command	`eth mactab conf set <ageing|migration|discunknown|pervlan> <on|off>`	
Description	Enabled/disables the given feature of the the MAC table.	
Parameter	`<learning|ageing|migration|discunknown|pervlan>`	The feature setting to change: • `learning`: Automatic MAC address learning • `ageing`: MAC address ageing for dynamic entries • `migration`: Allows the migration

		of MAC addresses between ports • `discunknown`: Discard frames with unknown destination address • `pervlan`: Learn MAC addresses per VLAN allowing the same MAC address in different VLANs
	`<on|off>`	• `on` – enable feature • `off` – disable feature

Command	`eth mactab conf show`
Description	Shows the state of the different MAC table configuration settings.
Parameter	None

Command	`eth mactab mac add <mac> <port>`	
Description	Adds the given port to the port map of the given MAC address. If the MAC address is not yet in the table, it is added as a static MAC address. Both, unicast and multicast MAC addresses are accepted.	
Parameter	`<mac>`	The MAC address where the port shall be added to. The address is given in the format `xx:xx:xx:xx:xx:xx`
	`<port>`	The port as number starting from 0 for the first port.

Command	`eth mactab mac rem <mac> <port>`	
Description	Removes the given port from the port map of the given MAC address. If the MAC address does not contain any more ports after command execution, it is removed from the MAC table. Both, unicast and multicast MAC addresses are accepted.	
Parameter	`<mac>`	The MAC address where the port shall be removed from. The address is given in the format `xx:xx:xx:xx:xx:xx`
	`<port>`	The port as number starting from 0 for the first port.

Command	`eth mactab mac show <mac>`	
Description	Shows the port map for the given MAC address.	
Parameter	`<mac>`	The MAC address where the port map shall be shown. The address is given in the format `xx:xx:xx:xx:xx:xx`

Command	`eth mactab mac clear <static|dynamic|all>`
Description	Deletes the MAC table.

Parameter	`<static|dynamic|all>`	The following part of the MAC table is cleared: • `static`: static • `dynamic`: dynamic • `all`: complete

5.5.7 Denial of Service Prevention

This command group provides an interface to TX as well as broadcast and multicast rate limiting.

Command	`eth dos txrate set <port> <limit>`	
Description	Sets the maximum allowed TX rate in percent.	
Parameter	`<port>`	The port as number starting from 0 for the first port.
	`<limit>`	The max. allowed TX rate in percent.

Command	`eth dos txrate show [port]`	
Description	Sets the maximum allowed TX rate in percent for the given port. If no port is given, the TX rates for all ports are shown.	
Parameter	`[port]`	The optional port as number starting from 0 for the first port where the TX rate shall be shown.

Command	`eth dos timebase set <timebase>`	
Description	Sets the time frame for broadcast/multicast rate limiting in ms. A timebase of 0 disables the rate limiting.	
Parameter	`<timebase>`	The time base in ms.

Command	`eth dos timebase show`
Description	Shows the time frame for broadcast/multicast rate limiting in ms. A timebase of 0 means that rate limiting is disabled.
Parameter	None

Command	`eth dos mlimit set <limit>`	
Description	Sets the rate limiting for multicast frames. The limit is interpreted as <limit> per <timebase>. The time base is set per `eth dos timebase set` command.	
Parameter	`<limit>`	The limit in number of frames.

Command	`eth dos mlimit show`
Description	Shows the rate limiting for multicast frames. The limit is interpreted as <limit> per <timebase>.
Parameter	None

Command	`eth dos blimit set <limit>`
Description	Sets the rate limiting for broadcast frames. The limit is interpreted as <limit> per

	<timebase>. The time base is set per `eth dos timebase set` command.	
Parameter	`<limit>`	The limit in number of frames.

Command	`eth dos blimit show`	
Description	Shows the rate limiting for broadccast frames. The limit is interpreted as <limit> per <timebase>.	
Parameter	None	

5.5.8 Port settings

Command	`eth port link show [port]`	
Description	Shows the link state of the given port. If no port is given, link state of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port adstate set <port> <on|off>`	
Description	Sets the admin state of the given port.	
Parameter	`<port>`	The port as number starting from 0 for the first port.
	`<on|off>`	Admin state of the port: • `on`: Port enabled. • `off`: Port disabled. Depending on the implementation, a port may still have a link when disabled but will not transmit/receive any frame.

Command	`eth port adstate show [port]`	
Description	Shows the port admin state of the given port. If no port is given, port state of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port speed show [port]`	
Description	Shows the port state of the given port. If no port is given, port state of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port duplex show [port]`	
Description	Shows the duplex mode of the given port. If no port is given, duplex mode of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port mirror set <port> <<portmap>|<mac>> <ida|insa|eda|inda|eport|inport>`
Description	Sets mirror mode of port

Parameter	`[port]`	The port as number starting from 0 for the first port.
	`<<portmap>|<mac>>`	Either port map or MAC address for mirrored ports.
	`<eda|esa|inda|insa|eport|inport>`	The port mirror mode. • `eda`: egress destination address (requires mac address) • `inda`: ingress destination address (requires mac address) • `esa`: egress source address (requires mac address) • `insa`: ingress sorce address (requires mac address) • `eport`: egress port (requires portmap) • `inport`: ingress port (requires portmap)

Command	`eth port mirror show [port]`	
Description	Shows the mirror mode of the given port. If no port is given, mirror mode of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port mdi state show [port]`	
Description	Shows the port MDI state of the given port. If no port is given, the state of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port mdi state set <port> <default|uncrossed|crossed>`	
Description	Set the port MDI state of the given port.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<default|uncrossed|crossed>`	The MDI state: • `default`: the default state • `uncrossed`: Rx and Tx paths are straight through connected • `crossed`: Rx and Tx paths are crossed

Command	`eth port mdi mode show [port]`	
Description	Shows the port MDI mode of the given port. If no port is given, the mode of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth port mdi mode set <port> <default|auto|manual>`	
Description	Set the port MDI mode of the given port.	
Parameter	`<port>`	The port as number starting from 0 for the first port
	`<default|auto|manual>`	The MDI mode: • `default`: the default mode • `auto`: the required MDI state is automatically detected • `manual`: the MDI state is manually set and will not change

5.5.9 QoS Settings

Command	`eth qos mode set <port> <etype|mac|ip|vlan> <on|off>`	
Description	Enables/disable the different QoS priority resolution modes for the given port. All modes may be active.	
Parameter	`<port>`	The port as number starting from 0 for the first port.
	`<etype|mac|ip|vlan>`	The priority type to use: • `etype`: Enables Ethertype priority resolution • `mac`: Enables MAC based priority resolution • `ip`: Enables IP DiffServ/COS priority resolution • `vlan`: Enables VLAN priority resolution
	`<on|off>`	Enables/disables the mode: • `on`: Mode enabled. • `off`: Mode disabled.

Command	`eth qos mode show [port]`	
Description	Shows the QoS priority resolution mode of the given port. If no port is given, the mode of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth qos defprio set <port> <defprio>`	
Description	Sets the default priority for a frame if none of the active QoS priority resolution modes for the given port provides a resolution.	
Parameter	`<port>`	The port as number starting from 0 for the first port.
	`<defprio>`	The default priority. Valid ranges may differ depending on the underlying hardware.

Command	`eth qos defprio show [port]`	
Description	Shows the default priority of the given port. If no port is given, the priority of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

Command	`eth qos vlanprio set <port> <vlanprio> <mapping>`	
Description	Sets the VLAN priority for the given port.	
Parameter	`<port>`	The port as number starting from 0 for the first port.
	`<vlanprio>`	The VLAN priority to map.
	`<mapping>`	The priority to which the VLAN priority is mapped to.

Command	`eth qos vlanprio show [port]`	
Description	Shows the priority mapping of the given port. If no port is given, the mapping of all ports is shown.	
Parameter	`[port]`	The port as number starting from 0 for the first port

5.5.10 Config Manager

The cm command provides a CLI interface to the GOAL config manager. It allows the manipulation of existing variables and is able to show the current values of variables.

Command	`cm set <modid> <varid> <newval>`	
Description	Sets the value of an existing variable in the config manager.	
Parameter	`<modid>`	The module id of the variable to set
	`<varid>`	The variable id of the variable to set
	`<newval>`	The new value to set. Integer values are entered as is with an optional sign. String values begin and end with “-character.

Command	`cm show [<modid> <varid>]`	
Description	Shows the variable identified by given module and variable id or all variables if no ids are given.	
Parameter	`<modid>`	The module id of the variable to set
	`<varid>`	The variable id of the variable to set

5.5.11 Network Interface

The network interface command group provides access to general network settings e.g. settings for the TCP/IP stack.

5.5.12 IP Settings

The ip sub command provides access to settings of the underlying TCP/IP stack.

Command	`net ip set <ip> <netmask> <gateway>`	
Description	Sets the IP address, the netmask and the default gateway of the underlying TCP/IP stack.	
Parameter	`<ip>`	The new IP address in the format `xxx.xxx.xxx.xxx`
	`<netmask>`	The new netmask in the format `xxx.xxx.xxx.xxx`
	`<gateway>`	The new default gateway in the format `xxx.xxx.xxx.xxx`

Command	`net ip show`
Description	Shows the current IP settings of the underlying TCP/IP stack.
Parameter	`None`

5.6 Statistics

GOAL files:

goal_stat.[h,c]

example:

…\goal\appl\00410_goal\eth_stats

GOAL provides the possibility to track statistics. Primarily this is used for Ethernet to propagate statistics and to analyse communication problems. GOAL provides the following typical ethernet statistics for each port:

GOAL number of ethernet statistic			Description
ID	Number	Identifier	
GOAL_ID_ETH	1	GOAL_ETH_STATS_TOTAL_DISC	number of total discarded frames
GOAL_ID_ETH	2	GOAL_ETH_STATS_TOTAL_BYTE_DISC	number of total discarded bytes
GOAL_ID_ETH	3	GOAL_ETH_STATS_TOTAL_FRAMES	number of total processed frames

GOAL number of ethernet statistic			Description
ID	Number	Identifier	
GOAL_ID_ETH	4	GOAL_ETH_STATS_TOTAL_BYTE_FRAMES	number of total processed bytes
GOAL_ID_ETH	5	GOAL_ETH_STATS_ODISC	number of discarded outgoing frames
GOAL_ID_ETH	6	GOAL_ETH_STATS_IDISC_VLAN	number of discarded wrong or missing VLAN-IDs
GOAL_ID_ETH	7	GOAL_ETH_STATS_IDISC_UNTAGGED	number of discarded missing VLAN tags
GOAL_ID_ETH	8	GOAL_ETH_STATS_IDISC_BLOCK	number of discarded due to blocking mode
GOAL_ID_ETH	9	GOAL_ETH_STATS_LEARN_CNT	number of learned MAC addresses
GOAL_ID_ETH	10	GOAL_ETH_STATS_AFRAMES_RECEIVED_OK	number of received valid frames including pause
GOAL_ID_ETH	11	GOAL_ETH_STATS_AFRAMES_CRC_ERRORS	number of received frames with CRC errors
GOAL_ID_ETH	12	GOAL_ETH_STATS_AALIGNMENT_ERRORS	number of received frames with alignment errors
GOAL_ID_ETH	13	GOAL_ETH_STATS_AOCTETS_TRANSM_OK	number of transmitted valid octets
GOAL_ID_ETH	14	GOAL_ETH_STATS_ATX_PAUSE_CTRL_FRAMES	number of received valid octets
GOAL_ID_ETH	15	GOAL_ETH_STATS_ATX_PAUSE_CTRL_FRAMES	number of transmitted pause frames
GOAL_ID_ETH	16	GOAL_ETH_STATS_ARX_PAUSE_CTRL_FRAMES	number of received pause frames
GOAL_ID_ETH	17	GOAL_ETH_STATS_IFIN_ERRORS	number of received errors
GOAL_ID_ETH	18	GOAL_ETH_STATS_IFOUT_ERRORS	number of transmit errors
GOAL_ID_ETH	19	GOAL_ETH_STATS_IFIN_UCAST_PKTS	number of received unicast frames
GOAL_ID_ETH	20	GOAL_ETH_STATS_IFIN_MCAST_PKTS	number of received multicast frames
GOAL_ID_ETH	21	GOAL_ETH_STATS_IFIN_BCAST_PKTS	number of received

GOAL number of ethernet statistic			Description
ID	Number	Identifier	
			broadcast frames
GOAL_ID_ETH	22	GOAL_ETH_STATS_IFOUT_DISC	number of discarded transmitted frames
GOAL_ID_ETH	23	GOAL_ETH_STATS_IFOUT_UCASR_PKTS	number of transmitted unicast frames
GOAL_ID_ETH	24	GOAL_ETH_STATS_IFOUT_MCAST_PKTS	number of transmitted multicast frames
GOAL_ID_ETH	25	GOAL_ETH_STATS_IFOUT_BCAST_PKTS	number of transmitted broadcast frames
GOAL_ID_ETH	26	GOAL_ETH_STATS_ETHERSTATS_OCTETS	number of all bytes (good and bad)
GOAL_ID_ETH	27	GOAL_ETH_STATS_ETHERSTATS_PKTS	number of all frames (good and bad)
GOAL_ID_ETH	28	GOAL_ETH_STATS_ETHERSTATS_UNDERSIZE	number of frames too short
GOAL_ID_ETH	29	GOAL_ETH_STATS_ETHERSTATS_OVERSIZE	number of frame too long
GOAL_ID_ETH	30	GOAL_ETH_STATS_ETHERSTATS_PKTS64	number of frames with size of 64 bytes
GOAL_ID_ETH	3	GOAL_ETH_STATS_ETHERSTATS_PKTS65TO127	number of frames with size of 65-127 bytes
GOAL_ID_ETH	32	GOAL_ETH_STATS_ETHERSTATS_PKTS128TO255	number of frames with size of 128-255 bytes
GOAL_ID_ETH	33	GOAL_ETH_STATS_ETHERSTATS_PKTS256TO511	number of frames with size of 256-511 bytes
GOAL_ID_ETH	34	GOAL_ETH_STATS_ETHERSTATS_PKTS512TO1023	number of frames with size of 512-1023 bytes
GOAL_ID_ETH	35	GOAL_ETH_STATS_ETHERSTATS_PKT1024TO1518	number of frames with size of 1024-1518 bytes
GOAL_ID_ETH	36	GOAL_ETH_STATS_ETHERSTATS_PKTS1519TOX	number of frames with size >= 1519 bytes
GOAL_ID_ETH	37	GOAL_ETH_STATS_ETHERSTATS_JABBERS	number of jabbers
GOAL_ID_ETH	38	GOAL_ETH_STATS_ETHERSTATS_FRAGS	number of

GOAL number of ethernet statistic			Description
ID	Number	Identifier	
			fragments
GOAL_ID_ETH	39	GOAL_ETH_STATS_VLAN_RECV_OK	number of received valid VLANs
GOAL_ID_ETH	40	GOAL_ETH_STATS_VLAN_TRANS_OK	number of transmitted valid VLANs
GOAL_ID_ETH	41	GOAL_ETH_STATS_FRAMES_RETRANS	number of retransmitted collision frames
GOAL_ID_ETH	42	GOAL_ETH_STATS_ADEFERRED	number of deferred at begin
GOAL_ID_ETH	43	GOAL_ETH_STATS_AMULTIPLE_COLL	number of frames transmitted after multiple collisions
GOAL_ID_ETH	44	GOAL_ETH_STATS_ASINGLE_COLL	number of frames transmitted after single collisions
GOAL_ID_ETH	45	GOAL_ETH_STATS_ALATE_COLL	number of too late collisions
GOAL_ID_ETH	46	GOAL_ETH_STATS_AEXCESS_COLL	number of frames discarded due to 16 consecutive collisions
GOAL_ID_ETH	47	GOAL_ETH_STATS_ACARR_SENSE_ERR	number of PHY carrier sense errors
GOAL_ID_ETH	48	GOAL_ETH_STATS_IFIN_DISC	number of discarded received frames
GOAL_ID_ETH	49	GOAL_ETH_STATS_IFIN_UNKNOWN_PROTO	number of received unknown protocols
GOAL_ID_ETH	50	GOAL_ETH_STATS_SQE_ERR	number of SQE test errors
GOAL_ID_ETH	51	GOAL_ETH_STATS_MAC_TX_ERR	number of internal MAC Tx errors
GOAL_ID_ETH	52	GOAL_ETH_STATS_MAC_RX_ERR	number of internal MAC Rx errors
GOAL_ID_ETH	53	GOAL_ETH_STATS_SYMBOL_ERR	number of symbol errors

Table 4: provided ethernet statistics by GOAL

GOAL tracks statistics, but some can be overwritten using a platform specific implementation.

5.6.1 Access

Read a statistics value:

```
/* get received octets from port 0 */
res = goal_statValGetById(&val, GOAL_ID_ETH, GOAL_STAT_ID_ETH_IFOUTOCTETS, 0);
if (GOAL_RES_ERR(res)) {
    goal_logErr("failed to retrieve statistics counter");
    return;
}
```

Reset a statistics value:

```
res = goal_statResetById(GOAL_ID_ETH, GOAL_STAT_ID_ETH_IFOUTOCTETS, 0);
if (GOAL_RES_ERR(res)) {
    goal_logErr("failed to reset statistics counter");
    return;
}
```

5.6.2 Ethernet statistics

Each platform manages the support of the ethernet statistics listed in Table 4 for the ID GOAL_ID_ETH by a bit-coded mask of the GOAL data type uint64_t. Bit 0 of the mask represents the ethernet statistic with the GOAL number 0.
The access to the statistic values are realized about the ethernet commands:

- GOAL_ETH_CMD_STATS_MASK_GET: read the supported ethernet statistics from the platform as bit-coded mask for all ethernet port
- GOAL_ETH_CMD_STATS_GET: read the values of all supported ethernet statistics for one ethernet port
- GOAL_ETH_CMD_STATS_RST: reset ethernet statistics for ethernet ports; it is platform-specific which statistics of one or all ethernet ports are reset

The ethernet commands are executed by function goal_ethCmd().

If the compiler-define GOAL_CONFIG_ETH_STATS_NAMES is set to 1, a short description for each ethernet statistic is available in code by function goal_ethStatsNameGet().

example:
…\goal\appl\00410_goal\eth_stats

5.7 Generic GOAL instances

This GOAL core module provides functions to manage instances of GOAL core modules or/and GOAL extension modules. Each instance is identifiable by an instance type and an instance-ID. The instance type specifies the GOAL core module or the GOAL extension module. The instance types are defined in …\goal\goal\goal_id.h. The instance-ID is an arbitrary number. Each instance-ID must be used once within the same instance type.

GOAL files:

goal_inst.[h,c]

example:

not available

5.8 Locking

This GOAL core module provides functions to lock resources in the GOAL system. This module supports two types of lock mechanism:

- counting semaphore, specified by the enum GOAL_LOCK_COUNT
- binary mutex, specified by the enum GOAL_LOCK_BINARY

The behavior for waiting on a semaphore or mutex can be configured. Active or passive waiting is possible.
The implementation of the lock mechanisms is platform-specific. In GOAL systems with an operating system the lock mechanisms use the appropriate services of the operating system.

The system is halted by function goal_targetHalt() in case of an error.

GOAL files:

goal_lock.[h,c]

example:

…\goal\appl\00410_goal\task_lock

5.8.1 Platform API

GOAL requires the following indication function to connect the GOAL system to the appropriate services of the operating system:

Prototype	`GOAL_STATUS_T goal_targetLockInit(void)`
Description	This indication function initializes the locking mechanism on the operating system. This function is called in the stage GOAL_STAGE_LOCK_PRE in state GOAL_FSA_INIT_GOAL.
Parameters	None
Return values	GOAL return status, see chapter 8.3
Category	Mandatory
Condition	None

Prototype	`GOAL_STATUS_T goal_targetLockShutdown(void)`
Description	This indication function shutdowns the locking mechanism on the operating system. This function is called in the stage GOAL_STAGE_LOCK_PRE in state

	GOAL_FSA_SHUTDOWN.
Parameters	None
Return values	GOAL return status, see chapter 8.3
Category	Mandatory
Condition	None

Prototype	`GOAL_STATUS_T goal_targetLockCreate(GOAL_LOCK_TYPE_T lockType, GOAL_LOCK_T *pLock, uint32_t valInit, uint32_t valMax)`	
Description	This indication function creates a lock on the operating system.	
Parameters	lockType	type of the lock: • GOAL_LOCK_BINARY: • GOAL_LOCK_COUNT:
	pLock	handle for the created lock
	valInit	• counting semaphores: initial value of the lock Number of instances which shall be marked as already in use, normally. • binary mutex: 0
	valMax	• counting semaphores: maximal value of the lock Number of maximal instances which shall be use this lock, normally. • binary mutex: 1
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	
Condition	None	

Prototype	`GOAL_STATUS_T goal_targetLockDelete(GOAL_LOCK_T *pLock)`	
Description	This indication function deletes the specified lock on the operating system.	
Parameters	pLock	handle for the lock
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	
Condition	None	

Prototype	`GOAL_STATUS_T goal_targetLockGet(GOAL_LOCK_T *pLock, uint32_t timeout)`	
Description	This indication function gets a lock from the operating system.	
Parameters	pLock	handle for the lock
	Timeout	behavior if it is not possible to lock the resource: >0: time for waiting on the lock in ms 0: infinite wait
Return values	GOAL return status, see chapter 8.3	

Category	Mandatory
Condition	None

Prototype	`GOAL_STATUS_T goal_targetLockPut(GOAL_LOCK_T *pLock)`	
Description	This indication function returns a lock to the operating system.	
Parameters	pLock	handle for the lock
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	
Condition	None	

5.8.2 Implementation guidelines

5.8.2.1 Use a lock

1. Create a handle for the lock:
```
GOAL_LOCK_T *pLockHdl = NULL;
```
2. Create a binary lock and mark the lock for the GOAL core module goal_lock:
```
goal_lockCreate(GOAL_LOCK_BINARY, &pLockHdl, 0, 1, GOAL_ID_LOCK);
```
3. Wait forever on a lock and set a lock:
```
goal_lockGet(pLockHdl, GOAL_LOCK_INFINITE);
```
4. Reset a lock:
```
goal_lockPut(pLockHdl);
```
5. Delete the lock:
```
goal_lockDelete(pLockHdl);
```

5.9 Logging

This GOAL core module provides functions to output data via an output channel like UART or ethernet. The data can be divided into the following categories, named logging levels:

- error messages
- warning messages
- information messages
- debug messages

For each logging level this module provides an output function:

Logging level	Output function
Error	goal_logErr()
Warning	goal_logWarn()
Information	goal_logInfo()
Debug	goal_logDbg()

The escape sequences \n and \r are filtered out from the before being send out through the channel.

