hitex
DEVELOPMENT TOOLS

TriCore™ AURIX™ Family

32-bit

Aurix Unleashed

Getting Started With Aurix

Application Note

V1.0 09-2015

Hitex UK Engineering Services

Edition 09-2015
Published by:
Hitex (U.K.) Limited.
University Of Warwick Science Park, Coventry, CV4 7HS, UK

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Warnings

Due to technical requirements, components may contain dangerous substances. For information on the types in question, please contact the nearest Infineon Technologies Office.

Infineon Technologies components may be used in life-support devices or systems only with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Revision History

Major changes since previous revision

Date	Version	Changed By	Change Description
12/08/2014	V0.1	M Beach	Sample with I2C completed. Remaining chapters incomplete
8/01/2015	V0.2	M Beach	QSPI added
13/07/2015	V0.3	M Beach/G Nath	Added GTM, GPT120. Added note about ASC pad driver.
20/07/2015	V0.4	M Beach/G Nath	Finalise VADC
14/08/2015	V0.5	M Beach/G Nath	Added reviewed GTM, VADC chapters. Added MSC.
19/08/2015	V0.6	M Beach/G Nath	Reviews of introductory chapter, ASCLIN, I2C
01/09/2015	V0.7	M Beach/G Nath	Proofed
08/09/2015	V1.0	M Beach/G Nath	Released

We Listen to Your Comments

Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of our documentation. Please send your proposal (including a reference to this document title/number) to:

ctdd@infineon.com

Table of Contents

List of Figures

List of Tables

1 AURIX Unleashed: An Introduction To Infineon's AURIX Microcontrollers

1.1 Purpose and Scope

This book is intended to be a basic introduction to the new Infineon Aurix 32-bit, multicore microcontroller for those working in general embedded systems development and is recommended reading before attempting the main user manual. It describes the most important features of the architecture, something of the safety credentials of the device, plus the most commonly used peripherals. The TC275 in the 176LQFP package is used as a basis.

The Aurix is one of the most powerful and sophisticated microcontrollers in existence and contains many leading edge features. However it can still be used and understood by those just armed with a few basic facts and this book is intended to set these out in simple terms. In some places, a lot of detail has been omitted for the sake of clarity (and brevity) and where this is the case, references are given for the main user manual so that the full information can be accessed.

1.2 Format

For each peripheral described, a simple usage example is given using the Infineon Framework and the Hightec GCC Free Entry Toolchain.

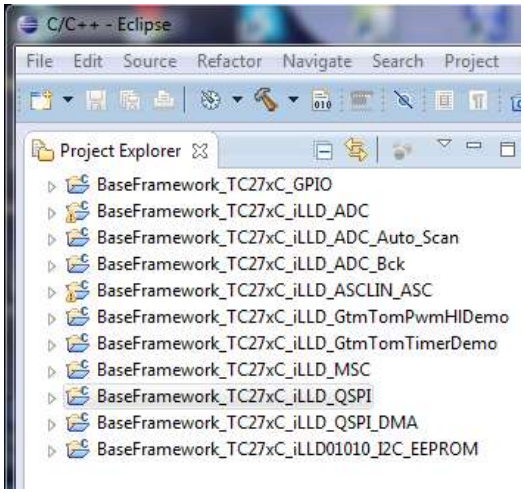


Figure 1 Aurix Basic Example Programs

The examples are available at:

ftp://internal.hitex.co.uk/pub/hitex/aurixunleashed/ExampleCode.zip

The examples are intended to run on the TC275 Application Kit, the TC2x5 Triboard and the ShieldBuddy TC275. They are built in the Infineon Framework, which is available by registering at MyInfineon on http://www.infineon.com. The Framework relies on the free version of the Hightec GCC toolchain which may be downloaded from:

http://free-entry-toolchain.hightec-rt.com/

2 Multicore Architecture & Memory Map

2.1 Introduction

The Aurix 32-bit microcontroller family covers a range of safety-orientated devices ranging from single core variants in 80pin TQFP packages up to triple core versions in 516BGA running at 300MHz. This book focuses on the TC275 (176LQFP), which is a mid-range device with three cores.

2.2 Aurix Overview

The Aurix TC275 is one of the high-end members of the Aurix family, having three independent 32-bit cores. The cores are derived from the Tricore family of single core 32-bit microcontrollers from Infineon AG. In this context the “Tri” refers to the instruction set of the device, which has a combination of microcontroller-like bit orientated instructions, DSP-style instructions and microprocessor features. One of the key enhancements in the Aurix is the addition of a ”lockstep” capability into the TriCore TC1.6x core. Here each CPU core consists of two identical cores executing the same instructions, but running two clock cycles out of phase. If the outcome from any instruction is not consistent between the cores, an exception is raised. Interestingly, the “checker” core is implemented as an “anticore” in that all of its operations are inverted, so that the XOR of the two cores’ outputs is zero under normal circumstances. Any disagreement between the cores might result from a localized disturbance caused by radiation or EMP. In a safety-related system such an upset could have dangerous consequences, so being able to detect them automatically is crucial. A precise description of the lockstep system can be found in the TC275 UM, section 5.12.8.

The TC275 has two lockstep cores and one non-lockstep core. Safety-related software is typically executed on the two lockstep cores whilst the remaining core might be used for non-critical functions. Besides lockstep and non-lockstep cores, there are Performance and Efficiency cores of type “TC1.6P” and “TC1.6E” respectively. The Performance cores have an additional loop pipeline, an intelligent branch predictor and can execute three instructions in parallel, yielding around a 15-25% execution speed improvement over the Efficiency core running at the same speed. The Performance cores can also run at up to 300MHz in some versions, giving a extra 50% boost.

Both core types have a single precision, 2FLOPS/cycle floating point unit to support floating point models.

