



NON-PREEMPTIVE EVENT-DRIVEN
OPERATING SYSTEM FOR 8/16-BIT MCU

AdxFSM

Reference Manual

Rev. 0.2

Advantex LLC

January 7, 2025

111402, Moscow,
Ketcherskaya st. 7, b. 6,
tel. +7 (495) 721-47-74, 728-08-03
info@advantex.ru
<http://advantex.ru>

Document Versions

Version	Date	Description
0.1	December 24, 2024	Preliminary version. Against AdxFSM version 0.1.324
0.2	January 07, 2025	Against AdxFSM version 0.1.347 Removed FSMW_t class. It is replaced by more flexible and clear TimerEvent_t class derived from Event_t. Revised header dependency. Fixed critical error in ProcessSystemStatus(), adx_SystemStatus.cpp.

Contents

1	Introduction	10
1.1	Licensing	10
1.2	Versioning and Distribution	10
1.3	Supported Devices and Toolchain	10
1.4	Source Code Organization	10
1.5	Header Files, Namespaces and Include Paths	11
1.6	Configuration Files and Symbols	12
1.7	Building AdxFSM	13
1.7.1	Setting Up Atmel Studio Project	13
1.7.2	Build Command-line Options	16
1.7.3	Setting Up SVN Properties	19
2	Main Concept	22
3	Under the Hood of AdxFSM	27
3.1	Kernel	27
3.2	Task Wrapper (FSM_t)	35
3.3	Signals (Signal_t)	38
3.4	Base Queue (SystemQueue_t)	42
3.5	Events (Event_t)	44
3.6	Error Handling (SystemStatus_t)	51
3.7	Scheduler (Main Loop)	54
3.8	Built-in Timer Events (TimerEvent_t)	57
3.9	Fast Data Communication (DataSignal_t and DataEvent_t)	61
3.10	FIFO With Notification (DataQueue_t)	61
3.11	Recursive Mutex (RecursiveMutex_t)	72
3.12	Rx/Tx Driver Model	72
4	AdxFSM API	76
4.1	core/adx_SystemStatus.h	76
4.1.1	Types and Aliases	76
	SystemStatus_t	76
	ErrorHandler_t	76
4.1.2	Functions	76
	Error()	76

	Warning()	77
	Exit()	77
	ProcessSystemStatus()	78
	CRC16_Modbus()	78
4.2	core/adx_fsm.h	78
4.2.1	Types and Aliases	78
	ptrSignal_t	78
	SignalCounter_t	78
	ptrFSM_t	78
	Task_t	79
4.2.2	FSM_t class methods	79
	Ctor	79
	Start()	79
	GetFsmID()	80
4.2.3	Functions	80
	GetFsmSignal()	80
	GetFsmCounter()	80
	GetRunningFsm()	80
	Wait()	81
4.3	core/adx_Signal.h	81
4.3.1	Types and Aliases	81
	SignalMode_t	81
4.3.2	Signal_t class methods	81
	Ctor	81
	Send()	82
	operator()	82
	SetMode()	82
	GetMode()	83
	GetFsm()	83
	SetErrorHandler()	83
4.3.3	Functions	83
	WaitFor()	83
4.4	core/adx_SystemQueue.h	84
4.4.1	SystemQueue_t<> class template	84
4.4.2	SystemQueue_t<> class methods	84
	IsNotFull()	84
	IsNotEmpty()	84
	Put()	84
	PutE()	85
	Get()	85
4.5	core/adx_Event.h	85
4.5.1	Types and Aliases	85
	EventPrty_t	85
4.5.2	Event_t class methods	86
	Ctor	86
	Send()	86
	operator()	86
	SetMode()	87
	SetPriority()	87
	GetPriority()	87

4.5.3	Functions	87
	ProcessNextEventInQue()	87
4.6	core/adx_TimerEvent.h	88
4.6.1	TimerEvent_t class methods	88
	Ctor	88
	SetMode()	88
	SetPeriod()	88
4.6.2	Functions	89
	SleepFor()	89
4.7	core/adxDataQueue.h	89
4.7.1	DataQueue_t<> class template	89
4.7.2	DataQueue_t<> class methods	90
	IsNotFull()	90
	IsNotEmpty()	90
	Put()	90
	Write()	91
	Get()	91
	Read()	91
	ConnectPutsideFunctor()	92
	GetPutsidePtrFunctor()	92
	ConnectGetsideFunctor()	92
	GetGetsidePtrFunctor()	93
4.7.3	Functions	93
	Put_bl()	93
	Write_bl()	94
	Get_bl()	94
	Read_bl()	94
4.8	core/adx_RecursiveMutex.h	95
4.8.1	RecursiveMutex_t class methods	95
	TryLock()	95
	Lock_bl()	95
	Unlock()	95
4.9	core/adx_DataEvent.h and core/adx_DataSignal.h	96
4.9.1	SaE_Data_t<> class template	96
4.9.2	SaE_Data_t class methods	96
	SetData()	96
	GetData()	96
4.9.3	DataSignal_t and DataEvent_t classes	96
5	Use Cases and Examples	97
6	Time Profiling	100
7	Troubleshooting	100
7.1	Compile-time Errors	100
7.2	Run-time Critical Errors	100
7.2.1	Stack Overflow	100
7.2.2	Signal Loop: scheduler can't get control	100
7.2.3	Function Call within Wrong Context	100
7.3	System Status Handling	100

List of Figures

1	Atmel Studio 7. Device selection dialog while creating new project	14
2	Atmel Studio 7. Include directories	15
3	Atmel Studio 7. Miscellaneous settings	15
4	Atmel Studio 7. Advanced settings	16
5	Atmel Studio 7. Tools settings	17
6	Atmel Studio 7. Adding source files to the project	18
7	Atmel Studio 7. Set as StartUp Project	19
8	Import SVN Ignore List property for Atmel Studio Solution	20
9	Apply property recursively	20
10	SVN Externals settings	21
11	SVN Keywords, Id	22
12	Abstract FSM Model	23
13	FSM states and transitions	24
14	Nested FSM	24
15	AdxFSM model. User application logic, represented as a set of FSMs	25
16	Behavior of the Event depending on its Mode	26
17	Header dependencies, types, functions and system objects	28
18	Class inheritance	29
19	struct Context_t layout	29
20	Kernel objects in SRAM	29
21	Task stack and Context initialization	30
22	Context switching implementation	31
23	CriticalSection_t class	33
24	Context switching application example	34
25	FSM_t constructor	35
26	State of FSM_t and kernel objects after last FSM_t object ctor execution	37
27	FSM_t::Start()	37
28	FSM_t::operator() and Wait()	38
29	FSM_t and kernel objects state after after main() call fsm_0(), Task_0 call fsm_1()	39
30	FSM_t and kernel objects state after recursive Wait() call (i.e. return to original caller)	39
31	Example of Signal use as an acknowledge	40
32	Signal states and transitions	42
33	Signal_t::Send() method logic	43
34	Signal_t::SetMode() method logic	43
35	WaitFor() can be used for waiting one known signal	44
36	SystemQueue_t<SIZE,T> template class	44
37	~Event_t() destructor	46
38	Signal states and transitions	47
39	Event_t::Send() method logic	48
40	Event_t::SetMode() method logic	49
41	Event_t data relationship	50
42	SystemStatus functions	51
43	ProcessSystemStatus()	51
44	Main loop	54

45	ProcessNextEventInQueue() Scheduler logic	56
46	TimerEvent_t constructor logic	58
47	Timer Event states and transitions diagram	58
48	Hardware TC ISR logic	60
49	DataQueue Connect put/get side notification methods	65
50	DataQueue Full/Empty criteria	65
51	DataQueue_t::Put() logic	66
52	DataQueue_t::Write() logic	67
53	DataQueue_t::Get() logic	68
54	DataQueue_t::Read() logic	69
55	DataQueue_t<> Block Diagram	70
56	RecursiveMutex_t states and transitions diagram	73
57	AdxFSM high-level Rx/Tx Driver Model block diagram	75
58	Driver Model ISR related logic	75
59	Driver Model Open/Close methods	76

List of Tables

1	Kernel summary (kernel/adx_kernel.h)	34
2	FSM_t and related functions summary (core/adx_fsm.h, adx_fsm::) 36	
3	Signal_t and related functions summary (core/adx_Signal.h, adx_fsm::) 41	
4	Event_t and related functions summary (core/adx_Event.h, adx_fsm::) 46	
5	SystemStatus summary (core/adx_SystemStatus.h, adx_fsm::) . 52	
6	TimerEvent_t and related functions summary (core/adx_TimerEvent.h, adx_fsm::) 59	
7	DataQueue_t<> and related functions summary (core/adx_DataQueue.h, adx_fsm::) 62	
8	RecursiveMutex_t and related functions summary (core/adx_RecursiveMutex.h, adx_fsm::) 74	
9	System Error and Warning summary	101

Preface

AdxFSM¹ is non-preemptive event-driven OS for 8/16-bit small embedded systems. It was originally designed as a framework for RF-block control applications in Test&Measurement instruments. Most Advantex T&M instruments consist of several interconnected RF-blocks, power supply system, optional HMI, and MCU-based PCB that controls the instrument low-level hardware, including power supply system. High-level SCPI or binary commands are received via UART from HMI or external PC and processed into the low-level command and data sequence for the RF-block control via several SPI interfaces, one for each RF-block. Although application itself is not time-critical, there are some time related requirements that should be met for the instrument to work properly. And, perhaps, the most important thing is reliability, because the instrument should be able to work for years without rebooting.

As a use-case example, combined with RFCCTL-2xM-PCB board, AdxFSM-based control application is able to provide the following features:

- Time-compressed SPI data transfer to minimize RF-block transient (not valid state) duration. It means that all register calculations which changes RF-block state should be done and stored before SPI transaction. All changing state transfers should be done as one packet.
- Optional waiting for RF-blocks to get to a valid state (e.g. PLL and AGC lock) with timeout support.
- Internal trigger events with determinate response time (time passed from internally generated timestamp to peripheral access).
- Lossless external and internal trigger event/timer handling.
- Software arbitrary waveform signal sources for RF-block parametric modulation (phase, frequency, etc.) which meet the following requirements:
 - Mean signal period should be in exact relation with MCU system clock.
 - Signal samples can float in time, but their number should be enough for each signal period and the value of the sample should correspond to its time stamp for correct waveform recovery.
 - Time passed from sample time-stamp to corresponding calculated command and data transfer to RF-block should be determinate and near to constant.
- Parallel execution of commands sent via UART, e.g. “read current frequency value” and “set signal level”.
- Reliable high-level command and data transfer protocol with sync loss self-recovery.
- Power supply system control.
- Efficient MCU resource usage.

¹Advantex Finite State Machine

1 Introduction

1.1 Licensing

Copyright 2024 Advantex LLC.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

1.2 Versioning and Distribution

Versioning scheme: X.Y.R, where X – major number, starts with 0 at working stage when all interfaces can be changed, 1 – first release, incremented when not backward compatible kernel or core changes are made; Y – minor number, incremented when kernel or core backward compatible changes are made; R – svn src/ directory last changed revision number.

Current work version of AdxFSM is under SVN rep: .../evb/RFCTL-2X/Soft/adx_avr.

Stable versions are available on <https://github.com/>.

1.3 Supported Devices and Toolchain

AdxFSM is tested on ATxmega256A3U device.

Supported devices: ATxmega256A3U / ATxmega192A3U / ATxmega128A3U / ATxmega64A3U. Kernel and core code may also work properly on other xmega devices, but not sure for peripheral hardware drivers. Other platforms require port of Kernel code and TimerEvent_t class since it uses hardware timer/counter. Also some attention should be made to the alignment, padding and data packing when porting on 16 or 32-bit platforms.

Compiler/Linker: avr-g++, (AVR 8 bit GNU Toolchain 3.6.2 1778) 5.4.0;

Language version: C++14 (-std=c++14);

Assembler: avr-gcc;

Debugger: avr-gdb;

Library: compiled and linked against library pack XMEGAA_DFP 1.1.68

1.4 Source Code Organization

AdxFSM root directory contains the following sub-directories and files:

docs/ – contains this Reference Manual, SVN Ignore List file (AtmelStudioSvnIgnoreList.svnprops), and other documents and project settings with its source files;

examples/ – contains Atmel Studio Solutions with example and test projects;

src/ – contains sub-directories listed below with source files:

src/config/ – configuration header files;

src/core/ – base features header and source files;

src/drivers/ – driver source files;

src/kernel/ – kernel source files.

All source files are to be compiled with user project files because AdxFSM is small enough, while this approach provides more more efficient code optimization.

Minimum files that should be added to the project:

src/kernel: adx_kernel.cpp, adx_kernel_SwitchContext.S;

src/core: adx_fsm.cpp, adx_Signal.cpp, adx_Event.cpp, adx_SystemStatus.cpp.

Optional files:

src/core/adx_TimerEvent.cpp – add if required software time counter and built-in timeout features.

src/core/adx_SystemClock.cpp – add if required ready-to-use clock configuration;

driver/sdx_*Driver.cpp – add if required ready-to-use corresponding peripheral driver.

1.5 Header Files, Namespaces and Include Paths

All header files are located in the same directory as the corresponding .cpp file. Search “include” path for compiler (-I option) should be set to src/ directory located in AdxFSM root. All AdxFSM source files use header names relative to src/ directory, like this:

```
#include "core/adx_Event.h"
```

, thus one include path to src/ is enough for all source files located in different sub-directories.

Configuration headers are referenced within AdxFSM source files in similar way:

```
#include "config/adx_core_config.h"
```

If you need your own configuration settings, just create config/ directory somewhere in your project, and copy required configuration header from AdxFSM/src/config to your config/ directory within project. Then add search “include” path (-I option) to the directory containing your config/ before previous search path. This ensures that preprocessor will try to find your local configuration file first, and if it’s not found then it will try to find the file in AdxFSM/src/config/ directory. Local configuration header file can be modified as you wish.

AdxFSM declares two namespaces:

adx_kernel – is used as internal namespace for core design;

adx_fsm – is user API namespace, all AdxFSM interface types, functions and system object names reside here.

1.6 Configuration Files and Symbols

Config/ directory contains following configuration files:

adx_kernel_config.h – defines main() stack size, and its bottom address;

adx_core_config.h – defines buffer sizes for various core objects (like event queue, mutex), some scheduler settings, and hardware timer/counter settings used for built-in software timer/counter support;

adx_driver_config.h – defines hardware and software settings for various driver objects.

Some hardware configuration is implemented using preprocessor #if defined() directive. This makes it possible to setup some configuration in two ways:

1. by modifying local configuration file directly (i.e. by uncommenting appropriate #define directive, e.g. // #define ADX_SYSTEM_TIMER_TCD1 // default is TCF0 if not defined).
2. by adding preprocessor symbol (e.g. ADX_SYSTEM_TIMER_TCD1 from the example above) using compiler -D option (Atmel Studio – Project Options ▸ Toolchain ▸ AVR/GNU C++ Compiler ▸ Symbols). This way there is no need to modify this setting in configuration header file.

This approach is widely used in driver configuration file.

Most value definitions use const keyword (not preprocessor #define directive) it allows to find type conversion errors at compile-time. It is possible to combine both methods for value definitions as well, but not done in current version.

Kernel and core config files are listed below.

```

// $Id: adx_kernel_config.h 294 2024-11-28 16:35:06Z apolv $
/* Configuration for kernel/xxx source files */
#if !defined(KERNEL_ADX_KERNEL_CONFIG_H_)
#define KERNEL_ADX_KERNEL_CONFIG_H_

#include <stdint.h>

namespace adx_kernel
{
    const uint16_t SystemMainStack_SIZE = 256U;
    uint8_t* const ptrSystemMainStackBottom = (uint8_t*)0x5fff;
}
#endif /* KERNEL_ADX_KERNEL_CONFIG_H_ */

```

In most cases compiled with -Os scheduler's main loop stack occupies less than 48 bytes. Actual required stack space depends on user application, and can be evaluated using debugger.

Main's stack bottom pointer is defined as a literal, although it can be replaced by device specific RAMEND macros defined in avr/io.h.

```

// $Id: adx_core_config.h 347 2025-01-06 22:53:39Z apolv $
/* Configuration for core/xxx source files */
#ifndef !defined (CONFIG_ADX_CORE_CONFIG_H_)
#define CONFIG_ADX_CORE_CONFIG_H_

#include <stdint.h>

namespace adx_fsm
{
    // Event System Config:
    const uint8_t LowPrioritySystemEventQueue_SIZE = 4;
    const uint8_t MiddlePrioritySystemEventQueue_SIZE = 8;
    const uint8_t HighPrioritySystemEventQueue_SIZE = 4;

    const uint8_t MaxNumberOfSuccessiveLowPrtyCalls = 4;
    const uint8_t MaxNumberOfSuccessiveMiddlePrtyCalls = 8;
    const uint8_t MaxNumberOfSuccessiveHighPrtyCalls = 16;

    // Mutex Config:
    const uint8_t RecursiveMutexSignalQueue_SIZE = 4;

    // TimerEvent t system timer Config:
    const uint8_t SystemTimerEventBuffer_SIZE = 8;
    // Allowed number of Tasks with built-in software timer/counter

    // #define ADX_SYSTEM_TIMER_TCxx // default is TCF0 if not
    // defined
    // uncomment and replace xx with C0,C1,D0,D1,E0,E1 for another
    // timer
    // #define ADX_SYSTEM_TIMER_INT_xxx // default is middle
    // interrupt priority level
    // uncomment and replace xxx with HIGH or LOW for another
    // priority level
    // System timer Prescaler:
    // x1 = 0x01,
    // div1 = 0x02,
    // div4 = 0x03,
    // div8 = 0x04,
    // div64 = 0x05,
    // div256 = 0x06,
    // div1024 = 0x07
    const uint8_t SystemTimerFperPrescaler = 0x06;
    // System timer period:
    // Time resolution, i.e. ISR Period = (Prescaler * (Period + 1))
    // Fper;
    const uint16_t SystemTimerPeriod = 624; // per624, div256 -> 5ms
    // at Fper==32MHz
}
#endif // !CONFIG_ADX_CORE_CONFIG_H_

```

For detailed description of settings see section 3 on page 27.

1.7 Building AdxFSM

1.7.1 Setting Up Atmel Studio Project

To setup new project using Atmel Studio 7 follow the steps below.

1. Create new C++ executable project within existing solution or new solution.

i Solution is a container for one or more projects with shared configuration and startup project settings. Build of all projects in a solution can be done by one click.

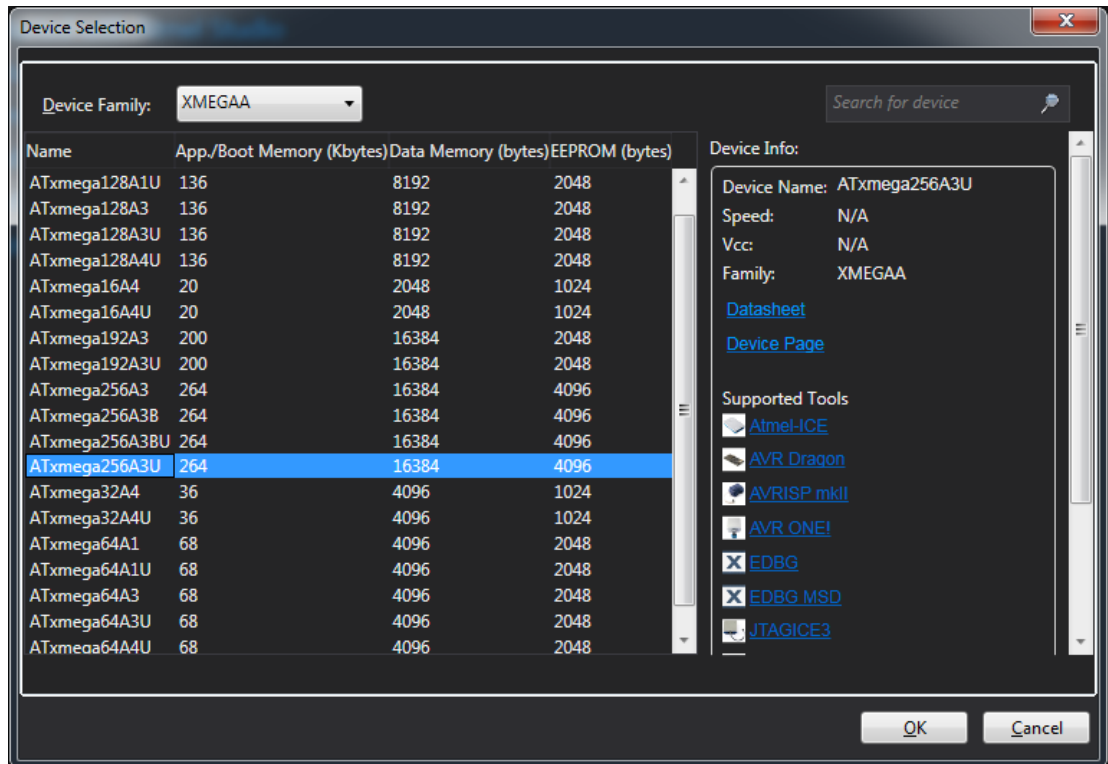


Figure 1: Atmel Studio 7. Device selection dialog while creating new project

2. Select Device (fig. 1):
Device Family: XMEGAA
Device Name: ATxmega256A3U
3. If you need your own configuration of `adx_avr` library then copy `config` folder from `adx_avr\src` to your project's root folder. Then modify any `xxx_config.h` file in your `_project/config` folder as you wish. Note: you can copy to your `_project/config` folder only those `xxx_config.h` files which you need to modify, not all files.
4. Set Up Project Properties.
 - (a) Toolchain->All Configurations->AVR/GNU C++ Compiler->Directories
Add Include paths exactly in the following order (fig. 2):
 - i. Include relative path to your `_project` root folder: `..`
 - ii. Include relative or absolute path to `adx_avr\src` (e.g. `.././.././src` if your `_project` resides in `adx_avr\examples\your_solution` folder).
 - (b) Toolchain->All Configurations->AVR/GNU C++ Compiler->Miscellaneous
Set Other flags field to: `-std=c++14` (fig. 3)
5. Setup Debug Configuration Project Properties(Optional)
 - (a) Toolchain->Debug->AVR/GNU C++ Compiler->Optimization
Set Optimization Level to `(-Og)` or `(-O0)`

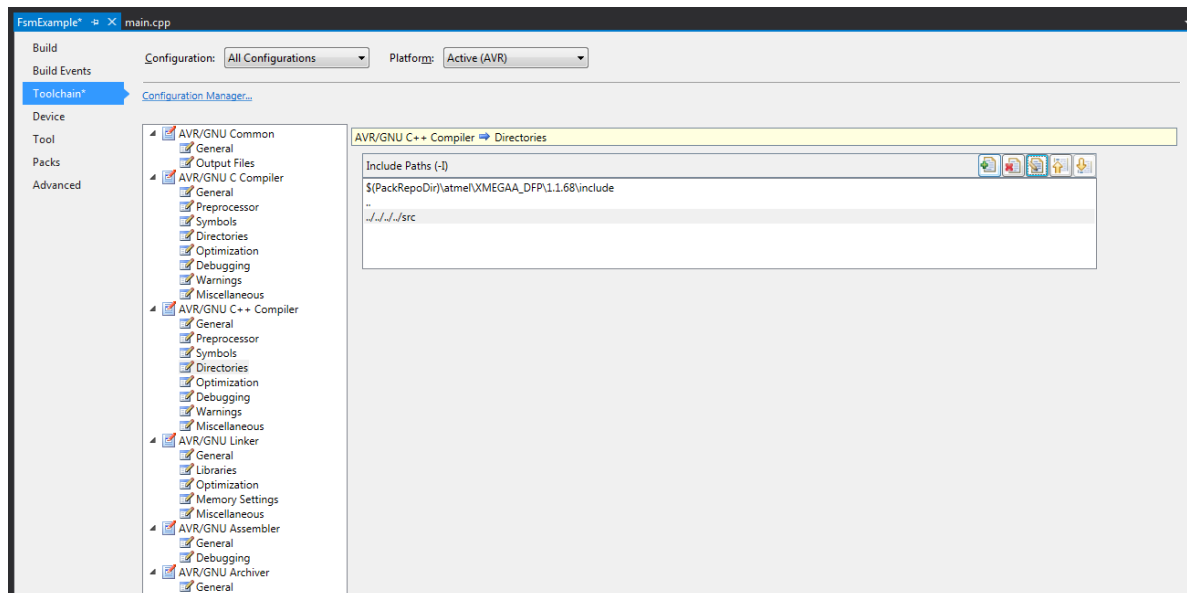


Figure 2: Atmel Studio 7. Include directories

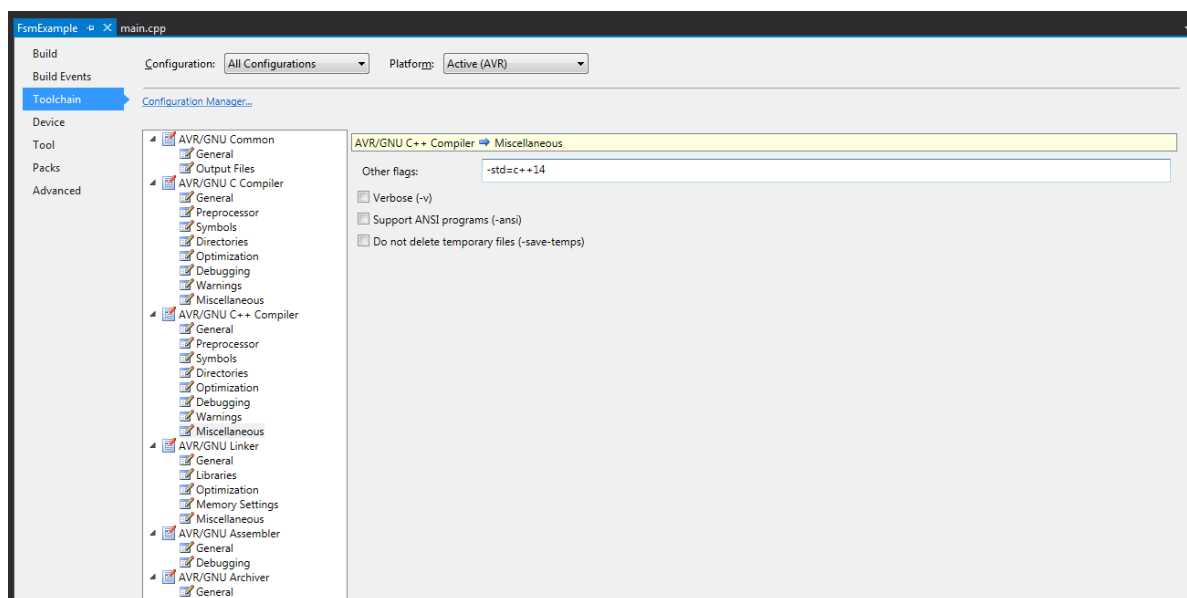


Figure 3: Atmel Studio 7. Miscellaneous settings

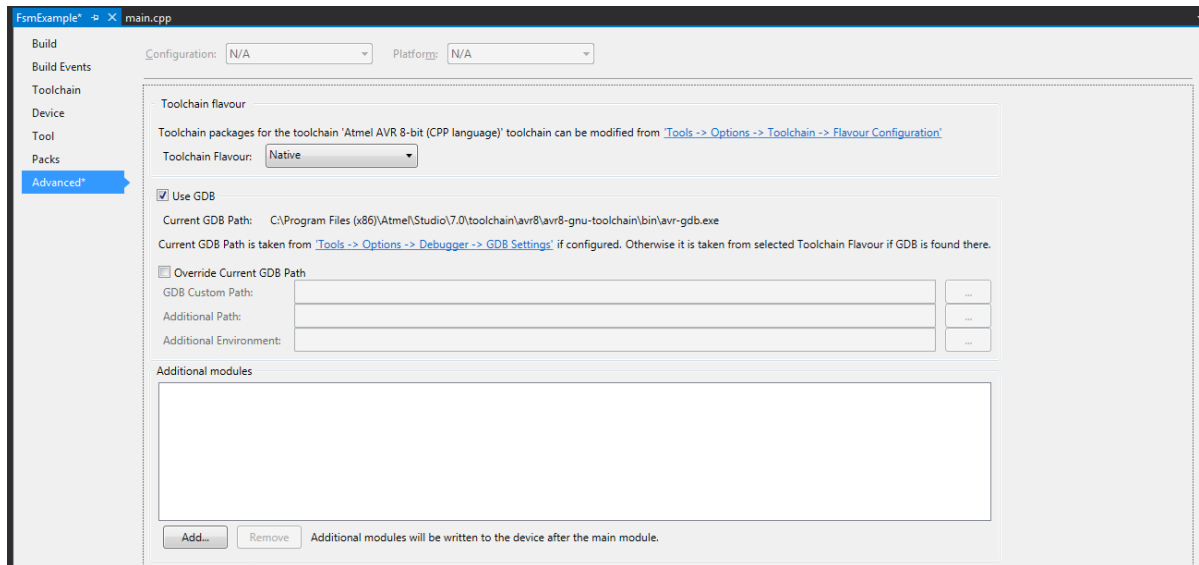


Figure 4: Atmel Studio 7. Advanced settings

- (b) Toolchain->Debug->AVR/GNU C++ Compiler->Debugging
Set Debug Level to Maximum (-g3)
 - (c) Tool->Selected debugger/programmer: Simulator
 - (d) Advanced
Set Use GDB checkbox on (fig. 4)
 - (e) Main Menu->Tools->Options
To enable interrupts while debugging set Tools->Mask interrupts while stepping to False (fig. 5)
6. Add required source files (only *.cpp and *.S, no need to add *.h files) from adx_avr/src to your project as link.
Right-click on your_project in Solution Explorer window to call pop-up menu, select Add->Existing Item (fig. 6). Select "Add As Link" option.
 7. Build your_project
Select Configuration (Debug or Release) in Configuration Manager or Task Bar.
Select Main Menu->Build->Rebuild *your_project*.
 8. Set as StartUp Project
To start debugging/upload select Set as StartUp Project by right-clicking on *your_project* in Solution Explorer (fig. 7).
 9. Run debugger Main Menu->Debug->Start Debugging and Break
 10. Enjoy the result!

