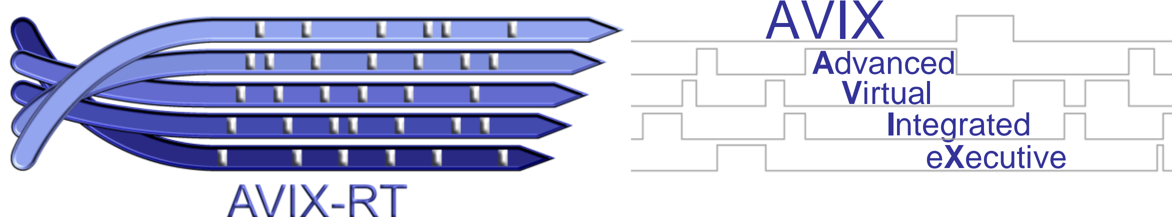
AVIX Real Time Operating System

User Guide
&
Reference Guide

Product Version 5.0.0



© 2006-2012, AVIX-RT

All rights reserved. This document and the associated AVIX software are the sole property of AVIX-RT. Each contains proprietary information of AVIX-RT. Reproduction or duplication by any means of any portion of this document without the prior written consent of AVIX-RT is expressly forbidden.

AVIX-RT reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of AVIX. The information in this document has been carefully checked for accuracy; however, AVIX-RT makes no warranty pertaining to the correctness of this document.

Trademarks

AVIX, AVIX for PIC24-dsPIC, AVIX for PIC32MX and AVIX for Cortex-M3 are trademarks of AVIX-RT. All other product and company names are trademarks or registered trademarks of their respective holders.

Warranty Limitations

AVIX-RT makes no warranty of any kind that the AVIX product will meet the USER's requirements, or will operate in the manner specified by the USER, or that the operation of the AVIX product will operate uninterrupted or error free, or that any defects that may exist in the AVIX product will be corrected after the warranty period. AVIX-RT makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the AVIX product. No oral or written information or advice given by AVIX-RT, its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty and licensee may not rely on any such information or advice.

AVIX-RT
Maïsveld 84
5236 VC 's-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of Contents

1 What is AVIX.....1

2 What makes AVIX unique.....2

3 Glossary.....3

4 About this document.....5

4.1 Reader guidance.....5

5 Hardware Resource Usage.....6

5.1 Interrupts.....6
5.2 Timer.....6
5.3 Memory.....6

6 User Guide.....7

6.1 Global Architecture.....8
6.2 Runtime Behavior and Responsibilities.....9
6.3 Thread and Interrupt Priorities.....10
6.4 Application Structural Building Blocks.....12
6.4.1 Threads.....12
6.4.2 Interrupt Handling.....19
6.4.3 Deferred Interrupt Handlers (DIH's).....23
6.4.4 System Stack.....28
6.5 Application Support Functionality.....30
6.5.1 Kernel Object Management.....30
6.5.2 Timeouts.....38
6.5.3 Error Handling.....40
6.5.4 Thread Activation Tracing.....41
6.5.5 Power Management.....43
6.6 Synchronization & Communication Services.....46
6.6.1 Mutex Services.....47
6.6.2 Semaphore Services.....50
6.6.3 Event Group Services.....53
6.6.4 Timer Services.....58
6.6.5 Message Services.....66
6.6.6 Pipe Services.....70
6.6.7 Memory Pool Services.....91
6.6.8 Exchange Services.....96
6.7 What to do in case of problems.....112

7 Reference Guide.....114

7.1 Conventions.....114
7.1.1 Naming.....114
7.1.2 Type Safety.....114
7.1.3 When is an AVIX function allowed to be used?.....114
7.2 Function Overview.....116
7.3 Definition Overview.....120
7.4 Type Overview.....122
7.5 Environment definitions.....124
7.6 AVIX Application Programming Interface (API).....125
7.6.1 Thread Support.....126
7.6.2 Mutex Support.....141
7.6.3 Semaphore Support.....147
7.6.4 Event Support.....154
7.6.5 Timer Support.....163
7.6.6 Message Support.....180
7.6.7 Pipe Support.....208
7.6.8 Memory Support.....229

7.6.9 Exchange Support 240
7.6.10 Power Mode support 268
7.6.11 Interrupt Support 275
7.6.12 Error Support 278
7.6.13 Object Support 282

List of Figures

Figure 1: Global AVIX Architecture 8
Figure 2: Runtime Behavior and Initialization 10
Figure 3: Thread and Interrupt Priorities 11
Figure 4: Priority inversion 17
Figure 5: Priority inheritance 18
Figure 6: Sample Thread Activation Tracing diagram 42
Figure 7: Timer, relation between system timer and application timer period 62
Figure 8: Cyclic timer period change with disturbance 63
Figure 9: Cyclic timer period change without disturbance 63
Figure 10: Basic pipe operation 71
Figure 11: Pipe structure with pending requests 72
Figure 12: Synchronous pipe usage with equal write-read size 75
Figure 13: Synchronous pipe usage with different write-read size 75
Figure 14: Using pipes from DIH's 80
Figure 15: Using pipes from ISR's 81
Figure 16: Pipe based ISR Thread communication, 10μs interrupt rate 83
Figure 17: Pipe based ISR Thread communication, 10μs interrupt rate 84
Figure 18: Pipe based ISR Thread communication, 4μs interrupt rate 84
Figure 19: Pipe based ISR Thread communication, 2μs interrupt rate 84
Figure 20: Pipe based UART transmission Thread Activation diagram 88
Figure 21: Exchange Sample Application without using Exchange Services 96
Figure 22: Exchange sample application using Exchange Services 98

List of Tables

Table 1: Terms and Abbreviations 4
Table 2: Kernel Object Id Types 30
Table 3: AVIX timer state transitions 65
Table 4: Exchange object access types and user lock sequences 104
Table 5: AVIX API Functions 119
Table 6: AVIX API Definitions 122
Table 7: AVIX API Types 123
Table 8: AVIX Environment Definitions 124

List of Code Samples

Code sample 1: How to use threads with individual thread code 13
Code sample 2: How to use threads with shared thread code 13
Code sample 3: How to declare an ISR using the compiler syntax 20
Code sample 4: How to use a DIH 24
Code sample 5: How to use a potentially blocking AVIX function from a DIH 26
Code sample 6: How to declare an ISR using the AVIX software system stack syntax 29
Code sample 7: Usage of thread id's as event group id's 31
Code sample 8: How to let threads share kernel object id's through a global variable 32
Code sample 9: How to let threads share kernel object id's through named objects 33
Code sample 10: How to use type safe function parameters and return values 35
Code sample 11: How to manipulate type safe variables and constants 36
Code sample 12: How to perform type safe value conversions 37

Code sample 13: How to use a timeout based on the AVIX API functions....................................38
Code sample 14: How to use a timeout based on the AVIX helper macro....................................39
Code sample 15: Install a user error handler 40
Code sample 16: How to use Thread Activation Tracing..42
Code sample 17: Using a mutex to guard access to shared resource 48
Code sample 18: Using a mutex to guard access to shared resource 49
Code sample 19: How to use a semaphore to control resources with multiple instances 51
Code sample 20: How to use event groups 54
Code sample 21: How to use thread event groups..55
Code sample 22: Using a cyclic timer...59
Code sample 23: Using a cyclic timer with a connected event group..60
Code sample 24: How to specify a timer period 62
Code sample 25: How to use a message queue..67
Code sample 26: How to use a message queue with connected event group 69
Code sample 27: Creating a pipe 70
Code sample 28: Synchronous pipe usage with equal write-read size 74
Code sample 29: Synchronous pipe usage with different write-read size 75
Code sample 30: Asynchronous pipe usage...76
Code sample 31: Status and abort usage of asynchronous pipe requests.....................................78
Code sample 32: How to use a pipe device callback function 85
Code sample 33: How to use a pipe device callback function to send UART data.........................88
Code sample 34: How to create a memory pool 92
Code sample 35: How to allocate and access memory blocks 93
Code sample 36: How to access memory blocks using plain 'C' array operations 93
Code sample 37: How to make heap operations thread safe 95
Code sample 38: How to create an exchange object 101
Code sample 39: Reading and writing an exchange object..102
Code sample 40: Unsafe modification of the content of an exchange object 102
Code sample 41: Safe modification of the content of an exchange object 103
Code sample 42: Direct modification of the content of an exchange object 104
Code sample 43: Setting up and using an exchange thread message queue connection 107
Code sample 44: Setting up and using an exchange thread event group connection 108
Code sample 45: Setting up and using an exchange callback connection 109

1 What is AVIX

AVIX is a preemptive real time operating system (RTOS) available for a number of the most modern hardware platforms.

An AVIX based application is structured as a collection of autonomous functions. These functions are activated by AVIX as threads. Communication between threads is established through AVIX supplied mechanisms. The result is an application with a high degree of modularity leading to shorter development time, easier testing and a high degree of reuse of the individual software modules.

An AVIX based application works fully event driven, leading to an optimal utilization of the available processing power with a deterministic timing.

Additionally, AVIX helps to make an application more flexible, reliable, and deterministic by offering a combination of features not found in any competing product.

The most prominent of these features is that AVIX never disables interrupts, not for a single cycle. With AVIX, interrupt latency equals that of the underlying hardware platform where competing products disable interrupts sometimes up to hundreds of cycles.

Still AVIX ISR's may use AVIX functions to communicate with threads, allowing them to be fully integrated in the application[1]. As a result AVIX is capable of dealing with extremely high interrupt rates, interrupt rates no competing product is able to reach. AVIX makes losing interrupts a thing of the past. This feature, combined with full support for the nested interrupt architecture of the underlying hardware platform makes AVIX deserve its title of:

The world's fastest True Zero Latency Interrupt Handling RTOS[2]

AVIX largely decreases the RAM footprint of an application by allowing a dedicated stack to be used by ISR's (system stack). This system stack is shared by all ISR's, lowering RAM consumption of a typical application up to 50% compared with competing products. Furthermore, use of the system stack leads to a highly predictable system since random stack overflows caused by coinciding interrupts is easier to detect.

This feature is present regardless if the underlying hardware platform implements a hardware system stack or not. For hardware platforms not offering a hardware system stack, the system stack is implemented in software offering the same advantages.

AVIX offers a tracing mechanism allowing non-intrusive real time monitoring of thread activity. This tracing mechanism provides unprecedented insight in all timing aspects of the application. It can be used for application tuning and fault seeking in a way not seen before and makes you feel confident your design decisions have the intended result.

[1] A number of competing products also claiming not to disable interrupts only allow this for Interrupt Service Routines that may not use any RTOS service to communicate with threads making this claim meaningless. For this reason we name this feature '**True** Zero Latency'.

[2] True Zero latency refers to the latency added by AVIX to the hardware latency. It shall be obvious that AVIX is not capable of decreasing the interrupt latency as dictated by the underlying hardware platform and related conventions.

2 What makes AVIX unique

AVIX offers a combination of features not found in any other RTOS. The total list of features is quite extensive, but a number of them deserve special mentioning:

- **Interrupt integration with Zero Added Latency:** Most embedded systems interact with their environment using interrupts. Interrupt latency is often required to be as low as possible. Often the worst case interrupt latency figure is determined by the RTOS because it is common for an RTOS to disable interrupt processing for a substantial period of time. Not so with AVIX. AVIX does not disable interrupts during its processing, not for a single machine cycle. Still, AVIX does allow for ISR's to communicate with threads and vice versa, as such offering full interrupt integration with the application. This is unparalleled in industry and means AVIX is suited to do what it is created for, offer a high programming abstraction level for the basic interrupt handling where interrupt latency is determined by the hardware only.

- **Advanced System Stack**: RTOS based applications support multiple threads. Each thread has its own stack. A downside of most RTOSes is that these thread stacks are also used for ISR context saving implying each of the thread stacks has to reserve space for this. Since hardware platforms targeted by AVIX support nested interrupts, the stack memory requirement for interrupts can be substantial. AVIX offers a system stack for exclusive use by ISR's implying ISR's do not use the stacks of individual threads. This system stack either is present in the underlying hardware platform (hardware system stack) or AVIX offers a software implementation (software system stack).

  The AVIX software system stack has an advantage over implementations offered by competing products. Often the switch to the software system stack is made only *after* the ISR's make initial use of the individual thread stacks. For this reason, even though a software system stack is offered, memory overhead still can be substantial. AVIX offers a unique mechanism for its software system stack which makes the load on the thread stacks really small. Depending on the hardware platform, this load can be as low as zero, saving up to 50% of RAM for a typical application.

- **High Level Application Support through Exchange Objects**: AVIX Exchange objects offer a high level inter thread communication mechanism allowing to construct highly modular and reusable software components.

- **Advanced Power Management**: AVIX is able to exploit the energy saving capabilities of the hardware platform it supports. This is done in a way much more advanced than seen with most competing products. AVIX power management can be as easy as making a single function call during application initialization.

- **Real Time Thread Activation Tracing**: Activation and deactivation of threads can be monitored in real time without influencing the applications timing using a logic analyzer.

- **Intuitive Type Safe Application Programming Interface:** The AVIX API offers a number of features making it intuitive to use, leading to less coding and through its type safety allows for compile time detection of programming errors.

- **Accurate round robin scheduling**. Most RTOSes offer round robin scheduling. The cycle time for round robin threads is often based on a system wide timer, often having a period of something like 1ms. Using this relatively coarse period can easily lead to starvation of threads since the currently active thread monopolizes its thread priority level. This is a well known but undocumented feature of most RTOSes. The AVIX round robin implementation is based on measuring the time a thread has consumed with a resolution of ~1μs. As a result, AVIX round robin timing is not only very accurate but threads are guaranteed not to monopolize their thread priority level leading to a fair distribution of available processing power.

3 Glossary

The following terms and abbreviations are used throughout this document.

Term / Abbreviation	Explanation
API	Application Programming Interface. Generic term meaning the total of types, functions and constants that constitute the static programming interface used by applications to access AVIX functionality.
application priority	When no interrupt is active, the hardware is said to operate at application priority level which is the lowest possible priority level. This is the priority level application threads operate on and this is not an interrupt priority since no interrupt is active.
AVIX	Advanced Virtual Integrated eXecutive, the name for the RTOS
AVIX for PIC24-dsPIC	AVIX port for Microchip PIC24F, PIC24H, dsPIC30F and dsPIC33F families of microcontrollers.
AVIX for PIC32MX	AVIX port for Microchip PIC32MX families of microcontrollers.
AVIX for Cortex-M3	AVIX port for ARM Cortex-M3 families of microcontrollers.
CPU	Central Processing Unit, the processor heart of a microcontroller.
DIH	Deferred Interrupt Handler. A piece of code (function) that can be queued by an ISR and will be activated from the scheduler core before any of the regular threads will be (re)activated.
hardware platform	Term identifying a group of related microcontrollers (microcontroller family).
hardware system stack	Hardware implementation for exclusive use by ISR's.
interrupt latency	Time expiring between the (hardware) event triggering an interrupt and execution of the first instruction of the ISR.
interrupt priority	All hardware platforms targeted by AVIX deploy a nested priority model for interrupt handling where a higher priority interrupt can interrupt a lower priority interrupt. The interrupt priority is identified by a numeric value. Dependant of the specific hardware platform, a lower priority is identified by a lower numeric value or vice versa. Note that interrupt priority and the related numbering schema is hardware related and differs from thread priority which is an AVIX property.
ISR	Interrupt Service Routine. The piece of code (function) having a one-to-one relation to a (hardware) interrupt activated when the related (hardware) interrupt is asserted.
Kernel object	Term used to refer one of the AVIX offered objects like threads, mutexes, semaphores etc.
MCU	Micro Controller Unit. Name for single chip solution containing a CPU, peripherals and memory.
MIPS	Million Instructions Per Second. The figure to express the speed of a CPU.
RMA	Rate Monotonic Analysis, an established method to prove the timing correctness of a hard real time system.

Term / Abbreviation	Explanation
RTOS	Real Time Operating System. Generic name for products offering preemptive multithreaded functionality.
scheduler interrupt	Interrupt used internally by AVIX to activate the scheduler core
scheduler priority	The AVIX scheduler runs as an ISR with the lowest possible interrupt priority. This interrupt priority is called the scheduler priority.
Software system stack	AVIX software implementation of a system stack for exclusive use by ISR's in case the underlying hardware platform does not offer a hardware implementation of a system stack.
System stack	Stack for exclusive use by ISR's largely decreasing the memory consumption of an AVIX based application. Depending on the hardware platform, the system stack is either implemented in hardware (hardware system stack) or software (software system stack).
SFR	Special Function Register, the generic name given to device level controller registers.
thread priority	The priority given to a thread when creating it and used by the scheduler to determine which thread is the most important. The AVIX thread priority numbering schema is based on higher values denoting more important threads. Note that thread priority and the related numbering schema is software related and differs from interrupt priority which is a hardware platform property.
TAD	Thread Activation Diagram. A trace diagram as shown on a logic analyzer being a result of Thread Activity Tracing (TAT).
TAT	Thread Activity Tracing. A mechanism built into AVIX that enables the controller's digital I/O ports to be assigned to threads offering the possibility to runtime trace thread activity using a Logic Analyzer.
timer interrupt	Interrupt related to the AVIX hardware timer
timer priority	The hardware timer used by AVIX works interrupt driven. The ISR related to this interrupt has an interrupt priority one above the scheduler priority. This interrupt priority is called timer priority.

Table 1: Terms and Abbreviations

4 About this document

This document serves the purpose of both a User Guide and a Reference Guide. The reader of this document is supposed to be familiar with the basic principles of a Real Time Operating System (RTOS). This document is not meant as a tutorial for those principles and should not be considered as such.

Although greatest care is taken to ensure the content of this document is accurate and correct, no guarantee is given that the content is fully in accordance with both the static programming interface and the dynamic behavior of AVIX.

In case of a difference between the content of this document and the static interface and/or dynamic behavior of AVIX, the latter is leading.

This document is generic and does not target one of the specific AVIX Ports. Hardware platform specific topics are described in a Port Guide, a separate document available for each of the hardware platforms AVIX is available for.

► *Issues requiring special attention are marked with a blue arrow in the left margin and text being printed in italic, a format of which this is an example.*

4.1 Reader guidance

To find the information you are looking for as efficient as possible, an overview is given of the different sections in this document and their content.

The document contains the following major parts:

- **Chapter 5, Hardware Resource Usage**, describes the system resources used by AVIX and the consequences this has for application interrupt usage.
- **Chapter 6, User Guide,** contains the AVIX User Guide. This section of the document presents information how to create an AVIX based application. Described are the structural elements such an application is composed of and how these work together. Also a description is given of the different service categories offered by AVIX and how and when to use them.
- **Chapter 7, Reference Guide**, contains a detailed description of all functions offered by AVIX with their programming interface including parameters and return values.

5 Hardware Resource Usage

Internally AVIX uses one of the controller's interrupts, one of the controller's hardware timers and a small amount of RAM. This chapter provides global information on the use of these resources and the consequences this has for application interrupt usage.

Detailed information concerning the controller's resource usage can be found in the hardware platform specific Port Guide.

5.1 Interrupts

For activation of the AVIX scheduler core one of the controllers interrupt sources is used. This interrupt is called the scheduler interrupt. The scheduler interrupt is for exclusive use by AVIX and may not be used by the application.

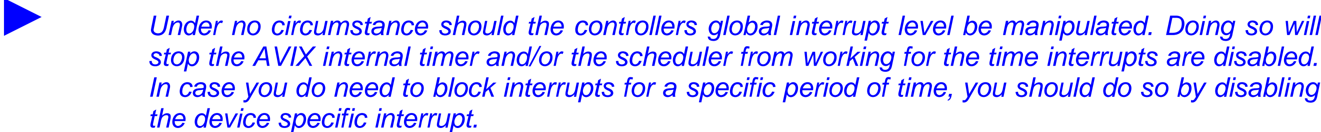
Depending on the specific AVIX Port, the scheduler interrupt is either fixed or user configurable. Detailed information can be found in the hardware platform specific Port Guide.

The scheduler interrupt operates at the lowest interrupt priority offered by the applicable controller family (scheduler interrupt).

The hardware timer used by AVIX also works interrupt driven. The applicable interrupt is called the timer interrupt. This timer interrupt runs at the scheduler priority plus one (timer interrupt).

More details about the scheduler interrupt and the timer interrupt and their relation to the priority models are found in §6.2.

AVIX assumes the controller to be initialized using prioritized nested interrupt handling. AVIX never disables interrupts and expects your application to do so neither. Under no circumstance should all interrupts be disabled since this prevents the AVIX scheduler and system timer from working. Because of the rich set of functions AVIX offers for use between interrupt handlers and threads it is very unlikely an AVIX based application ever needs to disable all interrupts. It is however allowed to disable specific interrupts by manipulating the device specific control registers



Under no circumstance should the controllers global interrupt level be manipulated. Doing so will stop the AVIX internal timer and/or the scheduler from working for the time interrupts are disabled. In case you do need to block interrupts for a specific period of time, you should do so by disabling the device specific interrupt.

5.2 Timer

The second hardware resource AVIX uses is a timer used as the time base for all AVIX timing functionality.

Since AVIX offers a number of software timers limited by the amount of memory only, using one of the hardware timers for AVIX will be no problem since most of the applications timing will be based on these software timers anyway.

5.3 Memory

For its internal bookkeeping, AVIX uses a small amount of RAM which is reserved. Furthermore, some RAM is needed for each of the AVIX kernel objects created by the application.

6 User Guide

This chapter presents an overview of the structure of AVIX, an AVIX based application, the services offered by AVIX and how to use these services to construct an application. Understanding the content of this section provides the necessary background to create an AVIX based application and provides understanding about when to use which service.

► *This chapter presents a large number of code samples. Code samples using controller specific code are based on the Microchip PIC24-dsPIC hardware platform. Note these samples may look different when used with other hardware platforms.*

The following sections are provided:

Chapter 6.1, Global Architecture, presents a global overview of AVIX, its components and the application components.

Chapter 6.2, Runtime Behavior and Responsibilities, presents an overview of the different software entities present in an AVIX application and who is responsible to provide these entities.

Chapter 6.3, Thread and Interrupt Priorities, presents an overview of the priority mechanism forming the core of preemptive scheduling and interrupt handling.

Chapter 6.4, Application Structural Building Blocks, presents the structural building blocks of an AVIX based application. These structural building blocks are Threads, DIH's and ISR's. These building blocks contain the code of every AVIX based application. This chapter also describes the AVIX system stack

Chapter 6.5, Application Support Functionality, presents supporting mechanisms offered by AVIX used in application development. These mechanisms are Object Management, Timeouts, Error Handling, Type Safety, Thread Activation Tracing and Power Management.

Chapter 6.6, Synchronization & Communication Services, presents the types of kernel objects offered by AVIX for use by an application to allow threads, DIH's and ISR's to communicate and synchronize.

Chapter 6.7, What to do in case of problems, presents a checklist to follow when an AVIX based application is not working as expected.

6.1 Global Architecture

A global overview of the architecture of AVIX and an AVIX based application is shown in Figure 1.

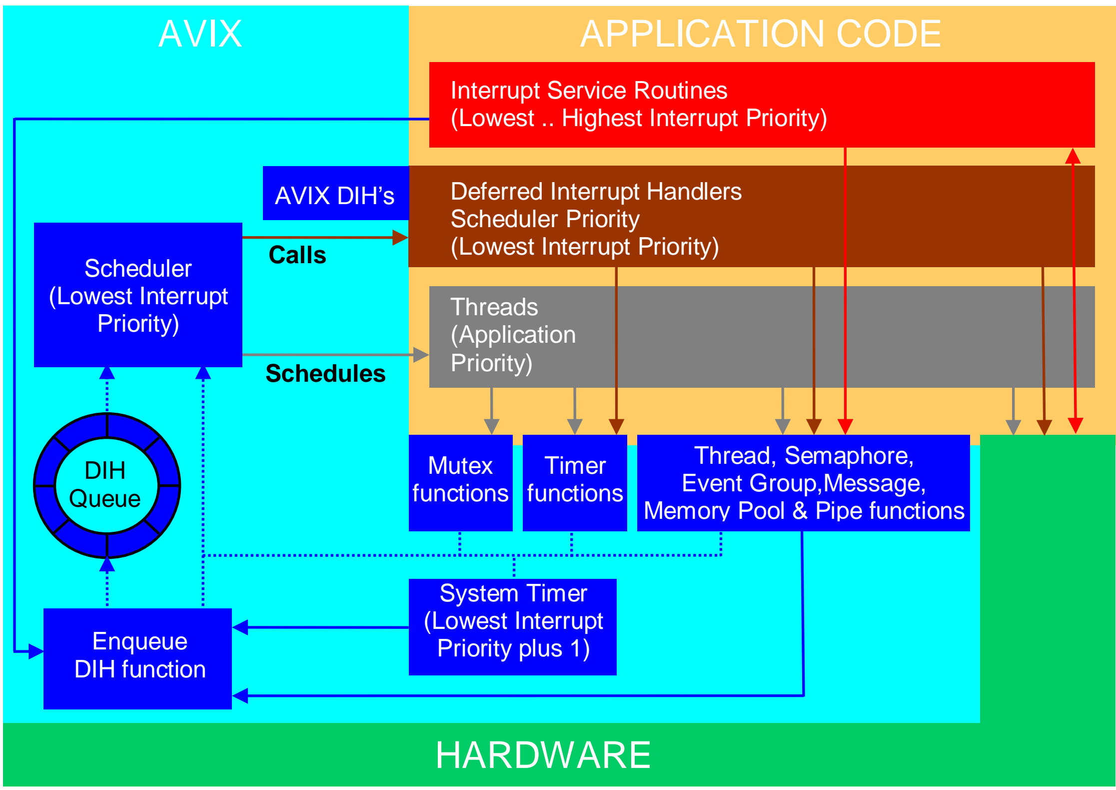


Figure 1: Global AVIX Architecture

Three main parts can be identified, application code, AVIX and the hardware. The application code contains three major parts, threads, Deferred Interrupt Handlers (DIH's) and Interrupt Service Routines (ISR's). These are all user supplied 'C' functions. Whether a function behaves as a thread, a DIH or an ISR depends on its definition and/or usage.

A function becomes a thread when it is registered with AVIX using AVIX provided function `avixThread_Create`.

A function becomes an ISR when declaring it as such using either AVIX provided macro `avixDeclareISR`, `avixDeclareISRShadow` or the regular hardware platform specific ISR declaration syntax.

A function becomes a DIH when enqueueing it using AVIX provided function `avixDIH_Queue`.

Although threads, DIH's and ISR's are all written as 'C' functions, none of these functions may be called directly; they are either called by AVIX (thread and DIH) or activated by a hardware event (ISR).

Threads contain most of the application code. Threads run under control of AVIX and can temporarily be halted in favor of another thread (preempted). Threads communicate with each other using AVIX provided services. Threads are dealt with in detail in §6.4.1.

ISR's are activated by a hardware event. ISR's are allowed to use a subset of the AVIX provided functions to communicate with threads. A characteristic of ISR's is that their activation is not synchronized with other functions. Interrupts occur at unpredictable moments. In order to

synchronize ISR's with threads and allow them to communicate with each other, a third type of function is offered which is a DIH. ISR's are dealt with in detail in §6.4.2.

A DIH is activated by AVIX when its address (function pointer) is present in a dedicated queue, the DIH queue. The address of a DIH is enqueued by function `avixDIH_Queue`, called from an ISR. Because DIH's are activated by AVIX, they run synchronous with thread activation and as such can interface the asynchronous ISR domain to the synchronous thread domain. DIH's run at scheduler priority, meaning they always take precedence over threads. DIH's are dealt with in detail in §6.4.3. Internally AVIX uses a small number of DIH's itself which is depicted in Figure 1.

The major parts of AVIX are the scheduler, the system timer, the DIH queue with its related functions and the function interface (API).

The scheduler is the core of AVIX. It runs at scheduler priority and has two responsibilities. When activated it will first call all DIH's present in the DIH queue. As a consequence, DIH's also run at scheduler priority. Next, in case a thread reschedule is required, the scheduler determines which thread must be activated. It will preserve the context of the currently active thread and restore the context of the new thread to activate. As a result this thread continues where it was formerly preempted. Before (re)activating a thread, priority is restored to the application priority level.

Activation of the scheduler happens when either…

- An ISR places a DIH in the DIH queue. Since ISR's occur asynchronous, this also leads to an asynchronous activation of the scheduler. The current running thread is preempted.

- A function is called leading to a rescheduling (current active thread is deactivated in favor of another thread). Since a function call is synchronous, this activation of the scheduler is synchronous also.

Finally AVIX contains a handler for the system timer. This is the central timing mechanism used for all timing related functionality. This handler too can cause a thread preemption for instance when a timer expires.

6.2 Runtime Behavior and Responsibilities

AVIX can be considered an add-on to a regular, non-AVIX, application. All system files required to create a non-AVIX application are required to be present having their standard responsibilities.

A non-AVIX application contains a function called `main` which is the entry point for the application. When using AVIX this is no longer the case. The `main` function is implemented by AVIX. User code may no longer provide a function called `main`.

An AVIX application must offer a function called `avixMain` instead. This function is required to be present and will be called by the AVIX initialization code. Like `main` in a non-AVIX application, `avixMain` contains the applications initialization code.

Typically the initialization code present in `avixMain` consists of two parts:

First, depending on the controller platform, some hardware specific initialization is executed. For instance when the controller speed requires certain code at application level to be executed, this is done in `avixMain`.

Second, since an AVIX application is based on multiple threads, these threads typically are created by `avixMain` using `avixThread_Create`.

The first thing AVIX does inside its private implementation of `main` is globally disable interrupts[3]. As a result, when `avixMain` is executing this is done with interrupts disabled. This is essential for AVIX to work correctly. Under no circumstance should any code executed in `avixMain` globally enable interrupts, neither directly nor indirectly. When calling third-party functions from `avixMain`, make sure these do not enable interrupts. Failure to meet this requirement will lead to an unstable application.

For AVIX to work correctly it is essential that during execution of `avixMain` interrupts remain disabled. Make sure not to enable interrupts during execution of this function, neither direct nor indirect by (third party) functions being called.

When `avixMain` returns, control is given back to AVIX. Interrupt handling will be enabled and from this moment on AVIX will never disable interrupts again. AVIX will start to schedule the threads registered in `avixMain`. The whole process described above is depicted in Figure 2.

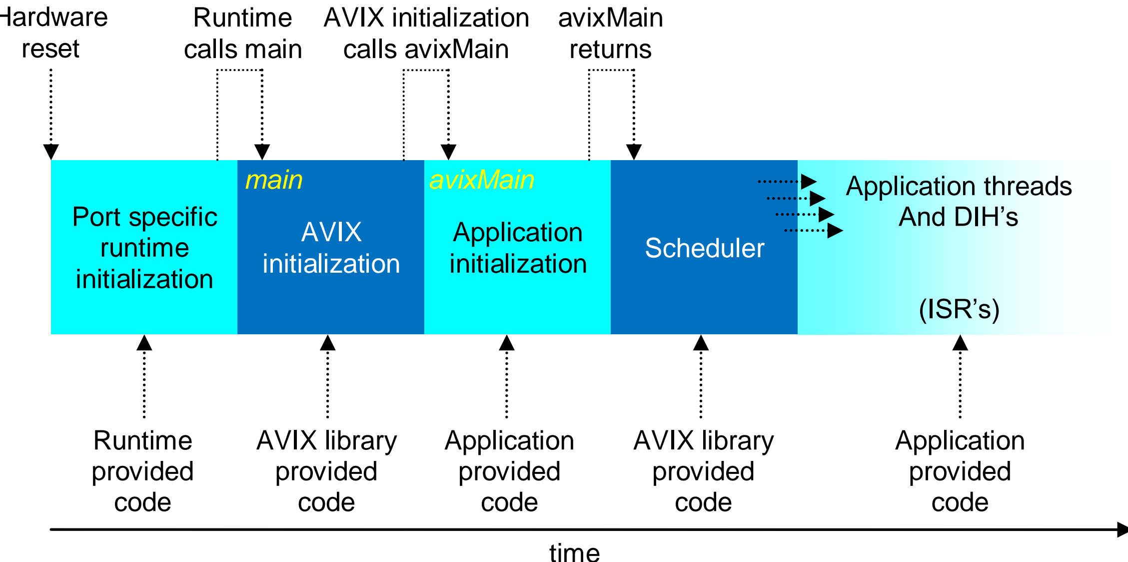


Figure 2: Runtime Behavior and Initialization

6.3 Thread and Interrupt Priorities

AVIX is a pre-emptive operating system meaning the currently active thread can be stopped in favor of another thread in case this other thread is more important than the currently active thread. To determine the importance of a thread AVIX uses a number based priority mechanism. On creation each thread is given a priority (thread priority) in the form of a numeric value. The lowest priority any application thread can have is one (1). The highest priority any application thread can have is user configurable. So the higher the numeric value the more important the thread. Internally AVIX has a thread also which by default has priority zero (0). This thread is the idle thread and since its priority is lower than any application thread, this idle thread will be active in case no application thread is able to run. The idle thread plays an important role in the AVIX power reduction mechanism.

The thread priority mechanism is fully implemented in software and it is AVIX that uses this priority mechanism to determine which thread is allowed to run.

The thread priority mechanism should not be confused with another type of priorities present in the underlying hardware platform, interrupt priorities. Interrupts are a hardware feature allowing

[3] AVIX is a True Zero Latency RTOS, meaning it never disables interrupts which is the case when the application is running. During initialization interrupts are disabled to prevent AVIX from starting too early.

postponing processing of the current software routine in favor of an ISR. The underlying hardware platforms offer a nested interrupt model where a more important interrupt can postpone a less important one. Here too the importance of the interrupt is determined using a numbering schema.

Some hardware platforms use a numbering schema where a *higher* number denotes a more important interrupt while other hardware platforms define a numbering schema where a *lower* number identifies a more important interrupt.

So the numbering schema used to denote the importance of an interrupt may differ from the numbering schema used to denote the importance of a thread. This might be confusing and therefore it is important to understand which interrupt priority numbering schema is used. Details can be found in the hardware specific Port Guide. To avoid confusion, in this document no use is made of any numbering schema. Instead a higher priority thread is said to be more important or have a higher priority. Likewise an interrupt is said to be more important or have a higher priority.

When no interrupt is active, the hardware is said to function at application priority. All threads run at application priority. When the hardware is operating at application priority, all interrupts come through, even the lowest priority interrupt.

AVIX uses two interrupts offered by the hardware. These are the scheduler interrupt and the timer interrupt. The scheduler is a part of AVIX implemented in the form of an ISR which runs at the lowest interrupt priority (scheduler priority). The AVIX hardware timer also uses an ISR which runs one interrupt priority level above the scheduler (timer priority). Since it is the AVIX scheduler that activates DIH’s, as a result DIH’s also run at scheduler priority.

It is important to realize the impact of this on the True Zero Latency feature. Because AVIX uses two interrupt priorities itself, application interrupts using these interrupt priorities are not guaranteed to be Zero Latency. An application interrupt using either the scheduler interrupt priority level or the timer interrupt priority level can be postponed since it occurs when one of these AVIX ISR’s is active. For this reason True Zero Latency is only available for ISR’s with an interrupt priority above the timer priority.

Figure 3 illustrates all described above

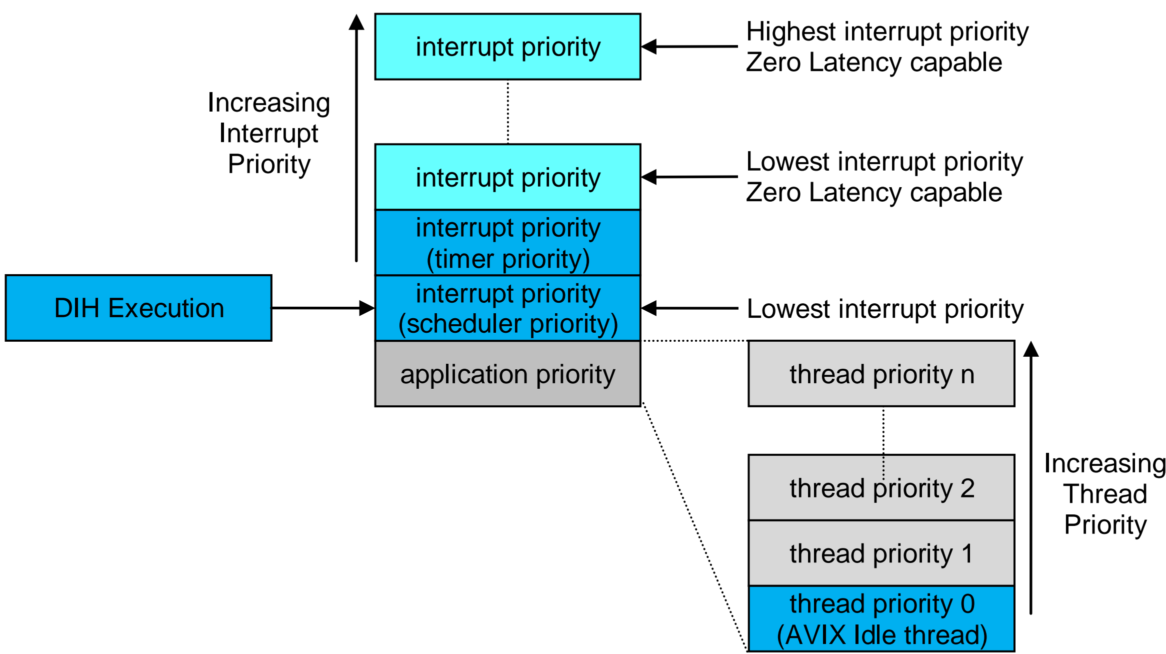


Figure 3: Thread and Interrupt Priorities

6.4 Application Structural Building Blocks

This chapter presents the structural building blocks offered by AVIX and how they work together to implement an application. These structural building blocks are Threads, ISR's and DIH's. Also described is how AVIX deals with stacks, understanding of which is important when constructing an application.

6.4.1 Threads

A thread is a self contained sub-program with well defined functionality preferably covering a single application topic. Threads are coded as functions and entirely activated under control of AVIX. Through the thread priority mechanism, the responsiveness and reaction speed of such a sub-program can be controlled. Threads are supposed to contain the majority of the application code.

The main object type offered by AVIX is a thread. Threads contain the vast majority of the application code and typically applications contain multiple threads.

A thread is a function running under control of AVIX. The address of a function to be activated as a thread is provided to AVIX as one of the parameters of function `avixThread_Create`.

This is comparable to a non-RTOS application which must contain a function called `main`. In this case, the runtime system expects the application to implement the `main` function. When started, the runtime will perform some initialization after which a call is made to `main` to execute the application code. This type of application can be considered single threaded; there is one 'thread of execution'.

An AVIX based application is multithreaded meaning there is a `main` for every thread. Since functions must have a unique name to prevent linker errors, the name of the actual thread function is not `main` but each thread function is given a unique user specified name instead.

Beside the thread functions, an AVIX based application must offer a function called `avixMain`. This function is the starting point of any AVIX based application and is called from AVIX initialization code. When `avixMain` executes, the AVIX scheduler is not yet active and no threads are executing. This function typically contains most of the applications initialization code, amongst which the creation of the required threads. When `avixMain` returns, AVIX takes control and starts to execute the functions that are registered with `avixThread_Create`. At any moment, only one of these threads is active and it is AVIX which decides which one, based on the thread priority and the state of the thread. This is explained in more detail later in this chapter.

Just like the `main` function in a single threaded application, the code of a thread typically contains an endless loop since the systems that AVIX is used for start executing when powered and continue to execute until powered down.

An example is shown in Code sample 1. Here two thread functions are shown (`threadFunc1` and `threadFunc2`) which are registered by passing their addresses to `avixThread_Create`.

```
 1 TAVIX_THREAD_REGULAR threadFunc1(void* p)      // Function thread 1
 2 {
 3   while(1)
 4   {
 5     LATA = ~LATA;                              // Microchip PIC24-dsPIC mechanism to
 6   }                                            // invert all bits of IO port A
 7 }
 8
 9 TAVIX_THREAD_REGULAR threadFunc2(void* p)      // Function thread 2
10 {
11   while(1)
12   {
13     LATD = ~LATD;                              // Microchip PIC24-dsPIC mechanism to
14   }                                            // invert all bits of IO port D
15
16 }
17
18 void avixMain(void)                            // Entry point of AVIX based application
19 {
20   AVIX_OBJECT_ID_DEFINE(tavixThreadId, tid);
21
22   // Create two threads
23   //
24   tid = avixThread_Create(NULL, threadFunc1, NULL, 1, 100, AVIX_THREAD_READY);
25   tid = avixThread_Create(NULL, threadFunc2, NULL, 1, 100, AVIX_THREAD_READY);
26 }
27 // Here avixMain returns and from this moment on, AVIX will
28 // activate the threads based on their thread priority and status.
```

Code sample 1: How to use threads with individual thread code

It is not required to write the code for every thread as a separate function. When multiple threads expose the same functionality, such threads can share code. When for instance an application controls two UARTS or two ports, it is likely the code for these threads is the same and they only need different data (e.g. the UART number or the I/O port address they control). To accomplish this, AVIX allows for a parameter to be passed to the thread function to allow the same code to behave different for the different threads it is used for based on the value of this parameter.

Code sample 2 shows an example of this mechanism. This sample implements the same functionality as Code sample 1 but now the two threads share their code. The difference is accomplished by passing a pointer to the I/O port the thread has to operate as parameter 3 to `avixThread_Create`. AVIX subsequently passes this parameter to the thread function.

```
 1 TAVIX_THREAD_REGULAR threadFunc1(void* p)      // Function for thread 1 and 2
 2 {
 3     volatile unsigned int* pLatchReg = p;      // void* parameter is pointer to LATCH
 4                                                // register. The LATCH register is the
 5     while(1)                                   // Microchip PIC24-dsPIC register used
 6     {                                          // to manipulate the bits of the I/O port
 7         *pLatchReg = ~(*pLatchReg);            // this latch register belongs to
 8     }
 9 }
10
11 void avixMain(void)
12 {
13   AVIX_OBJECT_ID_DEFINE(tavixThreadId, tid);
14
15   // Create two threads with the address of the applicable Microchip PIC24-dsPIC
16   // LATCH register as start parameter. The LATCH register is used to manipulate
17   // I/O port values.
18   //
19   tid = avixThread_Create(NULL, threadFunc1, (void*)&LATA, 1, 100, AVIX_THREAD_READY);
20   tid = avixThread_Create(NULL, threadFunc1, (void*)&LATD, 1, 100, AVIX_THREAD_READY);
21 }
22 // Here avixMain returns and from this moment on, AVIX will
23 // activate the threads based on their thread priority and status.
```

Code sample 2: How to use threads with shared thread code

After creating the desired threads, it is AVIX that decides when they are activated. At any given moment, only one of the threads is actually executing since the processor can only execute one instruction at a time. A thread is always in one of three possible states. These states are:

- **Suspended**: A thread is suspended when it voluntarily decides not to participate in the scheduling process. A thread can either be created suspended or enter the suspended state by calling `avixThread_Suspend`. A suspended thread can only leave this state when it is explicitly resumed by another thread or DIH (`avixThread_Resume`) or by an ISR (`avixThread_ResumeFromISR`).

- **Blocked**: A thread is said to be blocked when it is waiting for a resource that is not available. At any given moment multiple threads can be 'Blocked'. Threads can block on different resources or multiple threads can block on the same resource. A thread remains 'Blocked' until the resource is freed by another thread. More about this is explained later.

- **Ready**: A thread is 'Ready' when it is neither 'Blocked' nor 'Suspended'. Many threads can be 'Ready'. Of all threads being 'Ready', the one having the highest thread priority is 'Running'. When more than one thread shares the highest thread priority, the one at the front of the round robin list is selected as the 'Running' thread. Later in this chapter more information is provided about this round robin mechanism.

The most interesting of these states is the 'Blocked' state. A thread is in the 'Blocked' state when it is waiting for a non-available resource. Essential is that waiting for a non-available resource is not active waiting like in a 'for' loop. When a thread is said to be waiting for a resource to become available, it does not consume a single processor cycle and thus the processor can use its capacity to execute instructions of the thread that is 'Running'.

► *Multiple threads can be 'Blocked' on the same resource. Each resource contains a list holding the threads 'Blocked' on that resource. The position in these lists is based on the thread priority, the highest priority first. This guarantees that when the resource becomes available, the highest priority thread needing it will enter the 'Ready' state.*

So what are these resources a thread can be waiting for? Resources are other types of AVIX kernel objects allowing threads to communicate or synchronize, for example message queues. Message queues are used to pass messages between threads. A message is a small data structure that can be written by one thread and read by another. For the purpose of receiving messages, each thread contains a first-in-first-out message queue. A receiving thread indicates it wants to receive a message by calling AVIX function `avixMsgQThread_Receive`. When no message is available in the queue, this thread will enter the 'Blocked' state and another thread will start running while the receiving thread waits for a message to arrive. Once another thread has sent a message to the message queue, the thread owning this queue will enter the 'Ready' state again, since now a message is available and the original receive request can be honored.

Other resource types a thread can wait for are mutexes, semaphores, timers, event groups, pipes, memory pools and threads. These are all dealt with in the remainder of this document.

Thread Stack

A single threaded, non-RTOS application has a stack. This stack is used for local variables, storing return addresses when calling functions and saving the context when an ISR runs. When using multiple threads, each thread has its own stack which in principle is used for the same purposes. The size of a thread's stack is defined when creating the thread and is specified in the fifth parameter of `avixThread_Create`. The underlying hardware only has a single stack pointer register. When a thread is preempted, AVIX takes care the current stack pointer value is saved as part of the context of the preempted thread and the stack pointer of the thread that starts running is copied to the controllers stack pointer register. As a result, a thread's stack is 'local' to the thread and local variables can be accessed by the thread the stack belongs to only.

Why is the stack of a thread 'in principle' used for the same purposes as the stack of a single threaded application? Would the stack of a thread be used to save interrupt context, each of the stacks would have to reserve sufficient space to accommodate all possible interrupts. Reason is that the moment an interrupt occurs is unpredictable and therefore it is unknown which thread is active at that moment. To prevent this waste of valuable RAM, AVIX offers a single system stack for exclusive use by ISR's. This is a very important mechanism that can save a lot of RAM. The system stack is dealt with in §6.4.4.

Multi threading

A thread is a function executing on the CPU core of a microcontroller controlled by AVIX. AVIX allows the execution of multiple threads on a single CPU. All threads execute as if they completely own the entire CPU. In practice, threads are constantly preempted in favor of other threads. AVIX remembers where the 'Running' thread is preempted so when this thread is activated again, it can continue from there on. Two types of multi-threading exist, preemptive and cooperative and AVIX offers them both.

Preemptive multi-threading

Preemptive multi-threading is a type of multi-threading where thread execution can be halted at any possible moment and at any possible instruction and the CPU starts executing instructions of another thread. When this happens, a thread switch is said to have occurred and the halted thread is said to be preempted. A typical example of this is when a hardware interrupt occurs that changes the state of a thread from 'Blocked' to 'Ready'. When the thread priority of this thread is higher than the thread priority of the currently 'Running' thread, AVIX will ensure the thread having the higher thread priority to become the 'Running' thread. This form of multi-threading is asynchronous since it is triggered by an ISR. The moment the preemption occurs is not predictable since ISR's occur at unpredictable moments. Another possibility is the 'Running' thread frees a resource a thread with a higher thread priority is waiting for. The waiting thread will have its state changed to 'Ready' and since it has a higher thread priority than the first thread, it will become 'Running'. This type of preemption is synchronous since it is triggered by calling an AVIX function which happens at a deterministic moment in time.

Cooperative multi-threading

This form of multi-threading happens when a thread voluntarily indicates that it does no longer need to execute in favor of another thread at the same thread priority. A thread can do so by calling AVIX function `avixThread_Relinquish`.

Scheduling

Of all threads having the 'Ready' state, the thread having the highest thread priority will be 'Running'. The process used by AVIX to determine which of the threads having the 'Ready' state will actually be activated (Running) is called scheduling. The part of AVIX implementing this process is called the scheduler. Threads having the 'Blocked' state are ignored by the scheduler. The scheduler only considers threads having the 'Ready' state. AVIX offers two types of scheduling, thread priority based and round robin and these can be used together.

Priority based scheduling

The major AVIX scheduling mechanism is based on thread priorities. Priority based scheduling guarantees that timing constraints of the application are met, which is required in order for a system to be called real time. Every thread receives a thread priority when it is created. This is a numeric value from one (1) to a configurable upper limit where a higher numeric value denotes the thread to be more important. The number of different thread priorities that can be used is only limited by the amount of memory. Of all threads having the 'Ready' state, the scheduler will select the thread with the highest thread priority to be 'Running'. The scheduler is immediately activated when either the currently 'Running' thread enters the 'Blocked' state or a thread having a higher thread priority than the currently 'Running' thread enters the 'Ready' state. This means that every time a thread with a higher thread priority than the 'Running' thread enters the 'Ready' state, it

immediately becomes the 'Running' thread. This mechanism guarantees that more important threads, when they are able to run, will instantaneously be serviced in favor of less important threads

When an application is for instance responsible for controlling a motor and showing information on a display, one can imagine motor control to be more important than display control. By creating a separate thread for both pieces of functionality and giving the motor control thread the highest thread priority, it is guaranteed that every time the motor control thread needs attention, AVIX takes care of this by temporarily halting the display control thread in favor of the motor control thread.

► *The preemption mechanism implemented by AVIX works instantaneously. The moment a higher priority thread enters the 'Ready' state, AVIX activates the scheduler which within just a few microseconds takes care the new thread is 'Running'. For this reason AVIX may be called a true pre-emptive RTOS. Some competing products making the same claim behave different. These RTOSes only reschedule threads the moment a central system timer, most often having a period of one millisecond, expires. When a thread enters the 'Ready' state, only on the next expiration of the system timer that thread is made 'Running'. It shall be obvious this introduces unacceptable latency in the activation of threads, a problem AVIX threads are not confronted with.*

Round Robin scheduling

Multiple threads may share the same thread priority. The second schedule mechanism offered by AVIX deals with those threads. When multiple threads with the same thread priority are in the 'Ready' state, they can be considered to be present in a list where the thread at the head of the list will be 'Running', provided no thread with a higher thread priority has the 'Ready' state.

After the first thread in such a list has been executing for a configurable period of time, this thread is placed at the end of the list. The next thread in this list is at the head of the list now and becomes the 'Running' thread. By doing so, each of the threads at the same thread priority receives an equal time-slice, a reason why this mechanism also is called time-sharing. Round robin scheduling works alongside priority based scheduling. When during the active period of a round robin thread a thread with a higher thread priority enters the 'Ready' state, the scheduler will preempt the 'Running' round robin thread and make the higher priority thread the 'Running' thread. When the higher priority thread enters the 'Blocked' state again, the round robin thread at the head of the priority list continues with its time slice.

In contradiction with many competing products, AVIX offers a 'fair' round robin mechanism. A thread will always receive the processing time as defined by the value of the round robin time slice even when the thread is regularly preempted.

► *AVIX allows round robin scheduling to be disabled by configuring a round robin time slice of zero (0). Still threads can be created having the same thread priority. In this case when a thread does not block, it will be active forever without other threads at the same thread priority ever being activated. By calling `avixThread_Relinquish`, a thread places itself at the end of its priority list and activates the scheduler. This results in the next thread in the list starting to run.*

Priority inversion

A high priority thread can be blocked on a resource held by a lower priority thread. If this happens, the high priority thread can only continue when the low priority thread frees the resource. When the low priority thread in turn is preempted by a medium priority thread, it takes longer for the low priority thread to release the resource and as a result the high priority thread is delayed by a thread of less importance, the medium priority thread. This is called priority inversion and makes the timing of threads unpredictable.

Priority inversion, the problem

The background of priority inversion is illustrated in Figure 4.

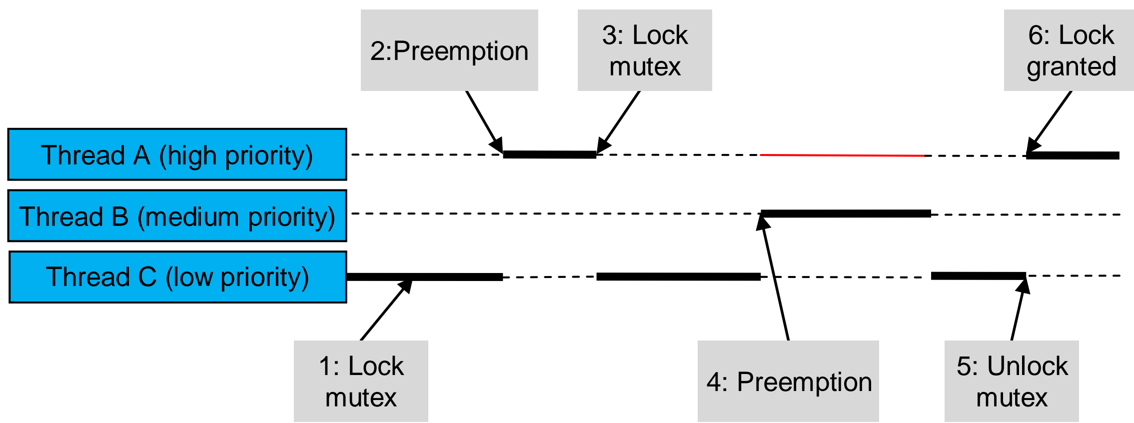


Figure 4: Priority inversion

Three threads are present with different priorities. Thread C is active and at moment 1 locks a mutex. At moment 2, for some reason preemption occurs and Thread A becomes active. At moment 3, this thread attempts to lock the mutex but since this is locked by Thread C, Thread A will block and Thread C may continue. Once Thread C unlocks the mutex (moment 5), Thread A will obtain the lock and may continue. In the meantime however Thread B is activated at moment 4. This delays Thread C and thereby the moment the lock on the mutex is released. This results in Thread A undergoing an unacceptable delay before it owns the lock on the mutex (the red line in Figure 4).

For an RTOS to deserve its name (Real Time), it must offer a solution for Priority Inversion. This is for instance a prerequisite when analyzing a systems timing requirements using Rate Monotonic Analysis (RMA). Not all competing products offer such a solution. The AVIX solution offers Priority Inheritance to the fullest possible extent in that it also solves recursive priority raising.

Priority inheritance, the solution

AVIX offers priority inheritance to solve the problems related to priority inversion. Priority inheritance changes the priority of a lower priority thread when a higher priority thread tries to lock a mutex owned by the lower priority thread. As a result, processing of the lower priority thread cannot be delayed and the owned mutex is unlocked at the earliest possible moment. The principle of priority inheritance is illustrated in Figure 5.

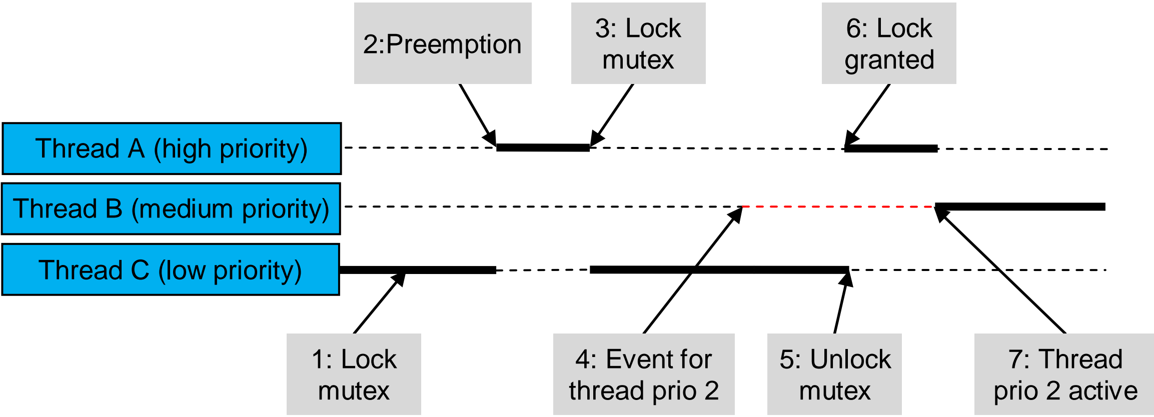


Figure 5: Priority inheritance

Three threads are present with different priorities. Thread C is active and at moment 1 locks a mutex. At moment 2, for some reason preemption occurs and Thread A becomes active. At moment 3, this thread attempts to lock the mutex but since this is locked by Thread C. Thread A will block and Thread C may continue.

The difference with the priority inversion scenario is however that at this moment (moment 3), AVIX raises the priority of Thread C to the priority of Thread A. So starting at moment 3, Thread C effectively runs at high priority.

When at moment 4 an event occurs making Thread B ready, this thread will not run since its priority is below the raised priority of Thread C. Thread C is allowed to continue and at moment 5 unlocks the mutex resulting in Thread A taking the ownership of the mutex. At this moment (moment 5), the priority of Thread C is restored to its original priority (low). Once Thead A is done, Thread B may start. In this scenario, the high priority thread (Thread A) does not undergo an unacceptable delay.

In contradiction with many competing products, AVIX offers nested priority inheritance which implies that the thread priority of a thread owning a mutex can be raised multiple times if threads with an even higher thread priority start to wait for the mutex.

Furthermore priority inheritance works for threads having the same thread priority which are scheduled in a round robin fashion. When a mutex is required by a thread at the head of the round robin list and it is owned by a thread at the same thread priority, this second thread is temporarily moved to the head of the list so it will be running until it has released the mutex. At that moment the requesting thread is moved to the head of the list again and continues.

► *AVIX offers a very advanced implementation of priority inheritance working for nested mutex requests and mutex requests among threads at the same thread priority. Not offering this can lead to unfair scheduling where one thread receives substantially more CPU time than others. The solution implemented by AVIX prevents this.*

6.4.2 Interrupt Handling

The purpose of Interrupt Handling is to react to events raised by hardware devices such that interaction with the device is performed with the least possible latency and overhead and without wasting precious CPU cycles on polling the device for its state

AVIX is based on the ‘Segmented Interrupt Architecture’. The advantage of this architecture over the so-called ‘Unified Architecture’ is that it allows interrupts to be used without ever disabling them. AVIX does not add a single cycle to the interrupt latency resulting in interrupt latency equaling that of the underlying hardware. The result is a predictable and deterministic system allowing very high interrupt rates.

► *AVIX never disables interrupts and therefore is called a ‘True Zero Latency RTOS[4]. AVIX based applications can predictably deal with very high interrupt rates without the risk interrupts are lost.*

Most competing products are based on the ‘Unified Architecture’. A characteristic of this architecture is that interrupts are frequently disabled having an unpredictable effect on interrupt latency and kind of ignores the effort spent by hardware vendors to decrease interrupt latency to the absolute minimum.

Implementing timing correct behavior is essential for any hard real time system. Methods exist to prove that a certain real time design implements the desired timing aspects, in other words, the application is timing correct. The most prominent of these methods is Rate Monotonic Analysis (RMA). For this method to be applicable, it is essential ISR’s are kept as short as possible.

Furthermore, when an ISR is active, the underlying hardware postpones other interrupts at the same or lower interrupt priority until the ISR returns. For this reason also, interrupt handling in real-time systems should be kept as short as possible.

The vast majority of processing in a real time system should be done in threads that are scheduled by the RTOS based on their thread priority. ISR’s should only implement the bare minimum processing to allow threads to communicate with the outside world. The actual data is processed by threads and thus it is required threads and ISR’s are able to communicate.

AVIX offers an extensive palette of functions to accomplish this communication, the most generic of which is use of a DIH. DIH’s are functions activated by the scheduler once they are ‘armed’ by an ISR. DIH’s may use almost all AVIX functions, allowing for tight integration between ISR’s and the applications threads[5]. Details about DIH‘s can be found in §6.4.3.

When using AVIX, ISR’s can even be shorter since often part of the ISR processing is postponed to a DIH. Since DIH’s can be interrupted, this mechanism in turn lowers the latency of ISR’s blocked because another ISR is active.