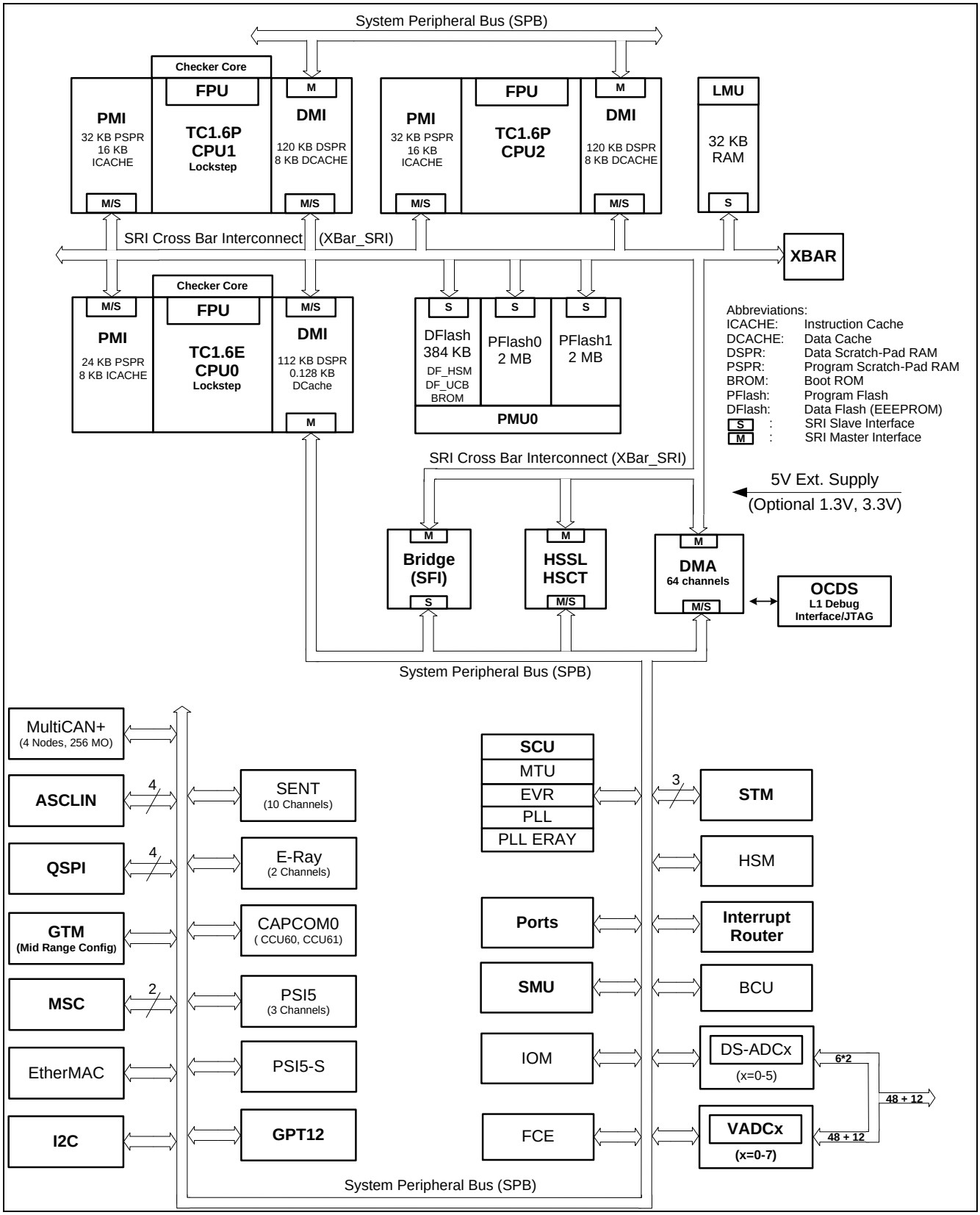


Figure 2 Aurix TC275 Internal Structure

One of the aims of the Aurix multicore design is to avoid the awkward programming issues that can arise in multicore processors and make the system architect's job easier. The three independent cores exist within a single memory space (0x00000000 – 0xFFFFFFFF), so they are all able to access all addresses without restriction. This includes all the peripherals and importantly all FLASH and SRAM areas. Perhaps the most useful feature is that two or even three cores can access the same memory address at the same time without causing a bus error or trap, which is a common problem on some non-Infineon multicore architectures. Not needing to take special measures when accessing SRAM can considerably ease the passing of data between cores using shared structures and is due to the implementation of a crossbar bus system to connect the cores, memories and DMA systems. Of course there are protection mechanisms that can generate traps for such accesses if the application requires it, as they may indicate a program malfunction which would need to be handled in an orderly manner. Access protection applies both to memory locations and peripheral SFRs. Where an existing application based on a number of physically separate microcontrollers is being condensed into a single Aurix, the ease with which data can be shared between processing units is a major advantage.

2.2.1 RAM

Again to ease programming, each core has its own close-coupled data SRAM (DSPR) which appears to exist at the normal TriCore address of 0xD0000000 but which is accessible to other cores at an alternative address. This is illustrated below.

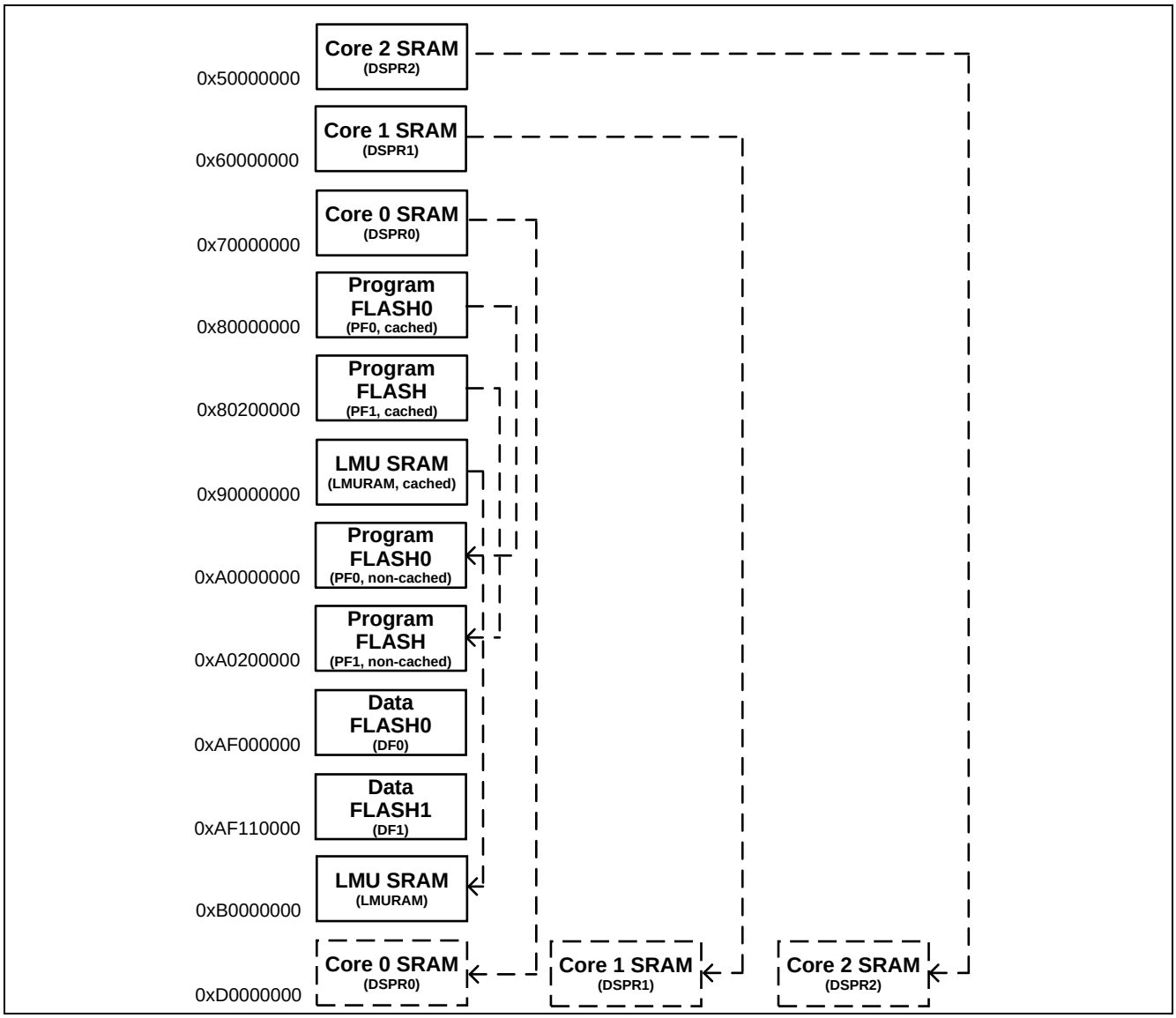


Figure 3 Simplified TC275 SRAM and FLASH Arrangement

Access to the 0xD0000000 address range is faster, in that it uses a short addressing mode so each core will have the bulk of its data in this “local” area. To access data in another core’s local memory requires a far addressing mode which is somewhat slower. The LMU RAM is a globally accessible SRAM area that is not tied to a particular core and is always far addressed.

Planning which data will go where is an important task that needs to be considered early in any new Aurix project. By default, compilers will place any data in CPU0’s DSPR, which is fine for simple programs and evaluation purposes, but is not really advisable for real applications. There is a Hitex application note on the subject of how to ensure that data is allocated to the appropriate core’s local SRAM, or the LMU.

The SRAMs have full hardware error detection and correction (ECC) with up to two bit errors being detectable and one being automatically corrected (SEC-DED). The access paths to the SRAMs also have error detection.

Table 1 TC275 SRAMs

Core 2	Type	TC1.6P non-lockstep
	PSPR	32k
	PCache	16k
	DSPR	120k
	DCache	8k
Core 1	Type	TC1.6P lockstep
	PSPR	32k
	PCache	16k
	SRAM	120k
	DCache	8k
Core 0	Type	TC1.6E lockstep
	PSPR	24k
	PCache	32k
	SRAM	112k
	DCache	4-line read buffer
General	LMU	32k
Total SRAM		728k

2.2.2 FLASH ROM

2.2.2.1 Program FLASH

The program FLASH (PFLASH) exists as a single memory area of 4MB and is shared between the 3 cores without restriction. As with the SRAM, all cores can access the FLASH at any time and at any address. Thus blocks of common code (maybe representing a frequently used function) can be shared freely by 3 cores. It is intended to be used as the main program storage area and has a minimum data retention time of 20 years, based on 1k write cycles. The PFLASH appears at two addresses in the memory space; once at 0xA0000000-0xA03FFFFF which is its true address and again at 0x80000000-0x803FFFFF. When accessed at the latter address, it is via a 32k cache for improved speed. This has no real impact on the user but gives a speed improvement. However it is important to remember when planning the memory map of a new design that these two ranges in fact relate to the same physical memory device. For example, it is not possible to locate a function at 0x80001000 and another at 0xA0001000, as these are in fact the same address.

The FLASH is ECC protected with dynamic correction of single-bit and double-bit errors and detection of triple-bit errors (“DEC-TED”).

How to program the PFLASH is covered in chapter 10 “On-Chip FLASH Programming”.

Non-volatile data that may be subject to frequent update such as adaptive parameters, calibration data etc. is stored in the dedicated Data FLASH. This is a special 384k region with a minimum of 125k write cycles and a 10 year data retention. Using a special driver library available from Infineon, the Data FLASH can be given the characteristics of EEPROM. By trading endurance for outright storage size, the number of write cycles can be dramatically

increased. For example, by restricting the stored data to 8kb, around three million write cycles are possible.