The output channel is configured by the compiler-defines GOAL_CONFIG_LOGGING_TARET_RAW and GOAL_CONFIG_LOGGING_TARGET_SYSLOG.

GOAL provides generic format descriptors to output data to make printf-like format specifiers portable compared to the architecture and compilers. The GOAL format descriptors are initialized architecture-specific in …\goal\plat\arch\common\goal_arch_common.h. The following format descriptors are available: FMT_d32, FMT_i32, FMT_u32, FMT_x32, FMT_d64, FMT_i64, FMT_u64, FMT_x64, FMT_size_t, FMT_ptr and FMT_ptrdiff. FMT_ptr represents a pointer address. FMT_ptrdiff represents a difference of two pointer addresses.
Example: The actual position value of data type int32_t shall be printed as information:

```
goal_logInfo(“actual position: “FMT_i32” inc”, (int32_t) actPosVal);
```

The logging functionality is available after the state GOAL_FSA_INIT_GOAL.
It is recommended only to enable logging during development as it can have a serious impact on the runtime behavior.

GOAL files:

goal_log.[h,c]

example:

…\goal\appl\task_lock

5.9.1 Configuration

The following compiler-defines are available to configure the logging:

GOAL_CONFIG_LOGGING:
0: logging is switched off for the complete GOAL system (default)
1: logging is switched on and the logging can be used by other GOAL components

GOAL_CONFIG_LOGGING_TARGET_RAW:
0: no board-specific output channel is available (default)
1: the board-specific output channel is used, most UART
The board-specific function goal_targetMsgRaw() must be available.

GOAL_CONFIG_LOGGING_TARGET_SYSLOG:
0: no output via a ethernet channel (default)
1: output via the ethernet channel as broadcast ethernet frame, e.g. to indicate the frame by Wireshark

5.9.2 Platform API

Prototype	`void goal_targetMsgRaw(const char *str, unsigned int len)`	
Description	This indication function transmits a raw message.	
Parameters	Str	raw message
	Len	length of the raw message in bytes
Return values	None	
Category	Optional	
Condition	compiler-define GOAL_CONFIG_LOGGING_TARGET_RAW must be set to 1	

5.10 Message Logger

The Message Logger (LM) is a module to buffer log messages generated by any other components, called generating components. The log messages can be processed by further components, called processing components, see Figure 1.

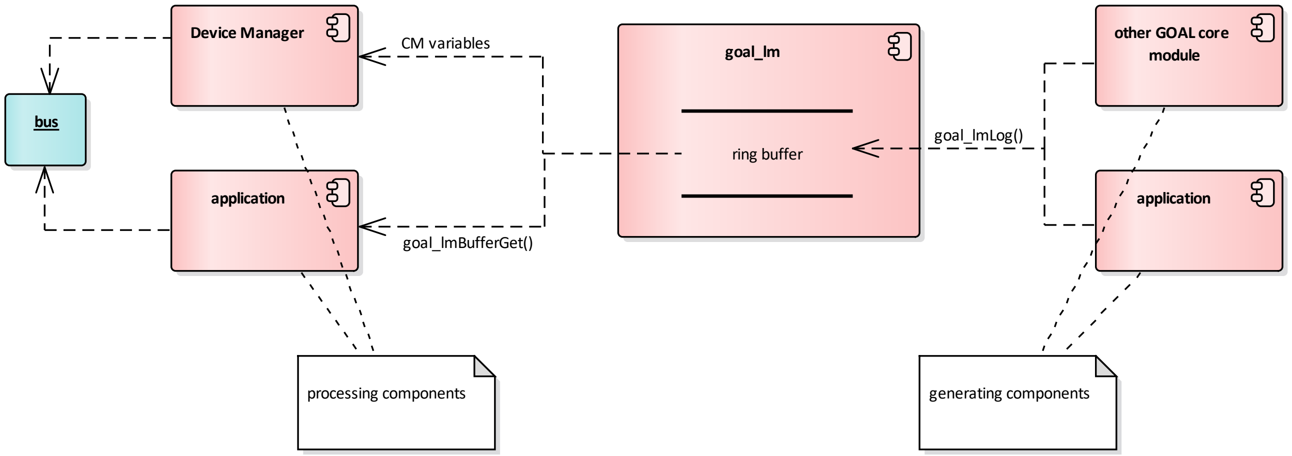


Figure 13 integration of the message logger

A log message consists of a header and a parameter block. The parameter block is optional and can include the values of up to 2 parameters in order to indicate current values or state information on processing-side. The structure of a log message is shown in Figure 14.

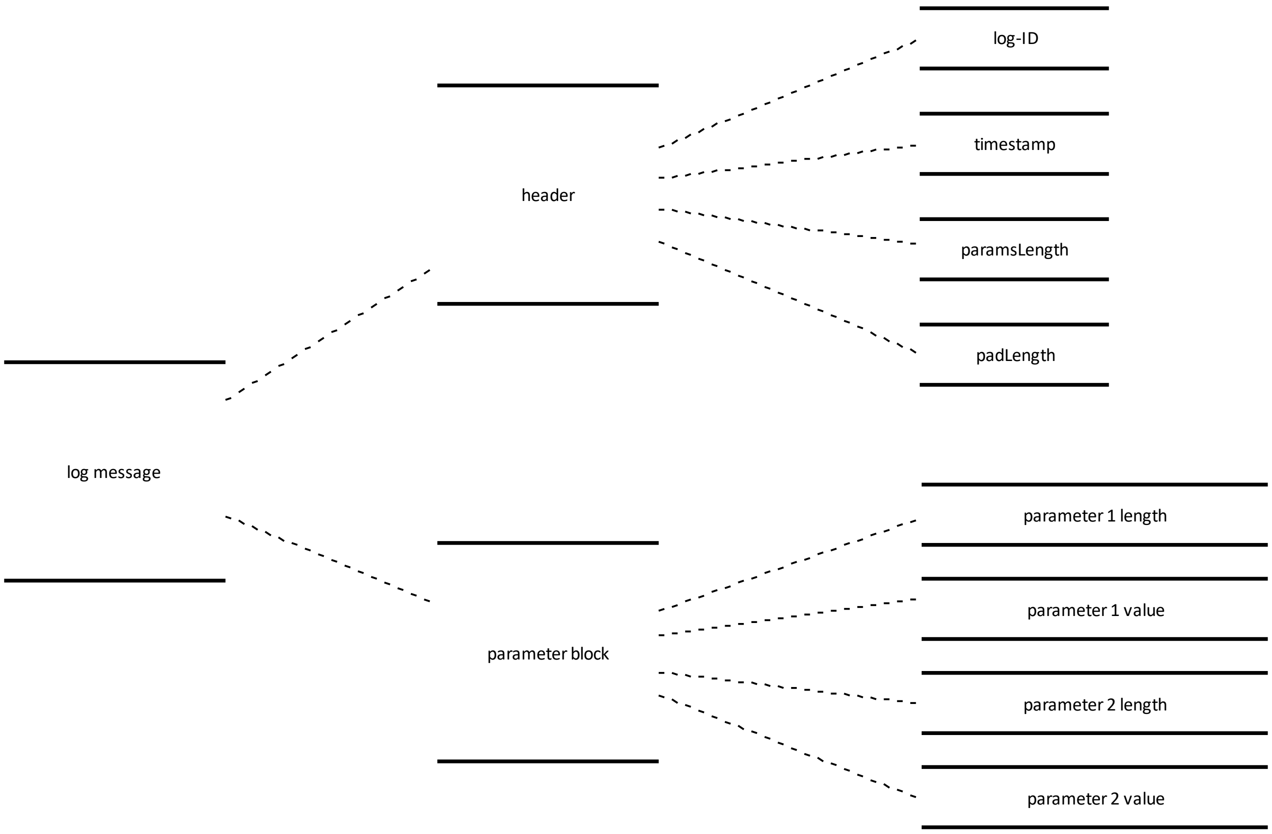


Figure 14: data structure of a log message

The header of the log message contains the following information:

- log-ID: a unique number to report a definite message (4 byte)
- timestamp: indicates the time in ms since the start of the device (8 byte)
- parameter length: length of parameter 1 and parameter 2 in the parameter block in byte (4 byte)
- padding length: number of padding bytes to fill up the log message (2 byte)

The generating components write the header of log messages into the ring buffer by function goal_lmLog(). This function generates the timestamp and adds the padding bytes automatically.

The parameter block contains for each parameter the length (2 bytes) and the parameter value. There are functions to write one parameter depending on the data type of the parameter, e.g. goal_lmLogParamUINT16(). The Message Logger supports parameters of the LM-parameter data types, see chapter 8.2.
The arguments CM-module-ID, text-ID and text of the function goal_lmLog() are implemented for future.
If the ring buffer is full, the next log message, which shall be stored in the ring buffer, overwrites the oldest log messages in the ring buffer. The log messages are stored in a platform dependent byte order.

On processing-side the log-ID shall be known and can be assigned to specific properties, e.g. a logging text and a severity class. The reduction of a unique log-ID allows a fast information transfer with less resources. The Message Logger supports the following severity classes:

- GOAL_LOG_EXCEPTION
- GOAL_LOG_ERROR
- GOAL_LOG_WARNING
- GOAL_LOG_INFO
- GOAL_LOG_DEBUG

The availability of log messages in the ring buffer can be checked by function goal_lmBufferGetCnt(). Log messages can be read from the ring buffer by function goal_lmBufferGet() according to the FIFO-method.

Each processing component has to administrate the read pointer of the ring buffer by itself. This allows that the same log message is interpreted by different processing components.

This module is used by the Device Manager about CM-variables.

GOAL files:

goal_lm.[h,c]

example:

not available

Conventional log messages generated by the logging api are also stored in the logging buffer.

5.10.1 Configuration

5.10.1.1 Compiler-defines

The following compiler-defines are available to configure the Message Logger:

GOAL_LM_BUFFER_SIZE:

size of the ring buffer for the logging messages in bytes (default: 5120 byte)

5.10.1.2 CM-variables

For the configuration of the Message Logger the following CM-variables are available:

CM-Module-ID	GOAL_ID_LM

CM-variable-ID	0
CM-variable name	LM_CM_VAR_READBUFFER
Description	Buffer for reading online logging from device
CM data type	GOAL_CM_GENERIC
Size	128 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_LM
CM-variable-ID	1
CM-variable name	LM_CM_VAR_CNT
Description	Control word for online log access
CM data type	GOAL_CM_UINT16
Size	2 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_LM
CM-variable-ID	2
CM-variable name	LM_CM_VAR_ EXLOG_READBUFFER
Description	Buffer for reading exception logging from device
CM data type	GOAL_CM_GENERIC
Size	128 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_LM
CM-variable-ID	3
CM-variable name	LM_CM_VAR_ EXLOG_CNT
Description	Control word for exception log access
CM data type	GOAL_CM_UINT16
Size	2 bytes
Default value	from NVS or 0

5.10.2 Implementation guidelines

5.10.2.1 Write a log message without parameters to the ring buffer

The log message is generated by the device detection module with the CM-module-ID GOAL_ID_DD. The log message “Error while enabling UDP channel” is classified as GOAL_LOG_ERROR and assigned to log-ID 4 and text-ID 5. Because no parameter shall be transferred, the length of parameter 1 and parameter 2 is 0.

1. write header of the log message:

```
goal_lmLog(GOAL_ID_DD, 4, 5, 0, 0, GOAL_LOG_ERROR, "Error while enabling UDP
channel");
```

5.10.2.2 *Write a log message with parameters to the ring buffer*

The log message is generated by the device detection module with the CM-module-ID GOAL_ID_DD. The log message “Error while opening UDP server channel on port $1” is classified as GOAL_LOG_ERROR and assigned to log-ID 1 and text-ID 2. In error case the port number of the UDP channel shall be reported. The port number has the data type uint32_t. The length of parameter 1 is 4 bytes. The following function sequence is necessary:

1. write header of the log message:

```
goal_lmLog(GOAL_ID_DD, 1, 2, 4, 0, GOAL_LOG_ERROR, "Error while opening UDP server
channel on port $1");
```

2. write the parameter value:

```
goal_lmLogParamUINT32((uint32_t) DD_UDP_PORT);
```

3. finish the entry of the log message in the ring buffer:

```
goal_lmLogFinish();
```

5.11 Network handling

This GOAL core module provides an interface to the application for TCP/IP connections, see Figure 15. A TCP/IP stack is required. The TCP/IP stack must be enabled by the compiler-define GOAL_CONFIG_TCPIP_STACK = 1.

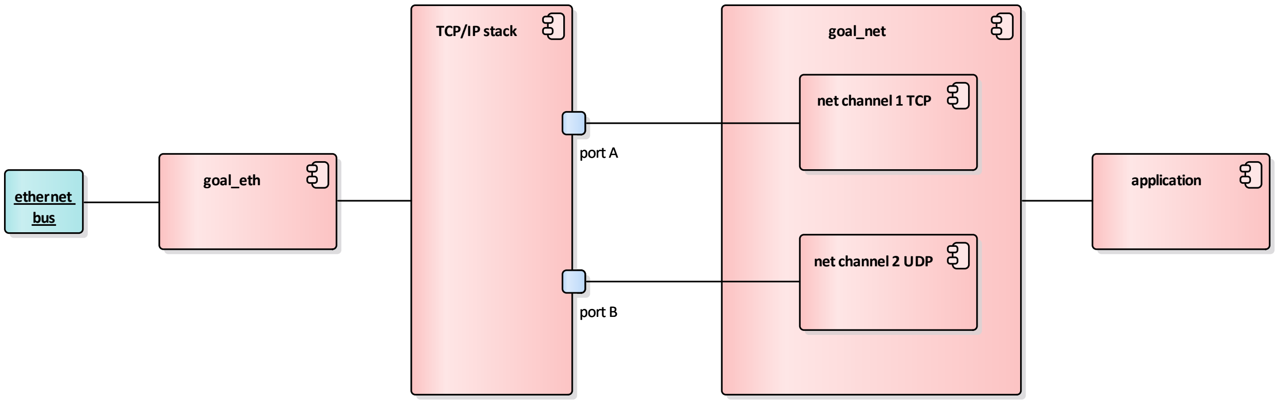


Figure 15: topology for net channels

GOAL creates the number of GOAL_CONFIG_NET_CHAN_MAX net channels during initialization automatically for this purpose. Each net channel can be opened as one of the following network connection types:

- GOAL_NET_UDP_SERVER
- GOAL_NET_UDP_CLIENT
- GOAL_NET_TCP_LISTENER represents the TCP server
- GOAL_NET_TCP_CLIENT

The connection between the net channels and the TCP/IP stack is addressed by the local IP address, local netmask and local gateway address. The connection between the TCP/IP Stack to a remote ethernet device is addressed by the remote IP address, the remote netmask and remote gateway address. The rules for the determination of the local address are shown in Figure 16. The local address is determined during creation of the net channels in the state GOAL_FSA_INIT automatically. The remote address is configured by calling the function goal_netOpen().

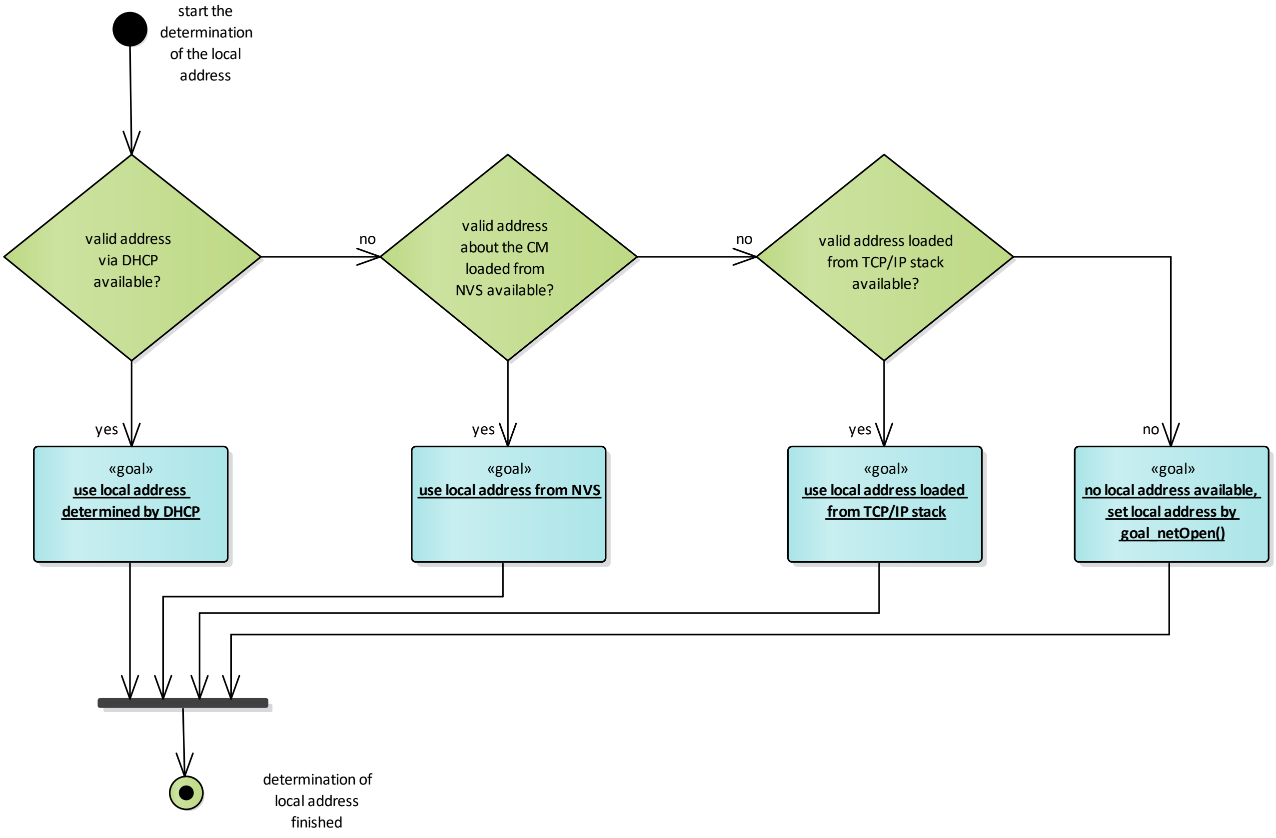


Figure 16: determination of the local address of net channels

The connection to the application is realized by a callback function, see chapter 5.11.2. Each net channel must be activated before data can be transmitted or received via the net channel. The activation is done by function goal_netActivate().

The following options are available to configure the TCP/IP stack:

- GOAL_NET_OPTION_NONBLOCK: socket connection between net channel and TCP/IP stack is
  - 0: blocking
  - 1: non-blocking
- GOAL_NET_OPTION_BROADCAST:
  - 0: no broadcast reception
  - 1: broadcast reception supported
- GOAL_NET_OPTION_TTL:
  - set TTL value in IP-header
- GOAL_NET_OPTION_TOS:
  - set TOS-value in IP-header
- GOAL_NET_OPTION_MCAST_ADD:
  - enable the specified multicast address for the receipt of multicast packets
- GOAL_NET_OPTION_MCAST_DROP:
  - disable the specified multicast address for the receipt of multicast packets
- GOAL_NET_OPTION_REUSEADDR:
  - 0: TCP/IP socket shall be not reusable
  - 1: TCP/IP socket shall be reusable

The options can be can be changed by function goal_netSetOption(). The availability and the default setting of the options depends on the TCP/IP stack.