1.7.2 Build Command-line Options

Most examples are build in Atmel Studio 7 (v. 7.0.2389) with the following options:

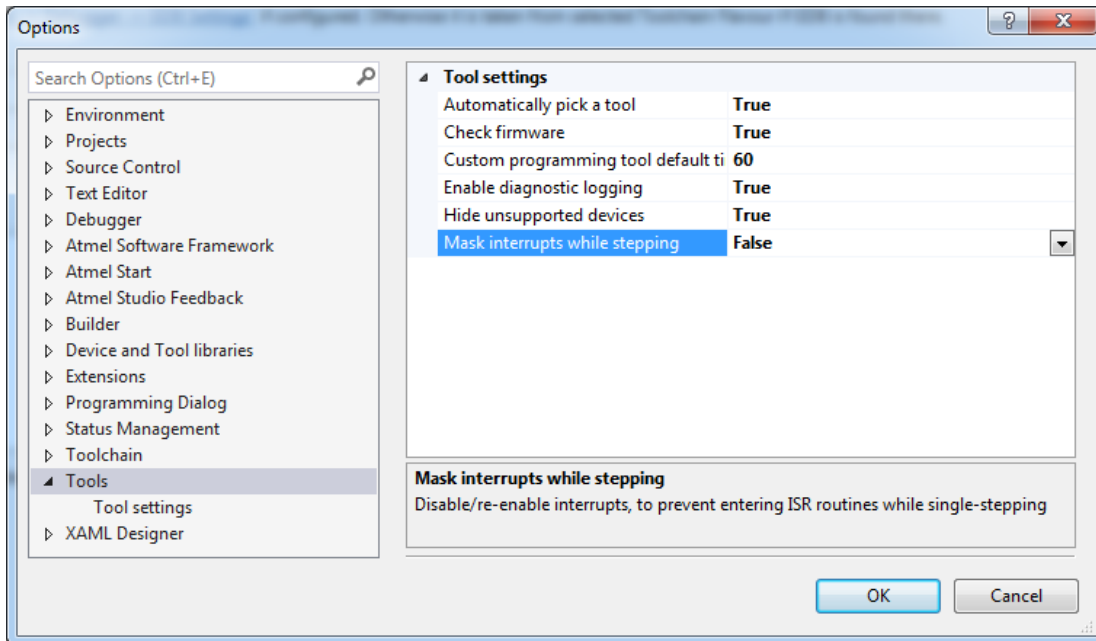


Figure 5: Atmel Studio 7. Tools settings

Release Configuration

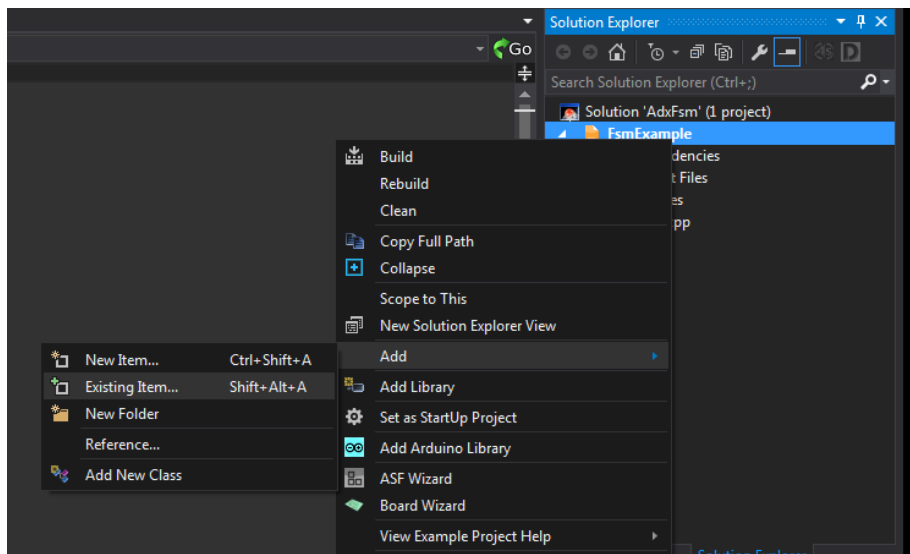
AVR/GNU C++ Compiler: `avr-g++ -funsigned-char -funsigned-bitfields -DNDEBUG -I"${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\include" -I".." -I"../..../src" -Os -ffunction-sections -fdata-sections -fpack-struct -fshort-enums -Wall -mmcu=atxmega256a3u -B "${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u" -c -std=c++14 -MD -MP -MF "$(@:%.o=%.d)" -MT "$(@:%.o=%.d)" -MT "$(@:%.o=%.o)"`

AVR/GNU Linker: `avr-g++ -Wl,-Map="${OutputFileName}.map" -Wl,-start-group -Wl,-lm -Wl,-end-group -Wl,-gc-sections -mmcu=atxmega256a3u -B "${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u"`

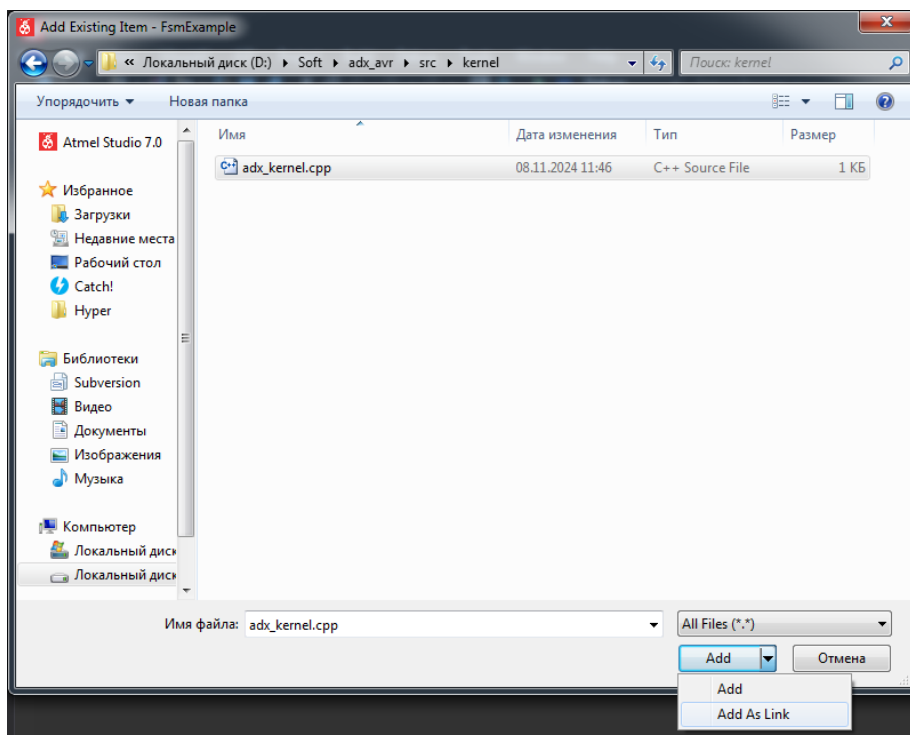
AVR/GNU Assembler: `avr-gcc -Wa,-gdwarf2 -x assembler-with-cpp -c -mmcu=atxmega256a3u -B "${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u" -I "${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\include" -MD -MP -MF "$(@:%.o=%.d)" -MT "$(@:%.o=%.d)" -MT "$(@:%.o=%.o)"`

Debug Configuration

AVR/GNU C++ Compiler: `avr-g++ -funsigned-char -funsigned-bitfields -DDEBUG -I"${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\include" -I".." -I"../..../src" -O0 -ffunction-sections -fdata-sections -fpack-struct -fshort-enums -g3 -Wall -mmcu=atxmega256a3u -B "${PackRepoDir}\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u" -c -std=c++14 -MD -MP -MF "$(@:%.o=%.d)" -MT "$(@:%.o=%.d)" -MT "$(@:%.o=%.o)"`



(a) Add Existing Item



(b) Add As Link

Figure 6: Atmel Studio 7. Adding source files to the project

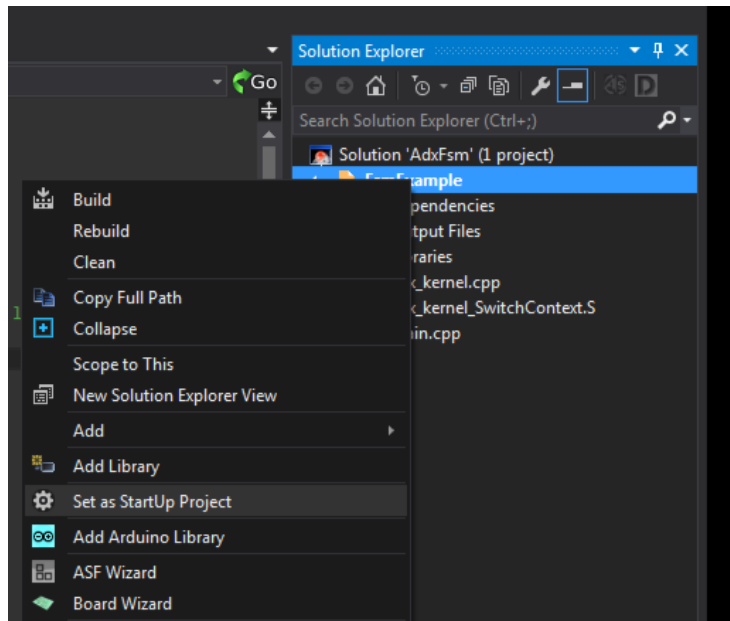


Figure 7: Atmel Studio 7. Set as StartUp Project

AVR/GNU Linker: `avr-g++ -Wl,-Map="$(OutputFileName).map" -Wl,-start-group -Wl,-lm -Wl,--end-group -Wl,--gc-sections -mmcu=atxmega256a3u -B "$(PackRepoDir)\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u"`

AVR/GNU Assembler: `avr-gcc -Wa,-gdwarf2 -x assembler-with-cpp -c -mmcu=atxmega256a3u -B "$(PackRepoDir)\atmel\XMEGAA_DFP\1.1.68\gcc\dev\atxmega256a3u" -I "$(PackRepoDir)\atmel\XMEGAA_DFP\1.1.68\include" -MD -MP -MF "$(@:%.o=%.d)" -MT "$(@:%.o=%.d)" -MT "$(@:%.o=%.o)" -Wa,-g`

For more information about avr-gcc options follow the link:

<https://gcc.gnu.org/onlinedocs/gcc/AVR-Options.html>.

GCC general command-line options are described here:

<https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html>

1.7.3 Setting Up SVN Properties

Add your new Atmel Studio Solution directory to SVN rep, but don't commit. SVN Ignore List should be setup for new Solution before commit! Otherwise temporary files will be put into the rep alongside with source project files.

1. Right-click on your Atmel Studio Solution directory and select from pop-up menu TortoiseSVN > Properties.
2. Press Import... button, and select docs/AtmelStudioSvnIgnoreList.svnprops file (fig. 8).
3. Select svn:ignore property in the property window and press Edit... button. Select Apply property recursively checkbox and press OK (fig. 9). Repeat this step when you add new project to the Solution.

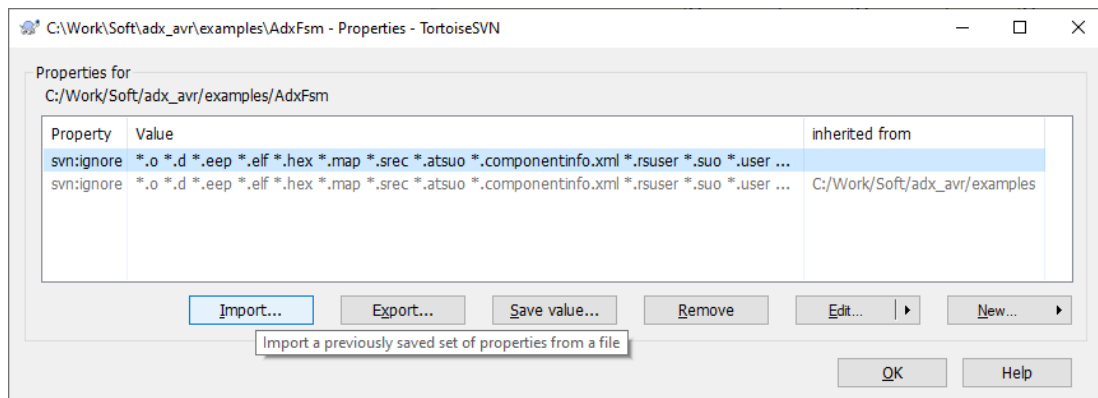


Figure 8: Import SVN Ignore List property for Atmel Studio Solution

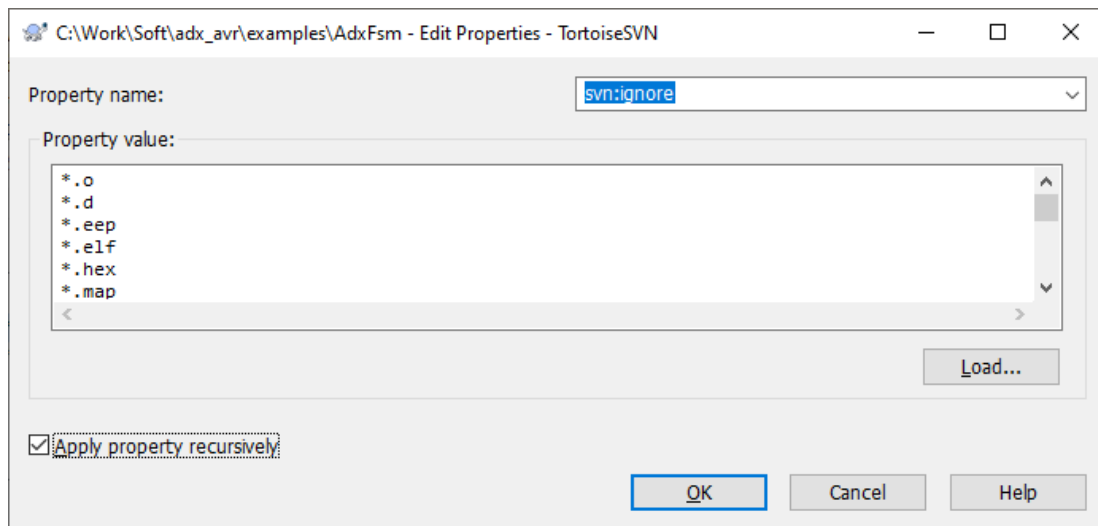


Figure 9: Apply property recursively

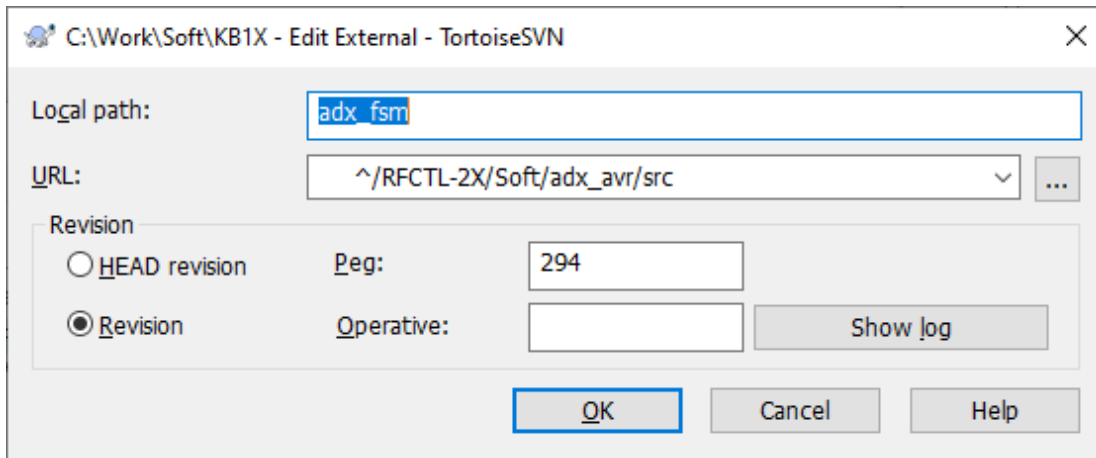


Figure 10: SVN Externals settings

To add AdxFSM src/ content as a link to your project under svn, you can use svn:externals property.

1. Right-click on the Solution directory which is under svn rep, and select from pop-up menu TortoiseSVN > Properties.
2. Click New > Externals.
3. Click New.. button in svn:externals window (fig. 10) .
4. Set local path (directory name in the current project) which will be used for src/ content link. This directory must not be under current svn repository.
5. Set URL path of AdxFSM rep to src/ directory, and set desired revision number or head revision. Then click OK.

Content of the src/ directory will be linked to created directory name in your svn rep.

To include svn revision, author and date info into the file, svn:keywords can be set for your project source files.

1. Type the following somewhere in your source file header:
// \$Id\$
2. Right-click on your .cpp or .h source file, and select from pop-up menu TortoiseSVN > Properties.
3. Press New > Keywords, and select ID checkbox in svn:keywords window (fig. 11)

After commit "\$Id\$" field will be replaced with the string like this:

```
// $Id: MatrixKbKey.cpp 295 2024-11-28 16:44:24Z apolv $
```

with the format: [filename] [file last changed revision] [yyyy-mm-dd] [time in UTC] [user name who made last change]. It will be automatically updated after each commit if there is any modification in the file.

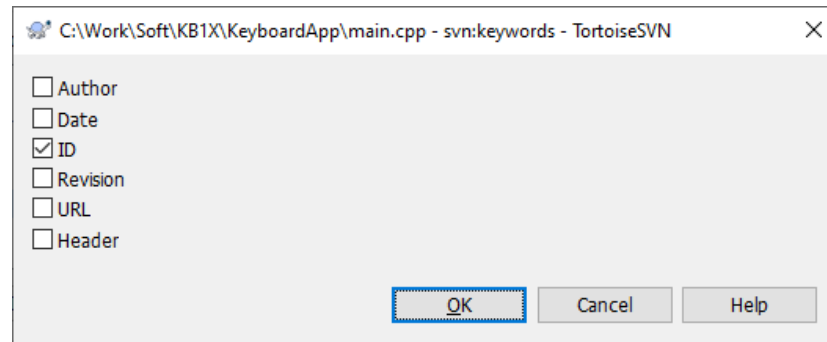


Figure 11: SVN Keywords, Id

2 Main Concept

AdxFSM is designed to simplify user application logic implementation which can be represented in terms of abstract model – as a set of finite state machines (FSMs), fig. 12. Each FSM in this abstract model has one or more states, each state represents waiting process for this FSM, when nothing is executed in this FSM. One of the states is initial state. Each state has **one or more** out transitions to other states. The FSM is activated and the transition is executed when any event assigned to this transition happened. Transition may contain any user application logic (fig. 13), including subroutine call which contains its own waiting states and transitions, fig. 14, i.e. FSMs can be nested. It should be noted that FSM is able to wake up from the waiting state if it waits for particular event, set of events, or any event assigned to the FSM and one of the events that are waited for has happened. Events can be generated by other FSM in user application logic while executing transition or by hardware interrupt while executing ISR². FSMs run and synchronize to each other by means of events, i.e. FSMs won't run (wake up) if no event is generated.

AdxFSM model is similar to the abstract model described above with some restrictions due to the fact that transitions (of different FSMs) can't be executed simultaneously by MPU, and transition execution time is not zero, so several various events can happen while its execution. Thus abstract model shown in figure 12 on the next page transforms to real-life model shown in figure 15 on page 25.

The following entities are used in AdxFSM model:

Tasks – top-level FSMs. Each Task has its own stack and Context.³ Task has one entry and no return.

Events – objects in memory used by Scheduler for Task switching, and also by Task to identify which transition should be executed. Address of the Event object is used as its unique ID to distinguish one event from another. Events can be sent from ISR, main() or Task. Event can be sent to Task only. They serve as the basic Tasks synchronization objects and, that

i Blocking Read/Write operations are the most widely used example of nested FSMs.

! Return from the Task is not supported by current implementation. Do not return from Task, use infinite loop!

²Interrupt Service Routine

³Context is the object in SRAM that stores Task state (MPU registers) before switching to another Task. Registers are loaded from the Context of the next Task.

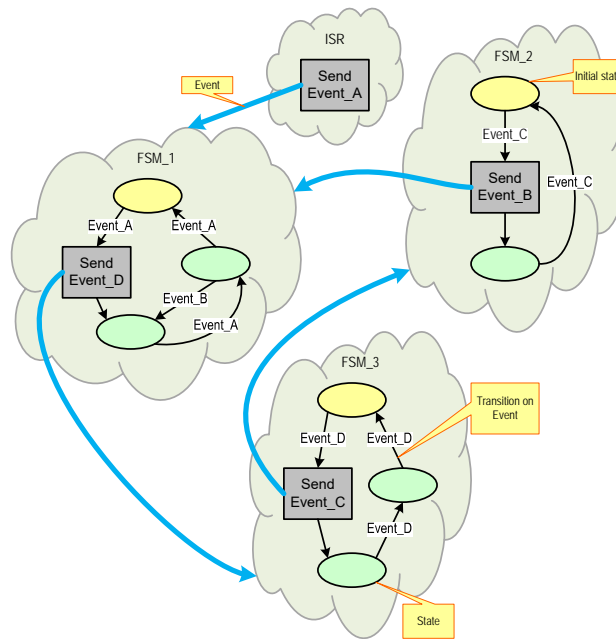


Figure 12: Abstract FSM Model

is most important, – provide deferred ISR handling which makes ISRs as short as possible enabling fast response to external hardware events. When sending, the Event (not Event object itself, but its address) is put into the queue, then Scheduler gets it from the queue, and if Event is in Active mode, Scheduler passes it to the assigned Task and runs the Task. Event object has the following properties:

Assigned Task – pointer to FSM object which is used to run/switch to the Task. Only one Task can be assigned to the Event;

Mode (state) – defines behavior of the Event while its sending and processing by Scheduler. Mode can be Disabled, Silent and Active;

Counter – incremented by one at sending (if it's in Active or Silent mode), and cleared to 0 when successfully processed by the Scheduler (i.e. “received” by the Task). Counter is used to find out how many times the Event has been sent since the last time successful processing by the Scheduler (from the moment the Task was received this Event previously). Counter clear is an atomic operation so no event occurrence will be missed until Event is in Active or Silent mode;

Priority – defines which queue the Event will be put in when sending. There are three priority levels: Low, Middle and High.

Signals – objects in memory similar to the Event but used for direct Task switch without Scheduler (see fig. 15, Signal_B). Signals have no Priority property because they are not put in the queue.

Scheduler – algorithm which runs (switching) Tasks according to the Events got from the queue. Scheduler runs in main()'s Context.

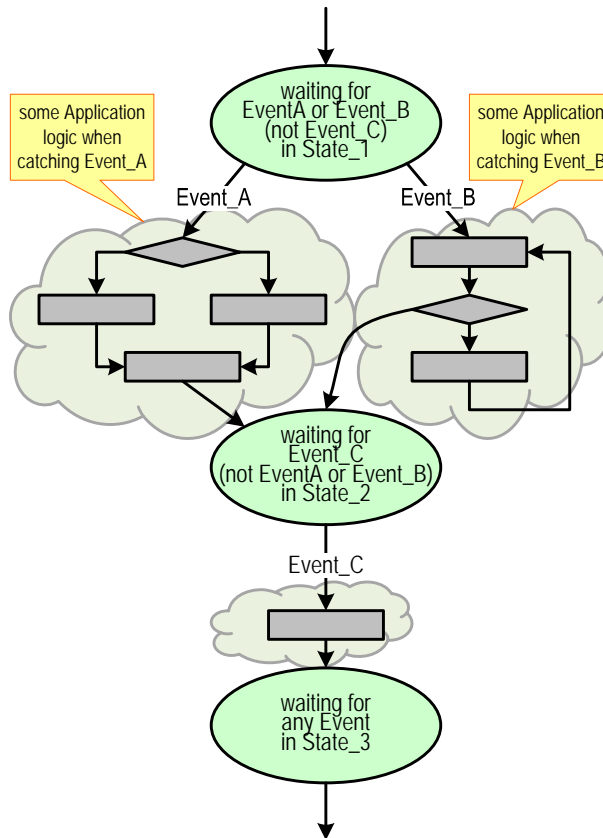


Figure 13: FSM states and transitions

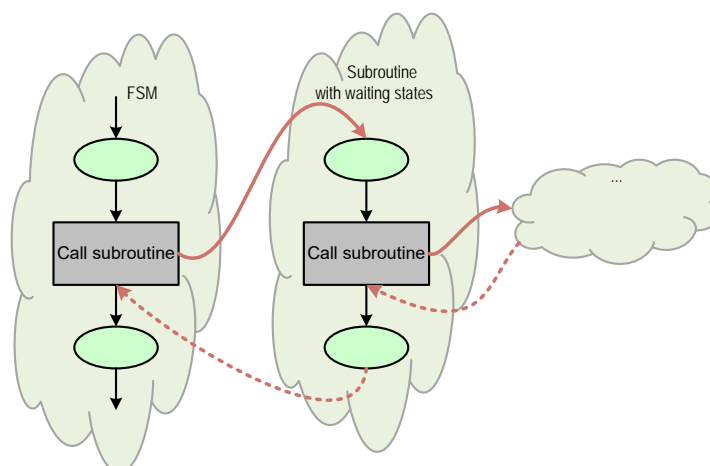


Figure 14: Nested FSM

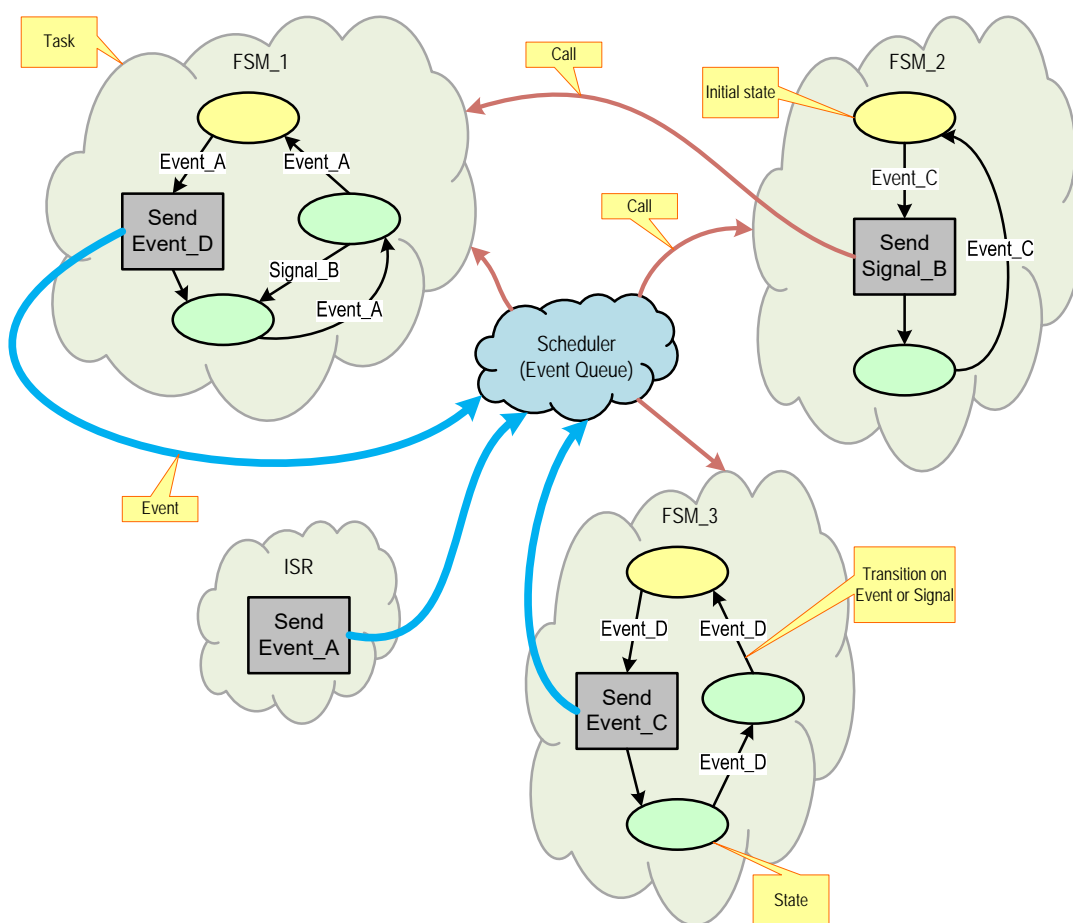


Figure 15: AdxFSM model. User application logic, represented as a set of FSMs

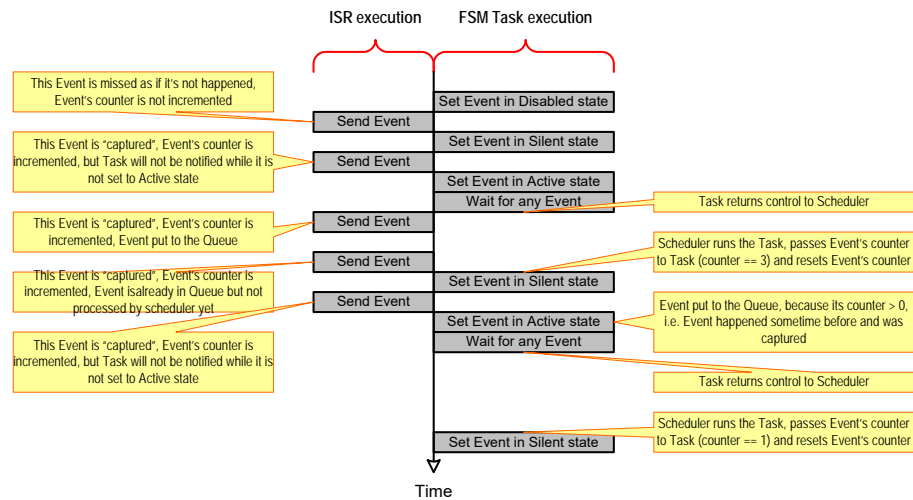


Figure 16: Behavior of the Event depending on its Mode

It should be noted that Scheduler runs the assigned to the Event Task regardless of the state in which the Task is waiting. It doesn't know if the Task is waiting for this particular Event or not. This can result to spur wakeups, when Task is activated at wrong time or condition. In most cases transition condition implies checking another variable, e.g. buffer full/empty check, so error will not occur in this case, but not the waited Event will be processed, and when it will be the time to wait for this particular Event, it may not be generated again. Any way, at least one occurrence of the Event will be lost, it may corrupt user logic execution.

To avoid this failures the Event Mode property is used – just set the Events which are waited for to Active Mode before waiting, and set them to Silent or Disabled Mode after the waiting. This way other (not waited) Events will not wake up Task even if they are already happened. At initial state all Events should be in Disabled or Silent Mode. If waiting is within a loop (like in read/write operations), there is no need to change the Mode in the loop, it's enough to make Active before the loop and Disable/Silent after the loop. Figure 16 shows the behavior of the Event depending on its Mode.

This mechanism is already built in most user API functions when logic waits for one Event, but when dealing with multiple Events and complex wait conditions, e.g. waiting for any one in the Event list, or any number in the list, it should be handled on the user side, because it implies too many variants for efficient implementation in API.

Another question – is what Mode should be used after waiting. If Events must be “captured” continuously, without loss of any occurrence, then Silent Mode should be used. If Events must be “captured” only starting some time moment and all occurrences which may happen before this moment must be missed, then Disabled Mode should be used until the moment when you need to begin the “capture”. Events can be captured only from the moment when you set the Event from Disabled to Silent or Active Mode. Task can be waked up only from the moment when you set the Event from Disabled or Silent Mode to

Active Mode. In Disabled Mode nothing happens with neither Event nor Task. Mode can be set from `main()`, ISR or from any Task, it's not necessary to be the Event's Task.

Tasks (Contexts), Events and Signals are supposed to be static memory objects (allocated in `.data` or `.bss` section), i.e. no memory management is required for kernel or core execution. Although Signals and Events can be local variables in Tasks (because there is no return from Task and the objects will never be destroyed), and even in functions (dtor which deletes the Event from the queue is implemented), but you must be sure that no object references on the Event object after function return, when the it is destroyed.

Next section describes in detail how the model works.



Don't create local Event objects without reason!