2.2.2.2 Program RAM

Although the Program FLASH has a fast 30ns, each core has access to a close-coupled SRAM, known as a Program Scratch Pad RAM or “PSPR”, designed for faster execution of critical code sections. This has a 0 clock cycle access time when a core accesses its own PSPR and up to 5 cycles when it accesses the PSPR from another core. The PSPR is 32KB on the Performance cores and 24KB on the Efficiency cores.

The placement of critical C functions into the PSPRs is supported by the compiler toolchains.

2.2.3 Memory Protection

In most safety-related applications and any multicore application, the prevention and trapping of unintended accesses to memory (especially writes) is essential. The Aurix has a number of different memory protection mechanisms. Each CPU has 16 data and 8 code ranges that can be setup to detect illegal write or execution accesses, with a granularity of 8 bytes making task-level memory protection feasible. Indeed several Aurix real time operating systems exploit this.

A related mechanism permits individual registers to be protected against access from particular cores. This allows peripherals to be reserved for use by just one core. The LMU SRAM has its own write protection scheme, where up to 8 ranges can be protected from write access from the CPU cores, DMA system, Ethernet and other sources of data writes.

2.2.4 Interrupts

Each core maintains its own interrupt vector table of up to 255 entries, although it is possible to use a single table for all three cores. There are up to a maximum of 255 interrupt priority levels. Interrupt pending flags are automatically cleared upon servicing. For all interrupt sources there is the option of a Direct Memory Access (DMA) transfer between any combination of peripheral register and memory address.

Software interrupts are supported and there is a completely separate per core, trap handling mechanism for traps arising from error conditions, with its own trap vector table.

2.2.5 Accessing Peripherals

In simpler applications, cores can access any peripheral as and when required. However, in safety-related systems, peripherals can effectively be assigned to particular cores using the register access protection scheme and through specifying which core is to provide any interrupt service during the configuration of the interrupt control registers. If a peripheral is accessed from an unintended core, a trap is created.

2.2.6 The Safety Management Unit

The management of all the possible errors that could occur in the Aurix is made by the Safety Management Unit (SMU). The error outputs from all major functional blocks in the device are routed to the SMU as “alarms”.

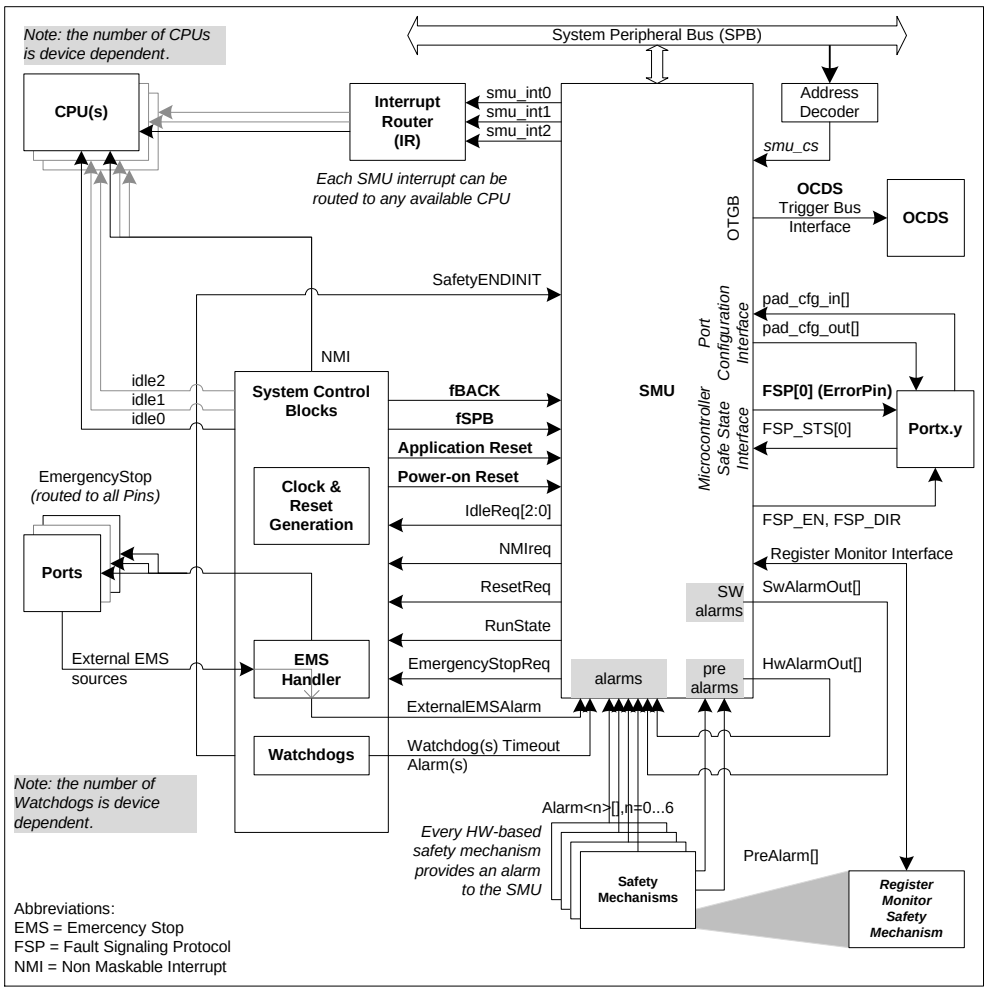


Figure 4 Safety Management Unit (SMU) Interfaces To Aurix Functional Blocks

The SMU combines all the alarms into a single error/no error status that is used to control either an internal safety monitor, or to create an overall error signal on a special pin (P33.8). This signal can be used to drive external watchdogs, or full scale safety monitors such as the TL35584. To accommodate different external safety monitor types, the error pin can be configured to perform a variety of actions in the event of an alarm. Many external watchdog devices require a pin toggle to keep them refreshed, so here the “fault signaling protocol” configured for the Aurix error pin would be “Stop Toggling On Error”. Other possibilities include “go high” or “go low”.

The SMU takes in alarms of the following types:

Table 2 SMU Alarm Inputs

CPU Lockstep	Processor interconnect (SRI) safety mechanisms
SRAM single bit error correction	Safety Flip-Flop
SRAM uncorrectable error	Clock monitors
SRAM Monitor: Address Buffer overflow	System PLL & Flexray-PLL loss of lock
SRAM addressing fault	Input/Output Monitoring Module
Program FLASH single bit error correction	Interrupt Router and Interrupt Control Units hardware monitor
Program FLASH double bit error correction	Voltage Monitor for internal voltage regulators
Program FLASH arbitrary multiple-bit uncorrectable error	Error reporting from Shared
Program FLASH Monitor: address buffer full	On-chip temperature sensor
Program FLASH addressing fault	Processor interconnect (SRI) safety mechanisms

2.2.6.1 Testing Fault Detection Mechanisms

A major requirement of most current functional safety standards is that any built-in error detection mechanism in microcontrollers must be testable at run time, to confirm that they work correctly. Thus for example, in the context of the Aurix, the ECC on FLASH and RAM must be tested to make sure that ECC errors really do cause traps to occur. For the lockstep system, it must be shown that a disagreement between the core and anticore does give a trap and that an alarm is sent to the SMU.

All built-in error detection systems on the Aurix are testable at start up using software. There is no requirement for special test-pattern generation functions as may be seen on older non-Infineon devices. Where software cannot realistically be used to test a function, a special fault-injection mode is incorporated into the device. For example, to demonstrate that the lockstep comparator between the main and checker cores is able to detect a discrepancy, a fault can be deliberately injected and the response monitored.

Thus in this case, the failure to detect a deliberately injected fault would create a SMU alarm, as would critical runtime faults like a PFLASH ECC error.

2.2.7 Voltage Supplies

The Aurix as a whole can be run as a pure 3V3 device, or a 5V device and there is a choice of using the on-chip voltage regulators or external voltage sources. How it is configured is determined by the state of the HWCFG[0..2] (P14.6, P14.5, P14.2) pins before the reset is released (#PORST). There are two on-chip voltage regulators which can generate the 3V3 IO and 1V3 core voltages, with or without an external power device. The 1V3 core voltage regulator can run in either switching mode (SMPS), or linear low drop-out mode (LDO). The possible options are given in the table below.

