

OS Architecture Design and User Manual

**GOS2022**

OS Architecture Design and User Manual

for GOS2022 version 1.0

**Description of OS architecture**

**API description and OS configuration options**

© Ahmed Gazar, 2024

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version history | | | | |
| Version | Date | Author | Change | Released |
| 1.0 | 2024-05-22 | Ahmed Gazar | Initial version of document. | - |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# **Content**

[Content 5](#_Toc167298596)

[Chapter 1 OS introduction 7](#_Toc167298597)

[1. Introduction 8](#_Toc167298598)

[2. Hardware Support 8](#_Toc167298599)

[3. OS Layered Structure 8](#_Toc167298600)

[4. Folder Structure 8](#_Toc167298601)

[5. OS Service Dependencies 11](#_Toc167298602)

[Chapter 2 Kernel 12](#_Toc167298603)

[6. Kernel 13](#_Toc167298604)

[6.1. Kernel Definitions 13](#_Toc167298605)

[6.2. Startup 13](#_Toc167298606)

[6.2.1. Kernel Initialization 13](#_Toc167298607)

[6.2.2. Kernel Start 13](#_Toc167298608)

[6.3. Scheduling 13](#_Toc167298609)

[6.3.1. Pre-emptive Scheduling 13](#_Toc167298610)

[6.3.2. Cooperative Scheduling 14](#_Toc167298611)

[6.3.3. Next Task Selection 14](#_Toc167298612)

[6.4. Stack Monitoring 14](#_Toc167298613)

[6.5. CPU-load Monitoring 14](#_Toc167298614)

[6.6. CPU-load Limits 15](#_Toc167298615)

[Chapter 3 Tasks 16](#_Toc167298616)

[7. Tasks 17](#_Toc167298617)

[7.1. Introduction 17](#_Toc167298618)

[7.2. Idle Task 17](#_Toc167298619)

[7.3. System Task 17](#_Toc167298620)

[7.4. System Daemons 17](#_Toc167298621)

[7.4.1. Message Daemon 17](#_Toc167298622)

[7.4.2. Shell Daemon 17](#_Toc167298623)

[7.4.3. Signal Daemon 17](#_Toc167298624)

[7.4.4. Sysmon Daemon 17](#_Toc167298625)

[7.4.5. Time Daemon 17](#_Toc167298626)

[7.4.6. Trace Daemon 17](#_Toc167298627)

[7.5. Task State-Machine 18](#_Toc167298628)

[7.6. Task Priority Levels 19](#_Toc167298629)

[7.7. Privileges 20](#_Toc167298630)

[7.7.1. Kernel Privileges 20](#_Toc167298631)

[7.7.2. User-defined Privileges 20](#_Toc167298632)

[7.8. Creating and Registering Tasks 21](#_Toc167298633)

[7.8.1. Mandatory Task Parameters 21](#_Toc167298634)

[7.8.2. Optional Task Parameters 21](#_Toc167298635)

[7.9. Single-shot, Cyclic, and Asynchronous Tasks 21](#_Toc167298636)

[Chapter 4 OS Drivers 23](#_Toc167298637)

[8. OS Drivers 24](#_Toc167298638)

[8.1. Introduction 24](#_Toc167298639)

[8.2. Driver Skeletons 24](#_Toc167298640)

[Table Of Figures 25](#_Toc167298641)

# **Chapter 1** OS introduction

## Introduction

GOS2022 is an open-source embedded operating system project. This document describes the architecture and main features of the OS to help users better understand the structure of it, give an overall view, and help them understand how it can be used to develop applications with it.

## Hardware Support

Currently, GOS2022 is being developed and tested on an STM32F446RE board (with and ARM Cortex-M4 core), therefore it is the only supported platform. Portability and support for other platforms is in the scope of future development.

## OS Layered Structure

The OS is organized into a basic layered structure. At the heart of this structure is the kernel. The kernel contains basic type, enum, and structure definitions that are used throughout the entire system, as well as the scheduling logic, stack monitoring, CPU-load monitoring, and task handling.

Driver skeletons is a separate layer containing prototype functions (to be implemented by user as they are platform-dependent) that are used by OS services.