3 Under the Hood of AdxFSM

AdxFSM header dependencies are shown in figure 17. Below header name (filled in green) there are types, system object names and functions are listed in groups. Some of them, filled in red, are in `adx_kernel` namespace, and used by core logic. Other, filled in gray, are in `adx_fsm` namespace, and represent user API interface. This dependency diagram helps to find out which source and header files should be added to the project if restricted functionality is going to be used in application.

Figure 18 shows class inheritance. Template class `SaE_Data_t` is container, that is inherited by `Signal_t` and `Event_t` classes to send data with Signal or Event. It worth to be noted that `DataEvent_t` is derived from `SaE_Data_t` and `Event_t` to avoid virtual inheritance. This restricts some functionality of the `DataEvent_t` class, but saves significant SRAM space. If this feature is really required, `DataEvent_t` can be inherited as shown by gray dash line.⁴

3.1 Kernel

Kernel is responsible for Task Context switching, Task stack allocation and initialization and `CriticalSection_t` object implementation. `CriticalSection_t` class is used as RAII object to control global interrupt enable flag in MPU status register for `main()` and each Task independently. `Context_t` POD⁵ structure⁶ is shown in figure 19. `ptrStackTop` is 16-bit `uint8_t*` type union field located over first two bytes of the Context structure. It is used for convinient access to `sph:spl` fields (storing stack pointer registers) of the structure. Last byte of the structure is a counter used by `CriticalSection_t` class for recursive critical section enter/exit operations. Initialized Kernel objects and their relationship are shown in figure 20. There are two kernel system objects: `SystemMainContext` – Sheduler's `main()` Context, and `ptrSystemCurrentContext` – pointer to the running Task's or `main()`'s Context. After initialization it contains address of `SystemMainContext`. Each Context has `ptrStackTop` member which points to the top of its own stack. Tasks may have different stack sizes.

⁴In this case constructor of the `Signal_t` must be explicitly set in the initializing list of the derived class.

⁵Plain Old Data

⁶`__attribute__((packed))` is used in `Context_t` declaration to avoid padding

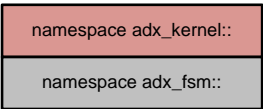


Figure 17: Header dependencies, types, functions and system objects

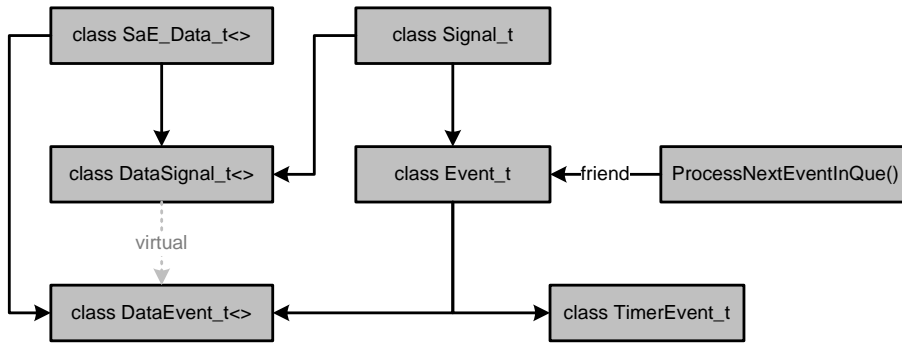


Figure 18: Class inheritance

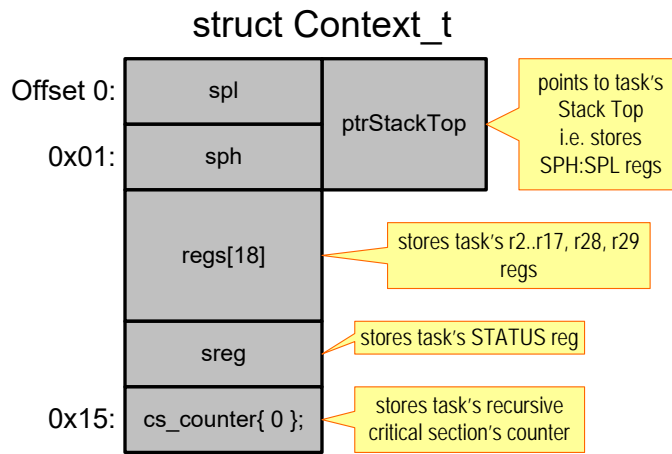


Figure 19: struct Context_t layout

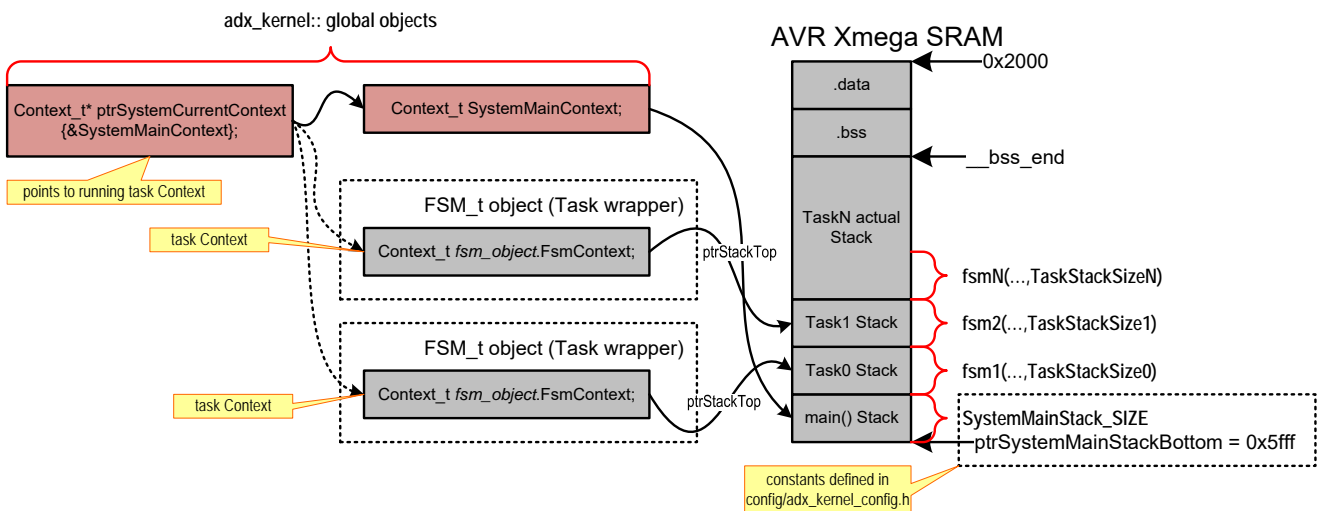


Figure 20: Kernel objects in SRAM

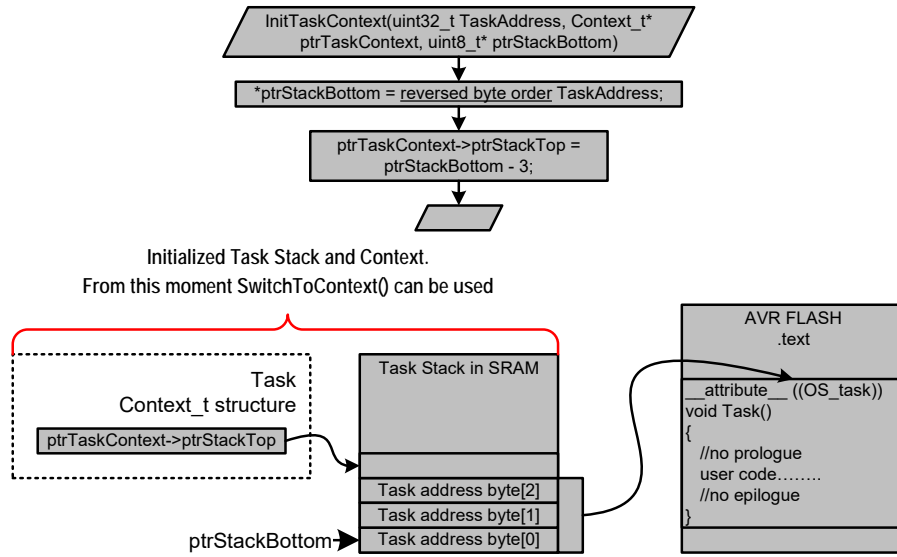


Figure 21: Task stack and Context initialization



To locate Task at some user-defined address, create FLASH segment (see Project properties > AVR/GNU Linker > Memory Settings: `.myprogramsection=0xMyFarAddress`), then assign section attribute to the Task: `void MyTask() __attribute__((section(\".myprogramsection\")))`;

The Task stack and Context initialization by `InitTaskContext()` function is shown in figure 21. It should be noted, that call and ret MPU instructions use 3-byte program space address, which is pushed to the stack by call, and popped by ret. But pointer to the function type in avr-gcc has 2-byte address that reduces the Task allocation to the first 64kB of FLASH. To avoid this restriction `uint32_t` is used as a type for Task address. To pass the address to `InitTaskContext()` function one may use three ways:

- `static_cast<uint32_t>(Task)` for ordinary near address. Pointer to Task type can be defined as `using Task_t = void (*)()`;
- `pgm_get_far_address(Task)` macros⁷ defined in `avr/pgmspace.h`;
- literal constant address defined by allocation of user section in program memory using function attribute.

To reduce `Task()` size `__attribute__((OS_Task))` can be used. Functions with this attribute⁸ don't save and restore call-saved registers in its prolog and epilog accordingly.

Context switching is implemented in `SwitchToContext()` function, figure 22. `ptrSystemCurrentContext` stores pointer to previous Context, next Context is passed to `SwitchToContext()` function. Saving and restoring MPU registers is implemented in helper `SwitchContext()` function declared as

```
extern "C" void SwitchContext(Context_t*
    ptrCurrentContext, Context_t* ptrNextContext);
```

⁷It seems that `pgm_get_far_address()` doesn't work properly for function addresses in avr-gcc 4.2.

⁸avr-gcc attributes are described here: <https://gcc.gnu.org/onlinedocs/gcc/AVR-Function-Attributes.html>

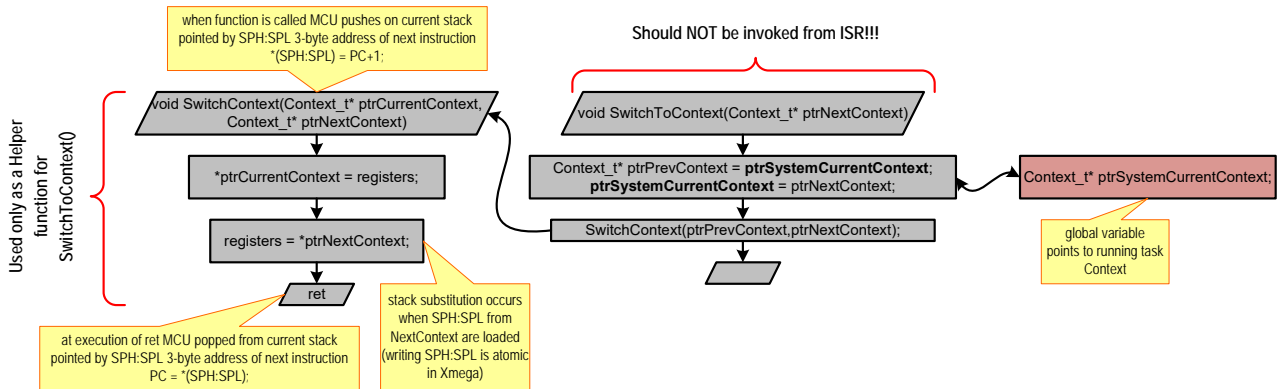


Figure 22: Context switching implementation

It is listed below:

```

1 ;$Id: adx_kernel_SwitchContext.S 263 2024-11-11 13:00:52Z apolv $
2
3 .global SwitchContext
4
5 SwitchContext:
6 ;void SwitchContext(ptrCurrentContext, ptrNextContext); //Do NOT
   invoke from ISR!!!
7 ;Save current Context (SPL, SPH, r2, .., r17, r28, r29, SREG), r1
   must be 0 after return
8 ;registers r0, r18, .., r27, r30, r31 may not be saved
9 movw r26, r24 ;X = ptrCurrentContext (r25:r24 - 1st 2-byte
   parameter)
10 in r0, 0x3f ;r0 = SREG
11 in r1, 0x3d ;r1 = SPL
12 st X+, r1 ;*(ptrCurrentContext + 0) = SPL
13 in r1, 0x3e ;r1 = SPH
14 st X+, r1 ;*(ptrCurrentContext + 1) = SPH
15 st X+, r2 ;*(ptrCurrentContext + 2) = r2
16 st X+, r3 ;...
17 st X+, r4
18 st X+, r5
19 st X+, r6
20 st X+, r7
21 st X+, r8
22 st X+, r9
23 st X+, r10
24 st X+, r11
25 st X+, r12
26 st X+, r13
27 st X+, r14
28 st X+, r15
29 st X+, r16
30 st X+, r17
31 st X+, r28
32 st X+, r29 ;*(ptrCurrentContext + 19) = r29
33 st X+, r0 ;*(ptrCurrentContext + 20) = SREG
34 ;Load new Context (SPL, SPH, r2, .., r17, r28, r29, SREG)
35 movw r26, r22 ;X = ptrNextContext (r23:r22 - 2nd 2-byte parameter)
36 ld r0, X+ ;r0 = *(ptrNextContext + 0) //next SPL
37 ld r1, X+ ;r1 = *(ptrNextContext + 1) //next SPH

```

```

38 ld r2, X+      ;r2 = *(ptrNextContext + 2) //next r2
39 ld r3, X+      ;...
40 ld r4, X+
41 ld r5, X+
42 ld r6, X+
43 ld r7, X+
44 ld r8, X+
45 ld r9, X+
46 ld r10, X+
47 ld r11, X+
48 ld r12, X+
49 ld r13, X+
50 ld r14, X+
51 ld r15, X+
52 ld r16, X+
53 ld r17, X+
54 ld r28, X+
55 ld r29, X+      ;r29 = *(ptrNextContext + 19) //next r29
56 ld r18, X+      ;r18 = next SREG
57 ;start of switching context
58 out 0x3d, r0     ;SPL = next SPL //Interrupts are off by XMEGA
    hardware until next I/O, i.e. SPH loading (next instruction)
59 out 0x3e, r1     ;SPH = next SPH
60 ;context switched
61 out 0x3f, r18    ;SREG = next SREG
62 clr r1          ;r1 = 0 //according to avr-gcc calling conventions
    r1 must be 0 after function return
63 ret             ;gets 3-byte pointer from new (already switched)
    Stack and jumps to new Task

```


When this function is called (using call instruction) MPU pushes the program address of the next instruction (that is after call instruction in program space) on stack according to the address stored in SPH:SPL stack pointer registers. Least significant byte of the instruction address is pushed first, stack pointer (SPH:SPL) is decremented⁹ after each push operation. As a result, three bytes of address are pushed, SPH:SPL is decremented by 3, and SPH:SPL points to the first free byte of the stack.

SwitchContext() function saves current stack pointer (SPH:SPL registers), r2 to r17, r28, r29 and SREG to current Context object. Then it loads appropriate values to these registers¹⁰ from the next Context object, including new SPH:SPL stack pointer which now points to another stack associated with next Context. When ret (last) instruction is executed, MPU pops 3 bytes of the instruction address (which is going to be executed next) from the stack using SPH:SPL stack pointer. But now SPH:SPL points to another (next) stack, so it will be popped the program address of another instruction that was previously saved by the same way using SwitchContext() function. Thus entry in SwitchContext() is in one Task, and exit – in another.

Since SwitchContext() is an ordinary function for compiler and is invoked from Task or main(), there is no need to save all registers, only call-saved registers according to avr-gcc calling conventions (see https://gcc.gnu.org/wiki/avr-gcc#Calling_Convention). This reduces context switching time. Also for xmega devices there is no need to disable interrupts when modifying SPH:SPL register pair, it is executed as atomic operation by hardware – interrupts are

⁹Stack grows from high addresses towards zero address.

¹⁰r1 is implicitly call-saved (fixed zero register), so it must be cleared to 0 before ret

 SwitchContext() so the SwitchToContext() functions can't be invoked from ISR!

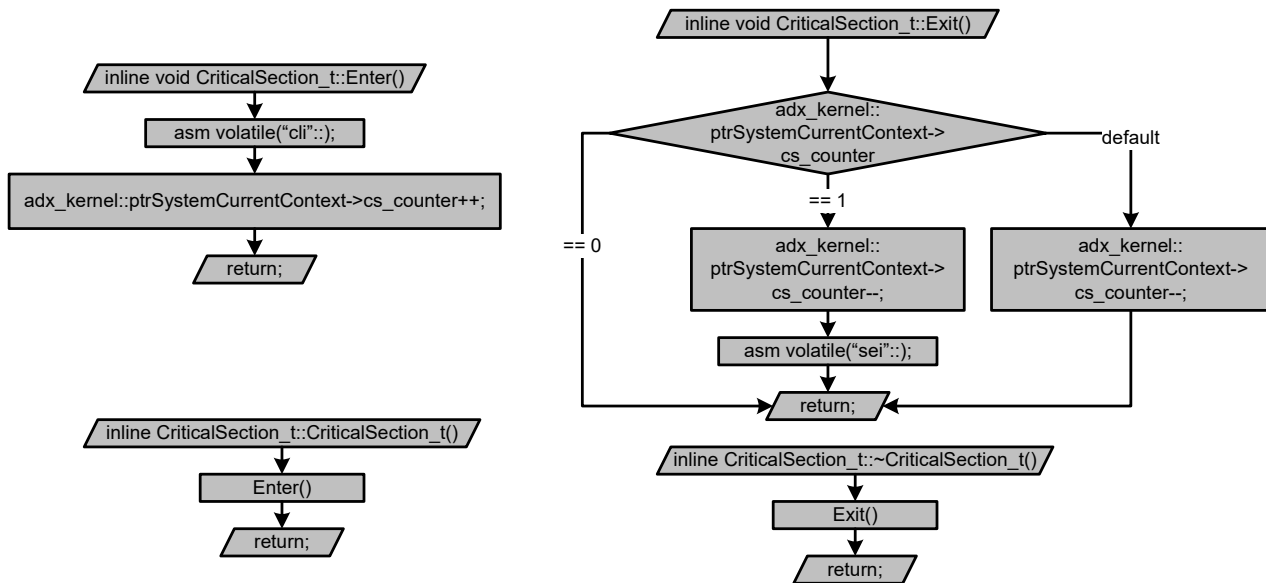


Figure 23: CriticalSection_t class

delayed after SPL writing for up to 4 cycles or until next I/O operation, i.e. SPH writing.

When working with multiple-byte data that is shared between Task and ISR, some parts of code may require protection, otherwise ISR can happen when this part is executing and can read inconsistent data which are partially modified by the Task, or modify shared data which are partially read by the Task. Anyway it will result in unpredictable consequences. So interrupts should be disabled while executing such critical sections of code.

To control global interrupt enable flag (IE bit of SREG) CriticalSection_t class is used (fig. 23). It supports RAII¹¹ idiom, and can be used as local object enclosed in scope {} that defines its lifetime, or by direct calling static Enter()/Exit() methods. Ctor of CriticalSection_t calls Enter() method, dtor – Exit(). Implementation is recursive, it means that nested Enter()/Exit() pair can be used. At first Enter() call interrupts are disabled, every Enter() call increments cs_counter. Exit() call decrements cs_counter. When cs_counter is zero, interrupts are enabled, further Exit() call do nothing. cs_counter is a part of Context (fig. 19) thus each Task and main() has its own cs_counter. It's implemented this way for more predictable and independent Task behavior. Although it is possible to call blocking¹² functions within critical section, and it **won't affect** to other Tasks, but one should remember that it will split critical section into several consequent sections.

Table 1 summarizes kernel header interfaces.

Figure 24 shows the example of context switching application using kernel interfaces.

¹¹Resource acquisition is initialization

¹²Blocking functions – functions which execution may result in context switching. AdxFSM blocking functions names are ended with _bl suffix.

i For xmega 16-bit registers when writing to low register first – data is written to temp register, then when writing to high, low is loaded from temp in the same cycle as high register.

! Critical sections should be as short as possible! Otherwise interrupt may be delayed for too long time for its proper handling. Always use pair Enter()/Exit() functions or local RAII object (preferred). Don't split critical section by blocking functions. Use only CriticalSection_t class for IE flag control, don't use other functions containing "cli"/"sei" instructions unless they don't save and restore IE state.

i At the entry to the Task interrupts are disabled until the first use of critical section Enter()/Exit() pair. CriticalSection_t object can be used anywhere in the code (ISR, Task, main()), except global space or static variable.

Table 1: Kernel summary (kernel/adx_kernel.h)

Name	Namespace	Description
Types		
struct Context_t	adx_kernel::	Structure type for Context objects
class CriticalSection_t	adx_fsm::	Class for recursive critical section object. Enters to cs at creation of local object, exits when it is destroyed. Also has static CriticalSection_t::Enter()/CriticalSection_t::Exit() pair for cs entry/exit.
Objects		
Context_t SystemMainContext	adx_kernel::	Scheduler's, i.e. main()'s Context
Context_t* ptrSystemCurrentContext	adx_kernel::	Pointer to the current running (main() or Task) Context.
Free Functions		
inline void SwitchToContext(Context_t* ptrNextContext)	adx_kernel::	Switches from current running to the next Context passed as parameter.
extern "C" void SwitchContext(Context_t* ptrCurrentContext, Context_t* ptrNextContext)	adx_kernel::	Written in assembly helper function for SwitchToContext().
inline void __nop()	adx_fsm::	asm volatile("nop"); One cycle no operation instruction, can be used in debugging purposes.

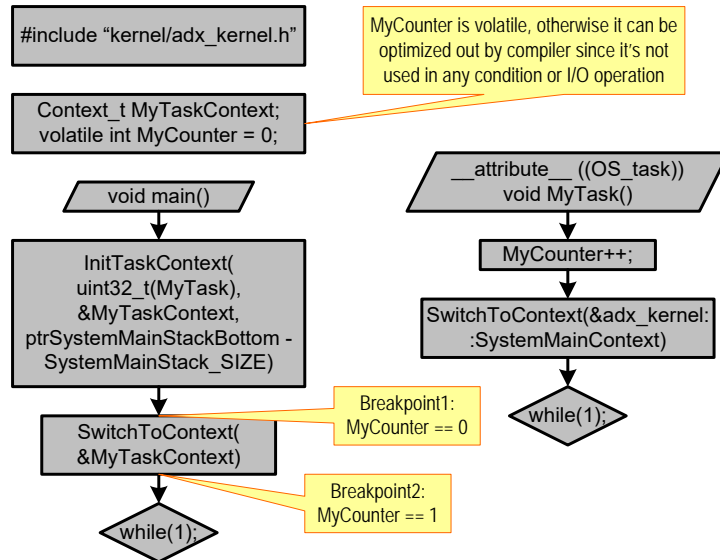


Figure 24: Context switching application example

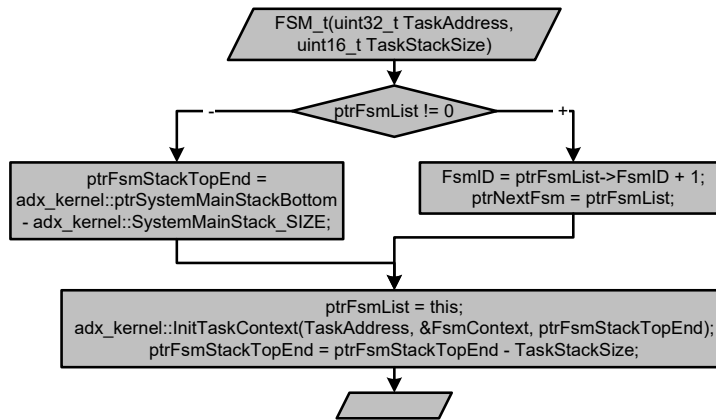


Figure 25: FSM_t constructor

3.2 Task Wrapper (FSM_t)

class FSM_t (Task wrapper) declared in core/adx_fsm.h is used to solve the following problems:

1. Task data (Context, ID, etc.) encapsulation and its safe handling;
2. Automatic safe initialization of Task stack and Context structure;
3. Automatic Task ID assignment;
4. Automatic first run of the Task;
5. Stores information about caller (previous Task or main()) from where context was switched to current Task. This information is used for return control to the caller, when needed.
6. Stores information about Signal or Event which was used by caller when switching context to current Task.

Table 2 contains summary of FSM_t class and related free functions. Declared type aliases:

```

using SignalCounter_t = uint16_t;
class Signal_t;
using ptrSignal_t = Signal_t*;
using ptrFSM_t = FSM_t*;
  
```

This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Figure 25 shows FSM_t constructor algorithm.

Figure 26 shows the state of FSM_t and kernel objects after last FSM_t object ctor execution.

Figure 27 shows FSM_t::Start() algorithm, used for start of each Task before Scheduler's main() loop. It should be done for proper Tasks operation. Otherwise events which should be sent by Tasks after entry to the Tasks won't be sent and event queue will be empty, so Scheduler will run idle loop.



FSM_t::Start() should be called after hardware initialization but before Scheduler's main() loop!

Table 2: FSM_t and related functions summary (core/adx_fsm.h, adx_fsm::)

Type/Name	Access	Description
Constructor		
FSM_t(uint32_t TaskAddress, uint16_t TaskStackSize)	public	Initializes assigned to FSM_t object Task stack and Context structure, FsmID, and creates the list of all FSM_t objects, which is used by Start() static method for the first run of each Task.
Methods		
virtual void operator() (ptrSignal_t ptrSignal, SignalCounter_t SignalCounter)	public	Saves input parameters and pointer to current running FSM to local members, sets current running FSM static member to its address, and switches to its Context (runs assigned Task).
Static Methods		
static void Start()	public	Iterates through the list of all objects, and runs each assigned Task by calling operator(). As input parameters are passed nullptr for signal pointer and 0 for signal counter. This method is used for the first run of the Tasks before Scheduler's main() loop.
Local Data Members		
ptrFSM_t ptrNextFsm{ nullptr };uint8_t FsmID{ 0 };	private	Used in init process for fsm objects list. FsmID actually is not used in internal mechanics, it's intended for user purposes only.
adx_kernel::Context_t FsmContext{};	protected	Used in calling and return process, i.e. operator() and Wait().
ptrFSM_t ptrPrevFsm { nullptr };	protected	Signal's address which was used when invoking FSM, and Signal's counter sampled at the moment of FSM calling. For more details see Signal_t and Event_t description.
ptrSignal_t ptrFsmSignal { nullptr };	protected	
SignalCounter_t FsmSignalCounter{ 0 };		
Static Data Members		
ptrFSM_t FSM_t::ptrFsmList{ nullptr };	private	Points to the last FSM object in the list of FSM objects. Used in init process to create list of FSM objects, and by FSM_t::Start() for first run of each Task.
ptrFSM_t FSM_t::ptrRunningFsm{ nullptr };	protected	Points to the running FSM. Used in friend free functions to retrieve data of the running FSM.
uint8_t* FSM_t::ptrFsmStackTopEnd{ nullptr };	private	Points to the Task Stack End (upper top) of the last FSM object. Used in init process for Task stack allocation.
Free Functions		
friend inline void Wait()		Restores current running FSM static member and switches to previous Context.

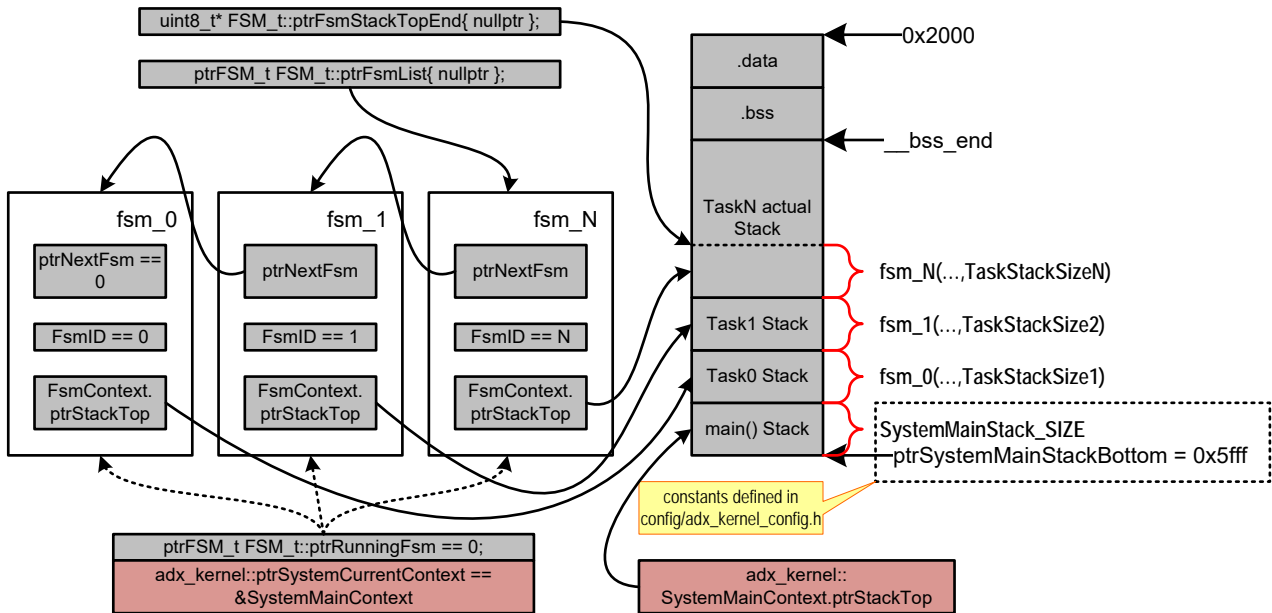


Figure 26: State of FSM_t and kernel objects after last FSM_t object ctor execution

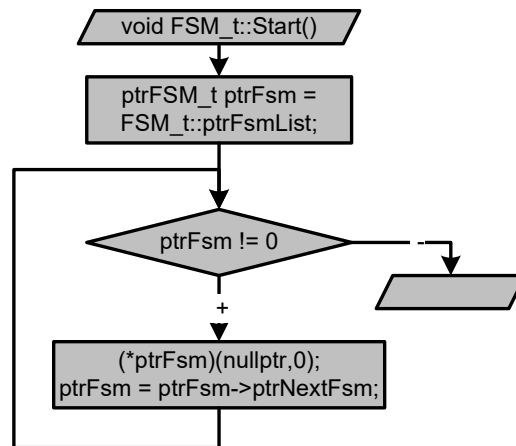


Figure 27: FSM_t::Start()

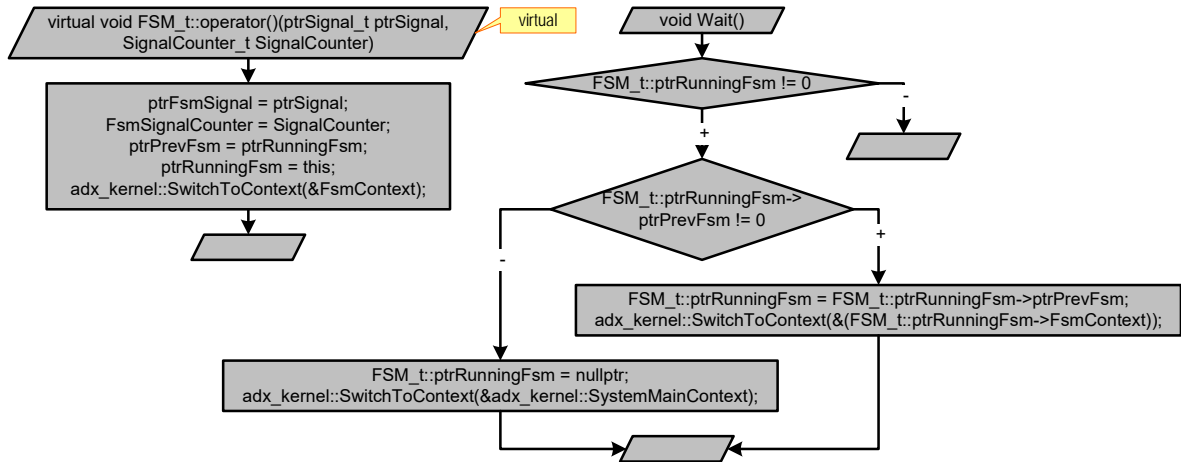


Figure 28: FSM_t::operator() and Wait()

Figure 28 shows Task “call” and “return” algorithms, implemented by FSM_t::operator() virtual method and Wait() free function. Operator() is used to enable functor behavior of FSM_t object, so it makes possible to use it in templates as it would be ordinary function. Thus one C++ template code can be written for calling functors and ordinary functions, although it result in different object code, because to call function it’s enough to know its address and signature, and to call functor – address of user type object is also required since it is passed to method as implicit “this” parameter.