Table 3 Aurix Voltage Supply Options

5 V single source supply, EVR13 in SMPS mode, EVR33 in LDO mode. 5 V or 3.3 V ADC domain. 5 V or 3.3 V Flexport domain.	5 V single source supply, EVR13 in LDO mode with external pass device, EVR33 in LDO mode. 5 V or 3.3 V ADC domain. 5 V or 3.3 V Flexport domain.	5 V & 1.3 V external supply, EVR13 inactive, EVR33 in LDO mode. 5 V or 3.3 V ADC domain. 5 V or 3.3 V Flexport domain.
3.3 V single source supply, EVR13 in SMPS mode, EVR33 inactive. 5 V or 3.3 V ADC domain. 3.3 V Flexport domain.	3.3 V single source supply, EVR13 in LDO mode with external pass device, EVR33 inactive. 5 V or 3.3 V ADC domain. 3.3 V Flexport domain.	5 V, 3.3 V and 1.3 V are supplied externally, EVR13 and EVR33 inactive. 5 V or 3.3 V ADC domain. 5 V or 3.3 V Flexport domain.

3 General Purpose IO Ports and Peripheral IO lines (ports)

3.1 GPIO Introduction

The most basic activity of a microcontroller is setting and clearing port pins. The Aurix groups together port pins as Ports, numbered “P0” to “P33” on the TC275 176LQFP, although some BGA versions have up to P40. Thus P00.00 is Port zero bit zero, P02.1 is Port 2, bit 1. Most port pins can be configured as outputs or inputs (except SAR pins which are only inputs to reduce potential analog noise).

```
/* Turn LED On */
IfxPort_setPinState(&MODULE_P33, 8u, IfxPort_State_high);

/* Read Port 10.2 into a variable */
Port10_2_State = IfxPort_getPinState(&MODULE_P10, 2u);
```

In addition, most port pins have up to 8 configurable output modes and 12 different input modes, depending on how they are configured. The basic pin modes are General Purpose Output and General Purpose Input. The alternate modes are to allow the various peripherals in the TC275 such as ASC, QSPI, CAN, GTM etc. to be connected to the outside world.

Table 4 P33.13 Input And Output Modes

P33.13	I	General-purpose input	P33_IN.P13	P33_IOCR12.PC13	0XXXXB
GTM input		TIN35			
QSPI3 input		MRST3D			
DSADC input		DSSGNB			
MSC1 input		INJ11			
ASCLIN1 input		ARX1F			
	O	General-purpose output	P33_OUT.P13		1X000B
GTM output		TOUT35			1X001B
ASCLIN1 output		ATX1			1X010B
QSPI3 output		MRST3			1X011B
QSPI2 output		SLSO26			1X100B
Reserved		1X101B			
SCU output		DCDCSYNC			1X110B
CCU61 output		CC60			1X111B

Taking P33.13 as an example, the possible input modes are:

- GTM input TIN35 (this is GTM input timer registers TIM2_1 and TIM3_1)
- QSPI module 3 MRST (Master Receive Slave Transmit)
- DSADC Delta-Sigma ADC input DSSGNB(sign input B, carrier signal)
- MSC1 input INJ 1 external signal injection
- ASCLIN1 UART channel 1 receive

The output modes are:

- GTM output TOUT35 (this is GTM output timer registers TOM1_13, TOM2_13, ATOM2_5 and ATOM3_5.
- ASCLIN1 UART channel 1 transmit
- QSPI module 3 MTSR (Master Transmit, Slave Receive)

- QSPI module 3 slave chip select 26
- SCU external step-down voltage regulator synchronization clock
- CCU6 module 1 output from compare register CC60 in the capture/compare unit

Some port pins have the usual general purpose input and output functions and some alternate peripheral functions but in addition have special purpose modes that relate to the overall system function. For example, P33.8 has a special mode related to the Safety Management Unit (SMU) fault signalling protocol “HWOUT” function. This pin is normally connected to an external safety watchdog device (e.g. TLF35584) so that the SMU can shut the system down in the event of a critical failure. Also, P21.2 can be configured as the Emergency Stop trigger to automatically force critical port pins into a safe state without software intervention.

P14.2 to 14.6 and P10.5 to P10.6 are used for device hardware configuration via signals HWCFG0 – HWCFG6. These are read during reset to determine for example, the type of power supply the Aurix will use (3V3, 5V, internal/external switching device etc.). It is important that any GPIO function they perform does not conflict with the required HWCFG settings!

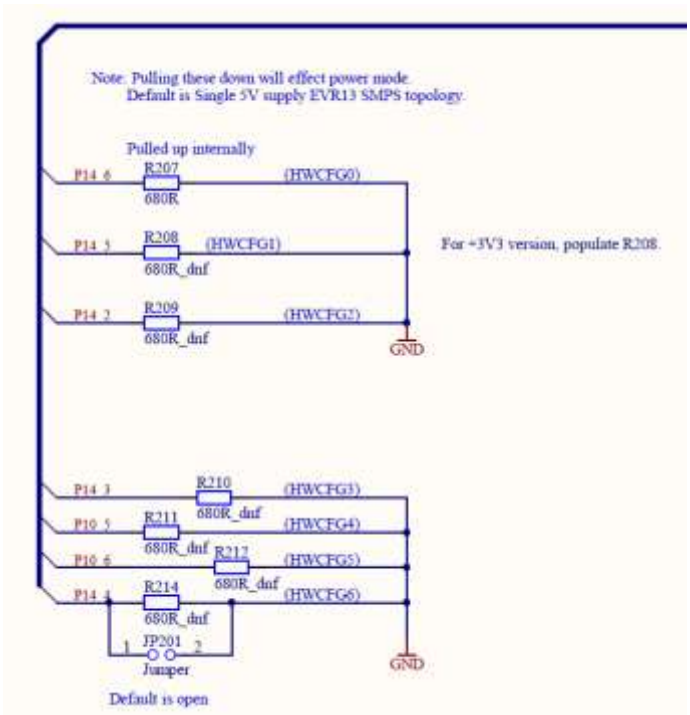


Figure 5 Hardware Configuration (HWCFG) Pins

The JTAG/DAP debug interface pins are always 3V3 regardless of whether the ports are running at 3V3 or 5V.

These special functions for certain port pins should be borne in mind when making the device pin allocation in new projects.

After reset and when HWCFG6 = 1, all pins except P33.8 (tristated whilst #PORTS is active), are set to input with pull-up mode. If HWCFG6 = 0, then all pins are tristated.

There are three main types or classes of pad driver to which a particular port pin can belong, namely MP, MP+, MPR and LP. These groupings also reflect the permissible operating voltage of the pins.

