AKOS

v1.0.0

Generated by Doxygen 1.14.0

1 AKOS Documentation 1
1.1 Source Code Repository 1
1.2 Start Here 1
1.3 Section Overview 2
1.4 Downloads 2

2 Introduction 3
2.1 What AKOS Provides 3
2.2 Supported Architectures 3
2.3 Supported Compiler 3

3 Getting started 5
3.1 What The Board Gives You 5
3.2 Flash Layout 5
3.3 Recommended Setup 6
3.4 First Build 6
3.5 Flashing The Board 6
3.6 What To Change First 7
3.7 Result 7
3.8 Next Steps 7

4 Kernel basics 9
4.1 Kernel Objects 9
4.2 Threads 10
4.3 Messages 11
4.4 Timers 11
4.5 Priority And Scheduling 12
4.6 Kernel Control 12
4.7 Interrupt Context 12
4.8 Why This Matters 12

5 Memory management 13

6 Threads management 15

7 Scheduler 17
7.1 Priority Bitmap 17
7.2 How The Scheduler Works 18
7.3 What Makes A Thread Ready 20
7.4 What Makes A Thread Block 20
7.5 Related Pieces 20

8 Timers management 21

9 Inter-thread Synchronization 23

10 Inter-thread Communication 25

11 Porting 27

11.1 ARM Cortex-M3 Architecture . . . 27

11.1.1 Registers . . . 28

11.1.2 Operation Modes . . . 29

11.1.3 Interrupts and Exceptions . . . 31

11.1.4 SysTick Timer . . . 32

11.1.5 Supervisor Call (SVC) . . . 32

11.1.6 Pendable Service Call (PendSV) . . . 33

11.1.7 Reference . . . 33

11.2 Port Interrupt Management . . . 33

11.3 Port Layer Overview . . . 33

11.4 Reference Port . . . 34

11.5 Stack Initialization . . . 34

11.6 First Task Startup . . . 35

11.7 Context Switching . . . 35

11.8 Board Porting . . . 36

11.9 Related Files . . . 36

Index 37

Chapter 1

AKOS Documentation

Welcome to the AKOS documentation portal.

This site is the main reference for AKOS, an embedded real-time operating system built around a priority-based scheduler and an event-driven programming model.

1.1 Source Code Repository

The AKOS source tree lives in a separate repository:

- `https://github.com/the-ak-foundation/akos`

1.2 Start Here

If you are new to AKOS, the fastest way to navigate the material is:

1. Read the Introduction for the big picture.
2. Continue with Getting started for the first practical steps.
3. Move into Kernel basics to understand the core runtime.
4. Use the later sections as a reference when working on memory, threads, scheduling, timers, synchronization, communication, or porting.

1.3 Section Overview

- Introduction - high-level overview of AKOS and its design
- Getting started - practical entry point for new readers
- Kernel basics - core kernel concepts and runtime behavior
- Memory management - heap and allocation-related material
- Threads management - thread lifecycle and scheduling context
- Scheduler - scheduling rules and behavior
- Timers management - timer-related APIs and concepts
- Inter-thread Synchronization - coordination primitives
- Inter-thread Communication - message passing and data exchange
- Porting - adapting AKOS to new hardware targets
- `Files` - fast navigation and browsing of the AKOS source code

1.4 Downloads

- `PDF manual`

The PDF is published alongside the site, so you can download it directly from the docs homepage.

Chapter 2

Introduction

AKOS is a small embedded real-time operating system built for microcontrollers. It is organized around a priority-based scheduler and an event-driven programming model, so application code can react to signals, timers, and state changes instead of relying on one large super-loop.

2.1 What AKOS Provides

- Memory management
- Preemptive task scheduling
- Software timers
- Inter-thread communication
- Inter-thread synchronization
- Hardware-dependent port layer for context switching and tick handling

2.2 Supported Architectures

AKOS is primarily documented around the ARM Cortex-M family, where the current port layer provides the core scheduler and context-switch integration. The architecture split is kept deliberately small so additional targets can be ported without changing the higher-level kernel model.

- **ARM Cortex-M3**

2.3 Supported Compiler

The examples and build scripts in this repository are written for the GNU Arm Embedded toolchain. The compiler support is kept intentionally focused so the documented build and example flow stays consistent across the repo.

- **GNU Arm Embedded Toolchain**

Chapter 3

Getting started

This page uses the `AK Embedded Base Kit for STM32L151` as the reference board.

That board is the best place to begin because it already matches the AKOS target family, it ships with a board-specific boot/application memory map, and it exposes a simple path for loading application firmware over USB.

3.1 What The Board Gives You

The board README describes it as an evaluation kit for embedded learners. The main features called out there are:

- STM32L151-based hardware
- 1.54 inch OLED LCD
- 3 push buttons
- 1 buzzer
- RS485, Qwiic Connect, and Grove expansion support

For AKOS documentation purposes, the most important part is that the kit is already organized around a bootloader plus application split, which makes it a good reference for how to place an RTOS application in flash.

3.2 Flash Layout

The board repository documents this memory map:

- `0x08000000` for boot firmware
- `0x08002000` for shared BSF data
- `0x08003000` for the application image

That means an AKOS application for this board should not assume it starts at the beginning of flash. The linker script and any flash programming workflow need to respect the application start address at `0x08003000`.

3.3 Recommended Setup

If you are bringing AKOS up on this board, the practical setup is:

1. Clone the AKOS source tree.
2. Enter the repository root.
3. Read the board README and schematic in the reference repository.
4. Use the AKOS example tree as the application template.
5. Build the example for STM32L151 with the GNU Arm Embedded Toolchain.
6. Adjust the linker script and flashing command to place the application at `0x08003000`.