Wait() is a free function because it’s called within Task which originally knows nothing about FSM_t object to which it’s assigned. Both FSM_t::operator() and Wait() always result in context switching, except the case when Wait() is called from main() that has no sense in practice.

When calling Wait() it works as “return” to previously running context (from where Task was “called”). When calling FSM_t object (as a functor via operator()), it behaves similar to “call”, but Task resumes its execution after the line where Wait() previously was called, not from the beginning of the Task function. At first run Task starts from the beginning. Figure 29 describe FSM_t and kernel objects state after nested call of FSM_t functors when Task0 is “called” from main() and Task1 – from Task0. Figure 30 shows “return” to main() from Task1 via Task0.

3.3 Signals (Signal_t)

Class Signal_t (base for class Event_t) declared in core/adx_Signal.h is used for Task synchronization. It has the following key features:

1. Send() method calls assigned to Signal FSM Task by means of direct context switching to that Task, thus scheduler is not used. Signals are at least three times faster than Events.
2. Task is called only if Signal is in Active state. In this case Send() will pass Signal’s address and Counter value as parameters to fsm before switching

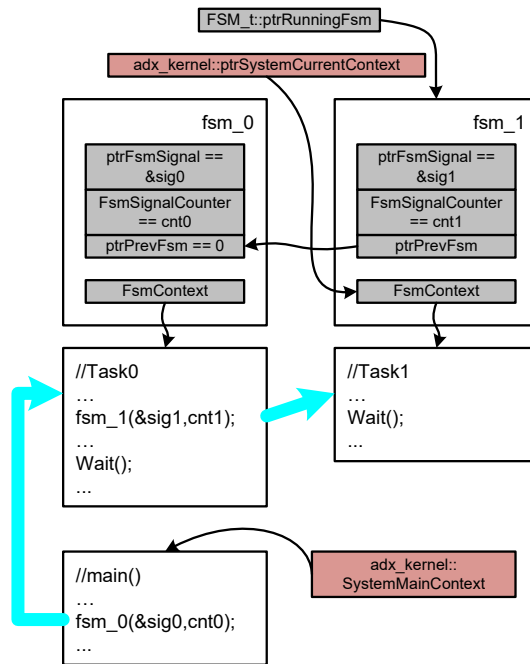


Figure 29: FSM_t and kernel objects state after after `main()` call `fsm_0()`, `Task_0` call `fsm_1()`

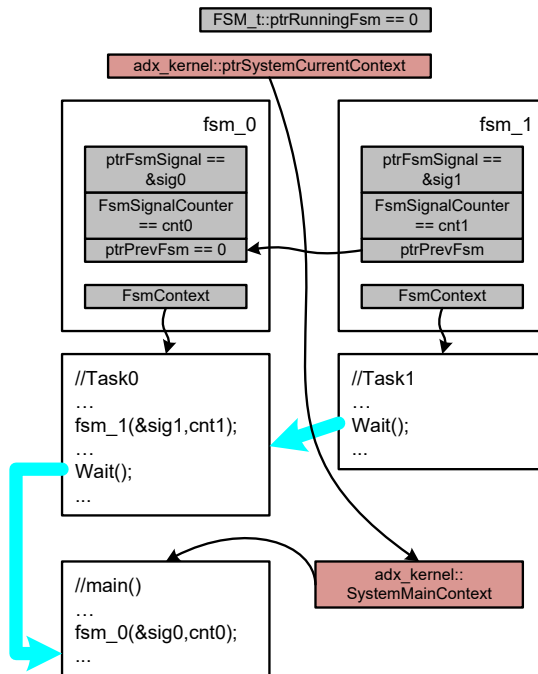
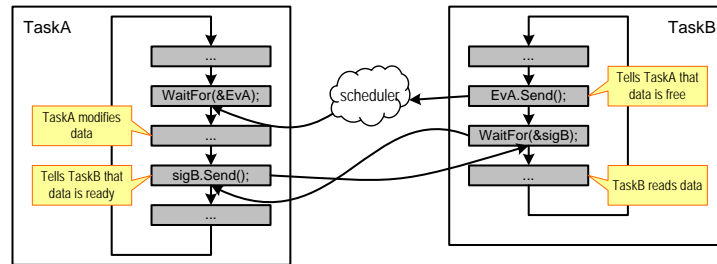
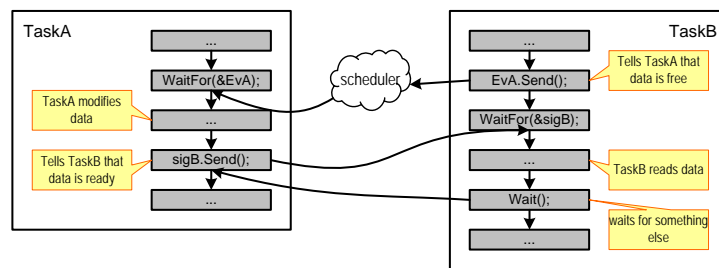


Figure 30: FSM_t and kernel objects state after recursive `Wait()` call (i.e. return to original caller)



(a) Short loop with one waiting state



(b) Long loop with multiple waiting states

Figure 31: Example of Signal use as an acknowledge

context to the Task, then Counter is reset to 0.

3. Wait() function in called Task will return control to the caller Task or main().
4. When Signal is in Active state Send() method returns true, otherwise it returns false.
5. Counter is up by 1 on each Send() call if Signal is in Active or Silent state.
6. Counter == 0 in Disabled state.
7. As opposed to Events, signals can form a loop, thus control will never return to scheduler. So be careful using Signals.
8. Signals can't be sent from or to ISR!
9. Signals can be safely used as an acknowledge from the Task after sending Event to that Task. This use case is shown in figure 31.
10. To "catch" a Signal in a "right place" in the Task, set mode of the Signal that you are waiting for to Active just before the Wait(), and set Mode to Silent or Disabled right after the Wait(). Free function WaitFor() is implemented this way.
11. Optional Error handling is supported.

Table 3 contains summary of Signal_t class and related free functions. Declared types and aliases:

Table 3: Signal_t and related functions summary (core/adx_Signal.h, adx_fsm::)

Type/Name	Access	Description
Constructors		
Signal_t(ptrFSM_t initPtrFsm, Mode_t initMode = Mode_t::Silent)	public	Creates Signal_t object: initializes pointer to FSM_t object which Task is going to be “called” using this Signal, and sets initial Mode (default value = Silent). FSM (i.e Task) assignment can be done only in the ctor, it can’t be changed later.
Methods		
virtual inline bool Send()	public	Used for direct Task “call” without Scheduler.
virtual inline bool operator ()()	public	{return Send();} Used as functor in DataQueue_t objects.
virtual void SetMode(Mode_t Mode)	public	Sets Mode of the Signal (Signal_t object).
Local Data Members		
ptrFSM_t const ptrFsm;	protected	Points to FSM object which Task should be “called” using Send() method. Initialized only once in ctor, can’t be modified.
SignalCounter_t Counter{ 0 };	protected	Counts how many timed Send() was called. Cleared (set to 0) when Send() called in Active Mode.
Mode_t Mode{ Mode_t::Disabled };	protected	State of the Signal (type alias SignalMode_t): Disabled, Silent, Active.
Static Data Members		
ErrorHandler_t Signal_t::ErrorHandler{ nullptr };	protected	Points to optional Error handler, provided by user. For more details see SystemStatus_t description.
Free Functions		
inline void WaitFor(ptrSignal_t ptrWaitingSignal)		Sets mode of input Signal to Active, calls Wait() then sets Signal mode to Silent.

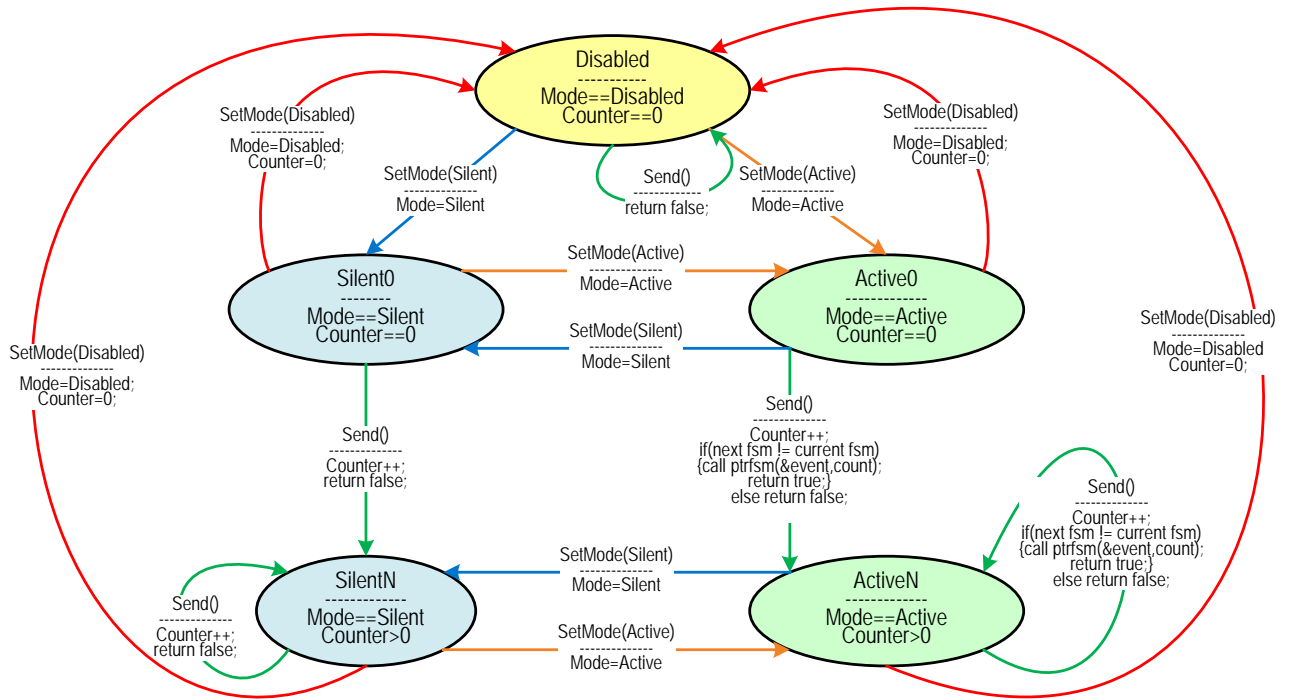


Figure 32: Signal states and transitions

```

enum class Signal_t::Mode_t : uint8_t
{
    Disabled = 0,
    Silent,
    Active = 3
};
using SignalMode_t = Signal_t::Mode_t;

```

This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Figure 32 shows states and transitions of Signal that defines its behavior. Send() and SetMode() logic which works according to this diagram is shown in figures 33 and 34 accordingly. These methods are made virtual because they are overridden in Event_t class derived from Signal_t. In most cases this enables to work with both object types the same way since its behavior is similar.

In most cases there are FSM waiting states with only one transition (i.e. waiting for one known signal). WaitFor() function is implemented to simplify this use case, fig 35.

3.4 Base Queue (SystemQueue_t)

Template Class SystemQueue_t<uint8_t SIZE = 16, typename T = uint8_t> is basic circular buffer FIFO. It is used in various system objects (e.g. in event queue), also can be applied in user logic.

1. SIZE template parameter is length of internal data array of type T, defined

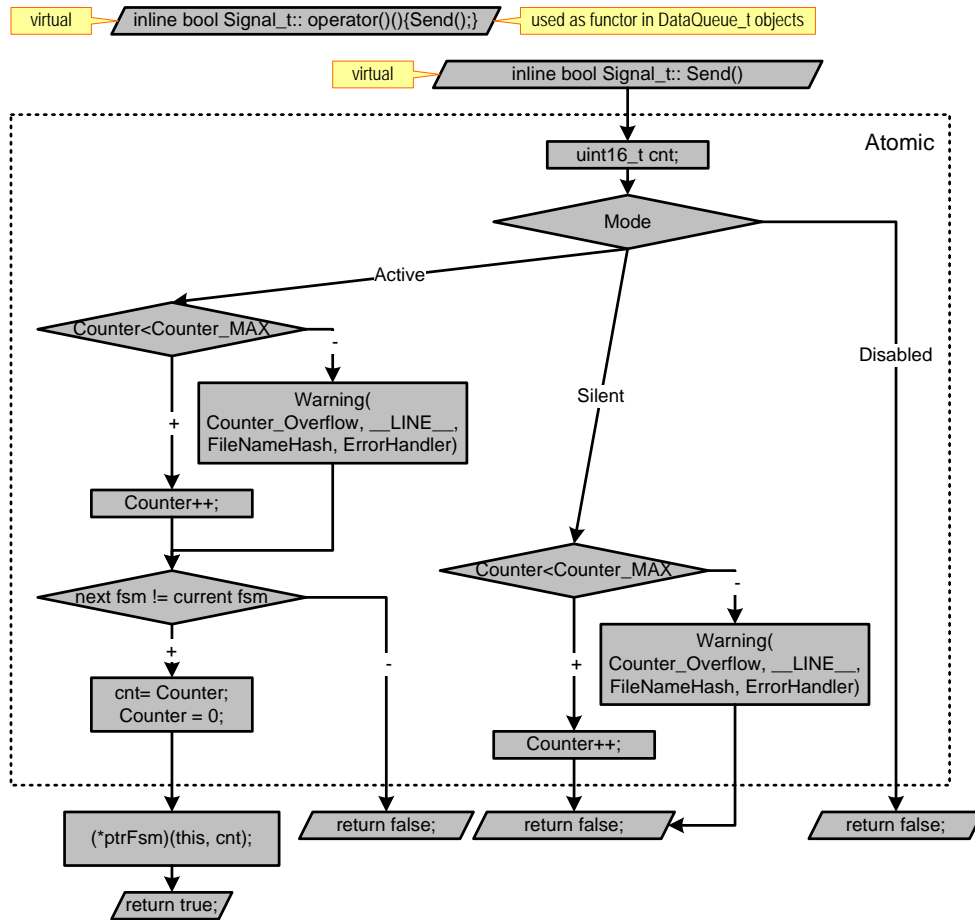


Figure 33: Signal_t::Send() method logic

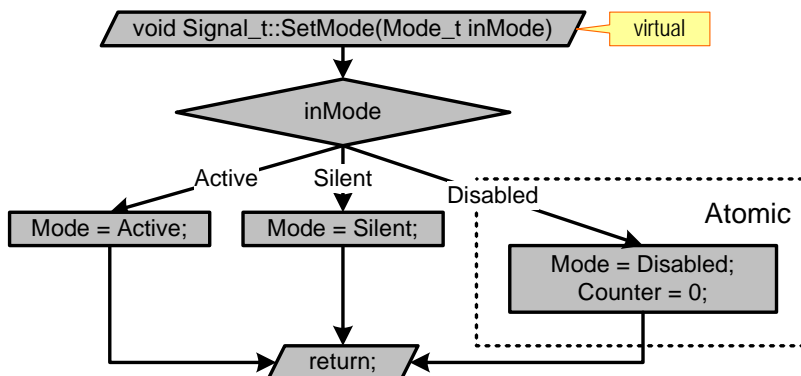


Figure 34: Signal_t::SetMode() method logic

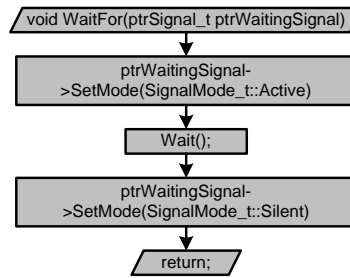


Figure 35: WaitFor() can be used for waiting one known signal

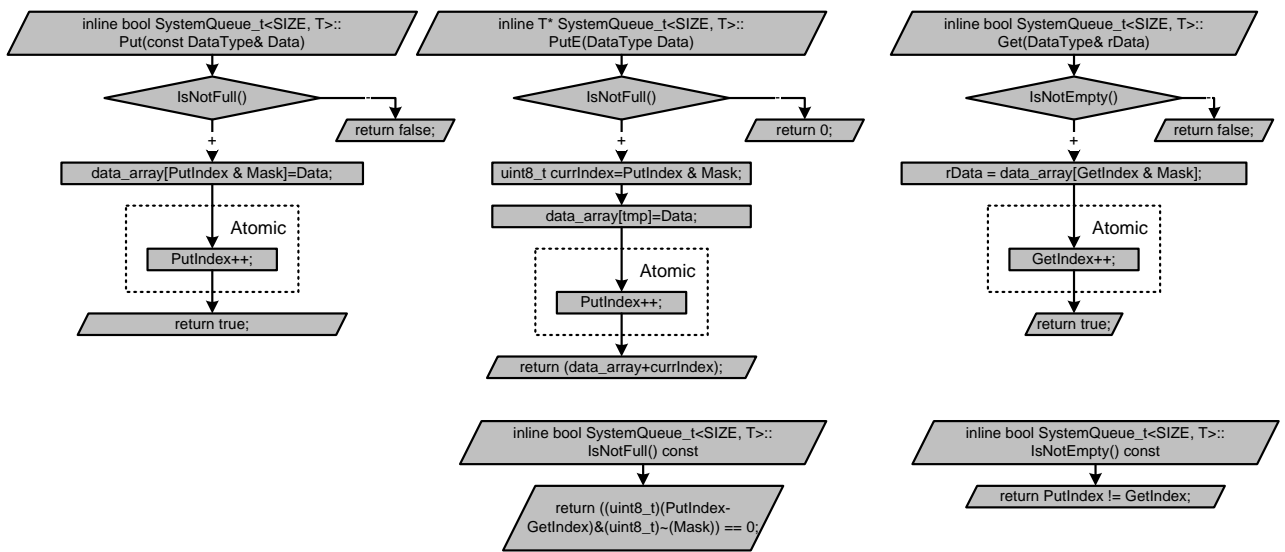


Figure 36: SystemQueue_t<SIZE,T> template class

as T data_array[SIZE].

2. T template parameter – data type.
3. Buffer size should be power of 2 in range from 1 to 128 including.
4. Since indexes are 1-byte size, put/get can be done from ISR as well from Task w/o critical section, except multiple put/get to/from the same queue.
5. PutE() is the modified ordinary Put() method, it returns pointer to the Data stored in the buffer if put operation was successful and nullptr otherwise. This method is used in Event_t class.

FIFO basic methods are shown in figure 36.

3.5 Events (Event_t)

Class Event_t (derived from class Signal_t) is the basic way for Task synchronization. Event_t has similar behavior as Signal_t except for the following:

1. Send() method puts Event to the event queue and returns control immediately. Scheduler will run the Task that is assigned to Event later.
2. Event can be processed by the scheduler only in Active state, otherwise it will be thrown away from queue until Active state is set.
3. If Event is set to Active state and its Counter>0, it will be automatically put in event queue if it's not in queue yet.
4. Event has three priority levels: EventPrty_t::Low, EventPrty_t::Middle, EventPrty_t::High. Thus there are three event queues - one for each priority level.
5. Events are processed in the same order as they have been put in the assigned priority queue.
6. Send() method always returns true except critical error (buffer overflow, etc.)
7. Events can be safely used anywhere including ISR.

Other features are the same as for Signals, including:

1. Wait() function in called Task will return control to the caller Task or main().
2. Counter is up by 1 on each Send() call if Signal is in Active or Silent state.
3. Counter == 0 in Disabled state.
4. To "catch" an Event in a "right place" in the Task, set mode of the Event that you are waiting for to Active just before the Wait(), and set Mode to Silent or Disabled right after the Wait(). Free function WaitFor() is implemented this way.
5. Optional Error handling is supported by base Signal_t.

Table 4 contains summary of Event_t class and related free functions. Declared types and aliases:

```
using ptrEvent_t = Event_t*;
enum class Event_t::Priority_t : uint8_t
{
    Low = 0,
    Middle,
    High
};
using EventPrty_t = Event_t::Priority_t;
```

This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Figure 38 shows states and transitions of Event that defines its behavior. Send() and SetMode() logic which works according to this diagram is shown in figures 39 and 40 accordingly. These methods override methods of Signal_t. In most cases this enables to work with both object types the same way since its behavior is similar.

Figure 41 on page 50 shows data relationship in the Event_t objects.

i Wait() and WaitFor() functions always result in Context switching ("return" to the caller). They don't check if Event is already happened and Active to proceed the execution without return to Scheduler. This is done for more predictable and reliable event sequence processing. At first it seems to be not efficient, but in normal applications most work is processed within loop while checking some variable (e.g. full/empty in FIFO operations), and Events are used only when data ready condition is not met and real waiting is required. So this behavior won't result in any significant loss in efficiency in most applications.

Table 4: Event_t and related functions summary (core/adx_Event.h, adx_fsm::)

Type/Name	Access	Description
Constructors		
Event_t(ptrFSM_t ptrFsmFuncion, Mode_t initMode = Mode_t::Silent, Priority_t initPriority = Priority_t::Middle)	public	Creates Event_t object: calls Signal_t base class ctor, and initializes Event Mode (default value is Silent) and Priority (default value is Middle).
Destructor		
~Event_t()	public	Removes the pointer to the Event from the event queue if it's not removed yet, fig 37.
Methods		
virtual inline bool Send()	public	Used for indirect Task "call" via Scheduler.
virtual inline bool operator()()	public	{return Send();} Used as functor in DataQueue_t objects.
virtual void SetMode(Mode_t Mode)	public	Sets Mode of the Event (Event_t object).
Local Data Members		
ptrEvent_t* pptrEventInQueue{ nullptr };	private	Points to location in event queue buffer which stores pointer to the Event object.
Priority_t Priority{ Priority_t::Middle };	private	Event priority defines the event queue which event pointer will be put in.
Static Data Members		
static SystemQueue_t< LowPrioritySystemEventQueue_SIZE, ptrEvent_t> LowPrtyEventQueue; static SystemQueue_t< MiddlePrioritySystemEventQueue_SIZE, ptrEvent_t> MiddlePrtyEventQueue; static SystemQueue_t< HighPrioritySystemEventQueue_SIZE, ptrEvent_t> HighPrtyEventQueue;	private	Event queues, sizes (queue lengths) are defined in config/adx_core_config.h These queues store only pointers on Event objects, not objects itself.
Free Functions		
bool ProcessNextEventInQue()	friend	Contains Scheduler's main logic, for more details see section 3.7 on page 54.