► *For a real time application to expose timing correct behavior, it is essential ISR’s are kept as short as possible. The segmented architecture AVIX is based on, allows for the shortest possible ISR’s, shorter than possible with any other architecture.*

Besides using DIH’s for ISR-thread integration, AVIX allows for a selected number of its functions to be used directly from an ISR without the necessity to write a DIH. These functions are clearly identified by their name which always ends in ‘`FromISR`’. This group of functions is dealt with later in this chapter.

[4] Interrupts will always have latency as dictated by the underlying hardware. With True Zero Latency is meant that AVIX does not increase the hardware latency.

[5] Competing products claiming Zero Latency often do not allow ISR’s to use RTOS services to communicate with threads. Such ISR’s are standalone, thereby neglecting one of the big advantages of applying an RTOS namely ISR-thread integration.

Interrupt Service Routines (ISR's)

Most RTOSes impose all kinds of rules and restrictions when writing ISR's. They do not allow the use of local variables. Specific functions MUST be called as first and last statement in the ISR which, when forgotten, leads to a crashing system. Use of very high speed interrupt handling based on hardware shadow registers[6] is not allowed etc.

With AVIX, use can be made of compiler generated ISR's, both regular and fast shadow register based. No special statements are required to be present in the ISR. ISR's may use local variables, call user functions and call a selected number of AVIX functions. ISR's can use a special AVIX function (`avixDIH_Queue`) to register a DIH. This DIH in turn may call many AVIX functions that may not be called directly by the ISR. Finally, interrupt nesting is fully supported. All this makes writing ISR's in an AVIX based application easier than with any other RTOS.

► *With AVIX writing ISR's is easier than with any other RTOS, there simply are no rules and restrictions.*

Code sample 3 shows a timer ISR which resumes a suspended thread[7].

```
 1  AVIX_OBJECT_ID_DEFINE(tavixThreadId, threadToResume);    // Thread id used by the DIH
 2
 3  // Interrupt Service Routine for timer 4 interrupt based on the Microchip PIC24-dsPIC
 4  // architecture.
 5  //
 6  void __attribute__((__interrupt__, no_auto_psv)) _T4Interrupt()
 7  {
 8    avixThread_ResumeFromISR(threadToResume);    // Place a DIH in a queue for processing
 9                                                 // following Interrupt Service Routines
10
11    IFS1bits.T4IF = 0;                           // Reset the interrupt flag based on the
12  }                                              // Microchip PIC24-dsPIC architecture
```

Code sample 3: How to declare an ISR using the compiler syntax

A disadvantage of combining an RTOS and interrupt handling is that stack usage can increase dramatically. Every thread running under control of the RTOS has its own private stack. Since interrupts occur at unpredictable moments, the related ISR's use the stack of the currently running thread and each of the thread stacks has to reserve enough space for the worst case interrupt nesting.

To counter this, AVIX offers a unique system stack implementation. Details of the system stack can be found in §6.4.4.

[6] Some of the hardware platforms AVIX is available for offer shadow register sets to speed interrupt handling. When using shadow registers, an ISR does not need to save and restore the registers of the interrupted code.

[7] In this code fragment, at the end of the Interrupt Service Routine the interrupt flag is cleared; It is very important this is an atomic operation in order to prevent other bits in the same register to be inadvertently changed leading to serious problems. Note the syntax to accomplish an atomic operation differs per hardware platform.

AVIX functions usable from an ISR

While the "Segmented Architecture" is superior to the "Unified Architecture" in terms of interrupt latency, still the "Unified Architecture" does have an advantage in that it allows more RTOS functions to be called directly from an ISR.

AVIX is based on the "Segmented Architecture" but its unique implementation allows many functions still to be called directly from an ISR. By doing so, AVIX offers the advantages of both architectures and this makes AVIX a hybrid RTOS.

The advantage of this approach is that programming becomes easier (writing a separate DIH is not necessary when using those functions).

To avoid errors, AVIX functions that may be used directly from an ISR are clearly identified by their name which always ends in `FromISR`.

It concerns the following functions:

- `avixEventGroup_ChangeFromISR`
- `avixMemPool_AllocateFromI`
- `avixMemPool_FreeFromISR`
- `avixMemPool_GetSizeBlockFromISR`
- `avixMsg_PutCharFromISR`
- `avixMsg_PutShortFromISR`
- `avixMsg_PutIntFromISR`
- `avixMsg_PutLongFromISR`
- `avixMsg_PutPtrFromISR`
- `avixMsg_PutKernelObjectIdFromISR`
- `avixMsg_PutIndirectFromISR`
- `avixMsg_AllocateFromISR`
- `avixMsgQThread_SendFromISR`
- `avixPipe_WriteFromISR`
- `avixPipe_ReadFromISR`
- `avixPipe_StopDeviceFromISR`
- `avixPower_SetModeFromISR`
- `avixPower_GetModeFromISR`
- `avixSemaphore_UnlockFromISR`
- `avixThread_ResumeFromISR`
- `avixTimer_StartFromISR`
- `avixTimer_ResumeFromISR`
- `avixTimer_StopFromISR`

Note that although these AVIX functions are available to be called directly from an ISR, still using an explicit DIH for some of them does have an advantage. More about this can be found in §6.4.3.

Interrupts and Pipes

Pipes are the most important AVIX mechanism for high speed data transfer. Details on pipes can be found in §6.6.6. Exchanging data between threads and ISR's is one of the more common tasks for most applications; AVIX allows pipes to be used from ISR's directly. Using Pipes from an ISR can be done by using one of two available AVIX functions.

A ISR receiving data from a peripheral must transfer this data to a thread. Doing so is accomplished by using `avixPipe_WriteFromISR` where the thread at the other end of the pipe uses one the regular functions `avixPipe_Read` or `avixPipe_ReadAsync`.

A thread wanting to send data to a peripheral writes to a pipe using `avixPipe_Write` or `avixPipe_WriteAsync`, where the ISR at the other end of the pipe uses `avixPipe_ReadFromISR` to read the data from the pipe and send to the hardware device.

An ISR must be able to control the hardware device it belongs to. When data is available to be sent, the hardware device must be enabled to generate interrupts and once the data is processed, interrupts must be disabled again. To accomplish this, pipes offer a callback mechanism. This callback mechanism is described in §6.6.6.

Pipes are typically used for communication purposes where for instance a UART ISR writes bytes to the pipe and the thread reading the pipe only enters the 'Ready' state when the desired number of bytes is present. This allows for interrupt based data transfer with the lowest possible system load.

Interrupts and Memory Pools

Although pipes offer a very efficient mechanism for data transfer between ISR's and threads, still the underlying pipe buffer is managed by AVIX and transferring data to or from a pipe implies more processing than would be the case when directly transferring data to or from a regular array variable.

Might performance requirements be such that an even faster mechanism is required, use can be made of AVIX memory blocks which are accessed through a regular 'C' pointer and therefore have the same efficiency as directly using a 'C' array to transfer the required data.

Memory blocks are allocated from memory pools. Details can be found in §6.6.7. For the above mentioned performance reason, AVIX allows memory pools to be used from ISR's allowing memory blocks to be allocated and freed directly from ISR's, a valuable service most competing products do not offer.

Two AVIX functions are offered to use memory pools directly from an ISR. Allocating a memory block from an ISR is accomplished using `avixMemPool_AllocateFromISR`. Freeing a memory block from an ISR is accomplished using `avixMemPool_FreeFromISR`.

Memory block usage from ISR's is typically used for communication purposes with very high performance requirements. An ISR that needs to transfer data to a thread can allocate a memory block. This memory block is filled on subsequent activations of the ISR and once the desired amount of data is written to the memory block, its id is communicated to the thread responsible for further processing.

For transferring the memory block id, different options exist. The id can be queued together with a DIH where the DIH will receive this id when activated. The DIH subsequently sends the id using a message to the interested thread. Alternatively use can be made of a pipe where the pipe is not used to contain the actual data now but instead it will contain the id of the memory block containing the data instead.

6.4.3 Deferred Interrupt Handlers (DIH's)

The purpose of DIH's is to allow information to be passed from ISR's to threads in case ISR's are not allowed to do so directly. As such, DIH's form an interface between ISR's and threads. Effectively DIH's allow ISR's to communicate with threads. DIH's should not contain essential application processing since they should be as short as possible.

Besides threads and ISR's, DIH's are the third active entity offered by AVIX. A DIH is a user written function allowing ISR's to communicate with threads. DIH's do not need to be created like threads. A DIH is just a 'C' function that is placed in a queue (DIH queue) by an ISR using AVIX function `avixDIH_Queue`.

Threads operate at application priority which is the lowest hardware priority. Threads can be interrupted by ISR's. The scheduler runs at the lowest interrupt priority thus in turn takes precedence over a thread. DIH's are called by the scheduler and therefore also run at the lowest interrupt priority. Therefore, when DIH's are present in the DIH queue, they will always be executed before a thread is (re)activated. In other words, when an ISR's enqueues a DIH, this DIH is guaranteed to be activated when the ISR is ready but before the interrupted thread is (re)activated.

Once a DIH is executed, it is removed from the DIH queue. In order to execute it again, it must be placed in the DIH queue again. Although DIH's are activated by the scheduler, this is done very fast consuming far less time than a thread switch.

The main application of DIH's is to allow ISR's to communicate with threads. This architecture has two advantages. First it helps to keep ISR's as short as possible. This is desirable since during the time an ISR is active, interrupts with the same or lower interrupt priority are blocked. Second it allows for interrupts never to be disabled (Zero Latency) which is one of the major advantages AVIX has over competing products.

DIH's have the characteristics of both threads and ISR's. Since DIH's are activated under control of AVIX, just like threads they are allowed to use almost all AVIX functions. Just like an ISR, once a DIH is started it will always run to completion. A DIH will not be pre-empted like a thread[8] (although it can be interrupted by an interrupt with an interrupt priority above the scheduler priority.

Just like ISR's, DIH's should be kept as short as possible in order to obtain a system with a predictable timing behavior which can for instance be proven with methods like Rate Monotonic Analysis (RMA). RMA asks for the vast majority of application processing to be taken care of by threads and thread interruption to last as short as possible. DIH's run at an interrupt priority above application priority and thus also interrupt the processing of threads.

Competing products based on the same architecture also offer DIH like mechanisms with exotic names like fibers or deferred procedure calls. Because of the speed of activation of these mechanisms, their use is advocated for extensive processing. This is principally wrong and should be prevented under all circumstances since it potentially hurts the timing correctness of the application.

► *For a real time application to expose timing correct behavior, it is essential thread interruption is kept as short as possible. Processing of a DIH also interrupts a thread which asks for DIH's also to be kept as short as possible. The major functionality of any real time system should be done in threads. ISR's and DIH's only exist to allow easy and fast communication with the outside world.*

Basic use of a DIH is illustrated in Code sample 4. Here all the ISR does is place a reference to a DIH (`myDIH`) in the DIH queue. Once the ISR is ready, AVIX will activate this DIH. When placing a DIH in the DIH queue, a `void*` parameter can be passed which in turn will be passed to the DIH

[8] Note the difference between pre-empted and interrupted. A DIH can be interrupted by a hardware interrupt.

as a parameter when activated. In this example, the ISR wants to resume a thread and the id of this thread is passed when the DIH is placed in the DIH queue. Utility macro's are provided to convert a kernel object id (like a thread id) to a `void*` and vice versa[9]

Note that in this sample, the ISR is declared using a macro. For controller families not exposing a hardware system stack, this macro will result in the ISR to use the software system stack leading to a reduction of RAM usage. Details about the (software) system stack can be found in §6.4.4.

```
 1 AVIX_OBJECT_ID_DEFINE(tavixThreadId, threadToResume);    // Thread id used by the ISR
 2
 3 void myDIH(void* arg)                                  // DIH, receiving a void* parameter
 4 {
 5   AVIX_OBJECT_ID_DEFINE(tavixThreadId, thread);
 6
 7   AVIX_TYPESAFE_FROM_VOID(thread, arg);                // Convert void* to thread id
 8
 9   avixThread_Resume(thread);                           // Resume the thread
10 }
11
12 // Interrupt Service Routine for timer 4 interrupt based on software system stack.
13 // Note that the parameters to macro avixDeclareISR are based on the Microchip
14 // PIC24-dsPIC architecture and can be different for other hardware platforms
15 //
16 avixDeclareISR(_T4Interrupt, no_auto_psv)
17 {
18   avixDIH_Queue                                        // Call to place DIH in DIH queue
19   (  myDIH,                                            // Pointer to DIH function
20      AVIX_TYPESAFE_TO_VOID(threadToResume));           // Thread id (passed as a void*)
21
22   IFS1bits.T4IF = 0;                                   // Reset interrupt flag based on the
23 }                                                      // PIC24-dsPIC architecture
```

Code sample 4: How to use a DIH

Although the previous code fragment is not complex, it is more complex than resuming a thread by calling `avixThread_ResumeFromISR` from the ISR directly. So a valid question is why to use the DIH based method when easier methods are available?

The answer is performance. By explicitly using the DIH based method, the ISR is very efficient and fast and the more time consuming part of the processing where actual AVIX internal data structures are manipulated is postponed to the DIH execution phase. So regardless the fact that a selection of AVIX functions may be called directly from an ISR, still the explicit DIH based variant can be used in case duration of ISR processing should be as short as possible.

▶ *Usage of DIH's is preferable above direct function calling from an ISR when execution of the ISR must be as short as possible and thus it is preferable to postpone part of its processing to a DIH.*

Another reason to use DIH's is that a DIH is allowed to directly call more AVIX functions than ISR's.

Using AVIX functions from DIH's

Since DIH's execute under control of AVIX, they are allowed to use most of the AVIX functions. This does however not mean it is useful for a DIH to do so. In general, threads, ISR's and DIH's send information to each other. This can be the data contained in a message or a count of a semaphore. When sending information, two roles are involved, sender and receiver. When sending a message, one thread is in the role of sender and the other in the role of receiver. When transferring data from an ISR to a thread the ISR is in the role of sender and the thread in the role of receiver.

[9] In this example the Deferred Interrupt Handler might just as well directly use the thread id contained in the global variable. By passing the id as a Deferred Interrupt Handler parameter, the same Deferred Interrupt Handler code can be used from multiple Interrupt Service Routines using different thread id's.

When using a DIH to create this connection between an ISR and a thread, based on the previous scenario, seen from the ISR, the DIH is in a receiving role where seen from the thread, the DIH is in a sending role. This is exactly what a DIH is meant for, form an intermediary between an ISR and a thread.

This approach already limits the AVIX functions that are useful or even possible to use from a DIH. Still a DIH is allowed to use AVIX functions that are not allowed to be used from an ISR. An example of this are the `…Get` functions used to obtain the id of an AVIX kernel object, for instance the id of a thread a DIH wants to communicate with.

When using this class of AVIX functions some precautions need to be taken care of. The following section offers information to give guidance related to this topic. First a general discussion is provided on how to deal with potentially blocking functions from a DIH. Second the different categories of services are dealt with to advice on their usage from a DIH.

How to deal with potentially blocking functions from a DIH

A large number of AVIX functions are potentially blocking, a mechanism actually forming the essence of any RTOS. When locking a semaphore, the calling thread will enter the 'Blocked' state (wait) when the semaphore is not available and continue when the semaphore becomes available again. This is the reason we speak about potentially blocking. Whether or not the function will block the calling thread depends on the availability of the resource and is transparent to the calling thread.

Quite some potentially blocking AVIX functions may be called from a DIH but a DIH cannot block! Just like an ISR, a DIH is a function that once started runs to completion.

Using a potentially blocking AVIX function from a thread will block the thread while the desired resource is not (yet) available. Optionally, a thread can use a time-out mechanism. Through this mechanism, a thread indicates that it is only willing to wait a specific time for the resource to become available. A thread can also indicate a time-out of zero (0), meaning it is not willing to wait at all. When calling a potentially blocking AVIX function from a thread with time-out zero (0), the function will never block and always return immediately. After this the thread must verify whether the resource was available or not.

The timeout mechanism is for exclusive use by threads and may not be used by DIH's. However, when calling a potentially blocking AVIX function from a DIH, the function automatically behaves as if it was called with a timeout value of zero (0), the only timeout value never leading to a AVIX function to Block, behavior which is exactly what is needed by a DIH. This behavior is implicit when calling a potentially blocking AVIX function from a DIH and under no circumstance a DIH may use the explicit timeout function `avixThread_ArmTimeOut`.

The subset of potentially blocking AVIX functions allowed to be used from a DIH return a resource id. In case the resource is not available, the returned resource id is invalid. In case the resource is available, the returned resource id is valid. To check for the validity of a resource id, make use of AVIX supplied macro `AVIX_OBJECT_ID_VALID`.

It shall be obvious the DIH must be prepared for the situation the resource is not available and act accordingly by using this macro.

Code sample 5 shows an example. Here an ISR needs to resume a thread and a DIH is used as an interface to accomplish this. In contradiction with the previous example, the ISR does not pass the thread id but the name of the thread which is a string. Now the DIH uses this string to obtain the required thread id[10]

```
 1  void myDIH(void* arg)                        // DIH, receiving a string as param
 2  {
 3    // The thread id to use from the DIH is 'cached' in a static variable
 4    // which is initialized to an invalid value.
 5    //
 6    static AVIX_OBJECT_ID_DEFINE(tavixThreadId, tid);
 7
 8    // If the local thread id is not yet valid, try to get the thread
 9    // id based on the name of the thread
10    //
11    if ( ! AVIX_OBJECT_ID_VALID(tid))
12    {
13      tid = avixThread_Get((char*)arg);        // Call a potentially blocking function
14    }
15
16    if (AVIX_OBJECT_ID_VALID(tid))             // Test the id to see if thread exists
17    {
18      avixThread_Resume(tid);                  // And only when existing, resume thread
19    }
20  }
21
22  // Interrupt Service Routine for timer 4 interrupt based on system stack. Note that
23  // the parameters to macro avixDeclareISR are based on the Microchip PIC24-dsPIC
24  // architecture and can be different for other hardware platforms
25  //
26  avixDeclareISR(_T4Interrupt, no_auto_psv)
27  {
28    avixDIH_Queue(myDIH, "TID");               // Queue a DIH to resume a thread passing
29                                               // the name of the thread as DIH param
30    IFS1bits.T4IF = 0;                         // Reset the interrupt flag based on the
31  }                                            // Microchip PIC24-dsPIC architecture
```

Code sample 5: How to use a potentially blocking AVIX function from a DIH

Of course a DIH can also use global variables containing id's of the desired resources but this introduces the risk this variable it is not yet initialized when the DIH uses it.

How to use different function categories from a DIH

The purpose of a DIH is to pass information or control from an ISR to a thread. AVIX offers many mechanisms to do so like setting a flag, sending a message, unlocking a semaphore and so on. This section gives a description of the different mechanisms available to be used by a DIH and the preconditions that should be taken into account.

▶ *All AVIX functions used by a DIH using a kernel object id must ensure the kernel object id is valid by either using macro `AVIX_OBJECT_ID_VALID` or by ensuring the kernel object id is valid through overall system design.*

- **Threads**: A DIH is allowed to resume a suspended thread.
- **Mutexes**: Under no circumstance may a DIH use a mutex. Mutexes are 'owned' by a thread and a DIH is no thread.
- **Semaphores**: A DIH may unlock a semaphore. For unlocking a semaphore there are no preconditions. To use a semaphore, the DIH must have access to its id. Details how to obtain such an id are presented before. *A DIH is not allowed to lock a semaphore.*

[10] Note the DIH will not call the …Get function every time it is activated. Once the service call succeeded, the thread id is 'remembered' in a static variable having a positive effect on performance.

- **Event groups**: A DIH may change the state of one or more event flags, either from a global event group or a thread local event group. The flags to change can be received from its related ISR through the DIH single word parameter. To use event flags, the DIH must have access to the id of a thread or an event group. Details how to obtain such an id are presented before. *A DIH is not allowed to wait for event flags.*

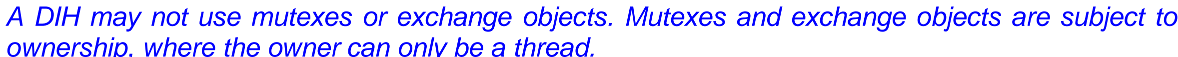
- **Timers**: A DIH may start of stop timers. Doing so, an ISR can influence periodical behavior of a thread waiting for this timer. To use a timer, the DIH must have access to its id. Details how to obtain such an id are presented before. *A DIH is not allowed to wait for a timer.*

- **Messages**: A DIH may allocate and send messages. These messages can for instance contain a single word value passed to the DIH from the ISR. Alternatively the message may contain the id of a memory block which is allocated and filled by the ISR and whose id is subsequently communicated to the DIH as its single word parameter. When sending a message, the DIH must always allocate a new one. *A DIH is not allowed to receive messages.*

- **Pipes**: Although a DIH may use pipes, it is not likely to do so. Pipes are typically used to transfer large amounts of data and since an ISR may directly use pipes it is more likely such functionality is implemented in the ISR directly.

- **Memory blocks**: Although a DIH may allocate/free memory blocks, it is not likely to be used. Memory blocks are typically used to transfer large amounts of data and since an ISR may directly allocate/free memory blocks it is more likely such functionality is implemented in the ISR directly. When the ISR needs to send a memory block to a thread through a message this may require usage of a DIH but in this case the DIH is just a pass-through entity.

- **Exchange services**: Under no circumstance may a DIH use an Exchange. Exchanges internally use mutexes which are subject to thread ownership.



A DIH may not use mutexes or exchange objects. Mutexes and exchange objects are subject to ownership, where the owner can only be a thread.

6.4.4 System Stack

The purpose of the system stack is to provide a dedicated stack for use by ISR's leading to a significant decrease of RAM usage and deterministic thread stack sizes.

In a non-RTOS application, the stack is used for storing local variables and function call return addresses. The stack is also implicitly used by ISR's. When using an RTOS, each thread has got its own private stack for the same purpose.

This does however introduce two potential issues.

First, each of these thread stacks has to preserve sufficient space for the ISR's to be allowed to save the interrupted context. Instead of one stack in a non-RTOS application this memory has to be reserved for every thread now, leading to a significant RAM consumption. Because the targeted hardware platforms allow interrupts to be nested, this RAM consumption can be very substantial.

Second a worst case situation exists where interrupt handlers are in turn interrupted by a higher priority interrupt for the entire interrupt priority range. This situation leads to the largest interrupt stack frame. Determining the worst case stack usage is often done by running the application for a long time, assuming this worst case situation if it can occur, will occur. When using multiple threads and therefore multiple stacks, the worst case situation has to occur once during the active time of every thread to determine the worst case stack usage. Chances this happens during the test are extremely low and therefore it is easy to misinterpret the observed stack usage and as a result under dimension the stack size.

To solve this, AVIX offers a unique system stack for exclusive use by ISR's. Some hardware platforms for which AVIX is available offer a hardware implementation of a system stack, in which case this mechanism is deployed by AVIX. For hardware platforms not offering such a feature, AVIX offers a software implementation. Bottom-line is AVIX offers a system stack for exclusive use by ISR's, regardless if the underlying hardware platform offers this or not.

► *For all supported hardware platforms, AVIX offers a global system stack for exclusive use by interrupts, regardless whether this feature is supported by hardware or not.*

Details concerning the decrease in RAM usage when using the software system stack are described in the platform specific Port Guide. Based on the hardware platform this decrease can be very substantial up to 10KB for a typical application.

For hardware platforms where the system stack is offered by the hardware itself, there is no choice whether to use this stack or not. Reason is there are only advantages namely large reductions in RAM usage.

When working with hardware platforms not supporting this, use of the software system stack is optional. Reason is that using the system stack will lead to a small increase of interrupt latency. By making the use of the software stack optional a trade-off can be made between higher speed (not using the software system stack) or lower RAM consumption (using the system stack). Whether or not the software system stack is used can be decided per interrupt. It is perfectly legal to mix ISR's using the software system stack with ISR's not using the software system stack in the same application.

AVIX is not unique in offering a software system stack. The AVIX implementation however differs a lot from most competing implementation. Most competing implementations only activate the software system stack once the initial part of the ISR has run. This implies that an ISR still places a substantial part of the interrupted context on the stack of the interrupted thread. The AVIX implementation kicks in at the very start of the ISR meaning the load on the thread stacks can be as low as zero (0).

►

The AVIX system stack mechanism saves a substantial amount of RAM and leads to a very deterministic system since stack usage is known from the start.

For ISR's to use the software system stack, all that needs to be done is declare the ISR using an AVIX supplied macro. Code sample 6 shows the same ISR as presented before in Code sample 3, now making use of the AVIX software system stack. The differences between Code sample 3 and Code sample 6 are shown in brown.

```
 1 AVIX_OBJECT_ID_DEFINE(tavixThreadId, threadToResume);    // Thread id used by the DIH
 2
 3 // Interrupt Service Routine for timer 4 interrupt based on system stack. Note that
 4 // the parameters to macro avixDeclareISR are based on the Microchip PIC24-dsPIC
 5 // architecture and can be different for other hardware platforms
 6 //
 7 avixDeclareISR(_T4Interrupt, no_auto_psv)
 8 {
 9   avixThread_ResumeFromISR(threadToResume);   // Place a DIH in a queue for processing
10                                               // following Interrupt Service Routines
11
12   IFS1bits.T4IF = 0;                          // Reset the interrupt flag based on the
13 }                                             // Microchip PIC24-dsPIC architecture
```

Code sample 6: How to declare an ISR using the AVIX software system stack syntax

6.5 Application Support Functionality

This chapter presents a number of supporting techniques and principles offered by AVIX to create an application. These are Kernel Object Management, Type Safety, Timeouts, Error Handling, Thread Activation Tracing and Power Management.

6.5.1 Kernel Object Management

The purpose of Kernel Object Management is to allow applications to obtain access to kernel objects in a synchronized way, such that a thread using a kernel object only does so when the kernel object really exists. This type safe mechanism prevents race conditions which might otherwise lead to system failure.

AVIX offers many different object types like threads, mutexes and so on. These objects are named kernel objects. Kernel objects have an AVIX internal structure containing the properties and state of the object. These structures are not 'visible' to the application. An application uses kernel object id's to identify these objects. A thread is identified by a thread id; a mutex is identified by a mutex id and so on. When using a function requiring access to a thread, the application uses the thread id of that thread as one of the parameters on the applicable function.

A kernel object id is a variable used by the application. Every type of kernel object id is identified by a unique 'C' type, defined in one of the AVIX header files. Most kernel object id's are obtained by creating a specific type of kernel object. When creating a thread using function `avixThread_Create`, a thread id is returned which from that moment on is used to identify the applicable thread. Other kernel object id's are obtained using a function specific to that type of kernel object. A memory block is not created using a create function but by allocating it from an existing memory pool using function `avixMemPool_Allocate`. This function returns a memory block id which from that moment on is used to identify the allocated memory block.

Table 2 provides an overview of the kernel object types offered by AVIX, the corresponding 'C' type used to define an instance of the applicable id and the major function used to create such a kernel object and obtain its id. Major, since the mentioned functions are not the only functions to obtain the id of a specific kernel object. More details on this are presented later in this chapter.

Kernel object	Kernel object id 'C' type	How to obtain the id
Thread	`tavixThreadId`	`avixThread_Create`
Mutex	`tavixMutexId`	`avixMutex_Create`
Semaphore	`tavixSemaphoreId`	`avixSemaphore_Create`
Event Group	`tavixEventId`	`avixEventGroup_Create`
Timer	`tavixTimerId`	`avixTimer_Create`
Pipe	`tavixPipeId`	`avixPipe_Create`
Memory pool	`tavixMemPoolId`	`avixMemPool_Create`
Memory block	`tavixMemBlockId`	`avixMemPool_Allocate`
Message	`tavixMsgId`	`avixMsg_Allocate`
Exchange	`tavixExchId`	`avixExch_Create`
Exchange connection	`tavixExchConnId`	`avixExch_Connect…`

Table 2: Kernel Object Id Types

Use of kernel object id's is type safe. A variable of type `tavixThreadId` cannot be passed to a function expecting a parameter of type `tavixMutexId`. When doing so, the compiler will issue an error. This error checking does not depend on a certain warning level to be used by the compiler. Error checking on mixing incompatible kernel objects id types cannot be disabled. Furthermore variables of these types cannot accidentally be given a value through a standard 'C' assignment which adds even more to the level of safety offered by AVIX.

▶ *AVIX offers type safety at compiler level. Programming errors are detected compile-time instead of runtime. This increases productivity and leads to more efficient code containing fewer errors.*

This mechanism is far more sophisticated than the mechanisms offered by competing products where just some pointer is used as object identification, potentially leading to many errors when changing its value or mixing incompatible types. Although the mechanism offered by AVIX is more sophisticated and offers the benefit of type safety, performance is not degraded compared to a more basic object identification method. The AVIX mechanism offers the exact same performance as a more basic model and overall performance is even better since AVIX functions receiving a kernel object id do not need to perform a runtime check to see whether the id refers the correct type of kernel object.

▶ *Kernel objects have a data structure which is entirely shielded from the application. Each kernel object is represented by an id which is used by the application. AVIX takes care an id is related to the correct kernel object. This leads to a simpler programming model introducing fewer errors.*

More details on the AVIX API type safety are found in section 'Kernel object id's and type safety'.

Special case, thread id converted to event group id

Kernel object id's of a certain type are used with functions expecting that type of id. Id's of type `tavixPipeId` are used with pipe services, id's of type `tavixSemaphoreId` are used with semaphore services. In principle there is a one-to-one relation between the type of kernel id and the class of services these id's are used with.

One exception to this rule exists. Event groups come in two types, global event groups and thread event groups. Global event groups are 'regular' event groups that are explicitly created and are shared between threads. A thread implicitly contains a local event group, a thread event group. When a thread exists, its local event group exists. A detailed description of this is found in §6.6.3.

Both types of event groups are manipulated using the `avixEventGroup…` class of functions. These functions receive an event group id of type `tavixEventId`. In principle, a thread event group is identified by the thread id which is of type `tavixThreadId`. Because of the AVIX type safety, it is not possible to pass a thread id to the class of event group functions.

To solve this, id's of type `tavixThreadId` offer a special attribute allowing such an id to be converted to an event group id which subsequently can be passed to the class of event group functions in order to manipulate the thread event group. This attribute is `.asEventId` and can be used on id's of type `tavixThreadId` only.

Code sample 7 shows an example of the usage of this attribute.

```
 1  AVIX_OBJECT_ID_DEFINE(tavixEventId,  eventGroupId);
 2  AVIX_OBJECT_ID_DEFINE(tavixThreadId, threadId    );
 3
 4  …;
 5
 6  // Set flag 0 in the global event group identified by id eventGroupId.
 7  //
 8  avixEventGroup_Change(eventGroupId, AVIX_EVENT_GROUP_SET, AVIX_EF(0));
 9
10  // Set flag 0 in the thread event group identified by id threadId. This thread id
11  // is converted to an event group id using attribute asEventId
12  //
13  avixEventGroup_Change(threadId.asEventId, AVIX_EVENT_GROUP_SET, AVIX_EF(0));
14
15  …;
```

Code sample 7: Usage of thread id's as event group id's

Kernel object naming and synchronization

Multiple threads use the same kernel objects. One thread locks a semaphore while another unlocks it, one thread sets event flags while another thread waits for certain event flags to become set. To accomplish this, threads using the same kernel object must have access to the same id since this is what kernel objects are identified by.

A very common approach among competing products is to store id's that must be shared by multiple threads in global variables. This does however impose a huge risk. How can a thread be sure the variable it is using refers a kernel object that already exists?

Using global variables to share kernel object id's can even lead to very subtle errors like illustrated in Code sample 8. Here two threads share a semaphore id. The semaphore is created by thread 1 which stores the semaphore id in global variable globalSem in order for thread 2 to use it.

The reason this works is because thread 1 is given a higher thread priority than thread 2. When thread 2 starts using the global variable, the semaphore is guaranteed to exist since thread 1 has already been active and created the semaphore.

Suppose now thread priorities are changed such that thread 2 receives a higher thread priority than thread 1. In this case thread 2 starts using the global variable while the semaphore is not yet created and the application will crash.

```
 1 AVIX_OBJECT_ID_DEFINE(tavixSemaphoreId, globalSem); // Global semaphore id
 2
 3 TAVIX_THREAD_REGULAR threadFunc1(void* p)           // Function thread 1
 4 {
 5   globalSem = avixSemaphore_Create                  // Create the semaphore on prio 2
 6                 (NULL, 0, AVIX_SEMAPHORE_UNLOCKED);
 7
 8   while(1)
 9   {
10     avixSemaphore_Unlock(globalSem);
11   }
12 }
13
14 TAVIX_THREAD_REGULAR threadFunc2(void* p)   // Function thread 2
15 {
16   while(1)
17   {
18     avixSemaphore_Lock(globalSem);          // Use semaphore created by thread1. Since
19   }                                         // this thread prio is lower than thread1
20 }                                           // semaphore is guaranteed to exist!
21
22 void avixMain(void)
23 {
24   avixThread_Create(NULL, threadFunc1, NULL, 2, 100, AVIX_THREAD_READY);
25
26   avixThread_Create(NULL, threadFunc2, NULL, 1, 100, AVIX_THREAD_READY);
27 }
```

Code sample 8: How to let threads share kernel object id's through a global variable

AVIX does not depend on global variables to be used for sharing kernel object id's between multiple threads. In principle use of global variables to share kernel object id's between threads is not advised.

AVIX allows most kernel objects to be given human readable names (strings) and use these names to synchronize access to the related kernel object. Every AVIX kernel object is created using a function like `avix<object type>_Create`. Threads are created using `avixThread_Create`, semaphores are created using `avixSemaphore_Create`. For each of the …`_Create` functions an accompanying …`_Get` function exists. Both the …`_Create` and the …`_Get` function receive a string parameter representing a user specified name of the kernel object. By using the same name, one thread can obtain access to a kernel object created by another thread.

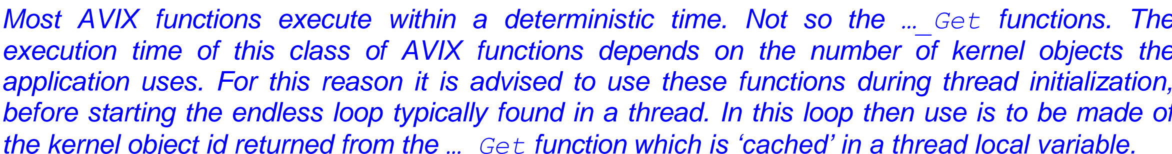
The power of the …`_Get` functions is that they are potentially blocking. When the kernel object with the specified name does not (yet) exits, the calling thread enters the 'Blocked' state. Once another thread creates the kernel object, 'Blocked' threads enter the 'Ready' state and are allowed to continue. Once the …`_Get` function returns, the designated kernel object is guaranteed to exist. The returned id is valid and can safely be used. Both the …`_Create` and the …`_Get` function return an id of the kernel object which can subsequently be used by the thread having access to that id.

Code sample 9 shows the same code fragment as before but instead of using a global variable to share the semaphore id, a name is used. Thread 1 creates the semaphore and using the same name, thread 2 obtains the semaphore id. The actual id's are contained in local variables now, living on the stacks of the using threads. Correct operation does no longer depend on the relative thread priorities. As long as the desired kernel object does not exist, the thread needing access to it simply enters the 'Blocked' state and AVIX takes care of the required synchronization.

► *Named kernel objects allow for correct synchronization of threads accessing shared kernel objects and for easier initialization of the application.*

```
 1 TAVIX_THREAD_REGULAR threadFunc1(void* p)         // Function thread 1
 2 {
 3   AVIX_OBJECT_ID_DEFINE(tavixSemaphoreId, sem);
 4
 5   sem = avixSemaphore_Create                      // Create the sempahore with a name
 6            ("sem", 0, AVIX_SEMAPHORE_UNLOCKED);
 7
 8   while(1)
 9   {
10     avixSemaphore_Unlock(sem);                    // Use the semaphore
11   }
12 }
13
14 TAVIX_THREAD_REGULAR threadFunc2(void* p)  // Function thread 2
15 {
16   AVIX_OBJECT_ID_DEFINE(tavixSemaphoreId, sem);
17
18   sem = avixSemaphore_Get("sem");           // Get the semaphore with the name, block if
19                                             // semaphore does not yet exist.
20   while(1)
21   {
22     avixSemaphore_Lock(sem);                // Use semaphore created by thread 1,
23   }                                         // semaphore is guaranteed to exist because
24 }                                           // of using the …_Get function!
25
26 void avixMain(void)
27 {
28   AVIX_OBJECT_ID_DEFINE(tavixThreadId, tid);
29
30   avixThread_Create(NULL, threadFunc1, NULL, 2, 100, AVIX_THREAD_READY);
31
32   avixThread_Create(NULL, threadFunc2, NULL, 1, 100, AVIX_THREAD_READY);
33 }
```

Code sample 9: How to let threads share kernel object id's through named objects



Most AVIX functions execute within a deterministic time. Not so the …`_Get` functions. The execution time of this class of AVIX functions depends on the number of kernel objects the application uses. For this reason it is advised to use these functions during thread initialization, before starting the endless loop typically found in a thread. In this loop then use is to be made of the kernel object id returned from the …`_Get` function which is 'cached' in a thread local variable.

Kernel object id naming

The name of a kernel object is determined when creating it by passing a pointer to a string as the first parameter to such a function.

Kernel object names must be unique in the first four characters. Although the string referred by the pointer may be as long as desired, only the first four characters are used. Furthermore, kernel object names must be system wide unique. When giving a thread and a mutex the same name, this is considered an error and reported as such.

Instead of just maintaining the pointer passed as the first parameter to the create function, AVIX copies the first four characters of the string to its internal bookkeeping. Would AVIX just maintain the pointer, the application would have to ensure the memory pointed to has the same content during the entire lifetime of the application. Although this is no problem when using a const string, it would be when using a RAM based variable for the string.

Suppose the name of the kernel object is composed in a char array variable and the thread would hold a pointer to this variable. After using the variable for a kernel object name, it could not be used for something else since its content is not allowed to change. By using the approach implemented by AVIX, after calling a create function, the application is allowed to reuse the array variable for a different purpose. However small the advantage, this again saves RAM.

A second reason exists to copy the identifying part of the name to the AVIX internal bookkeeping. String constants are stored in FLASH. Not all hardware platforms supported by AVIX have a linear address space but use some form of memory banks. Different threads may use a different memory bank and when just storing the pointer to the string, it is hard to guarantee both threads use the same memory bank containing the string. By copying the name to the AVIX internal bookkeeping, problems related to this hardware feature are prevented and usage is intuitive.

When using kernel object names, it is advised to define the name using a 'C' #define. Instead of directly using a string use is made of the defined symbol. Besides the obvious advantage this has related to code correctness, an additional advantage is that this symbol can have any desirable length increasing the readability of the code and encompassing the limitation imposed by the four character rule.

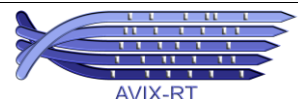
Kernel object id initialization

Whenever declaring a kernel object id (global or local), advised is to make use of AVIX supplied macro `AVIX_OBJECT_ID_DEFINE`. Using this macro, the kernel object id variable is guaranteed to be initialized to an invalid value. Only after assigning a valid kernel object id to the variable it becomes valid.

A second macro, `AVIX_OBJECT_ID_VALID`, can be used to test if a kernel object id is valid.

Use of global variables to store kernel object id's cannot always be avoided and especially in this case use of the mentioned macro's is strongly advised.

An example of this is when using kernel object id's from ISR's. ISR's cannot obtain a kernel object id by using a …`_Get` function since ISR's may not use potentially blocking calls. So the only way for an ISR to access a kernel object id is by declaring the applicable variable global.



Kernel object id's and type safety

The use of kernel object id's is type safe. Kernel object id's have a type that cannot be interchanged with other types. Type safety does not depend on compiler warning level settings. Whatever the warning level, the compiler checks the applicable parameters for being of the correct type and issues an error when this is not the case.

As a result, programming errors are detected during compilation and thus at the earliest possible moment. As a user of AVIX you will not notice anything when using this mechanism. Each API looks exactly like a regular API and is used just as easy. Only in case of a programming error you will be confronted with this mechanism but that is what it is intended for in the first place.

Most competing products perform a runtime check of function parameters. At its best this implies it takes more time to detect the error but there is also a chance the error is not detected at all since the parameter value is accepted by the function but still not the value intended by the programmer. This approach also leads to more elaborate RTOS code and thus a larger and slower application.

► *AVIX offers a real type safe programming interface leading to better error detection in a much earlier stage of the development process and smaller and thus faster code. This without a single cycle of overhead. Efficiency is equal or better compared to a non-type safe implementation.*

Type safe function parameters and return values

The most frequent use of the AVIX type safety mechanism is made when calling system functions. For most applications this will be the only use made of the mechanism. How does it work?

An example is shown in Code sample 10. Setting a timer is done through function `avixTimer_Set`. Of the parameters used with this function, the first is type safe. The timer id is of type `tavixTimerId`, so the parameter used here **must** be of that type. Any other type will result in a compilation error. This makes it impossible to accidentally pass for instance a thread id as the first parameter since this simply has another type.

Type safety is also used for a number of AVIX function result values. For instance function `avixThread_Get` returns a thread id which is of type `tavixThreadId`. This implies it must be assigned to a variable of this type. Assigning this return value to a variable of any other type results in a compiler error.

```
1 AVIX_OBJECT_ID_DEFINE(tavixTimerId,  timer);  // Type safe timer id
2 AVIX_OBJECT_ID_DEFINE(tavixThreadId, thread); // Type safe thread id
3
4 avixTimer_Set(timer, 1, AVIX_TIMER_CYCLIC);    // Correct function call
5 avixTimer_Set(thread, 1, AVIX_TIMER_CYCLIC);   // Type incorrect function call, error
6
7 timer  = avixTimer_Get("TH01");                // Correct function call
8 thread = avixTimer_Get("TH01");                // Type incorrect function call, error
```

Code sample 10: How to use type safe function parameters and return values

Explicit manipulation of type safe types

Besides using type safe values for AVIX function parameters, sometimes it can be required for an application to explicitly manipulate variables of these types. They can be used like any other type which is provided standard by the 'C' programming language with the added benefit that mixing types will result in a compilation error.

It is entirely legal and transparent to assign a variable of type `tavixThreadId` to another variable of the same type but assigning such a variable to a variable of type `tavixMutexId` results in a compilation error.

Because of the way the type safe mechanism is implemented comparing type safe variables with other variables of the same type is not directly possible. Macros are provided to accomplish this. These macros are `AVIX_TYPESAFE_EQ` for testing on equality and `AVIX_TYPESAFE_NEQ` for testing on inequality. Code sample 11 shows some samples with correct and incorrect assignments and comparisons.

```
 1  AVIX_OBJECT_ID_DEFINE(tavixEventId,   Ev1);
 2  AVIX_OBJECT_ID_DEFINE(tavixEventId,   Ev2);
 3  AVIX_OBJECT_ID_DEFINE(tavixMutexId,   Mu1);
 4
 5  // Assignments
 6  Ev1 = Ev2                               // Type correct variable assignment
 7  Ev1 = Mu1;                              // Type incorrect variable assignment
 8
 9  // Comparisons
10  //
11  if (Ev1 == Ev2)                         // Although the types match, the
12  {                                       // compiler will issue an error
13    …;                                    // because the type does not allow
14  }                                       // the == operator to be used
15
16  if (AVIX_TYPESAFE_EQ(Ev1, Ev2))         // Correct type safe comparison
17  {
18    …;
19  }
20
21  if (AVIX_TYPESAFE_EQ(Ev1, Mu1))         // Incorrect comparison resulting in a
22  {                                       // compiler error since Ev1 and Mu1 are
23    …;                                    // of incompatible types
24  }
```

Code sample 11: How to manipulate type safe variables and constants

Some AVIX functions just cannot make use of type safe parameter since they allow any possible type to be passed. A common approach in a ‘C’ program is to use `void*` for these situations.

Two examples for AVIX where this is used are parameter `threadParam` in `avixThread_Create` and parameter `arg` in `avixDIH_Queue`. These parameters are of type `void*`, any possible type can be passed as long as it has the same size as a `void` pointer. For these parameters it must also be possible to pass one of the AVIX type safe types but because of the implementation of these types this is not possible.

Since these types are safe, the compiler will not accept them to be implicitly cast to a `void*` either. To allow AVIX type safe types to be used in these situations, two macros are offered. One converts a type safe value to a `void*` and the other converts a `void*` to a type safe value. This is illustrated with an example shown in Code sample 12. Here a mutex is created in `avixMain` and its id is passed as the `threadParam` to a thread.

When using constructs like these, converting the `void*` back to the correct kernel object id type is the responsibility of the user.

```
1  TAVIX_THREAD_REGULAR threadFunc(void* arg)    // Thread function receiving mutex as arg
2  {
3    AVIX_OBJECT_ID_DEFINE(tavixMutexId, mutex);
4
5    mutex = AVIX_TYPESAFE_FROM_VOID(arg);       // Convert void* to id of mutex
6
7    while(1)
8    {
9      ..;
10   }
11 }
12
13 void avixMain(void)                           // Entry point of AVIX based application
14 {
15   AVIX_OBJECT_ID_DEFINE(tavixMutexId, mutex);
16
17   mutex = avixMutex_Create(NULL, AVIX_MUTEX_UNLOCKED);
18
19   avixThread_Create
20     (  NULL,
21        threadFunc,
22        AVIX_TYPESAFE_TO_VOID(mutex),          // Convert id of mutex to void*
23        1,
24        100,
25        AVIX_THREAD_READY );
26 }
```

Code sample 12: How to perform type safe value conversions

►

It is important to realize these conversion macros actually operate in a type unsafe manner with type safe variables. It is the programmers responsibility to make sure the type converted to a `void*` is converted back to the type it is meant to be.

6.5.2 Timeouts

The purpose of a time-out is to allow a thread waiting for a resource to cancel its wait in case the duration of the period identifies some unexpected situation must have occurred.

AVIX functions wanting to obtain a resource are potentially blocking. Potentially since when the required resource is available when the function is called, the thread will not enter the ‘Blocked’ state but continues without being preempted.

Only when the desired resource is not available the moment the function is called the calling thread will enter the ‘Blocked’ state and another thread will be activated.

Timeouts are used to specify a maximum period a thread is allowed to wait for a resource to become available. If the specified period expires without the resource becoming available, the thread will enter the ‘Ready’ state again. The thread will be removed from the resource wait queue and effectively the wait is canceled. Information is provided to the caller whether the function did succeed within the specified period so the program can react accordingly.

Competing products offer timeouts by adding a parameter to the API of each potentially blocking system function. This parameter specifies the period the calling thread is willing to wait. This leads to two potential problems.

In practice timeouts are only needed for a limited number of system functions, typically where the system communicates with the outside world. The core of any application should behave deterministic and when a timeout occurs, there is hardly anything that can be done. Actually a timeout occurring in the core of the application is caused by a programming error. If the API however does ask for a timeout to be specified on every function call, in a lot of places a default value is passed effectively disabling the timeout. This makes the code more complex, harder to understand and slower.

Second, the availability of the timeout parameter in each function easily tempts designers to use the timeout mechanism for application timing purposes. Since the principle of timeouts is however for safety purposes, this type of usage also obscures the purpose of the program and makes it harder to understand.

AVIX offers a unique approach by not adding a timeout parameter to the API of every potentially blocking function. Instead, when a timeout is needed, two extra functions are offered ‘encapsulating’ the AVIX function needing the timeout. This overcomes the mentioned problems and leads to more compact, faster and easier to maintain code.

Code sample 13 shows how to use the AVIX timeout mechanism. Before executing function `avixSemaphore_Get`, the time-out mechanism is armed using function `avixThread_ArmTimeOut`. After `avixSemaphore_Get` returns, the occurrence of a timeout can be tested using function `avixThread_TimeOutOccured`.

```
 1  AVIX_OBJECT_ID_DEFINE(tavixSemaphoreId, sem);
 2
 3  …;
 4  avixThread_ArmTimeOut (AVIX_DELAY_S(1));     // Arm a one second timeout
 5
 6  sem = avixSemaphore_Get(“sem”);              // Regular semaphore Get call
 7
 8  if (avixThread_TimeOutOccured())             // Test if timeout occured
 9  {
10    …;                                         // Timeout occurred.
11  }
```

Code sample 13: How to use a timeout based on the AVIX API functions

A special case is when the timeout period is set to zero(0). This identifies the thread is not willing to wait and thus never will. Under all circumstances the call to the potentially blocking AVIX function will return immediately. Effectively this implies the thread polls the resource for its availability.

To make the use of timeouts even easier, a special macro is supplied. This macro calls the desired AVIX function adding timeout functionality to it. Its use is shown in Code sample 14 which implements the same functionality as shown in Code sample 13.

```
1 AVIX_OBJECT_ID_DEFINE(tavixSemaphoreId, sem);
2
3 …;
4
5 if (AVIX_TIMEOUT(sem = avixSemaphore_Get(“SEM”), (AVIX_DELAY_S(1)))
6 {
7   …;                                       // Timeout occurred.
8 }
```

Code sample 14: How to use a timeout based on the AVIX helper macro

Note that timeouts may only be used from threads and not from ISR’s or DIH’s because ISR’s and DIH’s cannot block. For ISR’s a special category of functions exist. DIH’s are allowed to call most regular AVIX functions but these functions will always return immediately as if the function was called with time-out value zero(0).

A special case exists for Exchange callback functions. Exchange callback functions are running in the context of the thread that triggers these callbacks by writing to an Exchange. When calling a potential blocking function from an Exchange callback this function implicitly behaves as if the time-out is set to 0. Doing so, the behavior of the function is equal to the situation would the function be called from a DIH. Calling `avixThread_ArmTimeOut` from an Exchange callback function is not allowed and results in an error.

6.5.3 Error Handling

The purpose of error handling is to offer a centralized approach for catching errors leading to smaller and easier to comprehend application code.

AVIX implements an error handling approach which is as safe as that offered by competing products but leads to less complex user code resulting in fewer errors and smaller applications.

When an AVIX function call returns, it has succeeded. No error code is returned. In case the function detects an error, a call is made to a central error handling function which shuts down the application. In order to allow the system to react in a user specific manner, a user error handler can be registered through `avixError_SetHandler`. In this case, the AVIX supplied error handler will pass the error code to the user supplied error handler. Usage is shown in Code sample 15.

```
 1  // User supplied error handler
 2  //
 3  void userErrorHandler(tavixErrorCode errorCode)  // In case AVIX detects an error
 4  {                                                // this function is called since it
 5    if (errorCode == 10000)                        // is registered using function
 6    {                                              // avixError_SetHandler
 7      …;
 8    }
 9  }
10
11  void avixMain(void)
12  {
13    avixError_SetHandler(userErrorHandler);    // Register a user specific error handler
14    …;
```

Code sample 15: Install a user error handler

AVIX also allows user code to throw errors, resulting in the same mechanism being used. For this purpose function `avixError_Throw` is available. To make use of this mechanism easier, two assert macros are present that check an error code and optionally call `avixError_Throw` in a single line of code.

Macro `AVIX_ASSERT` is meant to be used during development. This macro is only active when compiling the application with symbol `AVIX_DEBUG` defined. When compiling the application without `AVIX_DEBUG` being defined, `AVIX_ASSERT` generates no code.

For errors that must be detected independent of the type of build (debug or release), macro `AVIX_ASSERT_ALWAYS` is present. This macro generates code, regardless of symbol `AVIX_DEBUG` being defined or not.

AVIX error codes are defined using type `tavixErrorCode`. For user error codes, predefined value `AVIXE_USER_ERROR_BASE` must be used as the base value. Doing so, user error codes always have a numeric value of `AVIXE_USER_ERROR_BASE` and higher.

► *The AVIX error handling approach offers the advantage of detecting errors without the problems that are caused by confronting the application code with a magnitude of error codes.*

Most competing products let the system functions they offer return a magnitude of different error codes. In order to build a robust system, the application code should check all these error values and act accordingly. In a correct working application, these errors however are not supposed to occur. Furthermore, if they occur, the application cannot deal with them since they indicate a situation that cannot be recovered from; after all they denote a programming error. AVIX allows all these situations to be dealt with at a central location resulting in more compact, easier to understand application code.

6.5.4 Thread Activation Tracing

The purpose of Thread Activation Tracing is to provide real-time non-intrusive insight in the activation of threads.

One of the more intriguing features of AVIX is Thread Activation Tracing. Thread Activation Tracing is a form of tracing that really helps you to get insight in the dynamic behavior of your application, solve timing issues and allow for easier application tuning.

Most RTOSes offer some form of tracing, typically implemented with a memory buffer being filled with all kinds of statistics. After the buffer is filled the content of this buffer can be analyzed off-line. This approach has two drawbacks. First the buffer is limited in size so it can take a lot of work to make sure the problem area in the application is present in this buffer. Second, this kind of tracing is intrusive. With tracing enabled, application behavior is different since the tracing changes the timing. This conflicts with the fact that the most occurring problems in RTOS based applications are timing related. It can be very hard or even impossible to find timing related problems because of the changed system timing. This often makes this type of tracing useless.

Thread Activation Tracing does not have these problems. First, no use is made of a memory buffer but tracing information is shown on a logic analyzer. Second, Thread Activation Tracing is non-intrusive. Application timing is the same whether the application is used with or without Thread Activation Tracing, not almost the same but exactly the same. As a result, Thread Activation Tracing really helps in finding timing issues since the information shown on the logic analyzer does represent the actual application behavior in real time. By using the trigger facilities of the logic analyzer it is possible to find application timings issues very fast.

Thread Activation Tracing works by assigning controller I/O pins to threads. The pin assigned to the running thread is pulled high while for all other threads the assigned pins are pulled low. Thread Activation Tracing works on the actual system running on real hardware but often it can also be used in the development environment when a simulated logic analyzer is present. Assigning an I/O pin to a thread is done using function `avixThread_SetTracePort`.

Thread Activation Tracing is unique in that it offers real time insight in the applications timing behavior while being non-intrusive. Application timing with or without tracing is equal.

Code sample 16 shows a sample of the usage of Thread Activation Tracing. Two low priority threads run forever and are scheduled round robin. A third high priority thread sleeps for a system tick. When the sleep period has elapsed, this thread preempts the then active low priority thread and starts sleeping again.

```
 1 TAVIX_THREAD_REGULAR threadHighPrio(void* p)       // High priority thread code
 2 {
 3   while(1)
 4   {
 5     avixThread_Sleep(1);                           // Sleep for a system tick
 6   }
 7 }
 8
 9 TAVIX_THREAD_REGULAR threadLowPrio(void* p)        // Low priority thread code
10 {
11   while(1){}
12 }
13
14 void avixMain(void)
15 {
16   AVIX_OBJECT_ID_DEFINE(tavixThreadId, tid);
17
18   tid = avixThread_Create(NULL, threadHighPrio, NULL, 2, 100, AVIX_THREAD_READY);
19   avixThread_SetTracePort(tid, AVIX_TRACE_PORT_D, 0);
20
21   tid = avixThread_Create(NULL, threadLowPrio,  NULL, 1, 100, AVIX_THREAD_READY);
22   avixThread_SetTracePort(tid, AVIX_TRACE_PORT_D, 1);
23
24   tid = avixThread_Create(NULL, threadLowPrio,  NULL, 1, 100, AVIX_THREAD_READY);
25   avixThread_SetTracePort(tid, AVIX_TRACE_PORT_D, 2);
26 }
```

Code sample 16: How to use Thread Activation Tracing

When connecting pin D0, D1 and D2 to a logic analyzer and running this application, the logic analyzer will show the pattern presented in Figure 6.

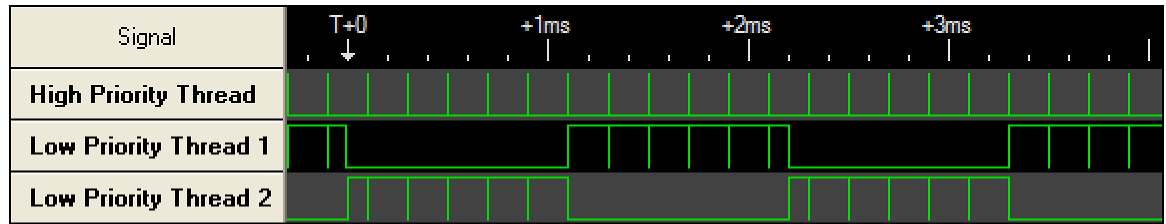


Figure 6: Sample Thread Activation Tracing diagram

This diagram shows exactly when and how long the threads are running and with the logic analyzers trigger and measure capabilities, unprecedented insight in the applications timing can be obtained.

Depending on the hardware platform being used, Thread Activation Tracing works 'out of the box' or some configuration settings might be needed. Consult the platform specific Port Guide for details.

Additional tuning facilities

Often more fine grained time measurements are required than just being able to know when a thread starts and stops as offered by basic Thread Activation Tracing. For this purpose function `avixThread_PulseTracePort` is offered. Using this function, the assigned I/O port will be pulled low for a very short time and this pulse is visible on the logic analyzer. Placing multiple calls to this function in a thread the time between pulses can be measured giving insight in the time spent by the code between those pulses.

Finally for tuning pipe performance function `avixPipe_SetHandlerTracePort` is offered. Using this function, an I/O port is assigned to a pipe handler making the activation of this handler visible on a logic analyzer. Using this trace facility performance ISR's using pipes can be tuned. More details are found in §6.6.6.

6.5.5 Power Management

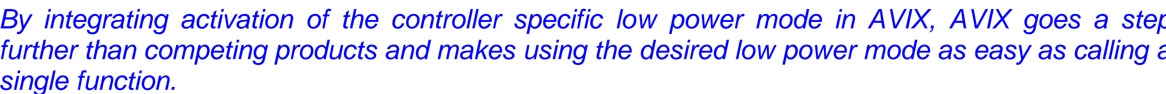
The purpose of Power Management is to let AVIX assist in reducing the power consumption of the microcontroller hardware.

Modern microcontrollers, also the ones that AVIX works with, have built in capabilities to reduce power consumption by activating a low power mode. These capabilities are especially important when the controller is used in battery powered equipment. Low power modes bring (parts of) the controller hardware to a halt at moments there is no need to execute application instructions. These capabilities are controlled by software and without an RTOS it is difficult if not impossible to structure the application such that these low power modes can effectively be used. To use a controller's low power mode, the application must be structured to work event driven where the absence of events might be an indication the controller can be switched to a low power mode.

The advantage of AVIX is that it both works event driven and contains a mechanism detecting the application does not need to execute instructions. This mechanism is the idle thread, a thread operating at a thread priority lower than any application thread and thus executing when no application thread, DIH or ISR is executing. It is safe to assume that when the idle thread is active, the controller can potentially be set to a low power mode and this is exactly what AVIX does.

Most competing products only allow for a callback to be registered for the RTOS to call when the idle thread is active. Activating the controller's low power mode is the responsibility of the application then.

AVIX goes a step further. Activating the desired low power mode is part of the AVIX code and all the application needs to do is tell AVIX what low power mode is desired by calling a single function.



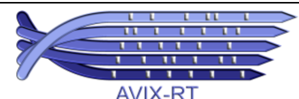
By integrating activation of the controller specific low power mode in AVIX, AVIX goes a step further than competing products and makes using the desired low power mode as easy as calling a single function.