GOAL files:

goal_net.[h,c], goal_net_dhcp.[h,c], goal_net_cli.c

example:

…\goal\appl\00410_goal\tcp_client

5.11.1 Configuration

5.11.1.1 Compiler-defines

The following compiler-defines are available to configure the network handling:

GOAL_CONFIG_TCPIP_STACK:
0: network handling is disabled (default)
1: network handling is enabled

GOAL_CONFIG_NET_CHAN_MAX:
number of network channels (default: 4)

GOAL_CONFIG_DHCP:
0: static assignment of IP-addresses (default)

1: dynamic assignment of IP-addresses via DHCP

GOAL_CONFIG_IP_STATS:
0: output of IP-statistics switched off (default)
1: output of IP-statistics switched on

5.11.1.2 CM-variables

The following CM-variables are available to configure the network handling:

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	0
CM-variable name	NET_CM_VAR_IP
Description	IP address of first interface
CM data type	GOAL_CM_IPV4
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	1
CM-variable name	NET_CM_VAR_ NETMASK
Description	netmask of first interface
CM data type	GOAL_CM_IPV4
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	2
CM-variable name	NET_CM_VAR_ GW
Description	gateway of first interface
CM data type	GOAL_CM_IPV4
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	3
CM-variable name	NET_CM_VAR_ COMMIT
Description	Write any value to this CM-variable applies the IP settings
CM data type	GOAL_CM_UINT8
Size	1 byte

Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	4
CM-variable name	NET_CM_VAR_ VALID
Description	validity of IP address: • 0: stored IP address is not valid, interface settings originate from network stack of system • 1: stored IP address is valid, will be applied to interface at start of device
CM data type	GOAL_CM_UINT8
Size	1 byte
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	5
CM-variable name	NET_CM_VAR_ DHCP_ENABLED
Description	CM-variable to disable/enable DHCP: • 0: DHCP disabled • 1: DHCP enabled
CM data type	GOAL_CM_UINT8
Size	1 byte
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	6
CM-variable name	NET_CM_VAR_ DHCP_STATE
Description	CM-variable to indicate the current state of DHCP if DHCP is enabled: • 0: DHCP initialized • 1: DHCP selecting server • 2: DHCP requesting configuration • 3: DHCP IP address bound • 4: DHCP renewing configuration • 5: DHCP rebinding IP address to interface
CM data type	GOAL_CM_UINT8
Size	1 byte
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	7

CM-variable name	NET_CM_VAR_ DNS0
Description	first DNS server of the first interface
CM data type	GOAL_CM_IPV4
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	8
CM-variable name	NET_CM_VAR_ DNS1
Description	second DNS server of the first interface
CM data type	GOAL_CM_IPV4
Size	4 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_NET
CM-variable-ID	9
CM-variable name	NET_CM_VAR_ HOSTNAME
Description	host name of the first interface
CM data type	GOAL_CM_STRING
Size	20 bytes
Default value	from NVS or 0

5.11.2 Callback functions

The user of this module can specify the following callback function for each network channel:

Prototype	`void cbNetFunc(GOAL_NET_CB_TYPE_T cbType, struct GOAL_NET_CHAN_T *pChan, struct GOAL_BUFFER_T *pBuf)`	
Description	This callback function is used for the following operations: • GOAL_NET_CB_NEW_DATA: to transfer received data to the application • GOAL_NET_CB_NEW_SOCKET: to inform the application, that a new connection of a net channel to the TCP/IP stack was opened • GOAL_NET_CB_CONNECTED: to inform the application, that the net channel was activated • GOAL_NET_CB_CLOSING: to inform the application, that the net channel was closed	
Parameters	cbType	type of operation: • GOAL_NET_CB_NEW_DATA, • GOAL_NET_CB_NEW_SOCKET, • GOAL_NET_CB_CONNECTED,

		• GOAL_NET_CB_CLOSING
	pChan	handle of the network channel
	pBuf	for GOAL_NET_CB_NEW_DATA: buffer with the received data else: NULL
Return values	none	
Category	optional If a callback function is not available, specify NULL in the call of goal_netOpen().	
Registration	during runtime via function goal_netOpen()	

5.11.3 IP statistics

GOAL provides the possibility to analyze communication problems by IP statistics. The supported IP statistics bases on /RFC_1213/ and depend on the platform. GOAL provides the following typical IP statistics:

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
0	GOAL_NET_IP_STATS_IPINHDRERRORS	The number of input datagrams discarded due to errors in their IP headers, including bad checksums, version number mismatch, other format errors, time-to-live exceeded, errors discovered in processing their IP options, etc.
1	GOAL_NET_IP_STATS_IPINADDRERRORS	The number of input datagrams discarded because the IP address in their IP headers destination field was not a valid address to be received at this entity. This count includes invalid addresses and addresses of unsupported classes. For entities which are not IP gateways and therefore do not forward datagrams, this counter includes datagrams discarded because the destination address was not a local address.
2	GOAL_NET_IP_STATS_IPINUNKNOWNPROTOS	The number of locally-addressed datagrams received successfully but discarded because of an unknown or unsupported protocol.
3	GOAL_NET_IP_STATS_IPINDISCARDS	The number of input IP datagrams for which no problems were encountered to prevent their continued processing, but

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
		which were discarded. Note that this counter does not include any datagrams discarded while awaiting re-assembly.
4	GOAL_NET_IP_STATS_IPINDELIVERS	The total number of input datagrams successfully delivered to IP user-protocols (including ICMP).
5	GOAL_NET_IP_STATS_IPOUTREQUESTS	The total number of IP datagrams which local IP user-protocols (including ICMP) supplied to IP in requests for transmission. Note that this counter does not include any datagrams counted in 14/GOAL_NET_IP_STATS_IPFORWDATAGRAMS.
6	GOAL_NET_IP_STATS_IPOUTDISCARDS	The number of output IP datagrams for which no problem was encountered to prevent their transmission to their destination, but which were discarded. Note that this counter would include datagrams counted in 14/GOAL_NET_IP_STATS_IPFORWDATAGRAMS if any such packets met this discard criterion.
7	GOAL_NET_IP_STATS_IPOUTNOROUTES	The number of IP datagrams discarded because no route could be found to transmit them to their destination. Note that this counter includes any packets counted in 14/GOAL_NET_IP_STATS_IPFORWDATAGRAMS which meet this “no-route” criterion. Note that this includes any datagram which a host cannot route because all of its default gateways are down.
8	GOAL_NET_IP_STATS_IPREASMOKS	The number of IP datagrams successfully reassembled.
9	GOAL_NET_IP_STATS_IPREASMFAILS	The number of failures detected by the IP reassembly algorithm. Note that this is not necessarily a count of discarded IP fragments since some algorithms can lose track of the number of fragments by combining them as they are received.
10	GOAL_NET_IP_STATS_IPFRAGOKS	The number of IP datagrams that have

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
		been successfully fragmented at this entity.
11	GOAL_NET_IP_STATS_IPFRAGFAILS	The number of IP datagrams that have been discarded because they needed to be fragmented at this entity but could not be.
12	GOAL_NET_IP_STATS_IPFRAGCREATES	The number of IP datagram fragments that have been generated as a result of fragmentation at this entity.
13	GOAL_NET_IP_STATS_IPREASMREQGDS	The number of IP fragments received which needed to be reassembled at this entity.
14	GOAL_NET_IP_STATS_IPFORWDATAGRAMS	The number of input datagrams for which this entity was not their final IP destination, as a result of which an attempt was made to find a route to forward them to that final destination. In entities which do not act as IP gateways, this counter will include only those packets which were source-routed via this entity, and the source-route option processing was successful.
15	GOAL_NET_IP_STATS_IPINRECEIVES	The total number of input datagrams received from interfaces, including those received in error.
16	GOAL_NET_IP_STATS_TCPACTIVEOPENS	The number of times TCP connections have made a direct transition from the CLOSED state to the SYN-SENT state.
17	GOAL_NET_IP_STATS_TCPPASSIVEOPENS	The number of times TCP connections have made a direct transition from the LISTEN state to the SYN-RCVD state.
18	GOAL_NET_IP_STATS_TCPATTEMPTFAILS	The number of times TCP connections have made a direct transition from either the SYN-SENT or SYN-RCVD state to the CLOSED state, plus the number of times TCP connections have made a direct transition from the SYN-RCVD state to the LISTEN state.
19	GOAL_NET_IP_STATS_TCPESTABRESETS	The number of times TCP connections have made a direct transition from either the ESTABLISHED or CLOSE-WAIT state to the CLOSE state.

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
20	GOAL_NET_IP_STATS_TCPOUTSEGS	The total number of segments sent, including those on current connections but excluding those containing only retransmitted octets.
21	GOAL_NET_IP_STATS_TCPRETRANSSEGS	The total number of segments retransmitted. That is the number of TCP segments transmitted containing one or more previously transmitted.
22	GOAL_NET_IP_STATS_TCPINSEGS	The total number of segments received, including those received in error. This count includes segments received on currently established connections.
23	GOAL_NET_IP_STATS_TCPINERRS	The total number of segments received in error.
24	GOAL_NET_IP_STATS_TCPOUTRSTS	The number of TCP segments sent containing the RST flag.
25	GOAL_NET_IP_STATS_UDPINDATAGRAMS	The total number of UDP datagrams delivered to UDP user.
26	GOAL_NET_IP_STATS_UDPNOPORTS	The total number of received UDP datagrams for which there was no application at the destination port.
27	GOAL_NET_IP_STATS_UDPINERRORS	The number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port.
28	GOAL_NET_IP_STATS_UDPOUTDATAGRAMS	The total number of UDP datagrams sent from this entity.
29	GOAL_NET_IP_STATS_ICMPINMSGS	The total number of ICMP messages which the entity received. Note that this counter includes all those counted by 30/ GOAL_NET_IP_STATS_ICMPINERRORS.
30	GOAL_NET_IP_STATS_ICMPINERRORS	The number of ICMP messages which the entity received but determined as having ICMP-specific errors.
31	GOAL_NET_IP_STATS_ICMPINDESTUNREACHS	The number of ICMP Destination Unreachable messages received.
32	GOAL_NET_IP_STATS_ICMPINTIMEEXDS	The number of ICMP Time Exceeded messages received.
33	GOAL_NET_IP_STATS_ICMPINPARMPROBS	The number of ICMP Parameter Problem messages received.
34	GOAL_NET_IP_STATS_ICMPINSRCQUENCHS	The number of ICMP Source Quench

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
		messages received.
35	GOAL_NET_IP_STATS_ICMPINREDIRECTS	The number of ICMP Redirect messages received.
36	GOAL_NET_IP_STATS_ICMPINECHOS	The number of ICMP Echo (request) messages received.
37	GOAL_NET_IP_STATS_ICMPINECHOREPS	The number of ICMP Echo Reply messages received.
38	GOAL_NET_IP_STATS_ICMPINTIMESTAMPS	The number of ICMP Timestamp (request) messages received.
39	GOAL_NET_IP_STATS_ICMPINTIMESTAMPREPS	The number of ICMP Timestamp Reply messages received.
40	GOAL_NET_IP_STATS_ICMPINADDRMASKS	The number of ICMP Address Mask Request messages received.
41	GOAL_NET_IP_STATS_ICMPINADDRMASKREPS	The number of ICMP Address Mask Reply messages received.
42	GOAL_NET_IP_STATS_ICMPOUTMSGS	The total number of ICMP messages which this entity attempted to send. Note that this counter includes all those counted by 43/ GOAL_NET_IP_STATS_ICMPOUTERRORS.
43	GOAL_NET_IP_STATS_ICMPOUTERRORS	The number of ICMP messages which this entity did not send due to problems discovered within ICMP such as a lack of buffers. This value should not include errors discovered outside the ICMP layer such as the inability of IP zo route the resultant datagram. In some implementations there may be no types of error which contribute to this counter's value.
44	GOAL_NET_IP_STATS_ICMPOUTDESTUNREACHS	The number of ICMP Destination Unreachable message sent.
45	GOAL_NET_IP_STATS_ICMPOUTTIMEEXCDS	The number of ICMP Time Exceeded messages sent.
46	GOAL_NET_IP_STATS_ICMPOUTECHOS	The number of ICMP Echo (request) messages sent.
47	GOAL_NET_IP_STATS_ICMPOUTECHOREPS	The number of TCMP Echo Reply messages sent.
48	GOAL_NET_IP_STATS_IFINOCTETS	The total number of octets received on the interface, including framing characters.
49	GOAL_NET_IP_STATS_IFINUCASTPKTS	The number of subnetwork-unicast

GOAL number of IP statistic		Description /RFC_1213/
Number	Identifier (object type)	
		packets delivered to a higher-layer protocol.
50	GOAL_NET_IP_STATS_IFINNUCASTPKTS	The number of non-unicast packets delivered to a higher-layer protocol.
51	GOAL_NET_IP_STATS_IFINDISCARDS	The number of inbound packet which were chosen to be discarded even though no errors had been detected to prevent their being deliverable to a higher-layer protocol.
52	GOAL_NET_IP_STATS_IFINERRORS	The number of inbound packets that contained errors preventing them from being deliverable to a higher-layer protocol.
53	GOAL_NET_IP_STATS_IFINUNKNOWNPROTOS	The number of packets received via the interface which were discarded because of an unknown or unsupported protocol.
54	GOAL_NET_IP_STATS_IFOUTOCTETS	The total number of octets transmitted out of the interface, including framing characters.
55	GOAL_NET_IP_STATS_IFOUTUCASTPKTS	The total number of packets that higher-level protocols requested be transmitted to a subnetwork-unicast address, including those that were discarded or not sent.
56	GOAL_NET_IP_STATS_IFOUTNUCASTPKTS	The total number of packets that higher-level protocols requested be transmitted to a non-unicast address, including those that were discarded or not sent.
57	GOAL_NET_IP_STATS_IFOUTDISCARDS	The number of outbound packets which were chosen to be discarded even though no errors had been detected to prevent their being transmitted. One possible reason for discarding such a packet could be to free up buffer space.
58	GOAL_NET_IP_STATS_IFOUTERRORS	The number of outbound packets that could not be transmitted because of errors.

Table 5: provided IP statistic by GOAL

Each platform manages the support of the IP statistics listed in Table 5 by a bit-coded mask of the GOAL data type uint64_t. Bit 0 of the mask represents the IP statistic with the GOAL number 0. The access to the statistic values are realized about the ethernet commands:

- GOAL_NET_CMD_IP_STATS_MASK_GET: read the supported IP statistics from the platform as bit-coded mask for all port
- GOAL_NET_CMD_IP_STATS_GET: read the values of all supported IP statistics
- GOAL_NET_CMD_IP_STATS_RST: reset IP statistics; it is platform-specific which statistics of one or all ports are reset

The IP commands are executed by function goal_targetNetCmd().

5.11.4 Platform API

GOAL requires the following indication function for the handling of net channels:

Prototype	`uint32_t goal_targetNetGetHandleSize(void)`
Description	This indication function returns the memory size, which is needed for a net channel handle.
Parameters	None
Return values	size of a net channel handle in bytes
Category	Mandatory

Prototype	`GOAL_STATUS_T goal_targetNetRecv(GOAL_BUFFER_T **ppBuf)`	
Description	This indication function is called everytime a TCP/IP packet is received.	
Parameters	ppBuf	GOAL ethernet buffer containing the received packet
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetIpSet(uint32_t addrIp, uint32_t addrMask, uint32_t addrGw, GOAL_BOOL_T flgTemp)`	
Description	This indication function allows to set the IP configuration for the TCP/IP stack. This function is called in state GOAL_FSA_INIT normally.	
Parameters	addrIp	local IP address
	addrMask	local subnet mask
	addrGw	local gateway address
	flgTemp	kind of the IP configuration • GOAL_TRUE: There are no CM-variables available to store the IP configuration. The IP configuration is handled temporary. • GOAL_FALSE: There are CM-variables available to store the IP configuration. The IP configuration is handled about CM-variables.
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetIpGet(uint32_t *pAddrIp, uint32_t *pAddrMask, uint32_t *pAddrGw, GOAL_BOOL_T *pFlgTemp)`	
Description	This indication function returns the current IP configuration used by the TCP/IP stack.	
Parameters	pAddrIp	current local IP address
	pAddrMask	current local subnet mask
	pAddrGw	current local gateway address
	pFlgTemp	current kind of the IP configuration • GOAL_TRUE: There are no CM-variables available to store the IP configuration. The IP configuration is handled temporary. • GOAL_FALSE: There are CM-variables available to store the IP configuration. The IP configuration is handled about CM-variables.
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetOpen(void **ppTargetHandle, GOAL_NET_TYPE_T type, GOAL_NET_ADDR_T *pAddr)`	
Description	This indication function allows to open a net channel.	
Parameters	ppTargetHandle	handle for the net channel
	type	connection type: • GOAL_NET_UDP_SERVER • GOAL_NET_UDP_CLIENT • GOAL_NET_TCP_LISTENER • GOAL_NET_TCP_CLIENT
	pAddr	local and maybe remote address of the net channel
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetReopen(char *pTgtHandle, GOAL_NET_TYPE_T type, GOAL_NET_ADDR_T *pAddr)`	
Description	This indication function allows to reopen the net channel specified by the handle.	
Parameters	pTgtHandle	handle for the net channel
	type	connection type: • GOAL_NET_UDP_SERVER • GOAL_NET_UDP_CLIENT • GOAL_NET_TCP_LISTENER

		• GOAL_NET_TCP_CLIENT
	pAddr	local and maybe remote address of the net channel
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetClose(void *pTargetHandle, GOAL_NET_TYPE_T type)`	
Description	This indication function allows to close the net channel specified by the handle.	
Parameters	pTargetHandle	handle for the net channel
	type	connection type: • GOAL_NET_UDP_SERVER • GOAL_NET_UDP_CLIENT • GOAL_NET_TCP_LISTENER • GOAL_NET_TCP_CLIENT
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetActivate(void *pTargetHandle)`	
Description	This indication function allows to activate the net channel specified by the handle.	
Parameters	pTargetHandle	handle for the net channel
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetDeactivate(void *pTargetHandle)`	
Description	This indication function allows to deactivate the net channel specified by the handle.	
Parameters	pTargetHandle	handle for the net channel
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetSend(void *pTargetHandle, GOAL_NET_TYPE_T type, GOAL_NET_ADDR_T *pAddr, GOAL_BUFFER_T *pBuf)`	
Description	This indication function transmit data via the net channel to the TCP/IP stack.	
Parameters	pTargetHandle	handle for the net channel
	type	connection type: • GOAL_NET_UDP_SERVER • GOAL_NET_UDP_CLIENT

		• GOAL_NET_TCP_LISTENER • GOAL_NET_TCP_CLIENT
	pAddr	local and maybe remote address of the net channel
	pBuf	buffer with the packet to transmit
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`GOAL_STATUS_T goal_targetNetOptSet(void *pTargetHandle, GOAL_NET_TYPE_T type, GOAL_NET_OPTION_T option, void *pValue)`	
Description	This indication function allows to change one property of the net channel.	
Parameters	pTargetHandle	handle for the net channel
	type	connection type: • GOAL_NET_UDP_SERVER • GOAL_NET_UDP_CLIENT • GOAL_NET_TCP_LISTENER • GOAL_NET_TCP_CLIENT
	option	property of the net channel: • GOAL_NET_OPTION_NONBLOCK: set socket to non-blocking • GOAL_NET_OPTION_BROADCAST • GOAL_NET_OPTION_TTL • GOAL_NET_OPTION_TOS • GOAL_NET_OPTION_MCAST_IF • GOAL_NET_OPTION_MCAST_ADD • GOAL_NET_OPTION_MCAST_DROP • GOAL_NET_OPTION_REUSEADDR
	pValue	value of the selected option
Return values	GOAL return status, see chapter 8.3	
Category	Mandatory	

Prototype	`void goal_targetNetPoll(void)`
Description	This indication function is called in the state GOAL_FSA_OPERATION execute loop-controlled actions.
Parameters	None
Return values	None
Category	Mandatory

Prototype	`GOAL_BOOL_T goal_targetNetAvail(void)`
Description	This indication function checks if new data was received.