Typical setup commands look like this:

```
git clone https://github.com/the-ak-foundation/akos.git
cd akos
cd examples/00-blink
make
```

The current AKOS blink example already shows the STM32L151 toolchain shape:

- `akos/examples/00-blink/Makefile`
- `akos/examples/00-blink/stm32l151xx.ld`
- `akos/examples/00-blink/startup_stm32l151xb.s`
- `akos/examples/00-blink/system_stm32l1xx.c`
- `akos/examples/00-blink/board.h`

Those files are the right place to adapt pin mapping, clock setup, and the flash offset needed for this board.

3.4 First Build

For a first pass, the normal workflow is:

```
cd akos/examples/00-blink
make
```

If you are targeting the AK Embedded Base Kit specifically, update the board configuration so the final image is linked for the application region rather than the boot region.

3.5 Flashing The Board

The board README notes that once the boot and application images are loaded, you can use `AK - Flash` to program the application directly over the USB port.

That makes the board useful in two stages:

- Load or update the boot image when you are working on low-level startup
- Flash the application image repeatedly while developing AKOS tasks

For AKOS development, the second path is the one you will use most often.

3.6 What To Change First

When you port the example to this board, start with these items:

- `board.h` for LED, button, buzzer, and GPIO pin mappings
- `stm32l151xx.ld` for the application flash offset
- `system_stm32l1xx.c` for clock and oscillator setup
- `Makefile` for the final flash address and board-specific defines

3.7 Result

After the board is configured correctly, the example should build and run on the AK Embedded Base Kit with the expected LED activity.

3.8 Next Steps

Once the board boots and the application image flashes correctly, continue to:

- Kernel basics
- Threads management
- Scheduler

From there you can start turning the board's buttons, display, and buzzer into actual AKOS tasks and events.

Chapter 4

Kernel basics

Kernel objects are the building blocks of an RTOS. In AKOS, they are the runtime structures the kernel manages on behalf of the application: threads, messages, timers, priorities, and the control state that keeps them moving together.

This chapter gives you a simple map of those objects before the later chapters go into each subsystem in detail.

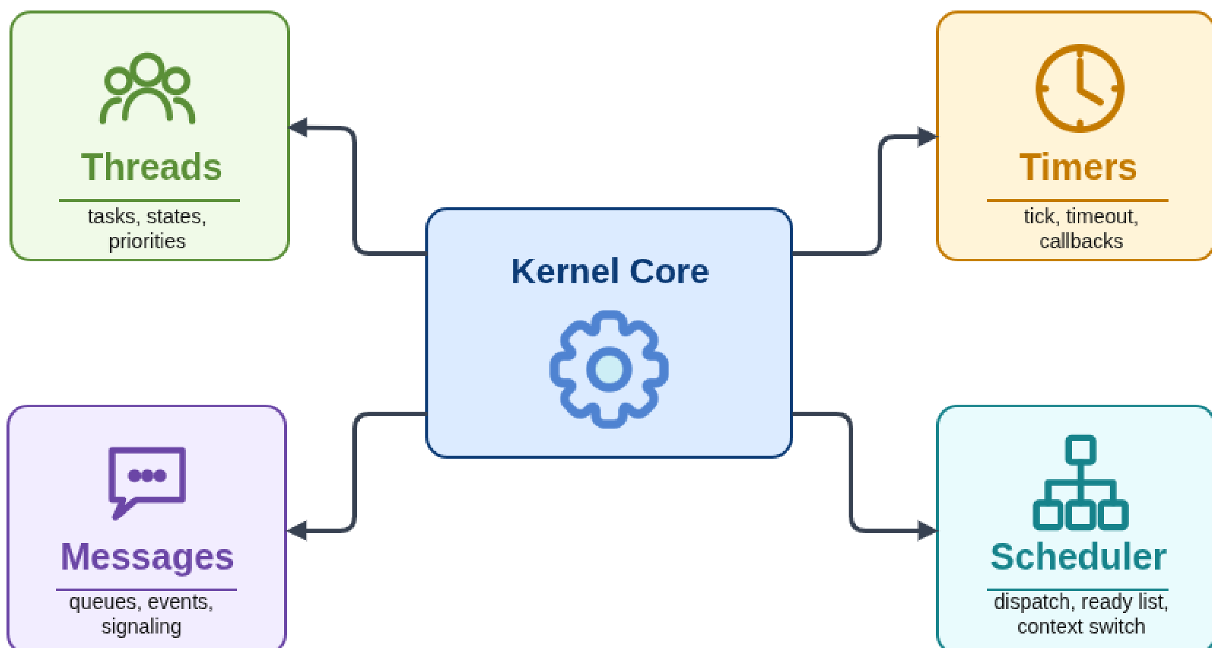

Figure 4.1 Kernel objects overview

4.1 Kernel Objects

At a high level, AKOS organizes its kernel around a few core objects:

- Threads for execution
- Message queues for communication
- Software timers for delayed work

- Priority state for run ordering
- Critical sections for protecting shared kernel data

These objects are not isolated features. They are connected: threads wait for messages, timers wake threads up, and the scheduler decides which ready thread runs next.

4.2 Threads

A thread is the basic unit of execution in AKOS. Each thread has:

- A thread ID
- An entry function
- An argument pointer
- A priority
- A message queue size
- A requested stack size

`AKOS_THREAD_DEFINE(...)` is the static thread-definition API. It creates the compile-time thread definition that the kernel later turns into a runtime thread object during system startup.