Low Power Modes

Most controllers offer multiple low power modes. Selecting the desired low power mode is accomplished by either calling function `avixPower_SetMode` or `avixPower_SetModeFromISR` with one of the following three parameter values[11]

- `AVIX_POWER_REDUCTION_NONE`: No use is made of one of the controller's low power modes. When the idle thread is active it sits in an endless loop doing nothing. All devices remain operational and the application maintains the highest level of responsiveness to external events.
- `AVIX_POWER_REDUCTION_LOW`: When the idle thread is active, indicating no application code needs to execute, AVIX activates the controller's low power mode resulting in a low level of power reduction. This mode can have effect on operation of certain devices used by your application and the applications responsiveness to events.
- `AVIX_POWER_REDUCTION_HIGH`: When the idle thread is active, indicating no application code needs to execute, AVIX activates the controller's low power mode resulting in a high level of power reduction. This mode can have effect on operation of certain devices used by your application and the applications responsiveness to events.

[11] The actual selected low power mode is controller specific. Not all controllers offer the same capabilities. For this reason, when applying the AVIX power management it is important to also read the power management chapter in the hardware specific Port Guide.



Power management in your application

Besides functions `avixPower_SetMode` and `SetModeFromISR` used to select the desired power mode, AVIX offers additional interfaces to its integrated power management capabilities. All functions and interfaces together allow your application to deploy the controller's low power capabilities at a level varying from basic to advanced.

The most basic form of power management is accomplished by calling `avixPower_SetMode` during the initialization of your application and forget about it. AVIX takes care of activating the desired low power mode every time the idle thread is active.

For more advanced control, AVIX allows a callback function to be registered using function `avixPower_SetCallback`[12,13]. The registered callback function is called whenever the idle thread is active and AVIX is about to activate the currently selected power mode. The callback receives a parameter indicating the selected power mode which AVIX is about to activate. When the selected power mode is `AVIX_POWER_REDUCTION_NONE`, the idle thread sits in an endless loop meaning the callback is repeatedly called at a high frequency. Each of the other two power modes will result in the controller coming to a halt a few instructions after the callback has been activated as a result of the selected power mode becoming active. When the current power mode as reported to the callback is not the desired one the callback can select another power mode by calling `avixPower_SetMode`. The new power mode overrides the current power mode.

► *When setting a power mode from inside the callback the following is very important: Setting the power mode will 'reset' an AVIX internal state machine resulting in the callback being called again, now with the newly selected power mode. Only after the callback is allowed to finish without calling `avixPower_SetMode`, the desired power mode will be effectuated. For this reason, setting a power mode from within the callback should only be done when the power mode has to change. Unconditionally calling `avixPower_SetMode` every time the callback is activated will never effectuate the desired power mode.*

For the most advanced power management, this may however not be sufficient. AVIX works event driven meaning that between the moment the callback is called and the controller's power mode is activated, an interrupt may occur, leading to one or more threads being activated.

This might lead to a situation where the power mode about to be activated is no longer the desired one. So the interrupt handler or threads are allowed to select a different power mode, overriding the currently selected one.

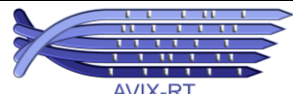
AVIX guarantees that the power mode which is selected up to the last machine instruction just before the power mode is activated is the one that will be used. When the current power mode is `AVIX_POWER_REDUCTION_HIGH` and the last machine cycle before this power mode is activated an event occurs leading to selection of a different power mode, this new power mode overrules `AVIX_POWER_REDUCTION_HIGH`.

► *Comparable mechanisms in competing products typically suffer from a race condition. Events occurring just before the selected power mode is activated are not able to change the currently selected power mode leading to the controller entering a power mode that does not comply with the actual application status. AVIX does not suffer from this race condition and activates the last power mode selected, regardless at which point in time this power mode is selected.*

---

[12] The callback is not called directly from the idle thread but runs as a DIH at scheduler priority. This is important since the functions used from the callback are those functions tagged as being allowed to be called from a DIH.

[13] When the application does no longer need the callback to be called, it can be disabled at any desired moment by calling `avixPower_SetCallback` with a `NULL` parameter value.



A typical advanced power management schema is implemented on a dedicated thread. This thread is in control of the entire application specific power reduction scenario. The thread is for instance able to enable and disable the backlight of an LCD either directly or by communicating with the thread responsible for controlling the LCD. This power management thread can be informed by the callback function about the power mode at controller level. For this purpose the callback function can use messages or event flags to inform this application specific power management thread.

Be aware that using power management can severely influence the timing of your application caused by significant hardware wake up delays. These delays are hardware platform specific.

Power management and the system timer

Depending on the controller's capabilities, certain low power modes may stop the main oscillator which is also used to clock the system timer. As a result, the system timer will come to a halt effectively halting all AVIX software timers. Some hardware platforms allow the system timer to be clocked from an external source. When doing so, when the main oscillator stops, the system timer will still continue. For this purpose a dedicated configuration parameter is present in the AVIX configuration file.

Note this capability is not offered by all hardware platforms. Consult the hardware platform specific Port Guide for details.

How to implement custom power management

Although AVIX power management offers functionality making it easy to exploit the controllers power management capabilities, there are situations you want to implement your own power management scenarios. For this purpose an application specific thread at the lowest priority can be used. Consider this like an application specific idle thread. When implementing a custom power management scenario, the AVIX mechanism should be disabled. This is simply accomplished by setting the AVIX power mode to `AVIX_POWER_REDUCTION_NONE` using function `avixPower_SetMode` which is the default mode also. By using this setting, the AVIX idle thread behaves like any other idle thread and just sits in an endless loop doing nothing. Still doing so you have to option to use the callback function in case you want to be informed the AVIX idle thread is active.

How effective is the reduction of energy consumption

The reduction in energy consumption accomplished depends entirely on the application and is not a parameter AVIX can influence. When used with an application having a high CPU load where the idle thread is not or hardly ever active, the reduction in energy consumption will be small. When on the other hand the application is not very active meaning the idle thread is active almost 100% of the time, the reduction in energy consumption can be substantial. Obtaining the highest possible reduction in energy consumption requires application specific tuning. AVIX is a facilitator for this.

6.6 Synchronization & Communication Services

This chapter presents the AVIX supplied synchronization and communication services. These services are used by threads, ISR's and DIH's. Each type of service uses a specific type of kernel object where the application code holds an id of the kernel objects.

A characteristic of kernel objects is that threads potentially Block on them. A thread can wait for a mutex to become unlocked. A thread can wait for a pipe to contain the required amount of data. A thread can wait for an Event Group to contain the desired combination of flags and so on.

When a thread has to wait for a kernel object to become available, it is placed in an internal wait list belonging to that kernel object.

►

Essential is that kernel object wait lists are ordered according the thread priority of the waiting threads. Suppose a thread is waiting for a semaphore. When during the time this thread is waiting a thread having a higher thread priority also starts' waiting for the same semaphore, this second thread is placed in the semaphore wait list ahead of the already waiting thread. Once the semaphore is unlocked, it is this second thread that will receive the semaphore and the first thread will remain waiting.

Understanding this mechanism is very important. Sometimes the implementation is very easy for instance when a thread waits for a message. A message queue is a private attribute of a thread. Each thread has its own message queue and a thread can only wait for the message queue it owns. So the wait list of a threads message queue will either be empty or contain the thread that owns the message queue. No other threads can be present in this wait list.

For pipes on the other hand the ordering of threads in the wait list can lead to interesting situations. Suppose a thread wants to write two blocks of data to a pipe but the pipe buffer only has empty space for one block. In this case one block will be written and the thread starts waiting for empty space to become available to write the second block. Before empty space becomes available, a second thread with a higher thread priority also wants to write data to the pipe. This second thread is placed in the pipes wait list ahead of the first thread and once empty space becomes available, it is this second thread that is allowed to write to the pipe. As a result the two blocks written by the first thread are not adjacent to each other in the pipe buffer but in between, the data blocks of the second thread are placed.

6.6.1 Mutex Services

The purpose of a mutex is to ensure access to a resource shared between threads cannot be preempted. Once a thread accesses the resource, only when all required operations are performed another thread is granted access to the resource.

A mutex is a synchronization object used to control access to a resource used by two or more threads (shared resource). In this context, a shared resource can be anything that is not 'private' to the thread that wants to use it. Examples of shared resources are global variables, I/O devices and so on. The name mutex comes from **mut**ual **ex**clusion which is what it is used for, using a mutex, a thread can prevent other threads from accessing a shared resource at the same moment.

A code sequence accessing the shared resource is called a critical section. A critical section is 'guarded' by a mutex. Before entering the critical section, a thread locks the mutex. When leaving the critical section it unlocks the mutex again. When executing code inside the critical section the thread can be preempted by another thread that also wants to enter the critical section. This second thread will also try to lock the mutex. A mutex can however be locked by one thread only and since the mutex is already locked by the first thread, the second thread will enter the 'Blocked' state until the first thread leaves the critical section by unlocking the mutex. As a result the critical section will always be executed from start to finish by a single thread. The actions within the critical section are executed atomically.

When a thread has locked a mutex, that thread is said to 'own' the mutex. This is very important since only the thread that locked the mutex is allowed to unlock it. When thread A has locked the mutex and while the lock exists thread B tries to unlock the mutex, this results in an error. Reason for this is that AVIX must be allowed to manipulate the priority of the 'owning' thread to solve the issue of priority inheritance which is dealt with later in this chapter.

► *Mutex ownership implies that the same thread that locked the mutex must unlock it. A mutex cannot be locked by one thread and unlocked by another. Neither can mutexes be used from ISR's nor DIH's since these are no threads and thus cannot own the mutex.*

Mutexes are intended to guard short critical sections with a deterministic behaviour. For this reason and for optimal performance, mutexes may not be used with a time-out.

AVIX mutexes allow for recursive lock calls. Once a thread owns the mutex, it can make additional lock calls and since the thread already owns the mutex, these additional locks succeed. This allows for recursive code without having to maintain some kind of bookkeeping that the mutex is already locked. This bookkeeping is maintained by the mutex itself. When using recursive locks, the thread must unlock the mutex as many times as it has locked it in order for the mutex to actually become unlocked.

A mutex is created using function `avixMutex_Create`. A lock is obtained with function `avixMutex_Lock` and the lock is freed again using `avixMutex_Unlock`.

Code sample 17 shows an example of mutex usage where a global variable is updated by two threads. Before performing the variable update, a mutex is locked and after the update the mutex is freed again. This ensures that while in this 'critical section', no other thread will be able to obtain a lock on the same mutex and the content of the global variable will not be corrupted.

```
 1  volatile int counter = 0;
 2
 3  TAVIX_THREAD_REGULAR thread1(void* p)            // Thread 1
 4  {
 5    AVIX_OBJECT_ID_DEFINE(tavixMutexId, mutex);
 6
 7    mutex = avixMutex_Get("mut");                  // Obtain access to the mutex
 8    while(1)                                       // (created elsewhere)
 9    {
10      avixMutex_Lock(mutex);                       // Enter the critical section
11      counter = counter + 1;                       // Increment global variable
12      avixMutex_Unlock(mutex);                     // Leave the critical section
13    }
14  }
15
16  TAVIX_THREAD_REGULAR thread2(void* p)            // Thread 2
17  {
18    AVIX_OBJECT_ID_DEFINE(tavixMutexId, mutex);
19    mutex = avixMutex_Get("mut");                  // Obtain access to the mutex
20    while(1)                                       // (created elsewhere)
21    {
22      avixMutex_Lock(mutex);                       // Enter the critical section
23      counter = counter + 1;                       // Increment global variable
24      avixMutex_Unlock(mutex);                     // Leave the critical section
25    }
26  }
```

Code sample 17: Using a mutex to guard access to shared resource

This example might seem artificial since only a global variable is incremented. No assumptions may however be made about the code generated by the ‘C’ compiler and it is likely that the increment of the variable is translated to three machine instructions, a Load, a Modify and a Write. When this sequence is not protected against preemption, the content of the variable may become corrupted when updating it from multiple threads.

Using mutexes directly from a threads code might easily lead to a complex application structure which is hard to oversee and maintain in case many mutexes are used. Looking to Code sample 17, when a thread updates the global variable without first locking the mutex, things still go wrong. The more threads that are allowed to manipulate the global variable, the more complex the code structure becomes and the easier it is to make a mistake.

A solution to this is to combine the code accessing the shared resource and the mutex lock and unlock functions in a single function. This function can be used from multiple threads, implicitly offering atomic access to the resource managed from that function. One can no longer forget to lock the mutex since locking is part of the function. This structure is shown in Code sample 18.

► *In general, a good rule is to prevent the use of global variables as much as possible. This is true in a single threaded application but even more in a multi-threaded application. A high level of modularity where the different software components have a local, well defined responsibility offered through a well defined interface is preferred and may prevent many problems. AVIX offers services that support this like Exchange objects that internally use mutexes to guard concurrent access. Detailed information on Exchange objects and how these facilitate a high level of modularity can be found in §6.6.8.*

```
 1 volatile int counter = 0;
 2 AVIX_OBJECT_ID_DEFINE(tavixMutexId, mutex);       // Must be created elsewhere
 3
 4 void atomicUpdateCounter(void)
 5 {
 6   avixMutex_Lock(mutex);                          // Enter the critical section
 7   counter = counter + 1;                          // Increment global variable
 8   avixMutex_Unlock(mutex);                        // Leave the critical section
 9 }
10
11 TAVIX_THREAD_REGULAR thread1(void* p)             // Thread 1
12 {
13   while(1)
14   {
15     atomicUpdateCounter();
16   }
17 }
18
19 TAVIX_THREAD_REGULAR thread2(void* p)             // Thread 2
20 {
21   while(1)
22   {
23     atomicUpdateCounter();
24   }
25 }
```

Code sample 18: Using a mutex to guard access to shared resource

Priority Inheritance

The AVIX mutex mechanism supports the principle of priority inheritance to solve the problem of priority inversion. Priority inversion is the name used for unacceptable delays in the activation of a high priority thread caused by a lower priority thread. A detailed description can be found in §6.4.1.

For an RTOS to deserve its name (Real Time), it must offer a solution for Priority Inversion. This is for instance a prerequisite when analyzing a systems timing requirements using Rate Monotonic Analysis (RMA). Not all competing products offer such a solution. The AVIX solution offers Priority Inheritance to the fullest possible extent in that it also solves recursive priority raising.

What differentiates a Mutex from a Semaphore?

In principle a mutex looks a lot like a semaphore with an initial count of one (1). In literature a mutex is sometimes even called a binary semaphore. While this may be correct for some RTOSes, for AVIX supplied mutexes and semaphores it is not and it is strongly advised not to call a mutex a binary semaphore.

Mutexes and semaphores are not the same and they have quite distinct properties and application areas.

Semaphores behave different with recursive calls. A semaphore with an initial value of one (1) being locked twice by the same thread will result in this thread being blocked while for a mutex this will result in an incremented internal nest count and the thread continuing since it already owns the mutex.

Another difference is that mutexes have ‘ownership’. Only the thread that owns (successfully locked) the mutex is allowed to unlock it again. AVIX ‘knows’ which thread owns a mutex and thus is able to offer Priority Inheritance. Semaphores have no ownership. One thread can lock a semaphore while another thread can unlock it.

6.6.2 Semaphore Services

The purpose of a semaphore is to control access to resources with a bound number of instances from multiple threads. When all resource instances are used by a thread, a semaphore ensures another thread cannot use one of the resources until a using thread indicates to need the resource no longer.

A semaphore is a synchronization object maintaining a count conceptually representing a number of permits. A thread requiring a permit attempts to lock the semaphore. When the semaphore has a permit available (internal count greater than zero), it decrements this count and the thread is allowed to continue. When no permit is available (count is zero), the thread enters the 'Blocked' state until another thread increments the count by unlocking the semaphore.

A semaphore is created using function `avixSemaphore_Create`. A lock is obtained with function `avixSemaphore_Lock` and the lock is freed again using `avixSemaphore_Unlock`.

Semaphores are often used to control access to a shared resource or control access to a resource that contains multiple elements.

An example is a mechanism offering two UART based communication channels where these UARTS must be usable from multiple threads. A sample implementation of such a mechanism is shown in Code sample 19. Here two functions are shown, one to claim a UART and one to free it again. These functions are intended to be used from threads wanting to use a UART.

In this case a semaphore is used with an initial value equal to the number of UARTS (two). When a thread wants to use a UART, it tries to lock the semaphore. This will succeed as long as the semaphore count is greater than zero, implying one or two UARTS are available. When all UARTS are in use, the lock will not succeed and the thread will enter the 'Blocked' state. In this scenario, there is also some bookkeeping data to determine which of the UARTs is available. Once the thread has obtained a lock on the semaphore, it will query this bookkeeping to obtain the UART it may use. In the sample, the bookkeeping is just a basic array with Booleans. In a real implementation, the bookkeeping will probably contain more information.

Once the thread is ready using the UART it will update the bookkeeping data identifying the UART is no longer needed and unlock the semaphore.

An interesting aspect about this example is that it shows the use of both semaphores and mutexes.

The semaphore is used to guard the permit to use a UART. Once a permit is handed over to a thread (the semaphore lock succeeds), the thread uses the UART for whatever period is needed and the semaphore remains locked all the time.

When a thread obtains a permit by locking the semaphore, the bookkeeping of the status of the UARTS must be updated. This is a short, deterministic operation that may not be preempted in order for the bookkeeping data to remain consistent. This asks for a critical section for which purpose a mutex is used. The mutex is only locked for a very short time, the time it takes to update the UART bookkeeping.

```
 1  struct                                              // Management structure for managing
 2  {                                                   // 2 UARTS. The semaphore is used
 3    tavixSemaphoreId sem;                             // to wait when no UART is available
 4    tavixMutexId     mut;                             // the mutex is used to guard access
 5    bool             uartFree[2];                     // to the bool array. The bool array
 6  } sManage;                                          // identifies which UART is free.
 7
 8  // Initialize the management struct, initially both UARTS are free so both
 9  // booleans are TRUE.
10  //
11  void initialize(void)
12  {
13    sManage.sem = avixSemaphore_Create(NULL, 2, AVIX_SEMAPHORE_UNLOCKED);
14    sManage.mut = avixMutex_Create(NULL, AVIX_MUTEX_UNLOCKED);
15    sManage.uartFree[0] = TRUE;
16    sManage.uartFree[1] = TRUE;
17  }
18
19  // Obtain index of available UART and block when no UART available.
20  //
21  int getAvailableUart(void)
22  {
23    int i;
24
25    avixSemaphore_Lock(sManage.sem);                  // Lock semaphore, block when no
26                                                      // UART is free (sem count is 0)
27    avixMutex_Lock(sManage.mut);                      // Lock access to the array
28    if (sManage.uartFree[0] == TRUE)                  // using a mutex and find the first
29    {                                                 // free element (critical section)
30      i = 0;
31    }
32    else
33    {
34      i = 1;
35    }
36    sManage.uartFree[i] = FALSE;                      // Set flag for used element FALSE
37    avixMutex_Unlock(sManage.mut);                    // End critical section
38    return i;
39  }
40
41  // Free an UART so a blocked thread can continue
42  //
43  void freeUart(int nr)
44  {
45    avixMutex_Lock(sManage.mut);                      // Inside a mutex protected
46    sManage.uartFree[nr] = TRUE;                      // critical section set freed
47    avixMutex_Unlock(sManage.mut);                    // UART flag TRUE again.
48
49    avixSemaphore_Unlock(sManage.sem);                // Increment available count in sem
50  }                                                   // allowing waiting threads to cont.
```

Code sample 19: How to use a semaphore to control resources with multiple instances

What differentiates a Semaphore from a Mutex?

In principle a semaphore with an initial count of one (1) looks a lot like a mutex. In literature a mutex is sometimes even called a binary semaphore. While this may be correct for some RTOSes, for AVIX supplied semaphores and mutexes it is not and it is strongly advised not to call a mutex a binary semaphore.

Semaphores and mutexes are not the same they have quite distinct properties and application.

A difference is that semaphores do not offer recursive calls like mutexes do. A semaphore with an initial value of one (1) being locked twice by the same thread will result in this thread being blocked while for a mutex this will result in an incremented internal nest count and the thread continuing since it already owns the mutex.

Another difference is that semaphores do not have 'ownership'. One thread can lock a semaphore while another can unlock it. As a result, semaphores do not offer Priority Inheritance and usage is not restricted to threads.

► *Semaphores may be unlocked from threads, ISR's and (DIH's). Locking a semaphore from an ISR or a DIH is of no use since an ISR and a DIH cannot block.*

6.6.3 Event Group Services

The purpose of event groups is to synchronize threads on the value of one or more binary flags where each flag represents a user specified status.

An event group is an AVIX kernel object holding a total of 16 event flags. Threads can wait for any combination of event flags to be set. As long as the specified event flags are not set, the thread will have the 'Blocked' state. When the specified event flags are set, the thread enters the 'Ready' state.

Event flags can explicitly be set, cleared or toggled by other threads, ISR's or DIH's. Event flags can also implicitly be changed by other services based on the status of the applicable kernel object like a timer that expires or an asynchronous pipe operation that finishes.

AVIX offers two types of event groups which functionally are nearly identical:

1. Global event groups: A global event group must explicitly be created. Other threads can obtain access to it by using the name given to this event group. Global event groups can be used by multiple threads to wait for the desired flags and at the same time by multiple threads to change the status of one or more flags.

2. Thread event groups: Every thread has got a local thread event group. When the thread exists, its local event group exists. Thread event groups do not need to be created explicitly but are implicitly created when the thread is created. A thread event group may only be waited for by the 'owning' thread. Just like global event groups, multiple threads may change the status of one or more flags.

► *Thread event groups combine the power and flexibility of the event group mechanism reducing memory consumption since no explicit creation is required.*

Event groups contain event flags. Event flags can be considered to be bit 0 to 15 of a 16-bit word. An event flag is addressed using a 16-bit word with the bit corresponding to the desired flag set one. When using multiple flags in a single call, the word contains multiple bits set one. Helper macros are available to translate flag numbers to the 16-bit word representation.

Event groups are a very powerful mechanism allowing a thread to wait for multiple triggers in a single function call. Imagine a thread that must react to any of a number of input conditions to switch off some heater element. By representing each of the conditions as an event flag, the thread can wait with a single AVIX function call for those conditions and when one or more are true, continue and switch off the heater element.

When changing one or more flags, the flags can be set (`AVIX_EVENT_GROUP_SET`), cleared (`AVIX_EVENT_GROUP_CLEAR`) or inverted (`AVIX_EVENT_GROUP_TOGGLE`). When waiting for multiple flags, the wait request is satisfied when either all flags (`AVIX_EVENT_GROUP_ALL`) or any flag (`AVIX_EVENT_GROUP_ANY`) is set.

► *Event flags may be changed from threads, ISR's and DIH's.*

An event group is identified by an id of type `tavixEventId`. When creating an event group using function `avixEventGroup_Create`, an id of this type is returned as a function result. When passing a name to this function, other threads can use this name to obtain access to the event group using function `avixEventGroup_Get`. The event group id can next be used to change the status of the flags in the event group using function `avixEventGroup_Change` or wait for the desired status of flags using function `avixEventGroup_Wait`.

Access to a thread event group is obtained by converting the thread id to an event group id by appending attribute `.asEventId` to the thread id. For more details on using a thread id to access the thread local event group see §6.5.1.

When waiting for a thread event group, the event group id passed to the wait function may only be the id of the calling thread converted to an event group id (`avixThread_GetIdCurrent().asEventId`). Passing the id of another thread will be reported as an error.

Code sample 20 shows an example where signalThread1 will set flag 0 in an event group when some condition is met. Flag 1 is set by signalThread2 when some other condition is met. Thread actionThread waits for both flags and enters the 'Ready' state once both flags are set. The event group is created by the actionThread and access is obtained by the other threads through the name passed to the create function.

```
1  TAVIX_THREAD_REGULAR signalThread1(void* p)   // Signal thread 1
2  {
3    tavixEventId evgrId;
4
5    evgrId = avixEventGroup_Get("evgr");         // Obtain access to event group
6                                                 // created by actionThread
7    while(1)
8    {
9      …;                                         // Action leading to set flag 0
10     avixEventGroup_Change                      // Set flag 0 in the event group
11       ( evgrId,                                // Id of event group to set flag in
12         AVIX_EVENT_GROUP_SET,                  // Specify the flag must be set
13         AVIX_EF(0)   );                        // Convert '0' to desired bit-mask
14   }
15 }
16
17 TAVIX_THREAD_REGULAR signalThread2(void* p)   // Signal thread 2
18 {
19   tavixEventId evgrId;
20
21   evgrId = avixEventGroup_Get("evgr");         // Obtain access to event group
22                                                // created by actionThread
23   while(1)
24   {
25     …;                                         // Action leading to set flag 1
26     avixEventGroup_Change                      // Set flag 0 in the event group
27       ( evgrId,                                // Id of event group to set flag in
28         AVIX_EVENT_GROUP_SET,                  // Specify the flag must be set
29         AVIX_EF(1)   );                        // Convert '1' to desired bit-mask
30   }
31 }
32
33 TAVIX_THREAD_REGULAR actionThread(void* p)    // Action thread
34 {
35   tavixEventId evgrId;
36
37   evgrId = avixEventGroup_Create              // Create an event group
38              (  "evgr",                       // Name for other threads to access
39                 AVIX_EF_NONE );               // Initially all flags are cleared
40
41   while(1)
42   {
43     avixEventGroup_Wait                        // Wait for flags
44       (  evgrId,
45          AVIX_EF_RANGE(0, 1),                  // Flags waited for are 0 and 1
46          AVIX_EVENT_GROUP_ALL,                 // Both must be set to continue
47          AVIX_EF_NONE,                         // Clear no flags when start waiting
48          AVIX_EF_RANGE(0, 1)   );              // Clear both flags when done
49       …;                                       // Action when both flags are set
50   }
51 }
```

Code sample 20: How to use event groups

Code sample 21 shows an example which is functionally the same as the example in Code sample 20. The difference is that instead of a global event group, use is made of the local event group of actionThread. In Code sample 20, the signal threads use function `avixEventGroup_Get` to obtain access to the event group created by the action thread. In Code sample 21, no event group is created since this exists implicitly in the action thread. So to obtain access to this thread event group, the signal threads use function `avixThread_Get` and subsequently the thread id returned by this function is converted to the thread local event group id. For this sample, the assumption is made the action thread is created with name “evth”. Note line 38 where the thread obtains its own id and converts this to an event group id using attribute `.asEventId`.

```
 1  TAVIX_THREAD_REGULAR signalThread1(void* p)    // Signal thread 1
 2  {
 3    tavixThreadId actionThreadId;
 4
 5    actionThreadId = avixThread_Get(“evth”);     // Obtain access to id of action thread
 6
 7    while(1)
 8    {
 9      …;                                         // Action leading to set flag 0
10      avixEventGroup_Change                      // Set flag 0 in the thread event group
11        ( actionThreadId.asEventId,              // Id of event group to set flag in
12          AVIX_EVENT_GROUP_SET,                  // Specify the flag must be set
13          AVIX_EF(0)   );                        // Convert ‘0’ to desired bit-mask
14    }
15  }
16
17  TAVIX_THREAD_REGULAR signalThread2(void* p)    // Signal thread 2
18  {
19    tavixThreadId actionThreadId;
20
21    actionThreadId = avixThread_Get(“evth”);     // Obtain access to id of action thread
22
23    while(1)
24    {
25      …;                                         // Action leading to set flag 1
26      avixEventGroup_Change                      // Set flag 0 in the thread event group
27        ( actionThreadId.asEventId,              // Id of event group to set flag in
28          AVIX_EVENT_GROUP_SET,                  // Specify the flag must be set
29          AVIX_EF(1)   );                        // Convert ‘1’ to desired bit-mask
30    }
31  }
32
33  TAVIX_THREAD_REGULAR actionThread(void* p)     // Action thread
34  {
35    while(1)
36    {
37      avixEventGroup_Wait                        // Wait for flags
38        (  avixThread_GetIdCurrent().asEventId,  // Thread id of this thread as event id
39           AVIX_EF_RANGE(0, 1),                  // Flags waited for are 0 and 1
40           AVIX_EVENT_GROUP_ALL,                 // Both must be set to continue
41           AVIX_EF_NONE,                         // Clear no flags when start waiting
42           AVIX_EF_RANGE(0, 1)   );              // Clear both flags when done
43        …;                                       // Action when both flags are set
44    }
45  }
```

Code sample 21: How to use thread event groups

How to control the value of flags

Two parameters to the event group wait function `avixEventGroup_Wait` require special attention; these are the `preClearMask` and the `postClearMask`. These parameters allow for a very large degree of flexibility when using event groups.

Parameter `preClearMask` specifies event flags that are cleared as the very first action of the wait function before doing anything else. Suppose a thread wants to wait for flag 0 to be set. The thread is not interested whether the flag is already set when executing the wait function. By specifying flag 0 in the `preClearMask`, whatever the value of this flag, it will be cleared as part of the function call and the thread will start waiting for the flag to become set again.

Parameter `postClearMask` specifies event flags that are cleared once the wait condition is fulfilled. Suppose one thread periodically sets a flag for another thread to react on. In this case it is likely the waiting thread will clear the flag once it became set. Not doing so will result in a subsequent wait operation to continue immediately whether or not the generating thread set the flag again or not. This most likely is not desired and can be solved by the waiting thread to specify the flag in the `postClearMask`. When the wait function returns because the desired flags are set, before returning to the caller the flags identified by `postClearMask` are cleared.

No relation exists between the values of `preClearMask`, `postClearMask` and the mask with the flags actually waited for. In one wait call it is possible to specify flag 0 to be cleared at the start of the function call, flag 1 to be the flag actually waiting for and once this flag is set and the wait function is ready, clear flag 2.

Event flags can be used to represent an absolute condition or a transient condition, depending on the requirements of the application.

When a flag is neither cleared through the `preClearMask` or the `postClearMask`, it will be controlled only by some other thread. An example of this is when a thread sets a flag when a temperature exceeds a certain value and clears the flag when the temperature drops below that value. In that case the semantics of the flag are ‘temperature above threshold’. In this case the value of the flag contains the desired information. ***These types of flags represent an absolute condition.***

When on the other hand a flag represents the occurrence of a user of the system pressing a button, it makes sense to clear the flag as part of the wait function using `postClearMask`. Would this not be done, the flag remains set and no new key pressed could be detected. ***These types of flag represent a transient condition.***

Note that whether an event group flag represents an absolute or a transient condition is entirely application dependant.

► *AVIX event groups combined with the richness of controlling the value of event group flags offer the ultimate flexibility to solve any possible situation when the value of application parameters can be expressed in (a combination) of binary values.*

Event Groups and ISR’s

Event flags in an event group can be changed directly from ISR’s For this purpose function `avixEventGroup_ChangeFromISR` exists.

Do note that ISR’s should be as short as possible. Instead of directly changing event flags from an ISR, the ISR can enqueue a DIH where the flags are passed as the DIH parameter. The DIH will then call `avixEventGroup_Change` to change the desired event flags. Note the DIH uses the regular function and not the ‘`…FromISR`’ function.

Event groups and other service categories

Besides explicitly controlling the value of event flags using function `avixEventGroup_Change`, event flags can also be controlled by the status of timers, asynchronous pipe operations, message queues and exchange operations.

- AVIX allows an event group to be ‘connected’ to a timer. When doing so, one must also specify one or more flags and one of the three possible actions, set, clear or toggle. When the timer expires, the specified action is executed on this event group for the specified flags. More details are found in §6.6.4.

- An event group can be ‘connected’ to a message queue. When this is used, one must also specify one or more flags but in this case no action is specified. After connecting an event group to a message queue, AVIX will set the flag(s) as long as there is at least one message present in the message queue and clear the flag as soon as the message queue becomes empty. Note that this is a typical example of an absolute condition. It makes no sense for the wait function to clear this flag connected to a message queue since waiting for this flag does not influence the filling of the message queue. This flag is entirely controlled by AVIX, based on the content of the message queue. More details are found in §6.6.4 .

- When using asynchronous pipe operations, optionally an event group id and one or more event flags can be passed as parameters to these operations. When the asynchronous pipe operation does not finish immediately, the specified event flags are cleared. Once the asynchronous pipe operation finishes, the specified event flags are set. More details are found in §6.6.6.

- When using exchange operations, callback functions may be connected to exchange objects resulting in event flags being set when data is written to the exchange object. Details on this type of services are found in §6.6.8.

Using this feature, a thread can wait at a single location for many different types of system events. Each system event is related to a flag and once the wait function returns, by testing which flags are set, it can be determined which system event occurred and the appropriate action can be taken.

This allows for a thread to have a single point in its code where it may block. Without this feature, a thread typically contains multiple blocking points which has a large disadvantage since when waiting for a one system event, the thread does not react to other system events. With other RTOSes, this problem is often solved by ‘polling’ for the desired events but polling is a very inefficient mechanism. One of the reasons to use an RTOS in the first place is that no use needs to be made of polling. The event group mechanism offered by AVIX prevents this and leads to highly modular threads which have an optimal performance.

Examples of described mechanisms are provided with the applicable service descriptions.

6.6.4 Timer Services

The purpose of timer services is to allow threads to enter the 'Blocked' state for a specified time, after which the thread is allowed to be active again. This either once or repeatedly.

Managing time is one of the most important services offered by an RTOS. AVIX offers extensive timer functionality in the form of timer kernel objects.

AVIX timers are software timers. All AVIX timers make use of a single hardware timer (system timer), regardless the number of software timers used. Depending on the applied platform, the system timer can be selected through the AVIX configuration or use is made of a dedicated timer offered by the platform. Details are found in the platform specific Porting Guide. The system timer is configured for a specific period through configuration parameter `avix_SYS_CLOCK_TICKus`[14] and runs continuously. Every time the system timer expires, a tick occurs which can be considered a clock pulse for the software timers. Therefore, the period of an AVIX timer is specified as ticks where a tick corresponds with the configured system timer period.

A timer is created using function `avixTimer_Create`. The timer will start counting using function `avixTimer_Start`. Before doing so, the number of ticks to count and the type of timer must be specified using function `avixTimer_Set`.

The type of a timer is either single-shot or cyclic. Since `avixTimer_Set` may be called as often as desired, once a timer is created, its type and the number of ticks can be changed at any time.

The two different types a timer can have are:

- **Single shot**: A timer initialized as 'single shot', upon expiration stops counting and enters the expired state. In order to let it count for another period, the timer must explicitly be started again using `avixTimerStart`.

- **Cyclic**: A timer initialized as 'cyclic', upon expiration automatically starts counting for another period after which it expires again. This repeats until the timer is explicitly stopped. A cyclic timer will never enter the expired state. As long as it is active, its state remains counting.

When starting a timer, it starts counting for a specified number of ticks. When the specified number of ticks have elapsed, the timer expires. A thread can wait for a timer to expire during which time the thread is in the 'Blocked' state. Waiting for a timer to expire can be done in two ways:

- A timer is a wait-able object in itself. Threads can wait for a timer to expire by using function `avixTimer_Wait`. Doing so, the thread is placed in the wait list of the timer until the timer expires.

- Optionally an event group can be connected to a timer (`avixTimer_ConnectEventGroup`). In this case the thread does not wait for the timer but for the specified flags in the event group to become set. When the timer expires the specified flags are either set, cleared or toggled in the event group. This allows for a thread to wait for multiple events in a single potentially blocking call.

► *When connecting an event group to a timer, a thread can wait for the timer or for the event group. This offers ultimate flexibility in the application of timers.*

[14] The actual period of a timer tick may differ slightly from the value configured through parameter `avix_SYS_CLOCK_TICKus`. The actual hardware timer period is available through macro `AVIX_SYS_CLOCK_ACTUAL_PERIOD`. When needing the actual hardware timer period, use this macro instead of macro `avix_SYS_CLOCK_TICKus`.

A timer is always in one of four possible states, which are ‘uninitialized’, ‘stopped’, ‘counting’ or ‘expired’. The state of a timer changes either by using specific functions or by a single shot timer expiring. The state of a timer determines the functions that are allowed to be used. Details on the state of a timer and the transitions between these states are found in section ‘Timer states’.

When counting, a timer may be stopped using function `avixTimer_Stop`. The current count value is remembered by the timer and it may be resumed again using function `avixTimer_Resume`.

The number of ticks of a timer may be changed ‘under fly’ using function `avixTimer_SetPeriod`.

Timers may be started, stopped and resumed from an ISR using functions `avixTimer_StartFromISR`, `avixTimer_StopFromISR` or `avixTimer_ResumeFromISR`.

Finally, at any given moment the number of ticks remaining before the timer will expire can be obtained using function `avixTimer_GetRemainingTicks`.

Since awareness of the way a timer deals with ticks is important, the remainder of this chapter and the provided code samples are based on timer ticks.

Code sample 22 shows a thread using a cyclic timer to start running every 5 system ticks. The timer is initialized as a cyclic timer and all the thread has to do is wait for the timer using function `avixTimer_Wait`.

```
 1 TAVIX_THREAD_REGULAR demoThread(void* p)
 2 {
 3   AVIX_OBJECT_ID_DEFINE(tavixTimerId, timer);
 4
 5   timer = avixTimer_Create(NULL);              // Create a timer object
 6
 7   avixTimer_Set(timer, 5, AVIX_TIMER_CYCLIC); // Set for 5 ticks and cyclic
 8
 9   avixTimer_Start(timer);                      // Start the timer
10
11   while(1)
12   {
13     avixTimer_Wait(timer);                     // Thread is blocked until timer expires
14
15     // Thread wakes up and does something usefull.
16   }
17 }
```

Code sample 22: Using a cyclic timer

Code sample 23 shows an example where the thread connects its local event group to a timer. Doing so, with a single function call (`avixEventGroup_Wait`) the thread can either be activated when the timer expires or when some other entity sets flag 1.

```
1  TAVIX_THREAD_REGULAR demoThread(void* p)
2  {
3    AVIX_OBJECT_ID_DEFINE(tavixTimerId,     timer);
4    tavixEventFlags                         flags;
5
6    timer = avixTimer_Create(NULL);               // Create a timer object
7
8    avixTimer_Set(timer, 5, AVIX_TIMER_CYCLIC);   // Set for 5 ticks and cyclic
9
10   avixTimer_ConnectEventGroup                   // Connect an event group to the timer.
11     (  timer,                                   // It concerns the thread local event
12        avixThread_GetIdCurrent().asEventId,     // Use thread local event group
13        AVIX_EVENT_GROUP_SET,                    // When the timer expires, flag 0 of
14        AVIX_EF(0)    );                         // this event group is set.
15
16   avixTimer_Start(timer);                       // Start the timer
17
18   while(1)
19   {
20     // In the thread main loop, it waits for either flag 0 or flag 1. Flag 1 is
21     // supposed to be set by some other thread and flag 0 is set when the timer
22     // expires. Both flags are cleared when the function is done (postClearFlags).
23     // This allows the thread to run when either the timer expires or when some
24     // other action causes flag 1 to be set.
25     Flags =
26       avixEventGroup_Wait                       // Wait for event flags
27       (  avixThread_GetIdCurrent().asEventId,   // Use thread local event group
28          AVIX_EF_RANGE(0, 1),                   // Flags waited for are 0 and 1
29          AVIX_EVENT_GROUP_ANY,                  // Wait for any flag to be set
30          AVIX_EF_NONE,                          // Clear no flags when start waiting
31          AVIX_EF_RANGE(0, 1)   );               // Clear flag 0 and 1
32
33     if(AVIX_EF_IN(AVIX_EF(0), flags))           // Test if flag 0 was set (timer)
34     {
35       …;                                        // Execute timer related actions
36     }
37     if(AVIX_EF_IN(AVIX_EF(1), flags))           // Test if flag 1 was set (message?,
38     {                                           // other timer? plain event flag
39       …;                                        // action ?
40     }
41   }
42 }
```

Code sample 23: Using a cyclic timer with a connected event group

Thread local timer usage

Besides explicitly created timers, every thread has got a thread local timer. This thread local timer does not need to be explicitly created, it exists when a thread exists. A thread local timer cannot be directly accessed but is used for two purposes:

- Sleep: When a thread has to wait for a specific time, use can be made of function `avixThread_Sleep`. Passed to this function are the number of ticks the thread wants to wait and during this time, the thread will enter the 'Blocked' state. When the thread local timer expires, the thread will enter the 'Ready' state again.

- Time out: When using the AVIX time-out mechanism through functions avixThread_ArmTimeOut and avixThread_TimeOut Occured, use is made of the thread internal timer.

Timer Accuracy & Resolution

AVIX timers are set to count for a number of ticks. A tick represents the time configured for the AVIX system timer through configuration parameter `avix_SYS_CLOCK_TICKus`. This configuration parameter typically has a value somewhere between 100µs and 1ms. All software timing is derived from the AVIX system timer. This system timer implements a central heartbeat which 'ticks' independent from the application. The period configured for the system timer determines the resolution and accuracy of the AVIX timers.

Due to rounding, it is possible the actual system timer period cannot be equal to the configured value. The actual system timer period is defined in symbol `AVIX_SYS_CLOCK_ACTUAL_PERIOD`. When calculations based on the system timer period are required, this symbol should be used instead of symbol `avix_SYS_CLOCK_TICKus`.

► *Most competing products don't allow the system heartbeat to have a configurable period. There seems to be consensus for a period of 1ms. AVIX offers the advantage the system tick can have a much lower period leading to increased timer accuracy.*

When the system timer is configured to have a period of 100µs, AVIX timers can count for an integer multiple of this time specified by the tick count. A single tick corresponds with a time of 100µs which causes AVIX timers to have a period which is a multiple of 100µs.

The tick generated by the AVIX system timer and the application software operate fully asynchronous from each other. For this reason, when starting a timer it is unknown how much time will elapse before the next system tick occurs. In practice this can be any time between 0µs and 100µs. As a result, when starting a timer to wait for one tick, this can result in the timer to expire anywhere between 0µs and 100µs.

To compensate for this, when starting an AVIX timer, the first period is always extended with an extra tick. When starting an AVIX timer for one tick, it is set to expire after two ticks. Doing so the actual time will be somewhere between 100µs and 200µs. This guarantees the actual time is guaranteed to be at least equal to the time specified.

As said before, this is true for the first period. When using a cyclic timer, the timer is automatically restarted when it expires. For a cyclic timer, all but the first period is exactly the number of ticks specified since for these periods the moment the timer is restarted coincides with the system tick.

The above is illustrated in Figure 7.

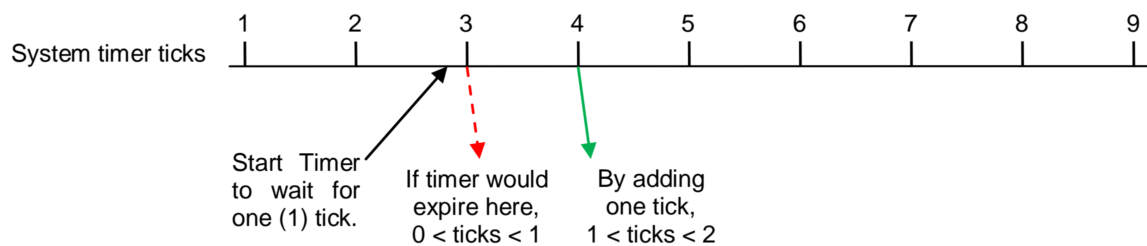


Figure 7: Timer, relation between system timer and application timer period

The numbers in the upper part of Figure 7 represent the system timer ticks which occur periodically at the configured rate. Every time the system timer expires, AVIX updates the then active software timers. Suppose a thread wants to wait for a single tick. The moment this wait request occurs is represented by 'Start Timer ...'. This moment is not synchronized with the system timer and may occur at any moment in between two system timer ticks. Would the timer actually wait for a single tick, then the moment system timer tick 3 occurs the timer will expire. Effectively the thread would have waited then for ~0,2 tick. By adding one (1) to the specified number of ticks, the thread will wait for the system timer to expire twice and the time waited will be ~1,2 ticks. This mechanism ensures the period of an AVIX timer is never shorter than the specified number of ticks.

When this compensation mechanism is not desired, instead of using `avixTimer_Start`, alternatively a timer can be started using `avixTimer_Resume`. This function uses the specified number of ticks and does not add a tick to the first period.

▶ *To determine the number of ticks based on a time expressed in hours, minutes, seconds, milliseconds and/or microseconds, AVIX offers a number of macros to convert a time in a regular notation to a number of ticks. Use of these macros is strongly advised instead of directly specifying the number of ticks as a plain numeric value. Besides ease of use, an important reason to use these macros is that when reconfiguring the period of the system timer, these macros will automatically generate the number of ticks based on the new system timer period.*

An example of the usage of these macros is shown in Code sample 24.

```
 1 // With the system timer configured for a period of 100µs, waiting 500µs
 2 // can be programmed like:
 3 //
 4 avixTimer_Set(timer, 5, AVIX_TIMER_CYCLIC);                 // Set for 5 ticks
 5
 6
 7 // But more readable and still correct when the system timer period is changed
 8 // the way below is advised to be used.
 9 //
10 avixTimer_Set(timer, AVIX_DELAY_US(500), AVIX_TIMER_CYCLIC); // Set for 500µs
```

Code sample 24: How to specify a timer period

The timer period is specified as a signed 32 bit parameter. The maximum value for the number of ticks that can be specified is `0x7FFFFFFD` or 2,147,483,645 decimal. The time this represents depends on the value which is configured for the system timer.

▶ *In case the system timer is configured to have a value of 100µs, the maximum time a timer can count before expiring is over 59 hours.*

Reconfiguring the period of a cyclic timer

Once a cyclic timer is active, it expires repeatedly every time the specified number of ticks has elapsed. Applications may require the period of a cyclic timer to be changed while it is active. One may expect the transition to a different period to go as smooth as possible. For a timer to have its period changed, it must be deactivated, re-initialized with the new period and restarted. This will however lead to a disturbance in the timer's period as illustrated by Figure 8.

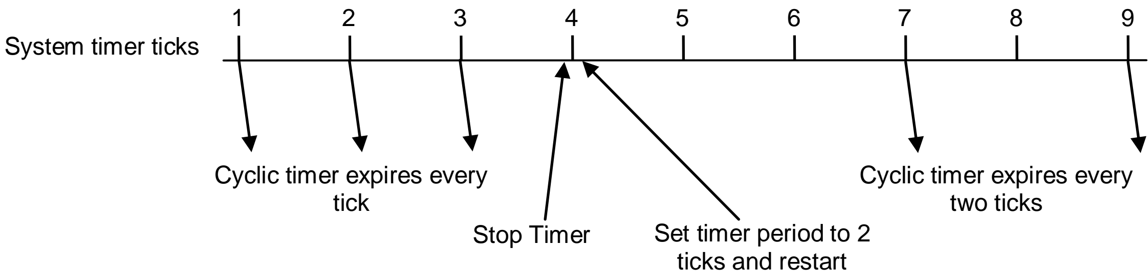


Figure 8: Cyclic timer period change with disturbance

A cyclic timer is active and expires on every tick (1, 2 and 3). When a thread wants to change the period to two ticks, it stops the timer, changes the period and starts the timer again. Suppose now the stopping is done just before system timer tick 4 occurs and starting the timer happens just after tick 4 occurs. The timer will be restarted with a period of two ticks. As described before, to the first period a tick is added so the first moment the timer will expire after having received a new period is at system timer tick 7, after which it will repeatedly expire every two system timer ticks, as intended. During the transition however the timer did not expire for three system timer ticks. In between the timer expiring every single tick and expiring every two ticks, there is a gap where the timer did not expire with one of the intended periods. Many different scenarios' can be thought of but the issue is that using the described mechanism will lead to a disturbance of the intended expiration periods.

To prevent this from happening, AVIX offers function `avixTimer_SetPeriod`. When using this function with an active cyclic timer, the specified period will be used to automatically restart the timer with this new period at the first expiration after setting this new period. The timer does not need to be stopped and the transition from one period to another goes smoothly without any disturbance. This is illustrated in Figure 9.

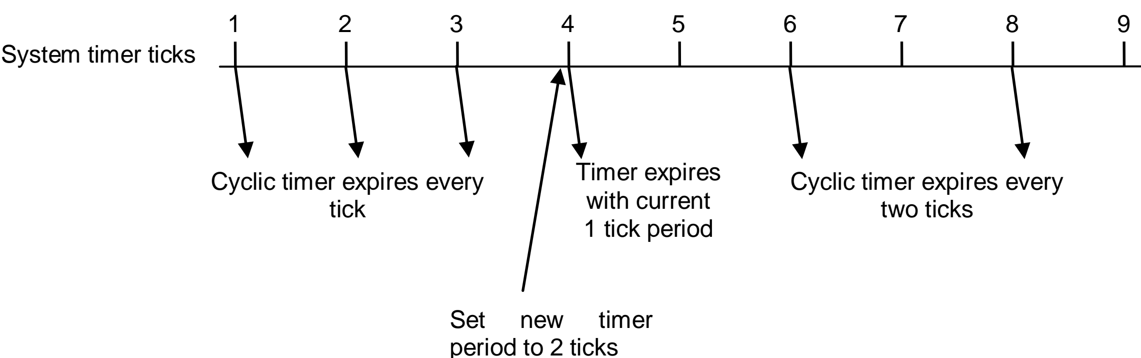


Figure 9: Cyclic timer period change without disturbance

The timer is repeatedly expiring every tick. In between tick 3 and tick 4, the period of the timer is changed to two ticks using function `avixTimer_SetPeriod`. The new period is stored with the timer but the timer remains active and will finish its current period so it expires at tick 4 with the old period. At this moment, the newly programmed timer period is used to automatically restart the timer for another period. At this moment the new period of 2 ticks is used and the next expiration will be after two ticks, so at system timer tick 6 and this will repeat as intended.

The transition from one period to another goes smoothly without any disturbance.

Function `avixTimer_SetPeriod` is intended to be used for counting cyclic timers as explained before. Use of this function is not restricted to this situation. When used for single-shot timers or timers that are not counting, this function effectively sets the number of ticks used the next time the timer is started.

Timer states

To use a timer, it must be created and initialized. The reason to separate initialization from creation is that by doing so, a timer can be reused for different periods or different types (cyclic or single-shot).

A timer object is always in one of the following states:

- **Uninitialized**: This is the state a timer object is in after creation. The only valid operation is to set the timers properties using `avixTimer_Set`. All other operations are considered programming errors.
- **Stopped**: This is the state a timer object is in after setting its properties before it is started or by setting or stopping a counting timer.
- **Counting**: This is the state a timer object is in after it has been started and while neither explicitly stopped or expired in the case of a single shot timer.
- **Expired**: This is the state a single shot timer object is in after its period has expired. This state is maintained until either the timer is set or started again.

The possible timer object states and the AVIX functions that change them are shown in Table 3.

It is very important to realize that the process of counting performed by a timer object runs concurrently with other processes under control of AVIX like threads, other timers and interrupt routines. Therefore one cannot always be sure in which state a timer object function is called since when the call is made, this state might just have changed.

	Timer State			
Function	**Uninitialized**	**Stopped**	**Counting**	**Expired**
`avixTimer_Set`	✓	✓	✓ (2)	✓
(resulting state)	**Stopped**	**Stopped**	**Stopped**	**Stopped**
`avixTimer_Start` `avixTimer_Resume`	✘	✓	✓ (3)	✓
(resulting state)		**Counting**	**Counting**	**Counting**
`avixTimer_Stop`	✘	✓	✓	✓
(resulting state)		**Stopped**	**Stopped**	**Stopped**
`avixTimer_Wait`	✓	✓ (1)	✓	✓ (5)
(resulting state)	**Uninitialized**	**Stopped**	**Counting**	**Expired**
Timer expires	n.a	n.a	✓ (4)	n.a.
(resulting state)			**Expired / Counting**	

Meaning of symbols

- Blue field: State transition
- ✘ : Function not allowed to be called in this state.
- ✓: Function allowed to be called in this state.
- n.a: Not applicable, this will not happen

Details:

1. When a thread waits for a Stopped timer, it enters the 'Blocked' state. The timer must be started and expire for this thread to enter the 'Ready' state again.
2. Calling `avixTimer_Set` for a timer in state counting aborts the current time period and leaves the timer in state stopped with the newly set properties.
3. Calling `avixTimer_Start` for a timer in state counting aborts the current time period and restarts the timer based on its current properties.
4. The timer actions are executed. A cyclic timer automatically starts another period while remaining in the counting state. A single shot timer goes to the expired state.
5. When a timer has been running and it is expired, a wait on this timer is honored immediately. This is needed since when starting a timer, it cannot be guaranteed that the timer is not expired before a thread starts waiting for it.

Table 3: AVIX timer state transitions

6.6.5 Message Services

The purpose of a message is to send a request to a thread. A request triggers the receiving thread to execute specific thread internal functionality. Requests are send as an identifying numeric code optionally accompanied by additional user specific data.

Message services allow a request to be sent to a thread in the form of a message. A messages is a block of memory with a fixed, though configurable, size. The semantics of the request are identified by the content placed in this memory by the requestor. Message content consists of two parts. First there is a numeric value, the message type, identifying the request. This numeric value is user specified and should be agreed upon between the requesting and the receiving thread. Often this is all that is needed. Suppose a thread is used to control a valve. In this case the message type will typically contain a value like OPEN_VALVE or CLOSE_VALVE which are constants defined in a header file shared between the requesting and the receiving thread. Optionally the message body may contain additional data in case the type attribute does not provide sufficient information to fully specify the request. Both the structure and the content of the message body is again user specified. Suppose the thread in the mentioned example controls more than one valve the message body may contain an integer specifying the applicable valve.

Memory blocks used for messages are allocated from a dedicated memory pool created during AVIX initialization. The number of memory blocks in this pool is configurable. After allocating a message, it will be filled with the required information and sent to a thread. The receiving thread is responsible for returning this message block to the memory pool in case it is not needed for other purposes. When allocating a message from an empty message pool the thread will enter the 'Blocked' state until a message is present again in the message pool because it is freed by another thread.

► *Messages can be allocated and sent from a thread, an ISR or a DIH. The destination of a message is always a thread.*

When sending a message, the content of the message is not copied. Instead a reference to the message in the form of a message id is transferred. This has a positive effect on performance but does also introduce a risk. After having sent the message the requestor still has access to the id of the message. Changes can be made to the message while the receiving thread is busy processing it. To prevent this, AVIX implements message ownership. Only the owner of a message is allowed to access its content and send the message. After allocating a message the allocating thread owns the message. After sending the message, ownership is lost (access to the message is flagged as an error). When receiving a message the receiving thread obtains ownership of the message.

After processing a received message, the message must be returned to the message pool. Instead of this, a receiving thread may also reuse the message in case it has to send a message itself. This can be because the receiver wants to request something from another thread. Alternatively the message can be reused to reply to the requesting thread for which purpose the message contains the id of the sending thread. This last option only exists when the message originally was sent by a thread. Messages received from an ISR or a DIH cannot be replied to since ISR's and DIH's are not capable of receiving messages. To know whether a message is received from a thread or not, the receiver can request the thread id of the sending thread. In case this is a valid kernel object id, it represents the thread id of the message originating thread. In case the message was sent from an ISR or a DIH, this will be a `NULL` id.

Message Queues

Messages can be sent while the receiving thread is not currently able to receive the message since it is doing something else. For this purpose every thread has a list where incoming messages (their id that is) are placed in a FIFO order. In case one or more messages are present in this list and the thread executes the receive function, the first message is removed from the list and returned as a

result of the receive function. When a thread calls the receive function while this list is empty, the thread enters the 'Blocked' state. When a message is send to a thread Blocked on this list, instead of being placed in the list, the message is directly handed over to the receiving thread which will enter the 'Ready' state. This has a positive influence on performance since the list operations are skipped in this case.

All message features like immediately transferring the message to a thread in receive mode, exchanging the message by id and the built in safety measures make the AVIX message mechanism a fast, flexible and user friendly means to exchange information between threads.

Code sample 25 shows basic message usage where one thread allocates a message, sends it to another thread which, after retrieving the required information from the message frees it. Subsequently, based on the type of the message functionality is executed.

```
1  TAVIX_THREAD_REGULAR srcThread1(void* p)
2  {
3    AVIX_OBJECT_ID_DEFINE(avixMsgId, msg);            // Variable to hold message id
4    AVIX_OBJECT_ID_DEFINE(tavixThreadId, destThread); // Variable to hold dest thread id
5
6    while(1)
7    {
8      …;
9
10     msg = avixMsg_Allocate(1);                   // Create a message with type value 1
11     avixMsg_PutInt(msg, 2);                      // Put an integer with value 2 in body
12     avixMsgQThread_Send(msg, destThread);        // Send the message to the destination
13   }
14 }
15
16 TAVIX_THREAD_REGULAR destThread1(void* p)
17 {
18   AVIX_OBJECT_ID_DEFINE(tavixMsgId, msg);        // Variable to hold message id
19   tavixMsgType  msgType                          // Variable to hold message type
20   int           msgParam;                        // Variable to hold additional param
21
22   while(1)
23   {
24     msg      = avixMsgQThread_Receive();         // Receive a message
25     msgType  = avixMsg_GetType(msg);             // Get the type from the message
26     msgParam = avixMsg_GetInt(msg);              // Get the additional param from msg
27     avixMsg_Free(msg);                           // Data is extracted, so free the msg
28
29     if (msgType == 0)
30     {
31       …;                                         // Processing for type 0 message
32     }
33     if (msgType == 1)
34     {
35       …;                                         // Processing for type 1 message
36     }
37   }
38 }
```

Code sample 25: How to use a message queue

Accessing the message body

The inherent safety offered by messages forbids the message body to be directly written or read using a pointer. Instead access functions are offered to put data in the message body or get data from the message body. When for instance two integers have to be placed in the message body, two calls to the `avixMsg_PutInt` function must be made. The values used as a parameter to these functions are placed in the body in the order the calls are made. For this purpose, the message implementation maintains an internal reference. With every `avixMsg_Put_...` function call this reference is incremented with the size of the parameter. When allocating a message, this reference is set to the beginning of the message body.

When receiving a message this reference is reset again. The receiver will read the parameters from the message body using `avixMsg_Get…` function calls. Again with every of these functions the internal reference is incremented with the size of the parameter read from the message body.

The message mechanism maintains a single reference which is used both for `avixMsg_Put…` and `avixMsg_Get…` functions.

As said before, instead of returning a message to the message pool, a thread can decide to reuse the message. Reusing a message is done through function `avixMsg_Reuse`. This function fills the message with the desired type value and resets the internal reference so that before sending the message, the thread can fill the body again using `avixMsg_Put…` functions.

Message data is always ‘packed’, regardless the packing used to build the application.

Messages and ISR’s

ISR’s are allowed to directly send messages. For this purpose functions `avixMsg_AllocateFromISR` and `avixMsg_SendFromISR` are present. For an ISR to fill the message body, special put functions are present. These are named like the regular put functions with the extension ‘`FromISR`’. So besides `avixMsg_PutInt`, there is also a function `avixMsg_PutInfFromISR`[15].

Do note that ISR’s should be as short as possible. Sending a message from an ISR requires a number of functions to be used. Alternatively, the ISR can enqueue a DIH, and pass the information to be send as a message to the DIH instead. Next the DIH will allocate a message and send it. Sending the message like this, the actual ISR is much faster.

Messages and event groups

A regular event group or a thread event group can be ‘connected’ to a message queue using `avixMsgQThread_ConnectEventGroup`. With this function, one or more event flags are specified although most common, only one event flag will be used. This event flag reflects the fact whether messages are present in the message queue. When there are no messages present in the message queue, AVIX will clear the specified event flag. When the number of messages in the queue is greater than zero, AVIX will set the specified event flag(s). Since other flags in the event group can be manipulated by other threads, this allows a thread to wait for multiple events by just waiting for the relevant event group flags.

When a thread waits for such a connected event group and the event flag becomes set, the thread subsequently executes a regular message receive function. In this case it is guaranteed the thread will not Block when calling the receive function since there is no other thread that can receive from the same message queue.

Note that the function used to wait for the applicable event flag may not clear this flag through its `preClearMask` or `postClearMask` parameter since the value of the event flag is entirely controlled by the message mechanism.

Code sample 26 shows a thread that connects an event group to the message queue. In its main loop this thread waits with a single call for either flag 0 or flag 1 being set. Effectively it waits for either a message or an event flag set by another thread and based on which flag is set acts accordingly.