Parameters	None
Return values	state of received data: - GOAL_TRUE: received data available - GOAL_FALSE: no data received
Category	Mandatory

Prototype	`GOAL_STATUS_T goal_targetNetCmd(GOAL_NET_CMD_T id, GOAL_BOOL_T wrFlag, void *pArg)`	
Description	This indication function allows to execute a net command.	
Parameters	id	command identifier
	wrFlag	access direction - GOAL_TRUE: write argument - GOAL_FALSE: read argument
	pArg	argument to the net command
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

5.11.5 Command line interface

Command	`net ip set <ip> <netmask> <gateway>`	
Description	Sets the remote IP-address, the netmask and the default gateway of the underlying TCP/IP stack.	
Parameter	`<ip>`	The new IP address in the format `xxx.xxx.xxx.xxx`
	`<netmask>`	The new netmask in the format `xxx.xxx.xxx.xxx`
	`<gateway>`	The new default gateway in the format `xxx.xxx.xxx.xxx`

Command	`net ip show`
Description	Prints the remote address consisting of the IP-address, netmask and gateway address of the underlying TCP/IP stack to the command line interface.
Parameter	`none`

5.11.6 Implementation guidelines

5.11.6.1 Configure, open and activate a net channel

1. All net channels are created automatically in the state GOAL_FSA_INIT_GOAL and the local addresses are determined.

2. Set the local IP address in state GOAL_FSA_INIT:

```
uint32_t ipAddr;
uint32_t netmask;
uint32_t gatewayAddr;

ipAddr= GOAL_NET_IPV4(192, 168, 0, 100);
netmask = GOAL_NET_IPV4(255, 255, 255, 0);
gatewayAddr = GOAL_NET_IPV4(0, 0, 0, 0);

goal_netIpSet(ipAddr, netmask, gatewayAddr);
```

3. Create a callback function to handle actions on the net channel application-specific:

```
void applNetCallback(GOAL_NET_CB_TYPE_T cbType, struct GOAL_NET_CHAN_T *pChan,
struct GOAL_BUFFER_T *pBuf) {
    …
}
```

4. Create a handle for the net channel:

```
GOAL_NET_CHAN_T *pNetChanHdl;
```

5. Create the address information of the net channel:

```
GOAL_NET_ADDR_T addr;
addr.localIp = ipAddr;
addr.localPort = 1234;
addr.remoteIP = GOAL_NET_IPV4(192, 168, 0, 10);
addr.remotePort = 1234;
```

6. Open a net channel by function goal_netOpen() and specify the remote address and a callback function:

```
goal_netOpen(&pNetChanHdl, &addr, GOAL_NET_UDP_CLIENT, applNetCallback);
```

7. Maybe change a property of the net channel by function goal_netSetOption(), e.g. configure the net channel as non-blocking:

```
uint32_t optVal:
optVal = 1;
goal_netSetOption(pNetChanHdl, GOAL_NET_OPTION_NONBLOCK, &optVal);
```

8. Activate the net channel by function goal_netActivate():

```
goal_netActivate(pNetChanHdl);
```

5.11.6.2 Send data

Use a buffer managed about a GOAL queue to transmit data.

1. Create a handle for the queue:

```
GOAL_QUEUE_T *pQueueHdl;
```

2. Create a queue with max. 10 buffers and allocate the memory for all buffers. The size of each buffer is 20 bytes. The queue has to be created by function goal_queueInit() in the state GOAL_FSA_INIT.

```
goal_queueInit(&pQueueHdl, 10, 10, 20);
```

3. Create a handle for a buffer:

```
GOAL_BUFFER_T *pBuf;
```

4. Take an uninitialized buffer from the queue:

```
goal_queueGetElem(pQueueHdl, &pBuf);
```

5. Initialize the buffer and mark the buffer as used:

```
pBuf->usage = GOAL_ID_QUEUE;
pBuf->relCb = NULL;  /* no callback function shall be called */
pBuf->pQueue = pQueueHdl;  /* return the buffer to the same queue */
pBuf->flags = GOAL_QUEUE_FLG_USED;
```

6. Write a value of 4 bytes to the buffer:

```
uint32_t value = 0x11223344;
pBuf->ptrData = (uint8_t *)&value;
pBuf->dataLen = 4;
```

7. Send data by function goal_netSend() and receive data via the specified callback function:

```
goal_netSend(pNetChanHdl, pBuf);
```

8. Close and deactivate the net channel by function goal_netClose():

```
goal_netCose(pNetChanHdl);
```

5.12 Queue buffer pool

This GOAL core module provides functions to manage a pool of buffers organized in queues. Single buffers can be taken from top of the queue. The buffers can be read, written or cleared by the application. After processing the buffers are returned at the end of the queue. The buffer handling of a queue is organized as FIFO. Accesses to the queue and the buffers is protected by the GOAL locking mechanism.

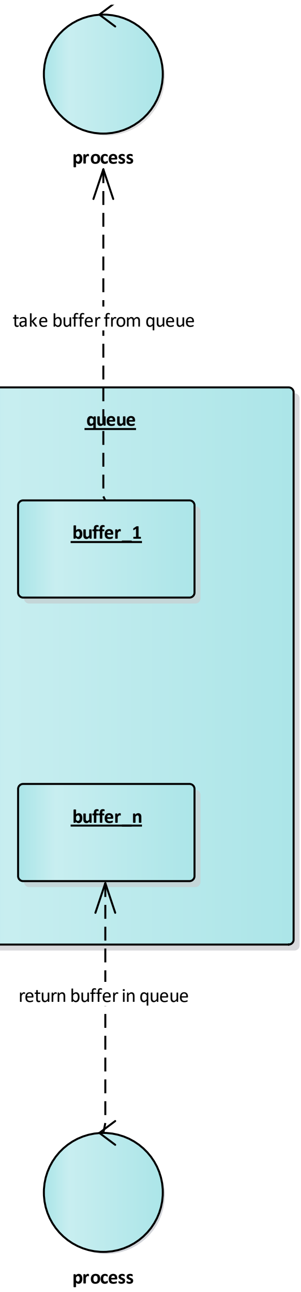


Figure 17: queue buffer handling

It is possible to store buffers to other queues. This method shall only be used in exceptional cases and shall be used very careful.

The function using the queue mechanism is responsible to manage the buffers and for the buffer content. Each buffer has a header for management purposes, described in chapter 5.12.2.

This module provides the following functions to manage the queue:

Function in goal_queue	Description

Function in goal_queue	Description
goal_queueInit()	create a queue with buffers This function allocates the memory of a specified number of buffers with the same size and assigns the buffers to the queue. It is also possible to create an empty queue without buffers and to add buffers in the state GOAL_FSA_OPERATION. In this case the memory for the buffers must be allocated in the state GOAL_FSA_INIT by another process.
goal_queuePoolBufsReq()	Initially create free buffers for application specific usage This function has to be called by each user of a pool. It tells the queue buffer pool how many buffers the user requires. There are two paremeters regarding the number of buffers. First parameter defines the number of buffers that are required at any given time. Seconds parameter defines the number of buffers, that may be required temporarily additionally. Those temporarily buffers can be shared between multiple applications. This function is required if the system pools are used (goal_queueGetNewBuf).
goal_queueSetReleaseCallback()	specify a callback function buffer-related, which is called by one of the functions goal_queueRelease*()
goal_queueGetNewBuf()	take a buffer from the queue and initialize buffer The following buffer properties are initialized, see chapter 5.12.2: flags: GOAL_QUEUE_FLG_USED dataLen: 0 netPort: GOAL_ETH_PORT_HOST relCb: NULL pEthBufHdlr: NULL
goal_queueGetElem()	take a buffer from the queue The buffer can be uninitialized or can contain valid data.
goal_queueAddElem()	return a buffer into the queue The content of the buffer remains unchanged.
goal_queueReleaseBuf() goal_queueReleaseBufToOrigQueue() goal_queueReleaseBufToNewQueue()	It is only allowed to release a buffer if: • the release was allowed for this buffer: see chapter 5.12.3/ GOAL_QUEUE_FLG_NO_RELEASE • content of the buffer is not in transmission: see chapter 5.12.3/ GOAL_QUEUE_FLG_TX If no release callback is specified by function goal_queueSetReleaseCallback(), the buffer is returned in

Function in goal_queue	Description
	the desired queue according to the called function: • goal_queueReleaseBuf(): return buffer into the queue specified in the buffer property pQueue • goal_ queueReleaseBufToOrigQueue(): return buffer into the queue, which has created the buffer during goal_queueInit() • goal_queueReleaseBufToNewQueue(): append buffer to the specified queue in the function call The flag GOAL_QUEUE_FLG_USED is cleared. If a release callback is specified by function goal_queueSetReleaseCallback(), the callback function is called and is responsible to return the buffer into a queue. The content of the buffer remains unchanged.

GOAL files:

goal_queue.[h,c]

example:

not available

5.12.1 Callback functions

GOAL allows to install a callback function to release a buffer to a queue application-specific. The name of the callback function is application-specific.

Prototype	`GOAL_STATUS_T cbQueueRelFunc(struct GOAL_BUFFER_T *pBuf, void *pArg)`	
Description	This callback function allows to do actions by the application before the buffer is return to the specified queue. If the actions are finished, the callback function has to call one of the functions goal_queueReleaseBuf() or goal_queueReleaseBufToNewQueue() or goal_queueReleaseBufToOrigQueue() to release the buffer.	
Parameters	pBuf	buffer, which shall be returned
	pArg	specific arguments used by the callback function
Return values	GOAL return status, see chapter 8.3	
Category	optional If a callback function is not available, GOAL returns the buffer to the specified queue.	
Registration	during runtime about function goal_queueSetReleaseCallback()	

5.12.2 Buffer header

The usage of each buffer can be controlled separately from the queue management by the user of the queue buffer pool. There are properties available for the buffer management represented by the structure GOAL_BUFFER_T in code. The structure GOAL_BUFFER_T contains public and private properties. The user of the queue buffer pool shall only change the public properties listed in Table 6.

Some properties are also changed by the queue management during initialization and re-initialization of buffers, see Table 6.

Public property of GOAL_BUFFER_T	Description
dataLen	length of the data in bytes, i.e. used bytes of the buffer This value is cleared by function goal_queueGetNewBuf().
flags	bit-coded flags to control special buffer tasks, see chapter 5.12.3
netPort	port number to the ethernet network, usable if the GOAL queue mechanism is used for sending and receiving ethernet frames
etherType	type field of the received ethernet frame according to IEEE-802.3
relCb	callback function called by goal_queueRelease*() The application can specify a callback function by function goal_queueSetReleaseCallback(). The callback function is deleted by the function goal_queueGetNewBuf().
tsSec	timestamp in s of the received ethernet frame The availability of a timestamp depends on the platform.
tsNsec	timestamp in ns oft he received ethernet frame The availability of a timestamp depends on the platform.

Table 6: public elements of queue buffers

5.12.3 Buffer flags

Each buffer of a queue buffer pool can be controlled by the following flags:

Flag of GOAL_BUFFER_T/flags	Description
GOAL_QUEUE_FLG_USED	0: buffer is free 1: buffer is used goal_queueGetNewBuf() set this bit. goal_queueRelease*() reset this bit.
GOAL_QUEUE_FLG_NO_RELEASE	0: buffer can be released 1: buffer must not be released

Flag of GOAL_BUFFER_T/flags	Description
GOAL_QUEUE_FLG_TX	0: no transmission is active, buffer can be released 1: buffer content is still transmitted, buffer cannot be released
GOAL_QUEUE_FLG_VLAN	This bit indicates if the received ethernet frame uses the VLAN protocol: 0: ethernet frame uses another protocol 1: VLAN is used This setting corresponds with the property etherType of GOAL_BUFFER_T.
GOAL_QUEUE_FLG_TIMESTAMP	This bit allows to activate the sending a ethernet frame with time stamp: 0: no timestamp is transmitted 1: timestamp is transmitted This property must be supported by the platform.

Table 7: control flags of queue buffers

5.12.4 Internal queue usage

GOAL uses 3 queues for internal purposes with different memory sizes: small, medium and big. The number and size of the data buffers can be configured and adapted to the user system.

The number and the size of each internal queue can be configured by a CM-variable in the CM-module with the module-ID GOAL_ID_QUEUE. If no values for these CM-variables are stored in the nonvolatile memory, GOAL uses default values. The memory configuration shall only be changed in a GOAL project if memory optimizations are required.

CM-variable-ID	0
CM-variable name	SMALLBUFSIZE
Description	size of a small memory buffer
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_SMALL_SIZE: 0 byte

CM-variable-ID	1
CM-variable name	SMALLBUFNUM
Description	amount of small memory buffers
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_SMALL_NUM: 0 byte

CM-variable-ID	2

CM-variable name	MEDBUFSIZE
Description	size of a medium memory buffer
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_MED_SIZE: 0 byte

CM-variable-ID	3
CM-variable name	MEDBUFNUM
Description	amount of medium memory buffers
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_MED_NUM: 0 byte

CM-variable-ID	4
CM-variable name	BIGBUFSIZE
Description	size of a medium memory buffer
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_BIG_SIZE: GOAL_NETBUF_SIZE for ethernet or TCP/IP usage, else 0

CM-variable-ID	5
CM-variable name	BIGBUFNUM
Description	amount of medium memory buffers
CM data type	GOAL_CM_INT16
Size	2 bytes
Default value	from NVS or GOAL_QUEUE_BIG_NUM: GOAL_CONFIG_BUF_NUM for ethernet or TCP/IP usage, else 0

5.12.5 Implementation guidelines

5.12.5.1 Get an uninitialized buffer from the queue and add the buffer to the queue

1. Create a handle for the queue:

   ```
   GOAL_QUEUE_T *pQueueHdl;
   ```

2. Create a queue with max. 10 buffers and allocate the memory for all buffers. The size of each buffer is 20 bytes. The queue has to be created by function goal_queueInit() in the state GOAL_FSA_INIT.

```
goal_queueInit(&pQueueHdl, 10, 10, 20);
```

3. Create a handle for a buffer:

```
GOAL_BUFFER_T *pBuf;
```

4. Take an uninitialized buffer from the queue:

```
goal_queueGetElem(pQueueHdl, &pBuf);
```

5. Initialize the buffer and mark the buffer as used:

```
pBuf->dataLen = 0;
pBuf->usage = GOAL_ID_QUEUE;
pBuf->relCb = NULL;  /* no callback function shall be called */
pBuf->pQueue = pQueueHdl;  /* return the buffer to the same queue */
pBuf->flags = GOAL_QUEUE_FLG_USED;
```

6. Use the buffer application-specific.

7. Return the buffer to the same queue:

```
goal_queueAddElem(pQueueHdl, pBuf);
```

5.12.5.2 Get an initialized buffer from the queue and release the buffer without a callback function

1. Create a handle for the queue:

```
GOAL_QUEUE_T *pQueueHdl;
```

2. Create a queue with max. 10 buffers and allocate the memory for all buffers. The size of each buffer is 20 bytes. The queue has to be created by function goal_queueInit() in the state GOAL_FSA_INIT.

```
goal_queueInit(&pQueueHdl, 10, 10, 20);
```

3. Create a handle for a buffer:

```
GOAL_BUFFER_T *pBuf;
```

4. Take an initialized buffer from the queue. The same queue is specified as return queue. No callback function is specified.

```
goal_queueGetNewBuf(&pBuf, pQueueHdl, GOAL_ID_QUEUE);
```

5. Use the buffer application-specific.

6. Release the buffer to the same queue:

```
goal_queueReleaseBuf(&pBuf);
```

5.12.5.3 Get an initialized buffer from the queue and release the buffer with a callback function

1. Create an application-specific callback function to release the buffer:

```
GOAL_STATUS_T cbQueueRelFunc(struct GOAL_BUFFER_T *pBuf, void *pArg) {
    …
    goal_queueReleaseBuf(&pBuf);
}
```

2. Create a handle for the queue:

```
GOAL_QUEUE_T *pQueueHdl;
```

3. Create a queue with max. 10 buffers and allocate the memory for all buffers. The size of each buffer is 20 bytes. The queue has to be created by function goal_queueInit() in the state GOAL_FSA_INIT.

```
goal_queueInit(&pQueueHdl, 10, 10, 20);
```

4. Create a handle for a buffer:

```
GOAL_BUFFER_T *pBuf;
```

5. Take an initialized buffer from the queue. The same queue is specified as return queue. No callback function is specified.

```
goal_queueGetNewBuf(&pBuf, pQueueHdl, GOAL_ID_QUEUE);
```

6. Register the callback function without arguments for the buffer:

```
goal_queueSetReleaseCallback(pBuf,  cbQueueRelFunc, NULL);
```

7. Use the buffer application-specific.

8. Return the buffer into the queue by function goal_queueReleaseBuf(). The function goal_queueReleaseBuf() calls the callback function and the buffer is returned to the queue.

```
goal_queueReleaseBuf(&pBuf);
```

5.13 Ring buffer

This GOAL core module provides functions for ring buffers. Data of different byte length can be stored in or loaded from the ring buffer. The access to the ring buffer is protected by the GOAL locking mechanism.

A fast writing is supported. This means the GOAL locking mechanism is only applied once. The following sequence allows the fast writing:

1. start writing by function goal_rbPut() and set the parameter flgLockKeep to GOAL_TRUE, the ring buffer remains locked
2. continue writing for all data by function goal_rbPutFast()
3. release the lock by function goal_rbPutFastFinish()

GOAL files:
goal_rb.[h,c]

example:
..\goal\appl\00410_goal\rb

5.14 Task abstraction layer

This GOAL core module connects the specific operating system to other GOAL components via a generic abstraction layer. The task abstraction layer allows to create and shutdown a task, to configure the task priority and to handle the state machine for the task and requires indication functions containing the special operating system functions. The indication functions are described in chapter 5.14.2.

Task priorities are generalized to the following categories:

- GOAL_TASK_PRIO_LOWEST
- GOAL_TASK_PRIO_MEDIUM
- GOAL_TASK_PRIO_HIGHEST

GOAL files:
goal_task.[h,c]

example:
…\goal\appl\00410_goal\task

5.14.1 Configuration

The following compiler-defines are available to configure the task abstraction layer:

GOAL_CONFIG_TASK:
0: task abstraction layer is disabled (default)
1: task abstraction layer is enabled

5.14.2 Platform API

GOAL requires the following indication function to connect a specific operating system to the task abstraction layer:

Prototype	`GOAL_STATUS_T goal_tgtTaskCreate(GOAL_TASK_T *pTask)`	
Description	This indication function allows to create a task specified by the task-handle.	
Parameters	pTask	handle for the task
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskStart(GOAL_TASK_T *pTask)`	
Description	This indication function allows to start the task specified by the task-handle.	
Parameters	pTask	handle for the task
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskExit(void)`
Description	This indication function allows to shutdown the current task.
Parameters	none
Return values	GOAL return status, see chapter 8.3
Category	mandatory

Prototype	`GOAL_STATUS_T goal_tgtTaskMsSleep(uint32_t msReq, uint32_t *pMsRem)`	
Description	This indication function allows to put the current task to sleep.	
Parameters	msReq	time in ms to sleep
	pMsRem	returns the remaining time in ms if sleep was interrupted and this function is available on the specific operating system
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskTestSelf(GOAL_TASK_T *pTask)`	
Description	This indication function allows to check if the ID of the current task matches to the task-handle.	
Parameters	pTask	handle for the task
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskPrioSet(GOAL_TASK_T *pTask, uint32_t prio)`	
Description	This indication function allows to configure the priority of the specified task.	
Parameters	pTask	handle for the task
	Prio	desired priority of the task: • GOAL_TASK_PRIO_LOWEST • GOAL_TASK_PRIO_MEDIUM • GOAL_TASK_PRIO_HIGHEST
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskSuspend(GOAL_TASK_T *pTask)`	
Description	This indication function allows to suspend the execution of the task specified by the task-handle.	
Parameters	pTask	handle for the task
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

Prototype	`GOAL_STATUS_T goal_tgtTaskResume(GOAL_TASK_T *pTask)`	
Description	This indication function allows to resume the execution of the task specified by the task-handle.	
Parameters	pTask	handle for the task
Return values	GOAL return status, see chapter 8.3	
Category	mandatory	

5.15 Timer

This GOAL core module provides functionalities for:
- hard timers with an operating system (Figure 18),
- hard timers without an operating system (Figure 19) and
- soft timers (Figure 20).

Hard timers are high prioritized and handled interrupt-controlled or operating system-specific. Soft timers are low prioritized and handled loop-controlled. Both kinds of timer base on platform-specific timers. The value range of the timers depends on the platform-specific timer configuration. The standard GOAL system requires a minimal time period of 1 ms.
The accesses to the GOAL timers are protected by the GOAL locking mechanism.