Service layer contains the provided OS services such as error handling, GCP (communication protocol implementation), messaging (inter-task data exchange), mutex (shared-resource protection), queue (FIFO-type data queue), shell (basic command interpreter), signal (light inter-task communication), sysmon (complex system monitoring service with communication interface), time (time definitions and system-time handling), trace (formatted tracing service), trigger (lightweight task synchronization). Detailed descriptions of the above mentioned services can be found in later chapters.

A diagram of a software application

Description automatically generated

Figure 1: Layers of OS

## Folder Structure

The OS source code is organized in the following structure:

OS root

* driver
  + inc
    - gos\_crc\_driver.h

*CRC driver skeleton public interface.*

* + - gos\_driver.h

*Driver skeleton layer public interface.*

* + - gos\_shell\_driver.h

*Shell driver skeleton public interface.*

* + - gos\_sysmon\_driver.h

*System monitoring driver skeleton public interface.*

* + - gos\_timer\_driver.h

*Timer driver skeleton public interface.*

* + - gos\_trace\_driver.h

*Trace driver skeleton public interface.*

* + src
    - gos\_crc\_driver.c

*CRC driver skeleton implementation.*

* + - gos\_driver.c

*Driver skeleton layer implementation.*

* + - gos\_shell\_driver.c

*Shell driver skeleton implementation.*

* + - gos\_sysmon\_driver.c

*System monitoring driver skeleton implementation.*

* + - gos\_timer\_driver.c

*Timer driver skeleton implementation.*

* + - gos\_trace\_driver.c

*Trace driver skeleton implementation.*

* kernel
  + inc
    - gos\_config.h

*OS configuration file containing configuration macros.*

* + - gos\_kernel.h

*Kernel public interface containing system-wide macros, type definitions.*

* + - gos\_port.h

*Platform-dependent code macros.*

* + Src
    - gos\_kernel.c

*Kernel implementation.*

* + - gos\_task.c

*Task-related function implementations.*

* services
  + inc
    - gos.h

*OS main header.*

* + - gos\_error.h

*Error service public interface.*

* + - gos\_gcp.h

*GCP public interface.*

* + - gos\_message.h

*Message service public interface.*

* + - gos\_mutex.h

*Mutex service public interface.*

* + - gos\_queue.h

*Queue service public interface.*

* + - gos\_shell.h

*Shell service public interface.*

* + - gos\_signal.h

*Signal service public interface.*

* + - gos\_sysmon.h

*Sysmon service public interface.*

* + - gos\_time.h

*Time service public interface.*

* + - gos\_trace.h

*Trace service public interface.*

* + - gos\_trigger.h

*Trigger service public interface.*

* + Src
    - gos.c

*OS main implementation.*

* + - gos\_error.c

*Error service implementation.*

* + - gos\_gcp.c

*GCP implementation.*

* + - gos\_message.c

*Message service implementation.*

* + - gos\_mutex.c

*Mutex service implementation.*

* + - gos\_queue.c

*Queue service implementation.*

* + - gos\_shell.c

*Shell service implementation.*

* + - gos\_signal.c

*Signal service implementation.*

* + - gos\_sysmon.c

*System monitoring service implementation.*

* + - gos\_time.c

*Time service implementation.*

* + - gos\_trace.c

*Trace service implementation.*

* + - gos\_trigger.c

*Trigger service implementation.*

## OS Service Dependencies

A screenshot of a computer

Description automatically generated

Figure 2: OS Service Dependencies

# **Chapter 2** Kernel

## Kernel

## Kernel Definitions

The basic OS definitions (kernel definitions) can be found in **gos\_kernel.h**. These definitions contain:

* Basic data types (e.g. u8\_t, s16\_t, bool\_t, void\_t, etc.)
* Task-related definitions (e.g. gos\_tid\_t, gos\_taskPrio\_t, etc.)
* Task state enumerators
* Task privilege macros and enumerators
* Hook function type definitions
* Result type enumerators
* Boolean values
* Time unit definitions
* Runtime type definition
* Task descriptor definition
* Other macros (e.g., interrupt enable/disable, max. priority levels, etc.)

## Startup

Kernel startup consists of the kernel initialization and the kernel start (which basically means the starting of the scheduler). These two events can be triggered separately.

## Kernel Initialization

In this phase, the internal task descriptor array is initialized, and the idle task stack frame is filled, the overflow threshold is calculated. Finally, the fault handlers are enabled.