The thread subsystem also keeps track of runtime thread state, such as whether a thread is running, ready, delayed, or waiting on a message.

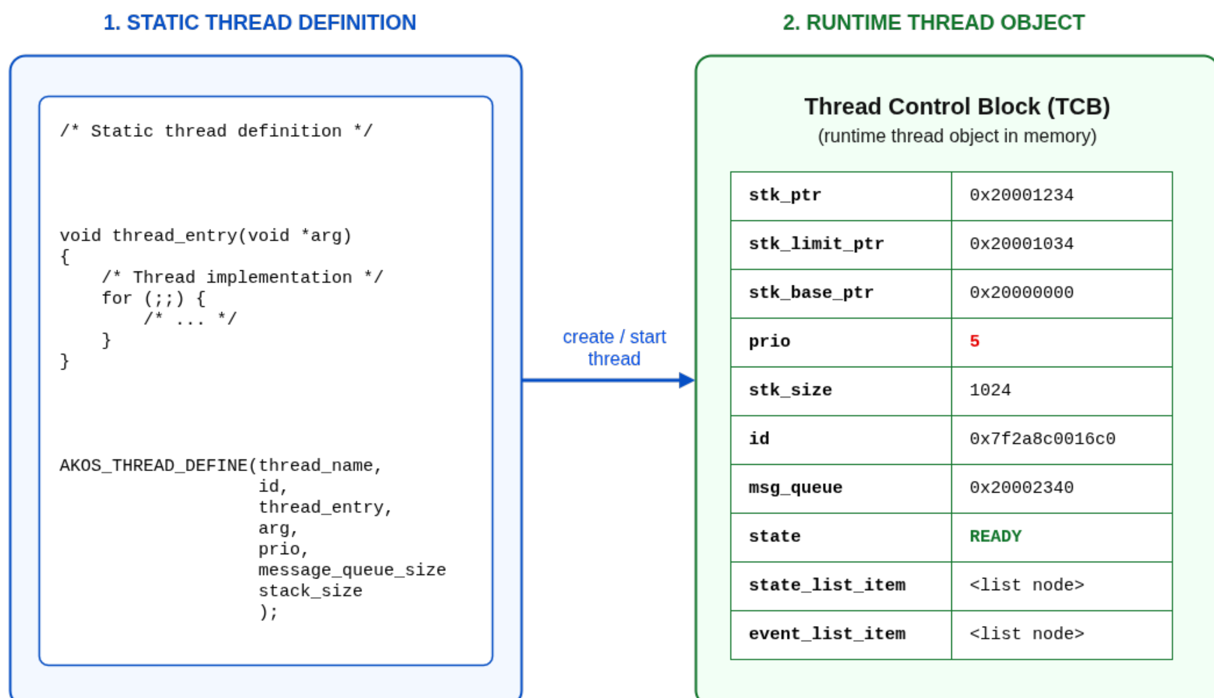

Figure 4.2 Thread objects diagram

4.3 Messages

Messages are the main communication object in AKOS. They let one thread wake another thread up with either:

- A pure signal
- A signal plus copied payload data

The message subsystem uses:

- `msg_t` for the message node
- `msg_queue_t` for the queue metadata
- `akos_message_queue_put_pure()`
- `akos_message_queue_put_dynamic()`
- `akos_message_queue_get()`

This gives application code a clean way to pass events between threads without sharing state directly.

4.4 Timers

Timers are another kernel object that build on top of the message system. A timer can expire once or repeat periodically, and when it expires it can post a signal to a destination thread or invoke a callback.

The timer subsystem revolves around `ak_timer_t` and APIs such as:

- `akos_timer_create()`
- `akos_timer_start()`
- `akos_timer_reset()`
- `akos_timer_remove()`

Timers are especially useful when you want delayed work, periodic polling, small callback functions, or time-based events without blocking a thread.

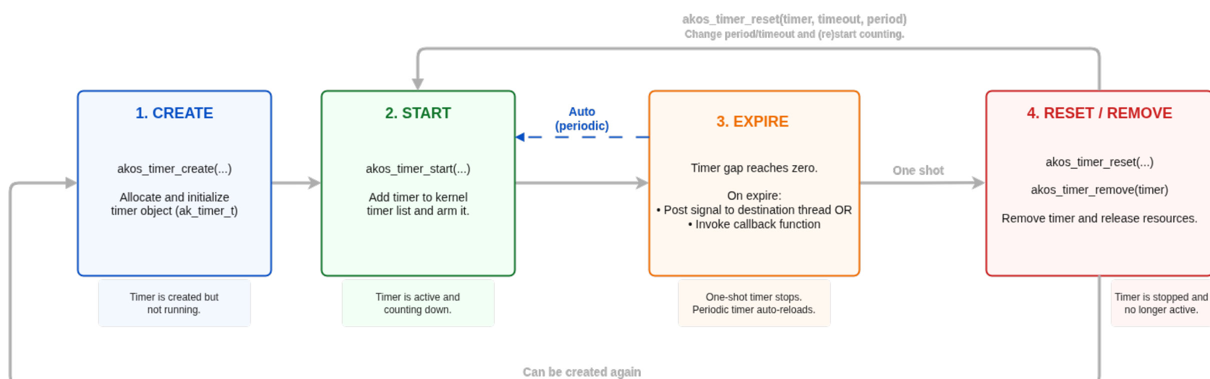

Figure 4.3 Timers life cycle

4.5 Priority And Scheduling

Priority is what ties the kernel objects together. AKOS uses priority tracking to decide which ready thread should run next.