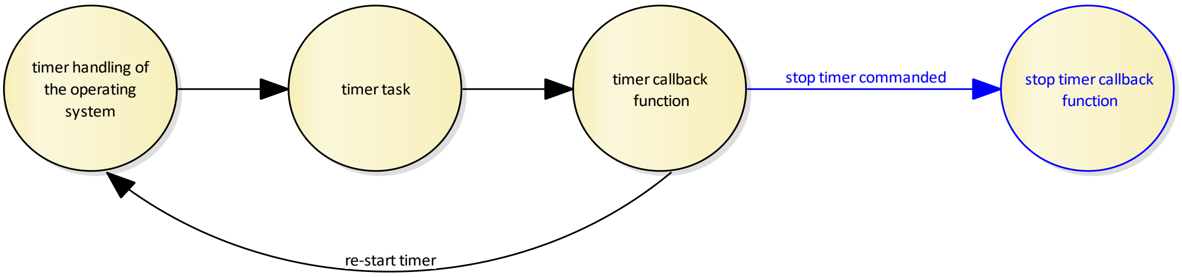


Figure 18: typical case for hard timer with operating system

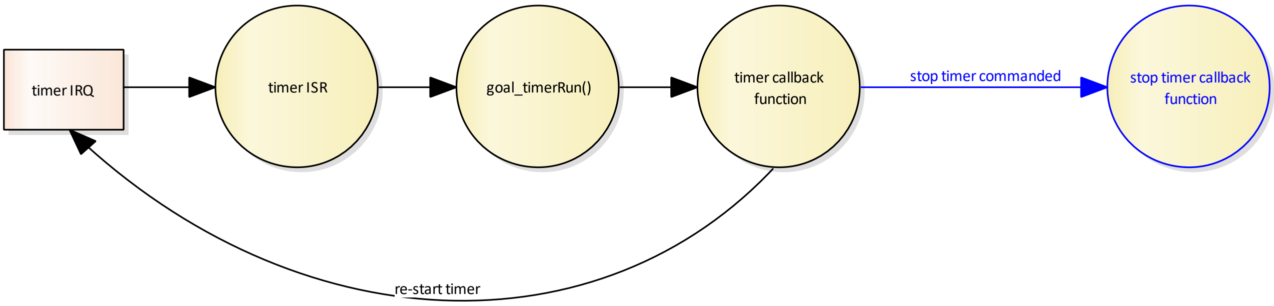


Figure 19: typical case for hard timer without operating system

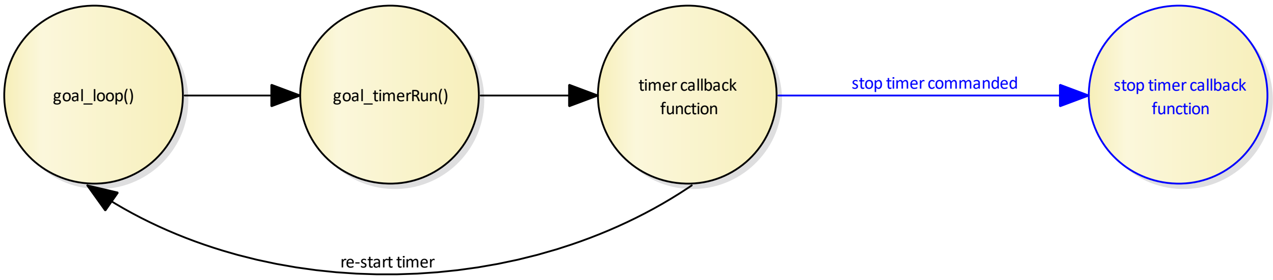


Figure 20: soft timer handling

The timers can be used as:

- single shot timer or
- periodic timer

The timer type can be configured by function goal_timerSetup().

GOAL files:
goal_timer.[h,c], goal_timer_cli.c

example:
not available

5.15.1 Callback functions

There are the following callback functions to connect the timer with other functionality:

Prototype	`void cbTimerFunc(void *pArg)`	
Description	This callback function is always called if the current time stamp is captured.	
Parameter	pArg	specific arguments used by the callback function
Return values	none	
Category	Mandatory	
Registration	during runtime via function goal_timerSetup()	

Prototype	`void cbStopTimerFunc(void *pArg)`	
Description	This function is called once if the timer is stopped.	
Parameter	pArg	specific arguments used by the callback function
Return values	none	
Category	optional If a callback function is not available, GOAL deletes the timer.	
Registration	during runtime via function goal_timerStopCb()	

5.15.2 Platform API

GOAL requires the following indication function to manage a platform-specific timer as base for GOAL timers:

Prototype	`GOAL_STATUS_T goal_targetTimerInit(void)`
Description	This indication function initializes a platform-specific timer and is called in stage GOAL_STAGE_TIMER_PRE in the state GOAL_FSA_INIT_GOAL.
Parameters	none
Return values	GOAL return status, see chapter 8.3
Category	mandatory

Condition	none

Prototype	`GOAL_STATUS_T goal_targetTimerCreate(GOAL_TIMER_T *pTmr)`	
Description	This indication function creates a platform-specific timer for a GOAL hard timer.	
Parameters	pTmr	handle for the GOAL hard timer
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL hard timers	
Condition	none	

Prototype	`GOAL_STATUS_T goal_targetTimerDelete(GOAL_TIMER_T *pTmr)`	
Description	This indication function deletes a platform-specific timer for a GOAL hard timer.	
Parameters	pTmr	handle for the GOAL hard timer
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL hard timers	
Condition	none	

Prototype	`GOAL_STATUS_T goal_targetTimerStart(GOAL_TIMER_T *pTmr)`	
Description	This indication function starts a platform-specific timer for a GOAL hard timer.	
Parameters	pTmr	handle for the GOAL hard timer
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL hard timers	
Condition	none	

Prototype	`GOAL_STATUS_T goal_targetTimerStop(GOAL_TIMER_T *pTmr)`	
Description	This indication function stops a platform-specific timer for a GOAL hard timer.	
Parameters	pTmr	handle for the GOAL hard timer
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL hard timers	
Condition	none	

5.15.3 Command line interface

Command	`time current`
Description	Prints the current timestamp of the GOAL system to the command line interface.
Parameter	`none`

5.15.4 Implementation guidelines

5.15.4.1 Use a periodic soft timer and start the timer immediately

1. Create a handle for the timer:

```
GOAL_TIMER_T *pSoftTimer;
```

2. Create a soft timer of low priority in state GOAL_FSA_INIT:

```
goal_timerCreate(&pSoftTimer, GOAL_TIMER_LOW);
```

3. The soft timer shall be triggered every 1000 ms periodically and shall be started immediately. The timer is configured as follow:

```
goal_timerSetup(pSoftTimer, GOAL_TIMER_PERIODIC, 1000, cbTimerFunc, NULL,
GOAL_TRUE);
```

4. The callback function is called if the timer is expired again and again.

```
goal_timerCreate(&pSoftTimer, GOAL_TIMER_LOW);
goal_timerSetup(pSoftTimer, GOAL_TIMER_PERIODIC, 1000, cbTimerFunc, NULL,
GOAL_TRUE);
```

5. Stop the timer without calling a callback function after it is expired the next time:

```
goal_timerStop(pSoftTimer);
```

6. Delete the timer:

```
goal_timerDelete(&pSoftTimer);
```

5.15.4.2 Use a single soft timer and start the timer in the application

1. Create a handle for the timer:

```
GOAL_TIMER_T *pSoftTimer;
```

2. Create a soft timer of low priority in state GOAL_FSA_INIT:

```
goal_timerCreate(&pSoftTimer, GOAL_TIMER_LOW);
```

3. The soft timer shall be triggered only once after 1000 ms and shall be started by the application. The timer is configured as follow:

```
goal_timerSetup(pSoftTimer, GOAL_TIMER_SINGLE, 1000, cbTimerFunc, NULL,
GOAL_FALSE);
```

4. Start the timer in the application:

```
goal_timerStart(pSoftTimer);
```

5. Stop the timer without calling a callback function:

```
goal_timerStop(pSoftTimer);
```

6. Delete the timer:

```
goal_timerDelete(&pSoftTimer);
```

5.15.4.3 Stop hard timer with callback function

1. Create a handle for the timer:

```
GOAL_TIMER_T *pHardTimer;
```

2. Create a hard timer of high priority in state GOAL_FSA_INIT:

```
goal_timerCreate(&pHardTimer, GOAL_TIMER_HIGH);
```

3. The hard timer shall be triggered every 1000 ms periodically and shall be started immediately. The timer is configured as follow:

```
goal_timerSetup(pHardTimer, GOAL_TIMER_PERIODIC, 1000, cbTimerFunc, NULL,
GOAL_TRUE);
```

4. The callback function is called if the timer is expired again and again.

5. Stop the timer with calling a callback function:

```
goal_timerStopCb(pHardTimer, cbStopTimerFunc, NULL);
```

6. Delete the timer:

```
goal_timerDelete(&pHardTimer);
```

5.16 Tracing

This GOAL core module provides macros for tracing data via a configurable interface. Helpful for getting additional information about the system on debugging or setting reference pins for e.g. timing analysis. Tracing data is disabled by default.

Macro	GOAL_TGT_TRACE8(_chan, _data)

Description	tracing an 8-bit value	
Parameters	_chan	output channel
	_data	data value
Category	Optional	
Condition	Compiler-define GOAL_CONFIG_TGT_TRACE must be set to 1 and a tracing interface has to be enabled.	

Macro	GOAL_TGT_TRACE16 (_chan, _data)	
Description	tracing a 16-bit value	
Parameters	_chan	output channel
	_data	data value
Category	Optional	
Condition	Compiler-define GOAL_CONFIG_TGT_TRACE must be set to 1 and a tracing interface has to be enabled.	

Macro	GOAL_TGT_TRACE32(_chan, _data)	
Description	tracing a 32-bit value	
Parameters	_chan	output channel
	_data	data value
Category	Optional	
Condition	Compiler-define GOAL_CONFIG_TGT_TRACE must be set to 1 and a tracing interface has to be enabled.	

Macro	GOAL_TGT_TRACE_BIT_SET(_chan, _bit)	
Description	Setting a single bit on the tracing data. Keeping the other bits unchanged. This feature is available for tracing via pin only.	
Parameters	_chan	output channel
	_bit	bit position
Category	Optional	
Condition	Compiler-define GOAL_CONFIG_TGT_TRACE and GOAL_CONFIG_TGT_TRACE_PIN must be set to 1.	

Macro	GOAL_TGT_TRACE_BIT_CLR(_chan, _bit)	
Description	Clearing a single bit on the tracing data. Keeping the other bits unchanged. This feature is available for tracing via pin only.	
Parameters	_chan	output channel
	_bit	bit position
Category	Optional	
Condition	Compiler-define GOAL_CONFIG_TGT_TRACE and GOAL_CONFIG_TGT_TRACE_PIN must be set to 1.	

5.16.1 Tracing via ITM

The Instrumentation Trace Macrocell (ITM) is a special ARM feature providing a tracing interface for output data via debugger.
Handling single data bits by GOAL_TGT_TRACE_BIT_SET or GOAL_TGT_TRACE_BIT_CLR is not implemented at this version.

5.16.2 Tracing via pin

Outputs the data on pins. The number and choice of pins is board specific and may not be available on all systems.

⚠ Please verify, that the configured tracing pins are free to use before enabling this GOAL feature.

There are no different output channels, so the *_chan* argument at the macros will not be considered.

5.16.3 Configuration

The following defines enable the tracing module and its interface.

- GOAL_CONFIG_TGT_TRACE
  - 0: tracing is switched off for the complete GOAL system (default)
  - 1: tracing is switched on for the complete GOAL system
- GOAL_CONFIG_TGT_TRACE_PIN
  - 0: tracing the data via board specific pins is switched off (default)
  - 1: tracing the data via board specific pins is switched on. The number and choice of pins is configured the board. Please read section *5.16.2 Tracing via pin* before enabling this feature.
- GOAL_CONFIG_TGT_TRACE_ITM:
  - 0: tracing the data via ITM is switched off (default)
  - 1: tracing the data via ITM is switched on.
- GOAL_CONFIG_TGT_TRACE_ITM_WITHOUT_PC:
  - 0: tracing the data via ITM with no additional information about the program counter (default)
  - 1: tracing the data via ITM next to the program counter when tracing

5.17 Utility functions

This GOAL core module provides utility functions for the GOAL system for:

- the CRC calculation according to the Fletcher-32 algorithm
- the generation of random values

GOAL files:

goal_util.[h,c]

6 GOAL media (goal_media)

The directory goal_media contains:

- media adapters: generic driver interfaces
- media interfaces: generic interfaces between media adapters and higher layers

One source and one header files exist for each GOAL media module. Only the sources for the necessary GOAL media modules shall be integrated in the compiler-project of the GOAL system. The registration is described in chapter 4.2.2. The functions are described in detail in the GOAL Reference Manual.
Figure 21 demonstrates the easy exchange of drivers.

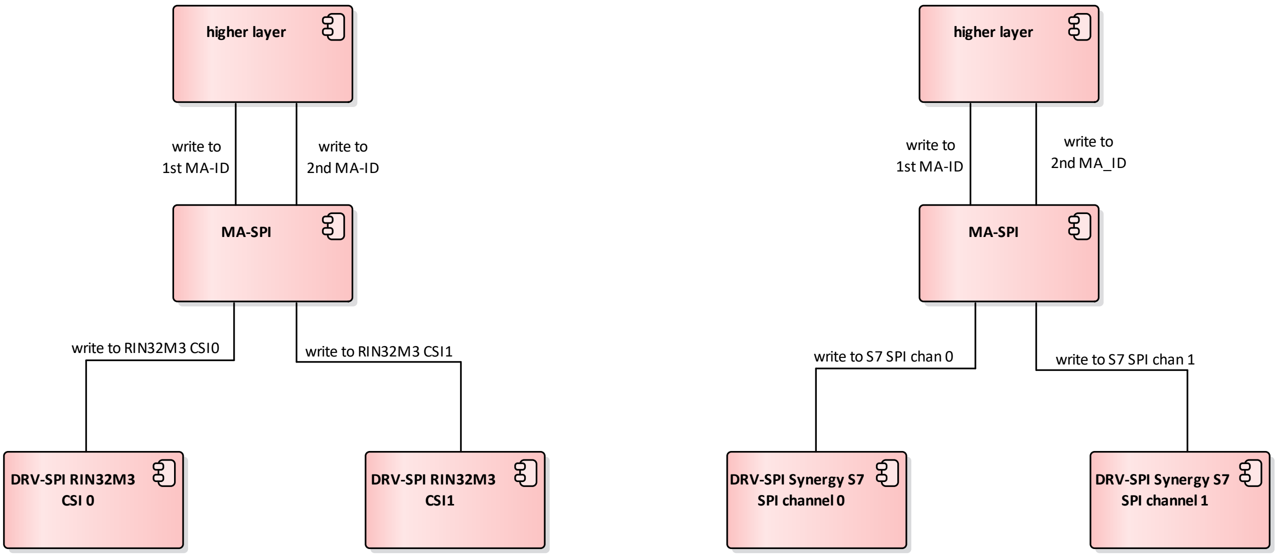


Figure 21: media adapter for SPI

6.1 Nonvolatile storage

GOAL provides a media adapter and media interface for the nonvolatile storage usable for program downloads and uploads by a bootloader or for the nonvolatile storage of configuration data. The nonvolatile storage media allows:

- to write data to the nonvolatile memory,
- to read data from the nonvolatile memory and
- to erase the nonvolatile memory.

6.1.1 NVS media interface

The media adapter is registered to the media interface by function goal_miNvsReg(). The resource "NVS media interface" must be allocated by function goal_miNvsAlloc(). The media interface is freed by function goal_miNvsFree().

The media interface allows to manage single memory ranges, called regions. Therewith it is possible to assign different memory ranges to various processes and to control the access to the nonvolatile memory process-specific. Each region is identified by an ID, called MI-NVS-REGION-ID, unique. This ID can be specified application-specific. But each ID must only exist once. During registration a unique handle is created for each MI-NVS-REGION-ID. Each region has to be registered to the media interface for nonvolatile storage by higher layers in the state GOAL_FSA_INIT. A region has the following properties:

Property of NVS region	Description
offset	start address of the memory region, value range: uint32_t
length	length of the memory region in bytes, value range: uint32_t
strName	name of the file for the nonvolatile storage about the file system for each memory region, strName is a zero-terminated string of the length of GOAL_MI_REGION_NAME_LENGTH in bytes (default: 255 byte)
mode	storage mode: • GOAL_MI_NVS_REGION_MODE_COMPLETE: load/save the complete memory region • GOAL_MI_NVS_REGION_MODE_STREAM: load/save single data within the memory region, data is addressable about an additional offset • GOAL_MI_NVS_REGION_MODE_BUFFERED: load/save to this region is handled through a memory buffer. Writing to the physical medium is decoupled by sequentially writing elements.
access	access right at the region: • GOAL_MI_NVS_REGION_ACCESS_READ: region is only readable • GOAL_MI_NVS_REGION_ACCESS_WRITE: region is writable and readable

Table 8: properties of NVS regions

6.1.1.1 *Implementation guidelines*

6.1.1.1.1 Registration of a memory region

1. Specify a region and define a MI-NVS-REGION-ID:
   ```
   #define GOAL_ID_MI_NVS_REGION_CONFIG_DATA 2
   ```

2. Create a MA-handle:
   ```
   GOAL_MA_NVS_T *pMaNvs;
   ```

3. Select the suitable NVS driver and initialize the driver. The driver registers to the media adapter by itself.

4. Create a MI-handle:

```
GOAL_MI_NVS_T *pMiNvsHdl;
```

5. Register the media interface:

```
goal_miNvsReg(&pMiNvsHdl);
```

6. Allocate the NVS service:

```
goal_miNvsAlloc(pMiNvsHdl);
```

7. Create a MI-NVS-REGION-handle:

```
GOAL_MI_NVS_REGION_T *pRegion;
```

8. Register and configure the memory region: The memory range starts at address 0x0001FFF and has a length of 0x100 byte. The region shall complete. Configurtion data shall be read and written.

```
    goal_miNvsRegRegion(&pRegion, pMiNvsHdl, pMaNvsHdl, 0x0001FFF, 0x00000100,
                        "config data",
                        GOAL_ID_MI_NVS_REGION_CONFIG_DATA);

    /* set mode */
    goal_miNvsSetMode(pRegion, GOAL_MI_NVS_REGION_MODE_COMPLETE);

    /* set access rights */
    goal_miNvsSetAccess(pRegion, GOAL_MI_NVS_REGION_ACCESS_WRITE);
}
```

6.1.1.1.2 Write data to nonvolatile memory

1. Load MI-NVS-REGION-handle of the memory region with the ID GOAL_ID_MI_NVS_REGION_CONFIG_DATA:

```
GOAL_MI_NVS_REGION_T *pRegion;
goal_miNvsRegionGetById(&pRegion, GOAL_ID_MI_NVS_REGION_CONFIG_DATA);
```

2. Erase the nonvolatile memory region:

```
goal_miNvsErase(pRegion);
```

3. Write data of size bytes to nonvolatile memory region:

```
goal_miNvsWrite(pRegion, (uint8_t *)pData,0, size);
```

6.1.1.1.3 Read data from nonvolatile memory

1. Load MI-NVS-REGION-handle of the memory region with the ID GOAL_ID_MI_NVS_REGION_CONFIG_DATA:

```
GOAL_MI_NVS_REGION_T *pRegion;
goal_miNvsRegionGetById(&pRegion, GOAL_ID_MI_NVS_REGION_CONFIG_DATA);
```

2. Read data from nonvolatile memory:

```
goal_miNvsRead(&pRegion, (uint8_t *)pData, 0, size);
```

6.1.2 NVS media adapter

The selected NVS driver registers itself to the NVS media adapter.

6.1.2.1 Implementation guidelines

These implementation guidelines refer to the case, that no NVS media interface is used.

6.1.2.1.1 Write data to nonvolatile memory

1. Create a MA-handle:
   ```
   GOAL_MA_NVS_T *pMaNvsHdl;
   ```

2. Select the suitable NVS driver and initialize the driver. The driver registers to the media adapter by itself.

3. Create a NVS description:
   ```
   GOAL_MA_NVS_DESC_T desc = {
       .strName = „config data“,
       .fCompleteAccess = GOAL_TRUE}
   ```

4. Erase 0x100 bytes in the nonvolatile memory from start address 0x0001FFF:
   ```
   goal_maNvsErase(pMaNvsHdl, &desc, 0x0001FFF, 0x100);
   ```

5. Write 0x100 bytes from the buffer pData to the nonvolatile memory on start address 0x0001FFF:
   ```
   goal_maNvsWrite(pMaNvsHdl, &desc, 0x0001FFF, (uint8_t *)pData, 0x100);
   ```

6.1.2.1.2 Read data from nonvolatile memory

1. Create a MA-handle:
   ```
   GOAL_MA_NVS_T *pMaNvsHdl;
   ```

2. Select the suitable NVS driver and initialize the driver. The driver registers to the media adapter by itself.

3. Create a NVS description:
   ```
   GOAL_MA_NVS_DESC_T desc = {
       .strName = „config data“,
       .fCompleteAccess = GOAL_TRUE}
   ```

4. Read 0x100 bytes from the start address 0x0001FFFF in the nonvolatile memory and store the data in pData:
   ```
   goal_maNvsRead(pMaNvsHdl, &desc, 0x0001FFFF, (uint8_t *)pData, 0x100);
   ```

6.2 LED

GOAL provides a media adapter for the controlling of LEDs. Standardized communication protocols often need status LEDs. The application can also use the LED media adapter to control LEDs application-specific. The media adapter for LEDs allows to handle

- single LEDS and
- groups of LEDs

The used hardware resources for the controlling of LEDs are encapsulated in the LED driver and depends on the platform. Details are described in the suitable GOAL Platform Manual. It is possible to control the LEDs via GPIOs or about a serial bus as IIC.

The media adapter provides the following functionality:

- open/close a media adapter for a single LED or a group of LEDs,
- get/set the state of a single LED,
- get/set the state of a group of LEDs.

The get-functions require, that the current LED state is readable from the platform.

The connection between the LED driver and the LED media adapter is identified by a MA-ID unique. The determination of the MA-ID is described in the suitable GOAL Platform Manual. The most LED drivers uses a MA-ID created by the application. The application has to assign single LEDs and/or groups of LEDs to MA-IDs during the platform initialization in the state GOAL_FSA_INIT.