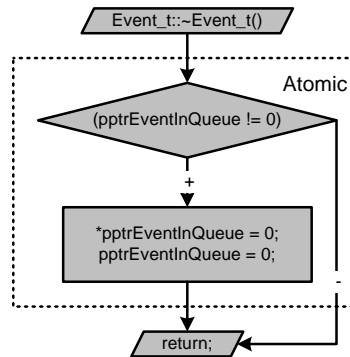


Figure 37: ~Event_t() destructor

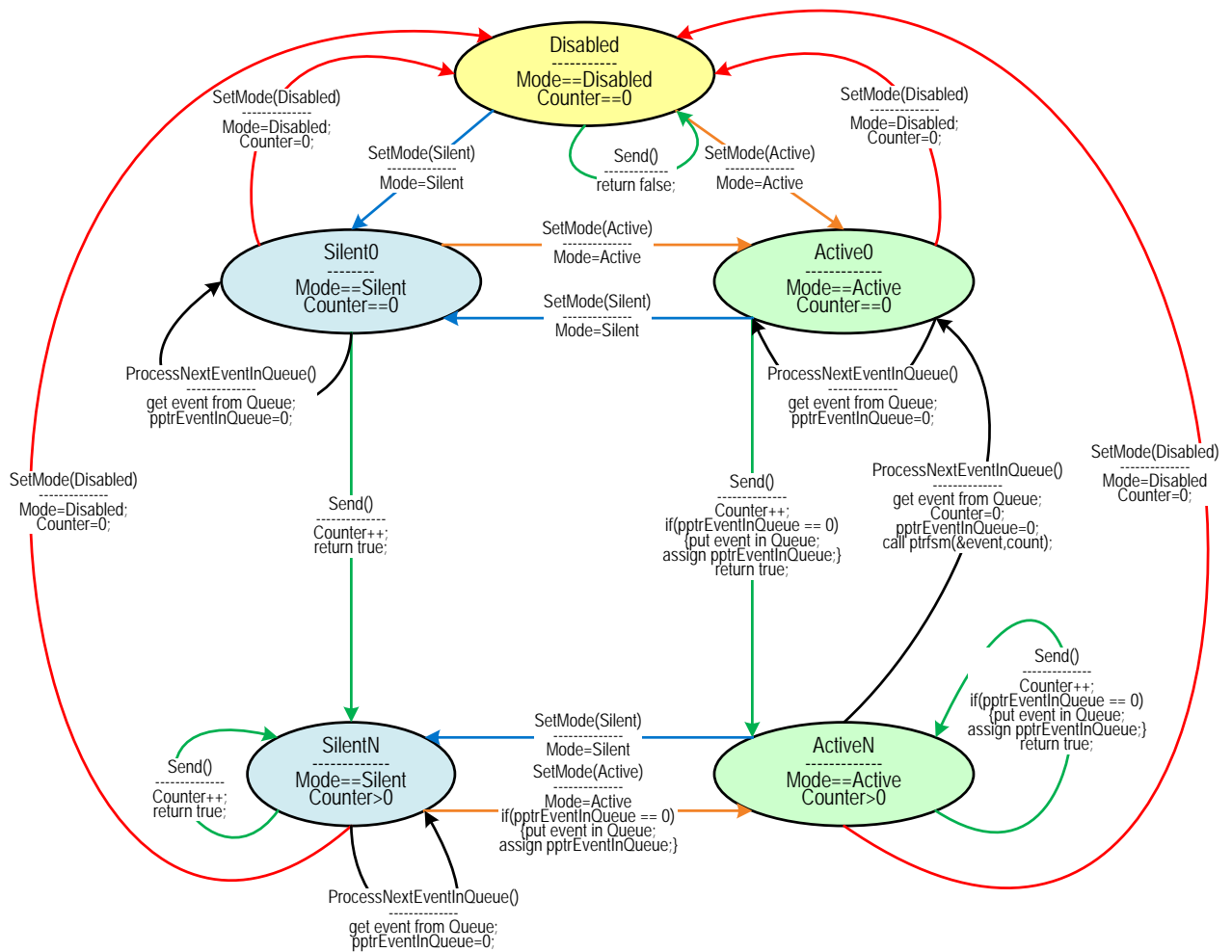


Figure 38: Signal states and transitions

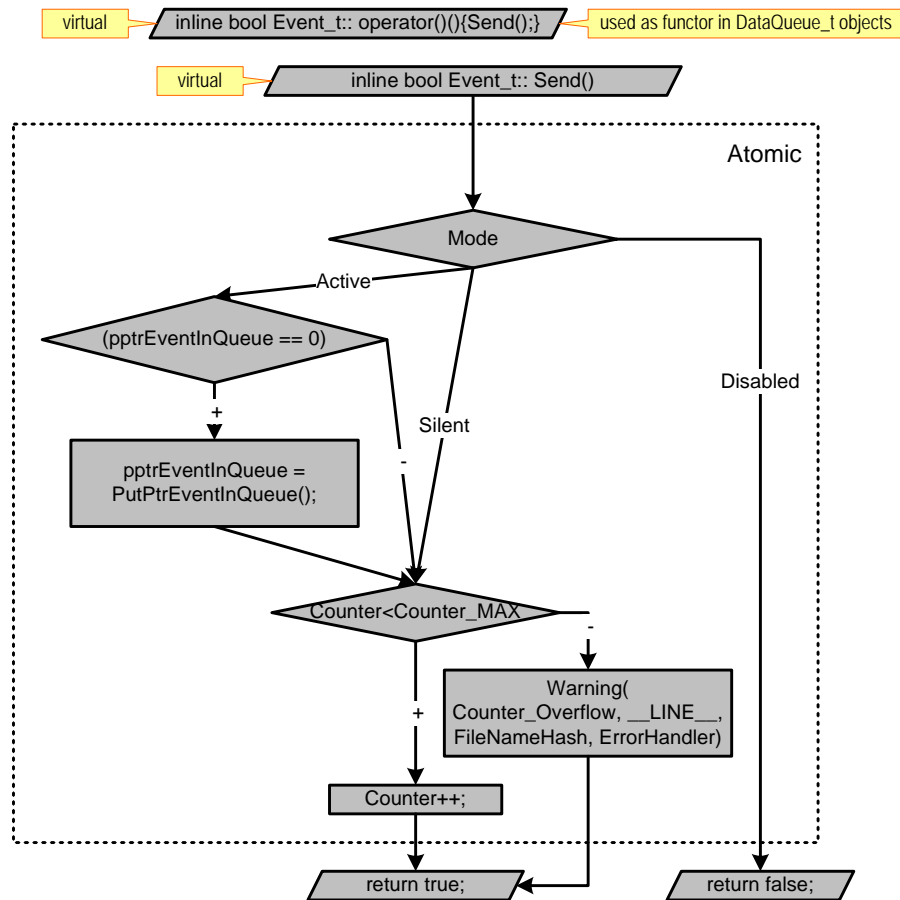


Figure 39: Event_t::Send() method logic

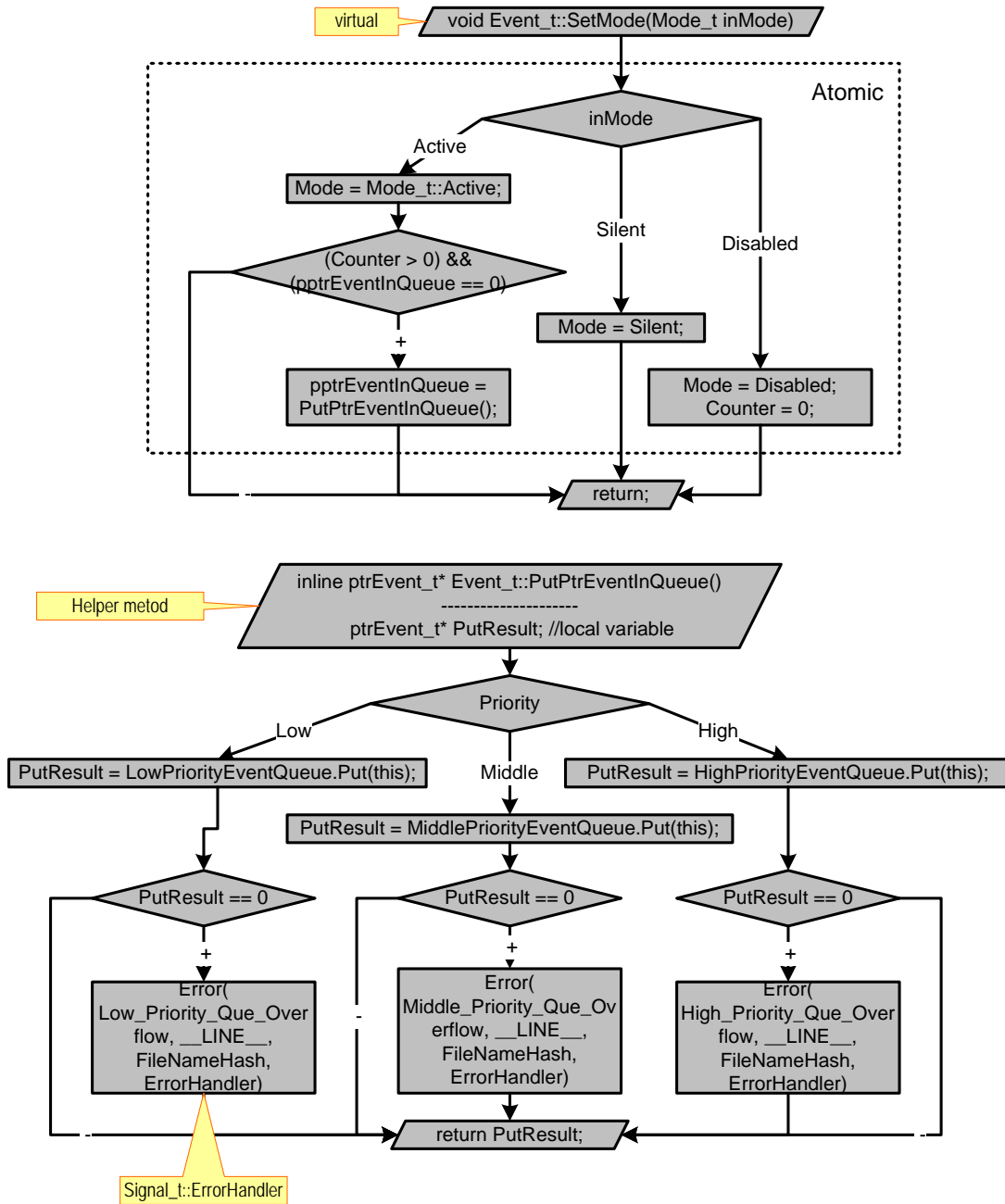


Figure 40: Event_t::SetMode() method logic

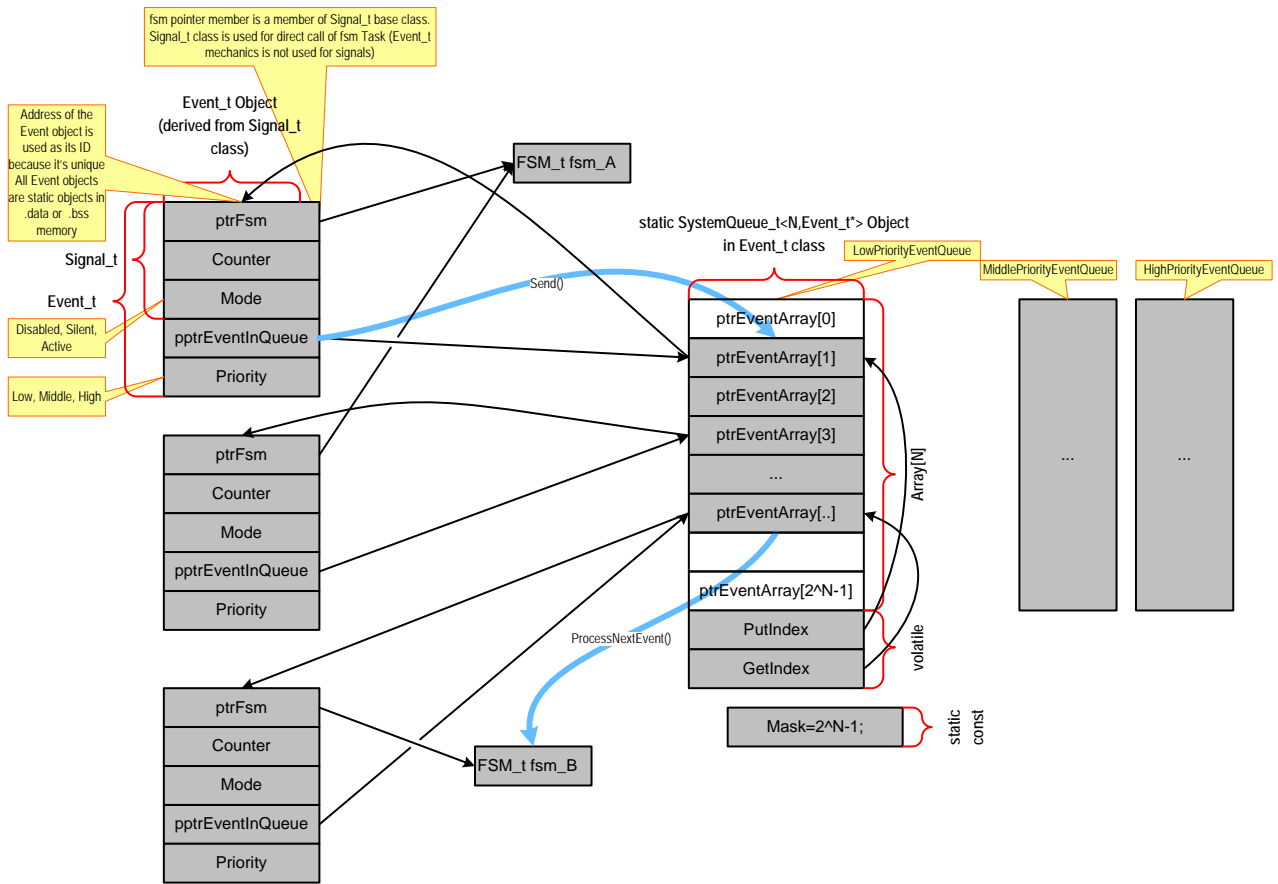


Figure 41: Event_t data relationship

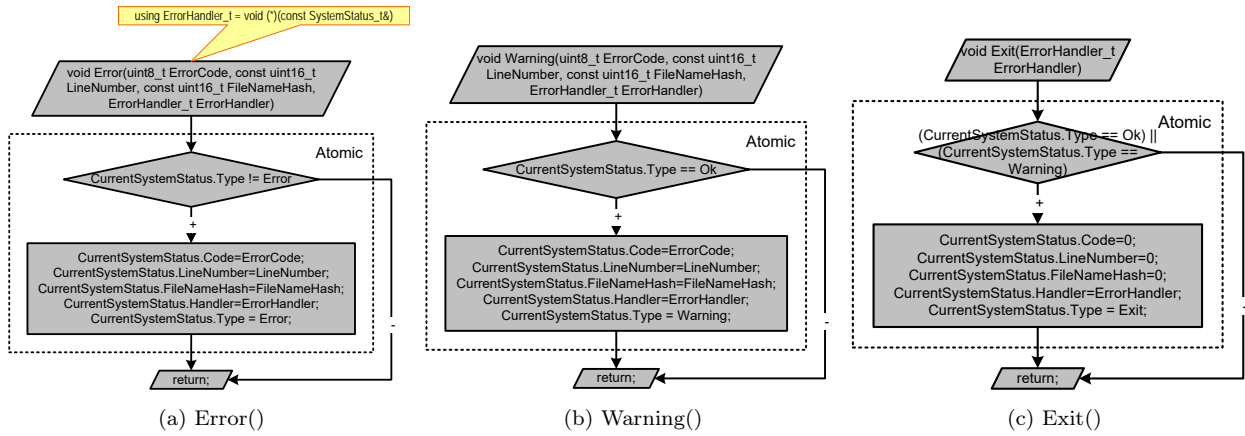


Figure 42: SystemStatus functions

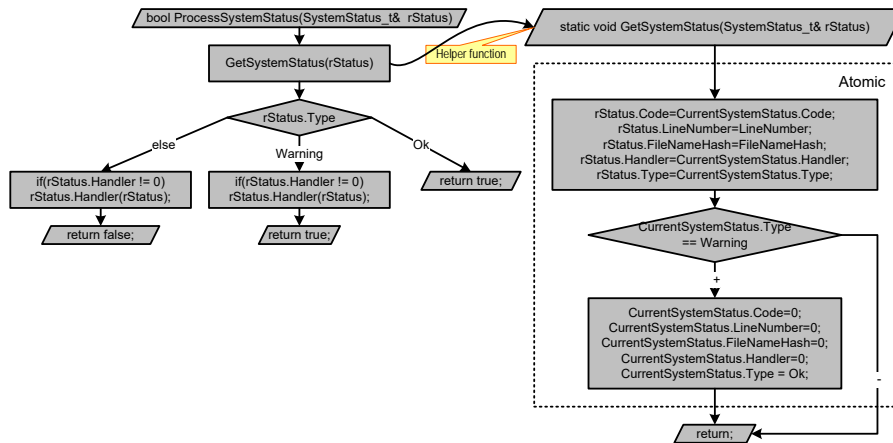


Figure 43: ProcessSystemStatus()

3.6 Error Handling (SystemStatus_t)

Structure SystemStatus_t and associated free functions declared in core/adx_SystemStatus.h are used for system error handling before the scheduler runs next Task. Table 5 summarizes SystemStatus header interfaces.

It is supposed that Error(), Warning() or Exit() is called in a place where error, warning or exit condition is encountered. The information about condition (type, source code line number, filename, application specific error/warning code and optional pointer to ErrorHandler) is passed to CurrentSystemStatus object. After Error(), Warning() or Exit() call one can continue execution current Task or “return” calling Wait() function. CurrentSystemStatus object will be processed when control returns to main(), before Scheduler execution. CurrentSystemStatus object is processed in ProcessSystemStatus() by the following way (fig 43):

1. If Type is Ok – just returns true (means proceed to scheduler);

Table 5: SystemStatus summary (core/adx_SystemStatus.h, adx_fsm::)

Name	Description
Types	
struct SystemStatus_t	Structure type for SystemStatus objects
enum SystemStatus_t::{Ok, Warning, Exit, Error}	Anonymous enum declared inside SystemStatus_t structure. Defines possible values for SystemStatus_t::Type field.
Global Objects	
static SystemStatus_t CurrentSystemStatus{0,0,0,0,adx_fsm:: SystemStatus_t::Ok};	Static (internal linkage) object. It stores status values passed by last Error(), Warning() or Exit() call. CurrentSystemStatus fields are cleared by ProcessSystemStatus() call.
Free Functions	
void Error(const uint8_t ErrorCode, const uint16_t LineNumber, const uint16_t FileNameHash, const ErrorHandler_t ErrorHandler = nullptr)	Sets CurrentSystemStatus.Type to Error (fig. 42a) that results in exit of the Sheduler's main() loop. Error can be handled in assigned ErrorHandler or in main() code after the loop.
void Warning(const uint8_t ErrorCode, const uint16_t LineNumber, const uint16_t FileNameHash, const ErrorHandler_t ErrorHandler = nullptr)	Sets CurrentSystemStatus.Type to Warning (fig. 42b). Warning can be handled by assigned ErrorHandler, then it returns to main() loop normal processing. It can be used in normal operation to indicate that something happened as well as for debugging and profiling purposes.
void Exit(const uint8_t ErrorCode = 0, const uint16_t LineNumber = 0, const uint16_t FileNameHash = 0, const ErrorHandler_t ErrorHandler = nullptr)	Sets CurrentSystemStatus.Type to Exit (fig. 42c) that results in exit of the Sheduler's main() loop. It can be treated as normal exit and handled after main() loop according to user application logic.
bool ProcessSystemStatus(SystemStatus_t& rStatus)	Copies CurrentSystemStatus object to local Status object passed by reference as a parameter and clears CurrentSystemStatus fields to their default values. If Status.Type is not Ok, calls assigned to status object Status.ErrorHandler() and then returns true if Status.Type is Ok or Warning, and false otherwise. This function is supposed to serve as an exit condition from main() loop. So it returns false when it's going to exit from Sheduler's loop, and true – to continue the loop and process next Event in the queue.

2. If Type is Warning – invokes user ErrorHandler() if assigned when calling Error() or Warning(), then returns true (proceed to scheduler);
3. Else (Error or Exit type) – invokes user ErrorHandler() if assigned when calling Error() or Exit(), then returns false (means break out of the scheduler's main() loop).

Error(), Warning() and Exit() can be used anywhere including ISR. These functions do not switch context and return as ordinary functions.

Declared types and aliases (adx_fsm namespace):

```
struct SystemStatus_t;
using ErrorHandler_t = void (*)(const SystemStatus_t&);
struct SystemStatus_t
{
    ErrorHandler_t ErrorHandler{ nullptr };
    uint16_t LineNumber{ 0 };
    uint16_t FileNameHash{ 0 };
    uint8_t ErrorCode{ 0 };
    enum : uint8_t
    {
        Ok,
        Warning,
        Exit,
        Error
    } Type{ Ok };
};
```

SystemStatus_t fields are described below:

ErrorHandler – Stores optional user error handler, which will be called in ProcessSystemStatus() function before scheduler if status Type is Warning or Error.

LineNumber – line number in source code where Error() or Warning() was originally called. It can be inserted automatically using built-in __LINE__ macros (see example below).

FileNameHash – 16-bit hash (CRC16_Modbus algorithm) of the source code file name in which Error() or Warning() was originally called.

ErrorCode – Error code specific to that source file. Error code is generally a static const, and should be unique only in scope of the file where Error() or Warning() is called.

Type – Type of Status: while processing Error will rewrite all types except Error. Warning will rewrite Ok. Exit will rewrite Ok and Warning.

Warning(), Error() and Exit() can be used the following way (example):

```
static const uint8_t Error_MyErrorName = 1; //Error description
static const uint16_t FileNameHash = CRC16_Modbus("source_file_name
.cpp");
static void MyErrorHandler(const SystemStatus_t& Status)
{
    //This function will be called in main() context before the
    Scheduler
    //Check Status and do something if required...
}
```

i If you need immediate critical error handling followed by the exit from main() loop, call Error() and then switch to main() using adx_kernel::SwitchToContext(&adx_kernel::SystemMainContext).

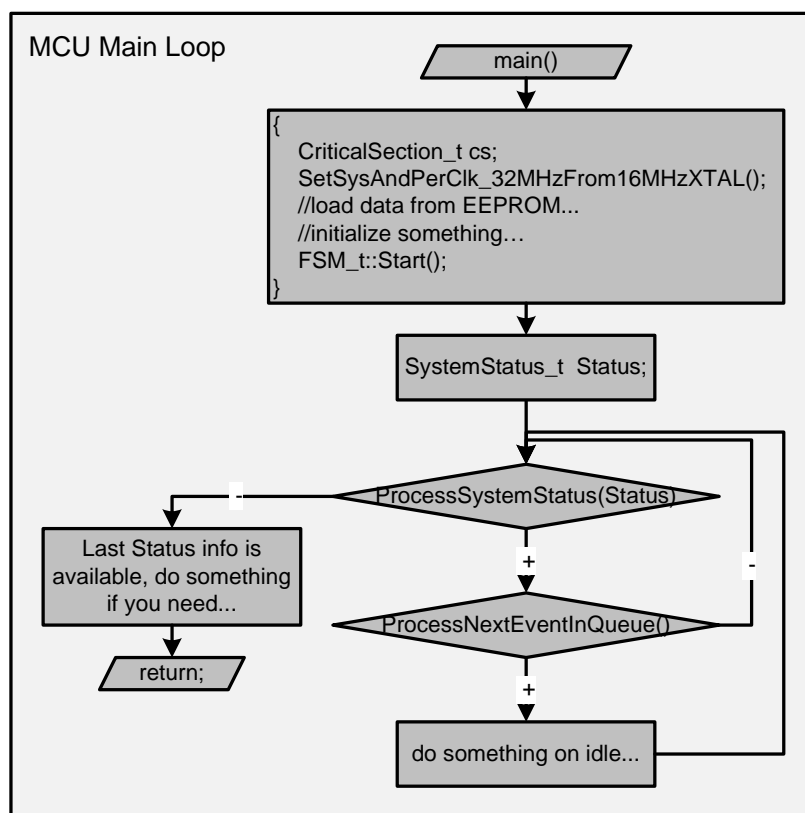


Figure 44: Main loop

```

//....
//Somewhere in the code:
Error(Error_MyErrorName, static_cast<uint16_t>(__LINE__),
      FileNameHash, MyErrorHandler);
//....

```

CRC16_Modbus() is a constexpr, so it has no implementation in MCU Flash, and executed at compile-time. Const string like "source_file_name.cpp" also will not be allocated in MCU memories, just 16-bit hash result. Do not place CRC16_Modbus() as a function parameter. Doing this in avr-gcc will result in run-time implementation, string allocation etc.

3.7 Scheduler (Main Loop)

main() function is an entry to application, it has its own context (SystemMain-Context) and stack allocated at the end of SRAM (ptrSystemMainStackBottom). main()'s stack size (SystemMainStack_SIZE) is defined in config/adx_kernel_config.h. Figure 44 shows typical main() logic. Its source code example is shown in main.cpp listing below:

```

#include "core/adx_Event.h"
#include "drivers/adx_SystemClock.h"

```

```

using namespace adx_fsm;
void MyTask() __attribute__((OS_task));
FSM_t MyFsm{ uint32_t(MyTask), 256 };
Event_t MyEvent{ &MyFsm };

void MyTask()
{
    while(true)
    {
        MyEvent.Send();
        WaitFor(&MyEvent);
    }
}

int main(void)
{
    //Enter CriticalSection
    CriticalSection_t cs;
    //////// Start of User Initialization Code //////////
    SetSysAndPerClk_32MHzFrom16MHzXTAL(); //Setup clock system
    //Connect empty Error Handler for Event system for debug
    purpose:
    Event_t::SetErrorHandler([](const SystemStatus_t& rStatus)
    { __nop(); });
    //////// End of User Initialization Code //////////
    FSM_t::Start();
    //Exit CriticalSection

    // Scheduler's Main Loop
    SystemStatus_t Status;
    while (ProcessSystemStatus(Status))
    {
        if (ProcessNextEventInQueue())
        {
            //Do something On Idle here...
            __nop(); //nop is for debug only
        }
    }
}

```

MyFsm and MyEvent objects are created and initialized by their ctors before main() entry. main() starts with CriticalSection_t ctor execution (which disables interrupts) followed by hardware and software system initialization which ends with FSM_t::Start() call that makes first run of each Task associated with corresponding FSM_t object. At the end of cs object scope its dtor is called which enables interrupts. After this system is ready for the entry to Scheduler's main loop. It starts with system status checking by ProcessSystemStatus(Status). It copies content of CurrentSystemStatus to Status local object (which can be processed by user logic after exit from loop) and returns true if no error or exit condition has happened. Next – one Event in the queue according to priority level will be processed by Scheduler if Event is available and maximum number of successive Event processings (i.e. Task calls) is not reached. Otherwise Idle block will be executed. Figure 45 shows ProcessNextEventInQueue() logic.

It consists of three stages – one for each priority event queue, starting from high priority. One ProcessNextEventInQueue() call result in only one Event processing or to Idle processing. Each stage has its own static counter that

i FSM_t::Start() is included in main()'s critical section because it makes first entry to the Tasks where their own hardware initialization can also be used at the beginning of the Task in its own critical section. If hardware resources are shared between Tasks, it's recommended to place initialization in main() since it runs before any Task entry.

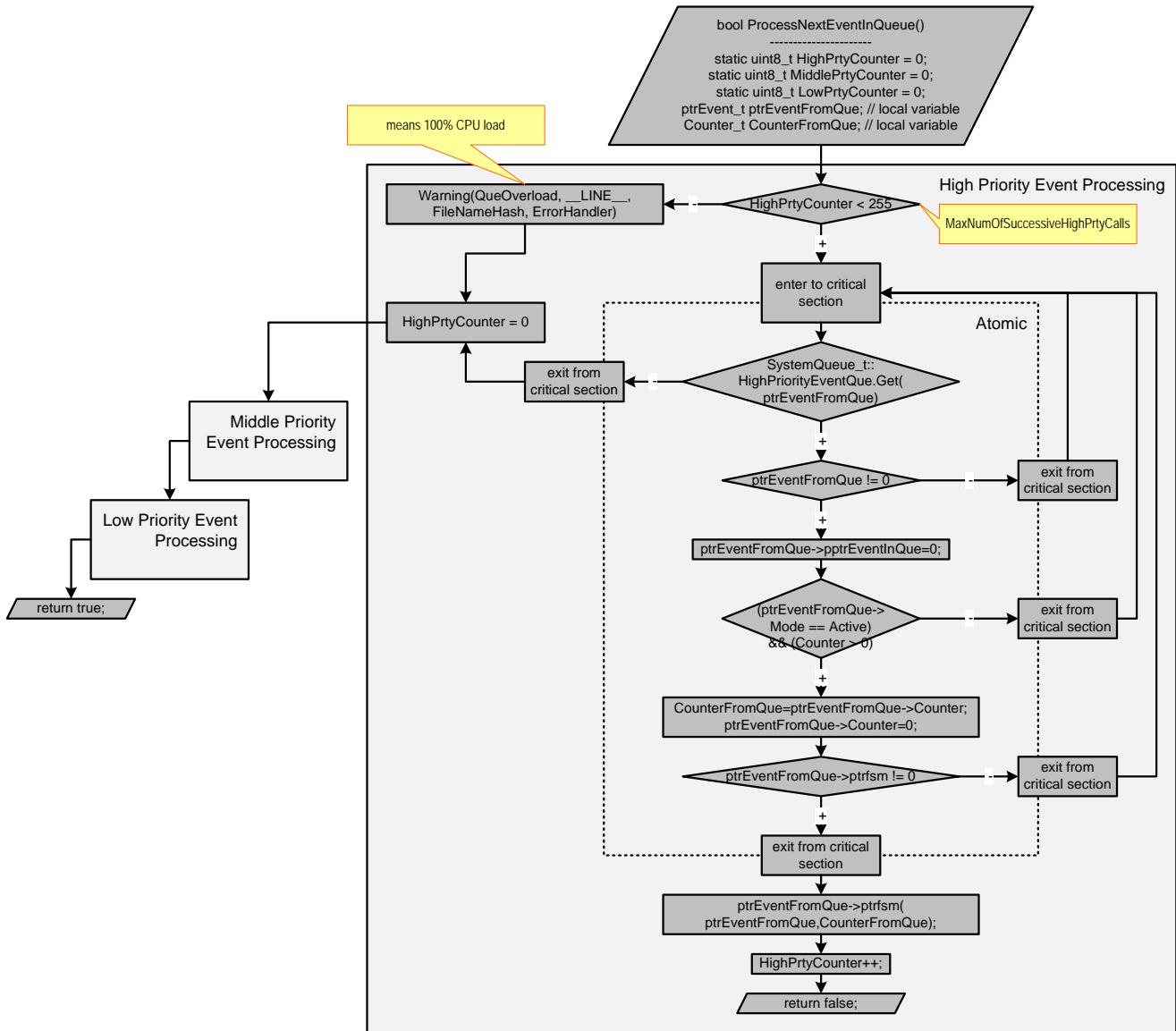


Figure 45: ProcessNextEventInQueue() Scheduler logic

increments when Event is successfully¹³ processed by this stage. If this stage event queue is empty, counter is cleared, and execution proceeds to next lower priority level stage. If counter is equal or greater than maximum allowed number, counter is also cleared, and execution proceeds to next lower priority level stage. Otherwise one Event is processed and function returns false. If all event queues are empty or maximum allowed number of successive Event processing is reached for the lowest priority stage, function returns true. It means that Idle block can be executed. This behavior allows to resolve some mutual blocking problems that overload event queues, e.g. when Task sends high priority Event to itself. It's not normal although. If any queue is always overloaded (always contain Active Event at each Scheduler's run) in most cases it means that something wrong with application design. Warning is generated if one of the event queues is overloaded, so SystemStatus ErrorHandler can be used for profiling. Event queue overloading as opposed to event queue overflow¹⁴, doesn't result in erroneous operation, it only means that probably event system is not efficiently used.

Maximum number of successive Event processing for each priority level is defined in config/adx_core_config.h.

3.8 Built-in Timer Events (TimerEvent_t)

Class TimerEvent_t is derived from Event_t base, and designed for applications where timeout functionality is required. Its behavior is similar to ordinary event, but it's generated (send) automatically in hardware timer ISR according to system timer tick settings and period value passed to particular TimerEvent_t object.

1. Time resolution is adjustable in range from 0.5ms to about 2.1s, and defined in core/adx_core_config.h
2. Continuous Time Events are supported, means that no tick is lost while TimerEvent processing until TimeEvent is set to Disabled state.
3. Number of ticks passed after previous TimerEvent processing is available by call of GetFsmCounter(), the same way as for ordinary Events.
4. One hardware TC is used for all TimerEvent_t objects, so time tick (resolution) is the same for all software timers.
5. Each TimerEvent_t object has it's own one software counter.
6. Software counter period can be adjusted dynamically for each TimerEvent_t event using SetPeriod() method.
7. Hardware timer (TC) interrupt is automatically disabled when all TimerEvents are in Disabled state for more efficient MPU use.

¹³Means that the following conditions are met: Event pointer got from the event queue is not nullptr, Event is Active and its Counter > 0.

¹⁴Event queue overflow occurs when there is not enough space in queue to send Event. It means that given priority event queue length must be increased according to the number of given priority Events in use. It should be noted that in most cases queue length is much less than total number of Events in the system, because Events usually are not sent all at one time.

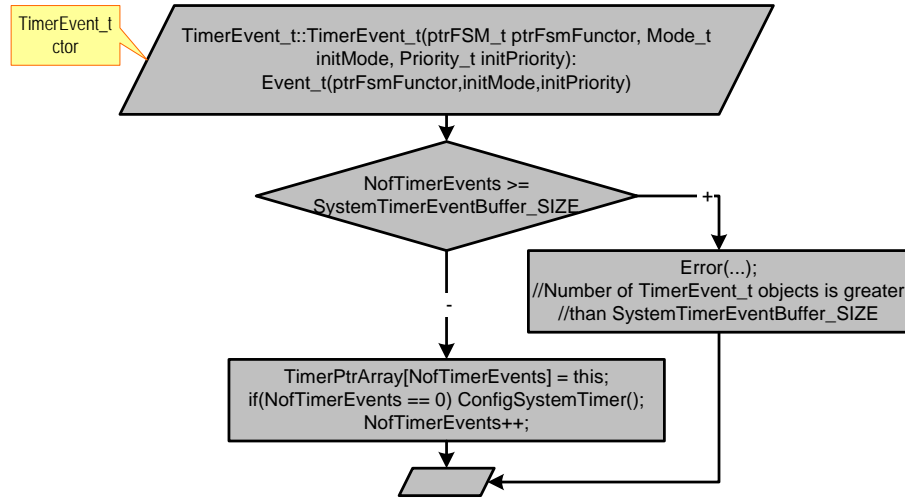


Figure 46: TimerEvent_t constructor logic

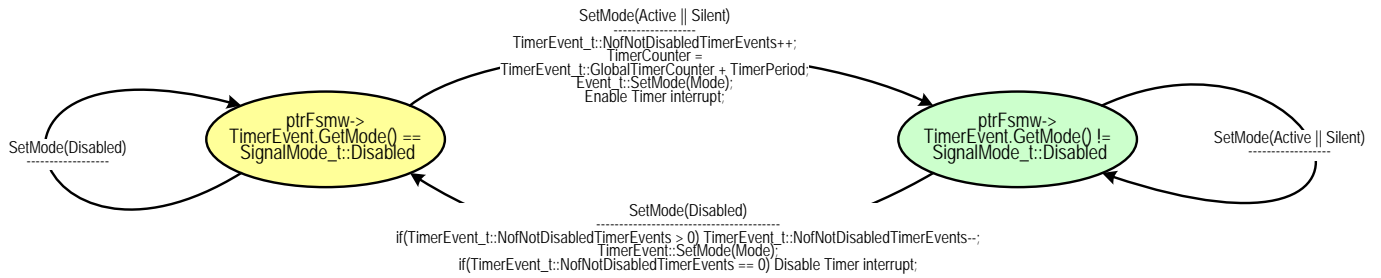


Figure 47: Timer Event states and transitions diagram

8. TimerEvent_t can use any hardware TC (Timer/Counter) available in MCU. Used TC and assigned interrupt priority level are defined in core/adx_core_config.h, default is TCF0, middle priority level.

Table 6 contains summary of TimerEvent_t class and related free functions. This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Figure 48 shows hardware timer/counter ISR logic. Its key feature is that only one global timer/counter is incremented in ISR. Software timers/counters are incremented only when their time expires the threshold defined by their period. Thus most time no math operation is performed per iteration in the loop. Another feature – is that timer interrupt is automatically disabled if all software timers are disabled. It's implemented in SetMode() logic. Other behavior of Timer Event is the same as ordinary Event.

The example of software timer/counter is listed below:

```

//MyTask.cpp
#include "core/adx_TimerEvent.h"
#include "drivers/adx_SystemClock.h"
using namespace adx_fsm;
  
```

Table 6: TimerEvent_t and related functions summary
(core/adx_TimerEvent.h, adx_fsm::)

Type/Name	Access	Description
Constructors		
TimerEvent_t(ptrFSM_t ptrFsmFuncion, Mode_t initMode = Mode_t::Disabled, Priority_t initPriority = Priority_t::Middle)	public	One time runs ConfigSystemTimer(). Calls base class ctor, initializes TimerPtrArray[] with TimerEvent_t object pointers, figure 46. Array is used instead of list for fast access to TimerEvent_t objects within TC ISR.
Methods		
virtual void SetMode(Mode_t)override final;	public	Sets Timer Event mode (Disabled, Silent, Active). The logic is described by diagram shown in figure 47. Set mode to Active before Wait(), and set to Silent or Disabled after Wait().
inline void SetPeriod(uint16_t Period);	public	Sets Timer period, after which Timer Event is sent. Period == 0 result in TimerEvent_t::Send() for each timer ISR call. Timer ISR call period is defined in core/adx_core_config.h
Static Methods		
void ConfigSystemTimer();	public	
Local Data Members		
uint16_t TimerPeriod{ 0 };	private	Stores period of TimerCounter.
uint16_t TimerCounter{ 0 };	private	TimerCounter used for Timer Event.
Static Data Members		
uint8_t NofTimerEvents = 0;	private	Equals to number of TimerEvent_t objects. Used as a counter of TimerEvent_t objects in ctor.
uint8_t NofNotDisabledTimerEvents = 0;	private	Number of TimerEvents in not Disabled state. If all TimerEvents are disabled, hardware timer interrupt is disabled.
uint16_t GlobalTimerCounter = 0;	private	Incremented by 1 at each timer ISR call. Used to check and increment software timer counters for each TimerEvent_t object.
TimerEvent_t* TimerPtrArray[SystemTimerEventBuffer_SIZE];	private	Array of Timer Event pointers. Used instead of the list for fast access by Timer ISR handler.
Free Functions		
void SleepFor(TimerEvent_t* ptrTimerEvent, uint16_t NofTicks);		Blocks Task for time equal to NofTicks * TickTime, where TickTime is period of hardware timer defined in config/adx_core_config.h. If NofTicks is 0, it sends TimerEvent (ptrTimerEvent->Send()) and calls Wait() to return control to Scheduler (or to "caller" Task if Signal was used to "call" Task).