That means:

- A higher-priority ready thread can preempt lower-priority work
- Delayed or blocked threads stay out of the ready set
- The kernel can quickly pick the next runnable thread

The scheduler is the control point that turns thread state into actual CPU execution.

4.6 Kernel Control

AKOS also needs a small amount of kernel control logic around those objects. That is where `core` comes in.

The main control APIs are:

- `akos_core_init()`
- `akos_core_run()`
- `akos_core_enter_critical()`
- `akos_core_exit_critical()`

`akos_core_init()` prepares the kernel subsystems, while `akos_core_run()` hands control to the scheduler. The critical-section APIs protect shared kernel data structures so the object lists stay consistent across interrupts and thread context.

4.7 Interrupt Context

Interrupt service routines are not kernel objects in the same sense as threads or timers, but they matter because they can interact with the kernel at the boundaries.

In AKOS, interrupts typically feed events into the system, while the kernel keeps the runtime objects consistent and decides when a thread should wake up and run.

4.8 Why This Matters

Once you understand the kernel objects, the rest of the documentation becomes much easier to read:

- Threads management explains how threads are created and managed.
- Scheduler explains how priorities turn into run order.
- Timers management explains delayed and periodic work.
- Inter-thread Communication explains how messages move between threads.

The short version is simple: AKOS is built from a handful of kernel objects, and the scheduler keeps them working together.

Chapter 5

Memory management

Chapter 6

Threads management

Chapter 7

Scheduler

The scheduler decides which thread runs next. In AKOS, it is the part of the kernel that keeps the ready threads organized and picks the next thread to run.

That gives AKOS a simple but practical scheduling model for embedded systems: urgent work runs first, and equal-priority work shares the CPU fairly.

7.1 Priority Bitmap

AKOS keeps a ready list for each priority level, and the priority bitmap tracks which of those levels currently have ready work.

The priority subsystem uses one bit per priority level:

- A set bit means that priority has at least one ready thread
- A clear bit means that priority has no ready threads

Lower numeric values mean higher priority, so the scheduler always looks for the smallest ready priority value first.

The related APIs are:

- `akos_priority_init()`
- `akos_priority_insert()`
- `akos_priority_remove()`
- `akos_priority_get_highest()`

When a thread becomes ready, the kernel inserts it into the list for its priority. If that list was empty before, the matching bit in the priority table is set. If the list becomes empty again, the bit is cleared.

The bitmap makes it fast to find the highest ready priority without scanning every ready list one by one.

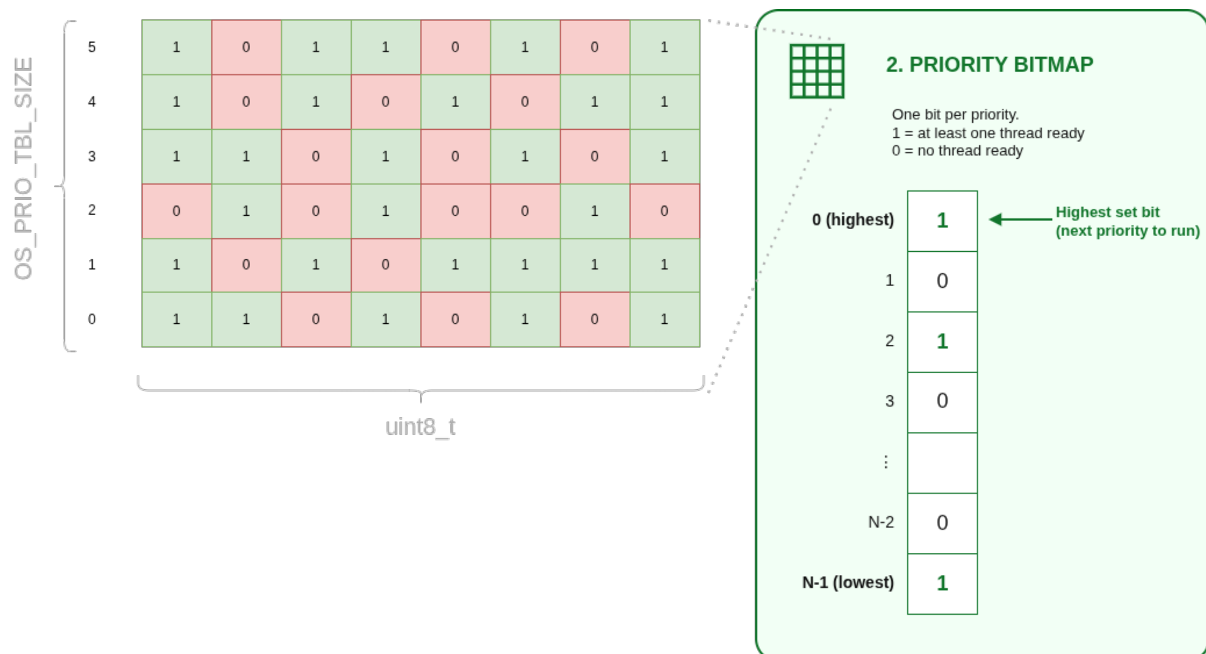

Figure 7.1 Priority bitmap table

7.2 How The Scheduler Works

The AKOS scheduler combines three ideas into one flow: **priority based** scheduling, **round robin** scheduling, and **tick and time slicing**.