[15] The functions to put data in the message body from an ISR are implemented as macros translating to the regular put functions. These macros exist so that all AVIX functions used from an ISR adhere to the same naming convention.

```
 1 TAVIX_THREAD_REGULAR serverThread1(void* p)       // Server thread
 2 {
 3   AVIX_OBJECT_ID_DEFINE(tavixMsgId, msg);
 4   tavixEventFlags, flags;
 5
 6   // The thread local event group is connected to the message queue. Flag 0 is used
 7   // to identify the message queue fill status.
 8   //
 9   avixMsgQThread_ConnectEventGroup
10     (  avixThread_GetIdCurrent(),                  // Thread id of thread message queue
11        avixThread_GetIdCurrent().asEventId,        // Id of thread event group
12        AVIX_EF(0));                                // Flag to use
13
14   while(1)
15   {
16     // In the thread main loop, it waits for either flag 0 or flag 1. Flag 1 is
17     // supposed to be set from some other thread and cleared when this function is
18     // done (parameter postClearFlags). Very important is flag 0 is not cleared since
19     // this is fully controlled by the message mechanism! When processing the flags
20     // both are tested since it is possible one of the two flags or both are set !!!!
21     //
22     flags =
23       avixEventGroup_Wait                          // Wait for flags
24       (  avixThread_GetIdCurrent().asEventId,      // Id of thread event group
25          AVIX_EF_RANGE(0, 1),                      // Flags waited for are 0 and 1
26          AVIX_EVENT_GROUP_ANY,                     // Wait for any flag to be set
27          AVIX_EF_NONE,                             // Clear no flags when start waiting
28          AVIX_EF(1)  );                            // Clear flag 1, do not clear flag 0
29
30     if(AVIX_EF_IN(AVIX_EF(0), flags))              // Test if flag 0 is set (message)
31     {
32       // If flag 0 is set, a message is guaranteed to be present in the message queue
33       // so it can be read, processed and freed.
34       //
35       msg = avixMsg_Receive();
36       …;                                           // Execute message related actions
37       avixMsg_Free(msg);
38     }
39
40     if(AVIX_EF_IN(AVIX_EF(1), flags))              // Test if flag 1 is set
41     {
42       …;                                           // Execute flag 1 related actions
43     }
44   }
   }
```

Code sample 26: How to use a message queue with connected event group

6.6.6 Pipe Services

The purpose of a Pipe is to allow very fast data exchange between threads, between an ISR and threads or between DIH's and threads. Pipes are the mechanism of choice to integrate interrupt based data handling devices in an application.

Pipe services offer an extremely fast and flexible communication mechanism between threads, between ISR's and threads and between DIH's and threads. Pipe services offer pipe kernel objects, from now on just called pipes. Pipe services consist of a number of functions that operate on a pipe.

Effectively, Pipe services allow data to be transferred from one active entity (thread, ISR or DIH) to another in a controlled fashion. A data producer writes to a pipe and 'knows' how much data is actually transferred or can decide to wait for all required data to be written. A data consumer reads from a pipe and 'knows' how much data is actually transferred or, again, can decide to wait for all data to be read.

For this purpose, both data producers and data consumers use local buffers containing the data to be written or offering space where the data that is read will be stored.

The active entities using a pipe to transfer data operate fully asynchronous from each other. Synchronization between data producers and consumers is taken care of by the Pipe services in a very flexible manner.

Creating a pipe

Before a pipe can be used it must be created. An example of pipe creation is shown in Code sample 27.

```
1  AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
2
3  pipe = avixPipe_Create
4         (  "pipe",         // 1: Name of the pipe
5            100,            // 2: Number of blocks in the pipe internal buffer
6            Sizeof(int),    // 3: Size of a single block in the pipe internal buffer
7            pCallback,      // 4: Address of a pipe callback function
8            pUserData  );   // 5: Address of user specified data to pass to the callback
```

Code sample 27: Creating a pipe

When creating a pipe, an id of type `tavixPipeId` is returned. This id is used in subsequent function calls to identify the pipe these functions operate on. A number of parameters is passed:

Parameter 1 is an optional name. This can be used with function `avixPipe_Get` to obtain access to an existing pipe from another thread.

A pipe has an internal buffer to temporarily hold data. Parameter 2 and 3 specify the number of blocks this buffer can hold and the size in bytes of an individual block.

When using pipes between threads, this buffer may have any desired number of blocks, even zero(0). More details can be found in section '*Pipe buffer size'* on page 79.

When using a pipe from an ISR, the number of blocks in this buffer is important to obtain the desired performance and make sure no data gets lost. In this case guidelines for the 'correct' number of blocks in this buffer are more strict. Details are found in section '*Pipe buffer size and performance tuning'* on page 82. Specifying the size of an individual block in the pipe buffer is important when using the pipe from multiple threads at the same time to prevent data from being corrupted. This is explained in section '*Block approach and transfer by value*' on page 89.

Parameter 4 and 5 optionally specify the address of a callback function and the address of user defined data. These parameters are only relevant when using a pipe from an ISR to allow device control like operations. More details on this and the use of these parameters is found in section '*Device control*' on page 85. When using pipes between threads or between threads and DIH's, a pipe callback function and user data have no relevance and should not be used. In these cases NULL may be passed for these two parameters.

Principle of operation

A pipe is a kernel object, the id of which is passed to the functions offered by Pipe services. Threads can write data *to* a pipe or read data *from* a pipe. For this purpose, a pipe contains an internal buffer, from now on called the pipe buffer. In principle, when writing to a pipe, data is *copied* from an application provided buffer to the pipe buffer. When reading from a pipe, data is *transferred* from the pipe buffer to an application provided buffer.

The pipe buffer is a circular buffer. When data is written to a pipe this data is appended to data already present in the pipe buffer. When data is read from a pipe and more data is present in the pipe buffer than requested, the remaining data is left in the pipe buffer to be transferred by the next read operation.

Note the use of the word *copied* when writing and *transferred* when reading. When writing to a pipe, the data in the application provided buffer is not changed, it is copied to the pipe buffer. When reading from a pipe, data is written to the application provided buffer and the original content of the application provided buffer is destroyed.

This principle of operation is illustrated in Figure 10.

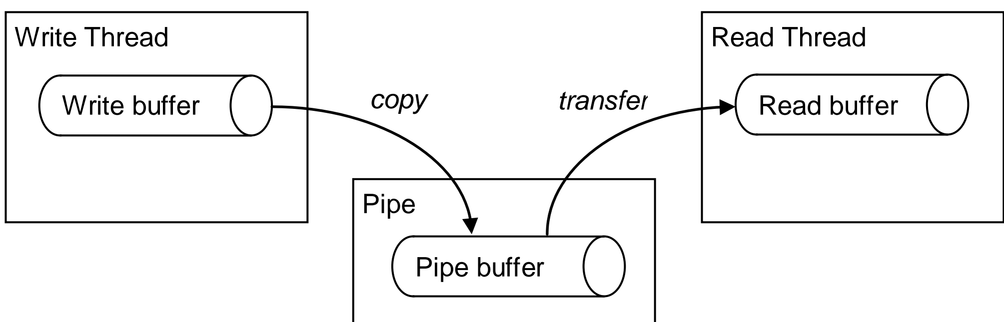


Figure 10: Basic pipe operation

In order to provide a basic understanding the above is a simplified description of how pipes work. In practice pipes offer much more functionality to deal with the following two issues:

1. What if a write operation is executed when the pipe buffer does not have enough free space to store the data or, alternatively, a read operation is executed while the pipe buffer does not contain enough data to satisfy this request?

2. From the perspective of performance, the basic approach would not be optimal. Would the pipe mechanism only offer the aforementioned functionality, transferring data from one thread to another would always imply two copy operations, first from the write thread application buffer to the pipe buffer and next from the pipe buffer to the read thread application buffer. This is far from optimal.

AVIX pipe services are created to deal with both issues, which is the subject of the coming section.

Keeping track of write and read operations progress

A pipe write request can be executed while the empty space in the pipe buffer is insufficient for all required data to be written. A pipe read request can be executed while the amount of data in the pipe buffer is less than required by the read request.

To deal with these situations pipe services implement functionality to keep track of the progress of a request. When a request cannot finish immediately the pipe allocates a so called request descriptor. This request descriptor is used to store information about the progress of the request like the amount of data that needs to be processed before the request is finished, the location in the user provided buffer where the request must continue in case more data becomes available and information about how the thread wants to be informed when the request finishes. This request descriptor is added to the internal bookkeeping of the pipe.

When a request cannot finish immediately resulting in a request descriptor to be allocated, the request is said to be pending. The information in the request descriptor is used with subsequent write or read operations to continue the pending request. As a result the pending request may finish or it may undergo some progress in which case the original request remains pending but the content of the request descriptor is updated to reflect the new status.

Pipes can be used from multiple threads, both for writing and reading and at any given moment, the pipe may contain multiple request descriptors.

Let's assume two threads attempt to read from a pipe while no data is present in the pipe buffer. For both requests a request descriptor is allocated holding the amount of data requested and the address of the read buffer. After the two read operations have executed, the pipe structure looks as shown in Figure 11. Both read requests are said to be pending.

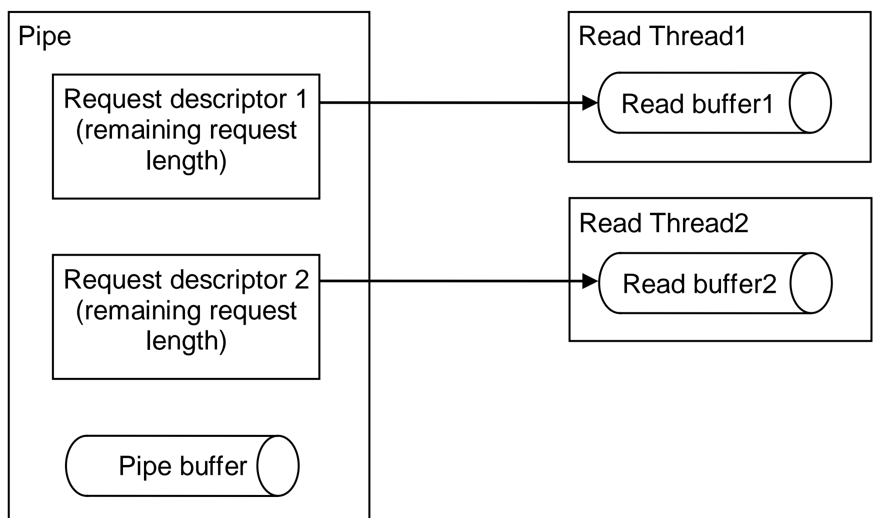


Figure 11: Pipe structure with pending requests

When at a later moment a write operation is executed on the pipe, the information in the request descriptors is used to determine where the written data is to be copied to. Note this is just an example of one of the possibilities. In total six possible scenario's exist which will be explained later.

Request descriptors are maintained in a pipe internal list where they are ordered according the priority of the thread that executed the read or write operation. When pending requests are processed, this is done in the order they are present in the pipe internal list.

Read-write scenarios

When a write operation is executed, three possible scenarios exist:

- *Write to a pipe while no requests are pending*: Data is copied from the client provided buffer to the empty space in the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data to be written.

- *Write to a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full and no data is transferred from the client provided buffer to the pipe buffer. A request descriptor is allocated and the new write request is set pending.

- *Write to a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty. The data of the write request will be copied to the buffers of the pending read requests *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending read requests are updated according the amount of data processed. Pending read requests receiving all required data are set finished and their request descriptors are freed. When all pending read requests are finished and still data remains to be written, this data is copied from the client provided buffer to the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data.

Likewise for a read operation three possible scenario's exist:

- *Read from a pipe while no requests are pending*: Data is transferred from the pipe buffer to the client provided buffer. When not all requested data is present in the pipe buffer, a request descriptor is allocated and the read request is set pending for the remaining amount of data to be read.

- *Read from a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty and no data is transferred to the client provided buffer. A request descriptor is allocated and the new read request is set pending.

- *Read from a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full. First data is transferred from the pipe buffer to the buffer of the read request. If this is not sufficient to finish the read request, data from the pending write requests is transferred to the buffer of the read request *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending write requests are updated according the amount of data processed. Pending write requests for which all data is transferred are set finished and their request descriptors are freed. When all pending write requests are finished and still data remains to be read, a request descriptor is allocated and the read request is set pending. When all requested data is read and still write requests are pending, as much as possible data is transferred from the pending write requests to the pipe buffer. Pending write requests for which all requested data is transferred to the pipe buffer are set finished and their request descriptors are freed.

The request descriptor based mechanism combined with these scenario's solve the aforementioned issues. Pipes deal with operations that cannot finish immediately and, as described in the above scenario's, for optimal performance data is transferred from the buffer of one request to the buffer of another without first being copied to the pipe buffer.

Still one more topic needs to be explained. When a request is not finished, a request descriptor is allocated and the request is set pending. Subsequent pipe operations may fulfil the pending request resulting in the request to be set finished. Requests are made by threads and a mechanism is needed to inform the thread about the status of its request. Two types of requests are offered, synchronous and asynchronous, both offering a dedicated mechanism for the thread to be informed about the status of its pending request.

Types of data transfers

Pipe services offer two types of data transfer, synchronous and asynchronous. Synchronous requests and asynchronous requests may be used concurrently on the same pipe. One thread may read data using asynchronous requests while another thread writes data using synchronous requests. No restrictions exist and it is up to the application designer to select the mechanism that best fits his requirements.

Synchronous data transfers

Synchronous data transfers are the easiest to use and are offered through functions `avixPipe_Write` and `avixPipe_Read`. When a synchronous request cannot finish immediately, a request descriptor is allocated and the request is set pending as described before. When the request is set pending the thread making the request enters the 'Blocked' state and is preempted so other threads may run. When at a later moment pipe operations made from other threads cause the pending request to finish, the thread having made the original request enters the 'Ready' state and may continue.

As a result the thread making the request is not aware of the fact the request has been pending or not. When the function returns, the specified amount of data is guaranteed to be processed. In the meantime, the thread may or may not have been preempted. Effectively the thread waits until the requested amount of data is processed.

The amount of data processed is returned as the result value of `avixPipe_Write` or `avixPipe_Read`. Normally this result value equals the amount of data requested to be processed. Optionally these synchronous functions may be used with the AVIX time-out mechanism to prevent them from waiting too long for the specified amount of data to be processed. In case a time-out occurs before all requested data is processed, the request is aborted, the thread enters the 'Ready' state and the function returns. In this case the function result value indicating the amount of data processed will be less than the amount of data requested.

An example of two threads using synchronous pipe communication is shown in Code sample 28.

```
1   TAVIX_THREAD_REGULAR producerThread(void* p)
2   {
3     char writeBuffer[100];                       // Local buffer of 100 bytes
4     AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
5     pipe = avixPipe_Create("pipe", 100, 1, NULL, NULL);
6
7     while(1)
8     {
9       avixThread_Sleep(AVIX_DELAY_MS(1));        // Pause before next burst
10      avixPipe_Write(pipe, writeBuffer, 100);    // Write the 100 block buffer to pipe
11    }
12  }
13  TAVIX_THREAD_REGULAR consumerThread(void* p)
14  {
15    char readBuffer[100];                        // Local buffer of 100 bytes
16    AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
17    pipe = avixPipe_Get("pipe");
18
19    while(1)
20    {
21      avixPipe_Read(pipe, readBuffer, 100);      // Read 100 blocks from pipe
22      // Process the data just read.
23    }
24  }
```

Code sample 28: Synchronous pipe usage with equal write-read size

This example shows two threads, a data producer and a data consumer. Initially the producer thread creates the pipe providing it with a name so the consumer thread can obtain access to the same pipe by passing this name to `avixPipe_Get`. Next, in an endless loop, the producer thread will sleep for 1ms and write 100 blocks of data to the pipe. The consumer thread, also in an

endless loop, just reads from the pipe. As long as the desired amount of data is not available the consumer thread will be in the 'Blocked' state. The moment the producer thread has written, the consumer thread will enter the 'Ready state and may process the data it just read.

Figure 12 shows the activation of the producer and the consumer thread using the AVIX Thread Activation mechanism. As shown, the consumer is active every time the producer has written its data since both request the same amount of data to be processed.

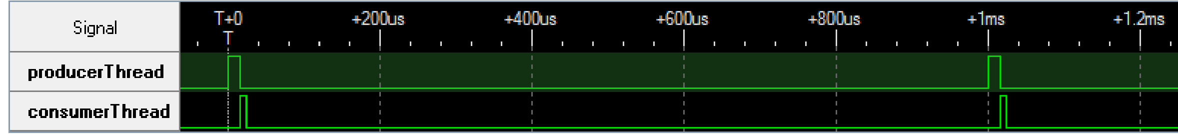


Figure 12: Synchronous pipe usage with equal write-read size

Code sample 29 shows the same code where the producer does now sleep for a shorter timer and instead of writing 100 blocks at once, it writes only 20 blocks. The consumer thread still reads 100 blocks. This will result in the producer to be activated five times before sufficient data is written for the consumer thread to become active.

```
 1  TAVIX_THREAD_REGULAR producerThread(void* p)
 2  {
 3    char writeBuffer[100];                    // Local buffer of 100 bytes
 4    AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
 5    pipe = avixPipe_Create("pipe", 100, 1, NULL, NULL);
 6
 7    while(1)
 8    {
 9      avixThread_Sleep(AVIX_DELAY_US(200));   // Pause before next burst
10      avixPipe_Write(pipe, writeBuffer, 20);  // Write the 20 block buffer to pipe
11    }
12  }
13  TAVIX_THREAD_REGULAR consumerThread(void* p)
14  {
15    char readBuffer[100];                     // Local buffer of 100 bytes
16    AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
17    pipe = avixPipe_Get("pipe");
18
19    while(1)
20    {
21      avixPipe_Read(pipe, readBuffer, 100);   // Read 100 blocks from pipe
22      // Process the data just read.
23    }
24  }
```

Code sample 29: Synchronous pipe usage with different write-read size

Figure 13 shows the activation diagram of the producer and the consumer thread again. As can be seen, the producer runs five times before sufficient data is written for the consumer to become active.

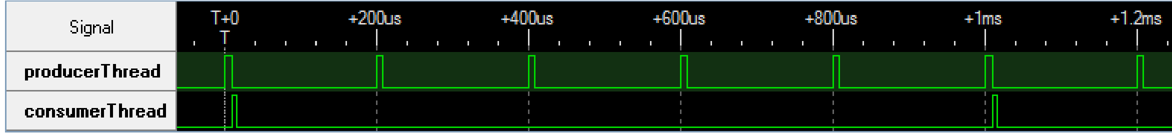


Figure 13: Synchronous pipe usage with different write-read size

Asynchronous data transfers

Asynchronous data transfers allow more flexibility than synchronous data transfers. Asynchronous data transfers are offered through functions `avixPipe_WriteAsync` and `avixPipe_ReadAsync`. When an asynchronous request cannot finish immediately, a request descriptor is allocated and the request is set pending as described before The difference with synchronous requests is that no relation exists between the state of the requesting thread and the state of the request. Asynchronous write and read functions always return immediately regardless the request state, the requesting thread will not enter the 'Blocked' state.

How then does a thread making an asynchronous request determine whether the request is ready or not?

For this purpose asynchronous requests receive two optional parameters, an event group id and one or more event flags. When the asynchronous request results in a pending request, the specified event flags are cleared. Once the request is finished, the specified event flags are set. A thread can wait at any given moment for the specified flags to be set which means that the request is finished and all specified data is processed.

Code sample 30 shows the same example as shown in Code sample 28 with the difference that the consumer thread uses an asynchronous read operation now.

```
1  TAVIX_THREAD_REGULAR producerThread(void* p)
2  {
3    char writeBuffer[100];                           // Local buffer of 100 bytes
4    AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
5    pipe = avixPipe_Create("pipe", 100, 1, NULL, NULL);
6
7    while(1)
8    {
9      avixThread_Sleep(AVIX_DELAY_MS(10));           // Pause before next burst
10     avixPipe_Write(pipe, writeBuffer, 100);        // Write the 100 block buffer to pipe
11   }
12 }
13 TAVIX_THREAD_REGULAR consumerThread(void* p)
14 {
15   char readBuffer[100];                            // Local buffer of 100 bytes
16   AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
17   pipe = avixPipe_Get("pipe");
18
19   while(1)
20   {
21     avixPipe_ReadAsync
22       (  pipe,
23          readBuffer,
24          100,
25          avixThread_GetIdCurrent().asEventId,      // Use the thread local event group
26          AVIX_EF(0),                               // Set flag 0 when read is finished
27          NULL  );                                  // Don't use a request id.
28
29     // Continue processing regardless whether the requested amount of data is
30     // present or not.
31
32     avixEventGroup_Wait                            // Wait for flags
33       (  avixThread_GetIdCurrent().asEventId,      // Use the thread local event group
34          AVIX_EF(0),                               // Wait for flag 0 to be set
35          AVIX_EVENT_GROUP_ALL,                     // Don't care since single flag
36          AVIX_EF_NONE,                             // Don't clear flags on start of wait
37          AVIX_EF(0)  );                            // Clear Flag 0 when done
38
39     // Process the data just read.
40     …;
41   }
42 }
```

Code sample 30: Asynchronous pipe usage

On line 25 and 26 of Code sample 30, an event group id and an event flag are passed as parameters to `avixPipe_ReadAsync`. After having made the request, the thread continues with whatever functionality required until at line 32 it decides to wait for the request to have finished. This is done by passing the same event group id and event flag as parameters to `avixEventGroup_Wait`.

Abort asynchronous requests and obtain the request status

For the status of a request to be obtained or the request to be aborted use can be made of a status descriptor. A status descriptor is a variable of type `tavixPipeAsyncRegStatus`, defined by the client code. As an optional fifth parameter to an asynchronous write or read operation, the address of a status descriptor variable may be passed. Doing so, when the asynchronous operation returns, this variable contains information about the progress of the asynchronous request. By passing the address of this variable to function `avixPipe_AbortAsyncRequest` the request can be aborted. Passing the address of this variable to function `avixPipe_GetStatusAsyncRequest` the status of the request can be obtained.

As described before, synchronous requests may be used with a time-out to prevent the thread from being in the 'Blocked' state too long. For asynchronous requests, this is no option, no use can be made of a time-out since the thread does not enter the 'Blocked' state. Still for exceptional situations, it may be required to 'cancel' a pending request. For this purpose function `avixPipe_AbortAsyncRequest` Is offered. Using this function, a pending request is removed from the pipe. The request to abort is identified by passing the address of a status descriptor which before is passed to an asynchronous write or read operation.

The result value of `avixPipe_AbortAsyncRequest` is the request status which may identify the request to be aborted but also to be finished. The request is only aborted when the moment `avixPipe_AbortAsyncRequest` executes, the status is pending. It may however well be the request was already finished or even finished just before `avixPipe_AbortAsyncRequest` actually started executing, in which case the request will not be aborted since it is already finished. This function may receive as an optional parameter the address of a variable where the number of blocks processed is returned. When an asynchronous request is aborted and an event group id and flag are passed to the request, the specified flag is set to identify the request to be ready.

Besides aborting a pending request, a status descriptor can also be used to obtain the current status of the asynchronous request. To obtain the status of a asynchronous request, the address of the request status variable is passed to function `avixPipe_GetStatusAsyncRequest`. The result value of this function is the request status. Optionally the address of a variable can be passed where the number of blocks processed until then is returned.

An asynchronous request is always in one of three possible states:

- *Finished*: All requested data is transferred. The pipe does not hold any information concerning the request in its internal bookkeeping.
- *Pending*: The requested data is not processed (yet). The pipe internal bookkeeping contains information describing the status of the request. Based on this information, subsequent pipe read or write operations may lead to additional data for the request being processed. Once all requested data is processed, the request is set finished.
- *Aborted*: Not all requested data was processed, leading to the request being set pending. While pending, the request was aborted. No more data will be processed and the request information in the pipe internal bookkeeping is removed.

Asynchronous requests offer much more flexibility than synchronous requests but require more discipline from the designer to be dealt with correctly. In general, while a request is pending, the pipe mechanism may access the event group id/flag, the buffer and the request id variable at unpredictable moments.

After making an asynchronous request until the moment the request is no longer pending:

1. Do not use the event group-flag combination for other purposes. The specified flag will be set by the pipe mechanism the moment the request is finished or aborted.

2. Do not access the buffer passed to the request. For a read request the pipe mechanism will write to this buffer and for a write request, the pipe mechanism will read from this buffer.

3. When using a request id variable, only access this variable to pass its address to either `avixPipe_GetStatusAsyncRequest` or `avixPipe_AbortAsyncRequest`. Do not pass the address of a request id variable to another asynchronous request while the request it was used for most recent is still pending.

Usage of `avixPipe_GetStatusAsyncRequest` and `avixPipe_AbortAsyncRequest` is shown in Code sample 31.

```
1  TAVIX_THREAD_REGULAR consumerThread(void* p)
2  {
3    AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
4    unsigned char           readBuffer[100];       // Local buffer of 100 bytes
5    tavixPipeAsyncReqId     requestId;
6    tavixPipeAsyncRegStatus requestStatus;
7    unsigned int            nrBlocks;
8
9    pipe = avixPipe_Get("pipe");
10
11   while(1)
12   {
13     avixPipe_ReadAsync
14       (  pipe,
15          readBuffer,
16          100,
17          avixThread_GetIdCurrent().asEventId,    // Use the thread local event group
18          AVIX_EF(0),                             // Set flag 0 when read is finished
19          &requestId  );                          // Use a request id
20
21     // Continue processing regardless whether the requested amount of data is
22     // present or not.
23
24     // Determine the current status of the request.
25     requestStatus = avixPipeGetStatusAsyncReq
26                     (  &requestId,               // Same variable as used on line 19
27                        &nrBlocks );
28
29     // Abort the request.
30     requestStatus = avixPipeAbortAsyncReq
31                     (  &requestId,               // Same variable as used on line 19
32                        &nrBlocks );
33   }
34 }
```

Code sample 31: Status and abort usage of asynchronous pipe requests

Combining synchronous and asynchronous requests

No restriction exists on combining synchronous and asynchronous requests on the same pipe. A thread may use the type of request best fitting the intended purpose.

Multiple reader and/or multiple writer threads

AVIX pipes allow multiple threads to concurrently read from and write to a pipe. When doing so, it is important to realize the potential consequence this may have on the ordering of the data.

When not all data of a request is processed at once, the request will be set pending. At any given moment the request of multiple threads may be pending where the request descriptors are ordered according the priority of the thread that made the request. Suppose a write request is partially processed before being set pending. This may for instance happen when a request is made for more blocks than fit in the pipe buffer. Next a write request is made from a thread having a higher priority. This will be set pending too but will be placed ahead of the already pending request that was partially processed. When read requests create space in the pipe buffer now, the second request will be processed before the first request leading to the data of the two requests to be mixed. The same may happen when multiple read requests are pending. A higher priority read request may receive data ahead of a pending, partially processed read request made by a lower priority thread.

Whether or not this is problematic depends on the application. When data blocks have no relation to each other no problem occurs. If however data blocks have a relation to each other it is very important to be aware of these scenarios.

AVIX pipes are block oriented. This means the basic size of an element written to or read from a pipe is user defined and not restricted to a single byte or integer as is the case with competing products. AVIX guarantees that a single block is always written or read as an atomic entity. The potential data fragmentation described above will only occur on data block boundaries. This allows for instance to use pipes for the transport of ‘C’ struct data elements where it is guaranteed a struct is always transferred as an atomic entity. This has a positive influence on the complexity of the application since no effort has to be spent to deal with partially transferred data blocks.

Pipe buffer size

A pipe contains an internal buffer, the size of which is specified when creating the pipe. The size of the pipe buffer influences the behavior or threads writing to the pipe. Since threads operate independently from each other, a thread writing to a pipe may never assume that other threads have already issued a read request or vice versa.

Suppose no read requests are pending and threads write to the pipe. When the pipe buffer is too small to contain all data, the write request will be set pending. The thread using the write functionality has to deal with this. It has to wait for the specified event flag to be set to know the write request is finished and the thread local buffer containing the data to write can be reused. In case of a synchronous write request, the calling thread will enter the ‘Blocked’ state which may be undesirable.

To prevent this, the size of the pipe buffer must be set such that under all circumstances, write requests can have their data copied to the pipe buffer without the write request being set pending.

Keeping this in mind, when using pipes between threads, the pipe buffer may have whatever size desired, even zero(0). When using a pipe buffer of size zero(0), write requests while no read requests are pending are guaranteed to be set pending.

The above being true for pipes being used between threads, it is not when pipes are used by DIH’s or ISR’s. Details are dealt with in the next sections.

Pipes used between threads and DIH's

Pipes may be read and written by DIH's. Compared to pipe usage between threads, a number of restrictions exist which are dealt with in this section.

When using a pipe from DIH's, the same pipe may also be used by threads. Under no circumstance however may such a pipe be used by an ISR. Using pipes concurrently from DIH's and ISR's is not allowed and doing so will lead to an instable system. AVIX has no way to check this rule and it is entirely the user's responsibility to implement the application such that this rule is obeyed. Figure 14 shows in **green** which transfer directions are allowed to be used concurrently, and in **red** which are not.

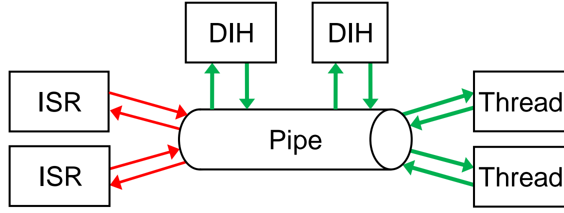


Figure 14: Using pipes from DIH's

The main difference between DIH's and threads is that DIH's run to completion. A DIH cannot enter a 'Blocked' state.

When used from a DIH, the scenarios specified in section 'Read-write scenarios' on page 73 are applicable with the difference that if not all requested data can be transferred, no request descriptor will be added to the pipe internal bookkeeping. The request will not be set pending.

When using synchronous pipe operations from a DIH, these operations will always return immediately, regardless the amount of data actually transferred. The actual amount of data transferred is returned as the function result of `avixPipe_Write` or `avixPipe_Read` and can be less than the amount specified.

When using asynchronous pipe operations from a DIH it is advised always to pass a reference to a request id variable. Upon return of the function this request id can be passed to function `avixPipe_GetStatusAsyncRequest` to obtain the number of blocks actually transferred. When not all specified blocks are transferred, the status returned by this function indicates the request is aborted (`PIPE_ASYNCREQ_ABORTED`). When on the other hand all requested blocks are transferred the status returned indicates the request to be finished (`PIPE_ASYNCREQ_FINISHED`).

In general, DIH's are meant to be used as an 'interface' from ISR's to threads. Like ISR's, DIH's should be very efficient and consume the smallest possible amount of execution time. For this reason, using pipes from DIH's has limited use and most likely when being tempted to use pipes from DIH's, it is probably better to use pipes from ISR's directly, which is dealt with in a coming section.

Pipes used between threads and ISR's

Pipes may concurrently be used by ISR's and threads. No special function call is needed to enable a pipe to be used by an ISR. Just create the pipe and an ISR may write to it or read from it. Pipe access from ISR's is offered by two dedicated functions, `avixPipe_WriteFromISR` and `avixPipe_ReadFromISR`. It is very important that ISR's only use these functions and not the regular read and write functions which are intended to be used by threads or DIH's only.

When pipes are used from ISR's a number of important rules must be obeyed:

- When using a pipe from an ISR, under no circumstance may a DIH make use of the same pipe.
- When a pipe is written by an ISR, threads may only read from that pipe. Under no circumstance may a thread write to a pipe when this is also written by an ISR.
- When a pipe is read by an ISR, threads may only write to that pipe. Under no circumstance may a thread read from a pipe when this is also read by an ISR.
- When using a pipe from an ISR, either to read or to write, the ISR may not be interrupted by another ISR accessing the same pipe. In principle a pipe may only be used from a single ISR. This rule is relaxed for multiple ISR's at the same interrupt priority level. ISR's at the same interrupt priority level will not interrupt each other and run sequentially. Under the strict condition the ISR's have the same priority, multiple ISR's may access the same pipe.

AVIX has no way to check the above mentioned rules and it is entirely the user's responsibility to implement the application such that these rules are obeyed.

Figure 14 shows in **green** which transfer directions are allowed to be used concurrently, and in **red** which are not.

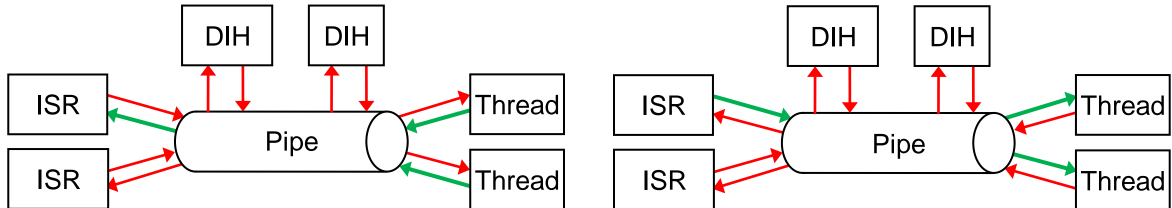


Figure 15: Using pipes from ISR's

Pipe ISR functions behave different from pipe thread functions. Pipe ISR functions always access the pipe buffer, regardless whether thread requests are pending or not. In contradiction to pipes used between threads, the pipe buffer size of a pipe used by ISR's may not be zero(0).

ISR's cannot block, once activated, an ISR runs to completion. Therefore when accessing a pipe from an ISR, only as many blocks will be processed as the current state of the pipe buffer allows. Attempting to write more blocks than the pipe buffer has available empty results in only part of the data to be actually written. Likewise, attempting to read more blocks than the pipe buffer currently contains results in only part of the desired number of blocks to be actually read.

The number of blocks actually processed is returned by functions `avixPipe_WriteFromISR` and `avixPipe_ReadFromISR`. Check this value to determine how many blocks have been processed.

The behavior on the 'thread side' of the pipe is the same as described before. Both synchronous and asynchronous functions may be used. A thread write request to a pipe where the pipe buffer does not contain enough empty space for all desired data to be written results in the write request to be set pending. Likewise, a thread read request from a pipe where the pipe buffer does not

contain enough data for the read request to finish will result in the read request to be set pending. In these cases, when using synchronous requests, the thread enters the 'Blocked' state.

Pipe buffer size and performance tuning

Pipe ISR functions always only access the pipe buffer. When using `avixPipe_WriteFromISR`, data is written to the pipe buffer regardless if thread read requests are pending or not. Even in the situation a thread read request is pending, `avixPipe_WriteFromISR` will only be able to write as many blocks as there is empty space in the pipe buffer. `avixPipe_WriteFromISR` will not directly write to the buffer of the pending thread read request.

Likewise when reading a pipe from an ISR, only as many blocks as currently present in the pipe buffer will be read, even when a write request is pending. In the context of the ISR function call, the buffer of the pending write request will not be accessed.

Reason for this behavior is that ISR's and threads operate fully asynchronous from each other and ISR's cannot block. This asks for a buffer that can be used from the ISR, regardless the current state of threads accessing the other end of the pipe which is exactly what the pipe buffer is meant for. Second, by always using the pipe buffer from the ISR functions, the implementation can be much more efficient than would be the case buffers of pending requests had to be managed. This allows for very high interrupt rates.

When using a pipe from an ISR, the effect these operations have on the 'thread side' of the pipe are again the same as in the case where the pipe is used by threads on both sides of the pipe. When a thread read request is pending, and the ISR writes sufficient data to satisfy that request, the pending thread request will be set finished. In case the thread request was synchronous, the thread is in the 'Blocked' state and will enter the 'Ready' state. When a thread write request is pending and the ISR reads sufficient data from the pipe buffer to be filled by the pending write request, once all data of the write request is transferred to the pipe buffer, the writing thread will enter the 'Ready' state.

To obtain the highest possible performance, managing pending requests and the related thread status is not done by the ISR itself. Instead, when required, the ISR 'spawns' an AVIX internal handler which runs at scheduler priority once the ISR is ready. When an ISR writes to the pipe buffer and a read request is pending, the ISR might 'spawn' a handler function taking care of transferring the data just written by the ISR to the buffer of the pending read request. Likewise, when empty space is created in the pipe buffer by an ISR reading and a write request is pending, the ISR might 'spawn' a handler function taking care of transferring the data from the pending write request to the pipe buffer.

Note the word 'might' in the above description. It would not make sense to spawn a handler on every ISR activation since this would have a negative effect on system performance and the maximum interrupt speed obtainable.

Threads and ISR's operate in a different timing domain. ISR's may operate in the microsecond domain while threads operate in the millisecond domain. To 'connect' these different domains and transfer data between them is exactly what pipes are meant for. Typically an ISR will process single elements like a single UART character or a single ADC sample while a thread will operate on multiple of these data elements at once. It is likely (and for performance reasons probably desirable) to activate a thread once for multiple ISR activations.

The mentioned handler will be 'spawned' by the ISR when the pipe buffer contains sufficient bytes to satisfy the pending read request. This may result in the pipe buffer filling up to being almost entirely full before the handler takes care of transferring the data to the buffer of the read request. The size of the pipe buffer must be sufficient to hold data corresponding to the largest read request. It shall be obvious this is undesirable. Not only would memory be wasted for the pipe buffer but also the risk of data loss exists since when the pipe buffer is full and the handler is not

activated fast enough to transfer this data out of the pipe buffer a situation may occur where an ISR cannot write its data.

To solve this a second mechanism exists. When an ISR writes to a pipe and a thread read request is pending, when the pipe buffer is filled for 50% or more, the ISR 'spawns' a handler to transfer the data to the buffer of the pending thread read request. This happens even if the pending thread read request requires more data than currently available. Doing so, the pipe buffer is emptied by the handler and since the amount of data is not sufficient yet, the thread read request remains pending. The pipe buffer however does offer additional empty space for subsequent ISR write requests to be used successfully.

The same happens when the ISR reads from the pipe and a thread write request is pending. The handler is 'spawned' when either the entire pending thread write request can be handled or when the pipe buffer becomes empty for 50% or more. This guarantees always sufficient data to be present in the pipe buffer for the ISR read operation to execute correctly.

It may be clear the handler does not come for free and consumes CPU cycles. When using pipes to communicate between ISR's and threads, it is important to tune the application so the right balance is reached between CPU load (ISR, handler) and memory usage (pipe buffer). Using a larger pipe buffer lowers the CPU load and vice versa.

In order to facilitate this tuning, a function is offered (`avixPipe_SetHandlerTracePort`) allowing one of the MCU's I/O pins to be assigned to the pipe handler. This allows the pipe handler activation to be shown on a logic analyzer providing direct real time insight in the tuning process. This function works according the same principle as Thread Activation Tracing, described in §6.5.4.

The above is illustrated with a sample[16]. This sample is based on a thread reading 10 integers from a pipe. The other end of the pipe is written by a timer based ISR where every time the ISR is active, a single integer is written to the pipe. So 10 interrupt activations are required to fulfil the thread read request. Function `avixPipe_SetHandlerTracePort` is used to show the pipe handler activation. Activation of the reading thread is shown using regular Thread Activation Tracing and finally activation of the ISR is shown by explicitly toggling an I/O port from the ISR.

Using a low interrupt rate of 10μs, the pipe buffer can be small and is created to hold only two integers. The effect this has on the handler activation is shown Figure 16.

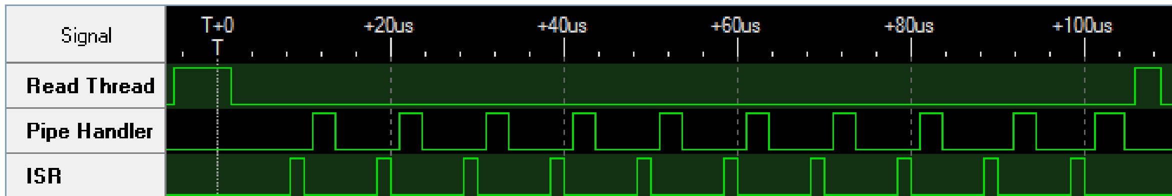


Figure 16: Pipe based ISR Thread communication, 10μs interrupt rate

First the read thread becomes active, attempting to read 10 integers. The pipe buffer is empty so the thread enters the 'Blocked' state. Next interrupts start to fire every 10μs. Since the pipe buffer is only two integers in size, every ISR write will fill the pipe buffer for 50%. As a result on every ISR write, the pipe handler is activated to write the data to the thread request read buffer. On the 10th activation, the read request is ready and the read thread is activated since all data is available.

Since for every ISR activation, the pipe handler is activated, CPU load is high in this situation and it shall be obvious the maximum ISR frequency is limited by this.

[16] This example is based on AVIX for PIC24-dsPIC used on a Microchip dsPIC33EP512MU810 microcontroller running at a clock of 70MHz. Performance with other microcontrollers will vary because of architectural differences or running at other clock speeds.

To deal with this, the handler must be activated less frequent. This is accomplished by increasing the pipe buffer size to 10 integers (blocks). Now only after the ISR has written 5 integers, the pipe buffer is filled for 50% and the pipe handler is activated to transfer the data to the thread request buffer. The result of this is shown in Figure 17 where it is shown that CPU load is much lower because the handler is activated less frequent.

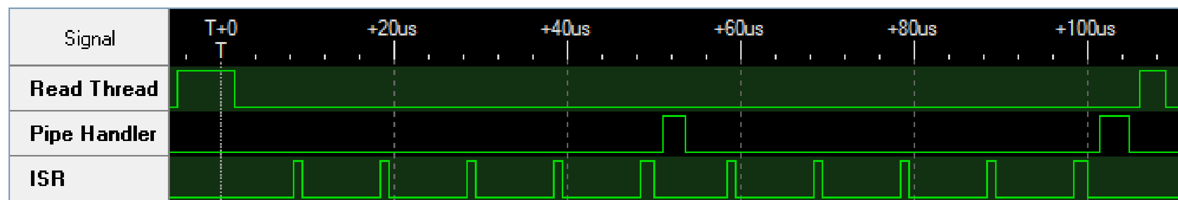


Figure 17: Pipe based ISR Thread communication, 10μs interrupt rate

This justifies the assumption that in this scenario the interrupt rate can be increased. The sample shown in Figure 18 shows the result with an interrupt rate of once every 4μs. The pipe buffer size is still 10 integers so again, the pipe handler is activated once every 5 writes.

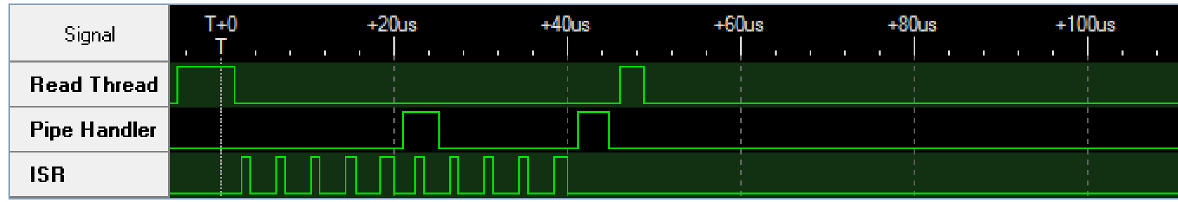


Figure 18: Pipe based ISR Thread communication, 4μs interrupt rate

As shown in Figure 18, interrupts start to overlap with the activation of the pipe handler now. Although this is no problem at all, it does imply that the pipe buffer will be filled with another integer before the handler is able to start reading it.

For the highest possible interrupt rate it is best to have the smallest number of handler activations. This is accomplished by creating the pipe with a buffer size sufficient to hold twice the number of blocks that is ever passed in a single thread write or read request. This scenario is shown in Figure 19. Here the interrupt fires once every 2μs and as shown, the system is able to deal with this without any problem. Do realize Figure 19 is based on an interrupt frequency of 500KHz!

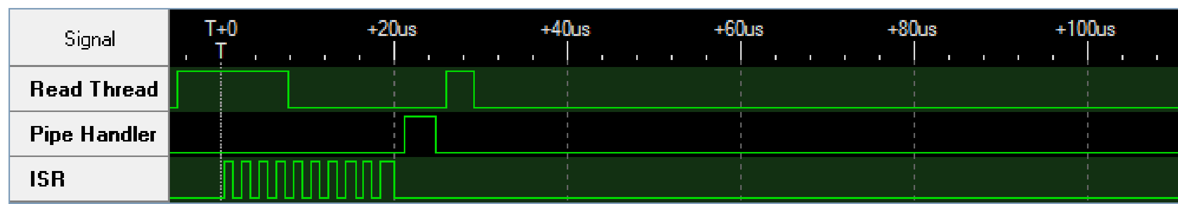


Figure 19: Pipe based ISR Thread communication, 2μs interrupt rate

The general rule to follow is, the higher the interrupt rate, the larger the pipe buffer must be.

The scenario illustrated above is based on the ISR to only start writing data to the pipe buffer when at that moment a read request is already pending. In a typical application, this will not always be the case. ISR's and threads are fully decoupled from each other and it may well be that the ISR writes data to the pipe buffer where the destination thread will only execute the read request after some time. In this case, it makes no sense for the ISR to activate the handler since there is no request buffer available to transfer the data to. The only option is to make the pipe buffer large enough to deal with the periods in time it can only be filled but is not emptied yet.

Device control

Pipe access from ISR's offers a very powerful mechanism enabling integration of interrupt driven I/O in an AVIX based application.

When using hardware devices like UARTS and the like, an application typically will need to implement a mechanism to control device activity. When using a UART to send data, the UART interrupt only needs to be enabled when data is available. While no data needs to be sent, the interrupt must be disabled. Pipes offer a mechanism to control device activity in such a way that the fact the data is transferred under control of an ISR is fully transparent to the application. All the application needs to do is write the data to a pipe from a thread and the pipe together with a user provided device control function takes care of the rest.

To allow the creation of such a device control function, pipes offer a powerful callback mechanism. This callback mechanism allows for a callback function to be related to a pipe when creating the pipe. As explained before, ISR's only access the pipe buffer and not the buffers of pending thread requests. For this reason, it is important for the ISR to be able to determine the status of the pipe buffer and this is exactly what the callback is used for. Based on the pipe functions used by a thread and the result of this operation on the status of the pipe buffer, this function is called with flags identifying the actions taken by the application such that the control function can take the required action towards the hardware device.

Pipe callback functions are only called when the pipe is accessed by a thread. Accessing a pipe from an ISR or a DIH will not activate the callback function.

A callback function is registered with a pipe when creating it. An example of this is shown in Code sample 32.

```
 1 // Pipe callback function
 2 void myCallback(tavixPipeEvent event, int nrBlocks, void* userData)
 3 {
 4   switch(event)
 5   {
 6     case PIPEINFO_PIPE_DATA_WRITTEN:
 7       // Actions to take when a thread writes data to the pipe.
 8       break;
 9
10     case PIPEINFO_READ_FROM_EMPTY_PIPE:
11       // Actions to take when a thread reads from the empty pipe.
12       break;
13
14     case PIPEINFO_DEVICE_STOP_REQUESTED:
15       // Actions to take when the device must be stopped.
16       break;
17 };
18
19 …;
20
21 // Create a pipe with a device callback function
22 AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipe);
23
24 pipe = avixPipe_Create("pipe", 100, 1, myCallback, NULL);
25
26 …;
```

Code sample 32: How to use a pipe device callback function

Pipe callback functions are only useful when using a pipe to exchange data between an ISR and a thread. When using pipes from threads and/or DIH's only, callbacks have no use and the information provided to these functions as parameters is not consistent. When using pipes from DIH's and/or threads only, pass NULL for the callback parameter when creating a pipe.

The parameters passed to a pipe callback function have the following meaning:

- `event`: This parameter identifies the reason why the callback is called and can have one of three possible values:

    - `PIPEINFO_PIPE_DATA_WRITTEN`: Data is transferred to the pipe buffer because of a thread originating synchronous or asynchronous write operation and thus data is guaranteed to be present in the pipe buffer. Every time data is actually transferred to the pipe buffer, the callback is called with this parameter value. A single pipe write operation may result in multiple transfer operations to the pipe buffer. The callback is called for each of these transfer operations. With each call, the number of blocks actually written is passed to the callback in parameter `nrBlocks`. When the callback is called with this flag, it can enable an interrupt of a device where the device ISR reads data from the pipe in order to send it out by using the device hardware facilities.

    - `PIPEINFO_READ_FROM_EMPTY_PIPE`: Flag identifying a thread has read from an empty pipe buffer using either a synchronous or an asynchronous read operations. The callback is not called on every read operation a thread performs on a pipe but only in case the read operation is executed when the pipe buffer contains no data. When called with this flag, the value of parameter `nrBlocks` is undefined and should not be used. When the callback is called with this flag, it can for instance enable a device that after some time generates an interrupt to identify it has data available. An example is an ADC which is started on the callback called with this flag and when ready writes its data to the pipe.

    - `PIPEINFO_DEVICE_STOP_REQUESTED`: The callback is activated with this flag when an ISR calls function `avixPipe_StopDeviceFromISR`. This flag only exists to allow all device activity to be controlled from the callback function. When called with this flag, the value of parameter `nrBlocks` is equal to the second parameter passed to `avixPipe_StopDeviceFromISR`.

- `nrBlocks`: See above.

- `userData`: Address of a user defined data structure. The address of this data structure is passed to the pipe upon creation. This data structure can be used for application specific bookkeeping data needed for the device related to the pipe.

To summarize, the pipe callback function is called when a write results in actually transferring data to the pipe buffer or when a read results in an attempt to read data from an empty pipe buffer. The callback is not activated because of ISR originating write or read operations (`avixPipe_WriteFromISR` or `avixPipe_ReadFromISR`). Finally, when an ISR explicitly wants to disable the device it belongs to, use can be made of `avixPipe_StopDeviceFromISR`, also resulting in an activation of the callback.

The implementation of a pipe callback function will typically use device Special Function Registers (SFR's) to control the device, enable/disable device interrupts and so on. When using AVIX functions from a pipe callback, these are subject to the same restrictions as those that exist for an ISR. This means that under no circumstance AVIX functions may be used directly from a pipe callback with the exception of those function that are allowed to be used from an ISR by design.

► *When using AVIX functions from within a pipe callback function make sure to only use functions allowed to be called from an ISR. Under no circumstance make calls to other AVIX functions. Applicable functions can be recognized by their name ending in …`FromISR`.*

Important to realize is that a pipe callback function is called both in a thread context when a thread writes or reads from the pipe and from an ISR context when the ISR calls function `avixPipe_StopDeviceFromISR`. This implies the operations done inside the callback need to be ‘interrupt safe’ which is the user’s responsibility.

When for instance updating a counter variable when called from a thread context while the same counter is updated by interrupt code, the counter operation must be atomic to prevent a read-modify-write problem. Special care must be paid to bit operations on SFR’s since they easily introduce read-modify-write problems where modifying one bit in an SFR unintentionally changes other bits in the same SFR. Always make sure SFR operations in an RTOS based application are atomic.

► *Make sure the operations done within a pipe callback are resistant against concurrency aspects related to ISR code and non-ISR code accessing the same resources.*

A ‘real world ’ example using an interrupt based device sending data delivered to it through a pipe is shown in Code sample 33. When a thread writes data to a pipe resulting in the data being transferred to the pipe buffer, the pipe callback function is called. Here the transmit interrupt is enabled. Next the ISR reads data from the pipe and writes it to the UART data buffer. When the transmit buffer is empty, another interrupt is generated and the above sequence repeats. Every time the ISR is activated, `avixPipe_StopDeviceFromISR` is called with the number of bytes read from the pipe. When no more bytes are present (count is 0), the transmit interrupt is disabled.

```
1  // ISR based on AVIX macro resulting in use of system stack. This is Microchip
2  // PIC24-dsPIC specific code, for other hardware platforms the syntax may be different
3  avixDeclareISR(_U1TXInterrupt, no_auto_psv)
4  {
5    unsigned char a[4];
6    int           i;
7    int           nrChars;
8
9    // Read from pipe and write to four byte deep UART buffer.
10   nrChars = avixPipe_ReadFromISR(pipeTransmit, &a[0], 4);
11   for (i = 0; i < nrChars; i++) {
12     WriteUART1(a[i]);
13   }
14
15   // If pipe is empty, stop transmitting, nrChars is 0, see callback for action
16   avixPipe_StopDeviceFromISR(pipeTransmit, nrChars);
17 }
18
19 // Pipe callback function. The SFR manipulating code in this function is based on the
20 // Microchip PIC24-dsPIC MCU syntax. For other hardware platforms code is different!
21 void uartSendPipeCallback(tavixPipeEvent pipeEvent, int nrBytes, void* pUserData)
22 {
23   switch(pipeEvent)
24   {
25     case PIPEINFO_PIPE_DATA_WRITTEN:          // When a thread writes data to the
26       IEC0bits.U1TXIE = 1;                    // pipe the transmit interrupt is
27       break;                                  // enabled.
28
29     case PIPEINFO_DEVICE_STOP_REQUESTED:      // Case when called from the ISR
30       if (nrBytes) {                          // through avixPipe_StopDeviceFromISR
31         IFS0bits.U1TXIF = 0;                  // When there are bytes read from
32       }                                       // the pipe just clear the interrupt
33       else {                                  // flag. When pipe is empty, disable
34         IEC0bits.U1TXIE = 0;                  // transmit interrupt but leave
35       }                                       // interrupt flag unchanged.
36       break;
37
38     default:
39       break;
40   }
41 }
42
43 // Creation of the pipe, 50 blocks of one byte each
44 …;
45 AVIX_OBJECT_ID_DEFINE(tavixPipeId, pipeTransmit);
46
47 pipeTransmit = avixPipe_Create(NULL, 50, 1, uartSendPipeCallback, NULL);
48 …;
49
50 // Usage of the pipe
51 …;
52 char* p = "Demo_ABCDEF";
53 avixPipe_Write(pipeTransmit, p, strlen(p));
```

Code sample 33: How to use a pipe device callback function to send UART data

Figure 20 shows the corresponding Thread Activation Diagram. From top to bottom this diagram shows activation of the thread writing to the pipe, activation of the ISR (just by setting/clearing an I/O pin) and the resulting UART data. As can be seen here the processor load is quite small and at thread level the code is very easy, just write to a pipe.

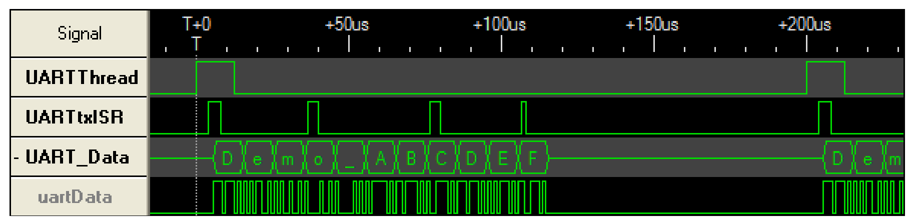


Figure 20: Pipe based UART transmission Thread Activation diagram

Block approach and transfer by value

When exchanging data using a pipe the data read by one thread must be equal to the data written by another thread. Although this might seem obvious, many competing products only guarantee this under strict conditions and rules one must obey to.

The AVIX pipe implementation does what is considered obvious without confronting the user with all kinds of details one has to pay attention to get things working. This is accomplished by two properties. AVIX pipes are block oriented meaning the size of an individual data element (a data block) is user specified. Furthermore pipes transfer data by value instead of reference. The advantages of these properties are explained in the coming sections.

Block approach

When writing an integer value to a pipe one may expect the same integer value to be read. The same is true when writing a struct. As a user you may define structs of whatever size and there is no reason why structs could not be transferred using a pipe. Here too one may expect when writing a struct to a pipe, the reader thread reads exactly the same value that was written. In other words, the basic data elements transferred through a pipe are treated as indivisible atomic entities.

As obvious as it may seem, this is not always the case. Many competing products only allow pipes with basic data elements having the size of a byte or an integer. This means these products only guarantee data elements of this basic size to be treated as an atomic entity. Of course these products still allow transfer of structs since these are nothing more than a sequence of bytes but this introduces a major flaw.

Suppose a pipe is defined with the pipe buffer having a size of 100 bytes and this pipe is used to transfer structs with a size of 30 bytes. Next, four structs are written. Three of these structs fit in the pipe entirely but of the fourth struct only the first 10 bytes can be written The writing thread will enter the ‘Blocked’ state until the pipe has sufficient space to finish writing of the remainder of the fourth struct. At this moment a higher priority thread wants to write to the same pipe. Since the pipe is full, this higher priority thread will also enter the ‘Blocked’ state but because of its higher thread priority, when data can be written to the pipe again, this thread will get precedence over the first thread. When allowed to write again, this second thread will start writing immediately behind the 10 bytes of the fourth 30 byte struct written before. As a result the data of the fourth struct is corrupted! This problem occurs because the pipe treats the data as a sequence of bytes and has no notion of the relation the individual bytes have to each other.

AVIX solves this problem by not only allowing a pipe to contain either byte or integer sized elements. AVIX pipes have a user defined block size and guarantee blocks are always transferred as an atomic entity, whatever the block size is.

► *The block oriented approach offered by AVIX pipes guarantee data integrity and prevents data corruption. As obvious as it sounds, a guarantee most competing products cannot provide.*

When transferring arrays using pipes, still multiple writer threads may cause individual elements of the array to be intermingled. Each individual element (a data block) however is guaranteed to be transferred as an atomic entity.

Transfer by value

Another problem with many competing products is that data is not transferred by value but by reference instead. Often this is considered a solution to the previous issue since a pointer can be written as an atomic entity. This approach does however introduce an entire new class of problems that again need to be solved by the user.

Suppose a thread generates data which has to be send using a pipe. This thread typically will have some private buffer that is filled and next a pointer to this buffer is send through the pipe. The receiver will use this pointer to access the data present in the sender’s buffer. The problem here is

that the sending thread will continue and reuse this buffer. But how does it know that the content of the buffer is already processed by the receiver? How can the sending thread be sure when filling the buffer again it will not overwrite data that still has to be processed by the receiver?

Often two possible solutions are proposed to address this problem. The first is that the receiver informs the sender when the buffer has been processed. This works, apart from the fact that the threads are heavily synchronized then, the sender has to wait for the receiver to be ready with the buffer. This results in the creation of some custom multiple buffer mechanism with all required management code surrounding it.

The second solution is to use buffers coming from some dynamic memory pool, the sender allocates the buffer, sends a pointer and the receiver frees the buffer again. In most cases this will not work when communicating with an ISR since ISR's in most products are not allowed to allocate and free memory.

AVIX offers a different solution; pipe transfers are by value. Data written to a pipe is copied, so the moment the write operation is ready, the thread may reuse its local buffer. No explicit synchronization is required since this is implicitly built into the mechanism.

Transfer by value is by far superior to transfer by reference since transfer by value can be used as transfer by reference might one still wants to use this. In this case a data block is just a pointer. When using this approach AVIX allows to use dynamically allocated buffers under all circumstances since AVIX memory buffers can be allocated and freed both by a thread and by an ISR and thus does not suffer from the mentioned disadvantage.

► *The AVIX 'transfer by value' approach with pipes leads to an easy and comprehensible code structure without being forced to create all kinds of custom mechanisms to manage buffers. Compared to the 'transfer by reference' approach there are only advantages since the 'transfer by value' approach offered by AVIX can always be used to implement the 'transfer by reference' approach.*

6.6.7 Memory Pool Services

The purpose of a memory pool is to allow threads to use memory only when needed and share this memory with other threads such that the total memory consumption decreases.

Memory is one of the most important resources of a real time system. Most real time systems depend heavily on memory that can be allocated and freed again which allows for reuse of memory. Reuse is needed while memory often is a scarce resource.

In general, real time systems cannot make use of the 'C' runtime system provided heap. A number of problems prevent this. First timing of the heap is not deterministic. Second, the heap cannot be used from multiple threads. Finally heap fragmentation may render the heap unusable. There are however ways to still use the 'C' runtime heap which is described later in this chapter.