A group of LEDs can consist of maximal 32 LEDs. The mask and state value have data type uint32_t and are bit-coded. Each LED in the LED group shall use the same bit position in the mask and state value. The interpretation of the bit values of the LED states is platform-specific. Maybe the application has to consider the polarity of the LEDs. The bit values for the mask are defined as follow:

Bit value	Meaning for LED group mask
0	LED is ignored and remains unchanged
1	LED is changed according to the desired state bit

Table 9: mask bit coding for groups of LEDs

6.2.1 Implementation guidelines

6.2.1.1 Switch on/off and get the state of a single LED

1. Define a MA-ID for a single LED:

```
#define GOAL_MA_LED_APPL_SINGLE_LED 1
```

2. Call the LED driver function to initialize the LED hardware resource and to register the LED driver for a single LED to the LED media adapter in state GOAL_FSA_INIT.

3. Open a media adapter instance and get the MA-SPI handle:
```
GOAL_MA_LED_T *pMaLedHdl;        /*MA-LED handle */
goal_maLedOpen(GOAL_MA_LED_APPL_SINGLE_LED, &pMaLedHdl);
```

4. Switch on the LED:
```
GOAL_MA_LED_STATE_T state = GOAL_MA_LED_STATE_ON;
goal_maLedSet(pMaLedHdl, &state);
```

5. Get the current state of the LED:
```
goal_maLedGet(pMaLedHdl, &state);
```

6. Close the media adapter instance:
```
goal_maLedClose(pMaLedHdl);
```

6.2.1.2 Switch on/off and get the state of a LED group

A group of 32 LEDs shall be controlled.

1. Define a MA-ID for a group of LEDs:
```
#define GOAL_MA_LED_APPL_GROUP_LED 2
```

2. Call the LED driver function to initialize the hardware resource for all LEDs and to register the LED driver for a group of LEDs to the LED media adapter in state GOAL_FSA_INIT.

3. Open a media adapter instance and get the MA-SPI handle:
```
GOAL_MA_LED_T *pMaLedHdl;        /*MA-LED handle */
goal_maLedOpen(GOAL_MA_LED_APPL_GROUP_LED, &pMaLedHdl);
```

4. Switch on the LEDs assigned to bit 31-24, do not change the LEDs assigned to bit 23-16, switch off LEDs assigned to bit 15-0:
```
uint32_t mask = 0xFF00FFFF;
uint32_t state = 0xFF000000;
goal_maLedGroupSet(pMaLedHdl, &mask, &state);
```

5. Get the current state of all LEDs in the LED group:
```
goal_maLedGroupGet(pMaLedHdl, &state);
```

6. Close the media adapter instance:
```
goal_maLedClose(pMaLedHdl);
```

6.3 SPI

GOAL provides a media adapter for the SPI communication. The media adapter provides the following functionality:

- open/close a media adapter for a SPI-channel
- get/set a general SPI-configuration

- read data from the SPI-bus
- write data to the SPI-bus
- write and read data to/from the SPI-bus
- report events to higher layers

GOAL defines the following general SPI configuration settings:

SPI configuration setting according to GOAL_MA_SPI_CONF_T	Description
type	type of the SPI communication: • GOAL_MA_SPI_TYPE_MASTER, • GOAL_MA_SPI_TYPE_SLAVE
mode	combination of clock polarity and phase as SPI mode: • GOAL_MA_SPI_MODE_0, • GOAL_MA_SPI_MODE_1, • GOAL_MA_SPI_MODE_2, • GOAL_MA_SPI_MODE_3
bitrate	SPI baudrate in Hz
unitsize	size of transferred data must be a multiple of unitsize: • GOAL_MA_SPI_UNITWIDTH_8BIT, • GOAL_MA_SPI_UNITWIDTH_16BIT, • GOAL_MA_SPI_UNITWIDTH_32BIT The minimal size must be equal to the data transfer length of the SPI controller at least.
bitorder	bit order of the transferred data via the SPI bus: • GOAL_MA_SPI_BITORDER_MSB, • GOAL_MA_SPI_BITORDER_LSB

Table 10: general SPI configuration settings

The SPI-configuration can be set by function goal_maSpiConfigSet(). The current SPI-configuration can be read by function goal_maSpiConfigGet(). The support of the SPI configuration settings depends on the SPI driver and the SPI controller. Details are described in the suitable GOAL Platform Manual.

SPI events are handled event-driven about an application-specific callback function. The supported events depend on the SPI driver and the availability on the SPI controller. GOAL provides the following events:

Event number according to GOAL_MA_SPI_EVENT_T	Description
GOAL_MA_SPI_EVENT_TRANSFER_COMPLETE	The SPI controller reports, that the data

Event number according to GOAL_MA_SPI_EVENT_T	Description
	transfer is completed.
GOAL_MA_SPI_EVENT_TRANSFER_ABORTED	The SPI controller reports, that the data transfer is aborted.
GOAL_MA_SPI_EVENT_MODE_FAULT	The SPI controller reports an error during configuration of the platform-specific SPI mode.
GOAL_MA_SPI_EVENT_READ_OVERFLOW	The SP controller reports a read overflow.
GOAL_MA_SPI_EVENT_ERR_PARITY	The SPI controller on the platform repots a parity error.
GOAL_MA_SPI_EVENT_ERR_DATA_CONSISTENCY	The SPI controller on the platform supports a data consistency check. The data consistency check is active and reports an error.
GOAL_MA_SPI_EVENT_ERR_OVERFLOW	The SPI controller works in a buffered mode and reports an overflow of the buffers.
GOAL_MA_SPI_EVENT_ERR_OVERRUN	The SPI controller reports an overrun during reception of data.
GOAL_MA_SPI_EVENT_ERR_BUF_OVERRUN	The internal SPI message buffer in the driver overflows.
GOAL_MA_SPI_EVENT_ERR_FRAMING	The SPI controller reports a framing error.
GOAL_MA_SPI_EVENT_MODE_UNDERRUN	The SPI controller reports an underrun, if it works as SPI slave and no transmission data are prepared and a serial transfer was initiated by the SPI master.

Table 11: general SPI events

The connection between the SPI driver and the SPI media adapter is identified by a MA-ID unique. The determination of the MA-ID is described in the suitable GOAL Platform Manual. The most SPI drivers determine the MA-ID by itself.

6.3.1 Callback functions

Prototype	`GOAL_STATUS_T cbMaSpiEvent(struct GOAL_MA_SPI_T *pMaSpiHdl, GOAL_MA_SPI_EVENT_T event, void *pArg)`	
Description	This callback function is called if an SPI event was occurred in the SPI driver to inform higher layers.	
Parameters	pMaSpiHdl	handle of the media adapter
	event	number of the occurred event, see Table 11
	pArg	specific arguments used by the callback function
Return values	GOAL return status, see chapter 8.3	

Category	mandatory
Registration	during runtime about function goal_maSpiOpen()

6.3.2 Implementation guidelines

6.3.2.1 Read and write data via the SPI-bus

1. Call the SPI driver function to initialize the SI controller and to register the SPI driver to the SPI media adapter in state GOAL_FSA_INIT. During this guideline GOAL_MA_ID_SPI is used to mark the MA-ID. During the registration a unique MA-SPI handle is created.

2. Implement a callback function to handle SPI events:

```
GOAL_STATUS_T cbApplMaSpiEvent(struct GOAL_MA_SPI_T *pMaSpiHdl, GOAL_MA_SPI_EVENT_T
event, void *pArg) {
    …
}
```

3. Open the media adapter, specify the callback function to handle SPI events and get the MA-SPI handle:

```
GOAL_MA_SPI_T *pMaSpiHdl;
goal_maSpiOpen(GOL_MA_SPI_ID, &pMaSpiHdl, cbApplMaSpiEvent, NULL);
```

4. Write 4 byte stored in pData to the SPI-bus:

```
goal_maSpiWrite(pMaSpiHdl, (uint8_t *)pData, 4);
```

5. Read data from the SPI-bus and store the data to pData:

```
uint16_t len;  /* length of read data in bytes */
goal_maSpiRead(pMaSpiHdl, (uint8_t *)pData, &len);
```

6. Write 4 byte stored in pWriteData to the SPI-bus and read data from SPI-bus and store the read data to pReadData at the same time:

```
unt16_t len;  /* length of data to write as input parameter and length of read data
as output in bytes */
len = 4;
goal_maSpiWriteRead(pMaSpiHdl, pWriteData, pReadData, &len);
```

6.3.2.2 Configure the SPI interface

1. Get the current general SPI configuration of an opened MA:

```
GOAL_MA_SPI_CONF_T spiConfig;
goal_maSpiConfigGet(pMaSpiHdl, &spiConfig);
```

2. Specify SPI mode 0:

```
spiConfig.mode = GOAL_MA_SPI_MODE_0;
```

3. Set SPI configuration:

```
goal_maSpiConfigSet(pMaSpiHdl, &spiConfig);
```

6.3.2.3 Handle SPI events

1. Call the SPI driver function to initialize the SI controller and to register the SPI driver to the SPI media adapter in state GOAL_FSA_INIT. During this guideline GOAL_MA_ID_SPI is used to mark the MA-ID. During the registration a unique MA-SPI handle is created.

2. Implement a callback function to handle SPI events:

```
GOAL_STATUS_T cbApplMaSpiEvent(struct GOAL_MA_SPI_T *pMaSpiHdl, GOAL_MA_SPI_EVENT_T
event, void *pArg) {
    if (GOAL_MA_SPI_EVENT_ERR_OVERRUN == event) {
        …
    }
    else if (GOAL_MA_SPI_EVENT_ERR_PARITY == event) {
        …
    }
    else {
        /* handle unknown events application-specific */
    }
}
```

3. Open the media adapter, specify the callback function to handle SPI events and get the MA-SPI handle:

```
GOAL_MA_SPI_T *pMaSpiHdl;
goal_maSpiOpen(GOL_MA_SPI_ID, &pMaSpiHdl, cbApplMaSpiEvent, NULL);
```

4. If a SPI event occurs, the callback function cbApplMaSpiEvent is called.

6.4 TLS

GOAL provides a functionality for encryption and authentication of TCP packets on the base of the Transport Layer Security (TLS) protocol /TLS_RFC_5246/. The functionality of TLS requires:

- a TLS library,
- a GOAL driver for the integration of the TLS library into the GOAL system and
- a GOAL media adapter for TLS in order to use a generic interface for TLS in the application.

GOAL allows to implement various libraries for cryptographic and transport layer security capabilities. The GOAL TLS media adapter makes it possible to exchange the TLS library with less effort. The TLS functionality is embedded into the GOAL core module for the network handling.

The TLS functionality comprises:

- encryption/decryption of TCP packets and
- the authentication by a X509-certificate.

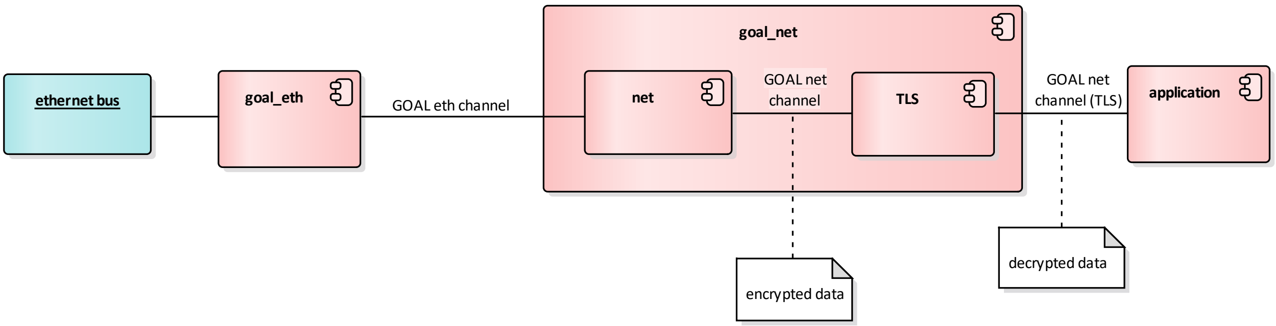


Figure 22: integration of TLS

The authentication is realized by a X509-certificate. The application can specify an own private key and an own X509-certificate by function goal_maTlsOpen(). The private key must have a length between 1024 bit and 2048 bit. If no X509-certificate is specified a default certificate is taken. The default certificate is *port*-specific.

The GOAL TLS media adapter allows:

- to open/close a GOAL TLS channel and
- to get information from the X509-certificate about the certification authority, the organization providing the web-server and the validity period

The encryption and decryption are made by the TLS library internally.

example:

…\goal\appl\00410_goal\tls*

6.4.1 Configuration

The following compiler-defines are available to configure TLS:

GOAL_TLS:
0: TLS is disabled (default)
1: TLS is enabled

6.4.2 mbed TLS library

GOAL supports the open source library mbed TLS. The following sources must be added to the compiler-project:

source	location
mbed TLS library	…\goal\ext\mbedtls*
GOAL driver for the mbed TLS library	…\goal\plat\drv\tls\mbedtls

The GOAL driver for the mbed TLS library provides the following function for the registration to the GOAL TLS media adapter:

Prototype	`GOAL_STATUS_T goal_tlsMbedtlsInit(GOAL_MA_TLS_T **ppTlsHdl, unsigned int maId)`	
Description	This function registers the GOAL driver for the mbed TLS library, i.e. the driver functions for initialization, opening a TLS channel and getting information from X509v3-certificate are made known in the GOAL TLS media adapter.	
Parameters	ppTlsHdl	handle for the TLS instance
	maId	MA-ID for the TLS instance
Return values	GOAL return status, see chapter 8.3	
Calling	in state GOAL_FSA_INIT_GOAL, stage GOAL_STAGE_TARGET_PRE (normally during the board initialization, see goal_target_board.c/goal_targetBoardInit())	

The application has to specify a MA-ID. There is no driver-specific rule for the construction of the MA-ID.

The execution of the algorithm for encryption/decryption needs some time and is processed in an own task to allow, that the algorithm can be interrupted by functions with higher priority. This method requires an operating system.

The function goal_maTlsInit() installs the initialization function of the GOAL TLS driver in the staging table. The initialization function of the GOAL TLS driver is called by GOAL in state GOAL_FSA_INIT_GOAL in stage GOAL_STAGE_MODULES.

The opening function of the GOAL TLS driver is executed by function goal_maTlsOpen(). The application has to call the function goal_maTlsOpen() in the state GOAL_FSA_INIT_SETUP to initialize and to open the channels.

The function for getting information from the X509-certificate the GOAL TLS driver function is mapped to the function goal_maTlsReadInfo(). The function goal_maTlsReadInfo() is called by the application in the state GOAL_FSA_OPERATION.

6.4.3 Implementation guidelines

6.4.3.1 Initialize TLS

This example uses the mbed TLS library.

1. Define a MA-ID.

```
#define APPL_MA_TLS_ID 1
```

2. Integrate the initialization of the GOAL TLS media adapter in the stage GOAL_STAGE_MODULES in application-specific function appl_init().

```
GOAL_STATUS_T appl_init (void) {
    goal_maTlsInit(APPL_MA_TLS_ID);
}
```

3. Create a handle for the TLS instance.

```
GOAL_MA_TLS_T *pTlsHdl;
```

4. Execute the specific function to register the GOAL driver for the selected TLS library to the GOAL TLS media adapter. Normally this function is called during the initialization of the board in state GOAL_FSA_INIT_GOAL in stage GOAL_STAGE_TARGET_PRE.

```
goal_tlsMbedtlsInit(&pTlsHdl, APPL_MA_TLS_ID);
```

5. Create a GOAL net channel for the output of the GOAL TLS media adapter.

```
GOAL_NET_CHAN_T pNetChanHdl;
GOAL_NET_ADDR_T netChanAddr;
GOAL_MEMSET(&netChanAddr, 0, sizeof(GOAL_NET_ADDR_T);
addr.localPort = 443;
goal_netOpen(&pNetChanHdl, &netChanAddr, GOAL_NET_TCP_LISTENER, NULL);
```

6. Configure the GOAL net channel as non-blocking.

```
uint32_t optVal = 1;
goal_netSetOption(pNetChanHdl, GOAL_NET_OPTION_NONBLOCK, &optVal);
```

7. Get the handle for the TLS channel determined by the MA-ID and open the TLS channel. GOAL_NET_CHAN_T pTlsChanHdl;

```
goal_maTlsGetById(&pTlsHdl, APPL_MA_TLS_ID);
goal_maTlsOpen(pTlsHdl, NULL, &pTlsChanHdl, pNetChanHdl);
```

8. Connect the GOAL net channel (TLS) to the GOAL net channel and specify a callback function to handle packets from the TCP client.

```
goal_netOpenTunnel(NULL, pTlsChanHdl, applTlsClearDataCb, NULL, NULL);
```

9. Create a callback function to handle packets from the TCP client, see chapter 5.11.2 function cbNetFunc().

```
void applTlsClearDataCb(GOAL_NET_CB_TYPE_T cbType, GOAL_NET_CHAN_T *pChan, struct
```

```
GOAL_BUFFER_T *pBuf) {
…
}
```

6.4.3.2 Use a TLS channel

1. TCP/IP packets are transmitted and received encrypted. Only valid TCP/IP packages pass the TLS module.

2. Information from the certificate can be required.

```
uint8_t certInfo[128];
goal_maTlsReadInfo(pTlsHdl, GOAL_CERTINFO_CA_CN, certInfo, 128);
```

6.5 CMFS

CMFS is a media interface working on top of the NVS media interface. It requires 2 serparate NVS regions for storing CM variables. Despite the plain CM implementation, which stores the whole CM variable store as a binary blob in flash, CMFS only writes modifications to the NVS region. Thereby the NVS region is sequientially written, thus a time consuming erase of the NVS region is not required. However if the NVS region is nearly fully written, the current state of variables is transferred to the secondary NVS region, where all continuing write operations take place.

This CMFS has some advantages over the plan CM implementation:
- NVS write operations can be performed much faster
- Data loss during reset while NVS is written can be omitted

To achive this, more NVS storage space needs to be reserved.

When CMFS is enabled, the NVS region with ID GOAL_ID_MI_NVS_REGION_CMCONFIG is not required anymore.

6.5.1 Integration of CMFS

Following except from goal_target_board.c shows integration of CMFS.

```
#include <goal_media/goal_mi_cmfs.h>

static GOAL_MI_NVS_REGION_LIST_T region_list[] = {
    {   /* CMFS Storage Region 1 */
        .posStart = 0x01FC0000,
        .length = 0x00020000, /* 128k kByte */
        .strName = "goal_cmfs_nvs1.bin",
        .id = GOAL_ID_MI_NVS_REGION_CMFS1,
        .mode = GOAL_MI_NVS_REGION_MODE_STREAM,
        .access = GOAL_MI_NVS_REGION_ACCESS_WRITE
    },
    {   /* CMFS Storage Region 2 */
        .posStart = 0x01FE0000,
```

```
        .length = 0x00020000, /* 128k kByte */
        .strName = "goal_cmfs_nvs2.bin",
        .id = GOAL_ID_MI_NVS_REGION_CMFS2,
        .mode = GOAL_MI_NVS_REGION_MODE_STREAM,
        .access = GOAL_MI_NVS_REGION_ACCESS_WRITE
    }
};

/****************************************************************************/
/** Board init
 *
 * Low level board initialization.
 *
 * @return GOAL_OK - success
 * @return GOAL_ERR_BOARD_INIT - error initializing board
 */
GOAL_STATUS_T goal_targetBoardInit(
    void
)
{
    GOAL_STATUS_T res = GOAL_OK;                /* result */
    GOAL_MI_CMFS_T *pMiCmfs = NULL;             /* CMFS handle */
    GOAL_MI_NVS_T *pMiNvs;                      /* NVS MI handle */

    /* register NVS MI with regions */
    if (GOAL_RES_OK(res)) {
        /* register a new nvs MI */
        res = goal_miNvsReg(&pMiNvs, GOAL_ID_MA_NVS_EEPROM_ETHERCAT,
eeprom_region_list, ARRAY_ELEMENTS(eeprom_region_list));
    }

    /* register CMFS */
    if (GOAL_RES_OK(res)) {
        res = goal_miCmfsReg(&pMiCmfs, 0);
    }

    /* register first region to CMFS */
    if (GOAL_RES_OK(res)) {
        res = goal_miCmfsRegRegion(pMiCmfs, GOAL_ID_MI_NVS_REGION_CMFS1);
    }

    /* register second region to CMFS */
    if (GOAL_RES_OK(res)) {
        res = goal_miCmfsRegRegion(pMiCmfs, GOAL_ID_MI_NVS_REGION_CMFS2);
    }

    return res;
}
```

In order to utilize CMFS, the following configuration option must be enabled.

GOAL_CONFIG_MEDIA_MI_CMFS:
0: CMFS is not utilized (default)
1: CMFS is utilized

This configuration option and the required files are added when the following feature is enabled in the Makefile, when using the GOAL build system:

```
CONFIG_MAKE_FEAT_MEDIA_MI_CMFS = 1
```

7 GOAL extension modules (protos)

Different kinds of GOAL extension modules can be divided:

- libraries for communication profiles provided by port GmbH
- more complex function blocks

7.1 Device Detection (DD)

The Device Detection represents a public interface to read and write CM-variables by other external components or remote devices. It is projected only for development and initial configuration purposes. During normal operation the Device Detection shall be disabled. The source code is located in the directory …\goal\protos\dd.