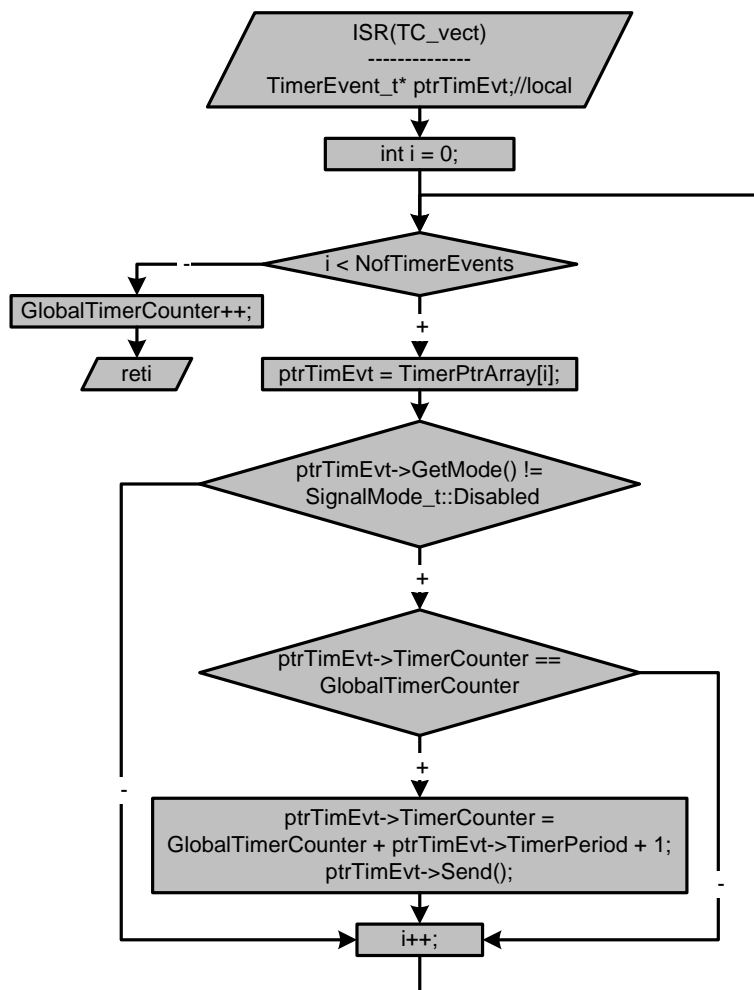


Figure 48: Hardware TC ISR logic

```

void MyTask() __attribute__((OS_task));
FSM_t MyFsm{ uint32_t(MyTask), 256};
TimerEvent_t MyTimerEvent{ &MyFsm };

void MyTask()
{
    volatile uint16_t MyCounter = 0;
    PORTB.DIRSET = PIN0_bm;
    while(true)
    {
        MyTimerEvent.SetPeriod(199); //period value can be adjusted
                                   any time
        MyTimerEvent.SetMode(SignalMode_t::Active);
        Wait();
        MyTimerEvent.SetMode(SignalMode_t::Silent);

        MyCounter += GetFsmCounter();
    }
}

```

This example implements lossless time counter, it means that for a long time period its value will be exactly equal to a number of counted time periods regardless of MPU loading and the fact is MyTask was processed in time or not. Actually it can be processed even once for all time, just before reading result counter value. The main requirement is that ISR has enough time to execute.

3.9 Fast Data Communication (DataSignal_t and DataEvent_t)

DataSignal_t<T> and DataEvent_t<T> classes are designed for fast data communication when data is encapsulated in Signal or Event object. DataSignal can be sent from main() or Task to Task, DataEvent can be sent from main(), Task and ISR to Task. When using DataEvent one should remember that if it was sent multiple times without implemented acknowledge synchronization mechanism, data will be overwritten, although consistent, because SetData()/GetData() methods are designed as atomic operations. Class inheritance is shown in figure 18 on page 29. For more information about user API, see section 4 on page 76.

3.10 FIFO With Notification (DataQueue_t)

Template class DataQueue_t<> declared in core/adx_DataQueue.h is designed as a part of Driver Model for AdxFSM. It is used for lossless data transfer between hardware and Task, as well as hardware interrupt and automatic Task notification. DataQueue_t object can be configured as a high-level pipeline read/write interface with polling or blocking capability from one side, and as a ready to use ISR handler from another side. Data type, buffer size, callback functor types for both r/w (put and get) sides are passed as template parameters. Connection of callback functors can be done via ctor parameters while object initialization or dynamically using “connect” methods which make it safely in runtime.

Table 7 contains summary of DataQueue_t<> template class and related free functions. DataQueue_t<> and related functions summary (core/adx_DataQueue.h, adx_fsm::)

Table 7: DataQueue_t<> and related functions summary
(core/adx_DataQueue.h, adx_fsm::)

Type/Name	Access	Description
Template Parameters		
typename ptrPutsideFuncorT		The pointer type of put side callback functor. It is used for notification that DataQueue is ready for Put()/Write() operations. Allowed types are: bool (*), i.e. free function with no parameters returning bool type, or pointer type of any class with defined bool operator(), like Signal_t* or Event_t*.
typename ptrGetsideFuncorT		The pointer type of get side callback functor. It is used for notification that DataQueue is ready for Get()/Read() operations. Allowed types are: bool (*), i.e. free function with no parameters returning bool type, or pointer type of any class with defined bool operator(), like Signal_t* or Event_t*.
uint8_t SIZE = 16		Length of FIFO buffer, default value is 16
typename DataT = uint8_t		Data type, default value is uint8_t
Constructors		
DataQueue_t(ptrPutsideFuncorT initPtrPutsideFuncor, ptrGetsideFuncorT initPtrGetsideFuncor, bool initPutsideWakeupFlag = false , bool initGetsideWakeupFlag = false)	public	initPtrPutsideFuncor – put side callback functor pointer, used for notification that DataQueue is ready for Put()/Write() operations. Can be nullptr, if put side notification is not used. initPtrGetsideFuncor – get side callback functor pointer, used for notification that DataQueue is ready for Get()/Read() operations. Can be nullptr, if get side notification is not used. initPutsideWakeupFlag – pass “true” if put side must be notified when first Get()/Read() operation is performed. Commonly used in drivers, where put side is an ordinary bool (*)() function that is used as driver’s final state machine. So this feature starts this machine, because FSM_t::Start() is not able to to this since driver’s fsm is not FSM_t object. initGetsideWakeupFlag – the same as above, but for get side.
Methods		
void ConnectPutsideFuncor (ptrPutsideFuncorT inPtrPutsideFuncor, bool inPutsideWakeupFlag) void ConnectGetsideFuncor (ptrGetsideFuncorT inPtrGetsideFuncor, bool inGetsideWakeupFlag)	public	Initializes local members used for notification logic, fig. on page 65. inPtrPutsideFuncor, inPtrGetsideFuncor, inPutsideWakeupFlag, inGetsideWakeupFlag – are the same as used in ctor.

(continued on next page)

(continued Table 7, beginning on page 62)

Type/Name	Access	Description
bool Put(const DataT& rData)	public	rData – input data passed by reference. Returns “true” if data was successfully put to the DataQueue, otherwise – “false”. Function is not blocking, figure on page 65 and on page 66.
bool Write(DataT* ptrData, uint8_t inSize, uint8_t& rinoutSizeWritten)	public	ptrData – pointer to the user data buffer from which data will be copied to DataQueue. inSize – user data buffer length (number of data in buffer). rinoutSizeWritten – number of data written. In/out parameter passed by reference. Before Write() call it contains start index of user buffer from which data should be copied to DataQueue. After Write() call it is incremented by number of data transferred, and contains start index for next Write() operation. It is implemented this way for convenient use in for/while loops. Function returns “true” if all (when rinoutSizeWritten == inSize) data was successfully put to the DataQueue, otherwise – “false”, fig. on page 67.
bool Get(DataT& rData)	public	rData – output data passed by reference. Returns “true” if data was successfully read from the DataQueue, otherwise – “false”. Function is not blocking, fig. on page 68.
bool Read(DataT* ptrData, uint8_t inSize, uint8_t& rinoutSizeRed)	public	Operates in similar way as Write(), see figure on page 69.
Local Data Members		
DataT data_array[SIZE];	private	Data array used for circular buffer, SIZE is defined by template parameter.
uint8_t PutIndex{ 0 };	private	Data array index, incremented at put/write operations.
uint8_t GetIndex{ 0 };	private	Data array index, incremented at get/read operations.
ptrPutsideFunctorT ptrPutsideFunctor{ nullptr };	private	Put side callback functor pointer, see ctor description.
ptrGetsideFunctorT ptrGetsideFunctor{ nullptr };	private	Get side callback functor pointer, see ctor description.

(continued on next page)

(continued Table 7, beginning on page 62)

Type/Name	Access	Description
bool PutsideWakeupFlag{ false };	private	Used in notification logic to make notification of put side after getting data and GetIndex increment, while the decision about notification is made before this. This flag is also influenced by notification functor return value. If notification was not successful (returned “false”) then WakeUpFlag remains “true”, that means the put side will be notified again at next Get()/Read() call. It is implemented for use in drivers for the cases when I/O register in spite of its generated IRQ may be not ready to write to for some reason.
uint8_t SizeRemainToWrite{ 0 };	private	Means data length which was not written to DataQueue due to Full buffer condition. Used in notification logic.
bool GetsideWakeupFlag{ false };	private	Used in notification logic to make notification of get side after putting data and PutIndex increment, while the decision about notification is made before this. This flag is also influenced by notification functor return value. If notification was not successful (returned “false”) then WakeUpFlag remains “true”, that means the get side will be notified again at next Put()/Write() call. It is implemented for use in drivers for the cases when I/O register in spite of its generated IRQ may be not ready to read from for some reason.
volatile uint8_t SizeRemainToRead{ 0 };	private	Means data length which was not read from DataQueue due to Empty buffer condition. Used in notification logic.
Static Data Members		
static const uint8_t Mask = SIZE – 1;	private	Mask imposed on PutIndex and GetIndex when accessing to data_array[] elements.
Template Free Functions		
inline void Write_bl(DataQueue_t<...>* ptrDataQueue, DataT* ptrDataBuf, uint8_t Size)		Write with blocking, see implementation on page 71.
inline void Read_bl(DataQueue_t<...>* ptrDataQueue, DataT* ptrDataBuf, uint8_t Size)		Read with blocking, see implementation on page 71.
inline void Put_bl(DataQueue_t<...>* ptrDataQueue, const DataT & rData)		Put with blocking, see implementation on page 71.

(continued on next page)

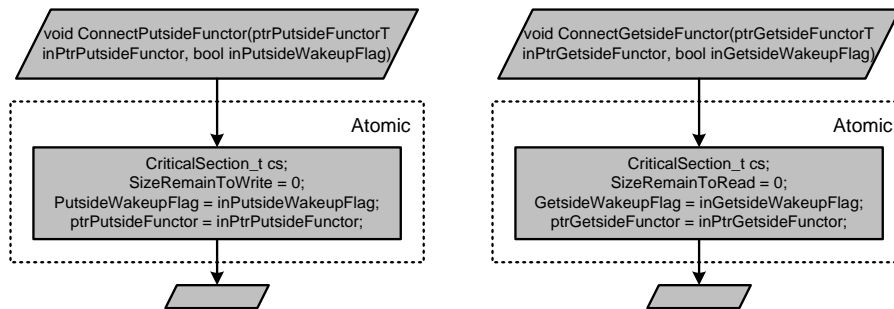


Figure 49: DataQueue Connect put/get side notification methods

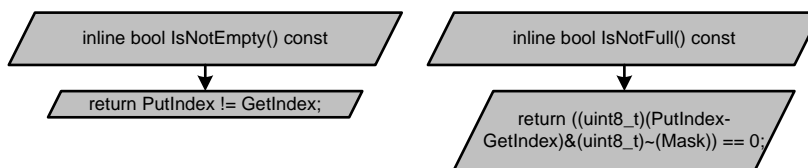


Figure 50: DataQueue Full/Empty criteria

(continued Table 7, beginning on page 62)

Type/Name	Access	Description
inline void Get_bl(DataQueue_t<...>* ptrDataQueue, DataT& rData)		Get with blocking, see implementation on page 71.

(continued on next page)

This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Block diagram shown in figure 55 on page 70 describes the behavior of DataQueue_t object. When both ptrPutsideFuncor and ptrGetsideFuncor are nullptr, DataQueue works the same way as an ordinary FIFO. Notification system is enabled for side “connected” to the functor object or callback function. For Task – it’s usually Event object, for ISR – free function invoked from its handler.

The put side notification:

Every time Get() or Read() is called the put side wake up condition is checked. If both conditions are met (1 AND 2):

1. the put side wants to write data, i.e. it tried to write/put data but not all data has been written because data buffer is full;
2. data which is going to be read by Get() or Read() frees enough space for the put side (i.e. it is going to read not less than SizeRemainToWrite)¹⁵

¹⁵This condition provides minimum number of notifications. It’s important for efficient use of Events as callback functors.

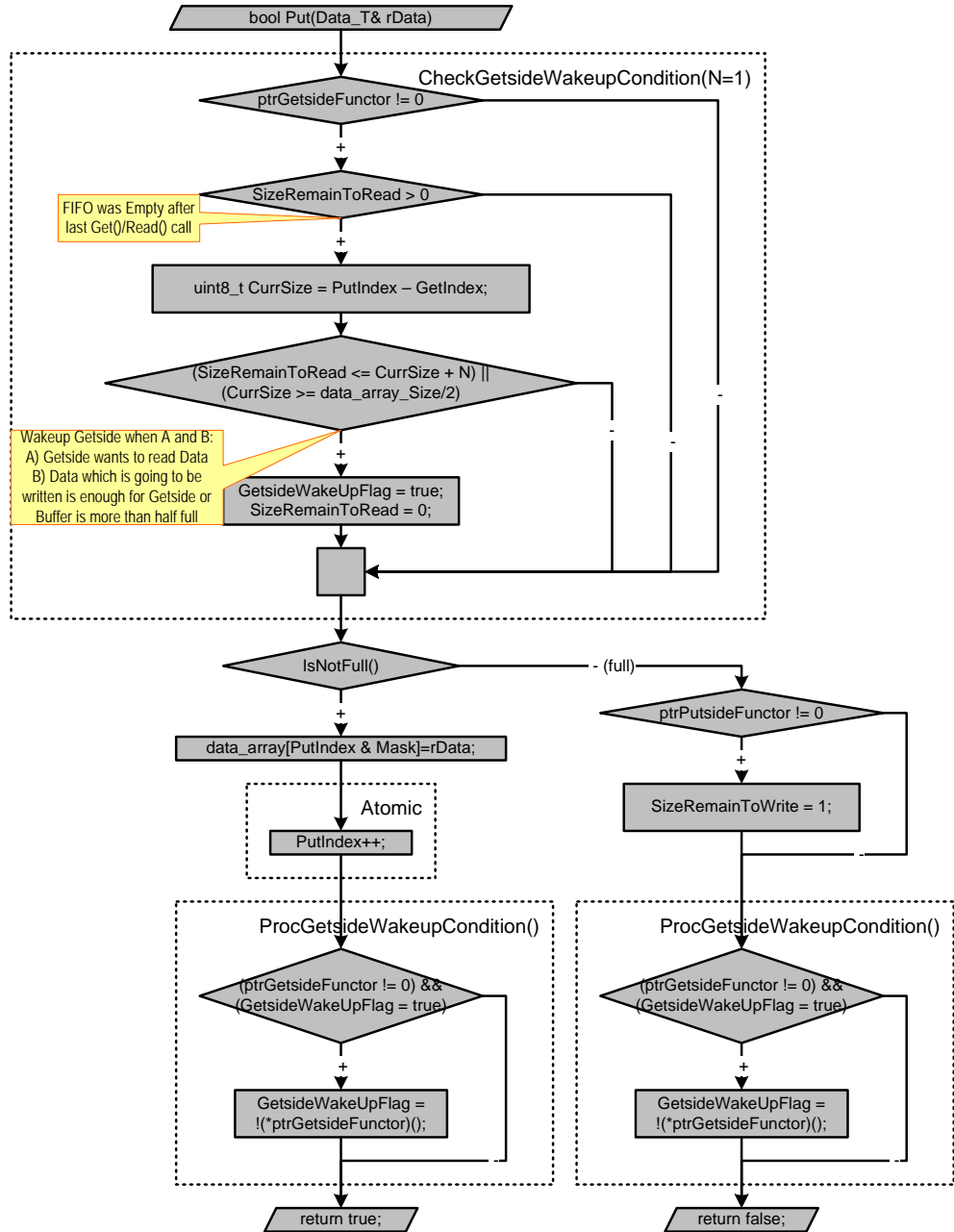


Figure 51: DataQueue_t::Put() logic

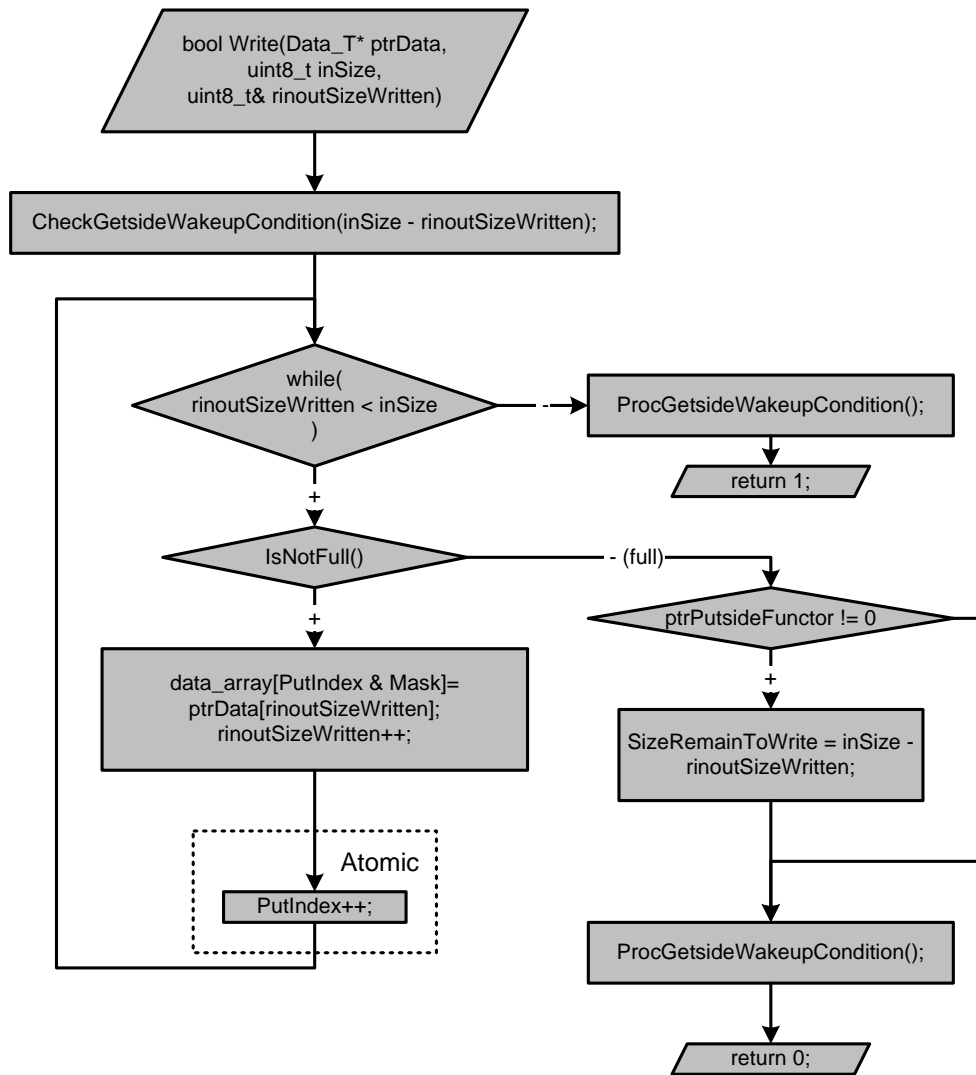


Figure 52: DataQueue_t::Write() logic

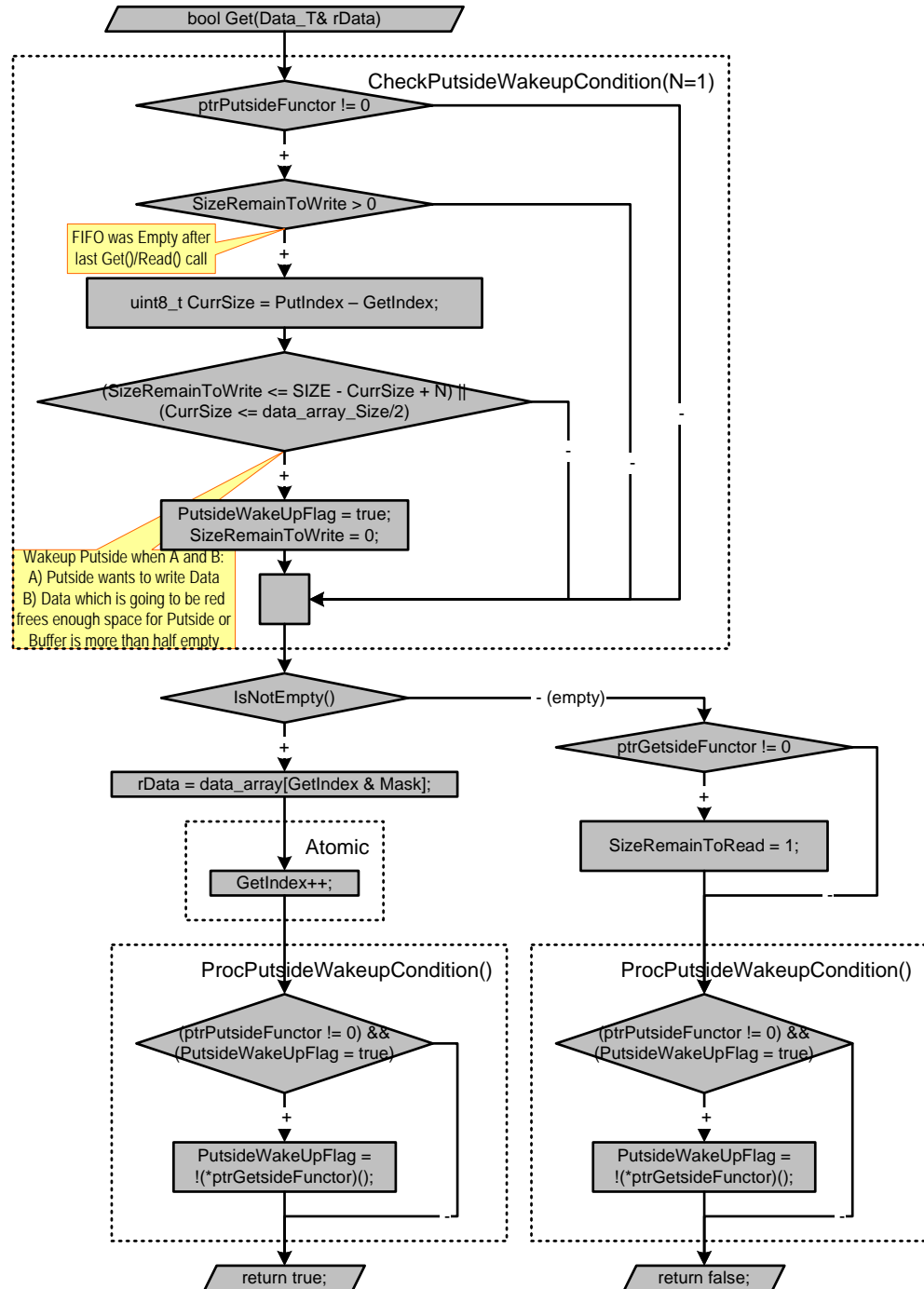


Figure 53: DataQueue_t::Get() logic

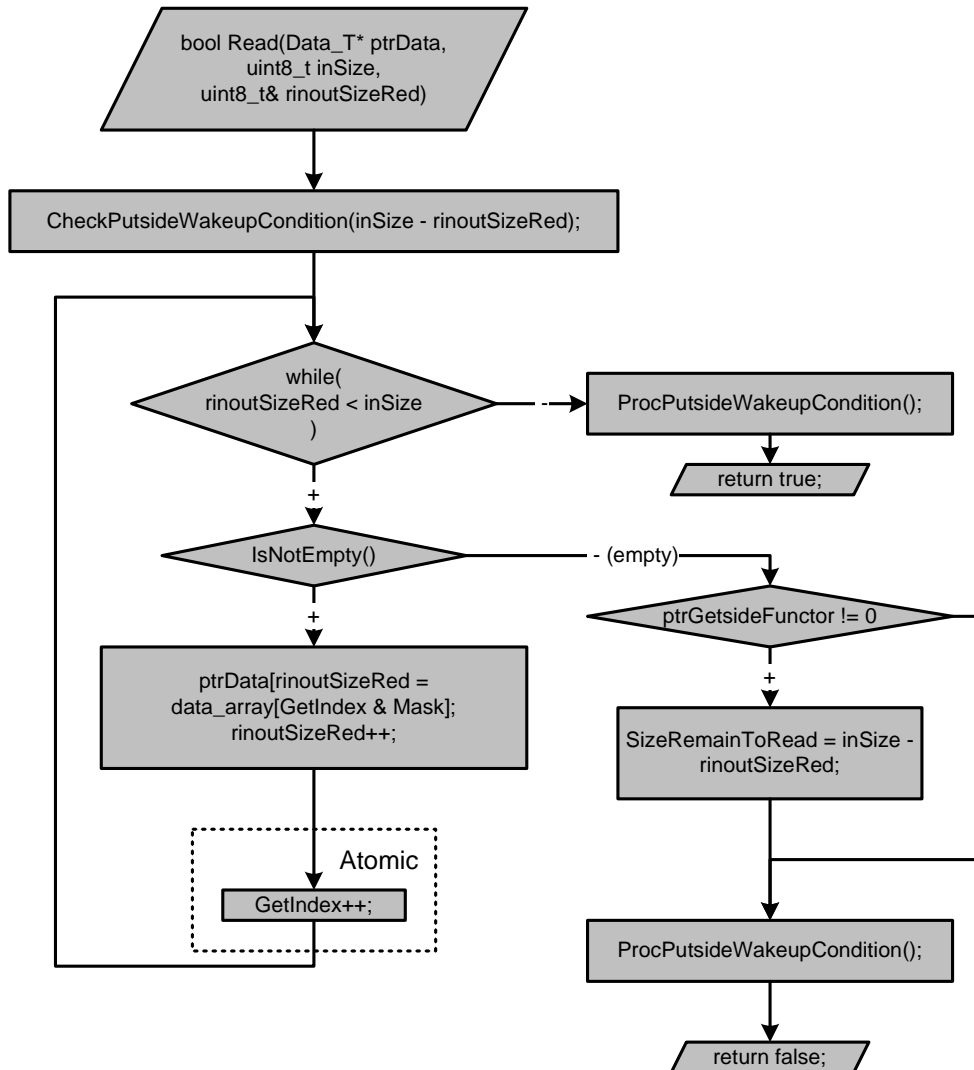


Figure 54: DataQueue_t::Read() logic

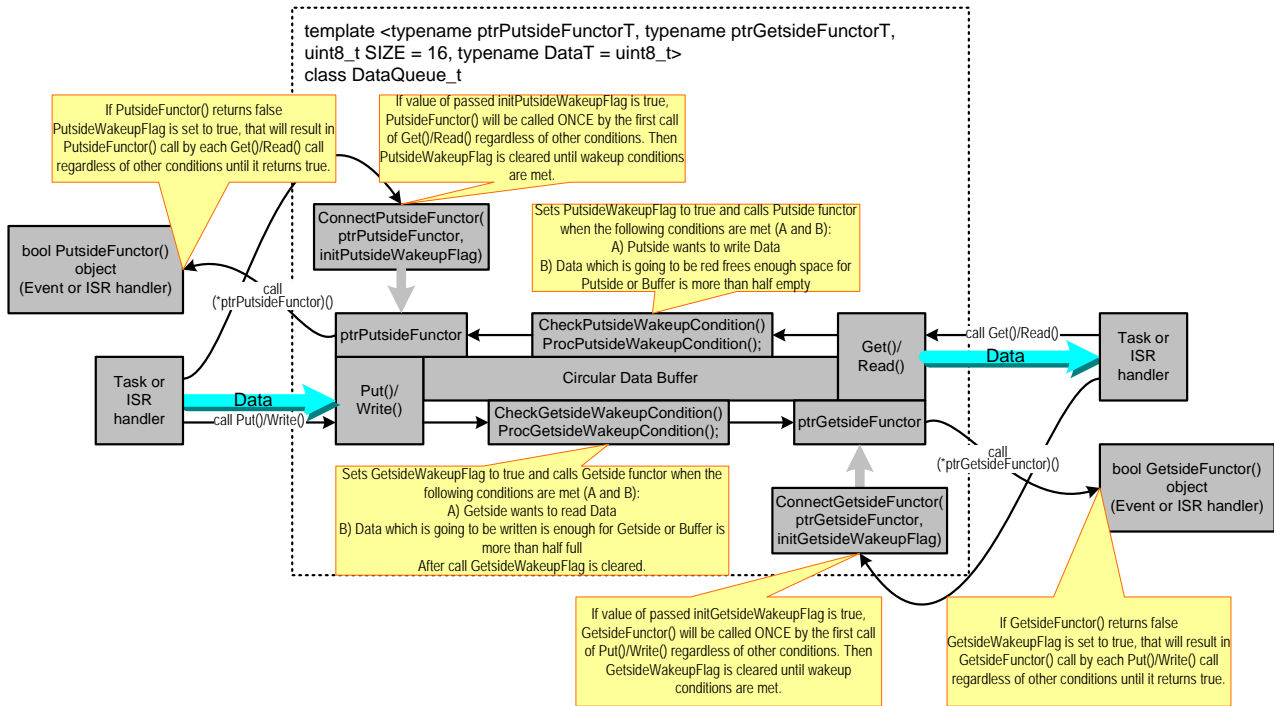


Figure 55: DataQueue_t<> Block Diagram

OR data buffer is more than half empty¹⁶;

then PutsideWakeupFlag is set “true”, data is read from the buffer, then the put side functor is called, and PutsideWakeupFlag cleared to “false” (if notification was successful).

The get side notification:

Every time Put() or Write() is called the get side wake up condition is checked. If both conditions are met (1 **AND** 2):

1. the get side wants to read data, i.e. it tried to read/get data but not all requested data has been read because data buffer is empty;
2. data amount which is going to be written by Put() or Write() is enough for the get side (i.e. it is going to write not less than SizeRemainToRead) **OR** data buffer is more than half full¹⁷;

then GetsideWakeupFlag is set “true”, data is written to the buffer, then the get side functor is called, and GetsideWakeupFlag cleared to “false” (if notification was successful).