## Kernel Start

In this phase, the low-level initialization of the kernel is carried out, and the idle task is selected as the first task to run. The system timer value is updated through a timer driver skeleton function call (see: Driver skeletons), the scheduling is enabled, and the execution of the first task is initiated.

## Scheduling

GOS2022 supports two scheduling strategies: pre-emptive and cooperative. This can be selected in **gos\_config.h**. The default setting is pre-emptive, and cooperative can be turned on (1) or off (0).

/\*\*

\* Cooperative scheduling flag.

\*/

**#define** CFG\_SCHED\_COOPERATIVE ( 0 )

## Pre-emptive Scheduling

Pre-emptive scheduling means that a context-switch can be initiated even if the currently running task needs to use the CPU to complete its activity. In GOS2022, when pre-emptive scheduling is selected, at every system tick the kernel runs the scheduler algorithm to select the next task to use the CPU. The system tick is periodic and is set to 1kHz (re-scheduling takes place every 1ms). This means that tasks can potentially be interrupted every 1ms. However, tasks can of course voluntarily release the CPU when they finish their activity (yield, non-blocking delay, or waiting).

## Cooperative Scheduling

In cooperative scheduling mode, the system tick is only used to measure elapsed time, and the scheduler algorithm is not run. This means that tasks must voluntarily release the CPU (sleep, yield, or non-blocking waiting). Otherwise, once a task is granted permission to use the CPU, will have it until it releases it.

## Next Task Selection

When a re-scheduling happens, the following steps are done to select the next task that is granted permission to use the CPU:

* + Stack overflow check, and stack usage statistics update
  + Check and wake up sleeping tasks if their sleep period has elapsed
  + Check and unblock blocked tasks if their timeout has elapsed
  + Select the highest priority task that is in ready state, and different from the currently running task, and its CPU usage does not exceed the limit set by the user
  + Check the overall CPU usage limit, and in case it is exceeded, select the idle task
  + In case the selected task is not the currently running one, increase the context switch counter of the current task, and call the kernel swap hook function if registered
  + Update the runtime of the current task
  + Set the next task

## Stack Monitoring

GOS2022 provides stack monitoring for stack overflow detection but this service has limitations. As MMU-handling is not supported (and unavailable on the development platform), stack overflow can only be detected after its occurrence. Stack checking happens both on system tick event, and rescheduling. The stack pointer value of the currently running task is checked against the pre-defined stack boundaries. When the stack pointer points to a memory address outside the boundaries, a system error is invoked, and the OS is stopped. Otherwise, the maximum stack usage statistics is refreshed.

This means, that stack monitoring does not always detect stack overflow events (if there is a sudden stack overflow, it may cause a fault, before it could be detected).

## CPU-load Monitoring

Built-in CPU-load monitoring is provided by the kernel and the OS system task. The runtime of each task and a current monitoring time are updated when the next task selection happens. The idle task periodically calls the functions that calculates the CPU statistics of each task.

The system monitoring time is converted from a runtime type to microseconds, as well as the runtime of each task.

CPU-load is stored as 32-bit unsigned value in the range of 0…10000 and is interpreted as follows: XXXYY, where XXX is 0..100 and represents the integer part of the CPU-load percentage, whereas YY represents the fractional part.

CPU-load is calculated in the following way:

The idle task only refreshes a temporary variable for CPU-load measurement (monitoring load). The OS system task also refreshes the statistics periodically (but with a much longer period time), and it also resets the monitoring values which forces the actual load variables to be updated.

## CPU-load Limits

It is possible to set a global and task-level CPU-load limit. In practice it means that the kernel always checks the current load of each task, and if the load value exceeds the preset limit, the task cannot be selected to run even if it is ready and has the highest priority at the moment.

CPU-load limit will not guarantee that the limit is not exceeded (similarly to stack monitoring, it can only react after the limit has been exceeded), but on average, it ensures that the given task does not overconsume CPU time.

# **Chapter 3** Tasks

## Tasks

## Introduction

Tasks are essential entities in the system. A task is a piece of code that is executed in an endless loop. Tasks have assigned properties that influence their importance and behavior within the system.

## Idle Task

The idle task is a dedicated task that runs when there is no other task eligible to use the CPU. The idle task increases its run counter, calls the idle hook function (if registered), and updates the CPU statistics.