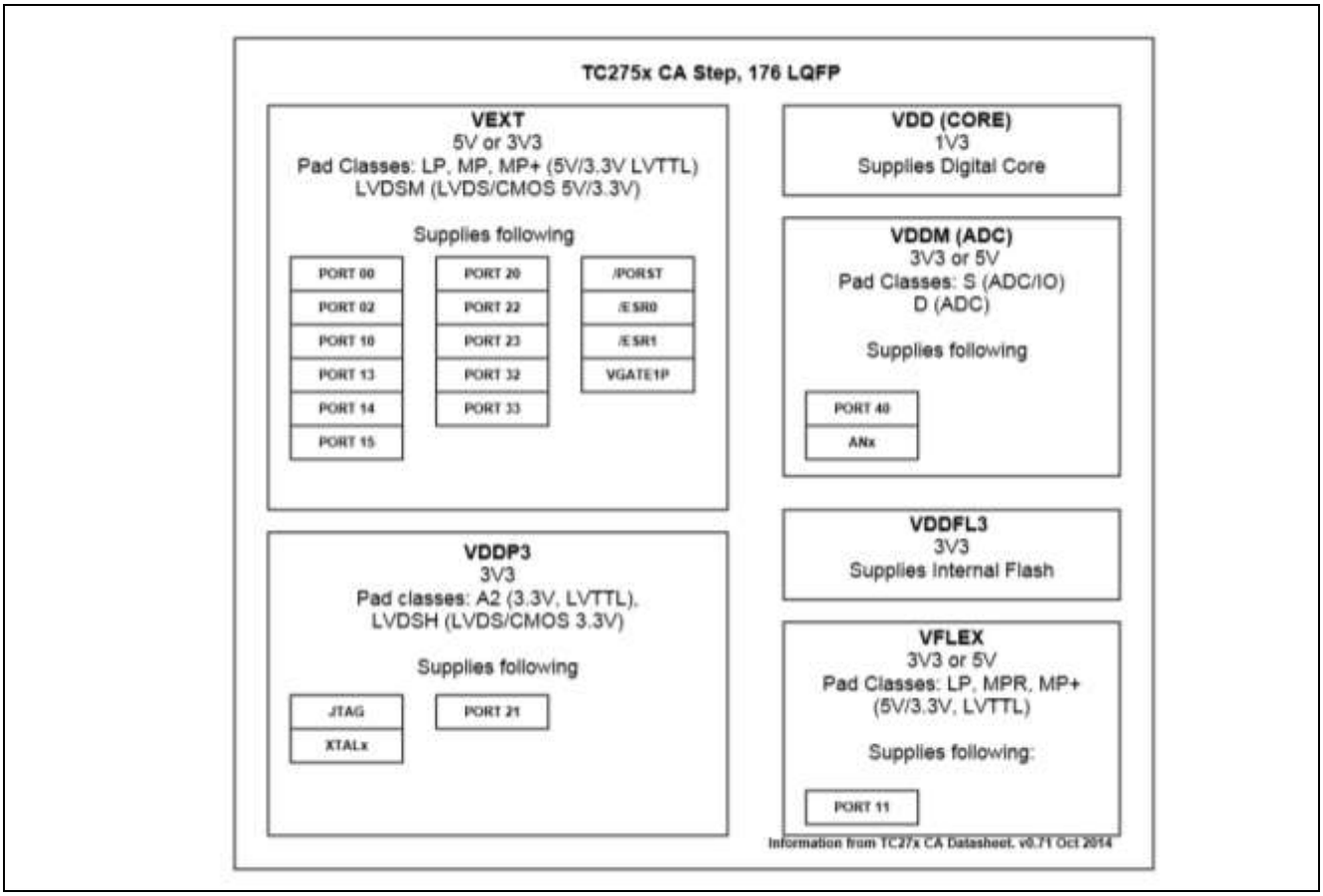


Figure 6 Allocation of ports and pad driver classes to power domains

Pins in these 4 classes can be run either all at 3V3 or all at 5V, depending on the chosen value of the V_{EXT} power supply pins. Each pad class can be tuned on a pin-by-pin basis for CMOS or TTL voltage levels, four different output driver strengths and even slew rate in the case of the LP pads. The input hysteresis can also be set (except for P14.2/4/5/6, P15.1 which always have hysteresis). Port pins are also grouped together according to the pad class that they belong to, as well as the actual port number. Note that not all the pins on a particular port belong to the same pad type group. For example, Port 33.9 is a LP pad, whereas Port 33.10 is a MP pad.

The user has 4 speed grades to choose from, except for the port pins with LP pads which only have 2. The default port pad driver configuration is the slowest speed grade (Grade 4) with CMOS voltage levels (“Automotive Level”) with hysteresis when configured as an input.

The characteristics of a number of common pad driver classes are listed below.

Table 5 Pad Type Characteristics

Pad Type	Slew
5V MP Characteristics Input hysteresis: CMOS 0.5V TTL 0.4V	95ns - 200ns with 50pF to 200pF capacitance, weak driver 25ns – 50ns medium strength driver, 17.5ns – 30ns, medium edge, strong driver, 7ns – 17ns sharp edge, strong driver (depending on capacitive loading)
3V3 MP Characteristics Input hysteresis: 0x165V	150ns - 320ns 50pF to 200pF capacitance, weak driver 30ns – 70ns medium strength driver 32ns - 50ns, medium edge, strong driver 14ns – 32ns, sharp edge, strong driver (depending on capacitive loading)
5V MP+ Characteristics Input hysteresis: CMOS 0.5V TTL 0.4V of Vext	95ns - 200ns 50pF to 200pF capacitance, weak driver 25ns – 50ns medium strength driver 9ns – 17ns, medium edge, strong driver 4ns – 12ns sharp edge, strong driver (depending on capacitive loading)
5V LP Characteristics Input hysteresis: 0.5V	95ns - 200ns 50pF to 200pF capacitance, weak driver 25ns – 50ns medium strength driver (depending on capacitive loading)
5V MP+ Characteristics Input hysteresis: CMOS 0.5V TTL 0.4V of Vext	95ns - 200ns 50pF to 200pF capacitance, weak driver 25ns – 50ns medium strength driver 9ns – 17ns, medium edge, strong driver 4ns – 12ns sharp edge, strong driver (depending on capacitive loading)
5V LP Characteristics Input hysteresis: 0.5V	95ns - 200ns 50pF to 200pF capacitance, weak driver 25ns – 50ns medium strength driver (depending on capacitive loading)

Note: Slew is measured between (10% to 90%) of V_{EXT}

The table below shows which port pins belong to which pad type.

Table 6 Port Pin To Pad Class Mapping

Pin	Port Pin	Pad Type		Pin	Port Pin	Pad Type		Pin	Port Pin	Pad Type
41	ADC, P40.3	S		98	P22.3	5V MP LVDS_P		142	P14.0	5V MP
42	ADC, P40.2	S		105	P21.0	3.3V A2		143	P14.1	5V MP
43	ADC, P40.1	S		106	P21.1	3.3V A2		144	P14.2	5V LP
44	ADC, P40.0	S		107	P21.2	3.3V F, LVDSH_N		145	P14.3	5V LP
70	P33.0	5V LP		108	P21.3	3.3V F, LVDSH_P		146	P14.4	5V LP
71	P33.1	5V LP		109	P21.4	3.3V F, LVDSH_N		147	P14.5	5V MP
72	P33.2	5V LP		110	P21.5,	3.3V F, LVDSH_P		148	P14.6	5V MP
73	P33.3	5V LP		111	P21.6, TDI	3.3V A2		149	P14.7	5V LP
74	P33.4	5V LP		113	P21.7 TDO	3.3V, A2 pullup		150	P14.8	5V LP
75	P33.5	5V LP		116	P20.0	5V MP		151	P14.9	5V MP
76	P33.6	5V LP		117	P20.1	5V LP		152	P14.10	5V MP
77	P33.7	5V LP		118	P20.2, TESTMODE (N)	pullup		156	P13.0	5V MP LVDSM_N
78	P33.8	5V MP		119	P20.3	5V LP		157	P13.1	5V MP LVDS_P
79	P33.9	5V LP		124	P20.6	5V LP		158	P13.2	5V MP LVDSM_N
80	P33.10	5V MP		125	P20.7	5V LP		159	P13.3	5V MP LVDSM_P
81	P33.11	5V MP		126	P20.8	IFX:MP		160	P11.2	5V MP [VFLEX)
82	P33.12	5V MP		127	P20.9	5V LP		161	P11.3	5V MP [VFLEX)
83	P33.13	5V MP		128	P20.10	IFX:MP		162	P11.6	5V MP [VFLEX)
86	P32.2	5V LP		129	P20.11	IFX:MP		163	P11.9	5V MP [VFLEX)
87	P32.3	5V LP		130	P20.12	IFX:MP		165	P11.10	5V LP (VFLEX)
88	P32.4	5V MP		131	P20.13	IFX:MP		166	P11.11	5V MP [VFLEX)
89	P23.0	5V LP		132	P20.14	IFX:MP		167	P11.12	5V MP [VFLEX)
90	P23.1	5V MP		133	P15.0	5V LP		168	P10.0	5V LP
91	P23.2	5V LP		134	P15.1	5V LP		169	P10.1	5V MP
92	P23.3	5V LP		135	P15.2	5V MP		170	P10.2	5V MP
93	P23.4	5V MP		136	P15.3	5V MP		171	P10.3	5V MP

Pin	Port Pin	Pad Type		Pin	Port Pin	Pad Type		Pin	Port Pin	Pad Type
94	P23.5	5V MP		137	P15.4	5V MP		172	P10.4	5V MP
95	P22.0	5V MP LVDSM_N		138	P15.5	5V MP		173	P10.5	5V LP
96	P22.1	5V MP LVDS_P		139	P15.6	5V MP		174	P10.6	5V LP
97	P22.2	5V MP LVDSM_N		140	P15.7	5V MP		175	P10.7	5V LP
				141	P15.8	5V MP		176	P10.8	5V LP

Note: The maximum pin sinking current is 10mA but it is advisable to keep well below this.

3.1.1 More Information

Section 3.5 in TC27xC_DS_v10.pdf has the characteristics of all pad classes in detail

Refer to ap32202_PinConfigurationPowerSupplyAndReset.pdf

3.2 Emergency Stop Mode

The port system has a degree of independence from both the Aurix peripherals that drive it and the application software through the Emergency Stop function. This allows pins P33.8 (“EMGSTOPA”) and P21.1 (“EMGSTOPB”) plus the SMU to force GPIOs into a defined state. In the event of an Estop (Emergency Stop), a pin is immediately switched to high impedance input mode with an optional internal pull-up, depending on the value of PMSWCR0.TRISTREQ or the HWCFG[6] pin is level latched whilst the Power-On Reset (PORTS) is active.