¹⁶This condition is added to fill the buffer before it becomes empty. It's important in some interrupt driven data processing applications when data must be available in the buffer each time the Get()/Read() is called.

¹⁷This condition is added to empty the buffer before it becomes full. It's important in some interrupt driven data processing applications when buffer must have enough free space for each time the Put()/Write() is called.

To simplify blocking r/w operations special function template is implemented. Function Template parameter Derivation in C++ result in function overloading, thus derived function with resolved signature will look like a simple function and can be called like this: Write_bl(&MyDataQueueObject, ptrDataBuf, BufSize); R/W functions templates for use with DataQueue_t<> object with Event as a notification functor is listed below:

```
//Universal call R/W interface wrappers with blocking for
ptrEvent_t
template <typename ptrGetsideFuncT, uint8_t SIZE, typename DataT
>
inline void Write_bl(DataQueue_t<ptrEvent_t, ptrGetsideFuncT, SIZE
, DataT>* ptrDataQueue, DataT* ptrDataBuf, uint8_t Size)
{
    uint8_t SizeWritten = 0;
    (ptrDataQueue->GetPutsidePtrFuncT())->SetMode(SignalMode_t::
        Active);
    while(SizeWritten < Size) if( !ptrDataQueue->Write(ptrDataBuf,
        Size, SizeWritten) ) Wait();
    (ptrDataQueue->GetPutsidePtrFuncT())->SetMode(SignalMode_t::
        Silent);
}
template <typename ptrPutsideFuncT, uint8_t SIZE, typename DataT
>
inline void Read_bl(DataQueue_t<ptrPutsideFuncT, ptrEvent_t, SIZE,
    DataT>* ptrDataQueue, DataT* ptrDataBuf, uint8_t Size)
{
    uint8_t SizeRed = 0;
    (ptrDataQueue->GetGetsidePtrFuncT())->SetMode(SignalMode_t::
        Active);
    while(SizeRed < Size) if( !ptrDataQueue->Read(ptrDataBuf, Size,
        SizeRed) ) Wait();
    (ptrDataQueue->GetGetsidePtrFuncT())->SetMode(SignalMode_t::
        Silent);
}
template <typename ptrGetsideFuncT, uint8_t SIZE, typename DataT
>
inline void Put_bl(DataQueue_t<ptrEvent_t, ptrGetsideFuncT, SIZE,
    DataT>* ptrDataQueue, const DataT& rData)
{
    (ptrDataQueue->GetPutsidePtrFuncT())->SetMode(SignalMode_t::
        Active);
    while( !ptrDataQueue->Put(rData) ) Wait();
    (ptrDataQueue->GetPutsidePtrFuncT())->SetMode(SignalMode_t::
        Silent);
}
template <typename ptrPutsideFuncT, uint8_t SIZE, typename DataT
>
inline void Get_bl(DataQueue_t<ptrPutsideFuncT, ptrEvent_t, SIZE,
    DataT>* ptrDataQueue, DataT& rData)
{
    (ptrDataQueue->GetGetsidePtrFuncT())->SetMode(SignalMode_t::
        Active);
    while( !ptrDataQueue->Get(rData) ) Wait();
    (ptrDataQueue->GetGetsidePtrFuncT())->SetMode(SignalMode_t::
        Silent);
}

```

3.11 Recursive Mutex (RecursiveMutex_t)

Class RecursiveMutex_t declared in core/adx_RecursiveMutex.h is designed as a part of Driver Model of AdxFSM. It is used in Open()/Close() methods of user driver classes for exclusive access to shared hardware resources.

1. Two versions of Lock are implemented: TryLock() – without Task blocking and Lock_bl() – that will block a Task if mutex is already locked by another Task. If mutex is not locked yet, it returns “true” immediately.
2. Lock is binded to fsm object address (Task), so Lock() call can be nested and used with the different Events assigned to the same fsm (Task). It will be resolved as multiple Lock() call with the same Event and will return immediately with “true” value, if previous Lock() call has been successful.
3. If TryLock(Signal_t*) returned false, don’t call TryLock(Signal_t*) until the assigned Event is “received” by Wait(). For polling TryLock(void) can be used.
4. Each successful Lock() call (i.e. when TryLock() return true, or Lock_bl() returns control to Task) should be unlocked with Unlock() call.
5. Unlock() call always returns immediately (not Task blocking).
6. Mutex may work with Signals but generally used with Events to prevent loops.
7. Blocked Tasks are notified when mutex is unlocked with the same order as they were trying to lock. Actual order of Task call (by scheduler) depends on the Priority of used Event.

Table 8 on page 74 contains summary of RecursiveMutex_t class and related free functions. This table contains only members that are essential for understanding its internal logic, for more details about class user interfaces see section 4.

Figure 56 on the next page shows mutex states and transitions diagram that defines its behavior. Mutex stores lock requests in its own event queue. Its length is set by RecursiveMutexSignalQueue_SIZE constant in config/adx_core_config.h. If number of lock requests exceeds this value, mutex begins to send the events in system event queue storing them this way. It will result in spur wake up of the Tasks waiting for the lock, so it it’s not efficient, but it resolves mutex event queue overflow problem. Anyway it’s not a normal operation, in most cases mutex event queue length should be enough for efficient lock request handling.

3.12 Rx/Tx Driver Model

High-level AdxFSM Driver Model is based on DataQueue_t<> and RecursiveMutex_t objects. Event_t objects are used as connected to driver functors to notify Task, and connected user functor object is used for driver’s fsm notification.

1. Open/Close are implemented with aid of mutex, Read/Write operations are performed directly to/from DataQueue_t<> objects.
2. Both polling and blocking read/write operations are supported.

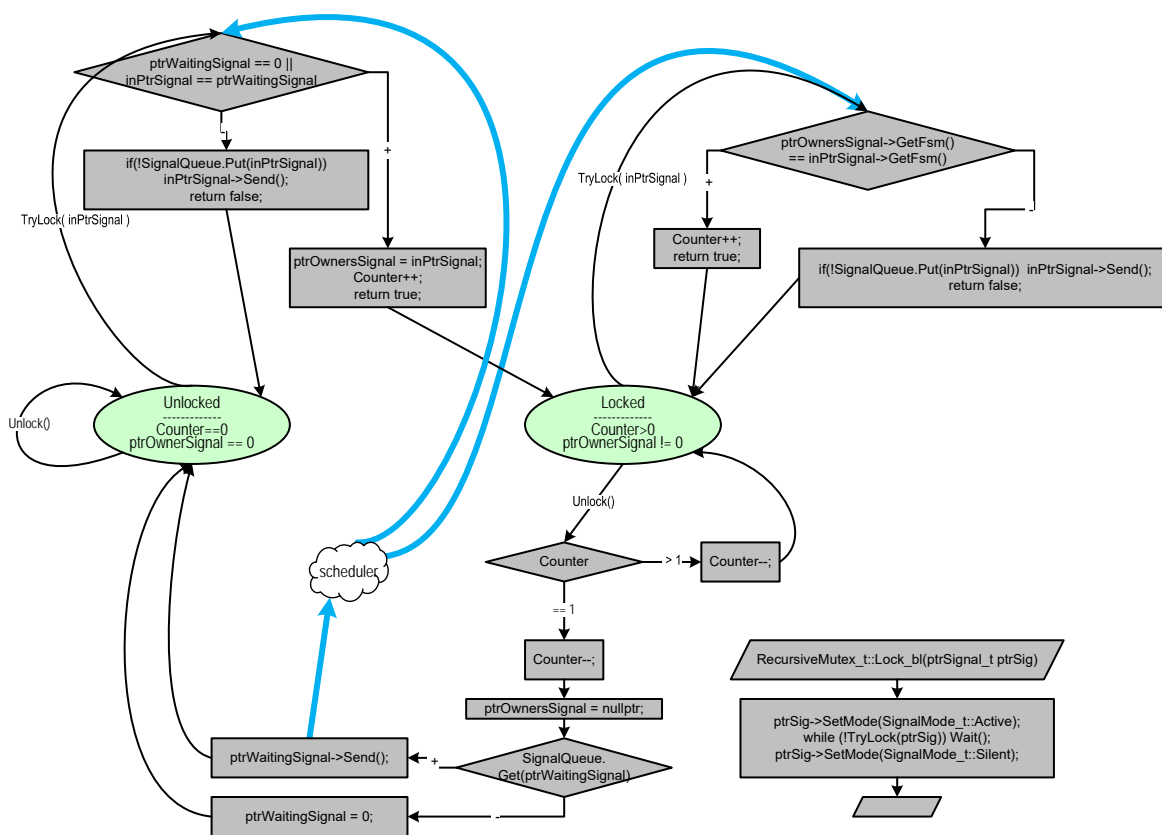


Figure 56: RecursiveMutex_t states and transitions diagram

Table 8: RecursiveMutex_t and related functions summary (core/adx_RecursiveMutex.h, adx_fsm::)

Type/Name	Access	Description
Constructors		
RecursiveMutex_t()	public	Default constructor. Copy and assignment constructors are deleted.
Methods		
bool TryLock(ptrSignal_t inPtrSignal)	public	Non-blocking lock method. If it returns “true”, mutex is locked, otherwise one should wait for Event, passed as a parameter, and try it again.
void Lock_bl(ptrSignal_t inPtrSignal)	public	Blocking lock method. When it returns, mutex is locked.
void Unlock()	public	Unlocks mutex.
Local Data Members		
SystemQueue_t< RecursiveMutexSignalQueue_SIZE, ptrSignal_t> SignalQueue;	private	Stores Event pointers of not successful Lock() calls. These Events are used to notify Tasks, when mutex is unlocked.
ptrSignal_t ptrOwnersSignal{ nullptr };	private	Stores owner’s Event.
ptrSignal_t ptrWaitingSignal{ nullptr };	private	Stores Event that was sent to notify Task, that mutex is unlocked.
uint8_t Counter{ 0 };	private	Stores counter of successive Lock() calls by the same Task.

3. For user-friendly configurable hardware resources implementation, preprocessor macros can be used in Driver object and ISR() definition section of code (see UsartDriver_t implementation and use example).
4. Multiple Driver objects can be created, provided that each object instance is configured to use different hardware resources.

Figure 57 on the facing page shows block diagram of the model. ISR related logic is shown in figure 58 on the next page. Open/Close methods are shown in figure 59 on page 76.

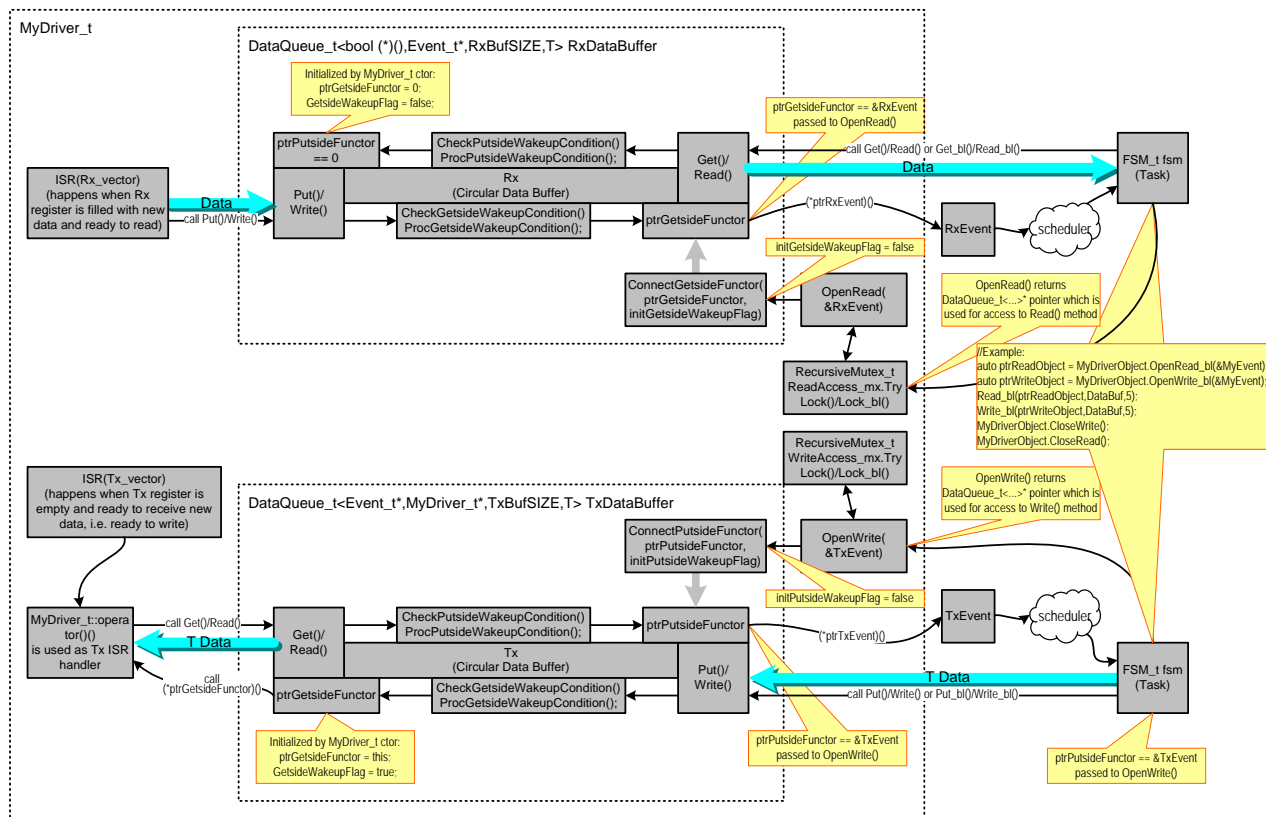


Figure 57: AdxFSM high-level Rx/Tx Driver Model block diagram

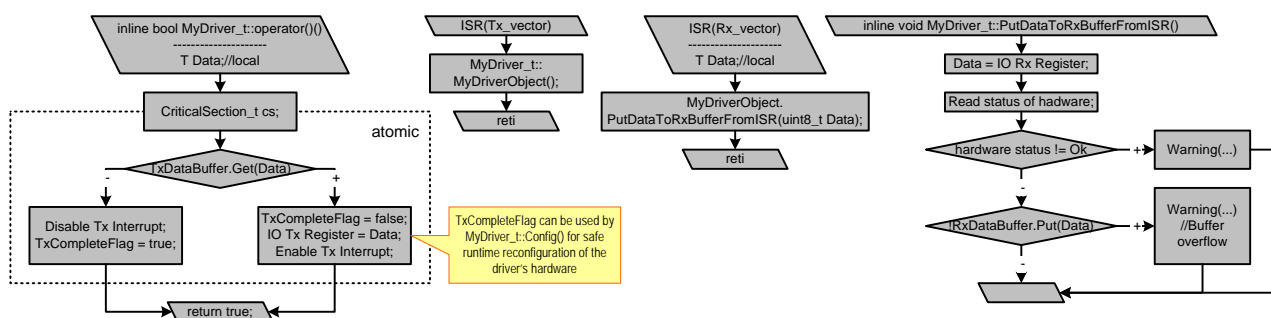


Figure 58: Driver Model ISR related logic

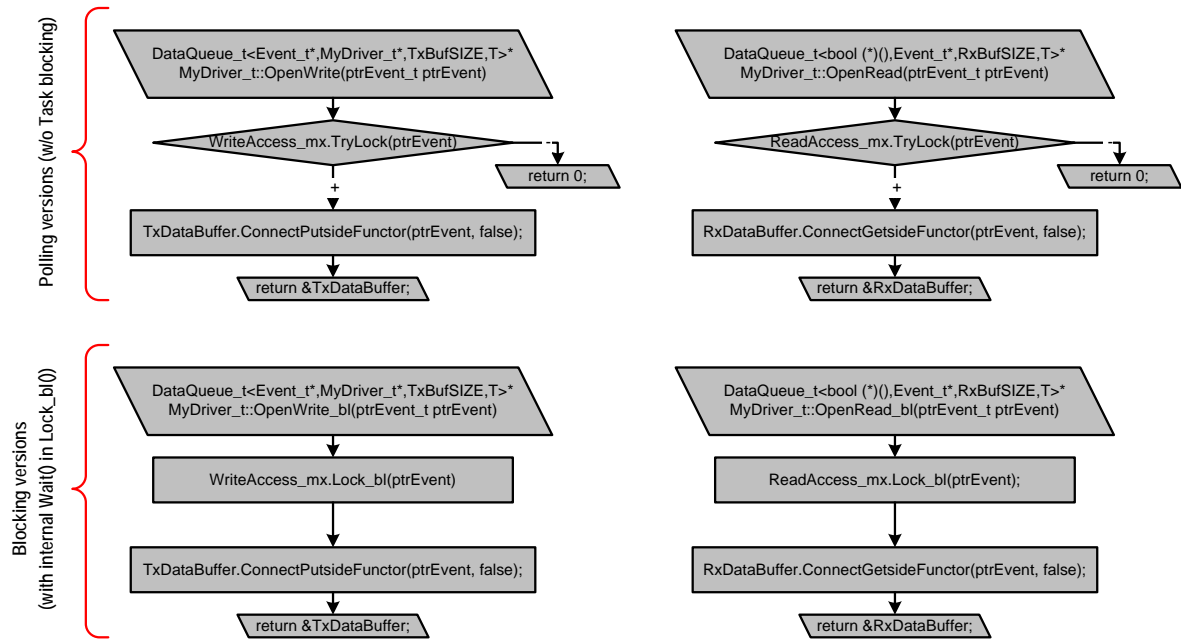


Figure 59: Driver Model Open/Close methods



User API interfaces listed below are in `adx_fsm` namespace, unless noted.

4 AdxFSM API

4.1 core/adx_SystemStatus.h

4.1.1 Types and Aliases

SystemStatus_t – structure type for system status object used in `ProcessSystemStatus()`. Declaration is listed in section 3.6 on page 53, use example is listed in section 3.7 on page 54.

ErrorHandler_t – type of function pointer to error handler callback function. Pointer of this type (in most case it's just a name of function declared as `void MyErrorHandler(const SystemStatus_t& rStatus);`) is passed to `Error()`, `Warning()` and `Exit()` as input parameter. Declaration is listed in section 3.6 on page 53, use example is listed in section 3.6 on page 53.

4.1.2 Functions

Error() – sends error message, that is handled in `ProcessSystemStatus()`. Logic is shown in figure 42a on page 51, the use example is listed in section 3.6 on page 53.

Declaration:

```

void Error(const uint8_t ErrorCode, const uint16_t LineNumber,
          const uint16_t FileNameHash, const ErrorHandler_t ErrorHandler
          = nullptr);

```

Input parameters:

ErrorCode – error code specific to the source file where it is used. It corresponds to the `SystemStatus_t::ErrorCode` field described in section 3.6 on page 53.

LineNumber – line number of the source file where it is used. Inserted automatically by `static_cast<uint16_t>(__LINE__)`. It corresponds to the `SystemStatus_t::LineNumber` field described in section 3.6 on page 53.

FileNameHash – used as short “alias” of the source file name where it is used. It corresponds to the `SystemStatus_t::FileNameHash` field described in section 3.6 on page 53. `FileNameHash` value can be generated with aid of constexpr `CRC16_Modbus()`.

ErrorHandler – pointer to user error handler which is automatically called in `ProcessSystemStatus()` if `Error()`, `Warning()` or `Exit()` was called before. If this feature is not used, omit this parameter.

Return value: none.

Behavior: returns immediately as an ordinary function, results in deferred call of user error handler in `main()`’s context, and exit of Scheduler’s `main()` loop.

Scope: can be called from Task, ISR, `main()`.

Warning() – sends warning message, that is handled in `ProcessSystemStatus()`. Logic is shown in figure 42b on page 51, the use example is listed in section 3.6 on page 53.

Declaration:

```
void Warning(const uint8_t ErrorCode, const uint16_t LineNumber,
            const uint16_t FileNameHash, const ErrorHandler_t ErrorHandler
            = nullptr);
```

Input parameters: the same as for `Error()`.

Return value: none.

Behavior: returns immediately as an ordinary function, results in deferred call of user error handler in `main()`’s context.

Scope: can be called from Task, ISR, `main()`.

Exit() – sends exit message, that is handled in `ProcessSystemStatus()`. Logic is shown in figure 42c on page 51, the use example is listed in section 3.6 on page 53.

Declaration:

```
void Exit(const uint8_t ErrorCode = 0, const uint16_t LineNumber =
          0, const uint16_t FileNameHash = 0, const ErrorHandler_t
          ErrorHandler = nullptr);
```

Input parameters: the same as for `Error()`.

Return value: none.

Behavior: returns immediately as an ordinary function, results in deferred call of user error handler in `main()`’s context, and exit of Scheduler’s `main()` loop.

Scope: can be called from Task, ISR, `main()`.

ProcessSystemStatus() – handles messages which were sent by `Error()`, `Warning()` and `Exit()`. This function is supposed to be a part of `main()` loop as shown in figure 44 on page 54. The use example is listed in section 3.7 on page 54, internal logic is shown in figure 43 on page 51.

Declaration:

```
bool ProcessSystemStatus(SystemStatus_t& rStatus);
```

Input parameters:

`rStatus` – output local status object (structure). System status message is copied to this status object. It can be handled after loop exit if necessary.

Return value: returns “true” if no error or exit message is sent, otherwise – “false”.

Behavior: copies system status message to passed by reference object, calls user error handler, assigned to error, warning or exit message, and returns bool value. “True” – means proceed in the loop, “false” – exit from the loop. For more details see table 5 on page 52.

Scope: `main()`.

CRC16_Modbus() – used for compile-time `FileNameHash` calculation, the use example is listed in section 3.6 on page 53.

Declaration/definition:

```
constexpr uint16_t CRC16_Modbus(const char* str) { ... }
```

Input parameters: zero ended const string.

Return value: 16-bit constant value calculated according CRC-16 Modbus algorithm (poly – 0x8005, Init – 0xffff, RefIn – true, RefOut – true, xorOut – 0x0000, check value (“123456789”) = 0x4b37).

Behavior: calculates return value at compile-time if input parameter is constant.

Scope: global scope¹⁸ at constant definition.

4.2 core/adx_fsm.h

4.2.1 Types and Aliases

ptrSignal_t – pointer to `Signal` type, declared as

```
using ptrSignal_t = Signal_t*;
```

SignalCounter_t – `Signal` counter type, declared as

```
using SignalCounter_t = uint16_t;
```

ptrFSM_t – pointer to `FSM` type, declared as

```
using ptrFSM_t = FSM_t*;
```

¹⁸Do not place `CRC16_Modbus()` as a parameter to `Error()`, `Warning()` or `Exit()`, it will result in run-time implementation in `avr-gcc`.

Task_t – pointer to Task function type, declared as

```
using Task_t = void (*)( );
```

4.2.2 FSM_t class methods

FSM_t class – wrapper for Task function. Used for automatic initialization of associated to Task objects. For more details see section 3.2 on page 35.

Ctor – initializes FSM object.

Declaration:

```
FSM_t(uint32_t TaskAddress, uint16_t TaskStackSize);
```

Input parameters:

TaskAddress – address of the Task associated to FSM object. To pass the parameter **static_cast<uint32_t>(MyTask)** can be used for ordinary near address. Task can be declared and defined as in the example listed below:

```
void MyTask() __attribute__((OS_task));
//...
void MyTask()
{
    //...
}
//or declaration and definition at the same place:
__attribute__((OS_task))
void MyTask2()
{
    //...
}
```

For more details see section 3.1 on page 30.

TaskStackSize – Task's stack size in bytes. Depends on size of local variables and nested function calls used in Task. For empty Task it is 3 bytes. For real application one can start with 128 bytes, then reduce it according to debugging results. For more details, see section 3.1 on page 27.

Scope: global scope at FSM object definition.

Start() – used for the first run of the Tasks before Sheduler's main() loop, see figure 44 on page 54 and section 3.7 on page 54.

Declaration:

```
static void Start();
```

Behavior: runs each Task (switches context), nullptr and 0 value are passed to FSM Task as pointer to Signal and Signal counter parameter accordingly.

Scope: main().

GetFsmID() – returns FSM (Task) unique ID value assigned automatically by FSM_t ctor, starting 0.

Declaration:

```
inline uint8_t GetFsmID() {...}
```

Return value: in range 0 to N-1, where N – is a number of FSM objects, i.e. Tasks.

Behavior: returns ID value immediately.

Scope: Task (can be called from Task using GetRunningFsm()→GetFsmID()); if FSM name or address is not accessible), also can be called from main() or ISR via object name or pointer.

4.2.3 Functions

GetFsmSignal() – returns pointer to the Signal (or Event) which was used to wake up the Task. Can be applied when multiple transitions from one waiting state are used: **if**(GetFsmSignal() == &MyEvent){...}.

Declaration:

```
inline ptrSignal_t GetFsmSignal() {...}
```

Return value: pointer to Signal_t object (base for Event_t).

Behavior: returns address of Signal (or Event) immediately. Returns nullptr at first Task run (when Task is called by FSM_t::Start()).

Scope: Task only.

GetFsmCounter() – returns Signal's (or Event's) Counter value. It equals to a number of times the Signal or Event was sent since previous Task wake up by this Signal or Event or since the last Signal/Event activation (SetMode to Silent or Active), depending on which is later.

Declaration:

```
inline SignalCounter_t GetFsmCounter() {...}
```

Return value: value in range from 1 to SignalCounter_t type max value (65535).

Behavior: returns Signal/Event counter value immediately. Returns 0 at first Task run (when Task is called by FSM_t::Start()). When counter reaches its maximum value, warning is sent to system status, and it will stay at its maximum until the Task is waked up by the Signal/Event, when it's passed to FSM and then cleared. For more details about counter behavior see description in section 3.3 on page 38 and section 3.5 on page 44.

Scope: Task only.

GetRunningFsm() – returns pointer to the running FSM object. Used to get access to FSM object interfaces from running Task which is assigned to this FSM.

Declaration:

```
inline ptrFSM_t GetRunningFsm() {...}
```

Return value: pointer to FSM type.

Behavior: returns address of running FSM object immediately.

Scope: Task only.

Wait() – waits for any Signal or Event assigned to the FSM (Task) from where it is called.

Declaration:

```
inline void Wait() { ... }
```

Behavior: returns to “caller’s” context, e.g. if Task was “called” by Sheduler or FSM_t::Start(), it will switch to main()’s Context, if the Task was “called” by another Task using Signal, it will switch to that Task. For more details see section 3.2 on page 35 and figure 30 on page 39.

Scope: Task only¹⁹.

4.3 core/adx_Signal.h

4.3.1 Types and Aliases

SignalMode_t – SignalMode (signal state) type.

```
using SignalMode_t = Signal_t::Mode_t;
```

Signal_t::Mode_t is enumeration declared inside Signal_t class:

```
class Signal_t
{
public:
    enum class Mode_t : uint8_t
    {
        Disabled = 0,
        Silent ,
        Active = 3
    };
    //.....
};
```

4.3.2 Signal_t class methods

For detailed description of Signal_t class see section 3.3 on page 38.

Ctor – initializes created Signal object.

Declaration:

```
Signal_t(ptrFSM_t initPtrFsm , Mode_t initMode = SignalMode_t::
    Disabled);
```

Input parameters:

initPtrFsm – pointer to FSM object. Signal can be sent only to this assigned FSM (i.e. to the Task assigned to this FSM). Assigned by ctor FSM can’t be changed.

¹⁹If it’s called in main(), it won’t result in error, it will just return immediately

initMode – initial Signal Mode (state). Can be `SignalMode_t::Disabled`, `SignalMode_t::Silent`, `SignalMode_t::Active`. It's recommended to set Disabled or Silent initial mode to avoid spur wakeups.

Scope: global scope at Signal object definition, Task (directly in Task as local object with infinite lifetime).

Send() – sends Signal to the assigned Task.

Declaration:

```
virtual inline bool Send ();
```

Return value: if Signal is in Active Mode (state) and the Task was “called” (waked up) by the Signal, returns “true”, otherwise return “false”.

Behavior: `Send()` will wake up (“call”) Task only if Signal is in Active state, when “called” Task execution encounter `Wait()`, control will return to the place (Context) where `Send()` was called. If Signal is in Silent state, Task won't be waked up but Signal's counter will also be incremented as it would be in Active state. In Disabled state `Send()` will do nothing. For more details see figure 33 on page 43.

Scope: `main()`, Task.

operator() – designed to enable functor behavior for Signal object. So it can be used in templates using the same semantics as for ordinary function call: `(*ptrSig)();`

Declaration:

```
virtual inline bool operator () () {return Send ();}
```

Return value: the same as `Send()`.

Behavior: the same as `Send()`.

Scope: `main()`, Task.

SetMode() – sets Signal Mode (Signal state). Most used to set Signal Active before `Wait()` and Silent or Disabled after. Example: `GetFsmSignal()—>SetMode(SignalMode_t::Silent)`.

Declaration:

```
virtual void SetMode(SignalMode_t inMode);
```

Input parameters:

inMode – Signal Mode: `SignalMode_t::Disabled`, `SignalMode_t::Silent`, `SignalMode_t::Active`. It defines `Signal_t::Send()` behavior. For more details see figure 34 on page 43.

Behavior: sets mode and return immediately. To get more information about Signal and Event modes, and why it's so important, see section 2 on page 22.

Scope: `main()`, Task, ISR.

GetMode() – returns current Mode (state) of Signal.

Declaration:

```
inline SignalMode_t GetMode()
```

Return value: current Signal Mode value.

Behavior: returns immediately.

Scope: main(), Task, ISR.

GetFsm() – returns pointer to FSM, assigned to Signal object.

Declaration:

```
inline ptrFSM_t GetFsm()
```

Return value: pointer to FSM_t object (FSM address).

Behavior: returns immediately.

Scope: main(), Task, FSM.

SetErrorHandler() – sets error handler that will be used in Signal_t and derived class internal logic when calling Error() or Warning(). It can be used for debugging and profiling purposes as a way to get run-time system status information..

Declaration:

```
static void SetErrorHandler(ErrorHandler_t InErrorHandler)
```

Input parameters:

InErrorHandler – pointer to user error handler function, defined as described in section 4.1.1 on page 76.

Behavior: function copies passed function address to Signal_t static member and returns immediately. When executing, Signal_t and Event_t logic may call Error() or Warning() and pass this address to system status message which then is processed by ProcessSystemStatus(). As a result, user error handler will be called in main() context, see section 4.1.2 on page 78.

Scope: preferred in main() before FSM_t::Start(), but Task or ISR is possible.

4.3.3 Functions

WaitFor() – waits for a signal, specified by input parameter.

Declaration:

```
inline void WaitFor(ptrSignal_t ptrWaitingSignal)
```

Input parameters:

ptrWaitingSignal – pointer to a Signal that is about to wait for.

Behavior: replaces three calls – SetMode(SignalMode_t::Active), Wait(), SetMode(SignalMode_t::Silent), see figure 35 on page 44.

Scope: Task only.

4.4 core/adx_SystemQueue.h

4.4.1 SystemQueue_t<> class template

Declaration:

```
template <uint8_t SIZE = 16, typename T = uint8_t>
class SystemQueue_t
{
    //...
};
```

Input parameters:

SIZE – FIFO length, should be a power of 2 in range from 1 to 128 inclusive.