AVIX addresses these problems by offering memory management in the form of memory pools containing fixed size blocks. AVIX allows for the creation of multiple memory pools, each having blocks of a potentially different size so the application can use a memory block that fits best to its requirements. A memory pool is created using function `avixMemPool_Create`, specifying an optional name, the number of blocks in the memory pool and the size in bytes of an individual block.

A memory block is allocated using function `avixMemPool_Allocate`. When allocating a memory block, this is done from a specific pool. The allocation function returns the id of a memory block. When no longer required, the memory block is returned to the pool again using function `avixMemPool_Free`. The memory block is available then to be allocated again. When freeing the memory block, all that is needed is its id. The application does not need to 'remember' which pool the block was allocated from; AVIX determines the correct pool to return the memory block to, based on internal bookkeeping information. This makes application development easier since the application does not need to maintain this bookkeeping. A memory block may be freed from any thread DIH or ISR that has access to the memory block id and not necessarily from the thread DIH or ISR that allocated it.

► *Freeing a memory block for reuse only requires the id of the block. The application does not need to remember which memory pool the block is allocated from making the code structure less complex and easier to maintain and understand.*

AVIX is designed to be safe to use. This is for instance reflected by all resources being identified with type safe id's preventing them from being mixed. When using memory blocks, this level of safety is impossible to offer since memory must be accessed directly using 'C' level functionality. Still AVIX goes a long way in offering safety with memory blocks also. Just like any other AVIX resource, memory blocks are identified by an id. When returning a memory block to its pool, the block is identified by this id preventing a wrong value to be passed to the free function, a situation that can easily occur would the block be identified by just a pointer. When reading and writing a memory block, this id must be explicitly converted to a plain 'C' pointer using macro `AVIX_MEM_BLOCK_PTR`. This zero overhead construct forces the developer to be explicitly aware what he is doing and prevents errors from being made. Still, once the memory is accessed, AVIX has no way to determine whether memory blocks are being written outside their bounds and care must be taken not to do so since this may lead to unpredictable system behavior.

Memory pools are wait able objects. When attempting to allocate a memory block from an empty pool, the allocating thread enters the 'Blocked' state until another thread, DIH or ISR has freed a memory block belonging to that pool.

In contradiction with many competing products, AVIX allows memory blocks to be allocated and freed from ISR's using `avixMemPool_AllocateFromISR` and `avixMemPool_FreeFromISR`. When a thread must send data to some device, the thread allocates a memory block, fills it and sends the id of the memory block to an ISR by using a pipe. The ISR subsequently reads the data and once all data is processed, the ISR frees the block so it can be reused. Likewise and ISR can allocate a memory block and fill this with device data. Once all data is present, the id of the memory block is send to a thread. The thread reads the data and once all data is processed, frees the memory block so it can be reused.

The deterministic memory management offered by AVIX which is also allowed to be used directly from ISR's make it a user friendly, flexible and fast mechanism.

How to use memory pools

This section provides a number of samples illustrating how to use a memory pool.

Creating a memory pool

When creating a memory pool, specified are the number of memory blocks and the size in bytes of each memory block contained in that pool. The safest way to use a memory pool is to define a 'C' structure specifying the content of each block in the pool. Code sample 34 shows how to do this

```
 1  …;
 2  typedef struct                                  // The preferred way of determining
 3  {                                               // the size of blocks in a memory
 4    int valOne;                                   // pool is by defining a type so
 5    int valTwo;                                   // the size of the blocks is
 6    int valThree;                                 // automatically adjusted when
 7  } demoStruct1;                                  // changes are recompiled
 8
 9
10  void avixMain(void)
11  {
12   AVIX_OBJECT_ID_DEFINE(tavixMemPoolId, poolId);
13    …;
14    …;
15    // Create the memory pool where the blocksize is based on the predefined struct
16    //
17    poolId = avixMemPool_Create("PDEM", 10, sizeof(demoStruct1));
18    …;
19    …;
20  }
```

Code sample 34: How to create a memory pool

Allocating a memory block and accessing it

After being created, blocks can be allocated from a memory pool. The sample shown in Code sample 35 builds on Code sample 34. A thread seeks access to the memory pool by its name. Like any other AVIX kernel object, memory pool id's can be obtained by using a name given when it is created. The memory block allocated from the pool is identified by an id.

Essential is the macro shown in line 19. This macro converts the memory block id to a pointer of the desired type. In this example the pointer is dereferenced so the struct to write to the memory block can be assigned directly.

```
1  TAVIX_THREAD_REGULAR demoThread(void* p)
2  {
3    AVIX_OBJECT_ID_DEFINE(tavixMemPoolId,  memPool);
4    AVIX_OBJECT_ID_DEFINE(tavixMemBlockId, memBlock);
5    demoStruct1      testData;                    // Struct that will be copied to block
6
7    testData.valOne = 1;                          // Fill the struct with data
8    testData.valTwo = 2;
9    testData.ValThree = 3;
10
11   // Seek access to the memory pool created in avixMain
12   //
13   memPool = avixMemPool_Get("PDEM");
14
15   memBlock = avixMemPool_Allocate(memPool);      // Allocate a memory block from pool
16
17   // Copy the content of the local struct to the allocated block
18   //
19   *AVIX_MEM_BLOCK_PTR(demoStruct1, memBlock) = testData;
20   …;
```

Code sample 35: How to allocate and access memory blocks

Often applications use arrays of data and these too can be placed in a block like any other type of data. Like in the previous example where the size of the struct determines the size of the blocks, here too the compiler can help to determine the block size provided the right definitions are present. Code sample 36 shows how the size of a block is based on the size of an array that will be copied to those blocks. Again macro `AVIX_MEM_BLOCK_PTR` is used to convert the block id to a pointer of the desired type for easy access.

```
1  #define NR_ELEMENTS 10                          // Definition for number of array el.
2
3  int demoArray[NR_ELEMENTS];                     // Array declaration
4
5  void avixMain(void)
6  {
7    AVIX_OBJECT_ID_DEFINE(tavixMemPoolId, poolId);
8    …;
9    // Create the memory pool with 5 blocks and let the compiler determine the blocksize
10   // Note that sizeof an array which is declaread externally returns the size of
11   // a pointer and not the size of an array!!!
12   //
13   poolId = avixMemPool_Create("PDEM", 5, sizeof(demoArray));
14   …;
15 }
16
17 TAVIX_THREAD_REGULAR d5_thread1(void* p)
18 {
19   AVIX_OBJECT_ID_DEFINE(tavixMemPoolId,  memPool);
20   AVIX_OBJECT_ID_DEFINE(tavixMemBlockId, memBlock);
21   int*              pAccess;
22   int               i;
23
24   memPool  = avixMemPool_Get("PDEM");            // Obtain id of avixMain created pool
25   memBlock = avixMemPool_Allocate(memPool);      // Allocate a block
26
27   // Convert the block id to a pointer the same type as the array
28   //
29   pAccess  = AVIX_MEM_BLOCK_PTR(int, memBlock);
30   for(i = 0; i < NR_ELEMENTS; i++)
31   {
32     // Copy the global array to the block
33     //
34     pAccess[i] = demoArray[i];
35   }
36   …;
37 }
```

Code sample 36: How to access memory blocks using plain 'C' array operations

Extended memory

Some hardware platforms AVIX targets do not allow all available RAM to be addressed in a single linear address space[17]. In this case a base amount of RAM is addressed in a limited linear address space and the extra RAM is addressed using a proprietary banking mechanism. AVIX allows this extra RAM to be used for memory pools. For this purpose function `avixMemPool_CreateExt` and macro `AVIX_MEM_BLOCK_PTR_EXT` exist. This memory is referred to as Extended Memory.

For portability reasons only, this function and macro are available with all AVIX ports. For hardware platforms not offering banked RAM, function `avixMemPool_CreateExt` behaves identical to `avixMemPool_Create` and macro `AVIX_MEM_BLOCK_PTR_EXT` behaves identical to macro `AVIX_MEM_BLOCK_PTR`.

Consult the applicable AVIX hardware platform Port Guide for details about Extended Memory.

The 'C' heap

As mentioned before, the 'C' runtime also offers memory management in the form of the heap. In general it is advised not to use the heap in any RTOS based application because of the problems mentioned in the beginning of this section. Still it is possible to use it be it that a number of precautions must be taken.

First of all, the heap is not thread safe. This means it may not concurrently be used from multiple threads. The heap is a 'C' construct and the 'C' language has no notion of concurrency and therefore the heap mechanism is not prepared to be used from a concurrent environment.

The most effective way around this is to just don't do this. When using the heap, make sure it is used from one thread only. Doing so there is no problem and malloc and free can be used as normal.

This does however put a burden on development and it is very easy to make mistakes which are hard to find and correct. Memory is for instance used quite often to communicate between threads and when accidentally a pointer to a heap allocated memory block is passed to another thread which just frees it, the above rule is violated. Chances are this may go undetected for a long time and one might easily be tempted to think everything is fine.

To prevent this and when still wanting to use the heap it just has to be made thread safe. This is very easy to do and illustrated here in the form of an example. The code presented here is not part of AVIX and will neither be in the future. Reason is that although thread safety can easily be applied to the heap, still the other problems, indeterminist timing and fragmentation exist and no solution for these problems is available. Furthermore it is not and will not be possible to allocate and free heap memory from an ISR like can be done with the AVIX memory pool mechanism.

Making the heap thread safe is done just by making sure the functions malloc and free cannot be preempted by another thread. This is done by creating two new functions, myMalloc and myFree which form wrappers around the standard malloc and free but before these are called, the wrapper locks a mutex. The example is shown in Code sample 37.

[17] Examples are Microchip PIC24F controllers with EDS (Extended Data Space) RAM and the Microchip PIC24EP and dsPIC33EP families.

```
 1  AVIX_OBJECT_ID_DEFINE(tavixMutexId, heapMutex);  // Global mutex
 2
 3  void* myMalloc(size_t size)                      // Function used i.s.o. malloc
 4  {
 5    void* pResult;
 6
 7    avixMutex_Lock(heapMutex);                     // Enter the critical section
 8    pResult = malloc(size);
 9    avixMutex_Unlock(heapMutex);                   // Leave the critical section
10
11    return pResult;
12  }
13
14  void myFree(void* p)                             // Function used i.s.o. free
15  {
16    avixMutex_Lock(heapMutex);                     // Enter the critical section
17    free(p);
18    avixMutex_Unlock(heapMutex);                   // Leave the critical section
19  }
```

Code sample 37: How to make heap operations thread safe

The sample code presented in Code sample 37 is just illustrative and is not part of AVIX. AVIX-RT strongly advises not to use the heap in any RTOS based application because of the problems mentioned in this section.

6.6.8 Exchange Services

The purpose of Exchange services is to offer high level inter-thread communication and synchronization supporting loose coupling, broadcasting and synchronization of threads operating at different speeds. Exchange Services makes reuse of software components easier and facilitates extending an application by adding new functionality with the least possible or, most of the time, even no changes to existing code.

When creating an application, a designer is faced with many challenges. Besides correct functional and temporal behavior, it is advantageous to have the right level of modularity in order to ease development, testing and reuse of software components in other applications. AVIX Exchange services assist in reaching these goals, not only leading to a decrease in initial development time but also make one feel more confident about the correctness of the resulting application.

To understand the real benefit of AVIX Exchange Services it helps to provide some background on the design decisions one is faced with when creating a basic application and the impact of these decisions on application behavior.

The first part of this chapter presents two designs of a sample application, one without and one with the use of Exchange services. This presents an overview of the Exchange services benefits.

The second part of this chapter goes into the details of Exchange services.

Sample Application using basic AVIX services

The sample application is based on three threads as shown in Figure 21 below.

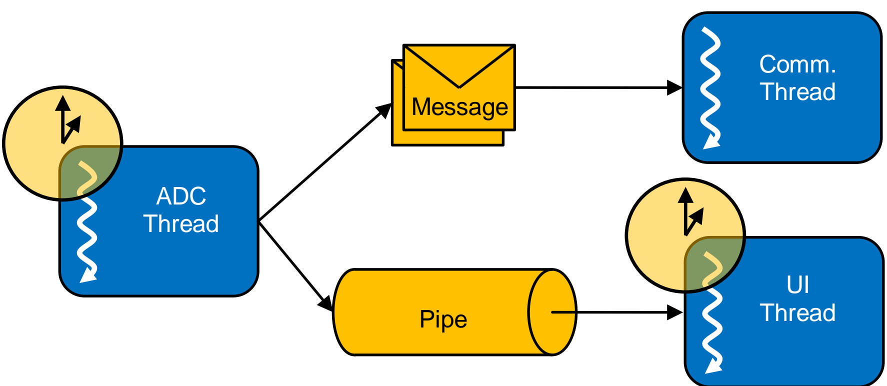


Figure 21: Exchange Sample Application without using Exchange Services

The ADC Thread is responsible for sampling analog inputs with a period of 1 millisecond and performs some algorithm on the values obtained from these channels.

The UI Thread is responsible for showing the output of the ADC thread on a LCD attached to the system. Because the UI does not need to be updated every millisecond, the UI thread uses a timer which activates this thread once every 50ms.

The Comm. Thread finally is responsible for communicating the ADC values to the outside world using serial communication.

The ADC thread writes its samples to a pipe which is read by the UI Thread and for the Comm. Thread, use is made of messages.

Although an application structured like this will work, a number of issues exist:

- **Undesired high coupling between threads:**
  The ADC Thread generates the core information and uses two different communication mechanisms to provide this information to the other threads. A pipe is used to communicate with the UI Thread and messages are used to communicate with the Comm. Thread.

  This has a number of disadvantages:

  - When extending the application with another thread requiring the information generated by the ADC Thread, not only must the new functionality be added but also the ADC thread has to be changed to write to whatever communication mechanism is used by this new thread.

  - ADC sampling is quite generic and it is well possible the ADC thread could be re-used in other applications. These other applications however are likely to have other threads requiring the information, again requiring changes to the ADC thread before it can be re-used.

  - Testing the ADC thread is specific for this application. The test environment must ‘connect’ to the pipe and the message system in order for the ADC thread to be tested.

- **Unnecessary high system load requiring customized solutions**:
  Following a straightforward approach, the UI Thread would be activated on every sample read from the pipe. Updating the UI 1000 times a second is not necessary and because of the speed of the attached LCD not even possible. An update rate of something like 20 times a second is more than sufficient.

  Several solutions exist for this problem, each having its downside.

  - The UI Thread uses one out of every 50 samples to update the LCD. Still the UI Thread would be scheduled once every millisecond. This generates unnecessary system load since 49 out of every 50 activations, the UI Thread does nothing.

  - The pipe is made large enough to hold 50 samples. The UI thread wakes up once every 20ms and reads 50 samples at once. The first 49 samples are discarded and only the last is used to update the LCD. This solution is a waste of memory since the pipe is much larger than actually required.

  - The ADC Thread only writes one out of every 50 samples to the pipe. This solves the issue of system load and does not waste memory. The downside is that the ADC Thread contains code actually belonging to the UI Thread. This hurts modularity even more.

  Finally, whatever solution is chosen, each of them require code changes when the scan rate of the ADC Thread changes, again another dependency between the different pieces of code the application is composed of.

Sample application using Exchange services

Exchange services offer a solution for all mentioned issues. Before going into details it is important to give a global overview on what Exchange services are.

Exchange services center around Exchange objects. An Exchange object is a data container threads can write to or read from. Exchange objects can be used to 'exchange' information between threads. One thread can write and others can read. Exchange objects do not offer a queuing mechanism. When an Exchange object is written, the old content is lost. This is comparable to the properties of a regular global variable. Unlike regular global variables, Exchange objects offer the following properties making them very usable in AVIX based applications:

- **Thread safe**: Exchange objects guarantee read and write operations to be atomic. The data contained in an Exchange object is always consistent. When writing to a regular global variable, the writing thread can be preempted potentially leading to data corruption.

- **Change notification**: When the content of an Exchange object changes, the Exchange object notifies threads requiring the new data. For notifications use can be made of the different services offered by AVIX like sending a message or setting an event flag. For a thread to be notified, it has to be connected to the Exchange object. Regular global variables do not offer a notification mechanism and threads requiring the new information have to poll the global variable. Polling is very inefficient and a mechanism that should be prevented from being used in any RTOS based application as much as possible.

The really interesting feature of Exchange objects is change notification. First this mechanism does decouple data 'producing' and data 'consuming' threads. A producer just writes to the Exchange object and has no knowledge which other threads are 'listening' to this Exchange. Once the data is written, the 'producer' is ready. One or more threads requiring the information are 'connected' to the Exchange. When data is written, every connection is 'informed' that new data is available. AVIX Exchange objects allow a large number of different mechanisms to be used for the actual notification.

For now it is sufficient to mention two basic types of notification exist, push and pull. A push notification informs a connected thread that new data is available and sends the data together with this notification. A pull notification informs a connected thread that new data is available but does not send the data together with this notification. It is up to the notified thread to read the data from the Exchange object. A push notification is lossless, every write to the Exchange results in the connected thread to have access to the new data. A pull notification can be lossy. In between the notification and the notified thread reading the Exchange, new data is potentially written to the Exchange object.

Figure 22 shows the sample application based on using an Exchange object.

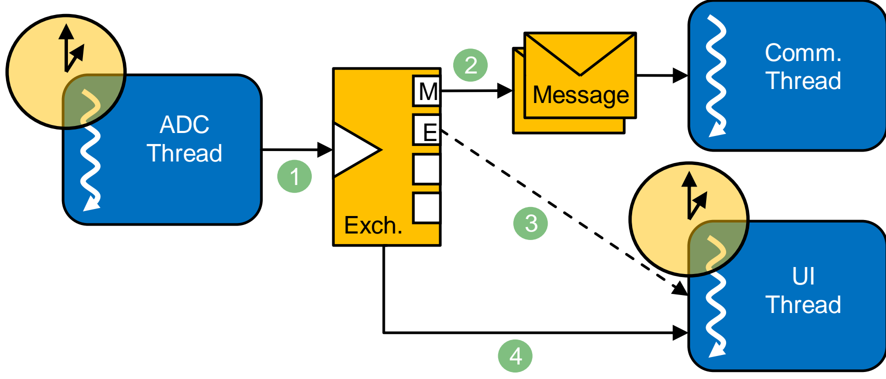


Figure 22: Exchange sample application using Exchange Services

This sample application has the exact same functionality as the application shown in Figure 21; still there are a lot of differences, addressing the issues of the earlier solution. How does this application work?

1. When the ADC Thread has a new sample available, this sample is written to the Exchange object. This is all the ADC Thread does and it has no ‘knowledge’ which other threads are using the data.

2. Between the Comm. Thread and the Exchange a message based connection exists. Every write operation to the Exchange object results in a message being send to the Comm. Thread. The Comm. Thread is notified new data is available since it receives this message. A copy of the data written to the Exchange object is present in the message. As such this is a push notification.

3. Between the UI Thread and the Exchange an Event Flag based connection exists. Every write operation to the Exchange object results in an Event Flag being set. The UI Thread is notified new data is available since it waits for this Event Flag. It is however up to the UI Thread itself to decide when to access the Exchange data by reading the Exchange object. Since in between the Event Flag being set and the Exchange object being read, new data may be written to the Exchange object, it is not guaranteed the UI Thread ‘sees’ all data. Potentially data may get lost. As such this is a pull notification.

Essential is the ADC Thread has no ‘knowledge’ what other threads are informed of the new data and how these threads are informed. This is determined by the connections made to the Exchange. Connections typically are established during initialization.

The message send to the Comm. Thread is placed in the message queue of this thread and the message will be processed once the Comm. Thread is activated.

The UI Thread behaves different. As shown, the UI Thread uses a local timer. This timer will wake up the thread every 50ms. Only then, the UI Thread tests the flag set by the Exchange and when set, the data of the Exchange is read and processed (step 4 in Figure 22). Before the UI Thread runs, the Exchange is written many more times and each time the Exchange data is overwritten. So once the UI Thread reads the Exchange data, only the last value written is used and the others are implicitly discarded.

What are the advantages of this approach?

- **The ADC Thread is no longer coupled to receiving threads communication mechanisms.**
  When the ADC thread has a new sample available, this sample is written to the Exchange object. The ADC thread has no knowledge which threads are consuming this information whether this is no thread at all or a multitude of threads, no code change is required for the ADC thread.

- **The UI Thread no longer receives data it has to discard.**
  No longer is any data sent to the UI Thread but an Event Flag is set instead. The UI thread does not need to be activated on this event. Only when the timer expires, the UI Thread is activated. At this moment it can check the Event Flag previously set by the Exchange object and when set, read the data from the Exchange object and write to the LCD. Still, the UI Thread discards 49 out of 50 samples but this no longer consumes any time or memory.

This solution does not only solve the issues that came with the earlier solution. Using Exchange services has a positive effect on modularity, testability (both module and integration testing), clarity of the design and development effort.

Exchange Services in Detail

An Exchange object is a kernel object containing user specified data and offers functionality to:

- Read and write this data in a thread safe way. During reading and writing, the Exchange object is locked so the thread executing the operation cannot be preempted. The read and write operations are said to be thread safe.

- Set up one or more notification connections in order for threads to be informed when the content of the Exchange object is changed (the Exchange object is written).

Lock functionality for thread safe Exchange access is accomplished by a Mutex which is contained in every Exchange object. Two types of access are offered. Basic access allows an Exchange object to be read and written without the need to explicitly lock the Exchange object. Advanced access allows the lock to be explicitly manipulated by the user. This allows faster access to the Exchange object data and other, more advanced operations.

Notification of changes to Exchange object content is accomplished by so called connections. When data is written to the Exchange object, a connection is activated to inform the connected thread of the fact that new data is available.

Connections use AVIX services to notify a connected thread. The type of service used is determined when creating the connection. A connection being activated can result in a message being send, a pipe being written or an event flag being set in either an event group or a thread event group.

Additionally, callback functions can be connected to an Exchange. This type of connection results in the callback function being activated when the Exchange object is written. The action taken by the callback function is entirely up to the user. Callback connections allow preprocessing of the data written to the Exchange object before informing the consuming thread. Callbacks allow ultimate flexibility.

Exchange objects do not implement a queue. When new data is written, this data replaces the current content of the Exchange object. It is correct to compare an Exchange to a global variable, a global variable however extended with the required functionality to use it in a multithreaded environment making application development much easier compared to an RTOS not offering this type of objects.

The following sections contain a description of the different types of functionality offered by Exchange objects. Code Samples are used to illustrate Exchange usage. The following topics are dealt with:

- How to create and access an Exchange object

- How to access the content of an Exchange object

- How to subscribe to an Exchange object to be informed of changes to its content

- Practical hints and tips when using Exchange objects

How to create and access an Exchange object

An Exchange object is created using function `avixExch_Create`. Exchange objects, can be given a human readable name. Using this name, other threads can obtain access to an existing Exchange object through function `avixExch_Get`. These functions adhere to the standard AVIX object naming and management mechanism. More details about this mechanism are found in §6.5.1.

When creating an Exchange object, its size in bytes is specified. The specified size is rounded up to the nearest multiple of the word size of the applied MCU.

Optionally a pointer can be passed, referring data used to initialize the Exchange object data section. When no pointer is passed (NULL), the Exchange data section is filled with all zeros.

Since an Exchange object can be considered to be an advanced type of global variable, it makes sense to use the type of this 'variable' when creating the Exchange to determine its size. Code sample 38 shows how to create an Exchange object that will hold a structure containing the x, y and z values of an accelerometer.

```
 1  typedef struct                  // Define the type used to be held by the
 2  {                               // Exchange object.
 3    int x;
 4    int y;
 5    int z;
 6  } tAccelerometer;
 7
 8  …;
 9
10  AVIX_OBJECT_ID_DEFINE(tavixExchId, exchange);
11
12  // Create an exchange object using the size of the struct it will hold
13  // Parameter 3 (NULL) takes care the exchange is filled with all zeros
14  //
15  exchange = avixExch_Create("exc", sizeof(tAccelerometer), NULL);
```

Code sample 38: How to create an exchange object

Basic Exchange object access

Exchange objects can be written and read. Access to an Exchange object is 'thread safe'. Write and read operations are atomic and it is not possible the thread accessing the Exchange is preempted in the process of doing so. A write operation leads to a consistent new content of the Exchange object. A read operation leads to a consistent copy of the Exchange object content.

For basic access use is made of `avixExch_Read` and `avixExch_Wite`. The read function copies the content of the Exchange object to a user supplied buffer. The write operation copies the content of a user supplied buffer to the Exchange object.

When executing the write operation, the Exchange object will activate connections to inform consumers of its data that new data is available.

Based on a structure holding the x, y and z values of an accelerometer, Code sample 39 shows how to use these functions.

```
1  AVIX_OBJECT_ID_DEFINE(tavixExchId,   exchange);
2  AVIX_OBJECT_ID_DEFINE(tavixThreadId, writerThread);
3  tAccelerometer userVar;
4
5
6  userVar.x = 1;              // Fill the buffer variable with the data
7  userVar.y = 2;              // to copy to the Exchange object
8  userVar.z = 3;              //
9  …;
10
11 // Copy the content of userVar to the Exchange. This function executes in a
12 // critical section which is taken care of internally. Also connected consumers are
13 // informed of the fact new data is available.
14 //
15 avixExch_Write(exchange, &userVar);
16 …;
17
18 // Copy the content of the Exchange to the userVar. This function executes in a
19 // critical section which is taken care of internally. The read function returns
20 // the id of the thread that executed the last Write operation in the variable
21 // refered by the third parameter.
22 //
23 avixExch_Read(exchange, &userVar, &writerThread);
```

Code sample 39: Reading and writing an exchange object

Only reading and writing an Exchange object is however not sufficient. Often it will be necessary to manipulate the content of an Exchange object. This requires the Exchange object to be read, manipulate the copy of its content and write back this manipulated copy. Code sample 40 shows how this can be done where it is very important to realize this approach is not always suitable.

```
1  AVIX_OBJECT_ID_DEFINE(tavixExchId, exchange);
2  tAccelerometer  userVar;
3
4  …;
5
6  // Copy the content of the Exchange to a local variable (read), modify one of the
7  // fields and copy the content of the local variable back to the Exchange.
8  //
9  avixExch_Read(exchange, &userVar, NULL);
10
11 userVar.x += 1;      //Here the Exchange is not locked and the thread can be preempted
12
13 // Write the modified data back to the Exchange object and activate connections to
14 // inform consumers of the data that new data is available.
15 //
16 avixExch_Write(exchange, &userVar);
```

Code sample 40: Unsafe modification of the content of an exchange object

As said, this method is however not always suitable, why is that? The read operation is locked so the copy of the Exchange object data made to variable `userVar` is consistent. The write operation is also locked so the copy written to the Exchange object is consistent also. During the manipulation of the copy however (line 11), the Exchange object is not locked. When at this point the thread is preempted by another thread also manipulating the content of the Exchange object, this content may become corrupted. This is comparable to the classic Read-Modify-Write problem in concurrent systems.

AVIX offers additional functions to prevent this.

Besides implicit locking of the Exchange by `avixExch_Read` and `avixExch_Write`, an explicit lock can be placed on an Exchange object through function `avixExch_Lock`. This lock can be released using function `avixExch_Unlock`. The use of these functions is illustrated in Code sample 41.

```
 1 AVIX_OBJECT_ID_DEFINE(tavixExchId, exchange);
 2 tAccelerometer  userVar;
 3
 4 …;
 5
 6 avixExch_Lock(exchange);
 7
 8 // Copy the content of the Exchange to a local variable (read), modify one of the
 9 // fields and copy the content of the local variable back to the Exchange.
10 //
11 avixExch_Read(exchange, &userVar);
12
13 userVar.x += 1;
14
15 // Write the modified data back to the Exchange object and activate connections to
16 // inform consumers of the data that new data is available.
17 //
18 avixExch_Write(exchange, &userVar);
19
20 avixExch_Unlock(exchange);
```

Code sample 41: Safe modification of the content of an exchange object

In the above sample, the entire Read-Modify-Write sequence is locked with the function executed on line 6 and unlocked with the operation executed on line 20. The Mutex used internally for the lock supports nesting. A thread owning the Mutex may lock again, as long as the number of unlocks equals the number of locks. As a result it does not matter the read and write operations place a lock themselves, these locks just increase and decrease the nesting level of the already existing lock.

Advanced Exchange object access

Performance of the presented solution is not optimal. Just to increase the value of one of its fields, the entire data structure is copied twice.

A better performing solution is offered by using the more advanced Direct Access method. Using Direct Access, the content of the Exchange object data section is manipulated directly using a pointer. Still manipulation of the data must be protected against concurrent access; the operation may not be preempted.

A direct access sequence is started by locking access to the Exchange object using function `avixExch_Lock`. Besides activating the lock, this function returns a pointer to the data section of the Exchange object. It is this pointer that can be used to directly manipulate the content of the Exchange object. No use is made of functions `avixExch_Read` or `avixExch_Write`.

When ready accessing the Exchange object data, the lock must be released. In case the Exchange object is written, connections must be activated to notify consumers of the fact new data is available. Since no use is made of `avixExch_Write`, this must be done explicitly. For this purpose a second unlock function is offered, `avixExch_UnlockAndNotify`.

When the content of the Exchange object is only read, no changes are made and no notifications are required. In this case the access sequence is unlocked by using `avixExch_Unlock`.

A direct access sequence where the content of the Exchange object is manipulated is shown in Code sample 42.

Functionally, this example is equal to the example shown in Code sample 41 but now using direct access so offering a much better performance.

```
1  AVIX_OBJECT_ID_DEFINE(tavixExchId, exchange);
2  tAccelerometer*  pUserVar;    // Note this is a pointer and no longer a complete struct
3
4  …;
5
6  // Lock the exchange and obtain a pointer to the internal data
7  //
8  pUserVar = avixExch_Lock(exchange);
9
10
11 // Update a struct field using the pointer returned by the lock operation
12 //
13 pUserVar->x += 1;
14
15 // Unlock the exchange allowing other threads write access and  inform consumers of
16 // the data that new data is available.
17 //
18 avixExch_UnlockAndNotify(exchange);
```

Code sample 42: Direct modification of the content of an exchange object

Although the approach shown in Code sample 42 is more efficient than the approach shown in Code sample 41 one must be very careful since programming errors might easily lead to writing outside the boundaries of the Exchange data section. For this reason it is strongly advised to always define a struct for the basic type of an Exchange and be very careful always to use the same struct definition when accessing the Exchange through a pointer.

Summarized, `avixExch_Lock` can be used together with basic access functions `avixExch_Read` and `avixExch_Write` to allow modifications of the Exchange object data. In this case the lock is always released using `avixExch_Unlock` since the notification is taken care of by `avixExch_Write`.

The pointer returned by `avixExch_Lock` can be used for direct access. When only reading the data, the lock is released using `avixExch_Unlock`. When writing the Exchange object data, the lock must be release using `avixExch_UnlockAndNotify` to notify consumers of the presence of this new data. Table 4 shows a summary of the different Exchange Access functions and the type of locking involved/required.

Access type	How to	Explicit User Lock Sequence
Read only access	`avixExch_Read`	None
	Direct access	`avixExch_Lock` – ***`avixExch_Unlock`***
Write only access	`avixExch_Write`	None
	Direct access	`avixExch_Lock` – ***`avixExch_UnlockAndNotify`***
Modify access	`avixExch_Read` – `avixExch_Write`	`avixExch_Lock` – ***`avixExch_Unlock`***
	Direct access	`avixExch_Lock` – ***`avixExch_UnlockAndNotify`***

Table 4: Exchange object access types and user lock sequences

► *After releasing a lock, make sure no longer to use the pointer obtained through `avixExch_Lock` for any access to the Exchange object.*

Connections, how to be notified of changes to the content of an Exchange object

When an Exchange object is written, threads requiring this new information can be *notified* of this. For this purpose, Exchange services offer *connections*. In order for a thread to be *notified* about changes to the Exchange object content, a *connection* to the Exchange object must exist. Connections use basic AVIX services to notify threads requiring to be notified. To a single Exchange object multiple connections can be made.

Two types of connection are offered, *push connections* and *pull connections*.

- Push connections: This type of connection is 'data centric' when an Exchange object is written, a copy of the new data is send to the connected thread. Every write operation results in a 'send' operation. As a result, the connected threads are informed of every update and no data is lost. Push connections are either 'thread-message-queue-connections' or 'pipe-connections'.
    - Thread-message-queue-connections: Every time new data is written to an Exchange object a message is allocated and filled with the new Exchange object data. Next this message is send to message queue of the connected thread. A 'thread-message-queue-connection is created with function `avixExch_ConnectMsgQThread`.
    - Pipe-connections: Every time new data is written to an Exchange object, the data is also written to a pipe as a block. For a thread to be informed of this, the thread must read the pipe. A 'pipe-connection' is created with function `avixExch_ConnectPipe`.
- Pull connections: This type of connection is not data centric. The Exchange object only notifies the content of the Exchange object has changed. The notification is not accompanied by this new data and it is up to the thread listening to the notification whether it reads the new data or not. Pull connections are either, 'event-group-connections' or 'thread-event-group-connections'.
    - Event-group-connections: Every time new data is written to an Exchange object one or more flags are set in the connected event group. A thread waiting for this event group is informed of this by specifying the applicable flag(s) in the wait operation. An 'event-group-connection' is created with function `avixExch_ConnectEventGroup`.

An additional class of connections is formed by callback connections. The aforementioned connection types use an AVIX kernel object for the notification. A call-back connection consists of a user written 'C' function that is connected to the Exchange object. When new data is written to the Exchange object, this function is called. It is entirely up to this function what the reaction to the notification is. When setting up a call-back connection, an id of a thread, a pipe or an event group is specified. When the call-back connection is activated, this kernel object id is passed as one of the parameters to the call-back function. This allows the call-back function to take whatever appropriate action.

Call-back connections are neither push or pull connections. It depends on the functionality of the call-back function what class the connection belongs to. Suppose the call-back function receives a thread id and the function allocates a message to send to the thread identified by this id, effectively the connection is a push connection. When on the other hand the function sets a flag in the thread event group identified by this thread id, the connection is effectively a pull connection.

Depending on the type of kernel object id that must be passed to the call-back function, a call-back connection is established using function `avixExch_ConnectCallbackEventGroup`, `avixExch_ConnectCallbackPipe` or `avixExch_ConnectCallbackThread`.

When to use a specific type of connection

With all the different types of connections offered by Exchange services, some guidance is needed on when to use which type of connection. Exchange objects are used to transport information from one thread to one or more other threads. Threads receiving information use a connection to the Exchange object The first question is whether a receiving thread must have access to every new data item written to the Exchange or if it is allowed to 'miss' data.

Push connections are typically used when no data may get lost, so either a 'thread-message-queue-connection' or a 'pipe-connection'.

An example of this is shown in the first part of this section where a basic example application is presented (Figure 22 on page 98). The Comm. Thread must serialize all data generated by the ADC Thread and no samples may get lost. To make a choice between the two types of push connection depends on other criteria again. The AVIX message mechanism allows a thread to block while waiting for a message or alternatively allow an event group flag to be set when a message is present in the message queue. In this second case the thread does block on the event group. This allows the thread to not only wait for a message but at the same time for other sources setting an event flag. When this is not required, a 'pipe-connection' may be used just as well.

Pull connections are typically used when the receiving thread does not need access to each new data entity written to the Exchange object, so either an 'event-group-connection' or a 'thread-event-group-connection'.

Again, an example of this is shown in the basic example application shown in Figure 22 on page 98. The UI Thread must display the ADC samples on the LCD but with a much lower frequency than the samples are generated. The chosen connection is a 'thread-event-group' connection. Whenever new data is written to the Exchange object an event flag is set. The UI Thread however only wakes up at the rate of its timer which is much lower than the data being generated. The UI thread reads the most recent value from the Exchange object and shows this value on the LCD. The fact that in between subsequent updates of the LCD many samples are not used does not matter and is even advantageous for overall application performance.

Furthermore, pull connections are very useful in those places where application behaviour is controlled. Let's again illustrate this with an example. Suppose the core thread of the application samples analogue inputs with a selectable period. To hold the sample period, use can be made of an Exchange object. The value of this Exchange object may change when the user presses a switch or based on some application internal event. Essential is the ADC Thread is informed of changes to this Exchange object to adapt its sample rate according to the new value. For this it is sufficient for the ADC thread to be informed using an event flag and subsequently read the Exchange object to learn the desired sample rate. If the sample rate changes very fast, it is fine if the ADC thread ignores intermediate sample rates and only uses the last one selected.

Call-back connections are typically used when some form of pre-processing is required before informing a thread of new data being available.

Suppose the Graphical LCD connected to the system has to show a graphical representation of the analog values sampled by the ADC Thread. The UI Thread is still subject to the constraints presented before and only becomes active at a much lower rate than the core of the application. In this case however the UI thread does need the intermediate samples in order to show these as a graph. A callback connection is needed here. The callback is activated on every new sample and what it does is store these in an array. No samples are lost. Subsequently the callback function sets a flag in the thread event group, just as done implicitly by a regular 'thread-event-group-connection'. Once the UI Thread is activated, it can access the data in the array filled by the callback and use this to present a graph on the LCD.

For a number of connection types code samples are presented. First let's see how to setup and use a 'thread-message-queue-connection' (Code sample 43).

```
 1 AVIX_OBJECT_ID_DEFINE(tavixThreadId, producerThread);
 2 AVIX_OBJECT_ID_DEFINE(tavixThreadId, consumerThread);
 3 AVIX_OBJECT_ID_DEFINE(tavixExchId,   exchange);
 4
 5 void avixMain()
 6 {
 7   producerThread = avixThread_Create(….);
 8   consumerThread = avixThread_Create(….);
 9
10   // Create an Exchange object with an integer sized data section.
11   //
12   exchange = avixExch_Create(NULL, sizeof(int), NULL);
13
14   // Create a thread message queue connection so every write to the Exchange results
15   // in a message containingn the new Exchange data being send to the consumerThread
16   //
17   avixExch_ConnectMsgQThread(exchange, consumerThread, 0);
18 }
19
20 TAVIX_THREAD_REGULAR producer(void* p)
21 {
22   int data;
23   …;
24   while(1)
25   {
26     …;
27
28     // Write the Exchange resulting in a message being send. The fact that a message
29     // is send is determined by the connection type and not known to the producer.
30     //
31     avixExch_Write(exchangeId, &data);
32   }
33 }
34
35 TAVIX_THREAD_REGULAR consumer(void* p)
36 {
37   tavixMsgId msg;
38   …;
39
40   while(1)
41   {
42     msg = avixMsgQThread_Receive();
43
44     // Process the message
45     //
46     …;
47
48     avixMsg_Free(msg);
49   }
50 }
```

Code sample 43: Setting up and using an exchange thread message queue connection

The connection between the Exchange and the consumer thread is made during application initialization with the call to `avixExch_ConnectMsgQThread` on line 17. The producer thread just writes to the Exchange without knowing what connections exist. The consumer thread receives a message without knowing that it comes from an Exchange.

For the consumer thread there is no difference between a message being send directly from a thread or indirectly through a connection. Messages have types and in this case the message type is the value passed as the third parameter to `avixExch_ConnectMsgQThread`, which is 0 in this example. Second, messages contain the thread id of the thread that sends the message. In this case this is the id of the thread that triggered the message being sent which is the id of the producer thread.

Code sample 44 shows how to use a thread event group connection. Compared to Code sample 43 no changes are made to the producer thread. What is different is the connect call on line 17 and the consumer thread.

```
1  AVIX_OBJECT_ID_DEFINE(tavixThreadId, producerThread);
2  AVIX_OBJECT_ID_DEFINE(tavixThreadId, consumerThread);
3  AVIX_OBJECT_ID_DEFINE(tavixExchId,   exchange);
4
5  void avixMain()
6  {
7    producerThread = avixThread_Create(….);
8    consumerThread = avixThread_Create(….);
9
10   // Create an Exchange object with an integer sized data section.
11   //
12   exchange = avixExch_Create(NULL, sizeof(int), NULL);
13
14   // Create a thread message queue connection so every write to the Exchange results
15   // in a flag being set in the event group of the consumer thread
16   //
17   avixExch_ConnectEventGroup(exchange, consumerThread.asEventId, AVIX_EF(0));
18 }
19
20 TAVIX_THREAD_REGULAR producer(void* p)
21 {
22   int data;
23   …;
24   while(1)
25   {
26     …;
27
28     // Write the Exchange resulting in a flag being set in the consumer thread.
29     // The fact that an event flag is set is determined by the connection type and
30     // not known to the producer.
31     //
32     avixExch_Write(exchangeId, &data);
33   }
34 }
35
36 TAVIX_THREAD_REGULAR consumer(void* p)
37 {
38   tavixEventFlags flags;
39   int             data;
40   …;
41
42   while(1)
43   {
44     flags =                                  // Wait for flag 0 which is set when
45       avixEventGroup_Wait                    // the exchange is written
46         ( avixThread_GetIdCurrent().asEventId, // Id of thread event group
47           AVIX_EF(0),
48           AVIX_EVENT_GROUP_ANY,
49           AVIX_EF_NONE,
50           AVIX_EF(0)     );
51
52     If(AVIX_EF_IN(AVIX_EF(0), flags)        // Test for flag 0 and if set
53     {                                        // read the most recent content of
54       avixExch_Read(exchange, &data, NULL);  // the exchange
55       …;
56     }
57   }
58 }
```

Code sample 44: Setting up and using an exchange thread event group connection

The consumer thread waits for event flag 0 to be set. In practice it is likely a thread will wait for multiple event flags so it can react to multiple triggers. When the consumer thread is triggered, it tests flag 0 to know it is activated because of a write to the Exchange object. Subsequently it reads the data. The major difference between this code sample and Code sample 43 is that the consumer thread cannot be sure it ‘sees’ all data written to the Exchange object. In between the

flag being set and the consumer reacting, it is well possible other write operations to the Exchange object have taken place.
Code sample 45 finally shows an example of a callback connection where the callback receives a pipe id as one of its parameters.

```
 1 AVIX_OBJECT_ID_DEFINE(tavixThreadId, producerThread);
 2 AVIX_OBJECT_ID_DEFINE(taviPipeId,    consumerPipe);
 3 AVIX_OBJECT_ID_DEFINE(tavixExchId,   exchange);
 4
 5 // Callback function prototype.
 6 //
 7 void callbackFunc(…);
 8
 9 void avixMain()
10 {
11   producerThread = avixThread_Create(….);
12   consumerPipe   = avixPipe_Create(….);
13
14   // Create an Exchange object with an integer sized data section.
15   //
16   exchange = avixExch_Create(NULL, sizeof(int), NULL);
17
18   // Create a callback connection receiving a pipe id as one of its parameters.
19   //
20   avixExch_ConnectCallbackPipe(exchange, callbackFunc, consumerPipe, 0);
21 }
22
23 TAVIX_THREAD_REGULAR producer(void* p)
24 {
25   int data;
26   …;
27   while(1)
28   {
29     …;
30
31     // Write the Exchange resulting in the callback function being called. The
32     // fact that a callback function is called is determined by the connection type
33     // and not known to the producer.
34     //
35     avixExch_Write(exchangeId, &data);
36   }
37 }
38
39 void callbackFunc
40 ( tavixExchId    exchange,    // Id of Exchange callback is connected to
41   const void*    pData,       // pointer to Exchange data section
42   unsigned int   size,        // size in bytes of Exchange data section
43   tavixPipeId    pipeId,      // pipeId passed to connect function
44   unsigned short userParam )  // userParam passed to connect function
45 {
46   // In this example, the callback is used to filter the data. Suppose the
47   // generated values  come from an ADC thread, the filter only sends the
48   // values between 200 and 800 to the pipe. The remaining values are ignored.
49   //
50   int value = *((int*)pData);
51
52   if ((value >= 200) && (value <= 800))
53   {
54     avixPipe_Write(pipeId, &value, 1);
55   }
56 }
```

Code sample 45: Setting up and using an exchange callback connection

In this code sample, the callback is used to filter the data before it is send to the pipe. The assumption is made the thread reading the pipe is only interested in a subset of all samples written by the producer thread. Doing so reduces system load since the thread reading the pipe does not need to do the filtering and is only activated when samples it actually is interested in are generated.

Connection identification and its use

Every connection made to an Exchange object is identified by a connection id (type `tavixExchConnId`). The connection id is returned by the `avixExch_Connect…` functions. Connection id's are application wide unique. AVIX does not allow connections to be removed. Once a connection is created, it remains for the lifetime of the application and so does the connection id.

Situations exist where it is beneficial for a connection to be inactive. Suppose for instance threads are connected to Exchange objects where the application is in such a state that the thread does not need to do anything with the Exchange data.

Think of a communication thread where the communication needs to temporarily shut down or a user interface thread where the user interface is temporarily disabled.

In these situations the thread could decide for itself it has nothing to do. Every time the thread is notified, it wakes up and based on its state does nothing. It would however be better to temporarily disable the connection. When connections are disabled, write operations to the Exchange object do not activate the disabled connection and prevent threads from being unnecessarily activated.

For this purpose it is possible to control the mode of connections. AVIX allows an individual connection to be disabled or enabled and also to enable or disable connections of a specific Exchange all at once. The applicable functions are:

- `avixExch_DisableConnection`
- `avixExch_EnableConnection`
- `avixExch_DisableAllConnections`
- `avixExch_EnableAllConnections`

A second purpose for connection id's is to allow a thread to access the related Exchange object without having direct access to the Exchange id. This allows a thread to access an Exchange object without knowing which Exchange object it is connected to. In this case, the consuming thread is provided a Connection id and by using this id instead of the Exchange id offers more flexibility and even looser coupling between the different software components making up the application.

To obtain the id of the Exchange object a connection belongs to, use can be made of function `avixExch_GetConnectionExch`. This function receives a Connection id and returns the id of the Exchange object it belongs to.

Exchange Services dynamic behavior

Exchange objects are fully thread safe. An Exchange object can be used from as many threads as desired with whatever function desired. Exchange services guarantee the Exchange object and its connections to remain in a consistent state. Mentioned before are the data accessing functions and how these are locked either implicitly or explicitly. The same Mutex based locking mechanism is used with many other Exchange services functions in order to guarantee the Exchange object, its data and its connections to have a consistent, deterministic state. As a result every function can be used by whatever thread required. Multiple threads are allowed to make connections to a single Exchange object at the same time. The Exchange services locking mechanism guarantees these operations to work as expected. The advantage of using Mutex functionality as the basis for the Exchange locking mechanism is that Mutexes offer priority inheritance. When a thread holds a lock on an Exchange and the lock is claimed by another thread having a higher priority, the priority of the owning thread is temporarily raised to that of the requesting thread until the owning thread releases the lock. This ensures fair scheduling. More details on this mechanism can be found in § 6.6.1.

Another advantage of using Mutex functionality for the Exchange locking mechanism is that when activating connections, the scheduler is not locked out for the entire duration of the operation required to activate all connections of an Exchange object. Since an Exchange object may have as many connections as desired, it shall be obvious that activating all connections can take quite some time. The more connections, the longer the processing. When activation of a connection results in a consuming thread being able to run, in case the consuming thread has a higher priority than the thread executing the write operation, the consuming thread is activated the moment it becomes ready to run. So in this case the write thread is preempted in favor of the consuming thread. Only when the consuming thread is blocked again the process of activating all connections is continued by the writing thread. This again is a prerequisite for fair scheduling.

When using callback functions the following is important to take into account; Callback functions are activated on the thread executing the Exchange object write operation. In general, a callback function should be as fast as possible. No substantial processing should be done in a callback function since this will hold up the writing thread. Most of the time, one of the actions of a callback function will be to notify a thread by sending a message, writing to a pipe or setting an event flag. Some of these operations are potentially blocking. To send a message, first a message has to be allocated from the message pool and in case the pool is empty, the allocation function will block. Writing to a pipe blocks if the pipe is full. To circumvent this, a thread can use a time out of zero. In this case the functions will not block but return a time-out status. When called from an Exchange callback function, potentially blocking functions automatically behave as if a time-out value of zero (0) has been set. For this, no time-out needs to be set using `avixThread_ArmTimeOut` and use of this function is even prohibited. When using this function from a callback function, an error will be thrown. So when using potentially blocking functions from within a callback function it is required to test the result of the function to determine whether it has succeeded.

► *Potentially blocking functions called from a callback function automatically behave as if a time-out value of zero has been selected. As a result these functions will never block and it is required to test for the result of the function to determine whether it has succeeded.*

Finally, from a callback function, most Exchange functions are not allowed to be called for the Exchange object the connection belongs to. Reason is to safeguard the consistency of the Exchange object and its connections. Because of the parameters received by a callback function, this does not impose a limitation.

The only operation a callback is allowed to execute on its own Exchange object is to disable the callback connection. This is done using function `avixExch_DisableActiveConnection`. Calling this function disables the connection so on subsequent write operations to the Exchange object the callback connection will not be activated unless in the meantime it has been enabled again by another thread.

6.7 What to do in case of problems

This section contains a list of topics to check in case an AVIX based application is not working as expected.

- **Do not enable interrupts in `avixMain`**:
  For its internal working AVIX uses software interrupts. The functionality related to these interrupts is only guaranteed to work correctly after AVIX is fully initialized. For this reason, initialization is done with all interrupts disabled. Special care must be given to the user supplied function `avixMain`. Make sure nowhere in `avixMain` or the functions called from `avixMain` interrupts are enabled.

- **Do not use ISR dedicated functions from threads or DIH's**:
  Threads and DIH's may under no circumstance use functions intended for exclusive use by ISR's. Doing so will result in a frozen or crashing system. Functions intended for exclusive use by ISR's can be recognized by their name ending in `...FromISR`. The only exception to this rule is function `avixDIH_Queue`. Although intended for exclusive use by ISR's, this function does not follow the mentioned naming convention.

- **Do not use non-ISR functions from ISR's**:
  Interrupt Service Routines may under no circumstance call regular AVIX functions. Doing so will result in a frozen or crashing system. AVIX offers dedicated functions for use by ISR's. These functions can be recognized by their name ending in `...FromISR`. Make sure ISR's or user functions called from ISR's do not call any non-ISR AVIX function. The only exception to this rule is function `avixDIH_Queue`. Although intended for exclusive use by ISR's, this function does not follow the mentioned naming convention.

- **Do not use non-ISR functions from a pipe callback**:
  For pipes, callback functions can be specified intended for device control. Although intended for basic MCU device control, AVIX functions may be used from a pipe call-back. In general it is advised not to use any AVIX function from a pipe call-back. When still doing so, make sure only to use the same functions as intended for use from ISR's. These functions can be recognized by their name ending in `...FromISR`. Not following this rule might lead to an instable or crashing application.

- **When using the same pipe from multiple ISR's make sure they all have the same priority**
  When using a pipe from an ISR and this ISR is interrupted by a higher priority ISR, this second ISR may not use the same pipe. AVIX cannot check this so make sure a situation like this does not occur.

- **Install a user error handler**:
  AVIX uses a centralized error detection mechanism. An AVIX function detecting an error does not return but will end up in this centralized error handler. In order to know what error occurred, you can install a custom error hander. During development this allows for easy error detection by using a breakpoint in this handler.

- **Check thread stack sizes**: Check for thread stacks being large enough to hold all local variables, return addresses of functions being called and ISR context. When using the AVIX system stack the stack area to reserve for ISR context may be as small as zero (0) bytes, depending on the controller family. See the controller specific Port Guide for details. Pay special attention to local variables inside the thread function or functions called from the thread function. Large local variables may increase the stack pointer to point behind the end of the thread stack.

- **Check system stack size*:*** The system stack is used by the idle thread and interrupt handlers defined with the AVIX supplied macros. The required size of the system stack depends on the number of unique interrupt priorities for those interrupts defined using the AVIX supplied macros. The more unique interrupt priorities, the larger the required size of the system stack. Read the port specific user guide for more details. Pay special attention to local variables inside the ISR or functions called from the ISR. Large local variables may increase the stack pointer to point behind the end of the system stack.

7 Reference Guide

This section contains the AVIX reference guide. Here a detailed description is given of all functions, types and definitions offered by AVIX. First some generic background information is presented relating to all API detailed descriptions.

7.1 Conventions

This chapter presents the AVIX naming and usage conventions.

7.1.1 Naming

AVIX offers a coherent and easy to understand naming convention for all its functions and types. Every AVIX function is named like:

avix<object_type>_<operation>[<additional object>]

<object type> is the identification of the kernel object type or other entity offered by AVIX. Possible values for this field are 'Thread', 'Mutex', 'Semaphore', 'EventGroup', , 'MsgQThread' 'Msg', 'Pipe' or 'MemPool'. Some additional functions not directly related to a specific kernel object type are offered where for field <object type> possible values are 'Error' or 'DIH'. Field **<operation>** contains and identification of the operation the function executes. Examples are 'Create' and 'Get'. Field **<additional object>**, which is optional, is used with a number of functions to further detail their semantics.

As an example the function to create a thread is named `avixThread_Create` and to lock a mutex the function is named `avixMutex_Lock`.

All data types offered by AVIX are named like:

tavix<identification>

Where **<identification>** is a logical name for the type.

7.1.2 Type Safety

AVIX offers type safety for kernel object id's used as function parameters and function return values. The advantage of type safety is that a programming error is detected compile time instead of runtime. This leads to faster development and simpler and easier to comprehend code. A thread for instance is identified by an id of type `tavixThreadId`. In case a variable of this type is accidentally passed to a mutex function, the compiler will generate an error. More information on type safety and usage of these types in your application is found in §**Error! Reference source not found.**6.5.1.

7.1.3 When is an AVIX function allowed to be used?

AVIX offers three active entities, threads, ISR's and DIH's. Furthermore, before AVIX actually takes control, an initialization function called `avixMain` is executed. From this initialization function kernel objects like threads, semaphores etc. can be created.

Not every AVIX function is allowed to be used by every type of active entity or by `avixMain;` also for certain types of usage restrictions are applicable.

A thread is an active entity that can be preempted by another thread. One of the reasons for this to happen is when a thread claims a resource, not being available at the moment of the call. The thread is blocked until either the resource becomes available or an optional timeout expires.

ISR's and DIH's on the other hand, once started, must run to completion. These entities cannot be blocked like threads. Therefore, AVIX functions that potentially cause the caller to enter the 'Blocked' state cannot be used from these activate entities or only in such a way the function is guaranteed not to Block its caller.

Each of the AVIX function descriptions is accompanied by a small table specifying whether the function is allowed to be used from each of the four types of active entity. The following convention is used.

- When the function is allowed to be called the name of the active entity is marked with ✓.
- When additional information for the active entity is present in the form of a footnote the name of the active entity is shown in an **orange** color together with ✓
- When the function is not allowed to be called the name of the active entity is marked with -.
- For threads only, when the function can optionally be used with a timeout, the following icon is shown: 04:02. *Note this icon identifies optional use of a timeout. When not using a timeout, the function will only return when the desired resource is available, effectively being equal to an infinite timeout.*
- For DIH's only, when the function must be followed by a test whether it succeeded the following icon is shown: ?. The test to perform depends on the performed operation. For functions returning a kernel object id, the returned id must be tested for being valid using macro `AVIX_OBJECT_ID_VALID`. For other functions like pipe read and write, the result value of the function must be tested for the actual number of blocks being transferred. *Note this icon identifies an obligatory test.*

Below each of the function descriptions is a text '>> function overview'. When reading this document electronically, this is a link, selection of which brings you to a table containing an overview of all functions in the AVIX API.

Below a sample function description is shown. The interpretation of this description is the following:

- `avixSome_Function` may not be used from `avixMain`.
- `avixSome_Function` may be used from a thread, where optionally timeouts may be used.
- `avixSome_Function` may be used from a DIH where testing if the function has succeeded is obligatory. The test must be done using macro `AVIX_OBJECT_ID_VALID`. Furthermore, additional information is present in the form of a footnote.
- `avixSome_Function` may not be used from an ISR.

```
void avixSome_Function
(
    tavixSomeType yetAnotherParam
);
```

avixMain	-	
Thread	✓	04:02
DIH[18]	✓	?
ISR	-	

>> function overview

AVIX does not check whether the active entity calling a function is allowed to do so. If an active entity calls a function it is not allowed to call, system behavior becomes unpredictable and is likely to lead to system failure or system hang-up. It is the programmer's responsibility to check for this.

[18] Sample footnote

7.2 Function Overview

Table 5 presents an overview of all functions available in the AVIX API. With each function is specified the page number where a detailed description is found and for each of the active entities types whether it is allowed to use the function. When reading this document in electronic form, each of the function names is a link that, when selected, brings you to the applicable section.

THREAD FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixThread_ArmTimeOut	127	–	✓	–	–
avixThread_Create	128	✓	✓	✓	–
avixThread_Get	129	–	✓ 04:02	✓ ?	–
avixThread_GetIdCurrent	130	–	✓	–	–
avixThread_PulseTracePort	131	–	✓	–	–
avixThread_Relinquish	132	–	✓	–	–
avixThread_Resume	133	✓	✓	✓	–
avixThread_ResumeFromISR	134	–	–	–	✓
avixThread_SetTracePort	135	✓	✓	–	–
avixThread_SetTracePortAndResume	136	✓	✓	–	–
avixThread_Sleep	137	–	✓	–	–
avixThread_Suspend	138	–	✓	–	–
avixThread_TimeOutOccured	139	–	✓	–	–

MUTEX FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMutex_Create	142	✓	✓	✓	–
avixMutex_Get	143	–	✓ 04:02	✓ ?	–
avixMutex_Lock	144	–	✓	–	–
avixMutex_Unlock	145	–	✓	–	–

SEMAPHORE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixSemaphore_Create	148	✓	✓	✓	–
avixSemaphore_Get	149	–	✓ 04:02	✓ ?	–
avixSemaphore_Lock	150	–	✓ 04:02	–	–
avixSemaphore_Unlock	151	–	✓	✓	–
avixSemaphore_UnlockFromISR	152	–	–	–	✓

EVENT GROUP FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixEventGroup_Change	155	✓	✓	✓	–
avixEventGroup_ChangeFromISR	156	–	–	–	✓
avixEventGroup_Create	157	✓	✓	✓	–
avixEventGroup_Get	158	–	✓ 04:02	✓ ?	–
avixEventGroup_GetEventFlags	159	–	✓	✓	–
avixEventGroup_Wait	160	–	✓ 04:02	–	–

TIMER FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixTimer_ConnectEventGroup	164	✓	✓	✓	–
avixTimer_Create	165	✓	✓	✓	–
avixTimer_DisconnectEventGroup	166	✓	✓	✓	–
avixTimer_Get	167	–	✓ 04:02	✓	–
avixTimer_GetRemainingTicks	168	✓	✓	✓	–
avixTimer_Resume	169	✓	✓	✓	–
avixTimer_ResumeFromISR	170	–	–	–	✓
avixTimer_Set	171	✓	✓	✓	–
avixTimer_SetPeriod	172	✓	✓	✓	–
avixTimer_Start	173	✓	✓	✓	–
avixTimer_StartFromISR	174	–	–	–	✓
avixTimer_Stop	175	–	✓	✓	–
avixTimer_StopFromISR	176	–	–	–	✓
avixTimer_Wait	177	–	✓	–	–

MESSAGE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMsg_Allocate	181	–	✓ 04:02	✓	–
avixMsg_AllocateFromISR	182	–	–	–	✓
avixMsg_Free	183	–	✓	✓	–
avixMsg_GetChar	184	–	✓	✓	–
avixMsg_GetIndirect	185	–	✓	✓	–
avixMsg_GetInt	186	–	✓	✓	–
avixMsg_GetKernelObjectId	187	–	✓	✓	–
avixMsg_GetLong	188	–	✓	✓	–
avixMsg_GetPtr	189	–	✓	✓	–
avixMsg_GetSender	190	–	✓	✓	–
avixMsg_GetShort	191	–	✓	✓	–
avixMsg_GetType	192	–	✓	✓	–
avixMsg_PutChar<FromISR>[19]	193	–	✓	✓	✓
avixMsg_PutIndirect<FromISR>[19]	194	–	✓	✓	✓
avixMsg_PutInt<FromISR>[19]	195	–	✓	✓	✓
avixMsg_PutKernelObjectId<FromISR>[19]	196	–	✓	✓	✓
avixMsg_PutLong<FromISR>[19]	197	–	✓	✓	✓
avixMsg_PutPtr<FromISR>[19]	198	–	✓	✓	✓
avixMsg_PutShort<FromISR>[19]	199	–	✓	✓	✓
avixMsg_Reuse	200	–	✓	✓	–
avixMsgQThread_ConnectEventGroup	201	✓	✓	✓	–
avixMsgQThread_DisconnectEventGroup	202	–	✓	✓	–
avixMsgQThread_Flush	203	–	✓	–	–
avixMsgQThread_Receive	204	–	✓ 04:02	–	–
avixMsgQThread_Reply	205	–	✓	–	–

[19] All avixMsg_Put… functions are allowed to be called from threads, DIH's and ISR's. The …<FromISR> version of those functions is a macro which evaluates to the regular function. The reason the …<FromISR> version is offered is to adhere to the convention that ISR's may only use …<FromISR> functions which aids in checking the application for correctness.