Emergency stop mode is not available for:

1. P20.2 (General Purpose Input/GPI only, overlayed with Testmode)
2. P40.x (analoge input ANx overlayed with GPI)
3. P32.0 EVR13 SMPS mode.
4. Dedicated I/O

3.3 External Request Unit (ERU) Inputs

The External Request Unit (ERU) allows some GPIO pins to generate interrupts directly on defined pin transition. Up to any 8 or the 17 ERU request pins can be used this way. Trigger events may be either rising or falling edges. A more advanced application of the ERU is the detection of user-defined patterns on multiple pins. As with the basic detection of pin changes, other peripherals such as the VADC can be triggered as an alternative to interrupt generation.

Table 7 ERU Pin Mapping

Request No.	Port Pin	Request No.	Port Pin
REQ0	P15.4	REQ9	P20.0
REQ1	P15.8	REQ10	P14.3
REQ2	P10.2	REQ11	P20.9
REQ3	P10.3	REQ12	P11.10
REQ4	P10.7	REQ13	P15.5
REQ5	P10.8	REQ14	P02.1
REQ6	P02.0	REQ15	P14.1
REQ7	P00.4	REQ16	P15.1
REQ8	P33.7		

4 Asynchronous /Synchronous Interface (ASCLIN)

4.1 ASCLIN Introduction

The ASCLIN peripheral is used to communicate with the UART serial protocol using a simple 2 wire bus to transfer data both ways between 2 devices. The baudrate has to be predetermined by both sides as there is no clock to keep the 2 devices in sync. The 2 data lines are identical, except for the direction that the data travels in. Data is usually transferred in 8-bit frames with start and stop bits either side of the data.

The ASCLIN also supports SPI and LIN modes, but these are not covered in this document.

4.2 AURIX Implementation

The ASC mode in the ASCLIN module has the following general features:

• Full-duplex asynchronous operating modes
– 7-bit, 8-bit or 9-bit (or up to 16-bit) data frames, LSB first
– Parity-bit generation/checking
– One or two stop bits
– Max baud rate 6.25 MBaud
– Min. baud rate 0.37 Baud
• Optional RTS / CTS handshaking
• 16 bytes TxFIFO
• 16 bytes RxFIFO
• Pack / unpack capabilities of the Tx and Rx FIFO
• Interrupt generation
– On a configurable transmit FIFO level
– On a configurable receive FIFO level
– On an error condition (frame, parity, overrun error)
– On various module internal events (end of ASC/SPI frame, LIN events)
• Either CPU or DMA interrupt service
• Programmable digital glitch filter and median filter for incoming bit stream
• Shift direction LSB first
• Internal loop-back mode
– Up to 4 times higher baud rates (25 MBaud) using special hardware features such as programmable sampling point and filters.

4.3 Using The ASCLIN ASC iLLDs

4.3.1 Basic Overview

The ASCLIN iLLD is configured using the following steps:

Create Rx and Tx Buffers
Instantiate ASC structure
Define interrupt priorities
Define interrupt routines
Create module config structures
Fill the config structure with default values.
Edit any config values required.
Initialize the module.

Now the read and write functions detailed in the next section can be used to communicate on the bus.

4.3.2 Setting up ASCLIN

First the Asclin library is imported:

```
#include <Asclin/Asc/IfxAsclin_Asc.h>
```

Then the Asclin module is instantiated:

```
static IfxAsclin_Asc asc;
```

The ASC Tx and Rx buffers are created:

```
#define ASC_TX_BUFFER_SIZE 64
static uint8 ascTxBuffer[ASC_TX_BUFFER_SIZE + sizeof(Fifo) + 8];
#define ASC_RX_BUFFER_SIZE 64
static uint8 ascRxBuffer[ASC_RX_BUFFER_SIZE + sizeof(Fifo) + 8];
```

The interrupt priorities are defined:

```
#define IFX_INTPRIO_ASCLIN0_TX 1
#define IFX_INTPRIO_ASCLIN0_RX 2
#define IFX_INTPRIO_ASCLIN0_ER 3
```

The interrupt service routines are then defined:

```
IFX_INTERRUPT(asclin0TxISR, 0, IFX_INTPRIO_ASCLIN0_TX)
{
IfxAsclin_Asc_isrTransmit(&asc);
```

```
}
IFX_INTERRUPT(asclin0RxISR, 0, IFX_INTPRIO_ASCLIN0_RX)
{
IfxAsclin_Asc_isrReceive(&asc);
}
IFX_INTERRUPT(asclin0ErISR, 0, IFX_INTPRIO_ASCLIN0_ER)
{
IfxAsclin_Asc_isrError(&asc);
}
```

Then the interrupt handlers are installed:

```
IfxCpu_Irq_installInterruptHandler(&asclin0TxISR, IFX_INTPRIO_ASCLIN0_TX);
IfxCpu_Irq_installInterruptHandler(&asclin0RxISR, IFX_INTPRIO_ASCLIN0_RX);
IfxCpu_Irq_installInterruptHandler(&asclin0ErISR, IFX_INTPRIO_ASCLIN0_ER);
IfxCpu_enableInterrupts();
```

Next the module config structure is instantiated and filled with default values:

```
IfxAsclin_Asc_Config ascConfig;
IfxAsclin_Asc_initModuleConfig(&ascConfig, &MODULE_ASCLIN0);
```

Then the default values are changed including setting the interrupt priorities and setting the pins:

```
/* set the desired baudrate */
ascConfig.baudrate.prescaler = 1;

/* FDR values will be calculated in initModule */
ascConfig.baudrate.baudrate = 1000000;

/* ISR priorities and interrupt target */
ascConfig.interrupt.txPriority = IFX_INTPRIO_ASCLIN0_TX;
ascConfig.interrupt.rxPriority = IFX_INTPRIO_ASCLIN0_RX;
ascConfig.interrupt.erPriority = IFX_INTPRIO_ASCLIN0_ER;
ascConfig.interrupt.typeOfService = (IfxSrc_Tos)IfxCpu_getCoreId();

/* FIFO configuration */
ascConfig.txBuffer = &ascTxBuffer;
ascConfig.txBufferSize = ASC_TX_BUFFER_SIZE;
ascConfig.rxBuffer = &ascRxBuffer;
ascConfig.rxBufferSize = ASC_RX_BUFFER_SIZE;
```

```
/* pin configuration */
const IfxAsclin_Asc_Pins pins = {
NULL, IfxPort_InputMode_pullUp, /* CTS pin not used */
&IfxAsclin0_RXA_P14_1_IN, IfxPort_InputMode_pullUp, /* Rx pin */
NULL, IfxPort_OutputMode_pushPull, /* RTS pin not used */
&IfxAsclin0_TX_P14_0_OUT, IfxPort_OutputMode_pushPull, /* Tx pin */
IfxPort_PadDriver_cmosAutomotiveSpeed1
};
ascConfig.pins = &pins;
```

The receive pin (here P14.1) pad driver needs to be manually set to speed grade 1 otherwise a 3v3 input voltage is not seen as a '1' . This can be a problem with some serial devices e.g. GPS modules.

```
/* Manually set pad driver to speed grade 1 */
/*(otherwise 3v3 is not seen as a '1' )     */
    IfxPort_setPinPadDriver(&MODULE_P14,1,
                            IfxPort_PadDriver_cmosAutomotiveSpeed1) ;
```

Finally the module is initialized:

```
IfxAsclin_Asc_initModule(&asc, &ascConfig);

IfxPort_setPinPadDriver(&MODULE_P14,1,
                        IfxPort_PadDriver_cmosAutomotiveSpeed1) ;
```

4.3.3 Important functions

4.3.3.1 Blocking Write

The Blocking Write function allows one byte to be written to the bus.

```
IfxAsclin_Asc_blockingWrite(&asc, byte);
```

The Blocking Write function accepts these arguments:

```
Asc:  The asclin peripheral to be transmitted on.
Byte: The byte you want to transmit
```

4.3.3.2 Blocking Read

The Blocking Read function reads a single byte off the bus and returns it.

```
uint8 data = IfxAsclin_Asc_blockingRead(&asc);
```

The Blocking Read function accepts these arguments:

```
Asc: A pointer to the asclin peripheral to read from
```

The Blocking Read function returns these arguments:

```
Byte: The byte read off the UART bus.
```

4.3.3.3 Streamed Write

The Streamed Write function allows for multiple bytes of data to be written into the buffer and will be streamed out when the bus is ready.