The Device Manager tool provides a graphical user interface to read and write CM-variables by a host computer using the Device Detection. This chapter describes a GOAL device used as counterpart to the Device Manager tool.

The Device Detection works according to the producer/consumer model, i.e. a DD-request of the DD-producer is received by one or more DD-consumers and each DD-consumer transmits a DD-response. Each DD-request must be answered by one or more DD-responses.

The data transfer between the DD-producer and DD-consumers is realized via a TCP/IP connection using the UDP protocol. All UDP datagrams are transmitted as broadcast packets to be independent on the IP configuration.
The data in the UDP datagram contains a DD-packet, which is coded according to the Device Detection protocol. The Device Detection protocol allows:

- to build groups of devices via a DD-customer-ID,
- to assign DD-requests and DD-responses to unique devices via the MAC address and
- to assign DD-requests to DD-responses.

Each DD-consumer can be assigned to a group by a DD-customer-ID. The DD-packet involves the DD-customer-ID of the group, which shall receive the DD-packet. The DD-customer-ID allows a filtering of the received DD-packets. The DD-customer-ID can be configured about the CM-variable DD_CM_VAR_CUSTOMERID. On the local device the CM-variable DD_CM_VAR_CUSTOMERID can be set by function goal_ddCfgCustomerID().
The DD-customer-ID 0 disables the filtering of the received DD-packets. A DD-consumer with the DD-customer-ID 0 accepts all DD-packets. A DD-packet with the DD-customer-ID 0 is received from all DD-consumers.

A symbolic name can be assigned to each device usable for graphical user interfaces. The remote device can set the symbolic name by the CM-Variable DD_CM_VAR_MODULENAME. On the local device the CM-variable DD_CM_VAR_MODULENAME can be set by function goal_ddCfgModuleName().

GOAL initializes the Device Detection automatically in the state GOAL_FSA_INIT if the Device Detection is enabled by the compiler-define GOAL_DD.

7.1.1 Configuration

7.1.1.1 Compiler-defines

The following compiler-defines are available to configure the Device Detection:

GOAL_DD:
  0: Device Detection is disabled (default)
  1: Device Detection is enabled

7.1.1.2 CM-variables

The following CM-variables are available to configure the Device Detection:

CM-Module-ID	DD_CM_MOD_ID
CM-variable-ID	0
CM-variable name	DD_CM_VAR_MODULENAME
Description	name of the local device, usable by tools for symbolic names This CM-variable can be set by function goal_ddCfgModuleName().
CM data type	GOAL_CM_STRING
Size	20 bytes
Default value	from NVS or 0

CM-Module-ID	DD_CM_MOD_ID
CM-variable-ID	1
CM-variable name	DD_CM_VAR_CUSTOMERID
Description	DD-customer-ID of the local device This CM-variable can be set by function goal_ddCfgCustomerID().
CM data type	GOAL_CM_UINT32
Size	4 bytes
Default value	from NVS or 0

7.1.2 Implementation guide

7.1.2.1 Configure the local device

The local device shall support the Device Detection, therewith the Device Manager tool can set and get CM-variables.

1. GOAL initializes the Device Detection automatically in the state GOAL_FSA_INIT if the Device Detection is enabled by the compiler-define GOAL_DD.

2. Set the DD-customer-ID to 1:
   ```
   goal_ddCfgCustomerID(1);
   ```

3. Set the device name:
   ```
   uint8_t str[] = "myDev";
   goal_ddCfgModuleName(str);
   ```

4. Now the Device Manager tool can read and write CM-variables on the device “myDev”.

7.2 Command line interface (CLI)

GOAL provides a command line interface, which is used by GOAL core modules and other GOAL extension modules. The available commands for the command line interface are described in the appropriate chapters. But it is also possible to integrate a command line interface for the own application, see chapter 0. The source code of the GOAL command line interface is located in the directory …\goal\protos\cli.

The command line interface supports the auto-completion of commands and provides a command history. The size of the command history is configurable during compilation.

The data exchange about the command line interface is realized

- via a UART connection

For further medias please contact *port*.

The command line interface provides an interface for debugging, see chapter 7.2.5.

example:

…\goal\appl\00410_goal\cli*

7.2.1 Configuration

The following compiler-defines are available to configure the command line interface:

GOAL_CONFIG_CLI:
0: command line interface is disabled (default)
1: command line interface is enabled

GOAL_CONFIG_CLI_HISTORY:

0: history of the command line interface is disabled (default)
1: history of the command line interface is enabled

GOAL_CONFIG_CLI_HISTORY_SIZE:
number of history entries

GOAL_CONFIG_CLI_UART:
0: command line interface is not connected via UART (default)
1: command line interface is connected via UART

GOAL_CONFIG_CLI_DBG:
0: debug interface of the command line interface is disabled (default)
1: debug interface of the command line interface is enabled

7.2.2 Platform API

7.2.2.1 UART connection

Prototype	`GOAL_STATUS_T goal_tgtCharGet(char *pBuf)`	
Description	This indication function receives a single char from the UART connection.	
Parameters	pBuf	buffer for a single received char from UART
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL command line interface via UART	
Condition	none	

Prototype	`GOAL_STATUS_T goal_tgtCharPut(char c)`	
Description	This indication function transmits a single char via the UART connection.	
Parameters	c	single char to send via UART
Return values	GOAL return status, see chapter 8.3	
Category	mandatory for GOAL command line interface via UART	
Condition	none	

7.2.3 Command structure

Each CLI command is composed of:

- a main-command,
- one or more sub-commands,
- an action and
- one or more optional parameters.

7.2.3.1 Main-command

port recommends to use “appl” as main-command for application-specific commands to separate these commands from the existing commands in the GOAL system.

7.2.3.2 Sub-command

The sub-command is any specific name.

7.2.3.3 Action

Table 12 provides an overview about binding action names for standard actions. Not all actions must be implemented by a specific command.

Action	Description
add	adding a value to a set of values, e.g. an entry to a table
help	put out a help string for the main-/sub-command
set	set the value of a specified parameter
show	put out the value of a specified parameter
rem	removing a value from a set of values, e.g. remove an entry from a table

Table 12: command line standard actions

7.2.3.4 Parameters

7.2.3.4.1 Integer values

Integer values are currently only accepted with a base of 10 and may optionally contain a sign.
Example: The following command sets the port membership of port 1 to VLAN 1024:

```
$ eth vlan port add 1 1024
```

7.2.3.4.2 Strings

Strings are started and ended with a “-character.
Example: The following command sets the value of config variable 0-1 to value “example”

```
$ cm set 0 1 “example”
```

7.2.3.4.3 Port numbers

Ports are entered as integer values starting with 0 up to max. port number + 1. Max. port number +1 represents the management port. A 5 port switch provides ports 0 – 3 (external ports) and port 4 as management port.

Example: The following commands set the default VLAN tag for port 1 to 1024 with prio 7:

```
$ eth vlan default set 1 1024 7
```

7.2.3.4.4 MAC addresses

MAC addresses are given in the format `xx:xx:xx:xx:xx:xx` where `xx` stands for a two char hex number.
Example: The following command adds port 3 to MAC address 00:11:22:33:44:55

```
$ eth mactab mac add 00:11:22:33:44:55 3
```

7.2.3.4.5 IP addresses

IP addresses are given in the format `xxx.xxx.xxx.xxx` where `xxx` stands for a one- to three-digit decimal number.
Example: The following command sets the IP address, netmask and gateway for the TCP/IP stack:

```
$ net ip set 192.168.1.133 255.255.255.0 0.0.0.0
```

7.2.4 Creating application-specific commands

The following steps are necessary to implement application-specific commands for the command line interface:

1. Create commands, see chapter 7.2.3.
2. Implement the initialization of the application-specific command line interface.
3. Implement command handlers for main- and sub-commands.
4. Register commands to the command line interface by function goal_cliCmdReg() and make the command handlers of the own commands known.
5. The commands are processed loop-controlled in the state GOAL_FSA_OPERATION. It is possible to return a response by function goal_cliPrintf().

Chapter 7.2.6.1 shows an example.

7.2.5 Command line interface for debugging

The following commands are available for the debug interface:

Command	`dbg memb show <address> [count]`	
Description	Shows the byte memory value (8 bit) at the given address. If count is given, up to count values will be shown starting at the given address.	
Parameter	`<address>`	The memory address where the reading begins in hex format (`0xXXXXXXXX`).
	`[count]`	Specifies the number of values to be read starting at the given memory address.

Command	`dbg memw show <address> [count]`	
Description	Shows the word memory value (16 bit) at the given address. If count is given, up to count values will be shown starting at the given address.	
Parameter	`<address>`	The memory address where the reading begins in hex format (`0xXXXXXXXX`).
	`[count]`	Specifies the number of values to be read starting at the given memory address.

Command	`dbg memd show <address> [count]`	
Description	Shows the double word memory value (32 bit) at the given address. If count is given, up to count values will be shown starting at the given address.	
Parameter	`<address>`	The memory address where the reading begins in hex format (`0xXXXXXXXX`).
	`[count]`	Specifies the number of values to be read starting at the given memory address.

7.2.6 Implementation guidelines

7.2.6.1 Create application-specific commands

This example assumes that the command line interface is implemented in an own C module, e.g. appl_cli.c.

1. Digital outputs shall be set via the command line interface. The given value is bit-coded. Each bit relates to a specific DOUT channel. The relevance of the bits in the value can be managed by a bit mask. According to chapter 7.2.3 the command name is:

```
appl dout set <value> <mask>
```

2. Define the string variable in the source code:

```
const char strAppl[] = “appl”;     /* main-command */
const char strApplDout[] = “dout”; /* sub-command */
const char strApplSet[] = “set”;   /* action */
```

3. Implement the initialization function to register the commands and to make the handler applCmdHandler() for all commands known:

```
GOAL_STATUS_T applInitCli(void) {
    GOAL_STATUS_T res;           /* GOAL return value */
    GOAL_CLI_CMD_T *pApplCliHdl; /* handle to main-command */
    GOAL_CLI_CMD_T *pApplCliSubHdl;  /* handle to sub-commands */
```

```
    /* register main-command */
    res = goal_cliCmdReg(strAppl, NULL, applCmdHandler, NULL,
          pApplCliHdl);

    /* register sub-command */
    if (GOAL_RES_OK(res)) {
     res = goal_cliCmdReg(strApplDout, NULL, NULL, pApplCliHdl,
           &pApplCliSubHdl);
    }

    /* register action */
    if (GOAL_RES_OK(res)) {
    res = goal_cliCmdReg(strApplSet, NULL, NULL, pApplCliSubHdl, NULL);
    }

    return res;
}
```

4. Implement the command handler applCmdHandler():

```
void applCmdHandler(
    GOAL_CLI_DATA_T *pData      /* [in] complete received command */
)
{
    GOAL_STATUS_T res;         /* GOAL return value */
    const char *pStr = NULL;   /* string argument from the received
                                  command */
    unsigned int len = 0;      /* length of the argument in byte(s) */
    uint8_t cmdFound = 0;      /* flag to check the existence
                                  of the command */

    /* The main-command is already analyzed by the GOAL command line
    interface and this command handler was called. */

    /* eliminate sub-command from the received command */
    res = goal_cliParamGet(pData, 1, &pStr, &len);
    if (GOAL_RES_OK(res)) {
        /* check sub-command */
        if (strApplDout == pStr) {
            /* eliminate action */
            res = goal_cliParamGet(pData, 2, &pStr, &len);
            if (GOAL_RES_OK(res)) {
                /* check support of action */
                if (strApplSet == pStr) {
                    /* execute application-specific action */
                    applDoutSet();
                    cmdFound = 1;
                }
            }
        }
    }
    /* print a response */
    if ((GOAL_RES_OK(res)) && (1 == cmdFound)) {
        goal_cliPrintf(“command executed\n”);
    }
    else {
        goal_cliPrintf(“unknown command\n”);
    }
    return res;
}
```

5. Implement the application-specific function applDoutSet() to handle the DOUTs.

6. Register the initialization of the application-specific command line interface. The initialization shall be executed in stage GOAL_STAGE_CLI in state GOAL_FSA_INIT_GOAL.

```
GOAL_STAGE_HANDLER_T stageApplCli;
goal_mainStageReg(GOAL_STAGE_CLI, &stageApplCli, GOAL_STAGE_INIT,
   applInitCli);
```

7.3 Web-server

GOAL provides a smart web-server for embedded systems. The web-server was designed:

- for file downloads and
- to get information about the current device state and properties.

The web-server supports the following properties:

transfer protocols:	• HTTP • HTTPS
request methods:	• GET • POST

One or more web-pages can be assigned to one instance of the web-server. The web-pages are part of the application and must be made available by the application. The web-server provides a callback function for this purpose, see cbHttpReqFunc() in chapter 7.3.4.

The current device state and properties can be read from CM-variables and application-specific variables. The application-specific variables can be organized as simple variables or as a one-dimensional list.
It is possible to store templates for web-pages with placeholders for current values of application-specific variables. The text substitutions are described in chapter 7.3.2. Web-templates make the dynamic management of web-pages possible.

The access to the web-server can be limited by user levels. The application can specify, which user levels shall be supported by the device and which rights the user levels shall have. The authentication data consisting of user name and the password for each user level are configurable by CM-variables. The GOAL web-server provides up to 4 user levels.
The user levels can be applied by all instances of the web-server. For each instance of the web-server the valid user level can be specified during registration. Web-requests are only transferred to the application after a successful authentication, i.e. the callback function cbHttpReqFunc() in chapter 7.3.4 is only called after a successful login. The transfer of the user name and the password via a web-server instance using the HTTP transfer protocol is unsafe. *port* recommends using the HTTPS transfer protocol.

HTTPS is activated by the compiler-define GOAL_CONFIG_HTTPS. HTTPS uses the external software component mbedTLS for encoding und authentication. The access to mbedTLS is realized about the media adapter for TLS, see chapter 6.4. TLS for HTTPS is initialized and opened by function goal_httpsNew().
About the CM-variables for HTTPS it is possible to install a private key and an own X509-certificate for. If no own certificate is stored, the web-server takes a default certificate provided by *port*.

example:

- …\goal\appl\goal_http\01_get*:
  for upload of a web-page
- …\goal\appl\goal_http\05_template_cm*:
  for upload of a web-template with CM-variables and application-specific variables
- …\goal\appl\goal_http\06_template_list*:
  for upload of a web-template with lists
- …\goal\appl\goal_http\04_auth*:
  for authentication about user levels
- …\goal\appl\goal_http\02_post*:
  for file download

7.3.1 Configuration

7.3.1.1 Compiler-defines

The following compiler-defines are available to configure the webserver:

GOAL_CONFIG_HTTP:
0: transfer protocol HTTP is not used (default)
1: transfer protocol HTTP is used

GOAL_CONFIG_HTTPS:
0: transfer protocol HTTPS is not used (default)
1: transfer protocol HTTPS is used

7.3.1.2 CM-variables

The following CM-variables are available to configure the web-server:

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	0
CM-variable name	HTTPD_CM_VAR_HTTPD_CHANNELS_MAX
Description	maximal number of connections available for the HTTP transfer protocol
CM data type	GOAL_CM_UINT16

Size	2 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	1
CM-variable name	HTTPD_CM_VAR_HTTPS_CHANNELS_MAX
Description	maximal number of connections available for the HTTPS transfer protocol
CM data type	GOAL_CM_UINT16
Size	2 bytes
Default value	from NVS or 0

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	2
CM-variable name	HTTPD_CM_VAR_USERLEVEL0
Description	authentication data for level 0
CM data type	GOAL_CM_STRING
Size	32 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	3
CM-variable name	HTTPD_CM_VAR_USERLEVEL1
Description	authentication data for level 1
CM data type	GOAL_CM_STRING
Size	32 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	4
CM-variable name	HTTPD_CM_VAR_USERLEVEL2
Description	authentication data for level 2
CM data type	GOAL_CM_STRING
Size	32 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPD
CM-variable-ID	5
CM-variable name	HTTPD_CM_VAR_USERLEVEL3

Description	authentication data for level 3
CM data type	GOAL_CM_STRING
Size	32 bytes
Default value	from NVS or an empty string

The following CM-variables allow to configure the TLS layer used by HTTPS:

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	0
CM-variable name	HTTPS_CM_VAR_TLS_SERVER_CERTIFICATE
Description	certificate of the web-server
CM data type	GOAL_CM_GENERIC
Size	1024 bytes
Default value	from NVS or certificate from *port*

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	1
CM-variable name	HTTPS_CM_VAR_TLS_PRIVATE_KEY
Description	private key of the web-server
CM data type	GOAL_CM_GENERIC
Size	1024 bytes
Default value	from NVS or an empty entry

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	2
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_CA_CN
Description	common name of the server of the certification authority
CM data type	GOAL_CM_STRING
Size	128 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	3
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_CA_O
Description	name of the certification authority organization, e.g. the company name
CM data type	GOAL_CM_STRING
Size	128 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	4
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_CA_C
Description	country, in which the certification authority organization is located
CM data type	GOAL_CM_STRING
Size	8 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	5
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_CN
Description	common name of the web-server
CM data type	GOAL_CM_STRING
Size	128 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	6
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_O
Description	name of the organization provided the web-server
CM data type	GOAL_CM_STRING
Size	128 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	7
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_C
Description	country, in which the organization provided the web-server is located
CM data type	GOAL_CM_STRING
Size	8 bytes
Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	8
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_NOT_BEFORE
Description	from what date and time the certificate is valid
CM data type	GOAL_CM_STRING
Size	20 bytes

Default value	from NVS or an empty string

CM-Module-ID	GOAL_ID_HTTPS
CM-variable-ID	9
CM-variable name	HTTPS_CM_VAR_TLS_SRV_CERT_NOT_AFTER
Description	from what date and time the certificate is invalid
CM data type	GOAL_CM_STRING
Size	20 bytes
Default value	from NVS or an empty string

7.3.2 Web-templates

The GOAL web-server allows to implement templates for web-pages with placeholders for current information. The placeholders are substituted by the current values by the web-server during the upload process. The web-server provides placeholders for

- CM-variables,
- application-specific variables and
- lists.

7.3.2.1 CM-variables

The placeholder for CM-variables contains the CM-module-ID and the CM-variable-ID. The web-server executes the substitution of the placeholder by the CM-variable automatically.
syntax:

```
[CM:<modNum>, <cmVarNum>]
```

example:

```
[CM:0, 2]
```

7.3.2.2 Application-specific variables

The placeholder for application-specific variables contains the name of the variable in the application. The web-server requires the current value of the variable from the application by calling a callback function, see chapter 7.3.4 cpHttpTemplateFunc(), and substitutes the placeholder in the web-page.

syntax:

```
[VAR:<applVarName>]
```

example:

[VAR:applVar]

7.3.2.3 Lists

The web-server provides an effective method to generate lists in HTML text. The HTML text for a single list entry can be enclosed in the placeholders FOREACH and /FOREACH in the web-template. FOREACH marks a one-dimensional list and the HTML text between the placeholders FOREACH and /FOREACH is execute for each list element. The place for the list entry is marked in the HTML text by the placeholder VAR with the desired variable name. After the substitution of the placeholder VAR the web-server changes to the next list entry automatically, i.e. it is not possible to substitute the same list entry twice. Therewith it is only necessary to describe the first list entry in the web-template.

The web-server only gets the ID and the name of the list and the number of list elements during the registration. The content of the list elements is managed by the application. The web-server calls a callback function to get the content of the next list element, see chapter 7.3.4 cpHttpTemplateFunc().

Nested lists are allowed. The maximal supported nesting depth is 4.

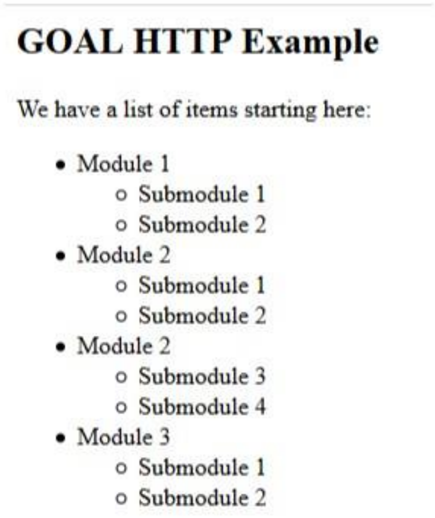
syntax:

```
[FOREACH:<listName>] … [/FOREACH]
```

example for HTML listing:

```
<ul>
    [FOREACH:mainList]
    <li> main: [VAR:mainName] </li>
    <li> sub-lists:
             <ul>
                  [FOREACH]
                  <li> sub: [VAR:subName] </li>
                  [/FOREACH]
             </ul>
    </li>
    [/FOREACH]
</ul>
```

Example …\goal\appl\goal_http\06_template_list* generates a HTML listing. The indication in the web-browser is shown in Figure 23:



GOAL HTTP Example

We have a list of items starting here:

- Module 1
    - Submodule 1
    - Submodule 2
- Module 2
    - Submodule 1
    - Submodule 2
- Module 2
    - Submodule 3
    - Submodule 4
- Module 3
    - Submodule 1
    - Submodule 2

Figure 23: web-page of example 06_template_list

The placeholder [FOREACH] … [\FOREACH] can also be integrated in other HTML formatting like tables.