MESSAGE FUNCTIONS (continued)					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMsgQThread_Send	206	–	✓	✓	–
avixMsgQThread_SendFromISR	207	–	–	–	✓

PIPE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixPipe_AbortAsyncReq	209	–	✓	✓	–
avixPipe_Create	210	✓	✓	✓	–
avixPipe_FlushAndAbort	211	–	✓	✓	–
avixPipe_Get	212	–	✓ 04:02	✓ ?	–
avixPipe_GetStatusAsyncReq	213	–	✓	✓	–
avixPipe_Read	214	–	✓ 04:02	✓ ?	–
avixPipe_ReadAsync	216	–	✓	✓ ?	–
avixPipe_ReadFromISR	218	–	–	–	✓
avixPipe_SetHandlerTracePort	220	✓	✓	–	–
avixPipe_StopDeviceFromISR	221	–	–	–	✓
avixPipe_Write	222	–	✓ 04:02	✓ ?	–
avixPipe_WriteAsync	224	–	✓	✓ ?	–
avixPipe_WriteFromISR	224	–	–	–	✓

MEMORY FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMemPool_Allocate	230	–	✓ 04:02	✓ ?	–
avixMemPool_AllocateFromISR	231	–	–	–	✓
avixMemPool_Create	232	✓	✓	✓	–
AvixMemPool_CreateExt[20]	233	✓	✓	✓	–
avixMemPool_Free	234	–	✓	✓	–
avixMemPool_FreeFromISR	235	–	–	–	✓
avixMemPool_Get	236	–	✓ 04:02	✓ ?	–
avixMemPool_GetSizeBlock	237	–	✓	✓	–
avixMemPool_GetSizeBlockFromISR	238	–	–	–	✓

EXCHANGE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixExch_ConnectCallbackEventGroup	241	✓	✓	–	–
avixExch_ConnectCallbackPipe	243	✓	✓	–	–
avixExch_ConnectCallbackThread	245	✓	✓	–	–
avixExch_ConnectEventGroup	247	✓	✓	–	–
avixExch_ConnectMsgQThread	248	✓	✓	–	–
avixExch_ConnectPipe	249	✓	✓	–	–
avixExch_Create	250	✓	✓	–	–
avixExch_DisableActiveConnection	251	–	✓	–	–
avixExch_DisableAllConnections	252	✓	✓	–	–
avixExch_DisableConnection	253	✓	✓	–	–
avixExch_EnableAllConnections	254	✓	✓	–	–

[20] Functionality is hardware platform dependant. Consult the applicable Port Guide for details.

EXCHANGE FUNCTIONS (continued)					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixExch_EnableConnection	255	✓	✓	–	–
avixExch_Get	256	✓	✓ 04:02	–	–
avixExch_GetConnectionExch	257	✓	✓	–	–
avixExch_GetConnectionMode	258	✓	✓	–	–
avixExch_GetLastWriteThread	259	✓	✓	–	–
avixExch_GetSize	260	✓	✓	–	
avixExch_Lock	261	✓	✓	–	–
avixExch_Read	262	✓	✓	–	–
avixExch_Unlock	263	✓	✓	–	–
avixExch_UnlockAndNotify	264	✓	✓	–	–
avixExch_Write	265	✓	✓	–	–

POWER MANAGEMENT FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixPower_GetMode	269	✓	✓	✓	–
avixPower_GetModeFromISR	270	–	–	–	✓
avixPower_SetCallback	271	✓	✓	✓	–
avixPower_SetMode	272	✓	✓	✓	–
avixPower_SetModeFromISR	273	–	–	–	✓

INTERRUPT FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixDIH_Queue	276	✓	–	✓	✓

DIAGNOSTIC FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixError_SetHandler	279	✓	✓	✓	✓
avixError_Throw	280	✓	✓	✓	✓

Table 5: AVIX API Functions

✓ Function is allowed to be called

✓ Function is allowed to be called but special care must be taken, read detailed description

– Function is not allowed to be called

04:02 Use **can** be made of an optional time-out

Use **must** be made of an obligatory test on kernel object id validity

7.3 Definition Overview

Table 6 contains an overview of all definitions that can be used with AVIX. When reading this document in electronic form, each of the service categories is a link that on selection brings you to the applicable section.

THREAD DEFINITIONS	
Definition	**Description**
AVIX_THREAD_READY	Flag to create thread not suspended
AVIX_THREAD_SUSPENDED	Flag to create thread suspended
AVIX_TIMEOUT	Macro to use a function with timeout
AVIX_TIMEOUT_WAIT_FOREVER	Definition for timeout usage to wait until resource is available
AVIX_TRACE_PORT_A	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_B	Trace port macro for thread activation tracing
....	Trace port macro for thread activation tracing
....	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_N	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_O	Trace port macro for thread activation tracing
AVIX_TRACE_NONE	Trace port macro for thread to set no thread activation tracing
TAVIX_THREAD_REGULAR	Declare a function that will be used as a thread
TRACE_IDLE_THREAD_ID	Trace port macro for setting a trace port on the idle thread

MUTEX DEFINITIONS	
Definition	**Description**
AVIX_MUTEX_UNLOCKED	Flag to create mutex initially unlocked
AVIX_MUTEX_LOCKED	Flag to create mutex initially locked

SEMAPHORE DEFINITIONS	
Definition	**Description**
AVIX_SEMAPHORE_UNLOCKED	Flag to create semaphore initially unlocked
AVIX_SEMAPHORE_LOCKED	Flag to create semaphore initially locked

EVENT GROUP DEFINITIONS	
Definition	**Description**
AVIX_EF	Event flags based on number between 0 and 15
AVIX_EF_ALL	Event flags all 1
AVIX_EF_IN	Compare if flags set is subset of other
AVIX_EF_IN_MASKED	Compare if flags set is subset of other over specific range
AVIX_EF_INVERT_ALL	Invert all flags
AVIX_EF_INVERT_MASKED	Invert range in event flags
AVIX_EF_IS	Compare two event flags, all flags
AVIX_EF_IS_MASKED	Compare two event flags for range of specific flags
AVIX_EF_NONE	Event flags all 0
AVIX_EF_RANGE	Event flags based on range, all flags in range are 1
AVIX_EVENT_GROUP_ALL	Flag to wait for all specified event group flags
AVIX_EVENT_GROUP_ANY	Flag to wait for any specified event group flag
AVIX_EVENT_GROUP_CLEAR	Flag to clear specified flags
AVIX_EVENT_GROUP_SET	Flag to set specified flags
AVIX_EVENT_GROUP_TOGGLE	Flag to toggle specified flags

TIMER DEFINITIONS	
Definition	**Description**
AVIX_DELAY_MS	Generate nr. of ticks for specified milliseconds
AVIX_DELAY_MS_US	Generate nr. of ticks for specified milli- microseconds
AVIX_DELAY_S	Generate nr. of ticks for specified seconds
AVIX_DELAY_S_MS	Generate nr. of ticks for specified seconds- milliseconds
AVIX_DELAY_S_MS_US	Generate nr. of ticks for specified seconds- milli- microseconds
AVIX_DELAY_US	Generate nr. of ticks for specified microseconds
AVIX_SYS_CLOCK_ACTUAL_PERIOD	Actual system clock period based on timer resolution
AVIX_TIMER_CYCLIC	Definition to set a timer as cyclic
AVIX_TIMER_MAX_NR_TICKS	Maximum tick value for timer
AVIX_TIMER_SINGLE_SHOT	Definition to set a timer as single shot

PIPE DEFINITIONS	
Definition	**Description**
PIPE_ASYNCREQ_ABORTED	Status when async request is aborted
PIPE_ASYNCREQ_FINISHED	Status when async request is finished
PIPE_ASYNCREQ_PENDING	Status when async request is pending
PIPEINFO_DEVICE_STOP_REQUESTED	Callback value for avixPipe_StopDeviceFromISR
PIPEINFO_PIPE_DATA_WRITTEN	Callback value when a thread writes data to a pipe
PIPEINFO_READ_FROM_EMPTY_PIPE	Callback value when a thread reads from an empty pipe

MEMORY POOL DEFINITIONS	
Definition	**Description**
AVIX_MEM_BLOCK_PTR	Convert a memory block id to a ‘C’ style pointer
AVIX_MEM_BLOCK_PTR_EXT[21]	Convert an id of an extended memory block to a ‘C’ style pointer
AVIX_EDS[21]	Tag a pointer to a memory block allocated in Extended RAM
AVIX_MEM_BLOCK_ID_VALID	Test if a memory block id is valid.

EXCHANGE DEFINITIONS	
Definition	**Definition**
AVIX_EXCH_DATA_PTR	Convert an Exchange data pointer to ‘C’ style pointer
AVIX_EXCH_DATA_PTR_READ	Convert an Exchange data pointer to ‘C’ style pointer to const

POWER MANAGEMENT DEFINITIONS	
Defnition	**Description**
AVIX_POWER_REDUCTION_NONE	No power reduction mode will be used
AVIX_POWER_REDUCTION_LOW	Controller will switch to low power mode basic energy saving
AVIX_POWER_REDUCTION_HIGH	Controller will switch to low power mode high energy saving

DIAGNOSTIC DEFINITIONS	
Defnition	**Description**
AVIX_ASSERT	Check a condition and throw error if false. (conditional macro)
AVIX_ASSERT_ALWAYS	Check a condition and throw error if false.

INTERRUPT DEFINITIONS	
Definition	**Description**
avixDeclareISR	Declare an ISR using the AVIX system stack
avixDeclareISRShadow	Declare an ISR using the AVIX system stack and shadow reg.

[21] Functionality is hardware platform dependant. Consult the applicable Port Guide for details.

MISCELLANEOUS DEFINITIONS	
Definition	**Description**
AVIX_OBJECT_ID_DEFINE	Declare a kernel object id guaranteed to be invalid
AVIX_OBJECT_ID_VALID	Test for a kernel object id to be valid
AVIX_OBJECT_ID_NULL	Returns a 'NULL' object id
AVIX_TYPESAFE_EQ	Compare two typesafe variables on equality
AVIX_TYPESAFE_FROM_VOID	Convert void* to typesafe variable
AVIX_TYPESAFE_NEQ	Compare two typesafe variables on inequality
AVIX_TYPESAFE_TO_VOID	Convert typesafe variable or constant to void*

Table 6: AVIX API Definitions

7.4 Type Overview

Table 7 contains an overview of the AVIX defined types. If a type is defined in a 'type safe' manner, this is identified by an ✓ in column 'Type Safe'.

TYPE	DESCRIPTION	TYPE SAFE
tavixDIH	Function pointer type. 'C' functions used as a DIH must adhere to this signature.	
tavixErrorCode	Integer type for AVIX error codes	
tavixEventFlags	Type specifying a 16 bit field of flags used with event group functions.	
tavixEventGroupCombine	Type used to identify when waiting for event flags what the type of operation is when the desired bitmask is found. Possible values: • AVIX_EVENT_GROUP_ALL: When waiting for multiple bits, all specified bits must be set for the wait functions to succeed (comparable to a logical AND). • AVIX_EVENT_GROUP_ANY: When waiting for multiple bits, at least one of the specified bits must be set for the function to succeed (comparable to a logical OR).	
tavixEventGroupOperation	Type used to identify the desired operation with one of the EventGroup class _Change operations. Possible values: • AVIX_EVENT_GROUP_CLEAR: The bits specified in parameter eventFlags in the _Change operations are cleared. • AVIX_EVENT_GROUP_SET: The bits specified in parameter eventFlags in the _Change operations are set. • AVIX_EVENT_GROUP_TOGGLE: The bits specified in parameter eventFlags in the _Change operations are toggled.	
tavixEventId	Type used to identify an event kernel object	✓
tavixExchCallbackEventGroup	Type used for an Exchange callback function receiving an event group id as parameter.	
tavixExchCallbackThread	Type used for an Exchange callback function receiving a thread id as parameter.	
tavixExchCallbackPipe	Type used for an Exchange callback function receiving a pipe id as parameter.	
tavixExchConnId	Type used to identify an exchange connection	✓
tavixExchId	Type used to identify an exchange object	✓
tavixKernelObjectId	Generic kernel object id used to put all possible kernel object types in a message	
tavixKernelObjectIdp	Generic kernel object id pointer used to get all possible kernel object id types from a message	

TYPE	DESCRIPTION	TYPE SAFE
`tavixKernelObjectName`	Type used to pass a kernel object name to a Create or a Get function.	
`tavixMemBlockId`	Type used to identify a memory block as managed in a memory pool.	✓
`tavixMemPoolId`	Type used to identify a memory pool kernel object	✓
`tavixMsgId`	Type used to identify a message object	✓
`tavixMsgType`	Type used to identify message content	
`tavixMutexId`	Type used to identify a mutex kernel object	✓
`tavixMutexLocked`	Flag identifying whether the mutex is locked as part of the create function: • `AVIX_MUTEX_LOCKED`: Mutex is locked as part of its creation. • `AVIX_MUTEX_UNLOCKED`: Mutex is not locked as part of its creation.	
`tavixPipeCallback`	Type used for a callback function used from a pipe. Pipe callback functions must adhere to this signature.	
`tavixPipeEvent`	Event type used with pipe callback to specify reason of call	
`tavixPipeId`	Type used to identify a pipe kernel object.	✓
`tavixPriority`	Type used for thread priority.	
`tavixPowerCallbackFunc`	Type for a power mode callback function	
`tavixPowerMode`	Type used to identify controller power mode. Possible values are: • AVIX_POWER_MODE_ACTIVE • AVIX_POWER_MODE_IDLE • AVIX_POWER_MODE_SLEEP	
`tavixSemaphoreId`	Type used to identify a semaphore kernel object	✓
`tavixSemaphoreLocked`	Flag identifying whether the semaphore is locked as part of the create function: • `AVIX_SEMAPHORE_LOCKED`: Semaphore is locked as part of its creation. • `AVIX_SEMAPHORE_UNLOCKED`: Semaphore is not locked as part of its creation.	
`tavixThreadFuncType`	Function pointer type. 'C' functions that are started as a thread must adhere to this signature.	
`tavixThreadId`	Type used to identify a thread kernel object	✓
`tavixThreadSuspended`	Type used to identify which state a thread is created in: • `AVIX_THREAD_READY`: Thread is created subject to scheduling. • `AVIX_THREAD_SUSPENDED`: Thread is created in the suspended state. Must call avixThread_Resume for being scheduled again.	
`tavixThreadTracePort`	Type identifying the trace I/O ports that can be used for thread activation tracing.	
`tavixTimerId`	Type used to identify a timer kernel object	✓
`tavixTimerTick`	Type used for the basic timer ticks used with timer related functions.	
`tavixTimerType`	Type used to identify the type of timer. Allowed values are: • `AVIX_TIMER_SINGLE_SHOT`: Value to set a timer to be a single shot timer, once expired the timer does not automatically restart. • `AVIX_TIMER_CYCLIC`: Value to set a timer to be cyclic, once expired the timer does automatically restart.	

Table 7: AVIX API Types

7.5 Environment definitions

During compilation a number of symbols are defined that can be used in the application for the purpose of conditional building. These symbols are called environment definitions and are found in Table 8.

ENVIRONMENT DEFINITIONS	
Definition	**Description**
__AVIX_VERSION_MAIN__	Symbol having as its value the main AVIX version number in 32 bit numeric format
__AVIX_VERSION_SUB__	Symbol having as its value the minor AVIX version number in 32 bit numeric format
__AVIX_VERSION_PATCH__	Symbol having as its value the patch AVIX version number in 32 bit numeric format
__AVIX_VERSION__	Symbol having as its value a combination of __AVIX_VERSION_MAIN__, __AVIX_VERSION_SUB__ and __AVIX_VERSION_PATCH__ in 32 bit numeric format 0x00xxyyzz where: xx is the value of __AVIX_VERSION_MAIN__ yy is the value of __AVIX_VERSION_SUB__ zz is the value of __AVIX_VERSION_PATCH__
__AVIX_PIC24E__	Symbol defined when building for a controller belonging to the PIC24E target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC24F__	Symbol defined when building for a controller belonging to the PIC24F target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC24H__	Symbol defined when building for a controller belonging to the PIC24H target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC30F__	Symbol defined when building for a controller belonging to the dsPIC30F target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC33E__	Symbol defined when building for a controller belonging to the dsPIC33E target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC33F__	Symbol defined when building for a controller belonging to the dsPIC33F target platform. When building for another target platform, this symbol is not defined.
__AVIX_PIC32MX__	Symbol defined when building for a controller belonging to the PIC32MX target platform. When building for another target platform, this symbol is not defined.
__AVIX_CORTEX_M3__	Symbol defined when building for a controller belonging to the Cortex M3 target platform. When building for another target platform, this symbol is not defined.

Table 8: AVIX Environment Definitions

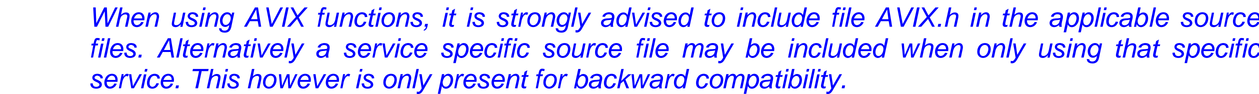
7.6 AVIX Application Programming Interface (API)

This section contains a description of all functions offered by AVIX. The functions are grouped by the service category they belong to.

To use any of the functions, definitions and types offered by AVIX, include header file AVIX.h.

Besides AVIX.h, use can be made of a service category specific header file. For instance when only using thread functions, it is sufficient to only include AVIXThread.h. This is however discouraged and it is strongly advised to use AVIX.h. The service category header files are only present for backward compatibility.

For completeness, with every service category in the coming sections, the service category specific header file is mentioned.



When using AVIX functions, it is strongly advised to include file AVIX.h in the applicable source files. Alternatively a service specific source file may be included when only using that specific service. This however is only present for backward compatibility.

7.6.1 Thread Support

Header file to include: AVIX.h
Service specific header file: AVIXThread.h

The following functions and definitions are present:

THREAD FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixThread_ArmTimeOut	127	-	✓	-	-
avixThread_Create	128	✓	✓	✓	-
avixThread_Get	129	-	✓ 04:02	✓	-
avixThread_GetIdCurrent	130	-	✓	-	-
avixThread_PulseTracePort	131	-	✓	-	-
avixThread_Relinquish	132	-	✓	-	-
avixThread_Resume	133	✓	✓	✓	-
avixThread_ResumeFromISR	134	-	-	-	✓
avixThread_SetTracePort	135	✓	✓	-	-
avixThread_SetTracePortAndResume	136	✓	✓	-	-
avixThread_Sleep	137	-	✓	-	-
avixThread_Suspend	138	-	✓	-	-
avixThread_TimeOutOccured	139	-	✓	-	-

THREAD DEFINITIONS	
Definition	**Description**
AVIX_THREAD_READY	Flag to create thread not suspended
AVIX_THREAD_SUSPENDED	Flag to create thread suspended
AVIX_TIMEOUT	Macro to use a function with timeout
AVIX_TIMEOUT_WAIT_FOREVER	Definition for timeout usage to wait until resource is available
AVIX_TRACE_PORT_A	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_B	Trace port macro for thread activation tracing
....	Trace port macro for thread activation tracing
....	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_N	Trace port macro for thread activation tracing
AVIX_TRACE_PORT_O	Trace port macro for thread activation tracing
AVIX_TRACE_NONE	Trace port macro for thread to set no thread activation tracing
TAVIX_THREAD_REGULAR	Declare a function that will be used as a thread
TRACE_IDLE_THREAD_ID	Trace port macro for setting a trace port on the idle thread

avixThread_ArmTimeOut

`void avixThread_ArmTimeOut`
`(`
`    tavixTimerTick ticks`
`);`
	avixMain –
	Thread ✓
	DIH –
	ISR –

>> function overview

Description

Set a timeout value to be used with the (potentially) blocking function succeeding this function. Accuracy and resolution of the timeout mechanism is equal to that of regular timers. Instead of specifying the number of ticks, use can be made of the timer support macros allowing a time to be specified in second based units. These macros are specified in section 'Timer related definitions' on page 178.

▶ *When using this function, it must always be followed by a (potentially) blocking function call which in turn must be followed by a call to function `avixThread_TimeOutOccurred`. Throughout this document functions that may be used with a timeout are identified with this icon: 04:02*

▶ *This function may not be used by Exchange callback functions. Potentially blocking functions called from Exchange callback functions behave as if the time out is set zero. This is taken care of by the Exchange services and as a result, these functions will never block. Calling `avixThread_ArmTimeOut` from an Exchange callback function results in an error.*

Parameters and return value

Parameter	Description
ticks	Number of kernel ticks to wait. The value is specified in the number of ticks of the AVIX system timer. The value for this parameter must be >= 0 and <= `AVIX_TIMEOUT_WAIT_FOREVER`. Specifying `AVIX_TIMEOUT_WAIT_FOREVER` means 'wait forever' meaning the succeeding function will only return when the resource it waits for is available. This is also default behavior when not using a timeout. Specifying a value of 0 means the blocking function following this function will never block and return immediately with either the desired resource or a timeout. Effectively with this value the resource is 'polled'. In order to specify the value of this parameter in a time based on seconds, milliseconds or microseconds, use can be made of the timer utility macros specified in section 'Timer related definitions' on page 178.
Return value	<none>

▶ *In contradiction with regular timers, timeout timers allow a value of zero (0) to be used for the ticks parameter. This value arms the timeout mechanism such that when the resource waited for is not available, the timer is not started but the function will assume a timeout situation to exist.*

avixThread_Create

<pre>tavixThreadId avixThread_Create
(
    tavixkernelObjectName pName,
    tavixThreadFuncType   pThread,
    void*   threadParam,
    tavixPriority   iPriority,
    unsigned int   iStackSize,
    tavixThreadSuspended fSuspended
);</pre>
	avixMain ✓
	Thread ✓
	DIH ✓
	ISR -

>> function overview

Description

Create a thread kernel object. If a valid name is specified and other threads are waiting for the created thread to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

The newly created thread can be in one of two states. If parameter `fSuspended` equals `AVIX_THREAD_READY`, after being created, the new thread will immediately be subjected to scheduling as implemented by AVIX (that is if `avixMain` has finished). If parameter `fSuspended` equals `AVIX_THREAD_SUSPENDED`, the thread is created but will not be subject to scheduling. In this second case, in order to be scheduled the thread must first be resumed by calling `avixThread_Resume`.

Parameters and return value

Parameter	Description
pName	Human readable name for the thread or `NULL` if no name is required. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
pThread	Address of the function containing the threads code. The thread function must be declared using macro `TAVIX_THREAD_REGULAR` like shown below: `TAVIX_THREAD_REGULAR functionName(void* threadArg){…}`
threadParam	Parameter typed as a `void*` used to pass initialization data to the thread. Any desired type can be passed as long as it can be represented in a `void*`. For passing a type safe parameter, use can be made of macro `AVIX_TYPESAFE_TO_VOID`. In the thread this can be converted back to the correct type with `AVIX_TYPESAFE_FROM_VOID`.
iPriority	Priority used to schedule the thread. Any value from 1 up to and including the value configured through configuration parameter `avix_MAX_PRIORITY` can be used where a higher number means a higher thread priority (more important)
iStackSize	Size of the stack in bytes. The specified size is scaled to adhere to the alignment requirements of the microcontroller in use.
fSuspended	`AVIX_THREAD_SUSPENDED`, thread will not be scheduled until a call to `avixThread_Resume` has been made. `AVIX_THREAD_READY`, the thread will be controlled by AVIX as soon as AVIX starts running.
Return value	Id representing the newly created thread.

avixThread_Get

```
tavixThreadId avixThread_Get
(
    tavixkernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Obtain the id of a thread kernel object with a specific name. If the thread with the specified name exists the function returns immediately with a thread id guaranteed to be valid. If the thread with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated thread is created or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the thread does not exist and this function is called from a DIH, the function returns immediately with an invalid thread id.

Parameters and return value

Parameter	Description
pName	Human readable name for the thread whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the thread with the specified name (pName).

avixThread_GetIdCurrent

```
tavixThreadId avixThread_GetIdCurrent(void);
```

avixMain	–
Thread	✓
DIH	–
ISR	–

>> function overview

Description

Get the id of the currently running thread.

Parameters and return value

Parameter	Description
<none>	
Return value	Id of the calling thread

avixThread_PulseTracePort

```
void avixThread_PulseTracePort(void);
```

avixMain	–
Thread	✓
DIH	–
ISR	–

>> function overview

Description

For the current active thread, pulse the trace port.

When using thread activation tracing, the trace port for the current active thread is high while the trace ports for all other threads are low. Using this function, the port for the current thread is pulled low for a very short time, resulting in a ‘spike’ that can be observed on a logic analyzer. By measuring the time between this spike and other trace events, performance measurements can be executed.

Parameters and return value

Parameter	Description
<none>	
Return value	<none>

avixThread_Relinquish

`void avixThread_Relinquish(void);`	**avixMain** – **Thread** ✓ **DIH** – **ISR** –

>> function overview

Description

In case there are more ready threads having the same thread priority as the calling thread, this call places the calling thread at the tail of its ready list, effectively deactivating it and activating the next ready thread at that priority.

This call has no effect if the calling thread is the only thread having the specific thread priority or if the calling thread is raised in priority since it owns a mutex needed by a higher priority thread.

When AVIX is configured to have a round robin period of zero (0), round robin scheduling is effectively disabled. In this case an active thread can voluntarily give up processing by calling this function. This allows for a cooperative scheduling mechanism to be implemented.

Parameters and return value

Parameter	Description
<none>	
Return value	<none>

avixThread_Resume

```
void avixThread_Resume
(
    tavixThreadId id
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Resume a suspended thread. A thread can become suspended either since it is created in a suspended state (parameter `fSuspended` for `avixThread_Create` equals `AVIX_THREAD_SUSPENDED`) or the active thread calls `avixThread_Suspend`.

Calling this function for a thread that is not suspended has no effect.

Parameters and return value

Parameter	Description
id	Id of the thread to resume
Return value	<none>

avixThread_ResumeFromISR

```
void avixThread_ResumeFromISR
(
    tavixThreadId id
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Resume a suspended thread directly from an ISR. A thread can become suspended either since it is created in a suspended state (parameter `fSuspended` for `avixThread_Create` equals `AVIX_THREAD_SUSPENDED`) or the active thread calls `avixThread_Suspend`.

Calling this function for a thread that is not suspended has no effect.

Parameters and return value

Parameter	Description
id	Id of the thread to resume
Return value	<none>

avixThread_SetTracePort

```
avixThread_SetTracePort
(
    tavixThreadId        thread,
    tavixThreadTracePort port,
    unsigned int         bit
);
```

avixMain	✓
Thread	✓
DIH	–
ISR	–

>> function overview

Description

Assign or de-assign one of the controller's digital I/O pins to a thread so the scheduler can set this I/O pin in accordance with the thread activation for tracing purposes.

When creating the thread and assigning the trace I/O port from `avixMain`, the I/O port is guaranteed to be assigned when the thread starts running. When however the thread is created from another thread and the newly created thread has a higher thread priority, it immediately starts running after being created. This preemption takes place before the creating thread has assigned the trace I/O port so the new thread starts running without tracing. Only after the creating thread has got another chance to run, the new thread has its trace I/O port assigned. This can imply that the start period of the new thread is not traced.

To solve this, a second function is offered, `avixThread_SetTracePortAndResume`.

▶ *Depending on the hardware platform being used, Thread Activation Tracing works 'out of the box' or some configuration settings might be needed. Consult the hardware platform specific Port Guide for details.*

Parameters and return value

Parameter	Description
thread	Id of the thread an I/O port is assigned to. When setting a trace port for the idle thread use `TRACE_IDLE_THREAD_ID`.
port	Use one of the following values: `AVIX_TRACE_PORT_A, AVIX_TRACE_PORT_B, … AVIX_TRACE_PORT_N, AVIX_TRACE_PORT_O or AVIX_TRACE_NONE.` Not all controllers have all these I/O ports. It is the user's responsibility to select an I/O port that is present in the used controller. Furthermore, make sure to select an I/O port/bit combination not used by the application.
bit	Bit number in the selected I/O port that is related to the thread. Not all I/O ports have all bits available as digital I/O. It is the user's responsibility to select a bit available in the selected I/O port and make sure the I/O port/bit combination is not used by the application.
Return value	<none>

avixThread_SetTracePortAndResume

```
avixThread_SetTracePortAndResume
(
    tavixThreadId        thread,
    tavixThreadTracePort port,
    unsigned int         bit
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Assign or de-assign one of the controller's digital I/O ports to a thread so the scheduler can set this I/O port in accordance with the thread activation for tracing purposes.

When a thread is created from `avixMain`, it is advised to use `avixThread_SetTracePort`. When a thread is created from another thread and the newly created thread has a higher thread priority, it starts running immediately after being created. This preemption takes place before the creating thread has assigned the trace I/O port so the new thread starts running without tracing. Only after the creating thread has got another chance to run, the new thread has its trace I/O port assigned. This can imply that the start period of the new thread is not traced.

Function `avixThread_SetTracePortAndResume` exists to solve this. When tracing is desired and the thread to trace is created from another thread, it must be created suspended by using flag `AVIX_THREAD_SUSPENDED` in the create function.

Next `avixThread_SetTracePortAndResume` will both assign the trace I/O port and resume the new thread so it is guaranteed to start running with tracing activated.

► *Depending on the hardware platform being used, Thread Activation Tracing works 'out of the box' or some configuration settings might be needed. Consult the hardware platform specific Port Guide for details.*

Parameters and return value

Parameter	Description
thread	Id of the thread an I/O port is assigned to.
port	Use one of the following values: `AVIX_TRACE_PORT_A, AVIX_TRACE_PORT_B, … AVIX_TRACE_PORT_N, AVIX_TRACE_PORT_O or AVIX_TRACE_NONE.` Not all controllers have all these I/O ports. It is the user's responsibility to select an I/O port that is present in the used controller. Furthermore, make sure to select an I/O port/bit combination not used by the application.
bit	Bit number in the selected I/O port that is related to the thread. Not all I/O ports have all bits available as digital I/O. It is the user's responsibility to select a bit available in the selected I/O port and make sure the I/O port/bit combination is not used by the application.
Return value	<none>

avixThread_Sleep

`void avixThread_Sleep`
`(`
`    tavixTimerTick ticks`
`);`
	avixMain –
	Thread ✓
	DIH –
	ISR –

>> function overview

Description

Function to let a thread wait for number of system ticks. The thread will be suspended for the specified number of system ticks, after which it automatically enters the ‘Ready’ state again.

This function has the same effect as using an explicitly created timer for which a thread calls the sequence `avixTimerSet` (single shot mode) - `avixTimerStart` - `avixTimerWait`. This function does however not need a timer since it uses the thread internal timer that exists whenever a thread exists. This thread internal timer is used for this function and for optional timeouts.

Parameters and return value

Parameter	Description
ticks	Number of kernel ticks to wait. In order to specify the value of this parameter in a time based on seconds, milliseconds or microseconds, use can be made of the timer utility macros specified in section ‘Timer related definitions’ on page 178.
Return value	<none>

avixThread_Suspend

```
void avixThread_Suspend(void);
```

avixMain	–
Thread	✓
DIH	–
ISR	–

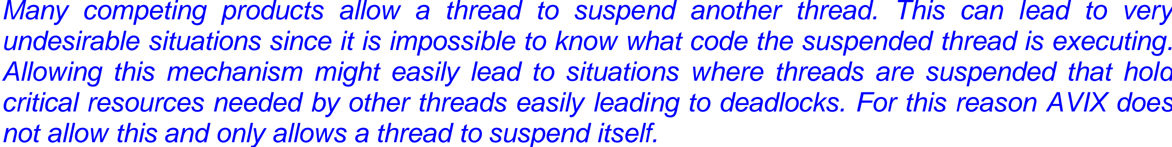
>> function overview

Description

Suspend the calling thread. A thread that calls this function becomes inactive. It no longer participates in scheduling and will not consume any CPU cycles. A thread that has suspended itself can only become active again when *another* thread calls `avixThread_Resume` or `avixThread_ResumeFromISR` for the suspended thread.

Parameters and return value

Parameter	Description
<none>	
Return value	<none>



Many competing products allow a thread to suspend another thread. This can lead to very undesirable situations since it is impossible to know what code the suspended thread is executing. Allowing this mechanism might easily lead to situations where threads are suspended that hold critical resources needed by other threads easily leading to deadlocks. For this reason AVIX does not allow this and only allows a thread to suspend itself.

avixThread_TimeOutOccured

`int avixThread_TimeOutOccured(void);`	**avixMain** – **Thread** ✓ **DIH** – **ISR** –

>> function overview

Description

Test whether a timeout situation occurred in the preceding function.

▶ *This function must be used immediately following a potential blocking function call which again must be preceded by a call to function* `avixThread_ArmTimeOut`.

Parameters and return value

Parameter	Description
<none>	
Return value	Integer value identifying whether a timeout occurred during execution of the preceding function. A value unequal to zero ('true') implies a timeout occurred. A value equal to zero ('false') implies no timeout occurred.

Thread related definitions

The following thread related definitions are provided: >> definition overview

AVIX_THREAD_READY
Flag to pass to avixThread_Create to create a thread not suspended ('Ready' state).

AVIX_THREAD_SUSPENDED
Flag to pass to avixThread_Create to create a thread suspended.

int AVIX_TIMEOUT(f, time);
Utility macro to ease usage of the timeout mechanism. The parameters to this macro are the function to which timeout functionality must be added including the result variable and the required parameters and the desired timeout value. The result of this macro is the result normally returned by avixThread_TimeOutOccured.

AVIX_TIMEOUT_WAIT_FOREVER
Definition for use with avixThread_ArmTimeOut to effectively disable the timeout. Using this definition the blocking function will wait until the resource waited for is available.

AVIX_TRACE_PORT_A … AVIX_TRACE_PORT_O AVIX_TRACE_NONE
Macros to pass as parameter 'port' to function avixThread_SetTracePort or function avixThread_SetTracePortResume to either set a specific trace I/O port or remove the tracing for the applicable thread.

TAVIX_THREAD_REGULAR
Macro used to declare a thread function which can be passed to avixThread_Create. TAVIX_THREAD_REGULAR funcName(void*) {…}

TRACE_IDLE_THREAD_ID
Macro to pass as the parameter 'thread' to avixThread_SetTracePort to enable tracing for the idle thread.

7.6.2 Mutex Support

Header file to include: AVIX.h
Service specific header file: AVIXMutex.h

The following functions and definitions are present:

MUTEX FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMutex_Create	142	✓	✓	✓	–
avixMutex_Get	143	–	✓ 04:02	✓	–
avixMutex_Lock	144	–	✓	–	–
avixMutex_Unlock	145	–	✓	–	–

MUTEX DEFINITIONS	
Definition	**Description**
AVIX_MUTEX_UNLOCKED	Flag to create mutex initially unlocked
AVIX_MUTEX_LOCKED	Flag to create mutex initially locked

avixMutex_Create

```
tavixMutexId avixMutex_Create
(
    tavixkernelObjectName pName,
    tavixMutexLocked      fInitiallyLocked
);
```

avixMain[22]	✓
Thread	✓
DIH[22]	✓
ISR	-

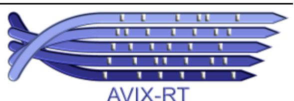
>> function overview

Description

Create a mutex kernel object. If a valid name is specified and other threads are waiting for the mutex to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

Parameter	Description
pName	Human readable name for the mutex or NULL if the mutex does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
fInitiallyLocked	If this flag equals AVIX_MUTEX_LOCKED, the thread that creates the mutex directly has the ownership (lockcount is 1). If AVIX_MUTEX_UNLOCKED, the creating thread has no lock on the mutex.
Return value	Id representing the newly created mutex.

[22] When called from inside avixMain, or a DIH, flag fInitiallyLocked **must** equal AVIX_MUTEX_UNLOCKED. Since avixMain and DIH's are not threads they can not be the owner of a mutex. Violation of this rule is not checked for and results in unpredictable behavior.



avixMutex_Get

```
tavixMutexId avixMutex_Get
(
    tavixkernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	?
ISR	–	

>> function overview

Description

Obtain the id of a mutex kernel object with a specific name. If the mutex with the specified name exists the function returns immediately with a mutex id guaranteed to be valid. If the mutex with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated mutex is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the mutex does not exist and this function is called from a DIH, the function returns immediately with an invalid mutex id.

▶ *Obtaining a mutex id from a DIH does not really make sense since a DIH is not allowed to use it. For reason of uniformity of the total API this function is however offered.*

Parameters and return value

Parameter	Description
pName	Human readable name for the mutex whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the mutex with the specified name (pName).

avixMutex_Lock

<table>
<tr><td>

```
void avixMutex_Lock
(
    tavixMutexId id
);
```

</td><td>

avixMain –
Thread ✓
DIH –
ISR –

</td></tr>
</table>

>> function overview

Description

Lock the specified mutex. If the mutex is not locked by another thread at the moment of calling this function, the thread gets the lock and returns. If the mutex is locked by another thread, the calling thread enters the 'Blocked' state and remains in this state until the owning thread has unlocked the mutex. This function may be called multiple times by the same thread. The actual locking of the mutex will be accomplished by the first call. Subsequent calls increase an internal lock count and do not block the thread. In order to unlock the mutex, the owning thread must make a number of calls to `avixMutex_Unlock` equaling the number of calls to `avixMutex_Lock`. This mechanism is to allow recursive functions to use a mutex.

When locking a mutex which is not available, in case the thread priority of the thread that owns the mutex is lower than the thread priority of the calling thread, the thread priority of the owning thread is raised to that of the calling thread (priority inheritance).

Parameters and return value

Parameter	Description
id	Id of the mutex to lock
Return value	<none>

AvixMutex_Unlock

```void avixMutex_Unlock (tavixMutexId id);```	**avixMain** – **Thread** ✓ **DIH** – **SR** –

```
void avixMutex_Unlock
(
    tavixMutexId id
);
```

avixMain	–
Thread	✓
DIH	–
SR	–

>> function overview

Description

Unlock the specified mutex. This function is only allowed if the thread owns the mutex, meaning that it acquired it before using a call to `avixMutex_Lock`. If threads are waiting for the mutex, the thread with the highest thread priority is given ownership of the mutex and will enter the 'Ready' state. Which of all threads having the 'Ready' state actually is allowed to execute is determined by the scheduler mechanism. To actually unlock the mutex, the owning thread must make a number of calls to `avixMutex_Unlock` equaling the number of calls it made to `avixMutex_Lock`. This mechanism is to allow recursive functions to use a mutex.

The thread priority of a thread having a mutex locked can be raised because the thread priority of one or more of the threads waiting for the mutex is higher (priority inheritance). When the thread unlocks the mutex its thread priority is restored to the value specified when creating the thread.

Parameters and return value

Parameter	Description
id	Id of the mutex to unlock
Return value	<none>

Mutex related definitions

The following mutex related definitions are provided: >> definition overview

AVIX_MUTEX_UNLOCKED
Flag to pass to avixMutex_Create to create a mutex initially not locked.

AVIX_THREAD_LOCKED
Flag to pass to avixMutex_Create to create a mutex initially locked.

7.6.3 Semaphore Support

Header file to include: AVIX.h
Service specific header file: AVIXSemaphore.h

The following functions and macros are present:

SEMAPHORE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixSemaphore_Create	148	✓	✓	✓	-
avixSemaphore_Get	149	-	✓ 04:02	✓ ?	-
avixSemaphore_Lock	150	-	✓ 04:02	-	-
avixSemaphore_Unlock	151	-	✓	✓	-
avixSemaphore_UnlockFromISR	152	-	-	-	✓

SEMAPHORE DEFINITIONS	
Definition	**Description**
AVIX_SEMAPHORE_UNLOCKED	Flag to create semaphore initially unlocked
AVIX_SEMAPHORE_LOCKED	Flag to create semaphore initially locked

avixSemaphore_Create

```
tavixSemaphoreId avixSemaphore_Create
(
    tavixKernelObjectName pName,
    int                   value,
    tavixSemaphoreLocked  fInitiallyLocked
);
```

avixMain[23]	✓
Thread	✓
DIH[23]	✓
ISR	-

>> function overview

Description

Create a semaphore kernel object. If a valid name is specified and other threads are waiting for the semaphore to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

The semaphore is initialized with the count specified in parameter value. If parameter `fInitiallyLocked` equals `AVIX_SEMAPHORE_LOCKED`, the count is set to value minus 1 meaning that the creating thread has a lock on the semaphore from the moment of creation. If parameter `fInitiallyLocked` equals `AVIX_SEMAPHORE_LOCKED` and parameter value is zero (0), the calling thread will block. In this case the thread will only enter the 'Ready' state again after another thread has unlocked the semaphore.

Parameters and return value

Parameter	Description
pName	Human readable name for the semaphore or `NULL` if the semaphore does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
fInitiallyLocked	When `AVIX_SEMAPHORE_LOCKED`, the creating thread immediately decrements the semaphore count and thus takes a lock. When `AVIX_SEMAPHORE_UNLOCKED`, the semaphore receives the value as specified by parameter value.
value	Initial count value the semaphore will receive. This value must be greater or equal to zero (0). Other values result in an error.
Return value	Id representing the newly created semaphore.

[23] When called from avixMain or a DIH, the combination of parameter 'value' being 0 and 'bInitiallyLocked' being `AVIX_SEMAPHORE_LOCKED` may not be used. This parameter combination would result in the caller being blocked. Neither avixMain nor a DIH is a thread and thus can not block. Violation of this rule is not checked for and results in unpredictable behavior.

avixSemaphore_Get

```
tavixSemaphoreId avixSemaphore_Get
(
    tavixKernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Obtain the id of a semaphore kernel object with a specific name. If the semaphore with the specified name exists the function returns immediately with a semaphore id guaranteed to be valid. If the semaphore with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the ‘Blocked’ state until either the designated semaphore is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the semaphore does not exist and this function is called from a DIH, the function returns immediately with an invalid semaphore id.

Parameters and return value

Parameter	Description
pName	Human readable name for the semaphore whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the semaphore with the specified name (pName).

avixSemaphore_Lock

<table>
<tr><td>

```
void avixSemaphore_Lock
(
    tavixSemaphoreId id
);
```

</td><td>

avixMain –
Thread ✓ 04:02
DIH –
ISR –

</td></tr>
</table>

>> function overview

Description

Take a lock on the semaphore specified by id. If the count of the semaphore is >0, the semaphore count is decremented by one (1) and the calling thread continues.

If the count of the semaphore is 0, the calling thread enters the 'Blocked' state until the semaphore is unlocked by another thread or DIH through function `avixSemaphore_Unlock` or by an ISR through function `avixSemaphore_UnlockFromISR`.

Parameters and return value

Parameter	Description
id	Id of the semaphore to lock
Return value	<none>

avixSemaphore_Unlock

```
void avixSemaphore_Unlock
(
    tavixSemaphoreId id
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Release a semaphore. The count of the semaphore is incremented by one. If threads are waiting for the semaphore, the first thread in the thread priority ordered wait queue enters the ‘Ready’ state. Which of all threads having the ‘Ready’ state actually is allowed to execute is determined by the scheduler mechanism.

Parameters and return value

Parameter	Description
id	Id of the semaphore to unlock
Return value	<none>

avixSemaphore_UnlockFromISR

```
void avixSemaphore_UnlockFromISR
(
    tavixSemaphoreId id
);
```

avixMain	-
Thread	-
DIH	-
ISR	✓

>> function overview

Description

Release a semaphore directly from an ISR. The count of the semaphore is incremented by one. If threads are waiting for the semaphore, the first thread in the thread priority ordered wait queue enters the 'Ready' state. Which of all threads having the 'Ready' state actually is allowed to execute is determined by the scheduler mechanism.

Parameters and return value

Parameter	Description
id	Id of the semaphore to unlock
Return value	<none>

Semaphore related definitions

The following semaphore related definitions are provided:

>> definition overview

AVIX_SEMAPHORE_UNLOCKED
Flag to pass to `avixSemaphore_Create` to create a semaphore initially not locked.

AVIX_SEMAPHORE_LOCKED
Flag to pass to `avixSemaphore_Create` to create a semaphore initially locked.

7.6.4 Event Support

Header file to include: AVIX.h
Service specific header file: AVIXEvent.h

The following functions and macros are present:

EVENT GROUP FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixEventGroup_Change	155	✓	✓	✓	–
avixEventGroup_ChangeFromISR	156	–	–	–	✓
avixEventGroup_Create	157	✓	✓	✓	–
avixEventGroup_Get	158	–	✓ 04:02	✓	–
avixEventGroup_GetEventFlags	159	–	✓	✓	–
avixEventGroup_Wait	160	–	✓ 04:02	–	–

EVENT GROUP DEFINITIONS	
Definition	**Description**
AVIX_EF	Event flags based on number between 0 and 15
AVIX_EF_ALL	Event flags all 1
AVIX_EF_IN	Compare if flags set is subset of other
AVIX_EF_IN_MASKED	Compare if flags set is subset of other over specific range
AVIX_EF_INVERT_ALL	Invert all flags
AVIX_EF_INVERT_MASKED	Invert range in event flags
AVIX_EF_IS	Compare two event flags, all flags
AVIX_EF_IS_MASKED	Compare two event flags for range of specific flags
AVIX_EF_NONE	Event flags all 0
AVIX_EF_RANGE	Event flags based on range, all flags in range are 1
AVIX_EVENT_GROUP_ALL	Flag to wait for all specified event group flags
AVIX_EVENT_GROUP_ANY	Flag to wait for any specified event group flag
AVIX_EVENT_GROUP_CLEAR	Flag to clear specified flags
AVIX_EVENT_GROUP_SET	Flag to set specified flags
AVIX_EVENT_GROUP_TOGGLE	Flag to toggle specified flags

avixEventGroup_Change

```
void avixEventGroup_Change
(
    tavixEventId                id,
    tavixEventGroupOperation operation,
    tavixEventFlags             eventFlags
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Change one or more event flags in an event group or a thread event group kernel object. Possible operations are:

- `AVIX_EVENT_GROUP_CLEAR`: Flags at positions corresponding to a set flag in parameter eventFlags are cleared. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.
- `AVIX_EVENT_GROUP_SET`: Flags at positions corresponding to a set flag in parameter eventFlags are set. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.
- `AVIX_EVENT_GROUP_TOGGLE`: Flags at positions corresponding to a set flag in parameter eventFlags are inverted. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.

After changing the event flags of the event group the threads in the wait queue of the event group are evaluated to see if because of the operation their requirement is met. This may result in waiting threads entering the ‘Ready’ state and the value of the event group being updated according the waiting criteria specified by these threads.

Which of all threads having the ‘Ready’ state actually is allowed to execute is determined by the scheduler mechanism.

Parameters and return value

Parameter	Description
id	Identification of the event group for which the flags have to be changed. This can either be the id of a regular event group or the id of a thread converted to an event group id (`.asEventId`) to identify the thread event group.
operation	Type of operation to execute on the event group flags, allowed values; `AVIX_EVENT_GROUP_CLEAR`, `AVIX_EVENT_GROUP_SET` or `AVIX_EVENT_GROUP_TOGGLE`
eventFlags	16-bit mask specifying the flags for the operation.
Return value	<none>

avixEventGroup_ChangeFromISR

```
void avixEventGroup_ChangeFromISR
(
    tavixEventId              id,
    tavixEventGroupOperation operation,
    tavixEventFlags           eventFlags
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Change one or more event flags in an event group or a thread event group kernel object directly from an ISR. Possible operations are:

- `AVIX_EVENT_GROUP_CLEAR`: Flags at positions corresponding to a set flag in parameter eventFlags are cleared. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.
- `AVIX_EVENT_GROUP_SET`: Flags at positions corresponding to a set flag in parameter eventFlags are set. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.
- `AVIX_EVENT_GROUP_TOGGLE`: Flags at positions corresponding to a set flag in parameter eventFlags are inverted. Flags at positions corresponding to a cleared flag in parameter event flags remain unchanged.

After changing the event flags of the event group the threads in the wait queue of the event group are evaluated to see if because of the operation their requirement is met. This may result in waiting threads entering the 'Ready' state and the value of the event group being updated according the waiting criteria specified by these threads.

Which of all threads having the 'Ready' state actually is allowed to execute is determined by the scheduler mechanism.

Parameters and return value

Parameter	Description
id	Identification of the event group for which the flags have to be changed. This can either be the id of a regular event group or the id of a thread converted to an event group id (`.asEventId`) to identify the thread event group.
operation	Type of operation to execute on the event group flags, allowed values; `AVIX_EVENT_GROUP_CLEAR`, `AVIX_EVENT_GROUP_SET` or `AVIX_EVENT_GROUP_TOGGLE`
eventFlags	16-bit mask specifying the flags for the operation.
Return value	<none>

avixEventGroup_Create

<table>
<tr><td>

```
tavixEventId avixEventGroup_Create
(
    tavixkernelObjectName pName,
    tavixEventFlags const initialEventFlags
);
```

</td><td>

avixMain ✓
Thread ✓
DIH ✓
ISR –

</td></tr>
</table>

>> function overview

Description

Create an event group kernel object. If a valid name is specified and other threads are waiting for event group object to become existent, those threads enter the ‘Ready’ state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

Note that every thread has got a local event group (thread event group) which is created the moment the thread is created so no explicit create operation is required. Access to a thread event group is obtained by converting the thread id to an event group id using attribute .asEventId on the thread id.

Parameters and return value

Parameter	Description
pName	Human readable name for the event group or `NULL` if the event group does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
initialEventFlags	16 bit value used to set the initial values for the flags maintained by the newly created event group.
Return value	Id representing the newly created event group.

avixEventGroup_Get

```
tavixEventId avixEventGroup_Get
(
    tavixkernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Obtain the id of an event group kernel object with a specific name. If the event group with the specified name exists the function returns immediately with an event group id guaranteed to be valid. If the event group with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated event group is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the event group does not exist and this function is called from a DIH, the function returns immediately with an invalid event group id.

Parameters and return value

Parameter	Description
pName	Human readable name for the event group whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the event group with the specified name (pName).

avixEventGroup_GetEventFlags

```
tavixEventFlags avixEventGroup_GetEventFlags
(
    tavixEventId    id,
    tavixEventFlags postClearMask
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Get the current status of the flags in the identified event group or thread event group. Optionally, specified flags can be cleared.

Using this function, the status of the event flags are effectively 'polled' for.

Parameters and return value

Parameter	Description
id	Identification of the event group for which the flags have to be changed. This can either be the id of a regular event group or the id of a thread converted to an event group id (`.asEventId`) to identify the thread event group.
postClearMask	16-bit mask specifying which flags to clear before the function returns. Use `AVIX_EF_NONE` when no flags may be cleared.
Return value	16-bit value representing the event group flags contained in the event group.

avixEventGroup_Wait

```
tavixEventFlags avixEventGroup_Wait
(
    tavixEventId            id,
    tavixEventFlags         desiredEventFlags,
    tavixEventGroupCombine  combine,
    tavixEventFlags         preClearMask,
    tavixEventFlags         postClearMask
);
```

avixMain	–	
Thread	✓	04:02
DIH	–	
ISR	–	

>> function overview

Description

Wait for a specific set of event flags in an event group kernel object to be set. The desired flags are specified by parameter desiredEventFlags. Parameter combine specifies whether all flags (`AVIX_EVENT_GROUP_ALL`) or at least one of the flags (`AVIX_EVENT_GROUP_ANY`) must be set for the wait to be honored. Before this check is done, the flags in the event group corresponding with a logical one in parameter preClearMask are cleared. After the wait operation succeeds, the flags in the event group corresponding with a logical one in parameter postClearMask are cleared.

Parameters and return value

Parameter	Description
id	Identification of the event group whose flags the calling thread will wait for. This can either be the id of a regular event group or the id of the calling thread converted to an event group id (`avixThread_GetIdCurrent().asEventId`) to identify the thread event group. The calling thread may not pass the id of another thread converted to an event group id as this will result in an unstable system.
desiredEventFlags	16-bit mask specifying the flags to wait for being set. When specifying AVIX_EF_NONE, this function will never block the calling thread. Using this parameter value, the status of the event flags are returned.
combine	Parameter specifying whether to wait for all specified flags to be set (`AVIX_EVENT_GROUP_ALL`) or for any of the desired flags to be set (`AVIX_EVENT_GROUP_ANY`)
preClearMask	16-bit mask specifying which flags to clear before the check is done for the desired flags.
postClearMask	16-bit mask specifying which flags to clear after the wait has ended.
Return value	16-bit value representing the event group flags the moment the wait was honored. *These are the flags before optionally flags are cleared as specified in parameter postClearMask.*

▶ *When using this function with a timeout and it returns since the timeout expires, the flags specified in parameter postClearMask are not cleared. Flags defined in parameter preClearmask are however cleared.*

▶ *When waiting for a thread event group, the event group id passed to the wait function may only be the id of the calling thread converted to an event group id (`avixThread_GetIdCurrent().asEventId`). Passing the id of another thread will be reported as an error.*

Event group related definitions

The following event group related definitions are provided:

>> definition overview

`AVIX_EF(f)`
Return an event flag value where the flag corresponding to the specified parameter f is set (1) and all other flags are cleared (0). The number specified by parameter f may be in the range 0 up to and including 15. Larger values do not result in an error but the result of the macro equals AVIX_EF_NONE.

`AVIX_EF_ALL`
Return an event flag value with all flags set (`0xFFFF`).

`AVIX_EF_IN(v,f)`
Compare the flags in v and f. If every set flag in v corresponds with a set flag in f the macro returns a value unequal to zero ('true') else it returns zero ('false').

`AVIX_EF_IN_MASKED(v,f,m)`
Compare the flags in v and f that correspond with a set flag (1) in m. If every set flag in v corresponds with a set flag in f where the corresponding flag in m is set, the macro returns a value unequal to zero ('true') else it returns zero ('false').

`AVIX_EF_INVERT_ALL(f)`
Return an event flag value where a flag in parameter f that is cleared (0) becomes set (1) and vice versa.

`AVIX_EF_INVERT_MASKED(f,m)`
Return an event flag value where those flags in parameter f that correspond with a set flag (1) in parameter m are inverted, e.g. a set flag (1) becomes a cleared flag (0) and vice versa. Flags in f that correspond with a cleared (0) flag in parameter m are left unchanged.

`AVIX_EF_IS(v,f)`
Compare all flags in v and f. If the result is equal, the macro returns a value unequal to zero ('true'). If the result is not equal, the macro returns a value equal to zero ('false'). There is no difference between using this macro and using a direct 'C' language level equality operator (==).

`AVIX_EF_IS_MASKED(v,f,m)`
Compare the flags in v and f that correspond with a set flag (1) in m. If the result is equal, the macro returns an integer value unequal zero ('true'). If the result is not equal, the macro returns an integer value equal to zero ('false').

AVIX_EF_NONE
Return an event flag value with all flags cleared (0x0000).

AVIX_EF_RANGE(a,b)
Return an event flag value where the group of consecutive flags from the lower value of parameters a and b, up to and including the flag corresponding to the higher value of parameters a and b are set one (1) and all other flags are cleared (0).

AVIX_EVENT_GROUP_ALL
Flag to pass to function avixEventGroup_Wait to wait for all specified flags to be set.

AVIX_EVENT_GROUP_ANY
Flag to pass to function avixEventGroup_Wait to wait for any specified flag to be set.

AVIX_EVENT_GROUP_CLEAR
Flag to pass to function avixEventGroup_Change to clear the identified flags.

AVIX_EVENT_GROUP_SET
Flag to pass to function avixEventGroup_Change to set the identified flags.

AVIX_EVENT_GROUP_TOGGLE
Flag to pass to function avixEventGroup_Change to toggle the identified flags.

7.6.5 Timer Support

Header file to include: AVIX.h
Service specific header file: AVIXTimer.h

The following functions and definitions are present:

TIMER FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixTimer_ConnectEventGroup	164	✓	✓	✓	-
avixTimer_Create	165	✓	✓	✓	-
avixTimer_DisconnectEventGroup	166	✓	✓	✓	-
avixTimer_Get	167	-	✓ 04:02	✓	-
avixTimer_GetRemainingTicks	168	✓	✓	✓	-
avixTimer_Resume	169	✓	✓	✓	-
avixTimer_ResumeFromISR	170	-	-	-	✓
avixTimer_Set	171	✓	✓	✓	-
avixTimer_SetPeriod	172	✓	✓	✓	-
avixTimer_Start	173	✓	✓	✓	-
avixTimer_StartFromISR	174	-	-	-	✓
avixTimer_Stop	175	-	✓	✓	-
avixTimer_StopFromISR	176	-	-	-	✓
avixTimer_Wait	177	-	✓	-	-

TIMER DEFINITIONS	
Definition	**Description**
AVIX_DELAY_MS	Generate nr. of ticks for specified milliseconds
AVIX_DELAY_MS_US	Generate nr. of ticks for specified milli- microseconds
AVIX_DELAY_S	Generate nr. of ticks for specified seconds
AVIX_DELAY_S_MS	Generate nr. of ticks for specified seconds- milliseconds
AVIX_DELAY_S_MS_US	Generate nr. of ticks for specified seconds- milli- microseconds
AVIX_DELAY_US	Generate nr. of ticks for specified microseconds
AVIX_SYS_CLOCK_ACTUAL_PERIOD	Actual system clock period based on timer resolution
AVIX_TIMER_CYCLIC	Definition to set a timer as cyclic
AVIX_TIMER_MAX_NR_TICKS	Maximum tick value for timer
AVIX_TIMER_SINGLE_SHOT	Definition to set a timer as single shot

avixTimer_ConnectEventGroup

```
void avixTimer_ConnectEventGroup
(
    tavixTimerId               timerId,
    tavixEventId               event,
    tavixEventGroupOperation   operation,
    tavixEventFlags            eventFlags
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

This function connects an event group to a timer. When the timer is started and its period has elapsed, the action specified in parameter operation is performed on the event group with the flags specified in parameter eventFlags. For a single shot timer this happens once and for a cyclic timer this happens repeatedly.

Both a global event group or a thread local event group may be connected to a timer. For a thread event group, the id of the thread must be converted to an event group id using attribute `.asEventId` on the thread id.

▶ *A single shot timer 'remembers' whether it has expired. When connecting an event group to an expired single shot timer, the specified operation is executed on the event group the moment the event group is connected to the timer.*

Only one event group can be connected to a timer being either a regular event group or a thread event group. When connection a thread event group to a timer while an event group is already connected, the existing connection is removed. The timer only controls event flags of the last event group that was connected.

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to connect an event group to.
event	Identification of the event group that will be connected to the timer. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId`.
operation	Operation to execute on the specified event group when the timer expires. Possible values for this parameter are; `AVIX_EVENT_GROUP_CLEAR`, `AVIX_EVENT_GROUP_SET` or `AVIX_EVENT_GROUP_TOGGLE`
eventFlags	16-bit value specifying the flags that will undergo the specified operation when the timer expires.
Return value	<none>

avixTimer_Create

```
tavixTimerId avixTimer_Create
(
    tavixkernelObjectName pName
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Create a timer kernel object. If a valid name is specified and other threads are waiting for the timer kernel object to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

Parameters and return value

Parameter	Description
pName	Human readable name for the timer or NULL if the timer does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
Return value	Id representing the newly created timer.

avixTimer_DisconnectEventGroup

```
void avixTimer_DisonnectEventGroup
(
    tavixTimerId   timerId,
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Disconnect an event group from a timer. After calling this function, the event group that was connected will no longer be influenced by an expiring timer. This function can be used regardless whether the event group connected is a regular event group or a thread event group. When called while no event group is currently connected, the function has no effect.

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to disconnect an event group from.
Return value	<none>

avixTimer_Get

```
tavixTimerId avixTimer_Get
(
    tavixkernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Obtain the id of a timer kernel object with a specific name. If the timer with the specified name exists the function returns immediately with a timer id guaranteed to be valid. If the timer with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated timer is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the timer does not exist and this function is called from a DIH, the function returns immediately with an invalid timer id.