The Streamed Write function accepts the following arguments:

```
Asc:     A pointer to the asc peripheral to stream to data on to.
txData:  An array of bytes to be transmitted on the bus.
Count:   The number of bytes to be transmitted onto the bus.
Timeout: The time the function should wait before timing out.
```

4.3.3.4 Streamed Read

The Streamed Read function will read the number of bytes specified from the ASC receive buffer and place them into an array.

```
IfxAsclin_Asc_read(&asc, rxData, &count, TIME_INFINITE);
```

The Streamed Read function accepts the following arguments:

```
Asc:       A pointer to the ASC peripheral from which the
data shall be read.
rxData:    The array into which the data shall be read.
Count:     The number of bytes to be read.
Timeout:   The time the function should wait before timing out.
```

4.3.3.5 Get Read Count

The Get Read Count function finds the number of bytes in the ASC receive buffer.

```
Ifx_SizeT count = IfxAsclin_Asc_getReadCount(&asc);
```

The function takes the following argument:

```
Asc: The ASC peripheral from which the number of bytes available is to be
read from.
```

The function returns the following:

```
Count: The number of bytes available for reading.
```

4.4 Config Options

The ASCLIN peripheral may be accessed via the pins given in the table below. Useful macros and structures for accessing the ASCLIN in 'C' may be found in the programming examples:

```
.\0_Src\4_McHal\Tricore\_PinMap\IfxAsclin_PinMap.c
.\0_Src\4_McHal\Tricore\_PinMap\IfxAsclin_PinMap.h
```

Table 8 ASCLIN ASC Pins

CTS	RTS	RX	SCLK	SLSO	TX
P00.12	P00.9	P15.7	P00.0	P00.3	P00.0
		P11.0	P00.2	P12.1	P00.1
		P20.3	P11.1	P14.3	P11.0
		P32.2	P11.4	P21.2	P11.1
		P00.1	P15.6	P21.6	P15.6
		P21.6	P15.8	P33.1	P15.7
		P21.2	P20.0		P20.0
		P21.3	P21.5		P20.3
			P21.7		P21.7
			P32.3		P22.0
			P33.2		P22.1
					P32.2
					P32.3

4.4.1 Module Config options

Config Option	Default Value
loopBack	FALSE
clockSource	IfxAsclin_ClockSource_kernelClock
baudrate.prescaler	1
baudrate.baudrate	115200
baudrate.oversampling	IfxAsclin_OversamplingFactor_4
bitTiming.medianFilter	IfxAsclin_SamplesPerBit_one
bitTiming.samplePointPosition	IfxAsclin_SamplePointPosition_3
frame.idleDelay	IfxAsclin_IdleDelay_0
frame.stopBit	IfxAsclin_StopBit_1
frame.frameMode	IfxAsclin_FrameMode_asc
frame.shiftDir	IfxAsclin_ShiftDirection_lsbFirst
frame.parityBit	FALSE
frame.parityType	IfxAsclin_ParityType_even
frame.dataLength	IfxAsclin_DataLength_8
fifo.inWidth	IfxAsclin_TxFifoInletWidth_1
fifo.outWidth	IfxAsclin_RxFifoOutletWidth_1
fifo.txFifoInterruptLevel	IfxAsclin_TxFifoInterruptLevel_15
fifo.rxFifoInterruptLevel	IfxAsclin_RxFifoInterruptLevel_1
fifo.buffMode	IfxAsclin_ReceiveBufferMode_rxFifo
interrupt.rxPriority	0
interrupt.txPriority	0
interrupt.erPriority	0
interrupt.typeOfService	IfxSrc_Tos_cpu0
pins	NULL_PTR
rxBuffer	NULL_PTR
txBuffer	NULL_PTR
txBufferSize	0
rxBufferSize	0
dataBufferMode	Ifx_DataBufferMode_normal

5 QSPI

5.1 The SPI Protocol

SPI is a bi-directional, synchronous serial data protocol based on a master-slave relationship. A large number of devices support SPI due to its high transmission speeds and support for multiple devices on the same bus. The physical bus consists of 3 common wires and 1 wire unique to each slave. The 3 common wires are the SCLK, MOSI and MISO lines. The dedicated line is the slave select.

Table 9 QSPI Signals

Signal	Function	Comment
SCLK	Shift clock	Synchronous clock
MOSI	Master Out, Slave In	Master transmitter, slave receiver
MISO	Master In, Slave Out	Slave transmitter, master receiver
SLS	Slave select	SPI slave device select line

The clock line is used to keep the 2 data lines in sync and to indicate when to shift data out and when to read it in. The Master Out Slave In (MOSI) line is used to output data from the master. The Master In Slave Out (MISO) line is used by the slave to return data to the master. Finally the unique line is the slave select line and is used to distinguish which slave the master wants to communicate with. When the master wants to initiate communication with a slave, it drops that slave's select pin low, the clock line and the slave starts to listen on the MOSI.

In full duplex mode the master sends a bit on the MOSI and the slave then responds with a bit on the MISO half a clock cycle later. Both of these transfers will always happen, but they won't necessarily be meaningful. For instance, when the master is writing to the slave device, the slave won't necessarily be sending useful information back to the master. Often, the slave's response to data sent by the master will be contained in the reply to the master's second transmission. The SPI can be configured in other modes so that only the master will write and so on.

SPI can be regarded as having "modules" and "channels". A module is a SPI transmitter receiver block that can address multiple SPI slave devices that are individually enabled via chipselect or "slave select" lines. Here each slave select represents a SPI channel.

5.1.1 Why Queued SPI?

The Aurix TC275 has up to fifty-seven slave device selects served by four QSPI modules. In a typical application there might be a large number of SPI devices. For example, a typical application might have:

- Real Time Clock Chip e.g. PCF2123
- EEPROM e.g AT25010B
- SD card
- Wiznet Ethernet/TCPIP driver e.g. WZ5400
- Relay driver e.g. TLE7231G
- Smart power switches with SPI interfaces e.g. BTS54220
- LCD TFT display controller e.g. ILI9341
- SPI-programmable H-bridge e.g. TLE8209
- Safety monitor e.g. CIC61508 (note: must not share SPI module with any other device for safety reasons)

Each of these devices requires a slave select to enable it, plus each has different high-level protocol and hardware settings required to access it. For example, the data width could be anything from 4 to 16 bits and the shift clock speed could be up to 25MBit/s. The data might be latched on the falling or rising edge of the shift clock, sent MSB or LSB first and the clock might need to idle high or low. The result of this large number of device-specific configuration items is that the software required to manage the devices can become very complex and in the case of high speed devices, a major processing overhead.