## System Task

The system task is a dedicated system service task. It calls the initializer functions of the configured OS services, and after it is done, it enables the scheduler and as a background task, it periodically updates the task CPU statistics, and handles the kernel dump request.

## System Daemons

System daemons are asynchronous background tasks of the OS services that handle time-insensitive, potentially CPU-heavy activities with low priority.

## Message Daemon

This task is responsible for message transfer between tasks. It loops through the internal message array and the waiter task array and if a message and a waiter match, it copies the message to the target buffer and unblocks the previously blocked task.

## Shell Daemon

Receives a character from the log serial line, if the echoing is on, then sends the same character back. When an enter key is received, it processes the command typed in, and loops through the command array. When the command is found, it calls the command handler function with the parameter list as a string.

## Signal Daemon

Polls the signal invoke queue, and completes the necessary signal invokings.

## Sysmon Daemon

Serves the incoming system monitoring messages (built-in and user-defined).

## Time Daemon

Increases the system time approximately every second and invokes the elapsed signals.

## Trace Daemon

Polls the trace queue and transmits the elements in the trace queue via the registered trace driver.

## Task State-Machine

Tasks are scheduled based on their current state. A task can be in the following states:

* + **Ready** – task is ready to run
  + **Sleeping** – task is waiting for a definite time to elapse and shall not be scheduled until it then
  + **Suspended** – task is waiting for an indefinite time to elapse and shall not be scheduled until then
  + **Blocked** – task is waiting for a resource and shall not be scheduled
  + **Zombie** – task is deleted and shall not be scheduled (permanent)

Only ready tasks can be executed. When sending to sleeping, suspended, or blocked state, tasks release the CPU immediately. Deleting a task means that the given task continues to exist in the memory but will not be scheduled in the future (virtually non-existent).

The possible state changes are as follows:

A computer screen shot of a computer

Description automatically generated

Figure 3: Task State-Machine

## Task Priority Levels

Task priority levels range from 0 to 255, 0 being the highest while 255 being the lowest priority. 255 is a dedicated priority and cannot be assign as it is the priority of the idle task.

The priorities of OS tasks can be changed in **gos\_config.h.**

/\*\*

\* Trace daemon task priority.

\*/

**#define** CFG\_TASK\_TRACE\_DAEMON\_PRIO ( 193 )

/\*\*

\* Message daemon task priority.

\*/

**#define** CFG\_TASK\_MESSAGE\_DAEMON\_PRIO ( 198 )

/\*\*

\* Signal daemon task priority.

\*/

**#define** CFG\_TASK\_SIGNAL\_DAEMON\_PRIO ( 197 )

/\*\*

\* Shell daemon task priority.

\*/

**#define** CFG\_TASK\_SHELL\_DAEMON\_PRIO ( 192 )

/\*\*

\* Time daemon task priority.

\*/

**#define** CFG\_TASK\_TIME\_DAEMON\_PRIO ( 196 )

/\*\*

\* System task priority.

\*/

**#define** CFG\_TASK\_SYS\_PRIO ( 195 )

/\*\*

\* Sysmon daemon task priority.

\*/

**#define** CFG\_TASK\_SYSMON\_DAEMON\_PRIO ( 40 )

The recommended priority distribution is as follows:

|  |  |  |
| --- | --- | --- |
| **Priority level** | **Name** | **Purpose** |
| 255 | Idle task | Dedicated |
| 254 … 200 | User low priority | Time-consuming user background activities |
| 199 … 100 | OS task and daemon priority | OS background activities |
| 99 … 0 | User medium and high priority | Important, time-critical activities |

Figure 4: Recommended Priority Levels

## Privileges

Tasks have assigned privileges. These privileges control whether a task has access to certain OS features (e.g. tracing, manipulation of other tasks). Privileges are represented by a series of privilege bits that are assigned to each task.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Bit** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **Privilege** | Task state manipulation | Task priority change | Tracing | Signal invoking | Reserved | Reserved | Reserved | Reserved | User-defined | User-defined | User-defined | User-defined | User-defined | User-defined | User-defined | User-defined |
| **Space** | Kernel | | | | | | | | User | | | | | | | |

Figure 5: Privilege Bits

## Kernel Privileges

Kernel privileges are dedicated privilege bits in the upper byte of the privilege word that control access to certain OS features.