Parameters and return value

Parameter	Description
pName	Human readable name for the timer whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the timer with the specified name (pName).

avixTimer_GetRemainingTicks

```
tavixTimerTick avixTimer_GetRemainingTicks
(
    tavixTimerId timerId
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

For an active timer, this function returns the remaining number of ticks before the timer will expire.

Note the value returned by this function is a snapshot. When using this value, it is very likely the actual timer value is different since timers are updated by a central timer interrupt handler.

When the timer is not active, the returned value has the following meaning:

A value of zero means the timer either has expired or is (re)initialized by using function `avixTimer_Set`.

A negative value means a timer is stopped before being expired. The absolute value of the return value is the number of ticks the timer had left to go before expiring when it was stopped.

To convert the number of ticks returned by this function to an actual time, multiply the number of ticks with macro `AVIX_SYS_CLOCK_ACTUAL_PERIOD`. This macro represents the number of microseconds of a single tick.

Parameters and return value

Parameter	Description
timerId	Identification of the timer object for which to retrieve the remaining number of ticks before the timer expires.
Return value	When the timer is running, the remaining number of ticks before the timer will expire, this is always a value of one (1) or greater. When the timer is not running, a return value of zero (0) identifies the timer has expired or was (rre)initialized. When the timer is not running, a negative value identifies the timer was stopped and the absolute value of the return value identifies the number of ticks that were left before the timer would expire.

avixTimer_Resume

```
int avixTimer_Resume
(
    tavixTimerId timerId
);
```

avixMain[24]	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

This function resumes a timer.

When resuming a timer that previously has been stopped, the timer starts counting with the tick count the timer had when it was stopped.

When resuming a timer that is initialized using `avixTimer_Set`, the timer starts counting with the tick count used with the last `avixTimer_Set`.

When resuming a timer that previously expired, the timer starts counting with the tick count used with the last `avixTimer_Set`.

When resuming a timer that is running, the function does nothing!

The tick count for the initial period is not incremented by one (1) as is the case with `avixTimer_Start`. When using this function to start a timer that previous has been initialized using `avixTimer_Set`, the tick count for the first period equals the tick count used with `avixTimer_Set`. The tick count of the first period is not compensated for the fact this function may be called just before a system tick expires.

► *When resuming a previously expired single shot timer, this timer loses its expired state.*

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to resume.
Return value	A value equal zero (‘false’), identifies this function is called while the timer was in the counting state meaning the function did nothing. Calling this function in all other states returns a value not equal to zero(‘true’) (except state Uninitialized, which leads to a system error).

[24] When resuming a timer from avixMain, the timer starts running as soon as AVIX takes control, which is at the point avixMain returns.

avixTimer_ResumeFromISR

```
int avixTimer_ResumeFromISR
(
    tavixTimerId timerId
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

This function resumes a timer.

When resuming a timer that previously has been stopped, the timer starts counting with the tick count the timer had when it was stopped.

When resuming a timer that is initialized using `avixTimer_Set`, the timer starts counting with the tick count used with the last `avixTimer_Set`.

When resuming a timer that previously expired, the timer starts counting with the tick count used with the last `avixTimer_Set`.

When resuming a timer that is running, the function does nothing!

Although this function starts a timer, the tick count for the initial period is not incremented by one (1) as is the case with `avixTimer_Start`. So when using this function to start a timer that previous has been initialized using `avixTimer_Set`, the tick count for the first period equals the tick count used with `avixTimer_Set`. So the tick count of the first period is not compensated for the fact this function may be called just before a system tick expires.

▶ *When resuming a previously expired single shot timer, this timer loses its expired state.*

▶ *Note this function has no return value in contradiction to its peer function `avixTimer_Resume`.*

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to resume.
Return value	<none>

avixTimer_Set

```
void avixTimer_Set
(
    tavixTimerId   timerId,
    tavixTimerTick ticks,
    tavixTimerType type
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Set the period and type of a timer object. If the timer is counting, using this function, the timer is stopped.

When setting a previously expired single shot timer, the state of this timer is set not-expired.

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to set.
ticks	Period of the timer in kernel ticks. In order to specify the value of this parameter in a time based on seconds, milliseconds or microseconds, use can be made of the timer utility macros specified in section 'Timer related definitions' on page 178.
type	The type of timer. This parameter can have one of two possible values: • `AVIX_TIMER_SINGLE_SHOT`: This value specifies that upon expiration the timer is not automatically re-armed. • `AVIX_TIMER_CYCLIC`: This value specifies that upon expiration the timer is automatically re-armed and continues to count.
Return value	<none>

avixTimer_SetPeriod

```
void avixTimer_SetPeriod
(
    tavixTimerId   timerId,
    tavixTimerTick ticks
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Set the period of a timer. This function can be used regardless whether the timer is stopped or counting. If the timer is counting, this function will not stop the timer.

The newly set period will be used immediately after the next expiration of the timer. When used while the timer is counting, the current period will not be changed.

This function is especially useful to change the period of a cyclic timer that is counting. Such a timer will finish its current period with the current number of ticks. On expiration, a cyclic timer will automatically restart another period, the duration of which will be the number of ticks specified with this function. This will result in a smooth transition to the newly specified period.

Changing the period of a counting cyclic timer using function `avixTimer_Set` will result in the timer being stopped and after setting the new period the timer must be restarted using `avixTimer_Start`. It is however unpredictable where in the current period this will happen resulting in a transition period with an unpredictable duration.

Parameters and return value

Parameter	**Description**
timerId	Identification of the timer object to set.
ticks	Period of the timer in kernel ticks. In order to specify the value of this parameter in a time based on seconds, milliseconds or microseconds, use can be made of the timer utility macros specified in section 'Timer related definitions' on page 178.
Return value	<none>

avixTimer_Start

```
int avixTimer_Start
(
    tavixTimerId timerId
);
```

avixMain[25]	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

This function activates a timer object to start counting. The properties used are those that have been set with the last call to `avixTimer_Set` which must be called at least once before `avixTimer_Start` is called. Regardless the timer object state this function is called in, the timer always starts using the `avixTimer_Set` properties.

To guarantee the time that expires for the first expiration is at least equal to a system timer period, when starting a timer, one(1) is added to the programmed tick count. This compensates for the situation where the timer is started just before the system timer expires.

▶ *When starting a previously expired single shot timer, this timer loses its expired state.*

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to start.
Return value	A value unequal zero('true') identifies that this function is called while the timer was in the Counting state. Calling this function in all other states returns a value equal to zero('false') (except state Uninitialized, which leads to a system error).

[25] When starting a timer from avixMain, the timer starts running as soon as AVIX takes control, which is at the point avixMain returns.

avixTimer_StartFromISR

<pre>void avixTimer_StartFromISR
(
    tavixTimerId timerId
);</pre>
	avixMain –
	Thread –
	DIH –
	ISR ✓

>> function overview

Description

This function activates a timer object to start counting. The properties used are those that have been set with the last call to `avixTimer_Set` which must be called at least once before `avixTimer_Start` is called. Regardless the timer object state this function is called in, the timer always starts using the `avixTimer_Set` properties.

To guarantee the time that expires for the first expiration is at least equal to a system timer period, when starting a timer, one is added to the programmed tick count. This compensates for the situation where the timer is started just before the system timer expires.

▶ *When starting a previously expired single shot timer, this timer loses its expired state.*

▶ *Note this function has no return value in contradiction to its peer function `avixTimer_Start`.*

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to start.
Return value	<none>

avixTimer_Stop

```
void avixTimer_Stop
(
    tavixTimerId timerId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

This function stops a running timer.

A thread waiting for the timer to expire will remain in the blocked state.

When an event group is connected to the timer, stopping the timer will not execute the action specified for that event group.

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to stop.
Return value	<none>

avixTimer_StopFromISR

```
void avixTimer_StopFromISR
(
    tavixTimerId timerId
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

This function stops a running timer.

A thread waiting for the timer to expire will remain in the blocked state.

When an event group is connected to the timer, stopping the timer will not execute the action specified for that event group.

Parameters and return value

Parameter	**Description**
timerId	Identification of the timer object to stop.
Return value	<none>

avixTimer_Wait

```void avixTimer_Wait (tavixTimerId id);```	**avixMain** – **Thread** ✓ **DIH** – **ISR** –

>> function overview

Description

Through this function thread(s) can wait for a timer object to expire. Threads calling this function enter the 'Blocked' state until the timer expires. When the timer expires, all threads waiting for that timer enter the 'Ready' state. Which of all threads having the 'Ready' state actually is allowed to execute is determined by the scheduler mechanism.

If a single shot timer expires, its internal state is set accordingly. Making an `avixTimer_Wait` call to a timer in the expired state effectively does not Block the calling thread and the call returns immediately. The expired state is not reset by this call and multiple calls to `avixTimer_Wait` under this circumstance have the described behavior. The expired state of a timer is reset with either a call to `avixTimer_Set` or `avixTimer_Start`.

► *Forgetting to activate a timer a thread is waiting for, implies that the waiting thread will never be activated again, unless the timer is started again. Care must be taken that this situation does not happen since it is impossible for AVIX to assist in detecting this.*

► *Stopping a timer through a call to* `avixTimer_Stop` *implies that threads waiting for that timer remain in the 'Blocked' state if the stopped timer is not activated again through* `avixTimer_Start`*.*

Parameters and return value

Parameter	Description
timerId	Identification of the timer object to wait for.
Return value	<none>

Timer related definitions

The following timer related definitions are provided: >> definition overview

It is strongly advised to use the AVIX_DELAY_... macros as much as possible with constant parameters. Doing so the compiler will calculate the number of ticks and these macros are very efficient only taking ~6 cycles. When using variables as arguments, the calculation time can be significantly longer!

`AVIX_DELAY_MS(ms)`
The number of milliseconds passed as a parameter to this macro is converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter ms is 2,100,000 corresponding to 35 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

`AVIX_DELAY_MS_US(ms, us)`
The combined time as specified by parameters ms and µs are converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter ms is 2,100,000 corresponding to 35 minutes. The maximum value for parameter us is 2,100,000,000 corresponding to 35 minutes. The maximum total delay that can be reached with this macro is the sum equaling 70 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

`AVIX_DELAY_S(s)`
The number of seconds passed as a parameter to this macro is converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter s is 2,100 corresponding to 35 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

`AVIX_DELAY_S_MS(s, ms)`
The combined time as specified by parameters s and ms are converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter s is 2,100 corresponding to 35 minutes. The maximum value for parameter ms is 2,100,000 corresponding to 35 minutes. The maximum total delay that can be reached with this macro is the sum equaling 70 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

AVIX_DELAY_S_MS_US(s, ms, us)
The combined time as specified by parameters s, ms and us are converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter s is 2,100 corresponding to 35 minutes. The maximum value for parameter ms is 2,100,000 corresponding to 35 minutes. The maximum value for parameter us is 2,100,000,000 corresponding to 35 minutes. The maximum total delay that can be reached with this macro is the sum equaling 105 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

AVIX_DELAY_US(us)
The number of microseconds passed as a parameter to this macro is converted to a number of system timer ticks. The number of ticks is the ceiling of the specified time divided by the system timer period. This macro can be used anywhere where a parameter of type `tavixTimerTick` is expected. The maximum value for parameter us is 2,100,000,000 corresponding to 35 minutes. The AVIX timer mechanism is capable of reaching far longer times when directly specifying timer ticks.

AVIX_SYS_CLOCK_ACTUAL_PERIOD
The actual period of the system timer in microseconds. Based on the resolution of the clock source used for the system timer the actual system timer period can differ from the configured period (`avix_SYS_CLOCK_TICKus`). This macro specifies the actual system timer period in microseconds. This macro is used by the `AVIX_DELAY_...` macros. This macro can be used by the application code for time calculations depending on the system timer period.

AVIX_TIMER_CYCLIC
Flag to pass to function `avixTimer_Set` to initialize the timer as a cyclic running timer.

AVIX_TIMER_MAX_NR_TICKS
Definition representing the maximum number of ticks a timer can be set to.

AVIX_TIMER_SINGLE_SHOT
Flag to pass to function `avixTimer_Set` to initialize the timer as a sing shot timer.

7.6.6 Message Support

Header file to include: AVIX.h
Service specific header file: AVIXMsg.h

The following functions are offered:

MESSAGE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixMsg_Allocate	181	–	✓ 04:02	✓	–
avixMsg_AllocateFromISR	182	–	–	–	✓
avixMsg_Free	183	–	✓	✓	–
avixMsg_GetChar	184	–	✓	✓	–
avixMsg_GetIndirect	185	–	✓	✓	–
avixMsg_GetInt	186	–	✓	✓	–
avixMsg_GetKernelObjectId	187	–	✓	✓	–
avixMsg_GetLong	188	–	✓	✓	–
avixMsg_GetPtr	189	–	✓	✓	–
avixMsg_GetSender	190	–	✓	✓	–
avixMsg_GetShort	191	–	✓	✓	–
avixMsg_GetType	192	–	✓	✓	–
avixMsg_PutChar<FromISR>[26]	193	–	✓	✓	✓
avixMsg_PutIndirect<FromISR>[19]	193	–	✓	✓	✓
avixMsg_PutInt<FromISR>[19]	194	–	✓	✓	✓
avixMsg_PutKernelObjectId<FromISR>[19]	195	–	✓	✓	✓
avixMsg_PutLong<FromISR>[19]	196	–	✓	✓	✓
avixMsg_PutPtr<FromISR>[19]	197	–	✓	✓	✓
avixMsg_PutShort<FromISR>[19]	198	–	✓	✓	✓
avixMsg_Reuse	199	–	✓	✓	–
avixMsgQThread_ConnectEventGroup	201	✓	✓	✓	–
avixMsgQThread_DisconnectEventGroup	202	–	✓	✓	–
avixMsgQThread_Flush	203	–	✓	–	–
avixMsgQThread_Receive	204	–	✓ 04:02	–	–
avixMsgQThread_Reply	205	–	✓	–	–
avixMsgQThread_Send	206	–	✓	✓	–
avixMsgQThread_SendFromISR	207	–	–	–	✓

[26] All avixMsg_Put… functions are allowed to be called from threads, DIH's and ISR's. The …<FromISR> version of those functions is a macro which evaluates to the regular function. The reason the …<FromISR> version is offered is to adhere to the convention that ISR's may only use …<FromISR> functions which aids in checking the application for correctness.

avixMsg_Allocate

```
tavixMsgId avixMsg_Allocate
(
    tavixMsgType msgType
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	?
ISR	–	

>> function overview

Description

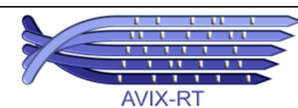
Allocate a message from the message pool. When a message is available in the pool, the function returns immediately with the message id guaranteed to be valid. When no message is available in the pool, behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the ‘Blocked’ state until either a message becomes available since it is freed by another thread or the optionally timeout expires. In case of a timeout the returned id is invalid and may not be used. When the message pool is empty and this function is called from a DIH, the function returns immediately with an invalid message id.

After obtaining a valid message, the message internal data pointer is set to refer the beginning of the message data block. Subsequent calls to the `avixMsg_Put…` functions will update this internal pointer according the size of the data written.

► *Allocating a message with a zero sized message pool (configuration parameter `avix_MSG_POOL_NR_MESSAGES` equals zero) asserts.*

Parameters and return value

Parameter	Description
msgType	User specified numeric value that will be placed in the message to be able to differentiate it from other messages.
Return value	In case of success, id representing the allocated message.



avixMsg_AllocateFromISR

`tavixMsgId avixMsg_AllocateFromISR`
`(`
`    tavixMsgType msgType`
`);`
	avixMain –
	Thread –
	DIH –
	ISR ✓

>> function overview

Description

Allocate a message from the message pool. This function is only allowed to be called from an ISR. Under all circumstances the function will return immediately. When a message is available in the pool, the function returns with the message id guaranteed to be valid. When no message is available in the pool, the functions returns with an invalid message id.

After obtaining a valid message, the message internal data pointer is set to refer the beginning of the message data block. Subsequent calls to the `avixMsg_Put…` functions will update this internal pointer according the size of the data written.

► *Allocating a message with a zero sized message pool (configuration parameter `avix_MSG_POOL_NR_MESSAGES` equals zero) asserts.*

Parameters and return value

Parameter	Description
msgType	User specified numeric value that will be placed in the message to be able to differentiate it from other messages.
Return value	In case of success, id representing the allocated message.

avixMsg_Free

```
void avixMsg_Free
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH[27]	✓
ISR	–

>> function overview

Description

When a thread has ownership of a message and does no longer need it, the message can be placed in the AVIX message pool through this function. After doing so, the message is available for other threads to be allocated. After freeing a message the thread is no longer allowed to access the message. Attempts to do so will result in an error.

Parameters and return value

Parameter	Description
msgId	Id of a message
Return value	<none>

[27] Although a DIH can use this function, it is unlikely to happen since this function typically is used with a message that has been received. Since a DIH can not receive messages a DIH can only use this function on messages obtained through `avixMsg_Allocate`.

avixMsg_GetChar

```
unsigned char avixMsg_GetChar
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read a char data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
Return value	Value read from the message

avixMsg_GetIndirect

```
void avixMsg_GetIndirect
(
    tavixMsgId msg,
    void*      msgContent,
    int        msgContentSize
);
```

avixMain	-
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Through this function a thread can read a user specified data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value as specified in parameter msgContentSize. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
msgContent	Address of variable where data retrieved from message will be copied to
msgContentSize	Number of bytes that will be copied.
Return value	<none>

avixMsg_GetInt

```
unsigned int avixMsg_GetInt
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read an integer data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
Return value	Value read from the message

avixMsg_GetKernelObjectId

```
void avixMsg_GetKernelObjectId
(
    tavixMsgId           msgId,
    tavixKernelObjectIdp pKernelId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read a kernel object id from a message. The kernel object is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the kernel object id. This function is only allowed when the thread owns the message.

In contradiction with other AVIX functions, the kernel object id read from the message is not returned as a function result. Instead, the address of the kernel object id variable is passed to this function.

For passing the address of the kernel object id variable through parameter pKernelId, use is made of a C99 feature called designated initializers. Type tavixKernelObjectIdp is a union containing a pointer for every possible type of kernel object. These fields are named after the kernel object type. Example usage for all available kernel object types is shown in the code fragment below where the designated initializers are shown in brown. The names behind the dot (.) are the field names present in type tavixKernelObjectId. Assigned to a field can be the address of a kernel object id variable of the corresponding type. Using this mechanism offers the flexibility required to pass every available type of kernel object to a message and still maintain the highest possible level of type safety.

```
 1 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.thread    = &threadId    } );
 2 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.mutex     = &mutexId     } );
 3 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.semaphore = &semaphoreId } );
 4 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.event     = &eventId     } );
 5 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.timer     = &timerId     } );
 6 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.pipe      = &pipeId      } );
 7 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.memPool   = &memPoolId   } );
 8 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.exch      = &exchId      } );
 9 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.msg       = &msgId       } );
10 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.memBlock  = &memBlockId  } );
11 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectIdp){.exchConn  = &exchConnId  } );
```

Although being largely type safe (only addresses of kernel object id variables can be passed for parameter pKerneld), this function must be used with care. The type of kernel object id passed must comply with the type present in the message. AVIX does not check whether the type of the parameter and the type present in the message comply.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
pKernelId	Address of kernel object id variable where the result of the function will be placed.
Return value	<none>

avixMsg_GetLong

```
unsigned long avixMsg_GetLong
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read a long data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
Return value	Value read from the message

avixMsg_GetPtr

```
void* avixMsg_GetPtr
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read a pointer data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
Return value	Value read from the message

avixMsg_GetSender

```
tavixThreadId avixMsg_GetSender
(
    tavixMsgId msgId
);
```

avixMain	-
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Function to retrieve the id of the sending thread. This function only returns a valid thread id when used on a message that actually has been sent by a thread.

The sender id contained in the message remains unchanged until the message is send again, in which case the id of the new sender is saved in the message.

When receiving a message from an ISR through `avixMsgQThread_SendFromISR` or from a DIH, the value returned by this function is `AVIX_OBJECT_ID_NULL` since these active entities do not have a thread id.

When using this function on a newly allocated message, also a value of `AVIX_OBJECT_ID_NULL` is returned.

Parameters and return value

Parameter	Description
msgId	Id of the message to retrieve the sender thread id from.
Return value	Id of the thread from which the message was received or `AVIX_OBJECT_ID_NULL` in case of the described exceptions.

avixMsg_GetShort

```
unsigned short avixMsg_GetShort
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function a thread can read a short data value from a message. The data value is read from the message through the message internal data pointer. This internal data pointer is updated according the size of the data value. This function is only allowed when the thread owns the message.

Parameters and return value

Parameter	Description
msgId	Id of the message to get the data item from
Return value	Value read from the message

avixMsg_GetType

```
tavixMsgType avixMsg_GetType
(
    tavixMsgId msgId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Through this function the type of the message identified with msgId can be obtained. This is typically used by a receiving thread to determine the type of message that has been received in order to determine the structure of the data contained in the message. This structure is a mutual agreement between the sending and the receiving entity. If messages with multiple different internal structures can be received by a thread, the value of the message type typically is used to discriminate between these different types and access the structure of the message data section accordingly.

Parameters and return value

Parameter	Description
msgId	Id of the message to retrieve the type from.
Return value	Type of message

avixMsg_PutChar<FromISR>

```
void avixMsg_PutChar
(
    tavixMsgId    msgId,
    unsigned char msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutCharFromISR
(
    tavixMsgId    msgId,
    unsigned char msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a char type data value is placed in a message. The data value is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value.

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

► *The …`FromISR` version of this function is intended to be used exclusively by an ISR.*

avixMsg_PutIndirect<FromISR>

```
void avixMsg_PutIndirect
(
    tavixMsgId  msg,
    const void* msgContent,
    int         msgContentSize
);
```

Context	
avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutIndirectFromISR
(
    tavixMsgId  msg,
    const void* msgContent,
    int         msgContentSize
);
```

Context	
avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a user specified data value is put in a message. The data value is put in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value as specified in parameter msgContentSize.

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Address of value to put in the message body
msgContentSize	Size in bytes to copy.
Return value	<none>

The …`FromISR` version of this function is intended to be used exclusively by an ISR.

avixMsg_PutInt<FromISR>

```
void avixMsg_PutInt
(
    tavixMsgId   msgId,
    unsigned int msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutIntFromISR
(
    tavixMsgId   msgId,
    unsigned int msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function an integer type data value is placed in a message. The data value is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value.

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

▶ *The …`FromISR` version of this function is intended to be used exclusively by an ISR.*

avixMsg_PutKernelObjectId<FromISR>

```
void avixMsg_PutKernelObjectId
(
    tavixMsgId          msgId,
    tavixKernelObjectId msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutKernelObjectIdFromISR
(
    tavixMsgId          msgId,
    tavixKernelObjectId msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a kernel object id is placed in a message. Use of AVIX kernel object id's is compile time type safe. A variable of type `tavixThreadId` cannot be assigned to a variable of type `tavixMutexId` and doing so will result in a compile time error. In order to allow any possible kernel object id to be placed in a message, an additional type is specified, `tavixKernelObjectId`. This type is specified such that all AVIX kernel object id's can be passed for the msgContent parameter. The value of the msgContent parameter is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the id. This function is only allowed to be used when the thread owns the message.

For passing the kernel object id through parameter msgContent, use is made of a C99 feature called designated initializers. Type tavixKernelObjectId is a union containing a field for every possible type of kernel object. These fields are named after the kernel object type. Example usage for all available kernel object types is shown in the code fragment below where the designated initializers are shown in brown. The names behind the dot (.) are the field names present in type tavixKernelObjectId. Assigned to a field can be a kernel object id of the corresponding type. Using this mechanism offers the flexibility required to pass every available type of kernel object to a message and still maintain the highest possible level of type safety.

```
12 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.thread    = threadId    });
13 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.mutex     = mutexId     });
14 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.semaphore = semaphoreId });
15 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.event     = eventId     });
16 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.timer     = timerId     });
17 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.pipe      = pipeId      });
18 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.memPool   = memPoolId   });
19 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.exch      = exchId      });
20 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.msg       = msgId       });
21 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.memBlock  = memBlockId  });
22 avixMsg_PutKernelObjectId(msg, (tavixKernelObjectId){.exchConn  = exchConnId  });
```

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

▶ *The …`FromISR` version of this function is intended to be used exclusively by an ISR.*

avixMsg_PutLong<FromISR>

```
void avixMsg_PutLong
(
    tavixMsgId    msgId,
    unsigned long msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutLongFromISR
(
    tavixMsgId    msgId,
    unsigned long msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a long type data value is placed in a message. The data value is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value.

Parameters and return value

Parameter	**Description**
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

▶ *The …`FromISR` version of this function is intended to be used exclusively by an ISR.*

avixMsg_PutPtr<FromISR>

```
void avixMsg_PutPtr
(
    tavixMsgId  msgId,
    const void* msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutPtrFromISR
(
    tavixMsgId  msgId,
    const void* msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a pointer type data value is placed in a message. The data value is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value.

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

The ...`FromISR` version of this function is intended to be used exclusively by an ISR.

avixMsg_PutShort<FromISR>

```
void avixMsg_PutShort
(
    tavixMsgId      msgId,
    unsigned short msgContent
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

```
void avixMsg_PutShortFromISR
(
    tavixMsgId      msgId,
    unsigned short msgContent
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Through this function a short type data value is placed in a message. The data value is placed in the message through the message internal data pointer. This internal data pointer is updated according the size of the data value.

Parameters and return value

Parameter	Description
msgId	Id of the message to put the data item in
msgContent	Data item to put in the message
Return value	<none>

► *The …`FromISR` version of this function is intended to be used exclusively by an ISR.*

avixMsg_Reuse

```
void avixMsg_Reuse
(
    tavixMsgId   msgId,
    tavixMsgType msgType
);
```

avixMain	–
Thread	✓
DIH[28]	✓
ISR	–

>> function overview

Description

Reuse a message. Messages are allocated from a pool and after receiving a message the message must be returned to this pool by calling `avixMsg_Free`. Alternatively when a thread that received a message wants to send a message itself, the thread can reuse the message it received. Reusing a message is started by calling this function which sets the message type and resets the message internal data pointer so that the next `avixMsg_Put…` function will place its data at the beginning of the message body.

Parameters and return value

Parameter	Description
msgId	Id of a message
msgType	User specified numeric value that will be placed in the message to be able to differentiate it from other messages.
Return value	<none>

[28] Although a DIH can use this function, it is unlikely to happen since this function typically is used with a message that has been received. Since a DIH can not receive messages a DIH can only use this function on messages obtained through `avixMsg_Allocate`.

avixMsgQThread_ConnectEventGroup

```
void avixMsgQThread_ConnectEventGroup
(
    tavixThreadId   receivingThread,
    tavixEventId    eventGroup,
    tavixEventFlags eventFlags
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Connect an event group to the message queue of a thread. When calling this function the flags specified in parameter eventFlags will be set according the content of the message queue. When the message queue is empty the flags will be cleared. When the message queue is not empty, the flags will be set.

Both a global event group or a thread local event group may be connected to a thread message queue. For a thread event group, the id of the thread must be converted to an event group id using attribute `.asEventId` on the thread id.

After connecting an event group to a message queue, the specified event flags will be controlled by AVIX such that they constantly reflect the status of the message queue. When a message enters an empty message queue, the event flags are set. When the last message is retrieved from the message queue, leaving it empty, the event flags will be cleared.

Only one event group can be connected to a message queue being either a regular event group or a thread event group. Using this function when an event group is already connected to the message queue will result in the existing connection being removed. The message mechanism only controls event flags of the last event group that was connected to the message queue.

Parameters and return value

Parameter	Description
receivingThread	Id of the thread whose message queue the event group identified by parameter eventGroup will be related.
eventGroup	Identification of the event group that will be related to the message queue. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId`.
eventFlags	16-bit value specifying the flags that will be set when a message enters an empty message queue and will be reset when the last message is removed from the message queue.
Return value	<none>

avixMsgQThread_DisconnectEventGroup

```
void avixMsgQThread_DisonnectEventGroup
(
    tavixThreadId   receivingThread
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Disconnect an event group from a thread message queue. After calling this function, the event group that was connected will no longer be influenced by the filling of the message queue. This function can be used regardless whether the event group connected is a regular event group or a thread event group. When called while no event group is currently connected, the function has no effect.

Parameters and return value

Parameter	Description
receivingThread	Id of the thread whose message queue the event group is disconnected from.
Return value	<none>

avixMsgQThread_Flush

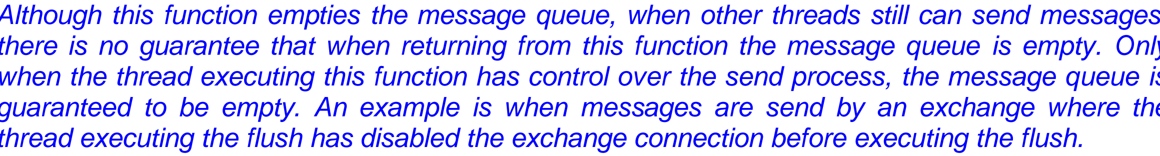
```
void avixMsgQThread_Flush(void);
```

avixMain	–
Thread	✓
DIH	–
ISR	–

>> function overview

Description

Flush all messages present in the thread message queue. The messages are discarded and the memory is returned to the message pool. When an event group is connected to the message queue, the specified flags are cleared.



Although this function empties the message queue, when other threads still can send messages, there is no guarantee that when returning from this function the message queue is empty. Only when the thread executing this function has control over the send process, the message queue is guaranteed to be empty. An example is when messages are send by an exchange where the thread executing the flush has disabled the exchange connection before executing the flush.

Parameters and return value

Parameter	Description
<none>	
Return value	<none>

avixMsgQThread_Receive

`tavixMsgId avixMsgQThread_Receive(void);`	**avixMain** – **Thread** ✓ 04:02 **DIH** – **ISR** –

>> function overview

Description

Receive a message from another thread, DIH or ISR. Once received, the receiving thread becomes the owner of the message. Calling this function while the message queue is empty, the thread will enter the ‘Blocked’ state. While blocked on the message queue and a message is send, the message will not enter the message queue but is handed over to the waiting thread immediately. This has a positive impact on performance while in this case the queue manipulation is skipped.

Parameters and return value

Parameter	Description
<none>	
Return value	Id of the received message.

avixMsgQThread_Reply

```
void avixMsgQThread_Reply
(
    tavixMsgId msg
);
```

avixMain	–
Thread	✓
DIH	–
ISR	–

>> function overview

Description

Send a message to the thread the message is received from.

When a thread sends a message (`avixMsgQThread_Send` or `avixMsgQThread_Reply`), the thread id of the sender is stored inside the message. This enables the receiver of the message to reply which is equal to sending the message to the thread the message was originally received from.

Messages can be send from threads, DIH's and ISR's. When receiving a message from a DIH or an ISR, this message cannot be used with avixMsgQThread_Reply since a DIH or ISR cannot receive messages.

Parameters and return value

Parameter	Description
msg	Id of message used for the reply.
Return value	<none>

avixMsgQThread_Send

```
void avixMsgQThread_Send
(
    tavixMsgId    msg,
    tavixThreadId destThread
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Send a message to a thread. The sender loses the message ownership. The receiving thread will obtain message ownership.

The message will be placed at the tail of the message queue of the destination thread. When at the moment of sending the destination threads message queue is empty and the destination thread is Blocked on this empty message queue (waiting for a message to arrive), the message is directly transferred to the thread without entering the message queue first.

Parameters and return value

Parameter	Description
msg	Id of message to send
destThread	Id of thread the message will be sent to
Return value	<none>

avixMsgQThread_SendFromISR

```
void avixMsgQThread_SendFromISR
(
    tavixMsgId    msg,
    tavixThreadId destThread
);
```

avixMain	-
Thread	-
DIH	-
ISR	✓

>> function overview

Description

Send a message to another thread from an ISR. The receiving thread will obtain message ownership.

The message will be placed at the tail of the message queue of the destination thread. When at the moment of sending the destination threads message queue is empty and the destination thread is Blocked on this empty message queue (waiting for a message to arrive), the message is directly transferred to the thread without entering the message queue first.

Parameters and return value

Parameter	Description
msg	Id of message to send
destThread	Id of thread the message will be sent to
Return value	<none>

7.6.7 Pipe Support

Header file to include: AVIX.h
Service specific header file: AVIXPipe.h

The following functions and definitions are offered:

PIPE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixPipe_AbortAsyncReq	209	–	✓	✓	–
avixPipe_Create	210	✓	✓	✓	–
avixPipe_FlushAndAbort	211	–	✓	✓	–
avixPipe_Get	212	–	✓ 04:02	✓ ?	–
avixPipe_GetStatusAsyncReq	213	–	✓	✓	–
avixPipe_Read	214	–	✓ 04:02	✓ ?	–
avixPipe_ReadAsync	216	–	✓	✓ ?	–
avixPipe_ReadFromISR	218	–	–	–	✓
avixPipe_SetHandlerTracePort	220	✓	✓	–	–
avixPipe_StopDeviceFromISR	221	–	–	–	✓
avixPipe_Write	222	–	✓ 04:02	✓ ?	–
avixPipe_WriteAsync	224	–	✓	✓ ?	–
avixPipe_WriteFromISR	224	–	–	–	✓

PIPE DEFINITIONS	
Definition	**Description**
PIPE_ASYNCREQ_ABORTED	Status when async request is aborted
PIPE_ASYNCREQ_FINISHED	Status when async request is finished
PIPE_ASYNCREQ_PENDING	Status when async request is pending
PIPEINFO_DEVICE_STOP_REQUESTED	Callback value for avixPipe_StopDeviceFromISR
PIPEINFO_PIPE_DATA_WRITTEN	Callback value when a thread writes data to a pipe buffer.
PIPEINFO_READ_FROM_EMPTY_PIPE	Callback value when a thread reads from an empty pipe buffer.

avixPipe_AbortAsyncReq

```
tavixPipeAsyncRegStatus avixPipe_AbortAsyncReq
(
    tavixPipeAsyncReqId* pReqId,
    unsigned int*        pNrBlocksProcessed
);
```

avixMain	-
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Abort a pending asynchronous write or read request and return the request status.

When executing an asynchronous write or read request, optionally the address of a request id variable may be passed. Doing so, relates the identified request id variable to that asynchronous request. From that moment on, the request may be aborted using `avixPipe_AbortAsyncReq` by passing it the address of the request id variable passed to the asynchronous request.

The asynchronous request is only *effectively aborted* when at the moment of executing `avixPipe_AbortAsyncReq` the status of the request is pending (`PIPE_ASYNCREQ_PENDING`). In this case only, the status is changed to `PIPE_ASYNCREQ_ABORTED`.

When at the moment of executing `avixPipe_AbortAsyncReq` the status of the request is `PIPE_ASYNCREQ_FINISHED` or `PIPE_ASYNCREQ_ABORTED` the asynchronous request is not *effectively aborted* and the status of the request is not changed. In this situation, the function actually does nothing.

An asynchronous request may optionally be passed an event group id and event flags. Only when by calling `avixPipe_AbortAsyncReq` the asynchronous request is *effectively aborted*, the specified event flags are set in the event group as a result of calling this function.

When at the moment of executing `avixPipe_AbortAsyncReq` the status of the request is `PIPE_ASYNCREQ_FINISHED` or `PIPE_ASYNCREQ_ABORTED`, no change is made to the current state of the specified event group.

Parameters and return value

Parameter	Description
pReqId	Address of a user defined request id variable. This variable must be the same variable the address of which was passed to an earlier call to `avixPipe_WriteAsync` or `avixPipe_ReadAsync`.
pNrBlocksProcessed	Address of variable that will receive the number of blocks read or written so far. When this information is not required, NULL may be passed for this parameter.
Return value	The status of the request identified by parameter pReqId. Possible values are: • `PIPE_ASYNCREQ_ABORTED`: The asynchronous request was effectively aborted or the status of the asynchronous request was `PIPE_ASYNCREQ_ABORTED` already when executing this function. • `PIPE_ASYNCREQ_FINISHED`: The asynchronous request was finished already when executing this function.

avixPipe_Create

```
tavixPipeId avixPipe_Create
(
    tavixKernelObjectName pName,
    unsigned int          numBlocks,
    unsigned int          blockSize,
    tavixPipeCallback     pCallback,
    const void*           pUserData
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Create a pipe kernel object. If a valid name is specified and other threads are waiting for the pipe object to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

When using a callback function, this callback must have the following signature:

```
void pipeCallback
    (   tavixPipeEvent event,
        int            nrBlocks,
        void*          pUserData );
```

Parameters and return value

Parameter	Description
pName	Human readable name for the pipe or `NULL` if the pipe does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in its first four characters.
numBlocks	The number of blocks the pipe buffer can maximally hold. The size in bytes of the pipe buffer will be the product of numBlocks and blockSize.
blockSize	Number of bytes in a single data block. The size in bytes of the pipe will be the product of numBlocks and blockSize. When writing or reading a pipe, the number of bytes specified by this parameter is guaranteed to be processed atomically.
pCallback	Address of a pipe call-back function or `NULL` if no call-back is used. Call-back functions are used to control ISR based device activity and thus are only applicable when the pipe is used between an ISR and one or more threads.
pUserData	Pointer to user defined information that will be passed to the pipe callback.
Return value	Id of the newly created pipe.

avixPipe_FlushAndAbort

```
void avixPipe_FlushAndAbort
(
    tavixPipeId pipeId
);
```

avixMain	-
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Flush all data from the pipe buffer and abort all pending requests, both synchronous and asynchronous. After executing this function the pipe buffer is empty and no requests are pending.

Synchronous pending requests are initiated by a thread executing either `avixThread_Read` or `avixThread_Write`. Since the request is pending, the related thread is in the 'Blocked' state. Executing this function will change the state of the thread to 'Ready'. The number of blocks processed until the moment this function is executed is passed to the thread as the return value of either `avixThread_Read` or `avixThread_Write`.

For pending asynchronous requests the request descriptor is removed from the pipe and the request status is set aborted (`PIPE_ASYNCREQ_ABORTED`). An asynchronous request may optionally be passed an event group id and event flags. When the asynchronous request is aborted by executing this function, the specified event flags are set in the event group to indicate the request is ready.

For the thread that initiated the asynchronous request to obtain information on the fact the request is aborted and the number of blocks processed until the moment this happens the asynchronous read or write function that initiated the request must be passed the address of a request id variable. To obtain the status of the request use can be made of `avixPipe_GetStatusAsyncReq`.

When during this function the pipe is read from an ISR using `avixPipe_ReadFromISR` or written from an ISR using `avixPipe_WriteFromISR`, the return value of these functions is negative to indicate to the ISR the pipe is currently being flushed.

Parameters and return value

Parameter	Description
pipeId	Id of the pipe to flush.
Return value	<none>

avixPipe_Get

```
tavixPipeId avixPipe_Get
(
    tavixKernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	?
ISR	–	

>> function overview

Description

Obtain the id of a pipe kernel object with a specific name. If the pipe with the specified name exists the function returns immediately with a pipe id guaranteed to be valid. If the pipe with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated pipe is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the pipe does not exist and this function is called from a DIH, the function returns immediately with an invalid pipe id.

Parameters and return value

Parameter	Description
pName	Human readable name for the pipe whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the pipe with the specified name (pName).

avixPipe_GetStatusAsyncReq

```
tavixPipeAsyncReqStatus avixPipe_GetStatusAsyncReq
(
    const tavixPipeAsyncReqId* pReqId,
    unsigned int*              pNrBlocksProcessed
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Request the status of an earlier executed asynchronous write or read request.

When executing an asynchronous write or read request, optionally the address of a request id variable may be passed. Doing so, relates the identified variable to that asynchronous request. From that moment on, the status of the request may be obtained using `avixPIpe_GetStatusAsyncReq` by passing it the address of the same request id variable passed to the asynchronous request.

Parameters and return value

Parameter	Description
pReqId	Address of a user defined request id variable. This variable must be the same variable the address of which was passed to an earlier call to function `avixPipe_WriteAsync` or `avixPipe_ReadAsync`.
pNrBlocksProcessed	Address of variable that will receive the number of blocks read or written at the moment the function is called. When the return status is: • `PIPE_ASYNCREQ_PENDING`: The returned value is always smaller than the number of blocks specified in the read or write request since the request is still pending for data to be processed. • `PIPE_ASYNCREQ_FINISHED`: The returned value is always equal to the number of blocks specified in the asynchronous request. • `PIPE_ASYNCREQ_ABORTED`: The returned value is always smaller than the number of blocks specified in the asynchronous request since the request was explicitly aborted before being finished or the request was executed from a DIH. When this information is not required, NULL may be passed for this parameter.
Return value	The status of the request identified by parameter pReqId. Possible values are: • `PIPE_ASYNCREQ_PENDING`: The asynchronous request is (still) pending. This return value is only possible when the asynchronous request was executed from a thread since when called from a DIH, the request will never enter the pending state. • `PIPE_ASYNCREQ_FINISHED`: The asynchronous request has finished and the requested number of blocks is processed. • `PIPE_ASYNCREQ_ABORTED`: The asynchronous request was executed from a thread and aborted using function `avixPipe_AbortAsyncReq` or `avixPipe_FlushAndAbort` or the asynchronous request was executed from a DIH where the desired number of blocks could not be processed.

avixPipe_Read

```
int avixPipe_Read
(
    tavixPipeId  pipeId,
    void*        pReadbuffer,
    unsigned int nrBlocksToRead
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Read data from a pipe. The following scenario's are applicable:

- *Read from a pipe while no requests are pending*: Data is transferred from the pipe buffer to the client provided buffer. When not all requested data is present in the pipe buffer, a request descriptor is allocated and the read request is set pending for the remaining amount of data to be read.

- *Read from a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty and no data is transferred to the client provided buffer. A request descriptor is allocated and the new read request is set pending.

- *Read from a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full. First data is transferred from the pipe buffer to the buffer of the read request. If this is not sufficient to finish the read request, data from the pending write requests is transferred to the buffer of the read request *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending write requests are updated according the amount of data processed. Pending write requests for which all data is transferred are set finished and their request descriptors are freed. When all pending write requests are finished and still data remains to be read, a request descriptor is allocated and the read request is set pending. When all requested data is read and still write requests are pending, as much as possible data is transferred from the pending write requests to the pipe buffer. Pending write requests for which all requested data is transferred to the pipe buffer are set finished and their request descriptors are freed.

For each of these scenario's:

- *When called from a thread*: When all specified data is read, the function returns immediately. When the requested amount of data is not available, the calling thread will enter the 'Blocked' state until sufficient data is written to the pipe, an optional time-out occurs or the buffer is flushed by calling `avixPipe_FlushAndAbort`.

- *When called from a DIH*: Whether or not all specified data is read, the function returns immediately, returning the number of blocks actually transferred. In case of insufficient data being available to fulfil the read request, in contradiction with calling this function from a thread, no request descriptor is allocated and the request will not be set pending. Effectively the part of above scenario's which is underlined is not executed when called from a DIH.

When a pipe callback is registered and the first scenario is applicable where the pipe buffer is found to be empty, the callback function is activated with flag `PIPEINFO_READ_FROM_EMPTY_PIPE`.

Parameters and return value

Parameter	Description
pipeId	Id of the pipe to read from.
pReadBuffer	Pointer to a memory location where the read data blocks will be transferred to. It is the responsibility of the user to make sure the referred location is large enough to hold the number of blocks that are read.
nrBlocksToRead	The desired number of blocks to read from the pipe.
Return value	Number of blocks actually read from the pipe. This is not necessarily the same number of blocks as specified with parameter `nrBlocksToRead` since a timeout may occur, the pipe is flushed or the read operation is performed by a DIH.

avixPipe_ReadAsync

```
void avixPipe_ReadAsync
(
    tavixPipeId          pipeId,
    void*                pReadBuffer,
    unsigned int         nrBlocksToRead,
    tavixEventId         readyEvent,
    tavixEventFlags      readyFlags,
    tavixPipeAsyncReqId* pReqId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Read data from a pipe. The following scenario's are applicable:

- *Read from a pipe while no requests are pending*: Data is transferred from the pipe buffer to the client provided buffer. When not all requested data is present in the pipe buffer, a request descriptor is allocated and the read request is set pending for the remaining amount of data to be read.

- *Read from a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty and no data is transferred to the client provided buffer. A request descriptor is allocated and the new read request is set pending.

- *Read from a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full. First data is transferred from the pipe buffer to the buffer of the read request. If this is not sufficient to finish the read request, data from the pending write requests is transferred to the buffer of the read request *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending write requests are updated according the amount of data processed. Pending write requests for which all data is transferred are set finished and their request descriptors are freed. When all pending write requests are finished and still data remains to be read, a request descriptor is allocated and the read request is set pending. When all requested data is read and still write requests are pending, as much as possible data is transferred from the pending write requests to the pipe buffer. Pending write requests for which all requested data is transferred to the pipe buffer are set finished and their request descriptors are freed.

For each of these scenario's, the function returns immediately:

- *When called from a thread*: When the requested amount of data is available, the request is said to have finished, the optionally specified event flag is set and/or the request id is updated.

  When the requested amount of data is not available a request descriptor is added to the pipe internal bookkeeping specifying the remaining amount of data to read. The request is said to be pending, the optionally specified event flag is cleared and/or the request id is updated. The status remains pending until either sufficient data is written, the request is aborted using `avixPipe_AbortAsyncReq` or the buffer is flushed by calling `avixPipe_FlushAndAbort`.

- *When called from a DIH*: When the requested amount of data is available, the request is said to have finished. When the requested amount of data is not available, the request is said to be aborted. The optionally specified event flag is set and/or the request id is updated. In contradiction with calling this function from a thread, no request descriptor is allocated and the request will not be set pending. Effectively the part of above scenario's which is underlined is not executed when called from a DIH.

When a pipe callback is registered and the first scenario is applicable where the pipe buffer is found to be empty, the callback function is activated with flag `PIPEINFO_READ_FROM_EMPTY_PIPE`.

Parameters and return value

Parameter	Description
pipeId	Id of the pipe to read from.
pReadBuffer	Pointer to a memory location where the read data blocks will be transferred to. It is the responsibility of the user to make sure the referred location is large enough to hold the number of blocks that are read.
nrBlocksToRead	The desired number of blocks to read from the pipe.
readyEvent	Id of event group in which flags specified by parameter `readyFlags` are set when the asynchronous read request is finished. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId` • A NULL event group id when no flags need to be set when the request is finished. Use `AVIX_OBJECT_ID_NULL(tavixEventId)`.
readyFlags	Event flags that are manipulated in the event group identified by parameter `readyEvent`. When the asynchronous read request is set pending, the specified flags are cleared. When the asynchronous read request is set finished, the specified flags are set. When for parameter `readyEvent` a NULL event group id is passed, the value of this parameter is don't care.
pReqId	Address of a user defined request id variable. This variable may be used to abort a pending request using `avixPipe_AbortAsyncReq` or obtain the status of a pending request using `avixPipe_GetStatusAsyncReq`. When this functionality is not required, NULL may be passed for this parameter.
Return value	<none>

avixPipe_ReadFromISR

```
int avixPipe_ReadFromISR
(
    tavixPipeId  pipeId,
    void*        pReadBuffer,
    unsigned int nrBlocksToRead
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Read data from a pipe for use by an ISR.

This function only reads from the pipe buffer and therefore will never read more data than present in the pipe buffer, regardless whether thread write requests are pending.

This is a non-blocking function since an ISR always runs to completion and cannot block. *When the pipe buffer does not contain the desired number of data blocks, it is up to the programmer to decide what to do in this situation.*

It is essential to check the return value of this function and take appropriate action.

This function creates empty space in the pipe buffer. When thread write requests are pending, after the ISR has read from the pipe buffer one of the following two scenario's is applicable:

- If the pending thread write request at the head of the list can write all required data to the pipe buffer, data is copied from the pending thread write request to the pipe buffer, the pending thread write request is set finished and the request descriptor is removed from the list. If present, the next pending thread write request will be at the head of the list and the above will be repeated or, if the thread write request needs to write more data than empty space present in the pipe buffer, the second scenario is applicable.

  For thread write requests that are set finished and belong to a synchronous transfer, the related thread is given the 'Ready' state. For thread write requests that are set finished and belong to a asynchronous transfer, the event flag(s) specified with this request is set and/or the request id is updated.

- If the pending thread write request at the head of the list cannot write all required data but the pipe buffer becomes empty for 50% or more, as much as possible data from the pending thread write request is copied to the pipe buffer. Doing so, AVIX assures the filling of the pipe is always sufficient for ISR handling.

This function does not activate the callback function which is optionally connected to the pipe on creation.

▶ *When reading a pipe from an ISR, the pipe may only be used for thread originating write requests. AVIX does not check for this and it is the users responsibility to guard this.*

Parameters and return value

Parameter	Description
pipeId	Id of the pipe. Since an ISR cannot obtain the id of its pipe through a Get function, this id must be made available by using a global variable.
pReadBuffer	Pointer to a memory location where the read data will be transferred to. It is the responsibility of the user to make sure the referred location is large enough to hold the number of data blocks read.
nrBlocksToRead	The desired number of blocks to read from the pipe.
Return value	Number of blocks actually read from the pipe. This is not necessarily the same number of blocks as specified with parameter nrBlocksToRead since the pipe buffer may not contain the desired number of blocks and an ISR cannot block to wait for the desired amount to be present. Negative value (< 0) when `avixPipe_ReadFromISR` is called while a thread is flushing the pipe using `avixPipe_FlushAndAbort`. In this case no data is read from the pipe buffer.

avixPipe_SetHandlerTracePort

```
void avixPipe_SetHandlerTracePort
(
    tavixPipeId          pipeId,
    tavixThreadTracePort port,
    unsigned int         bit
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Assign or de-assign one of the controller's digital I/O pins to a pipe. The specified I/O pin will be pulled high when for the pipe a handler is activated and pulled low when the handler is finished.

When using a pipe between threads and ISR's, by observing the state of the specified pin on a logic analyzer the CPU load of a pipe handler can be determined. This can be used to determine the optimal pipe buffer size and tune performance.

► *Depending on the hardware platform being used, Tracing works 'out of the box' or some configuration settings might be needed. Consult the hardware platform specific Port Guide for details.*

Parameters and return value

Parameter	Description
pipeId	Id of the pipe an I/O port is assigned to.
port	Use one of the following values: `AVIX_TRACE_PORT_A, AVIX_TRACE_PORT_B, … AVIX_TRACE_PORT_N, AVIX_TRACE_PORT_O or AVIX_TRACE_NONE.` Not all controllers have all these I/O ports. It is the user's responsibility to select an I/O port that is present in the used controller. Furthermore, make sure to select an I/O port/bit combination not used by the application.
bit	Bit number in the selected I/O port that is related to the pipe. Not all I/O ports have all bits available as digital I/O. It is the user's responsibility to select a bit available in the selected I/O port and make sure the I/O port/bit combination is not used by the application.
Return value	<none>

avixPipe_StopDeviceFromISR

```
void avixPipe_StopDeviceFromISR
(
    tavixPipeId pipeId,
    int         count
);
```

avixMain	-
Thread	-
DIH	-
ISR	✓

>> function overview

Description

Function to call from an ISR resulting in the pipe callback being called with flag `PIPEINFO_DEVICE_STOP_REQUESTED`. The only reason this function exists is to allow all device controlling functionality to be coded in the pipe callback function resulting in a cleaner and more comprehensible code structure.

The callback triggered by this function is only allowed to use AVIX functions that are also allowed to be called from an ISR.

When using AVIX functions from within a pipe callback function make sure to only use functions allowed to be called from an ISR. Under no circumstance make calls to other AVIX functions.

Parameters and return value

Parameter	Description
pipeId	Id of the pipe. Since an ISR cannot obtain the id of its pipe through a Get function, this id must be made available by using a global variable.
count	Value passed from ISR with user specified meaning.
Return value	<none>

avixPipe_Write

```
intavixPipe_Write
(
    tavixPipeId  pipeId,
    const void*  pWritebuffer,
    unsigned int nrBlocksToWrite
);
```

avixMain	-	
Thread	✓	04:02
DIH	✓	
ISR	-	

>> function overview

Description

Write data to a pipe. The following scenario's are applicable:

- *Write to a pipe while no requests are pending*: Data is copied from the client provided buffer to the empty space in the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data to be written.

- *Write to a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full and no data is transferred from the client provided buffer to the pipe buffer. A request descriptor is allocated and the new write request is set pending.

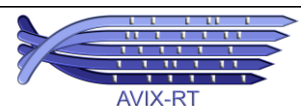
- *Write to a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty. The data of the write request will be copied to the buffers of the pending read requests *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending read requests are updated according the amount of data processed. Pending read requests receiving all required data are set finished and their request descriptors are freed. When all pending read requests are finished and still data remains to be written, this data is copied from the client provided buffer to the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data.

For each of these scenario's:

- *When called from a thread*: When all specified data is written, the function returns immediately. When the requested amount of data cannot be written, the calling thread will enter the 'Blocked' state until sufficient data is read from the pipe, an optional time-out occurs or the buffer is flushed by calling `avixPipe_FlushAndAbort`.

- *When called from a DIH*: Whether or not all specified data is written, the function returns immediately, returning the number of blocks actually transferred. In case of insufficient empty space being available to fulfil the write request, in contradiction with calling this function from a thread, no request descriptor is allocated and the request will not be set pending. Effectively the part of above scenario's which is underlined is not executed when called from a DIH.

When a pipe callback is registered and the first scenario is applicable where one or more blocks are transferred to the pipe buffer, the callback function is activated with flag `PIPEINFO_PIPE_DATA_WRITTEN`.



Parameters and return value

Parameter	Description
pipeId	Id of the pipe to write to.
pWriteBuffer	Pointer to a memory location where the data is present that will be copied to the pipe. It is the responsibility of the user to make sure the content of the referred location holds a valid number of data blocks corresponding to the number passed with parameter nrBlocksToWrite. If this is not the case, data present behind the end of the user supplied buffer will be copied to the pipe.
nrBlocksToWrite	The desired number of data blocks to write to the pipe.
Return value	Number of data blocks actually written to the pipe. This is not necessarily the same number of blocks as specified with parameter nrBlocksToWrite since a timeout may occur, the pipe is flushed or the write operation is performed by a DIH.

► *When using `avixPipe_Write` with a timeout, the number of data blocks written may be less than the specified number. If this happens the pipe content will not be as expected and the data of a subsequent write operation will be appended to the data left behind by the timed out write operation. This might lead to undesirable system behavior. The application must anticipate on this. When not using a timeout, the write operation is guaranteed to write the specified number of data blocks. Each individual data block is guaranteed to be entirely written.*

avixPipe_WriteAsync

```
void avixPipe_WriteAsync
(
    tavixPipeId           pipeId,
    const void*           pWriteBuffer,
    unsigned int          nrBlocksToWrite,
    tavixEventId          readyEvent,
    tavixEventFlags       readyFlags,
    tavixPipeAsyncReqId*  pReqId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Write data to a pipe. The following scenario's are applicable:

- *Write to a pipe while no requests are pending*: Data is copied from the client provided buffer to the empty space in the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data to be written.

- *Write to a pipe while write requests are pending*: Since write requests are pending, the pipe buffer is full and no data is transferred from the client provided buffer to the pipe buffer. A request descriptor is allocated and the new write request is set pending.

- *Write to a pipe while read requests are pending*: Since read requests are pending, the pipe buffer is empty. The data of the write request will be copied to the buffers of the pending read requests *without first being transferred to the pipe buffer*. The content of the request descriptors of the pending read requests are updated according the amount of data processed. Pending read requests receiving all required data are set finished and their request descriptors are freed. When all pending read requests are finished and still data remains to be written, this data is copied from the client provided buffer to the pipe buffer. When the empty space in the pipe buffer is not sufficient to copy all requested data, a request descriptor is allocated and the write request is set pending for the remaining amount of data.

For each of these scenario's, the function returns immediately:

- *When called from a thread*: When all data is written, the request is said to have finished, the optionally specified event flag is set and/or the request id is updated.

  When not all data can be written a request descriptor is added to the pipe internal bookkeeping specifying the remaining amount of data to write. The request is said to be pending, the optionally specified event flag is cleared and/or the request id is updated. The status remains pending until either sufficient data is read, the request is aborted using `avixPipe_AbortAsyncReq` or the buffer is flushed by calling `avixPipe_FlushAndAbort`.

- *When called from a DIH*: When the requested amount of data is written, the request is said to have finished. When the requested amount of data is not written, the request is said to be aborted. The optionally specified event flag is set and/or the request id is updated. In contradiction with calling this function from a thread, no request descriptor is allocated and the request will not be set pending. Effectively the part of above scenario's which is underlined is not executed when called from a DIH.

When a pipe callback is registered and the first scenario is applicable where one or more blocks are transferred to the pipe buffer, the callback function is activated with flag `PIPEINFO_PIPE_DATA_WRITTEN`.

Parameters and return value

Parameter	Description
pipeId	Id of the pipe to write to.
pWriteBuffer	Pointer to a memory location where the data is present that will be copied to the pipe. It is the responsibility of the user to make sure the content of the referred location holds a valid number of data blocks corresponding to the number passed with parameter nrBlocksToWrite. If this is not the case, data present behind the end of the user supplied buffer will be copied to the pipe.
nrBlocksToWrite	The desired number of blocks to write to the pipe.
readyEvent	Id of event group in which flags specified by parameter `readyFlags` are set when the asynchronous write request is finished. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId` • A NULL event group id when no flags need to be set when the request is finished. Use `AVIX_OBJECT_ID_NULL(tavixEventId)`.
readyFlags	Event flags that are manipulated in the event group identified by parameter `readyEvent`. When the asynchronous write request is set pending, the specified flags are cleared. When the asynchronous write request is set finished, the specified flags are set. When for parameter `readyEvent` a NULL event group id is passed, the value of this parameter is don’t care.
pReqId	Address of a user defined request id variable. This variable may be used to abort a pending request using `avixPipe_AbortAsyncReq` or obtain the status of a pending request using `avixPipe_GetStatusAsyncReq`. When this functionality is not required, NULL may be passed for this parameter.
Return value	<none>

avixPipe_WriteFromISR

```
int avixPipe_WriteFromISR
(
    tavixPipeId  pipeId,
    const void*  pWriteBuffer,
    int          nrBlocksToWrite
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Write data to a pipe for use by an ISR.

This function only writes to the pipe buffer and therefore will never write more data than empty space is present in the pipe buffer, regardless whether thread read requests are pending.