- **Priority based** scheduling decides which priority level should run first.
- **Round robin** scheduling decides which thread should run next inside that priority level.
- **Tick and time slicing** logic drives wakeups, timeout expiry, and rotation through the ready list.

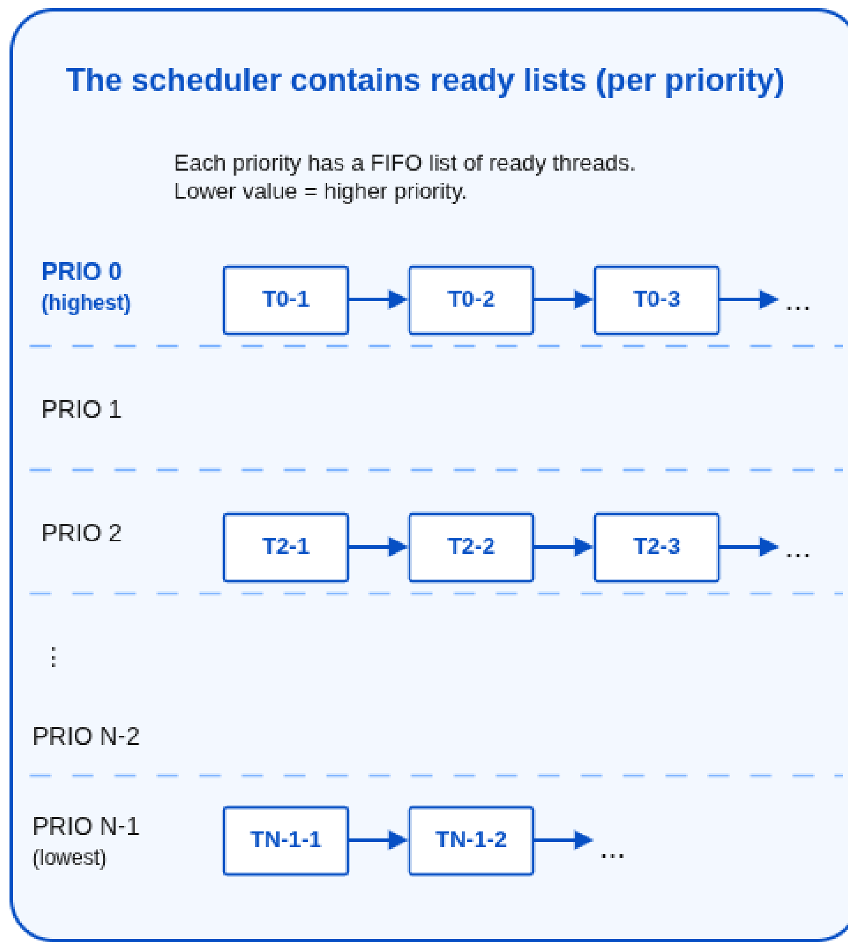

Figure 7.2 Scheduler's work overview

The full flow looks like this:

1. The tick counter increments.
2. Delayed threads whose wake time has arrived move back to the ready lists.
3. The scheduler checks the priority bitmap to find the highest ready priority.
4. If only one thread is ready at that priority, the scheduler runs it.
5. If more than one thread is ready at that priority, the scheduler advances to the next thread in that ready list.

That is why the scheduler works well for AKOS:

- High-priority work can preempt lower-priority work.
- Equal-priority threads share CPU time fairly.
- Timer and delay events can feed directly back into scheduling.

AKOS uses priority-based preemptive scheduling. A thread with a higher priority can take the CPU from a lower-priority thread when it becomes ready.

This can happen when:

- A delayed thread wakes up
- A message arrives for a waiting thread
- The tick handler moves a thread from blocked to ready
- Application code unblocks a higher-priority thread through a kernel service

When that happens, AKOS can trigger a context switch so the higher-priority thread runs as soon as possible.

7.3 What Makes A Thread Ready

A thread enters the ready state when it is able to run again. Common cases are:

- The thread is created during startup
- A delay expires
- A message arrives for a waiting thread
- The kernel moves it back from a blocked state

When that happens, AKOS inserts the thread into the appropriate ready list and updates the priority bitmap if needed.

7.4 What Makes A Thread Block

Threads can leave the ready state when they:

- Call `akos_thread_delay()`
- Wait for a message
- Are suspended by the kernel or application logic

Blocked threads stay out of the ready lists until the kernel decides they are eligible to run again.

7.5 Related Pieces

The scheduler chapter connects directly to a few other parts of the kernel:

- Kernel basics introduces the main kernel objects
- Threads management explains thread lifecycle and state
- Timers management explains how timer expiry feeds back into scheduling

The short version is: AKOS always runs the highest-priority ready thread, and threads at the same priority take turns in the ready list.

Chapter 8

Timers management

Chapter 9

Inter-thread Synchronization

Chapter 10

Inter-thread Communication

Chapter 11

Porting

Porting is the work needed to make AKOS run on a specific CPU architecture and board. The kernel provides the scheduling and runtime logic, while the port layer supplies the low-level CPU and startup support that the kernel depends on.

For the current tree, the Cortex-M3 port is the reference implementation.

11.1 ARM Cortex-M3 Architecture

ARM Cortex-M3 is the CPU architecture AKOS targets in this repository. It uses the ARM exception model, a separate system tick timer, and a nested interrupt controller to handle hardware events and kernel scheduling work.

For AKOS, the most important Cortex-M3 concepts are the register model, processor modes, interrupt controller, memory layout, instruction set, and exception flow.