7.3.3 Characters

square brackets: If the HTML text shall show square brackets, the square brackets must be written double, because placeholders in web-templates are bordered by square brackets.
example: An array instruction shall be shown on a web-page.
HTML text: applArray[[5]]
Web-browser view: applArray[5]

double quotes: Double quotes in the HTML-text must be protected by a backslash, because strings in the C code are enclosed by double quotes.
example: uint8_t webPage[] = “<html><meta charset = \”utf-8\”> … </html>”;

The rules for HTML text are valid for **all other characters**.

7.3.4 Callback functions

Prototype	`GOAL_STATUS_T cbHttpReqFunc(GOAL_HTTP_APPLCB_DATA_T *applData)`
Description	The received and valid web-request is passed to the application. The application has to process the web-request and to produce a web-response.

Parameters	applData	contains data for the application and data returned by the application
Return values	GOAL return status, see chapter 8.3	
Category	optional	
Registration	during runtime about function goal_httpdResReg()	

Prototype	`GOAL_STATUS_T cbHttpTemplateFunc(GOAL_HTTP_APPLCB_TEMPL_T *pWebTemplate)`	
Description	About this callback function the application provides the current information for the specified placeholder. This callback function is called for each placeholder in the web-template. There are multiple placeholders for the same information, this callback function is called for each placeholder.	
Parameters	pWebTemplate	Contains the information to specify a variable or list and the current return value of the specified variable.
Return values	GOAL return status, see chapter 8.3	
Category	optional	
Registration	during runtime about function	

7.3.5 Implementation guideline

7.3.5.1 Upload a web-page

The web-page “device.html” shall be uploaded from the device to the web-browser. The content of the web-page is stored in variable webPage[] as string. No text substitutions on the web-page are necessary.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 8080 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;     /* handle for the web-server instance */
goal_httpNew(&pWebInstanceHdl, 8080, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP: No callback function for text substitution is registered.

```
GOAL_HTTP_HDL_T webResourceHdl;
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/device.html“,
GOAL_HTTP_METHOD_ALLW_GET, applWebReqCb, NULL, &webResourceHdl);
```

4. Provide the web-page as string variable:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
<title> Device </title></head> \r\n\
<body><h1>Device</h1> \r\n\
<p>The device implementation bases on the GOAL middleware of port.
</p> \r\n\
</body></html>”
```

5. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */
    res = GOAL_ERROR;  /* no valid web-handle or
                        *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
        if (GOAL_HTTP_FW_GET == pWebData->reqType) {
            GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                GOAL_STRLEN((const char*) webPage));
            res = GOAL_OK;
        }
    }
    return res;
}
```

6. The callback function applWebReqCb() is called by the web-server after the receipt of the web-request.

7.3.5.2 Read a CM-variable

The web-page “device.html” shall be uploaded from the device to the web-browser. The web-page bases on the template webPage[]. The template includes a placeholder for the CM-Variable DD_CM_VAR_MODULENAME with the CM-module-ID 34 and the CM-variable-ID 0. The web-server substitutes the placeholder by the current value of the CM-variable automatically. No text substitutions on the web-page are necessary.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 8080 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;     /* handle for the web-server instance */
goal_httpNew(&pWebInstanceHdl, 8080, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP: No callback function for text substitution is registered.

```
GOAL_HTTP_HDL_T webResourceHdl;
```

```
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/device.html“,
GOAL_HTTP_METHOD_ALLW_GET, applWebReqCb, NULL, &webResourceHdl);
```

4. Provide a template for the web-page as string variable with placeholder for the CM-variable:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
<title> Device </title></head> \r\n\
<body><h1>Device</h1> \r\n\
<p>The device with the name [CM:34, 0] bases on the GOAL middleware of port.</p>
\r\n\
</body></html>”
```

5. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res;  /* GOAL return value */
    res = GOAL_ERROR;   /* no valid web-handle or
                         *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
        if (GOAL_HTTP_FW_GET == pWebData->reqType) {
            GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                GOAL_STRLEN((const char*) webPage));
            res = GOAL_OK;
        }
    }
    return res;
}
```

6. The callback function applWebReqCb() is called by the web-server after the receipt of the web-request.

7.3.5.3 Read application-specific variable

The web-page “device.html” shall be uploaded from the device to the web-browser. The web-page bases on the template webPage[]. The template includes a placeholder for the application-specific Variable applVar. The web-server calls the callback function applWebGetValCb() to provide the current value of the application-specific variable for the web-server. The web-server substitutes the placeholder by the current value of the application-specific variable.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 8080 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;     /* handle for the web-server instance */
goal_httpNew(&pWebInstanceHdl, 8080, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_HDL_T webResourceHdl;
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/device.html“,
GOAL_HTTP_METHOD_ALLW_GET, applWebReqCb, applWebGetValCb, &webResourceHdl);
```

4. Provide a template for the web-page as string variable with placeholder for the application-specific variable:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
<title> Device </title></head> \r\n\
<body><h1>Device</h1> \r\n\
<p>The device with the name [VAR:applVar] bases on the GOAL middleware of port.</p>
\r\n\
</body></html>”
```

5. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */
    res = GOAL_ERROR;  /* no valid web-handle or
                       *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
        if (GOAL_HTTP_FW_GET == pWebData->reqType) {
            GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                GOAL_STRLEN((const char*) webPage));
            res = GOAL_OK;
        }
    }
    return res;
}
```

6. Implement a callback function to get the current value of the application-specific variable from the application:

```
uint8_t deviceName[] = “Sample Gadget”;
GOAL_STATUS_T applWebGetValCb (GOAL_HTTP_APPLCB_TEMPL_T *pWebData) {
    if (0 == GOAL_MEMCMP(pWebData->in.name, “applVar”,
    GOAL_STRLEN(“applVar”))) {

        /* provide the complete device name */
        if (GOAL_STRLEN(deviceName) <= pWebData->in.retLenMax) {
            GOAL_MEMCPY(pWebData->out.strReturn, deviceName,
                GOAL_STRLEN(deviceName));
        }
        /* the device name is too long, cut the device name down
           * to the maximal allowed length */
        else {
            GOAL_MEMCPY(pWebData->out.strReturn, deviceName,
                pWebData->in.retLenMax);
        }
    }
    return GOAL_OK;
}
```

7. The callback function applWebReqCb() is called by the web-server after the receipt of the web-request. The desired template for the web-page is made available for the web-server.

8. The callback function applWebGetValCb() is called for substitution.

9.

7.3.5.4 Read a list

The web-page “device.html” shall be uploaded from the device to the web-browser. The web-page bases on the template webPage[]. The template includes a placeholder for the list of device components. The web-server calls the callback function applWebGetValCb() to provide the current value of the list entries. The web-server substitutes the placeholder by the current value of the application-specific variable.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 8080 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;     /* handle for the web-server instance */
goal_httpNew(&pWebInstanceHdl, 8080, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_HDL_T webResourceHdl;
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/device.html“,
GOAL_HTTP_METHOD_ALLW_GET, applWebReqCb, applWebGetValCb, &webResourceHdl);
```

4. Create the list information and register the list information for the web-server. The list information contains a list-ID, a list name and the number of list entries.

```
GOAL_HTTP_TEMPLATE_LIST_INIT_T webList;
GOAL_MEMSET(&webList, 0, sizeof(webList));
webList.listId = 1;
webList.cntMemb = 4;
GOAL_MEMCPY(webList.listName, “deviceComponents”,
    sizeof(“devcieComponents”));
goal_httpTmpMgrNewList(pWebInstanceHdl, &webList);
```

5. Provide a template for the web-page as string variable with placeholders for the list and list entries:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
<title> Device </title></head> \r\n\
<body><h1>Device</h1> \r\n\
<p> The device contains the following components: \r\n\
    <ul> \r\n\
        [FOREACH:deviceComponents] \r\n\
        <li> [VAR:devComponent] </li> \r\n\
        [/FOREACH]
    </ul> \r\n\
</p> \r\n\
```

```
</body></html>”
```

6. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */
    res = GOAL_ERROR;  /* no valid web-handle or
                       *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
        if (GOAL_HTTP_FW_GET == pWebData->reqType) {
            GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                GOAL_STRLEN((const char*) webPage));
            res = GOAL_OK;
        }
    }
    return res;
}
```

7. Implement a callback function to get the current value of the list entries from the application:

```
GOAL_STATUS_T applWebGetValCb (GOAL_HTTP_APPLCB_TEMPL_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */
    res = GOAL_ERR_NOT_FOUND;
    if (NULL != pWebData->in.pPath) {
        if (0 == GOAL_MEMCMP(pWebData->in.name, “deviceComponents”,
        GOAL_STRLEN(“deviceComponents”))) {

            switch ((pWebData->inPath)->path[0].index) {
                case 0:
                    GOAL_MEMCPY(pWebData->out.strReturn, “I/O module”,
                        GOAL_STRLEN(“I/O module”));
                    res = GOAL_OK;
                    break;
                case 1:
                    GOAL_MEMCPY(pWebData->out.strReturn, “drive”,
                        GOAL_STRLEN(“drive”));
                    res = GOAL_OK;
                    break;
                case 2:
                    GOAL_MEMCPY(pWebData->out.strReturn, “encoder”,
                        GOAL_STRLEN(“encoder”));
                    res = GOAL_OK;
                    break;
                case 3:
                    GOAL_MEMCPY(pWebData->out.strReturn, “power supply”,
                        GOAL_STRLEN(“power supply”));
                    res = GOAL_OK;
                    break;
                default:
                    break;
            }
        }
    }
    return res;
}
```

8. The callback function applWebReqCb() is called by the web-server after the receipt of the web-request. The desired template for the web-page is made available for the web-server.

9. The callback function applWebGetValCb() is called for each substitution.

7.3.5.5 Set a user level

The upload of the web-page “admin.html” shall only allowed for users with the USERLEVEL0. The login data for the USERLEVEL0 are:

- user name: admin
- password: a1b2c3:UL

The HTTPS transfer protocol is used.
The web-page “admin.html” does not contain any placeholder. No text substitutions on the web-page are necessary.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 443 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;    /* handle for the web-server instance */
goal_httpsNew(&pWebInstanceHdl, 443, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_HDL_T webResourceHdl;
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/admin.html“,
GOAL_HTTP_METHOD_ALLW_GET | GOAL_HTTP_AUTH_USERLEVEL0, applWebReqCb, NULL,
&webResourceHdl);
```

4. Install the authentication for USERLEVEL0 in state GOAL_FSA_INIT_SETUP: The authentication data are written to the CM-variable USERLEVEL0.

```
goal_httpAuthBasSetUserInfo(pWebInstanceHdl, GOAL_HTTP_AUTH_USERLEVEL0,
    “admin”, “a1b2c3:UL”);
```

5. Provide a template for the web-page as string variable:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
                           <title> Administration </title></head> \r\n\
                           <body><h1>Administration</h1> \r\n\
<p> Internal information: … </p> \r\n\
</body></html>”
```

6. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */
    res = GOAL_ERROR;  /* no valid web-handle or
                       *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
```

```
        if (GOAL_HTTP_FW_GET == pWebData->reqType) {
            GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                GOAL_STRLEN((const char*) webPage));
            res = GOAL_OK;
        }
    }
    return res;
}
```

7. The callback function applWebReqCb() is called by the web-server after the receipt web-request. The login for this web-server instance must be successful before.

7.3.5.6 Download files

The web-server provides a download dialog. The received file is transferred to the application.

1. Initialize the web-server in state GOAL_FSA_INIT_APPL or GOAL_FSA_INIT_GOAL:

```
goal_httpInit();
```

2. Open 1 instance of the web-server using the TCP port 8080 in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_T *pWebInstanceHdl;    /* handle for the web-server instance */
goal_httpNew(&pWebInstanceHdl, 8080, 1);
```

3. Register the callback-functions and allowed methods for the opened web-server in state GOAL_FSA_INIT_SETUP:

```
GOAL_HTTP_HDL_T webResourceHdl;
goal_httpResReg(pWebInstanceHdl, (uint8_t *) „/download.html“,
                GOAL_HTTP_METHOD_ALLW_GET | GOAL_HTTP_METHOD_ALLW_POST,
                applWebReqCb, NULL, &webResourceHdl);
```

4. Provide a web-page as string variable:

```
const uint8_t webPage[] = “<html><head><meta charset = \”utf-8\”>\
                           <title> Download dialog </title></head> \r\n\
                           <body><h1>Download dialog</h1> \r\n\
<form method = \”post\” enctype = \”multipart/form-data\”> \r\n\
    <input type = \”file\” name = \”file\”> <br> \r\n\
    <input type = \”submit\” value = \”POST\”> \r\n
</form> \r\n\
</body></html>”
```

5. Implement an application-specific function applDownload() to receive and install the new firmware.

6. Implement an application-specific function applDownloadFinished() to report the result of the web-request to the application.

7. Implement a callback function to process web-requests:

```
GOAL_STATUS_T applWebReqCb (GOAL_HTTP_APPLCB_DATA_T *pWebData) {
    GOAL_STATUS_T res; /* GOAL return value */

    res = GOAL_ERROR;  /* no valid web-handle or
                        *  request method not supported */

    if (webResourceHdl == pWebData->hdlRes) {
        switch (pWebData->reqType) {
            case GOAL_HTTP_FW_GET:
                GOAL_HTTP_GET_RETURN_HTML(pWebData, webPage,
                    GOAL_STRLEN((const char*) webpage));
                break;
            case GOAL_HTTP_FW_POST_START:
                res = applDownload(pWebData);
                GOAL_HTTP_RETURN_OK_204(pWebData);
                break;
            case GOAL_HTTP_FW_POST_DATA:
                res = applDownload(pWebData);
                GOAL_HTTP_RETURN_OK_204(pWebData);
                break;
            case GOAL_HTTP_FW_POST_END:
                res = applDownload(pWebData);
                GOAL_HTTP_RETURN_OK_204(pWebData);
                break;
            case GOAL_HTTP_FW_REQ_DONE_OK:
            case GOAL_HTTP_FW_REQ_DONE_ERR:
                res = applDownloadFinished(pWebData);
                break;
            default:
                break;
        }
    }
    return res;
}
```

8. The callback function applWebReqCb() is called by the web-server after the receipt web-request.

7.4 Firewall

GOAL provides a basic firewall for ARP, TCP and UDP. After calling goal_fwInit() the firewall is active and working. All frames received in goal_ethRec() matching ether type ARP and IPv4 will be passed to and processed by the firewall.

7.4.1 ARP-Firewall

All received ethernet frames of ether type ARP will be filtered by goal_fwArp(). The firewall will process frames that match the following conditions:

- EtherType = 0x0806 (Address Resolution Protocol)
- Data length must be equal or higher than 96 bytes
- ARP hardware type is ethernet - 0x0001
- Opcode is request – 0x01
- Protocol type is IPv4 – 0x0800

Frames that doesn’t match these conditions will be passed to the queue. The ARP-firewall drops all frames that are not addressed to the device or have MAC-Address NULL. The firewall reads the IP, Gateway and Netmask settings at initialization and gets updated automatically at any change of these values.

7.4.2 IPv4-Firewall

All received ethernet frames of ether type IPv4 will be filtered by goal_fwIpv4(). All frames of type ICMP will be passed directly to the queue. For TCP and UDP the firewall:

- Checks frame length and drops malformed frames
- Iterates through all channels to check for matching port number on used channels
- Drops frames with no matching port number

The firewall reads the maximum number of net-channels and the location of the cannel list at initialization.

8 Implementation specifics

8.1 Naming rules

The prefix “goal_” is used by the GOAL system for:

- global variables
- GOAL function names

The prefix “GOAL_CONFIG_” is used by the GOAL system for:

- compiler-defines to configure the GOAL system by the user

The prefix “GOAL_TARGET_” is used by the GOAL system for:

- compiler-defines to configure hardware-dependent settings

The prefix “GOAL_” is used by the GOAL system for:

- compiler-defines for configuration
- compiler-defines for GOAL status values
- compiler-defines for specific constant values
- GOAL data types
- macros used by the GOAL system

8.2 GOAL data types

The GOAL system uses different classes of data types:

- GOAL basic data types: used in the source code
- CM-variable data types: used for CM-variables
- LM-parameter data types: used for parameters in a log message

GOAL basic data type	CM-variable data type	LM-parameter data type	Description
int8_t	GOAL_CM_VAR_INT8	INT8	signed integer, 8 bit
int16_t	GOAL_CM_VAR_INT16	INT16	signed integer, 16 bit
int32_t	GOAL_CM_VAR_INT32	INT32	signed integer, 32 bit
int64_t	--	--	signed integer, 64 bit
uint8_t	GOAL_CM_VAR_UINT8	UINT8	unsigned integer, 8 bit
uint16_t	GOAL_CM_VAR_UINT16	UINT16	unsigned integer, 16 bit
uint32_t	GOAL_CM_VAR_UINT32	UINT32	unsigned integer, 32 bit
uint64_t	--	--	unsigned integer, 64 bit
GOAL_BOOL_T	--	--	boolean: 0 = GOAL_FALSE, 1 = GOAL_TRUE

GOAL basic data type	CM-variable data type	LM-parameter data type	Description
PtrCast	--	--	pointer casting helper
--	STRING	--	char array, not zero terminated
--	--	STRING0	char array, zero terminated
--	IPV4	IPV4	IP-addresses vvv.xxx.yyy.zzz as uint32_t value: ((vvv << 24) | (xxx << 16) | (yyy << 8) | zzz)
--	--	MAC	MAC-address ss-tt-uu-xx-yy-zz as byte array value
--	GENERIC	GENERIC	byte array

Please do not rely on generic data types like char or int because the behavior and the width is compiler-specific and platform-specific and use the GOAL basic data types in the source code!

8.3 GOAL status

GOAL defines the data type GOAL_STATUS_T most used as function return value. The values of GOAL status are defined in …\goal\goal\goal_types.h. GOAL provides the following macros for the symbolic evaluation:

- GOAL_RES_OK():
  0: The GOAL status reports an error.
  1: The GOAL status reports success.
- GOAL_RES_ERR():
  0: The GOAL status reports success.
  1: The GOAL status reports an error.

8.4 Alignment

The alignment can be specified:

- for the processor about the compiler-define GOAL_TARGET_MEM_ALIGN_CPU or
- for network transfers about the compiler-define GOAL_TARGET_MEM_ALIGN_NET

Only the values 2, 4 and 8 are supported.

8.5 Heap memory size

The size of the heap memory must be specified about the compiler-define GOAL_CONFIG_HEAP_SIZE.

The heap memory size is determined by the GOAL system according to the following order:

1. The heap memory size can be specified application-specific. The compiler-define GOAL_CONFIG_HEAP_SIZE must be set in …\goal\appl\...\goal_config.h or in the compiler-project to guarantee that the definition is taken first.
2. The heap memory size is specified platform-specific in …\goal\plat\arch\...\goal_target.h. This value is valid if there is no application-specific setting.
3. The heap memory size is specified module-specific in …\goal\goal*.h if necessary. This value is valid if there is no application-specific and no platform-specific setting.

9 Additional platform-specific indication functions

Each platform provides the following further platform-specific indication functions:

Prototype	`GOAL_TIMESTAMP_T goal_targetGetTimestamp(void)`
Description	This indication function returns the time as 64-bit value since starting of the GOAL system.
Parameters	none
Return values	past time in ms
Category	mandatory

Prototype	`uint32_t goal_targetGetTicks(void)`
Description	This indication function returns the time as 32-bit value since starting of the GOAL system.
Parameters	none
Return values	past time in ms
Category	mandatory

Prototype	`uint32_t goal_targetGetTimestampDiffUs(void)`
Description	This indication function returns the time difference between two calls of this function in µs. This function shall be embedded in the compiler-define GOAL_CONFIG_DEBUG_SOFTSCOPE.
Parameters	none
Return values	time difference in µs
Category	optional

10 Version information

The version number of GOAL is documented in goal\goal\goal_includes.h.

11 Glossary

AC	Application Controller
ARP	Address Resolution Protocol
BM	Bit-Mapping
CC	Communication Controller
CLI	Command Line Interface
CM	Configuration Manager
DD	Device Detection
DHCP	Dynamic Host Configuration Protocol
FSA	Finite State Automaton
GOAL	Generic Open Abstraction Layer
GOAL components	Elements of the GOAL system
GOAL system	Embedded system using the GOAL middleware
IP	Internet Protocol
MA	Media Adapter
MAC address	Media Access Control address
MI	Media Interface
NVM	Non-Volatile Memory
NVS	Non-Volatile Storage
RPC	Remote Procedure Call
SQE	Signal Quality Error
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TLS	Transport Layer Security
TLV	Type Length Value
TOS	Type-Of-Service
TTL	Time-To-Live
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network

12 References

/Fletcher/	http://en.wikipedia.org/wiki/Fletcher%27s_checksum#Optimizations
/RFC_1213/	Management Information Base for Network Management of TCP/IP-based internets: MIB-II, March 1991
/TLS_RFC_5246/	The Transport Layer Security (TLS) Protocol, Version 1.2

13 Index