* + Bit 0: Task state manipulation. Affects the following calls:
    - gos\_taskSuspend
    - gos\_taskResume
    - gos\_taskWakeup
    - gos\_taskBlock (except itself)
    - gos\_taskUnblock
    - gos\_taskDelete
  + Bit 1: Task priority change. Affects the following calls:
    - gos\_taskSetPriority
    - gos\_taskSetOriginalPriority
  + Bit 2: Tracing. Affects the following calls:
    - gos\_traceTrace
    - gos\_traceTraceFormatted
  + Bit 3: Signal invoking. Affects the following calls:
    - gos\_signalInvoke
  + Bit 4-7: Reserved for future use.

## User-defined Privileges

These are custom privileges that can be added and handled in the user application. These privilege bits have no effect on the OS.

## Creating and Registering Tasks

A task is described by the task descriptor structure which contains all the parameters that are necessary to define a task. When registering tasks, the related descriptor structure is passed to the kernel where a copy is made for later use in scheduling.

## Mandatory Task Parameters

The following parameters are mandatory for each task:

* Task function pointer
* Task stack size
* Task priority
* Task privilege level

## Optional Task Parameters

* Task name
* Task ID external variable
* Task CPU max usage [x100%]

## Single-shot, Cyclic, and Asynchronous Tasks

Tasks cannot be defined as single shot, cyclic, or asynchronous, however, these behaviors can be achieved by using certain patterns.

A single shot task can be implemented in the following way:

GOS\_STATIC void\_t **APP\_SingleShotTask** (void\_t)

{

// Task activity here.

**for** (;;)

{

// Recommended to call delete in a loop

// to prevent returning from task function.

(void\_t) gos\_taskDelete(singleShotTaskId);

}

}

Cyclic tasks can be implemented in the following way:

GOS\_STATIC void\_t **APP\_CyclicTask** (void\_t)

{

**for** (;;)

{

// Task activity here.

(void\_t) gos\_taskSleep(sleepTicks);

}

}

Asynchronous tasks can be implemented in the following way:

GOS\_STATIC void\_t **APP\_AsynchronousTask** (void\_t)

{

**for** (;;)

{

**if** (gos\_triggerWait(&trigger, triggerValue, tmo) == *GOS\_SUCCESS*)

{

// Activity happens when trigger is incremented.

}

}

}

# **Chapter 4** OS Drivers

## OS Drivers

Some OS services require platform-specific drivers. These cannot be implemented in the OS itself as it is intended to be platform-independent, therefore the OS defines *driver skeletons*.

## Introduction

The OS driver layer is a semi-virtual software layer that serves as the low-level implementation of functions that are used by certain OS services. Apart from the CRC driver, all other drivers exist in the form of driver skeletons and shall be implemented by the user in a platform-specific way.

## Driver Skeletons

Driver skeletons define function types with a given syntax to provide a middle layer between the platform-independent OS and the platform-specific low-level drivers.

The following skeletons are implemented in GOS2022:

* Shell driver
* Sysmon driver
* Timer driver
* Trace driver

The user-implemented drivers can be registered by calling **gos\_driverInit** that takes a pointer to a **gos\_driver\_functions\_t** as a parameter. This structure is defined as:

/\*\*

\* Driver functions type.

\*/

**typedef** **struct**

{

gos\_shellDriverReceiveChar\_t shellDriverReceiveChar;

gos\_shellDriverTransmitString\_t shellDriverTransmitString;

gos\_traceDriverTransmitString\_t traceDriverTransmitString;

gos\_traceDriverTransmitString\_Unsafe\_t traceDriverTransmitStringUnsafe;

gos\_timerDriverSysTimerGetVal\_t timerDriverSysTimerGetValue;

gos\_sysmonDriverTransmit\_t sysmonDriverTransmit;

gos\_sysmonDriverReceive\_t sysmonDriverReceive;

}gos\_driver\_functions\_t;

# **Table Of Figures**

[Figure 1: Layers of OS 8](#_Toc167298642)

[Figure 2: OS Service Dependencies 11](#_Toc167298643)

[Figure 3: Task State-Machine 18](#_Toc167298644)

[Figure 4: Recommended Priority Levels 19](#_Toc167298645)

[Figure 5: Privilege Bits 20](#_Toc167298646)