Figure 11.1 ARM Cortex-M3 Architecture

11.1.1 Registers

The Cortex-M3 register set defines the CPU state that the port layer must create, save, and restore during startup and context switching.

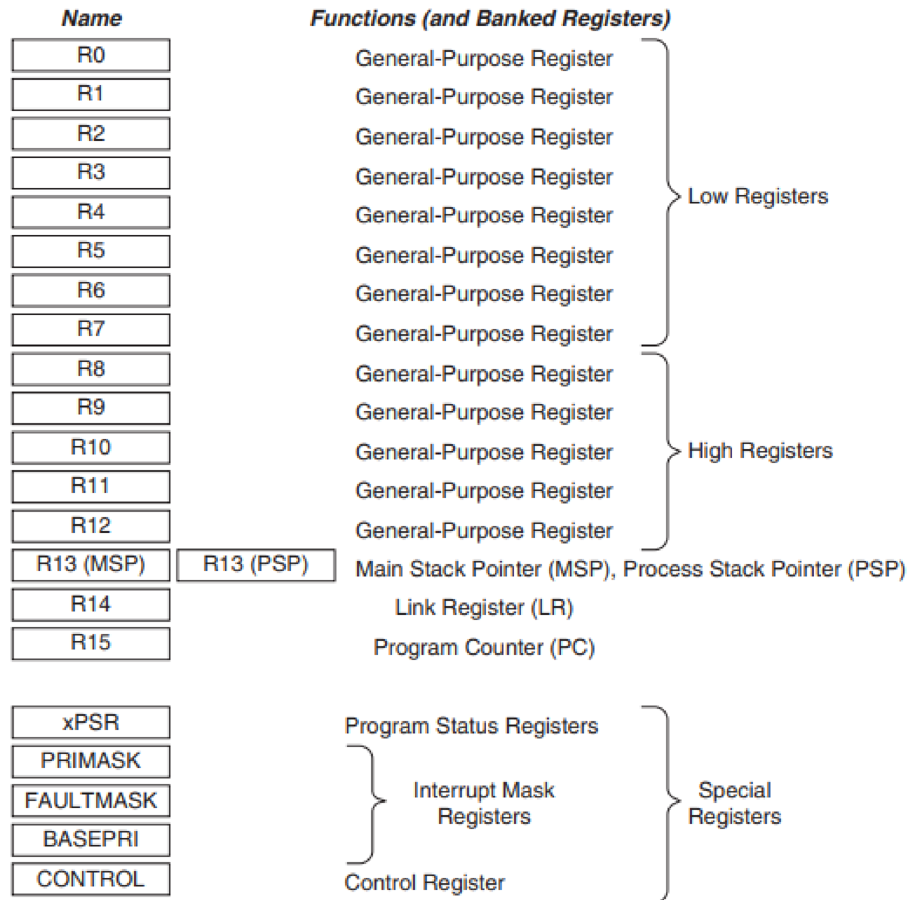

Figure 11.2 ARM Cortex-M3 registers set

The most important registers for AKOS are:

- `R0` to `R12` for general-purpose data and parameter passing
- `SP` for the current stack pointer
- `LR` for return state and exception return behavior
- `PC` for the current execution address
- `xPSR` for processor status, including Thumb state
- `MSP` for the main stack pointer used during reset and exceptions
- `PSP` for the process stack pointer used by threads in normal execution

For a new thread, the port layer builds an initial stack frame that matches the register state the CPU expects to restore on exception return.

11.1.2 Operation Modes

The Cortex-M3 has two related concepts:

- Execution mode: `Thread mode` and `Handler mode`

- Privilege level: `Privileged` and `Unprivileged`

These are not the same thing.

The execution modes are:

- Thread mode for normal application and RTOS thread execution
- Handler mode for exceptions such as SysTick, SVC, and PendSV

The privilege relationship is:

- Handler mode always runs as privileged
- Thread mode can run as privileged or unprivileged

	Privileged	Unprivileged
Handler mode	Interrupts + Exceptions	
Thread mode	Main program	User programs

Figure 11.3 Operation modes and privilege levels

That means normal program code usually runs in Thread mode, while exceptions run in Handler mode. If Thread mode is unprivileged, it cannot execute some privileged instructions or directly access some protected system resources.