T – FIFO data type.

SystemQueue_t<> is commonly used in system logic, like event queue, but it also can be used in user application as fast general purpose FIFO (First In first Out) circular buffer. It can be safely used for data transfer between one any type (main(), Task, ISR) source and one any type receiver without critical section due to 8-bit put/get indexes. If multiple sources or receivers are used then it may require protection. For more information see section 3.4 on page 42 and figure 36 on page 44.

4.4.2 SystemQueue_t<> class methods

IsNotFull() – returns “true” if FIFO is not full, otherwise – “false”. Can be used to check if queue is ready to receive data.

Declaration:

```
inline bool IsNotFull() const {...}
```

Behavior: returns immediately.

Scope: main(), Task, ISR.

IsNotEmpty() – returns “true” if FIFO is not empty, otherwise – “false”. Can be used to check if queue is ready for the data to be read.

Declaration:

```
inline bool IsNotEmpty() const {...}
```

Behavior: returns immediately.

Scope: main(), Task, ISR.

Put() – puts data of type T (template parameter) to queue by reference.

Declaration:

```
inline bool Put(const T& Data) {...}
```

Input parameters:

Data – any variable or literal value of type T (T – template typename parameter).

Return value: returns “true” if data was successfully written to queue buffer, otherwise “false”.

Behavior: writes data to the buffer if it’s not full and returns immediately.

Scope: main(), Task, ISR.

PutE() – puts data of type T (template parameter) to queue by value. Used in event queue.

Declaration:

```
inline T* PutE(T Data) { ... }
```

Input parameters:

Data – any variable or literal value of type T (T – template typename parameter).

Return value: the address of data in the buffer that was successfully written, otherwise nullptr.

Behavior: writes data to the buffer if it’s not full and returns immediately.

Scope: main(), Task, ISR.

Get() – gets data from queue by reference.

Declaration:

```
inline bool Get(T& rData) { ... }
```

Output parameters:

rData – reference (variable which is used to receive data) of type T (T – template typename parameter).

Return value: returns “true” if data was successfully read from queue buffer, otherwise “false”.

Behavior: gets data from the buffer if it’s not empty and returns immediately.

Scope: main(), Task, ISR.

4.5 core/adx_Event.h

4.5.1 Types and Aliases

EventPrty_t – Event priority type.

```
using EventPrty_t = Event_t::Priority_t;
```

Event_t::Priority_t is enumeration declared inside Event_t class:

```
class Event_t: public Signal_t
{
public:
    enum class Priority_t : uint8_t
    {
        Low = 0,
        Middle,
        High
    };
    /...
};
```

4.5.2 Event_t class methods

Event_t is derived from base Signal_t class, so all public methods of Signal_t are also available in Event_t class. For detailed description of Event_t class see section 3.5 on page 44.

Ctor – initializes created Event object.

Declaration:

```
Event_t(ptrFSM_t initPtrFsm, Mode_t initMode = Mode_t::Silent,
        Priority_t initPriority = EventPrty_t::Middle) ;
```

Input parameters:

initPtrFsm – pointer to FSM object. Event can be sent only to this assigned FSM (i.e. to the Task assigned to this FSM). Assigned by ctor FSM can't be changed.

initMode – initial Signal Mode (state). Can be SignalMode_t::Disabled, SignalMode_t::Silent, SignalMode_t::Active. It's recommended to set Disabled or Silent initial mode to avoid spur wakeups.

initPriority – initial Event priority. Can be EventPrty_t::Low, EventPrty_t::Middle, EventPrty_t::High.

Scope: global scope at Event object definition, Task (directly in Task as local object with infinite lifetime).

Send() – sends Event to the assigned Task. Overrides Signal_t::Send() method.

Declaration:

```
virtual inline bool Send() override final ;
```

Return value: returns “false” if Event is Disabled, otherwise returns “true”.

Behavior: Send() puts Event in the queue if it's Active. If it's Silent, SetMode() will put it in the queue, when mode changed from Silent to Active. Scheduler will wake up (“call”) Task only if Signal is in Active state, when “called” Task execution encounter Wait(), control will return to the Scheduler. If Signal is in Silent state, Task won't be waked up but Signal's counter will also be incremented as it would be in Active state. In Disabled state Send() will do nothing. For more details see figure 39 on page 48.

Scope: main(), Task, ISR.

operator() – designed to enable functor behavior for Event object. So it can be used in templates using the same semantics as for ordinary function call: (*ptrSig)(); Overrides Signal_t::operator()() method.

Declaration:

```
virtual inline bool operator()() override final {return Send();}
```

Return value: the same as Send().

Behavior: the same as Send().

Scope: main(), Task, ISR.

SetMode() – sets Event Mode (Event state). Most used to set Event Active before Wait() and Silent or Disabled after. Example: GetFsmSignal()—>SetMode(SignalMode_t::Silent). Overrides Signal_t::SetMode() method.

Declaration:

```
virtual void SetMode(SignalMode_t inMode) override final;
```

Input parameters:

inMode – Event Mode: SignalMode_t::Disabled, SignalMode_t::Silent, SignalMode_t::Active. It defines Signal_t::Send() behavior. For more details see figure 40 on page 49.

Behavior: sets mode and returns immediately. To get more information about Signal and Event modes, and why it's so important, see section 2 on page 22.

Scope: main(), Task, ISR.

SetPriority() – sets Event priority. Defines which queue it will be put into – low, middle or high priority. Scheduler processes queue with higher priority first and proceed to lower priority if queue with higher priority is empty or “queue overload” condition occurred.

Declaration:

```
inline void SetPriority(Priority_t InPriority) { Priority =  
    InPriority; }
```

Behavior: returns immediately. New priority will be applied only when Event is removed from previous queue by Scheduler.

Scope: main(), Task, ISR.

GetPriority() – returns current Event priority.

Declaration:

```
inline EventPrty_t GetPriority() const { return Priority; }
```

Return value: current Event priority value.

Behavior: returns immediately.

Scope: main(), Task, ISR.

4.5.3 Functions

ProcessNextEventInQue() – Scheduler, processes one Event from one of the queues per function call. For more information see section 3.7 on page 54.

Declaration:

```
bool ProcessNextEventInQue();
```

Behavior: gets Event from the queue, runs associated with the Event Task (switches Context). When “called” Task execution encounter Wait(), it switches Context back, and control returns to the Scheduler.

Returns “true” if all event queues are empty, it means that main() loop can run “on idle” block, otherwise returns “false”, it means – proceed to next iteration, for next Event processing. Example is listed on page 54.

Scope: main() only.

4.6 core/adx_TimerEvent.h

4.6.1 TimerEvent_t class methods

TimerEvent_t class is derived from Event_t class (section on page 44) therefore it provides the same base functions. In addition it encapsulates single hardware timer/counter and provides software timer/counter capabilities for each TimerEvent_t object. Generally used for timeout implementation. For more details see section 3.8 on page 57, the use example is listed on page 58.

Ctor – initializes TimerEvent_t object. For simplicity it's referred to as Timer Event.

Declaration:

```
TimerEvent_t(ptrFSM_t ptrFsmFuncion, Mode_t initMode = Mode_t::Silent, Priority_t initPriority = Priority_t::Middle);
```

Input parameters are the same as for ordinary Event_t class ctor, see section on page 86:

ptrFsmFuncion – pointer to FSM object.

initMode – initial Signal Mode (state). Can be SignalMode_t::Disabled, SignalMode_t::Silent, SignalMode_t::Active. It's recommended to set Disabled.

initPriority – initial Event priority. Can be EventPrty_t::Low, EventPrty_t::Middle, EventPrty_t::High.

Scope: global scope, Task (directly in Task as local object with infinite lifetime).

SetMode() – sets Mode (state) of Timer Event (overrides to Event_t::SetMode()).

Declaration:

```
virtual void SetMode(Mode_t) override final;
```

Input parameters:

Mode – state of the Timer Event (SignalMode_t::Disabled, SignalMode_t::Silent, SignalMode_t::Active).

Behavior: similar to base class SetMode(), but has additional effect – when Timer Events of all TimerEvent_t objects are disabled, hardware timer/counter interrupt is also disabled for more efficient MPU usage, see figure 47 on page 58.

Scope: main(), Task, ISR.

SetPeriod() – sets period of software timer/counter. Timer Event is sent (if it's not Disabled) one time for Period + 1. For example, if hardware TC ISR period defined in config/adx_core_config.h is 5 ms, and Period == 0, then Timer Event will be sent each ISR call, i.e. 5 ms. If Period == 1 – each 10 ms, if 199 – one time per second. Each software Timer/Counter built in TimerEvent_t object has its own independent Period value.

Declaration:

```
inline void SetPeriod(uint16_t Period);
```

Input parameters:

Period – software Timer period, after which Timer Event is sent. Time resolution, i.e. Period units, is defined by hardware settings defined in config/adx_core_config.h.

Behavior: Period is updated by new value when previous Period is expired. To update immediately and start time waiting from the beginning – disable Timer Event, set new Period, then enable (set Silent or Active Mode).

Scope: main(), Task, ISR.

4.6.2 Functions

SleepFor() – blocks Task for the time according to the number of ticks.

Declaration:

```
void SleepFor(TimerEvent_t* ptrTimerEvent, uint16_t NofTicks);
```

Input parameters:

ptrTimerEvent – pointer to the Timer Event that is going to be used for delay.

NofTicks – delay time in terms of number of system ticks (period of hardware timer/counter). If NofTicks is 0, function sets period to 0, sends passed event immediately and calls Wait().

Behavior: turns on hardware timer and switches control to “caller”, i.e. to Scheduler if Event was used to wake up current Task or to another Task from which current Task was “called” using Signal. When delay is expired, the event (passed as ptrTimerEvent pointer) is sent by hardware timer ISR, Scheduler gets the event from queue and resumes assigned Task. Function always blocks Task, even if NofTicks is zero. Function automatically disables Timer Event before exit.

Scope: Task only.

4.7 core/adxDataQueue.h

4.7.1 DataQueue_t<> class template

Declaration:

```
template <typename ptrPutsideFunctorT, typename ptrGetsideFunctorT,
          uint8_t SIZE = 16, typename DataT = uint8_t>
class DataQueue_t
{
    //...
};
```

Input parameters:

ptrPutsideFunctorT – pointer type of functor or callback function used for notification of put side (code that writes data to FIFO). Callback function type must be bool (*)(), functor must have bool operator()() method. Example of template parameter: ptrEvent_t, ptrSignal_t, bool (*)().

ptrGetsideFunctorT – pointer type of functor or callback function used for notification of get side (code that reads data from FIFO).

SIZE – FIFO length, should be a power of 2 in range from 1 to 128 inclusive.

DataT – FIFO data type.

DataQueue_t<> is designed as a part of Driver Model for interrupt driven lossless data transfer between ISR and Task. It combines ordinary FIFO and efficient notification logic, that can be used to wake up Task which is waiting for data or buffer free space, and to notify ISR handler to start transaction. Although ISR to Task data transfer is it's main application, it also can be used in ISR to ISR or Task to Task communication. For more details see section 3.10 on page 61.

4.7.2 DataQueue_t<> class methods

IsNotFull() – returns “true” if FIFO is not full, otherwise – “false”. Can be used to check if queue is ready to receive data.

Declaration:

```
inline bool IsNotFull() const { ... }
```

Behavior: returns immediately.

Scope: main(), Task, ISR.

IsNotEmpty() – returns “true” if FIFO is not empty, otherwise – “false”. Can be used to check if queue is ready for the data to be read.

Declaration:

```
inline bool IsNotEmpty() const { ... }
```

Behavior: returns immediately.

Scope: main(), Task, ISR.

Put() – puts data of type **DataT** (template parameter) to queue by reference.

Declaration:

```
inline bool Put(const DataT& rData) { ... }
```

Input parameters:

rData – any variable or literal value of type **DataT** (**DataT** – template type-name parameter).

Return value: returns “true” if data was successfully written to queue buffer, otherwise “false”.

Behavior: writes data to the buffer if it's not full and returns immediately.

Scope: main(), Task, ISR.

Write() – copies data of type DataT (template parameter) from input buffer passed as a parameter and writes data to queue (FIFO).

Declaration:

```
bool Write(DataT* ptrData, uint8_t inSize, uint8_t&
rinoutSizeWritten);
```

Parameters:

ptrData – pointer to data buffer of type DataT (DataT – template typename parameter) from which data will be copied to FIFO.

inSize – input buffer length (i.e. total length of data to be written).

rinoutSizeWritten – input/output parameter passed by reference. Means current index of data in ptrData[] to be copied to FIFO.

Return value: returns “true” if all data was successfully written to queue buffer, otherwise “false”.

Behavior: reads data from ptrData buffer beginning from index number passed via rinoutSizeWritten, writes data to the FIFO until it’s not full or index reaches inSize value and returns immediately. rinoutSizeWritten will contain current index of not yet written data (if all data was written rinoutSizeWritten will be equal to inSize, and return value will be “true”). These parameters are suited for use in loop, like in the following example: **while**(SizeWritten < Size) **if**(!ptrDataQueue->Write(ptrDataBuf, Size, SizeWritten))Wait();

Scope: main(), Task, ISR.

Get() – gets data from queue and copies to passed by reference parameter.

Declaration:

```
inline bool Get(DataT& rData) {...}
```

Output parameters:

rData – reference (variable which is used to receive data) of type DataT (DataT – template typename parameter).

Return value: returns “true” if data was successfully read from queue buffer, otherwise “false”.

Behavior: gets data from the buffer if it’s not empty and returns immediately.

Scope: main(), Task, ISR.

Read() – gets data of type DataT (template parameter) from FIFO and copies to input buffer passed as a parameter.

Declaration:

```
bool Read(DataT* ptrData, uint8_t inSize, uint8_t& rinoutSizeRed);
```

Parameters:

ptrData – pointer to data buffer of type DataT (DataT – template typename parameter) to which data will be copied from FIFO.

inSize – input buffer length (i.e. total length of data to be read).

rinoutSizeWritten – input/output parameter passed by reference. Means current index of data in `ptrData[]` to be copied from FIFO.

Return value: returns “true” if all data was successfully read from queue buffer and copied to input buffer, otherwise “false”.

Behavior: writes data to `ptrData` buffer beginning from index number passed via **rinoutSizeWritten**, reads data from the FIFO until it’s not empty or index reaches **inSize** value and returns immediately. **rinoutSizeWritten** will contain current index of not yet read data (if all data was read **rinoutSizeWritten** will be equal to **inSize**, and return value will be “true”). These parameters are suited for use in loop, like in the following example: **while**(SizeRed < Size)**if**(!ptrDataQueue->Read(ptrDataBuf, Size, SizeRed))Wait();

Scope: `main()`, Task, ISR.

ConnectPutsideFuncutor() – assigns put side (code block which performs write operation) notification functor to `DataQueue_t` object, and sets **Putside-WakeupFlag** value. Generally used in Driver open/close methods.

Declaration:

```
inline void ConnectPutsideFuncutor(ptrPutsideFuncutorT
    inPtrPutsideFuncutor, bool inPutsideWakeupFlag)
```

Input parameters:

inPtrPutsideFuncutor – pointer to callback function `bool (*)()` or functor with `bool operator()()`. Generally it’s a pointer to ISR handler or Event. Type is defined by template parameter.

inPutsideWakeupFlag – this value overrides current **PutsideWakeupFlag**. It has the same meaning and behavior as in ctor.

Behavior: it resets put side notification system of the `DataQueue_t` object, but it does nothing with FIFO content.

Scope: `main()`, Task, ISR.

GetPutsidePtrFuncutor() – returns currently connected put side functor.

Declaration:

```
inline ptrPutsideFuncutorT GetPutsidePtrFuncutor() const { return
    ptrPutsideFuncutor; }
```

Behavior: returns immediately.

Return value: pointer to put side functor of type defined by template parameter.

Scope: `main()`, Task, ISR.

ConnectGetsideFuncutor() – assigns get side (code block which performs read operation) notification functor to `DataQueue_t` object, and sets **Getside-WakeupFlag** value. Generally used in Driver open/close methods.

Declaration:

```
inline void ConnectGetsideFuncutor(ptrGetsideFuncutorT
    inPtrGetsideFuncutor, bool inGetsideWakeupFlag)
```

Input parameters:

inPtrPutsideFuncutor – pointer to callback function `bool (*)()` or functor with `bool operator()()`. Generally it's a pointer to ISR handler or Event. Type is defined by template parameter.

inGetsideWakeupFlag – this value overrides current `GetsideWakeupFlag`. It has the same meaning and behavior as in ctor.

Behavior: returns immediately.

Return value: pointer to put side functor of type defined by template parameter.

Scope: `main()`, Task, ISR.

GetGetsidePtrFuncutor() – returns currently connected get side functor.

Declaration:

```
inline ptrGetsideFuncutorT GetGetsidePtrFuncutor() const { return
    ptrGetsideFuncutor; }
```

Behavior: returns immediately.

Return value: pointer to put side functor of type defined by template parameter.

Scope: `main()`, Task, ISR.

4.7.3 Functions

Functions described in this section are template functions which template parameters are derived so that when they are used, they look like ordinary overloaded functions. Template provides `Event_t` or `Signal_t` as notification functor for one `DataQueue` side, and arbitrary functor for other side. Full declaration for `Event_t` is listed on page 71. These functions provide Driver interfaces for read/write operations.

Put_bl() – puts data of type `DataT` (template parameter) to queue by reference. Function with blocking – it will wait for the FIFO not full condition to put data.

Declaration:

```
inline void Put_bl(DataQueue_t<...>* ptrDataQueue, const DataT&
    rData) { ... }
```

Input parameters:

ptrDataQueue – pointer to `DataQueue_t` object.

rData – any variable or literal value of type `DataT` (`DataT` – template type-name parameter).

Behavior: returns only when data is written to the FIFO. Function call may result in Context switching.

Scope: Task only.

Write_bl() – copies data of type DataT (template parameter) from input buffer passed as a parameter and writes data to queue (FIFO). Function with blocking – it will wait for the FIFO not full condition to put data.

Declaration:

```
inline void Write_bl(DataQueue_t<...>* ptrDataQueue, DataT*  
ptrDataBuf, uint8_t Size) {...}
```

Input parameters:

ptrDataQueue – pointer to DataQueue_t object.

ptrData – pointer to data buffer of type DataT (DataT – template typename parameter) from which data will be copied to FIFO.

inSize – input buffer length (i.e. total length of data to be written).

Behavior: returns only when all data is written to the FIFO. Function call may result in Context switching.

Scope: Task only.

Get_bl() – gets data from queue and copies to passed by reference parameter. Function with blocking – it will wait for the FIFO not empty condition to get data.

Declaration:

```
inline void Get_bl(DataQueue_t<...>* ptrDataQueue, DataT& rData)  
{...}
```

Parameters:

ptrDataQueue – pointer to DataQueue_t object.

rData – (output parameter) any variable of type DataT (DataT – template typename parameter) where data will be copied to.

Behavior: returns only when data is read from the FIFO. Function call may result in Context switching.

Scope: Task only.

Read_bl() – gets data from queue and copies to ptrDataBuf[] input buffer. Function with blocking – it will wait for the FIFO not empty condition to get data.

Declaration:

```
inline void Read_bl(DataQueue_t<...>* ptrDataQueue, DataT*  
ptrDataBuf, uint8_t Size) {...}
```

Parameters:

ptrDataQueue – pointer to DataQueue_t object.

ptrData – pointer to data buffer of type DataT (DataT – template typename parameter) to which data will be copied from FIFO.

inSize – input buffer length (i.e. total length of data to be read).

Behavior: returns only when all requested data is read from the FIFO. Function call may result in Context switching.

Scope: Task only.

4.8 core/adx_RecursiveMutex.h

Recursive mutex is used for mutual exclusive access to shared resources. Generally used in Driver Model for open/close operations. When several Tasks tries to lock one mutex object, only one is succeeded. It means the Task can work with shared resource, protected by this mutex object, and no one will interfere until it's unlocked. Other Tasks, which are not succeeded in lock, will be blocked and waiting for an Event from mutex to wake up and try lock again. Mutex is implemented in such way, that next try after wait will be successful. When the Task unlocks mutex, it sends the Event (that was passed as a parameter when calling lock) to one of the Tasks which is waiting for lock (in the same order they were trying to lock). Mutex is recursive, so it can be used in nested function calls. For more details see section 3.11 on page 72

4.8.1 RecursiveMutex_t class methods

Default ctor is used, copy and assignment operators are deleted.

TryLock() – tries to lock the mutex.

Declaration:

```
inline bool TryLock(ptrSignal_t inPtrSignal) {...}
```

Parameters:

inPtrSignal – pointer to Event which is used for Task notification if mutex is already locked by other Task.

Return value: returns “true” if lock was successful, i.e. mutex was not locked before. Otherwise returns “false”.

Behavior: Returns immediately, but next TryLock() call is possible only after receiving Event. Otherwise it may result in erroneous behavior.

Scope: Task only.

Lock_bl() – locks the mutex. Method with blocking, returns only when mutex is successfully locked by the Task.

Declaration:

```
inline void Lock_bl(ptrSignal_t ptrSig) {...}
```

Parameters:

inPtrSignal – pointer to Event which is used for Task notification if mutex is already locked by other Task.

Behavior: if mutex is already locked by another Task, it switches Context.

Scope: Task only.

Unlock() – unlocks mutex.

Declaration:

```
inline void Unlock() {...}
```

Behavior: sends Event to waiting Task if any, and returns immediately.

Scope: main(), Task, ISR.

4.9 core/adx_DataEvent.h and core/adx_DataSignal.h

4.9.1 SaE_Data_t<> class template

Declaration:

```
template< typename T >
class SaE_Data_t
{
    //...
};
```

Input parameters:

T – encapsulated data type.

SaE_Data_t class is used as a base for DataSignal_t and DataEvent_t classes for data encapsulation in Event or Signal objects.

4.9.2 SaE_Data_t class methods

SetData() – stores data to local member as atomic operation.

Declaration:

```
inline void SetData(const T& rInData) {...}
inline void SetData(uint8_t InData) {Data = InData;} //template
specification for uint8_t
```

Parameters:

InData/rInData – any variable or literal value of type T (T – template type-name parameter).

Behavior: stores data and returns immediately.

Scope: main(), Task, ISR.

GetData() – copies data from local member to specified variable as atomic operation.

Declaration:

```
inline void GetData(T& rOutData) {...}
```

Parameters:

rOutData – any variable of type T (T – template typename parameter)..

Behavior: copies data and returns immediately.

Scope: main(), Task, ISR.

4.9.3 DataSignal_t and DataEvent_t classes

These classes are derived from SaE_Data_t, Signal_t and Event_t accordingly, fig. 18 on page 29. So all interfaces of base classes are available. These classes can be used for data transfer using Event or Signal objects, for more information see 3.9 on page 61.

5 Use Cases and Examples

This demo shows behavior of Events: they are caught by WaitFor() regardless of the time it was sent -- before or after waiting. Execution order is the following regardless of inserted random delays: A1/A (concurrent) → B → C1/C (concurrent) → D → A/A1 → ...

```
// $Id: main.cpp 342 2025-01-05 00:40:58Z apolv $
// MyProdTask:                               MyConsTask:
// (A) Random delay                          (A1)Random delay
// (B) Send "Produced" event    ->   Wait for "Produced" event
// (C1)Random delay                  (C) Random delay
// Wait for "Consumed" event    <-   (D) Send "Consumed" event

#include "core/adx_TimerEvent.h"
using namespace adx_fsm;

void MyProdTask() __attribute__((OS_task));
void MyConsTask() __attribute__((OS_task));
FSM_t MyProdFsm{ uint32_t(MyProdTask), 64}, MyConsFsm{ uint32_t(
    MyConsTask), 64};
TimerEvent_t MyProdTimEvt{ &MyProdFsm }, MyConsTimEvt{ &MyConsFsm
    };
Event_t ProducedEvt{&MyConsFsm}, ConsumedEvt{&MyProdFsm};

uint8_t Random()
{
    static uint8_t x = 1;
    x = (x<<3) - 1 + x;
    return x>>6;
}

void MyProdTask()
{
    while(true)
    {
        SleepFor(&MyProdTimEvt,Random()); // (A)
        ProducedEvt(); // (B)
        SleepFor(&MyProdTimEvt,Random()); // (C1)
        WaitFor(&ConsumedEvt);
    }
}

void MyConsTask()
{
    while(true)
    {
        SleepFor(&MyConsTimEvt,Random()); // (A1)
        WaitFor(&ProducedEvt);
        SleepFor(&MyConsTimEvt,Random()); // (C)
        ConsumedEvt(); // (D)
    }
}

int main(void)
{
    {
        CriticalSection_t cs;
        //Hardware can be initialized here...
        FSM_t::Start();
    }

    // Scheduler's Main Loop
}
```

```

SystemStatus_t Status;
while (ProcessSystemStatus(Status))
{
    if (ProcessNextEventInQueue())
    {
        //Do something On Idle here...
    }
}

```

This demo shows simple data transfer protocol with packet auto synchronization when some data is lost.

Two Tasks writes their own data packets to UART independently, third Task reads from UART and tries to sync to packet and checks the CheckSum. Consistency of Tx packets (when mutual writing to UART) for each writing Task is provided by OpenWrite()/CloseWrite() methods based on mutex.

To run this demo – connect Tx pin with Rx pin of the selected UART on your board, otherwise reading Task will be waiting for data forever. Writing Tasks do not wait for reading Task, they write packets independently with delay of 1 tick (5ms by default).

```

// $Id: main.cpp 345 2025-01-06 09:19:22Z apolv $

```

```

#include "drivers/adx_UsartDriver.h"
#include "drivers/adx_SystemClock.h"
#include "core/adx_TimerEvent.h"
using namespace adx_fsm;

ADX_USART(MyUart,USARTE0)

void MyWriteTaskN() __attribute__((OS_task));
void MyReadTask() __attribute__((OS_task));
FSM_t WrFsm_array[] = {{uint32_t(MyWriteTaskN),192}, {uint32_t(MyWriteTaskN),192}};
FSM_t RdFsm{uint32_t(MyReadTask),160};
Event_t MyRdEvt{&RdFsm};

static const uint8_t PackSiganture = 0xaa;
union Packet_t
{
    uint8_t Buf[7];
    struct
    {
        uint8_t Header[2]; // = {PackSiganture, CmdId}
        uint8_t Data[4];
        uint8_t CheckSum;
    };
};

uint8_t CalcCheckSum(Packet_t* ptrPack)
{
    uint8_t result = 0;
    for(uint8_t i = 0; i < sizeof(Packet_t)-sizeof(Packet_t::CheckSum); i++) result = result + ptrPack->Buf[i];
    return result -1;
}

void MyWriteTaskN()
{

```

```

TimerEvent_t TimEvt{GetRunningFsm()};
Event_t MyWrEvt{GetRunningFsm()};
uint8_t Id = GetRunningFsm()->GetFsmID();
Packet_t Pack{PackSiganture, Id, static_cast<uint8_t>(Id<<2),
  static_cast<uint8_t>((Id<<2)+1), static_cast<uint8_t>((Id
  <<2)+2), static_cast<uint8_t>((Id<<2)+3), 0};
Pack.CheckSum = CalcCheckSum(&Pack);
while(true)
{
  auto hUart = MyUart.OpenWrite(&MyWrEvt);
  Write_bl(hUart, Pack.Header, sizeof(Packet_t::Header));
  Write_bl(hUart, Pack.Data, sizeof(Packet_t::Data));
  Write_bl(hUart, &Pack.CheckSum, sizeof(Packet_t::CheckSum));
  MyUart.CloseWrite();
  SleepFor(&TimEvt, 1);
}
}

void MyReadTask()
{
  Packet_t Pack;
  uint8_t CheckSum;
  uint8_t Counter0 = 0, Counter1 = 0;

  while(true)
  {
    auto hUart = MyUart.OpenRead_bl(&MyRdEvt);
    Read_bl(hUart, Pack.Buf, sizeof(Packet_t));
    CheckSum = CalcCheckSum(&Pack);
    while(CheckSum != Pack.CheckSum || Pack.Header[0] !=
      PackSiganture)
    {
      //Search for PackSiganture and try again
      do{
        Read_bl(hUart, Pack.Header, 1);
      } while(Pack.Header[0] != PackSiganture);
      Read_bl(hUart, Pack.Buf+1, sizeof(Packet_t)-1);
      CheckSum = CalcCheckSum(&Pack);
      if(CheckSum == Pack.CheckSum) break;
    }
    //Packet below is valid, count received packets with Id==0
    and Id==1:
    if(Pack.Header[1] == 0)
      Counter0++;
    else
      Counter1++;
    MyUart.CloseRead();
  }
}

const UsartConfig_t MyCfg PROGMEM = GetConfig(
  32'000'000,
  115'200,
  UsartSettings_t::FrameLen::b8,
  UsartSettings_t::Parity::None,
  UsartSettings_t::StopBits::One,
  UsartSettings_t::RxInterruptPriority::Middle,
  UsartSettings_t::TxInterruptPriority::Middle);

int main(void)
{
  {
    CriticalSection_t cs;

```

```
        SetSysAndPerClk_32MHzFrom16MHzXTAL();
        MyUart.Config(&MyCfg);
        FSM_t::Start();
    }

    // Scheduler's Main Loop
    SystemStatus_t Status;
    while (ProcessSystemStatus(Status))
    {
        if (ProcessNextEventInQueue())
        {
            //Do something On Idle here...
        }
    }
}
```

6 Time Profiling

TBD

7 Troubleshooting

TBD

7.1 Compile-time Errors

TBD

7.2 Run-time Critical Errors

7.2.1 Stack Overflow

TBD

7.2.2 Signal Loop: scheduler can't get control

TBD

7.2.3 Function Call within Wrong Context

TBD

7.3 System Status Handling

Table 9 on the next page contains system Error and Warning summary.

Table 9: System Error and Warning summary

Error / Warning Code	Type	Description
CRC16_Modbus("adx_Signal.h") == 0x1C52 (dec 7 250)		
1	Warning	Warning_SignalCounterOverflow. Signal Counter tries to exceed maximum of the used integral type.
CRC16_Modbus("adx_TimerEvent.cpp") == 0x4D58 (dec 19 800)		
8	Error	Error_SystemTimerEventBufferOverflow. Number of TimerEvent_t objects is greater than SystemTimerEventBuffer_SIZE.
CRC16_Modbus("adx_Event.h") == 0x2AEA (10 986)		
1	Warning	Warning_EventCounterOverflow. Event Counter tries to exceed maximum of the used integral type
2	Error	Error_LowPriorityQueueOverflow. Number of Events put in the queue exceeds Low Priority queue SIZE, defined in settings
3	Error	Error_MiddlePriorityQueueOverflow. Number of Events put in the queue exceeds Middle Priority queue SIZE, defined in settings
4	Error	Error_HighPriorityQueueOverflow. Number of Events put in the queue exceeds High Priority queue SIZE, defined in settings
CRC16_Modbus("adx_Event.cpp") == 0x599B (dec 22 939)		
5	Warning	Warning_LowPriorityQueueOverload. Number of successive calls from Low priority queue exceeded threshold defined in settings.
6	Warning	Warning_MiddlePriorityQueueOverload. Number of successive calls from Middle priority queue exceeded threshold defined in settings.
7	Warning	Warning_HighPriorityQueueOverload. Number of successive calls from High priority queue exceeded threshold defined in settings.