This is a non-blocking function since an ISR always runs to completion and cannot block. *When the pipe buffer does not contain enough empty space for the write request to finish, it is up to the programmer to decide what to do in this situation.*

It is essential to check the return value of this function and take appropriate action.

This function fills the pipe buffer. When thread read requests are pending, after the ISR has written to the pipe buffer one of the following two scenario's is applicable:

- If the pipe buffer contains sufficient data to finish the thread read request at the head of the list, data is transferred from the pipe buffer to the thread read request buffer, the thread read request is set finished and the request descriptor is removed from the list. If present, the next pending thread read request will be at the head of the list and the above will be repeated or, if this thread read request requires more data than present in the pipe buffer, the second scenario is applicable.

  For thread read requests that are set finished and belong to a synchronous transfer, the related thread is given the 'Ready' state. For thread read requests that are set finished and belong to an asynchronous transfer, the event flag(s) specified with this request is set and/or the request id is updated.

- If the pipe buffer does not contain sufficient data to finish the thread read request at the head of the list but the pipe buffer is filled for 50% or more, as much as possible data is transferred from the pipe buffer to the buffer of the pending thread read request. Doing so, AVIX assures the empty space in the pipe buffer is always sufficient for ISR write handling.

This function does not activate the callback function which is optionally connected to the pipe on creation.

► *When writing to a pipe from an ISR, the pipe may only be used for thread originating read requests. AVIX does not check for this and it is the users responsibility to guard this.*

Parameters and return value

Parameter	Description
pipeId	Id of the pipe. Since an ISR cannot obtain the id of the pipe through a Get function, this id must made available by using a global variable.
pWriteBuffer	Pointer to a memory location where data is present that will be copied to the pipe. It is the responsibility of the user to make sure the content of the referred location holds a valid number of blocks corresponding to the number passed with parameter nrBlocksToWrite. If this is not the case, data present behind the end of the user supplied buffer will be copied to the pipe.
nrBlocksToWrite	The desired number of data blocks to write to the pipe.
Return value	Number of data blocks actually written to the pipe. This is not necessarily the same number of blocks as specified with parameter nrBlocksToWrite since the pipe might not contain enough empty space to fulfill the write request and an ISR cannot block to wait for the required empty space to become available. Negative value (< 0) when `avixPipe_WriteFromISR` is called while a thread is flushing the pipe using `avixPipe_FlushAndAbort`. In this case no data is written to the pipe buffer.

Pipe related definitions

The following pipe related definitions are provided:

>> definition overview

PIPE_ASYNCREQ_ABORTED
Return value for avixPipe_GetStatusAsyncReq or avixPipe_AbortAsyncReq identifying the request was aborted.

PIPE_ASYNCREQ_FINISHED
Return value for avixPipe_GetStatusAsyncReq or avixPipe_AbortAsyncReq identifying the request has finished.

PIPE_ASYNCREQ_PENDING
Return value for avixPipe_GetStatusAsyncReq or avixPipe_AbortAsyncReq identifying the request is pending.

PIPEINFO_DEVICE_STOP_REQUESTED
Parameter passed to pipe callback function when calling avixPipe_StopDeviceFromISR.

PIPEINFO_PIPE_DATA_WRITTEN
Parameter passed to pipe callback function when a thread writes data to a pipe.

PIPEINFO_READ_FROM_EMPTY_PIPE
Parameter passed to pipe callback function when a thread reads from an empty pipe.

7.6.8 Memory Support

Header file to include: AVIX.h
Service specific header file: AVIXMemory.h

The following functions and definitions are offered:

MEMORY FUNCTIONS						
Function	**page**	**avixMain**	**Thread**	**DIH**		**ISR**
avixMemPool_Allocate	230	-	✓ 04:02	✓	?	-
avixMemPool_AllocateFromISR	231	-	-	-		✓
avixMemPool_Create	232	✓	✓	✓		-
AvixMemPool_CreateExt[29]	233	✓	✓	✓		-
avixMemPool_Free	234	-	✓	✓		-
avixMemPool_FreeFromISR	235	-	-	-		✓
avixMemPool_Get	236	-	✓ 04:02	✓	?	-
avixMemPool_GetSizeBlock	237	-	✓	✓		-
avixMemPool_GetSizeBlockFromISR	238	-	-	-		✓

MEMORY POOL DEFINITIONS	
Definition	**Description**
AVIX_MEM_BLOCK_PTR	Convert a memory block id to a 'C' style pointer
AVIX_MEM_BLOCK_PTR_EXT[29]	Convert an id of an extended memory block to a 'C' style pointer
AVIX_EDS[29]	Tag a pointer to a memory block allocated in Extended RAM
AVIX_MEM_BLOCK_ID_VALID	Test if a memory block id is valid.

[29] Functionality is hardware platform dependant. Consult the applicable Port Guide for details.

avixMemPool_Allocate

```tavixMemBlockId avixMemPool_Allocate (tavixMemPoolId poolId);```	**avixMain**	–
	Thread	✓ 04:02
	DIH	✓
	ISR	–

```
tavixMemBlockId avixMemPool_Allocate
(
    tavixMemPoolId poolId
);
```

>> function overview

Description

Allocate a memory block from the designated memory pool.

When called from a thread and a memory block is available, the function returns immediately with a valid memory block id.

When called from a thread and no memory block is available, the thread enters the 'Blocked' state until a memory block is freed by another thread, DIH or ISR. When using a time-out and the specified time expires before a memory block is made available, the function returns with an invalid memory block id.

When called from a DIH, the function returns immediately, regardless whether a memory block was available or not. When a memory block was available, the returned memory block id is valid. When no memory block was available, the returned memory block id is invalid.

Testing a memory block id for being valid is done by using macro `AVIX_MEM_BLOCK_ID_VALID`[30].

Parameters and return value

Parameter	Description
poolId	Id of the memory pool the block is to be allocated from.
Return value	Id representing the memory block allocated from the pool. In order to access the memory of the block, this id must be converted to a pointer using macro `AVIX_MEM_BLOCK_PTR`.

▶ *The allocated memory is returned 'as is', it is not filled with a specific value. Initializing the memory with the desired value(s) is entirely the responsibility of the user.*

▶ *The size of the memory block may be larger than the size specified when creating the memory pool. This is because the size of the memory blocks is rounded to the first higher value being a multiple of the basic word size of the controller, e.g. 2 bytes for a 16-bit controller and 4 bytes for a 32-bit controller. To obtain the actual size use function avixMemPool_GetSizeBlock.*

▶ *To free the memory block, the id received from this function must be passed to either `avixMemPool_Free` or `avixMemPool_FreeFromISR`. The value passed to one of these functions must be the exact same value received from `avixMemPool_Allocate`. For this purpose, the application must retain the memory block id.*

[30] `AVIX_MEM_BLOCK_ID_VALID` has the same functionality as macro `AVIX_OBJECT_ID_VALID`. For reason of future extension however `AVIX_MEM_BLOCK_ID_VALID` must be used i.s.o. `AVIX_OBJECT_ID_VALID` when testing the validity of a memory block id.

avixMemPool_AllocateFromISR

```
tavixMemBlockId avixMemPool_AllocateFromISR
(
    tavixMemPoolId poolId
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Allocate a memory block from the designated memory pool.

This function returns immediately, regardless whether a memory block was available or not. When a memory block was available, the returned memory block id is valid. When no memory block was available, the returned memory block id is invalid.

Testing a memory block id for being valid is done by using macro `AVIX_MEM_BLOCK_ID_VALID`[31]

Parameters and return value

Parameter	Description
poolId	Id of the memory pool the memory block is to be allocated from.
Return value	Id representing the memory block allocated from the pool. In order to access the memory of the block, this id must be converted to a pointer using macro `AVIX_MEM_BLOCK_PTR`.

▶ *The allocated memory is returned 'as is', it is not filled with a specific value. Initializing the memory with the desired value(s) is entirely the responsibility of the user.*

▶ *The size of the memory block may be larger than the size specified when creating the memory pool. This is because the size of the memory blocks is rounded to the first higher value being a multiple of the basic word size of the controller, e.g. 2 bytes for a 16-bit controller and 4 bytes for a 32-bit controller. To obtain the actual size use function avixMemPool_GetSizeBlockFromISR.*

▶ *To free the memory block, the id received from this function must be passed to either `avixMemPool_Free` or `avixMemPool_FreeFromISR`. The value passed to one of these functions must be the exact same value received from `avixMemPool_Allocate`. For this purpose, the application must retain the memory block id.*

[31] `AVIX_MEM_BLOCK_ID_VALID` has the same functionality as macro `AVIX_OBJECT_ID_VALID`. For reason of future extension however `AVIX_MEM_BLOCK_ID_VALID` must be used i.s.o. `AVIX_OBJECT_ID_VALID` when testing the validity of a memory block id.

avixMemPool_Create

```
tavixMemPoolId avixMemPool_Create
(
    tavixKernelObjectName pName,
    unsigned int          numBlocks,
    unsigned int          blockSize
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Create a memory pool object. If a valid name is specified and other threads are waiting for the created kernel object to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

Parameters and return value

Parameter	Description
pName	Human readable name for the memory pool or `NULL` if the memory pool does not need to have a name. This must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
numBlocks	Number of blocks the memory pool will contain. Allowed values for this parameter range from 1 up to and including 255.
blockSize	Size in bytes of the blocks that can be allocated from the pool. The value passed for this parameter will be rounded to the first higher value being a multiple of the basic word size of the controller, e.g. 2 bytes for a 16-bit controller and 4 bytes for a 32-bit controller.
Return value	Id of the newly created memory pool.

avixMemPool_CreateExt

```
tavixMemPoolId avixMemPool_CreateExt
(
    tavixKernelObjectName pName,
    unsigned int          numBlocks,
    unsigned int          blockSize
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Create a memory pool object in Extended RAM. If a valid name is specified and other threads are waiting for the created kernel object to become existent, those threads enter the 'Ready' state. Do note that the moment such a waiting thread will actually start running is determined by the scheduler.

Behavior of this function is hardware platform and configuration setting specific. Please read the hardware platform specific Port Guide for the applicable controller to learn more.

When using this function with a controller not having Extended RAM or where the amount of Extended RAM that may be used by AVIX is configured to be zero (0), the function behaves identical to `avixMemPool_Create` and the memory pool is created in the main RAM bank. In this situation, use of this function is discouraged and use of `avixMemPool_Create` is preferred.

When using this function with a controller offering Extended RAM and the amount of Extended RAM that may be used by AVIX is configured as a non-zero value, the memory pool is allocated in this Extended RAM.

Parameters and return value

Parameter	Description
pName	Human readable name for the memory pool or `NULL` if the memory pool does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters.
numBlocks	Number of blocks the memory pool will contain. Allowed values for this parameter range from 1 up to and including 255.
blockSize	Size in bytes of the blocks that can be allocated from the pool. The value passed for this parameter will be rounded to the first higher value being a multiple of the basic word size of the controller, e.g. 2 bytes for a 16-bit controller and 4 bytes for a 32-bit controller.
Return value	Id of the newly created memory pool.

avixMemPool_Free

```
tavixMemBlockId avixMemPool_Free
(
    tavixMemBlockId memBlockId
);
```

avixMain	–
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Return a memory block to its memory pool so it is available for subsequent allocation again. When all memory blocks in the memory pool are allocated (pool empty) and one or more threads are waiting for a memory block to become available, the first waiting thread is removed from the wait list and enters the 'Ready' state. The moment this thread will start running is determined by the scheduler.

Parameters and return value

Parameter	Description
memBlockId	Identification of a memory block previously allocated by a call to `avixMemPool_Allocate` or `avixMemPool_AllocateFromISR`
Return value	Id of a memory block not representing an actual block. By assigning the result of this function to the same memory block id variable used as the parameter, the id will be invalidated. When accidentally passing such an invalidated memory block id to `avixMemPool_Free` or `avixMemPool_FreeFromISR` this will lead to an error. This prevents a memory block from accidentally being freed more than once.

▶ *A memory block can be freed without a reference to the pool it belongs to. This makes using memory blocks very easy and user-friendly*

▶ *After returning a memory block to its pool, the id may no longer be used for writing or reading. This might lead to unpredictable behavior since the actual memory block might either be in the pool or allocated by another thread.*

avixMemPool_FreeFromISR

`tavixMemBlockId avixMemPool_FreeFromISR`
`(`
`    tavixMemBlockId memBlockId`
`);`
	avixMain –
	Thread –
	DIH –
	ISR ✓

>> function overview

Description

Return a memory block to its pool so it is available for subsequent allocation again. When all memory blocks in the memory pool are allocated (pool empty) and one or more threads are waiting for a memory block to become available, the first waiting thread is removed from the wait list and enters the 'Ready' state. The moment this thread will start running is determined by the scheduler.

Parameters and return value

Parameter	Description
memBlockId	Identification of a memory block previously allocated by a call to `avixMemPool_Allocate` or `avixMemPool_AllocateFromISR`
Return value	Id of a memory block not representing an actual block. By assigning the result of this function to the same memory block id variable used as the parameter, the id will be invalidated. When accidentally passing such an invalidated memory block id to `avixMemPool_Free` or `avixMemPool_FreeFromISR` this will lead to an error. This prevents a memory block from accidentally being freed more than once.

► *After returning a memory block to its pool, the id may no longer be used for writing or reading. This might lead to unpredictable behavior since the actual memory block might either be in the pool or allocated by another thread.*

avixMemPool_Get

```
tavixMemPoolId avixMemPool_Get
(
    tavixKernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	✓	
ISR	–	

>> function overview

Description

Obtain the id of a memory pool kernel object with a specific name. If the memory pool with the specified name exists the function returns immediately with a memory pool id guaranteed to be valid. If the memory pool with the specified name does not exist behavior depends on the fact whether the function is called from a thread or from a DIH. When called from a thread, the calling thread enters the 'Blocked' state until either the designated memory pool is created by another thread or the optionally specified timeout expires. In case of a timeout the returned id is invalid and may not be used. When the memory pool does not exist and this function is called from a DIH, the function returns immediately with an invalid memory pool id.

Parameters and return value

Parameter	Description
pName	Human readable name for the memory pool whose id is to be obtained through this function. The name must be unique in the first four characters.
Return value	In case of success, id representing the memory pool with the specified name (pName).

avixMemPool_GetSizeBlock

```
int avixMemPool_GetSizeBlock
(
    tavixMemBlockId memBlockId
);
```

avixMain	-
Thread	✓
DIH	✓
ISR	-

>> function overview

Description

Return the size in bytes of a memory block. This function works in case argument memBlockId represents a valid memory block regardless whether this memory block is currently allocated or not.

This function is especially useful when a thread has access to a memory block it did not allocate itself. Before writing data to the memory block the thread can check the size in bytes that it is free to write.

Do not use the return value of this function to determine whether or not a memory block is currently allocated or present for allocation in its pool. This knowledge is required to be present in the application logic.

Parameters and return value

Parameter	Description
memBlockId	Id of a memory block.
Return value	In case memBlockId represents a valid memory block the size in bytes of the memory block is returned. In case the id does not represent a valid memory block, -1 is returned.

avixMemPool_GetSizeBlockFromISR

```
int avixMemPool_GetSizeBlockFromISR
(
    tavixMemBlockId memBlockId
);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Return the size in bytes of a memory block. This function works in case argument memBlockId represents a valid memory block regardless whether this memory block is currently allocated or not.

This function is especially useful when an ISR has access to a memory block it did not allocate itself. Before writing data to the memory block the ISR can check the size in bytes that it is free to write.

► *Do not use the return value of this function to determine whether or not a memory block is currently allocated or present for allocation in its pool. This knowledge is required to be present in the application logic.*

Parameters and return value

Parameter	Description
memBlockId	Id of a memory block.
Return value	In case memBlockId represents a valid memory block the size in bytes of the memory block is returned. In case the id does not represent a valid memory block, -1 is returned.

Memory pool / block related definitions

The following memory pool / block related definitions are provided: >> definition overview

`AVIX_MEM_BLOCK_PTR(type, id)`
Convert the id represented by parameter ‘id’ to a memory pointer of a type as represented by parameter ‘type’. The macro adds the ‘C’ pointer ‘*’ character to parameter ‘type’. The pointer value returned by this macro can be used to directly write to or read from the memory block. The pointer value returned by this macro cannot be manipulated, in order to do so; the macro return value must be copied to a plain ‘C’ pointer before exercising the required pointer manipulation.

`AVIX_MEM_BLOCK_PTR_EXT(type, id)`
Convert the id represented by parameter ‘id’ to a direct memory pointer of a type as represented by parameter ‘type’. The macro adds the ‘C’ pointer ‘*’ character to parameter ‘type’. The pointer value returned by this macro can be used to directly write to or read from the memory block. The pointer value returned by this macro cannot be manipulated, in order to do so; the macro return value must be copied to a plain ‘C’ pointer before exercising the required pointer manipulation. *Behavior of this macro is hardware platform and configuration setting specific. Please read the hardware platform specific Port Guide for the applicable controller to learn more.*

`AVIX_EDS`
Macro used to ‘tag’ pointers to memory blocks allocated from memory pools in Extended RAM. *Behavior of this macro is hardware platform and configuration setting specific. Please read the hardware platform specific Port Guide for the applicable controller to learn more.*

`AVIX_MEM_BLOCK_ID_VALID(id)`
Test if the memory block id represented by parameter ‘id’ is valid. When valid, a value unequal to zero(0) is returned else a value equal to zero(0). This macro is only to be used on block id’s returned by one of the memory pool functions. Using the macro on a memory block id that just has been declared but not yet have been assigned a result of one of the memory pool functions returns a meaningless value. Invalid memory block id’s can be returned by `avixMemPool_AllocateFromISR` and are returned by `avixMemPool_Free` and `avixMemPool_FreeFromISR`. The main use of this macro is to test whether an allocation done through `avixMemPool_AllocateFromISR` was successful since this function returns an invalidated memory block id in case the memory pool does not contain any free memory blocks the moment the call is made. *`AVIX_MEM_BLOCK_ID_VALID` has the same functionality as macro `AVIX_OBJECT_ID_VALID`. For reason of future extension however, always use this macro to check for valid memory block id’s and do not use `AVIX_OBJECT_ID_VALID` for this purpose.*

7.6.9 Exchange Support

Header file to include: AVIX.h
Service specific header file: AVIXExchange.h

The following functions and definitions are offered:

EXCHANGE FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixExch_ConnectCallbackEventGroup	241	✓	✓	–	–
avixExch_ConnectCallbackPipe	243	✓	✓	–	–
avixExch_ConnectCallbackThread	245	✓	✓	–	–
avixExch_ConnectEventGroup	247	✓	✓	–	–
avixExch_ConnectMsgQThread	248	✓	✓	–	–
avixExch_ConnectPipe	249	✓	✓	–	–
avixExch_Create	250	✓	✓	–	–
avixExch_DisableActiveConnection	251	–	✓	–	–
avixExch_DisableAllConnections	252	✓	✓	–	–
avixExch_DisableConnection	253	✓	✓	–	–
avixExch_EnableAllConnections	254	✓	✓	–	–
avixExch_EnableConnection	255	✓	✓	–	–
avixExch_Get	256	✓	✓ 04:02	–	–
avixExch_GetConnectionExch	257	✓	✓	–	–
avixExch_GetConnectionMode	258	✓	✓	–	–
avixExch_GetLastWriteThread	259	✓	✓	–	–
avixExch_GetSize	260	✓	✓	–	
avixExch_Lock	261	✓	✓	–	–
avixExch_Read	262	✓	✓	–	–
avixExch_Unlock	263	✓	✓	–	–
avixExch_UnlockAndNotify	264	✓	✓	–	–
avixExch_Write	265	✓	✓	–	–

EXCHANGE DEFINITIONS	
Definition	**Definition**
AVIX_EXCHANGE_DATA_PTR	Convert an Exchange data pointer to 'C' style pointer
AVIX_EXCHANGE_DATA_PTR_READ	Convert an Exchange data pointer to 'C' style pointer to const

avixExch_ConnectCallbackEventGroup

```
tavixExchConnId avixExch_ConnectCallbackEventGroup
(
    tavixExchId                  exchangeId,
    tavixExchCallbackEventGroup  pCallback,
    tavixEventId                 eventId,
    unsigned short               userParam
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create a ‘callback-event-group-connection’ by connecting a user specified callback function to an Exchange object. The callback function receives the specified event group id and user parameter as its parameters.

Both a global event group or a thread local event group may be used as a parameter for the callback function. For a thread event group, the id of the thread must be converted to an event group id using attribute `.asEventId` on the thread id.

After connecting the callback function to an Exchange object, the callback function will be called whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

A ‘callback-event-group-connection’ is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id’s will be the same, regardless the exchange object the belong to.

Trying to connect the same event group-callback combination to a specific Exchange object is not allowed and results in an error. Additional ‘callback-event-group-connections’ can only be made if either the specified event group id and/or callback address are different from existing ‘callback-event-group-connections’.

By default, the mode of a newly created connection is enabled.

An event group callback function must have the following prototype:

```
void exchangeCallbackEventGroup
(
    tavixExchId    exchange, // Id of Exchange callback is connected to
    const void*    pData,    // pointer to Exchange data section
    unsigned int   size,     // size in bytes of Exchange data section
    tavixEventId   eventId,  // eventId passed to connect function
    unsigned short userParam // userParam passed to connect function
);
```

This type of callback is typically used to process the data in the callback and subsequently change flags in the specified event group. In this case the flags must be changed by a call to `avixEventGroup_Change` made from the callback function.

When no custom data processing is required but only flags in the event group need to be changed, use `avixExch_ConnectEventGroup` instead. In this case the call to `avixEventGroup_Change` is made by the Exchange service itself and no user specified callback function is required.

A callback function runs in the context of the thread executing the write operation to the Exchange.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the callback to.
eventId	Id of an event group, the value of which will be passed to the callback function as the fourth parameter. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId`.
userParam	User specified numeric value passed to the callback function as the fifth parameter. The content of this 16-bit parameter is user specified. Typically this parameter holds event flags that are set in the specified event group by the callback function.
pCallback	Address of the actual callback function.
Return value	Id uniquely identifying the connection.

avixExch_ConnectCallbackPipe

```
tavixExchConnId avixExch_ConnectCallbackPipe
(
    tavixExchId             exchangeId,
    tavixExchCallbackPipe pCallback,
    tavixPipeId             pipeId,
    unsigned short          userParam
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create a 'callback-pipe-connection' by connecting a user specified callback function to an Exchange object. When called, the callback function receives the specified pipe id and user parameter as its parameters.

After connecting the callback function to an Exchange object, the callback function will be called whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

A 'callback-pipe-connection' is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id's will be the same, regardless the exchange object the belong to.

Trying to connect the same pipe-callback combination to a specific Exchange object is not allowed and results in an error. Additional 'callback-pipe-connections' can only be made if either the specified pipe id and/or callback address are different from existing 'callback-pipe-connections'.

By default, the mode of a newly created connection is enabled.

A pipe callback function must have the following prototype:

```
void exchangeCallbackPipe
(
    tavixExchId    exchange, // Id of Exchange callback is connected to
    const void*    pData,    // pointer to Exchange data section
    unsigned int   size,     // size in bytes of Exchange data section
    tavixPipeId    pipeId,   // pipeId passed to connect function
    unsigned short userParam // userParam passed to connect function
);
```

This type of callback is typically used to process the data in the callback and subsequently write data to the specified pipe. In this case the data must be written to the pipe by a call to `avixPipe_Write` or `avixPipe_WriteAsync` made from the callback function. When using `avixPipe_WriteAsync`, make sure that parameters passed to this function have a global scope and are not defined on the stack frame of the callback function since this will be freed when the callback returns.

When no custom data processing is required but only data needs to be written to the pipe, use `avixExch_ConnectPipe` instead. In this case the call to `avixPipe_Write` is made by the Exchange service itself.

A callback function runs in the context of the thread executing the write operation to the Exchange.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the callback to.
pipeId	Id of a pipe, the value of which will be passed to the callback function as the fourth parameter.
userParam	User specified numeric value passed to the callback function as the fifth parameter. The content of this 16-bit parameter is user specified. Typically this parameter holds event flags that are set in the specified event group by the callback function.
pCallback	Address of the actual callback function.
Return value	Id uniquely identifying the connection.

avixExch_ConnectCallbackThread

```
tavixExchConnId avixExch_ConnectCallbackThread
(
    tavixExchId                exchangeId,
    tavixExchCallbackThread pCallback,
    tavixThreadId              threadId,
    unsigned short             userParam
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create a ‘callback-thread-connection’ by connecting a user specified callback function to an Exchange object. When called, the callback function receives the specified thread id and user parameter as its parameters.

After connecting the callback function to an Exchange object, the callback function will be called whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

A ‘callback-thread-connection’ is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id’s will be the same, regardless the exchange object the belong to.

Trying to connect the same thread-callback combination to a specific Exchange object is not allowed and results in an error. Additional ‘callback-thread-connections’ can only be made if either the specified thread id and/or callback address are different from existing callback-thread-connections’.

By default, the mode of a newly created connection is enabled.

A thread callback function must have the following prototype:

```
void exchangeCallbackThread
(
    tavixExchId    exchange, // Id of Exchange callback is connected to
    const void*    pData,    // pointer to Exchange data section
    unsigned int   size,     // size in bytes of Exchange data section
    tavixThreadId  threadId, // threadId passed to connect function
    unsigned short userParam // userParam passed to connect function
);
```

This type of callback is typically used to process the data in the callback and subsequently send a message to the message queue of the thread identified by threadId. In this case the message must be allocated, filled and send by the callback function.

When no custom data processing is required but only a message needs to be send, use `avixExch_ConnectMsgQThread` instead. In this case the message allocation, filling and sending is taken care of by the Exchange service itself.

Optionally the thread id passed to the callback function may be converted to an event group id using attribute `.asEventId` which in turn may be used to change flags in the thread local event group.

A callback function runs in the context of the thread executing the write operation to the Exchange.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the callback to.
threadId	Id of a thread, the value of which will be passed to the callback function as the fourth parameter.
userParam	User specified numeric value passed to the callback function as the fifth parameter. The content of this 16-bit parameter is user specified. Typically this parameter holds event flags that are set in the specified event group by the callback function.
pCallback	Address of the actual callback function.
Return value	Id uniquely identifying the connection.

avixExch_ConnectEventGroup

```
tavixExchConnId avixExch_ConnectEventGroup
(
    tavixExchId     exchangeId,
    tavixEventId    eventId,
    tavixEventFlags eventFlags
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create an 'event-group-connection' by connecting an event group id to an Exchange object.

After connecting the event group to an Exchange object, the specified event flags will be set whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

An 'event-group-connection' is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id's will be the same, regardless the exchange object the belong to.

Trying to connect the same event group to a specific Exchange object is not allowed and results in an error. Additional 'event-group-connections' can only be made if the specified event group id is different from existing event-group-connections'.

By default, the mode of a newly created connection is enabled.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the thread event group to.
eventId	Id of an event group the event flags of which will be set. This id may either be: • The id of an event group. • The id of a thread event group. In this case the thread id must be converted to an event group id using attribute `.asEventId` on the thread id. When passing the thread event group id of the calling thread, use `avixThread_GetIdCurrent().asEventId`.
eventFlags	Mask specifying the event flags to set in the event group.
Return value	Id uniquely identifying the connection.

avixExch_ConnectMsgQThread

```
tavixExchConnId avixExch_ConnectMsgQThread
(
    tavixExchId    exchangeId,
    tavixThreadId  threadId,
    tavixMsgType   msgType
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create a 'thread-message-queue-connection' by connecting a thread id to an Exchange object.

After connecting the thread to an Exchange object, the content of the exchange will be send to the thread message queue as a message whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

A 'thread-message-queue-connection' is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id's will be the same, regardless the exchange object the belong to.

Trying to connect the same thread to a specific Exchange object is not allowed and results in an error. Additional 'thread-message-queue-connections' can only be made if the specified thread id is different from existing 'thread-message-queue-connections'.

A message contains the id of the thread it is sent by. For a message send as a result of a write operation to an Exchange object this is the id of the thread that executed the write operation.

Thread message queue connections can only be made for exchanges with a size smaller or equal to the message body size.

The message generated by the exchange is allocated from the regular message pool. When using this function it is required to have a non-zero sized memory pool allocated through configuration parameter `avix_MSG_POOL_NR_MESSAGES`.

By default, the mode of a newly created connection is enabled.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the thread message queue to.
threadId	Id of the thread whose message queue will be connected to the exchange.
msgType	User specified numeric value that will be written to the message as its type to be able to differentiate it from other messages.
Return value	Id uniquely identifying the connection.

avixExch_ConnectPipe

```
tavixExchConnId avixExch_ConnectPipe
(
    tavixExchId exchangeId,
    tavixPipeId pipeId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create a 'pipe-connection' by connecting a pipe id to an Exchange object.

After connecting the pipe to an Exchange object, the content of the exchange will be written to the pipe as a single block whenever data is written to the Exchange object using `avixExch_Write` or after a direct write operation terminated by `avixExch_UnlockAndNotify`.

A 'pipe-connection' is uniquely identified by the connection id this function returns. The connection will remain valid for the lifetime of the application. A connection id has a global scope, no two connection id's will be the same, regardless the exchange object the belong to.

Trying to connect the same pipe to a specific Exchange object is not allowed and results in an error. Additional 'pipe-connections' can only be made if the specified pipe id is different from existing 'pipe-connections'.

Pipe connections can only be made if the pipe block size equals the Exchange object data size.

By default, the mode of a newly created connection is enabled.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to connect the pipe to.
pipeId	Id of the pipe to connect to the exchange.
Return value	Id uniquely identifying the connection.

avixExch_Create

```
tavixExchangeId avixExch_Create
(
    tavixKernelObjectName pName,
    unsigned int          size,
    const void*           pData
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Create an Exchange object. If a valid name is specified and other threads are waiting for the Exchange object to become existent; those threads enter the 'Ready state. The moment such a waiting thread will actually start running is determined by the scheduler.

An Exchange object holds a data section sized after the value of parameter 'size'. The value is rounded up to the nearest value larger or equal to size being a multiple of the word size of the applied microcontroller (multiple of 2 for 16 bit controllers and 4 for 32 bit controllers).

The content of the data pointed to by parameter pData is copied to the Exchange object data section. When passing NULL for this parameter, the content of the Exchange object internal data section is filled with all zeros.

An Exchange holds the id of the thread that executed the last write operation. After creation, this id is set to be a null-id.

Parameters and return value

Parameter	Description
pName	Human readable name for the exchange or `NULL` if the exchange does not need to have a name. This name must be unique throughout the application, no two kernel objects are allowed to have the same name. The name must be unique in the first four characters
Size	Size in bytes of the Exchange data section.
pData	Pointer to data to be copied to the Exchange or NULL when the Exchange must be filled with all zeros.
Return value	Id representing the newly created exchange.

avixExch_DisableActiveConnection

```
void avixExch_DisableActiveConnection
(
    tavixExchId exchangeId
);
```

Context	
avixMain[32]	✓
Thread[32]	✓
DIH	-
ISR	-

>> function overview

Description

A connection can be either enabled or disabled. An enabled connection is activated every time the Exchange object is written. The action depends on the type of connection.

Using this function, the currently active callback connection is disabled. This function is for exclusive use from an Exchange callback function.

The Exchange id passed to this function must be the Exchange id received as the first parameter of the callback function.

Using this function with another Exchange id or from a non-callback context has no effect and does not change the mode of any connection of that Exchange object.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange for which to disable the currently active callback connection. This must be the Exchange id which is passed to the callback function as the first parameter.
Return value	<none>

[32] This function only has effect when called from an active callback function where the Exchange id is the id passed as the first parameter to the callback function. Calling the function with a different Exchange id or from a non-callback context has no effect.

avixExch_DisableAllConnections

```
void avixExch_DisableAllConnections
(
    tavixExchId exchangeId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

A connection can be either enabled or disabled. An enabled connection is activated every time the Exchange object is written. The action depends on the type of connection.

Using this function, for the specified Exchange object, all connections are disabled. While disabled, a connection is not activated when the Exchange object is written.

Connections that are already disabled are not influenced by this function and such connections remain disabled.

AVIX does not allow removing connections. Once a connection to an Exchange object is created, the connection remains related to the Exchange object for the entire duration of the application.

Parameters and return value

Parameter	Description
tavixExchId	Id of the exchange object to disable the connections for.
Return value	<none>

avixExch_DisableConnection

```
void avixExch_DisableConnection
(
    tavixExchConnId connectionId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

A connection can be either enabled or disabled. An enabled connection is activated every time the Exchange object is written. The action depends on the type of connection.

Using this function, the specified connection is disabled. While disabled, a connection is not activated when the Exchange object is written.

If the specified connection is already disabled it is not influenced by this function and remains disabled.

AVIX does not allow removing connections. Once a connection to an Exchange object is created, the connection remains related to the Exchange object for the entire duration of the application.

Parameters and return value

Parameter	Description
tavixExchConnId	Id of the connection to disable. A connection id is returned by the `avixExch_Connect...` functions.
Return value	<none>

avixExch_EnableAllConnections

```
void avixExch_EnableAllConnections
(
    tavixExchId exchangeId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

A connection can be either enabled or disabled. An enabled connection is activated every time the Exchange object is written. The action depends on the type of connection.

Using this function, for the specified Exchange object, all connections are enabled. While enabled, a connection is activated when the Exchange object is written.

Connections that are already enabled are not influenced by this function and such connections remain enabled.

AVIX does not allow removing connections. Once a connection to an Exchange object is created, the connection remains related to the Exchange object for the entire duration of the application.

Parameters and return value

Parameter	Description
tavixExchId	Id of the exchange object to enable the connections for.
Return value	<none>

avixExch_EnableConnection

```
void avixExch_EnableConnection
(
    tavixExchConnId connectionId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

A connection can be either enabled or disabled. An enabled connection is activated every time the Exchange object is written. The action depends on the type of connection.

Using this function, the specified connection is enabled. While enabled, a connection is activated when the Exchange object is written.

If the specified connection is already enabled, it is not influenced by this function and remains enabled.

AVIX does not allow removing connections. Once a connection to an Exchange object is created, the connection remains related to the Exchange object for the entire duration of the application.

Parameters and return value

Parameter	Description
tavixExchConnId	Id of the connection to enable. A connection id is returned by the `avixExch_Connect...` functions.
Return value	<none>

avixExch_Get

```
tavixExchangeId avixExch_Get
(
    tavixKernelObjectName pName
);
```

avixMain	–	
Thread	✓	04:02
DIH	–	
ISR	–	

>> function overview

Description

Obtain the id of an Exchange object with a specific name. If the Exchange object with the specified name exists, the function returns immediately with an Exchange id guaranteed to be valid.

If the Exchange object with the specified name does not exist, the calling thread enters the 'Blocked' state until the requested Exchange object is created or an optionally specified time-out expires.

Parameters and return value

Parameter	Description
pName	Human readable name of the exchange whose id is to be retrieved through this function. The name must be unique in the first four characters.
Return value	In case of success, id of the exchange represented by the specified name (pName). In case of an (optional) time-out, an invalid id.

avixExch_GetConnectionExch

```
tavixExchangeId avixExch_GetConnectionExch
(
    tavixExchConnId connectionId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Obtain the id of the Exchange object the specified connection belongs to.

Parameters and return value

Parameter	Description
connectionId	Connection id for which to obtain the related Exchange id.
Return value	Id of the exchange the specified connection belongs to.

avixExch_GetConnectionMode

```
int avixExch_GetConnectionMode
(
    tavixExchConnId connectionId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

A connection can either be in the enabled or the disabled mode. This function returns the current mode of the specified connection.

Since an Exchange object is accessible from many threads, the possibility exists that in between the function returning and the return value being used, another thread can change the connection mode. The value returned by this function is only guaranteed to reflect the actual mode of the connection if no other threads use connection enable/disable functions.

Parameters and return value

Parameter	Description
connectionId	Id of the connection to determine the state of.
Return value	Unequal zero(‘true’), the connection is enabled. Equal zero(‘false’), the connection is disabled.

avixExch_GetLastWriteThread

`tavixThreadId avixExch_GetLastWriteThread (     tavixExchId exchangeId );`	avixMain[33] ✓ Thread ✓ DIH – ISR –

>> function overview

Description

Get the thread id of the last thread that executed a write operation to the Exchange object. As long as the Exchange object has not been written by a thread, the returned id will be a null id.

Since an Exchange object is accessible from many threads, the possibility exists that in between the function returning and the return value being used, another thread has written the Exchange object. The value returned by this function is only guaranteed to reflect the correct thread if the Exchange object is written from one thread only.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange to obtain the last write thread id from.
Return value	Id of the last thread that executed a write operation.

[33] When called from avixMain, the Exchange object cannot yet been written by a thread since no thread has been active yet so the returned thread id will always be a null id.

avixExch_GetSize

```
unsigned int avixExch_GetSize
(
    tavixExchangeId exchangeId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Return the size in bytes of the data section of an Exchange object. The value returned by this function is not necessarily the same as the size specified when creating the Exchange.

When creating an Exchange, the value of the size parameter is rounded up to the nearest value larger or equal to size being a multiple of the word size of the applied microcontroller (multiple of 2 for 16 bit controllers and 4 four 32 bit controllers). It is this rounded value that is returned by this function.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to retrieve the size of.
Return value	Size of the exchange. This is the size as specified when creating the exchange through `avixExchange_Create`.

avixExch_Lock

```
void* avixExch_Lock
(
    tavixExchangeId exchangeId
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Lock the Exchange object to allow direct access to its data section within a critical section.

Direct access is possible by using the pointer returned by this function. For direct access to be used this pointer must be cast to the data type contained in the Exchange.

Using direct access, AVIX cannot check if write operations occur within the boundaries of the Exchange data section. Writing outside the boundaries of the Exchange data section will lead to unpredictable application behavior.

When using direct access, be very careful to use the correct data type.

This function must always be used paired with either function `avixExchange_Unlock` or `avixExchange_UnlockAndNotify`.

▶ *As long as the lock is active, the calling thread can be considered to 'own' the Exchange and may access the Exchange data by using the pointer returned by this function. After having finished the sequence by calling either `avixExchange_Unlock` of `avixExchange_UnlockAndNotify`, make sure no longer to use this pointer since this will lead to corruption of the Exchange data.*

▶ *A lock on the Exchange should be maintained as short as possible in order for the scheduling to remain fair. As long as the lock is active, no other threads can access the Exchange and these threads will be blocked until the Exchange is unlocked again.*

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to lock.
Return value	Pointer to the first byte of the Exchange data section.

avixExch_Read

```
void avixExch_Read
(
    tavixExchangeId exchangeId,
    void*           pBuffer,
    tavixThreadId*  pThreadIdWriter
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Read the content of the Exchange data section by making a copy of this data to a user supplied buffer.

The number of bytes copied to the user supplied buffer is equal to the size of the Exchange specified when it is created through `avixExchange_Create`. The user supplied buffer should be large enough to at least hold this number of bytes.

Supplying the address of a buffer having an insufficient size will result in application data being overwritten leading to an unstable application and unpredictable behavior.

Parameters and return value

Parameter	Description
exchangeId	Id of the Exchange object to read from.
pBuffer	Address of a user supplied buffer the Exchange data will be copied to.
pThreadIdWriter	Pointer to a thread id variable that will hold the id of the thread that executed the last write operation. This parameter is allowed to have value NULL, in which case this thread id will not be returned. The thread id returned to the location referred by this parameter is only a valid thread id when the Exchange is written by a thread. The thread id returned may be invalid (null id), meaning the Exchange is only written from `avixMain`.
Return value	<none>

avixExch_Unlock

```
void avixExch_Unlock
(
    tavixExchangeId exchangeId,
);
```

avixMain	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Unlock the Exchange to end a critical section started with `avixExchange_Lock` and allow other threads access to the Exchange.

Using this function will end a critical section guarding Exchange direct access.

This function must be used when the type of direct access is 'read' and within the critical section the content of the Exchange data section is not modified. In that case no changes are made to the Exchange object data section and no activation of connections is required.

When writing to the Exchange data section within the critical section use `avixExchange_UnlockAndNotify` to end the critical section instead and activate the connections to be informed that changes are made to the Exchange object data section.

This function must always be used paired with function `avixExchange_Lock`.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to unlock.
Return value	<none>

avixExch_UnlockAndNotify

<table>
<tr><td>

```
int avixExch_UnlockAndNotify
(
    tavixExchangeId exchangeId,
);
```

</td><td>

avixMain ✓
Thread ✓
DIH -
ISR -

</td></tr>
</table>

>> function overview

Description

Unlock the Exchange to end a critical section started with `avixExchange_Lock` and allow other threads access to the Exchange.

Using this function will end a critical section guarding Exchange direct access. This function must be used when the type of direct access is ‘write’ and within the critical section the content of the Exchange data section is modified using the pointer returned by `avixExchange_Lock`.

The id of the thread executing this function will be stored in the Exchange bookkeeping. This thread id is passed when reading the Exchange content through `avixExchange_Read` and as the sender thread id in messages send as a result of active thread message queue connections.

This function will result in all enabled connections being activated. The AVIX functions used in connections (explicit call-back functions or Exchange Services internal connections) may be potentially blocking. If so, these functions are called with a time-out value of zero so they will not block but behave like a timed-out activation instead.

The activation of the connections occurs within a critical section. This critical section is implemented using an Exchange local Mutex. This implies the operation is not atomic for its entire duration. When during the operation higher priority threads enter the ‘Ready’ state, the scheduler will be activated and the thread unlocking the Exchange is preempted in favor of such a higher priority thread. Only when such a thread tries to access the Exchange this will result in this thread being ‘Blocked’.

Just like a regular Mutex, the critical section implemented by an Exchange offers priority inheritance. As a result, although the duration of an Unlock operation is not deterministic (the number of connections may vary and thus the execution time), fair scheduling and fast activation of higher priority threads is still guaranteed.

When only reading from the Exchange data section within the critical section use `avixExchange_Unlock` to end the critical section instead.

This function must always be used paired with function `avixExchange_Lock`.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to unlock.
Return value	Unequal zero(‘true’), one or more of the connections used a potentially blocking function the activation of which resulted in a time-out. Equal zero(‘false’), none of the connection activated potentially blocking functions experienced a time-out.

avixExch_Write

```
int avixExch_Write
(
    tavixExchangeId exchangeId,
    void*           pBuffer
);
```

avixMain[34]	✓
Thread	✓
DIH	-
ISR	-

>> function overview

Description

Write the Exchange data section. The content of the user supplied buffer is copied to the Exchange object data section.

The number of bytes copied to the Exchange is equal to the size of the Exchange specified when it is created through `avixExchange_Create`, rounded up to the nearest value larger or equal to size being a multiple of the word size of the applied microcontroller (multiple of 2 for 16 bit controllers and 4 four 32 bit controllers).

Only in case this function is called from a thread:

The id of the thread executing this function will be stored in the Exchange bookkeeping. This thread id is passed when reading the Exchange content through `avixExchange_Read` and as the sender thread id in messages send as a result of active thread message queue connections.

This function will result in all enabled connections being activated. The AVIX functions used in connections (explicit call-back functions or Exchange Services internal connections) may be potentially blocking. If so, these functions are called with a time-out value of zero so they will not block but behave like a timed-out activation instead.

Both the actual write operation and the activation of the connections occur within a critical section. This critical section is implemented using an Exchange local Mutex. This implies the operation is not atomic for its entire duration. When during the operation higher priority threads enter the 'Ready' state, the scheduler will be activated and the thread writing to the Exchange is preempted in favor of such a higher priority thread. Only when such a thread tries to access the Exchange this will result in this thread being 'Blocked'.

Just like a regular Mutex, the critical section implemented by an Exchange offers priority inheritance. As a result, although the duration of a Write operation is not deterministic (the number of connections may vary and thus the execution time), fair scheduling and fast activation of higher priority threads is still guaranteed.

When called from `avixMain` the thread id remains a null-id, connections are not activated and the function return value is always zero (0). Using this function from `avixMain` is especially during the initialization phase of the application.

[34] When called from `avixMain`, existing connections are not activated and no thread id is remembered by the Exchange.

Parameters and return value

Parameter	Description
exchangeId	Id of the exchange object to unlock.
pBuffer	Address of a user supplied buffer containing the data to write to the Exchange.
Return value	Unequal zero('true'), one or more of the connections used a potentially blocking function the activation of which resulted in a time-out. Equal zero('false'), none of the connection activated potentially blocking functions experienced a time-out or the function is called from `avixMain`.

Exchange related definitions

The following Exchange related definitions are provided:

>> definition overview

`AVIX_EXCH_DATA_PTR(_type, _p)`
For efficient read/write operations, the data section of an Exchange can be accessed directly by using a pointer. This pointer is returned by function `avixExchange_Lock`. The type of this pointer is `void*`. The actual content of an Exchange will always be some user specific type. This macro can be used to convert the pointer returned by `avixExchange_Lock` to the user specific type. The pointer returned by this macro can be used both for reading and writing. Do not use this macro when: • Only read access to the Exchange data section is required. • Using the data pointer passed to a callback for Exchange data section access For these situations, use macro `AVIX_EXCHANGE_DATA_PTR_READ`. This macro generates a compile error when used for writing.

`AVIX_EXCH_DATA_PTR_READ(_type, _p)`
For efficient read/write operations, the data section of an Exchange can be accessed directly by using a pointer. This pointer is returned by function `avixExchange_Lock`. The type of this pointer is `void*`. The actual content of an Exchange will always be some user specific type. This macro can be used to convert the pointer returned by `avixExchange_Lock` to the user specific type. The pointer returned by this macro can be used both for reading only. Use this macro when: • Only read access to the Exchange data section is required. • Using the data pointer passed to a callback for Exchange data section access

7.6.10 Power Mode support

Header file to include: AVIX.h
Service specific header file: AVIXPower.h

The following functions and definitions are offered:

POWER MANAGEMENT FUNCTIONS						
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixPower_GetMode	269	✓	✓	✓	-
avixPower_GetModeFromISR	270	-	-	-	✓
avixPower_SetCallback	271	✓	✓	✓	-
avixPower_SetMode	272	✓	✓	✓	-
avixPower_SetModeFromISR	273	-	-	-	✓

POWER MANAGEMENT DEFINITIONS	
Defnition	**Description**
AVIX_POWER_REDUCTION_NONE	No power reduction mode will be used
AVIX_POWER_REDUCTION_LOW	Controller will switch to low power mode basic energy saving
AVIX_POWER_REDUCTION_HIGH	Controller will switch to low power mode high energy saving

Although the power management programming interface is hardware platform independent, the implementation may be different per hardware platform. Consult the hardware platform Port Guide for details.

avixPower_GetMode

`tavixPowerMode avixPower_GetMode(void);`	**avixMain** ✓ **Thread** ✓ **DIH** ✓ **ISR** –

>> function overview

Description

Return the currently selected power mode. The mode returned will be used by AVIX as soon as the idle thread is active which denotes no application code needs to run and thus a power saving mode can be applied.

Parameters and return value

Parameter	Description
<none>	
Return value	The currently selected power mode. Possible values: • `AVIX_POWER_REDUCTION_NONE` • `AVIX_POWER_REDUCTION_LOW` • `AVIX_POWER_REDUCTION_HIGH`

avixPower_GetModeFromISR

```
tavixPowerMode avixPower_GetModeFromISR(void);
```

avixMain	–
Thread	–
DIH	–
ISR	✓

>> function overview

Description

Return the currently selected power mode to the calling ISR. The mode returned will be used by AVIX as soon as the idle thread is active which denotes no application code needs to run and thus a power saving mode can be applied.

Parameters and return value

Parameter	Description
<none>	
Return value	The currently selected power mode. Possible values: • `AVIX_POWER_REDUCTION_NONE` • `AVIX_POWER_REDUCTION_LOW` • `AVIX_POWER_REDUCTION_HIGH`

avixPower_SetCallback

```
void avixPower_SetCallback
(
    tavixPowerCallbackFunc pFunc
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Register a callback allowing AVIX to inform the application about the power mode the idle thread is about to activate.

To unregister the callback pass `NULL`.

▶ *Although activated from the idle thread, the callback is running as a DIH and thus the AVIX functions allowed to be called from the callback are subject to the restrictions of a DIH.*

A power mode callback must have the following signature:

```
void powerCallback(tavixPowerMode powerMode);
```

Parameters and return value

Parameter	Description
pFunc	Pointer to power mode callback function. To disable the callback from being called use `NULL`.
Return value	<none>

avixPower_SetMode

```
void avixPower_SetMode
(
    tavixPowerMode mode
);
```

avixMain	✓
Thread	✓
DIH	✓
ISR	–

>> function overview

Description

Select the desired power mode. This function does not activate the controller's power mode but sets a flag which is used by the AVIX idle thread to activate the selected power mode the moment no application code is executing.

Regardless the moment this function is called, even the last instruction just before the power mode is activated at controller level, the selected power mode overrules the current and is the one that will be used next time the idle thread decides a power mode can be activated.

When setting a power mode from inside the callback the following is very important: Setting the power mode will 'reset' the internal bookkeeping. As a result, before actually effectuating the selected power mode, the callback will be called again, now with the newly selected power mode. Only after the callback returns without calling `avixPower_SetMode`, the desired power mode will be effectuated. For this reason, setting a power mode from within the callback should only be done when the power mode has to change. Unconditionally calling `avixPower_SetMode` every time the callback is activated will never effectuate the desired power mode.

Parameters and return value

Parameter	Description
mode	Desired power mode. Possible values: • `AVIX_POWER_REDUCTION_NONE` • `AVIX_POWER_REDUCTION_LOW` • `AVIX_POWER_REDUCTION_HIGH`
Return value	<none>

avixPower_SetModeFromISR

```void avixPower_SetModeFromISR (tavixPowerMode mode);```	**avixMain** – **Thread** – **DIH** – **ISR** ✓

>> function overview

Description

Select the desired power mode from an ISR. This function does not activate the controller's power mode but sets a flag which is used by the AVIX idle thread to activate the selected power mode the moment no application code is executing.

Regardless the moment this function is called, even the last instruction just before the power mode is activated at controller level, the selected power mode overrules the current and is the one that will be used next time the idle thread decides a power mode can be activated.

Parameters and return value

Parameter	Description
mode	Desired power mode. Possible values: • `AVIX_POWER_REDUCTION_NONE` • `AVIX_POWER_REDUCTION_LOW` • `AVIX_POWER_REDUCTION_HIGH`
Return value	<none>

Power Mode related definitions

The following power mode related definitions are provided:

>> definition overview

AVIX_POWER_REDUCTION_HIGH
Flag used to select the power mode offering the highest possible energy saving.

AVIX_POWER_REDUCTION_LOW
Flag used to select the power mode offering basic energy saving.

AVIX_POWER_REDUCTION_NONE
Flag used to deselect a power mode. No energy saving mode will be used.

7.6.11 Interrupt Support

Header file to include: AVIX.h
Service specific header file: AVIXGeneric.h

The following functions and definitions are offered:

INTERRUPT FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixDIH_Queue	276	✓	-	✓	✓

INTERRUPT DEFINITIONS	
Definition	**Description**
AvixDeclareISR	Declare an ISR using the AVIX system stack
AvixDeclareISRShadow	Declare an ISR using the AVIX system stack and shadow reg.

avixDIH_Queue

```
void avixDIH_Queue
(
    tavixDIH dih,
    void*    arg
);
```

avixMain[35]	✓
Thread	-
DIH[35]	✓
ISR	✓

>> function overview

Description

Place a pointer to a DIH function in the DIH queue. The DIH queue is a FIFO queue and the newly enqueued DIH is placed at the end of the queue. AVIX starts executing functions from the DIH queue as soon as the scheduler is in control. *The primary usage of this* function *is to be called from an ISR since this enables an ISR to communicate with threads by means of the functionality of the DIH.*[35]

Parameters and return value

Parameter	Description
dih	Pointer to DIH function to enqueue
arg	Value that will be passed to the DIH as its parameter when it is activated by the scheduler. For passing a type safe parameter, use can be made of macro `AVIX_TYPESAFE_TO_VOID`. In the DIH this can be converted back to the correct type with `AVIX_TYPESAFE_FROM_VOID`.
Return value	<none>

[35] This function is allowed to be called from avixMain and DIH's also. When called from `avixMain`, the DIH will be called as soon as `avixMain` returns since this is the moment the scheduler takes control. \When called from a DIH, it will be called as soon as the current DIH returns provided it is the only DIH in the queue.

Interrupt related definitions

The following interrupt related definitions are provided: >> definition overview

`avixDeclareISR`
This macro declares an ISR that will use the AVIX system stack. As a result the ISR will not (or hardly) use any space of the interrupted thread. The macro must be followed by 'C' style curly brackets {}, in between which the code of the ISR is present. Effectively this defines a 'C' style ISR function. *Although this macro is present for all supported hardware platforms, its parameters and implementation is hardware platform specific. For a detailed hardware platform specific description please consult the hardware platform specific Port Guide.*

`avixDeclareISRShadow`
This macro declares an ISR that will use the AVIX system stack. As a result the ISR will not (or hardly) use any space of the interrupted thread. The macro must be followed by 'C' style curly brackets {}, in between which the code of the ISR is present. Effectively this defines a 'C' style ISR function. *Although this macro is present for all supported hardware platforms, its parameters and implementation is hardware platform specific. For a detailed hardware platform specific description please consult the hardware platform specific Port Guide.*

7.6.12 Error Support

Header file to include: AVIX.h
Service specific header file: AVIXError.h

The following functions and definitions are offered:

DIAGNOSTIC FUNCTIONS					
Function	**page**	**avixMain**	**Thread**	**DIH**	**ISR**
avixError_SetHandler	279	✓	✓	✓	✓
avixError_Throw	280	✓	✓	✓	✓

DIAGNOSTIC DEFINITIONS	
Definition	**Description**
AVIX_ASSERT	Check a condition and throw error if false. (conditional macro)
AVIX_ASSERT_ALWAYS	Check a condition and throw error if false.

avixError_SetHandler

```
void avixError_SetHandler
(
    tavixErrorFunc pErrorFunc
);
```

avixMain	✓
Thread[36]	✓
DIH[36]	✓
ISR[36]	✓

>> function overview

Description

Install a user specific error handler. This error handler should never return and is meant to give the user access to the error code.

Parameters and return value

Parameter	Description
pErrorFunc	Pointer to the user supplied error function.
Return value	<none>

[36] Although this function will operate correctly regardless the active entity it is called from, typically it should be called in the initialization part of the application formed by the content of avixMain so from the beginning of the application all occurring errors are caught by the user defined handler.

avixError_Throw

```void avixError_Throw (tavixErrorCode userErrorCode);```	**avixMain** ✓ **Thread** ✓ **DIH** ✓ **ISR** ✓

>> function overview

Description

Throw a user specific error. Calling this function leads to a call to the AVIX central error handling mechanism with the user supplied error code.

AVIX defines a number of error codes for internal use. In order for the user error codes and the AVIX internal error codes not to overlap, user error codes should always have a numeric value of 10000 and higher. A symbol is defined (`AVIXE_USER_ERROR_BASE`) that can be used as the base value for specific user error codes.

User error codes should be specified in the following way:

```
#define USER_ERROR_1     (AVIXE_USER_ERROR_BASE + 0)
#define USER_ERROR_2     (AVIXE_USER_ERROR_BASE + 1)
etc.
```

Parameters and return value

Parameter	Description
userErrorCode	User specified number identifying the error to report.
Return value	<none>

Error Support related definitions

The following power mode related definitions are provided: >> definition overview

`AVIX_ASSERT(condition, errorCode)`
Macro checking parameter `condition`. When this parameter evaluates to a value unequal 0 (true), the macro does nothing. When this parameter evaluates to a value equal 0 (false), the central error handler is called with parameter `errorCode`. The above functionality is only present when building the application with symbol `AVIX_DEBUG` being defined. When building the application without this symbol being defined, macro `AVIX_ASSERT` results in no code being generated.

`AVIX_ASSERT_ALWAYS(condition, errorCode)`
Macro checking parameter `condition`. When this parameter evaluates to a value unequal 0 (true), the macro does nothing. When this parameter evaluates to a value equal 0 (false), the central error handler is called with parameter `errorCode`. The above functionality is always present regardless symbol `AVIX_DEBUG` being defined or not.

7.6.13 Object Support

Header file to include: AVIX.h
Service specific header file: AVIXObjectManager.h

The following definitions are offered:

MISCELLANEOUS DEFINITIONS	
Definition	**Description**
AVIX_OBJECT_ID_DEFINE	Declare a kernel object id guaranteed to be invalid
AVIX_OBJECT_ID_VALID	Test for a kernel object id to be valid
AVIX_OBJECT_ID_NULL	Returns a 'NULL' object id
AVIX_TYPESAFE_EQ	Compare two typesafe variables on equality
AVIX_TYPESAFE_FROM_VOID	Convert void* to typesafe variable
AVIX_TYPESAFE_NEQ	Compare two typesafe variables on inequality
AVIX_TYPESAFE_TO_VOID	Convert typesafe variable or to void*

Miscellaneous definitions

The following miscellaneous definitions are provided: >> definition overview

`AVIX_OBJECT_ID_DEFINE(t,v)`
Declare a kernel object id guaranteed to be invalid. This is especially useful when sharing kernel objects through global variables to allow code using the variable to test for the kernel object id variable to be valid. Parameter ‘t’ is the kernel object type and parameter ‘v’ is the name of the variable. Below a sample of the usage is shown: `AVIX_OBJECT_ID_DEFINE(tavixThreadId, threadId);` The above declaration does the same as a plain declaration with the added benefit the kernel object id is given a value designating it to be invalid.

`AVIX_OBJECT_ID_VALID(id)`
Test for a kernel object id to be valid. When the kernel object id is valid, this macro returns a value unequal zero (‘true’). When the kernel object id is invalid, this macro returns a value equal zero (‘false’).

`AVIX_OBJECT_ID_NULL(t)`
Return an invalid object id of the specified type. This macro can be used for user functions receiving or returning kernel object id’s. When using the value of this macro, this can be used to check for the kernel object id to be valid.

`AVIX_TYPESAFE_EQ`

Compare two type safe variables for being equal. AVIX type safe kernel object id’s cannot be compared for being equal with the basic ‘C’ operator == since the type of these variables is based on structs.

For example, when two kernel object id’s must be compared for being equal, they can be passed to this macro like

```
tavixThreadId threadId1;
tavixThreadId threadId2;
...
if (AVIX_TYPESAFE_EQ(threadId1, threadId2))
{
    ...
}
```

When the id’s are equal (reference the same kernel object), this macro returns a value unequal zero (‘true’). When the id’s are not equal this macro returns a value equal zero (‘false’).

AVIX_TYPESAFE_FROM_VOID
Convert a `void*` to a type safe variable. This macro is used in places where type safe values are passed using an inherent not type safe mechanism as provided by the 'C' language. Examples are the parameter to function `avixDIH_Queue` and the start parameter for a thread function. Both are typed as `void*`. Inside the DIH or the thread function, the `void*` parameter is converted back to the correct type safe variable using this macro.

AVIX_TYPESAFE_NEQ

Compare two type safe variables for not being equal. AVIX type safe kernel object id's cannot be compared for being unequal with the basic 'C' operator `!=` since the type of these variables is based on structs.
 	
For example, when two kernel object id's must be compared for not being equal, they can be passed to this macro like
 	
`tavixThreadId threadId1;`
`tavixThreadId threadId2;`
`...`
`if (AVIX_TYPESAFE_NEQ(threadId1, threadId2))`
`{`
`    ...`
`}`
When the id's are not equal (not reference the same kernel object), this macro returns a value unequal zero ('true'). When the id's are equal this macro returns a value equal zero ('false').

AVIX_TYPESAFE_TO_VOID
Convert a type safe variable to `void*`. This macro is used in places where type safe values are passed using an inherent not type safe mechanism as provided by the 'C' language. Examples are the parameter to function `avixDIH_Queue` and the start parameter for a thread function. Both are typed as `void*`. When passing a type safe value or variable in these situations, the value or variable is converted to a `void*` using this macro.