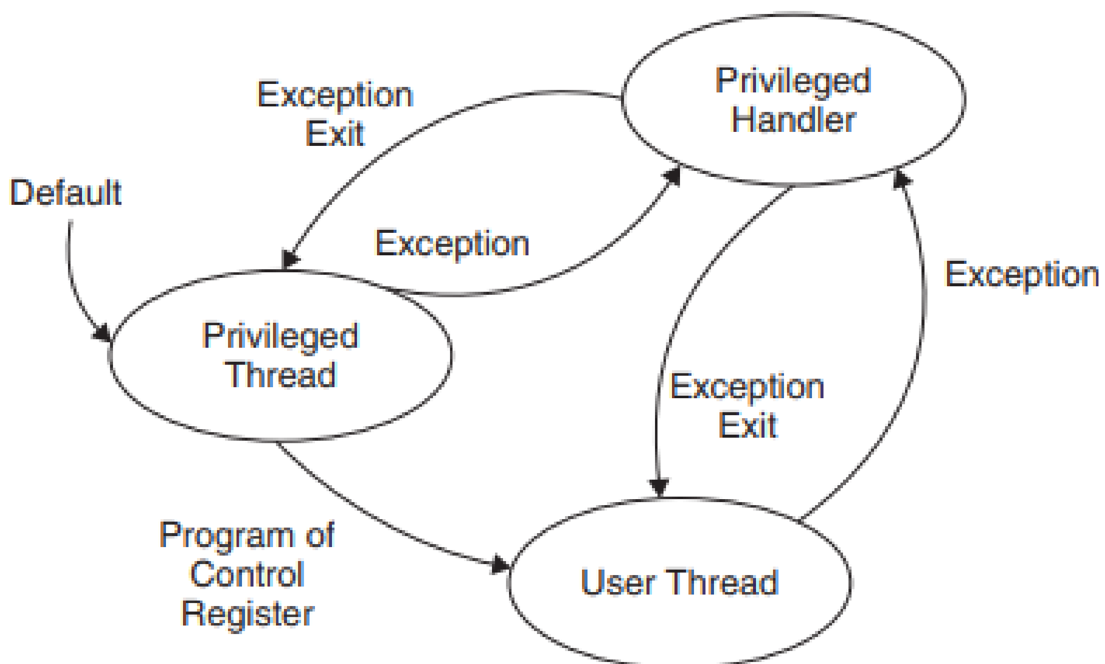

Figure 11.4 Modes transition

AKOS relies on this split so application code runs in Thread mode while the port layer performs startup, tick handling, and context switching in Handler mode.

11.1.3 Interrupts and Exceptions

On Cortex-M3, interrupts are handled through the exception system. The CPU uses exceptions for both external interrupts and internal events, so the port layer works through that model when it starts threads and requests scheduling work.

For AKOS, the important parts are:

- SysTick for the kernel time base
- SVC for controlled entry into the first thread
- PendSV for deferred thread switching
- External interrupts for device-driven events around the kernel

These features let the kernel keep normal thread execution separate from the low-level exception flow that the port layer manages.

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (external NMI input)
3	Hard fault	–1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system service
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...	...	...	...
255	IRQ #239	Programmable	External interrupt #239

Figure 11.5 Cortex M3 exception types

11.1.4 SysTick Timer

SysTick is a built-in Cortex-M3 system timer. It is a 24-bit down-counter that can generate a periodic exception when it reaches zero.

For AKOS, SysTick is important because it provides the regular time base used by the kernel.

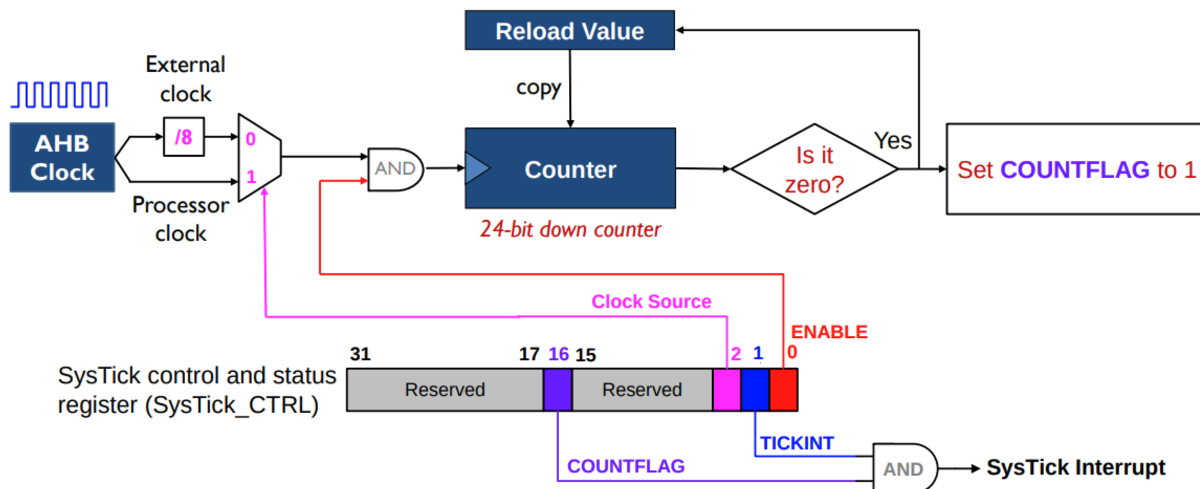

Figure 11.6 Systick diagram

The typical SysTick flow is:

- Load the reload register with the tick interval
- Start the counter with the selected clock source
- Count down to zero
- Raise the SysTick exception
- Reload automatically and begin the next tick period

In AKOS, this periodic exception is used to:

- Increment the kernel tick count
- Wake delayed threads when their timeout expires
- Decide whether a context switch is needed

That is why SysTick is one of the core hardware blocks the port layer must configure during startup.

11.1.5 Supervisor Call (SVC)

`SVC` is a system exception triggered by software through the `SVC` instruction. It is used when software wants to enter a controlled supervisor service path through the exception mechanism.

In AKOS, the purpose of `SVC` is to start the first runnable thread after the kernel has finished its startup work. The port triggers `SVC`, enters the SVC handler, restores the first thread context, and then returns into thread execution through the normal exception-return path.

11.1.6 Pendable Service Call (PendSV)

`PendSV` is a system exception used for deferred service work. Software can set it to pending and let the CPU handle it later through the normal exception mechanism.

In AKOS, the purpose of `PendSV` is to perform context switching. When the kernel decides that another thread should run, it pends `PendSV`, enters the PendSV handler, saves the current thread context, restores the next thread context, and then returns to thread execution.

11.1.7 Reference

- Cortex-M3 processor from ARM
- The Definitive Guide to the ARM Cortex-M3, 2nd Edition
- The Definitive Guide to the ARM Cortex-M3.pdf
- Cortex-M3 Technical Reference Manual
- Systick

11.2 Port Interrupt Management

AKOS uses the port layer to control the interrupt state around kernel-critical work. The kernel depends on the port to temporarily mask interrupts when it needs to protect shared runtime data.

The reference port provides the basic interrupt helpers:

- `port_disable_interrupts`
- `port_enable_interrupts`

These helpers are used by kernel critical sections so the kernel can update ready lists, message queues, and other shared structures safely.

The port layer also configures exception priorities for scheduler-related exceptions. In the current Cortex-M3 port, PendSV is assigned a very low priority so context switching can be deferred safely, while SysTick is configured as the periodic kernel tick source.

11.3 Port Layer Overview

The port layer sits between the kernel and the hardware. It is responsible for the pieces that are too architecture-specific for the kernel itself:

- Building the initial thread stack frame
- Starting the first runnable thread
- Requesting and handling thread switches at the CPU level
- Setting up the system tick source
- Providing interrupt-masking helpers for kernel critical sections

11.4 Reference Port

AKOS uses a reference CPU port in this repository. It lives under `akos/port/arm/cortex-m3/` and provides the CPU-facing hooks that the kernel calls during startup and scheduling.

The main port APIs are:

- `akos_port_task_stack_init()`
- `akos_port_start_first_task()`
- `akos_port_systick_init_freq()`

The port layer also exposes helper macros such as:

- `port_disable_interrupts`
- `port_enable_interrupts`
- `port_setup_PendSV()`
- `port_trigger_PendSV()`

The reference port also provides exception handlers that connect the CPU's exception model to the kernel runtime:

- `port_SVCHandler`
- `port_PendSVHandler`
- `port_SysTickHandler`

11.5 Stack Initialization

Before a thread can run, the port layer prepares the stack so the CPU can enter the thread entry function using the normal exception return path.

`akos_port_task_stack_init()` does three important things:

- It starts from the top of the thread stack buffer
- It aligns the stack pointer for Cortex-M exception use
- It writes an initial register frame that looks like a thread was already interrupted and is ready to be restored

The prepared frame includes values for:

- `xPSR`, with the Thumb-state bit set
- `PC`, set to the thread entry function
- `LR`, set to an initial placeholder return value
- `R0`, set to the thread argument

- Space for the remaining registers restored during startup or switching

The stack layout is prepared to match the normal Cortex-M exception model:

- Hardware-restored registers: `R0` to `R3`, `R12`, `LR`, `PC`, and `xPSR`
- Software-restored registers in the AKOS port: `R4` to `R11`

That means a new thread starts with:

- `PC` pointing to the thread entry function
- `R0` holding the thread argument
- `xPSR` in valid Thumb state

When the port later restores this stack, the CPU can continue as if the thread had already been running and was simply being resumed after an exception.

That gives a newly created thread the same kind of starting state that a restored thread would have after a context switch. In other words, the first thread start and a later thread restore both follow the same Cortex-M return model.

11.6 First Task Startup

AKOS starts the first runnable thread through `akos_port_start_first_task()`. That function sets up the processor state and then triggers `SVC 0` so the port layer can restore the initial thread context.

The startup flow is:

1. The kernel initializes its subsystems.
2. The kernel selects the first runnable thread.
3. The port layer restores that thread context.
4. Execution continues in thread mode.

11.7 Context Switching

When the kernel decides a different thread should run, it asks the port layer to perform a switch.

In the current reference port, that work is handled through PendSV. The kernel updates `tcb_high_rdy_ptr` and then pends the exception, and the handler performs the CPU-level context save and restore.

The switch flow is:

1. Read the current thread stack pointer from `PSP`.
2. Save `R4` to `R11` onto the current thread stack.
3. Store the updated stack pointer back into the current thread control block.

4. Update `tcb_curr_ptr` to the next ready thread.
5. Load the next thread stack pointer from its control block.
6. Restore `R4` to `R11` from the next thread stack.
7. Write the new stack pointer back into `PSP`.
8. Return from PendSV so the CPU restores the remaining hardware-stacked registers and resumes the next thread.

The Cortex-M hardware already stacks `R0` to `R3`, `R12`, `LR`, `PC`, and `xPSR` on exception entry. The PendSV handler only has to save and restore the remaining software-saved registers `R4` to `R11`.

PendSV is a good fit for this role because it runs at very low priority and lets the CPU defer the actual switch until it is safe to do so.

11.8 Board Porting

The CPU port is only part of the job. A new board still needs its own startup, clock, memory map, and peripheral setup around AKOS.

For a new target, the board-side work usually includes:

- Startup code and vector table setup
- Clock tree setup and providing the correct core clock value
- Linker script changes for flash and RAM layout
- Board-specific pin, LED, button, and console setup

In AKOS, that board-specific work lives mostly in the example and platform files, while the port layer stays focused on CPU behavior such as stack-frame layout, exception handling, and SysTick register programming.

11.9 Related Files

The Cortex-M3 port is implemented in:

- `akos/port/arm/cortex-m3/port.c`
- `akos/port/arm/cortex-m3/port.h`

For scheduler behavior, see Scheduler. For thread behavior, see Threads management.

Index

AKOS Documentation, 1

Getting started, 5

Inter-thread Communication, 25
Inter-thread Synchronization, 23
Introduction, 3

Kernel basics, 9

Memory management, 13

Porting, 27

Scheduler, 17

Threads management, 15
Timers management, 21