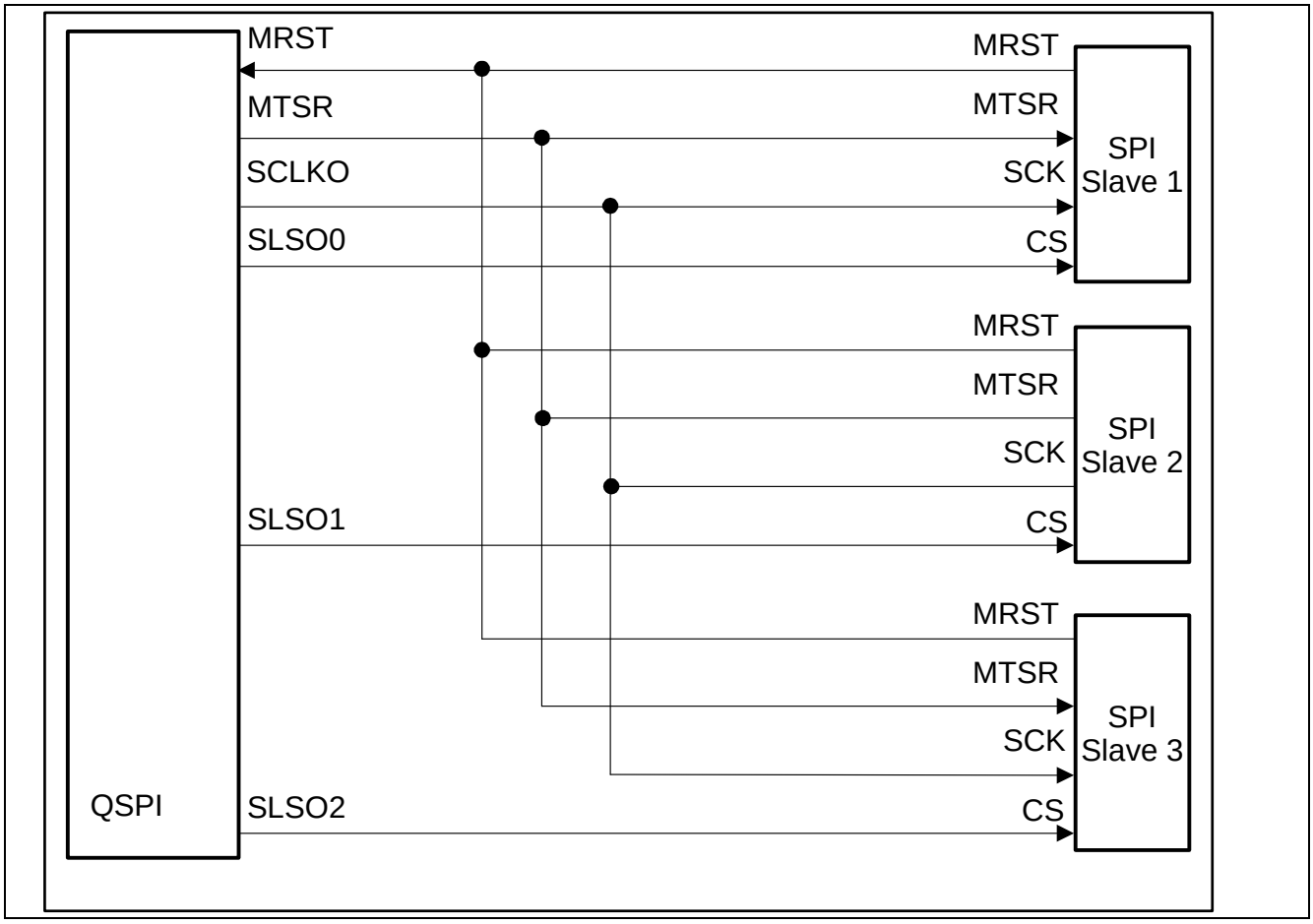


Figure 7 Multiple SPI Devices On A Single SPI Module

Normally the data to be transferred to a SPI device consists of a number of bytes (or words) in a defined sequence. These might be transferred to the SPI interface by a transmit interrupt that places each data item into the transmit buffer singly. More usually, some form of DMA is used whereby a sequence of data is held in an array and automatically transferred to the SPI transmit buffer without discrete service interrupts. This works well provided that each device connected to the SPI module has the same baudrate, data width, shift clock polarity and phase. If not, then before anything can be sent, the SPI module will need to be reconfigured to suit the hardware and protocol of the specific SPI device being addressed. This is complicated and time-consuming.

As has been said, the Aurix TC275 has up to fifty-seven slave device selects and four QSPI modules. This gives the potential of driving a very large number of devices. However, the number of different combinations of device settings could easily become unmanageable. In a simple SPI module implementation, before each device is addressed, the configuration of the SPI module would have to be reconfigured to suit that device i.e. data direction, clock phase, slave select timing. This can become a major software (and CPU) overhead.

To solve this problem, the Aurix has “queued SPI” modules. Here, the settings for each slave device are stored in a queue, interleaved with the actual data to be sent via SPI.

The “queue” is in a 32-bit array in general purpose RAM and contains two types of data:

Configuration data (how the data should be transmitted)
Transmit data (what data should be transmitted).

The simplest queue format interleaves the configuration and transmit data every other 32-bit word.

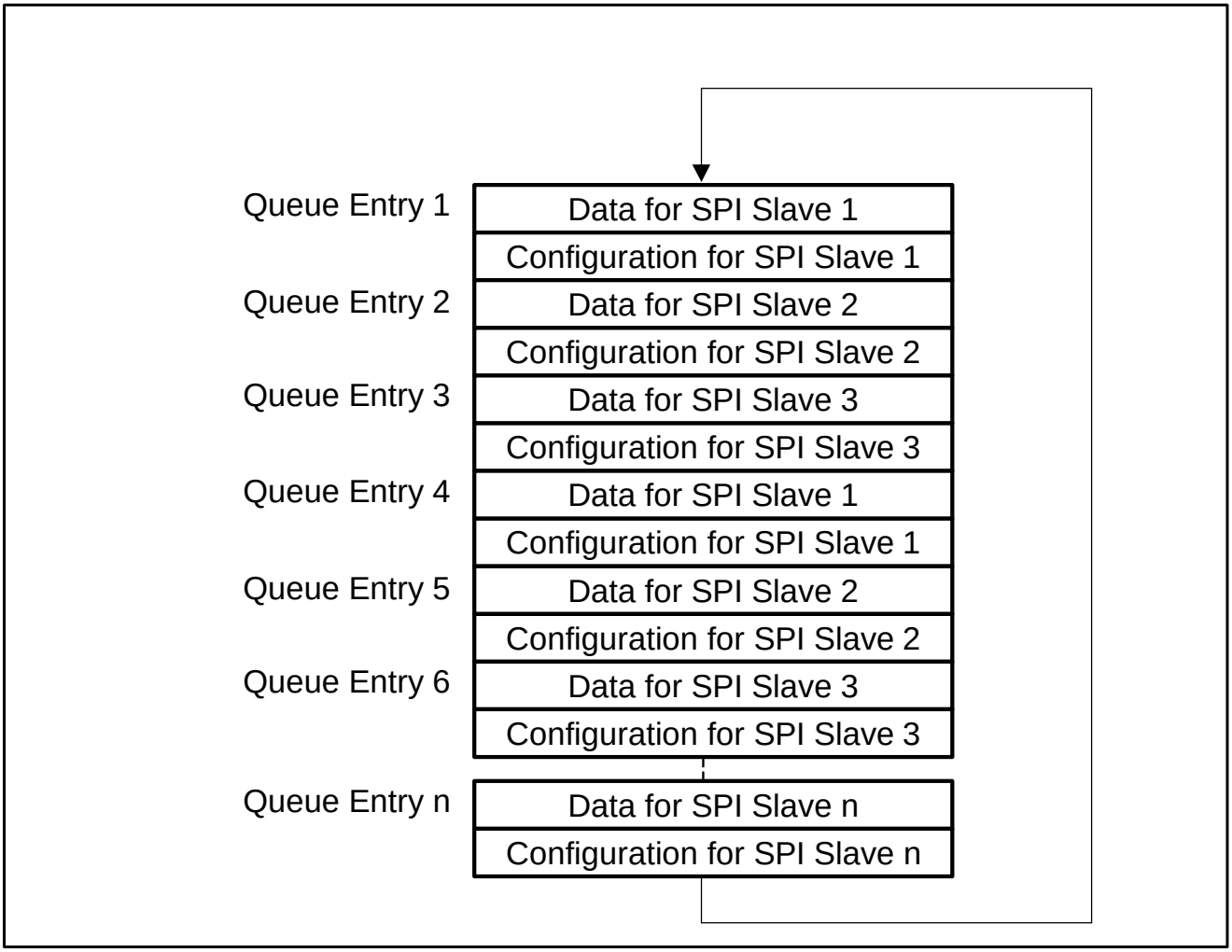


Figure 8 Queue for 3 SPI channels

Each transfer to a slave device consists of the following automatic operations.

Get the hardware configuration for the SPI device to be addressed from the next slot in the queue and write it to the Extended Configuration Register for the slave select channel to which the required SPI peripheral is connected.
Enable the slave select for the required SPI slave device
Get the data for that slave from the next slot in the queue and shift it out.
Disable the slave select.
Move onto the next queue slot to activate the next channel.

The above transaction is completely automatic and requires no intervention from the CPU(s) to operate, once the queued SPI module has been initially configured.

Using a queued approach enables efficient streaming of SPI data to large capacity devices such as SD cards or LCD displays. The queued data can be easily transferred to the QSPI peripheral using a single DMA channel with the SPI data format and timing configuration being changed automatically with no need for CPU involvement.

5.1.2 QSPI Summary

- Automatic hardware control of signal timing
- Automatic slave select signal activation
- Slave selects configurable for leading, trailing, and idle delay times
- Support for high speed SPI communications (up to 50 Mbit/s at 200 MHz shift engine clock)
- Programmable time quanta per bit (similar to CAN)
- Overrun protection – communication stop when receive FIFO is full
- Data lengths can be specified in bytes (instead of bits) for long data streams
- Support for hardware parity checking
- Control and data handling via DMA
- Option for differential MRST and MTSR signals on QSPI2 and QSPI3.
- Up to 16 slave select signals per SPI channel

Table 10 QSPI Module Slave Selects

QSPI Module	Slave Selects
QSPI0	14
QSPI1	11
QSPI2	13
QSPI3	19

5.2 A More Detailed Look At The QSPI

The iLLD driver for the QSPI takes care of the low-level configuration and operation of the QSPI module, but it is a good idea to have some idea what is going on at a low level, especially if debugging is required! To use the iLLD driver you will need to understand what settings are possible, especially as regards the timing and polarity of the SPI waveforms generated by the Aurix QSPI modules.

5.2.1 BACON and ECONz

The format of the SPI transmission is controlled by both the BACON and ECONz registers:

BACON = BAsic CONfiguration Register
ECONz = Extended CONfiguration Registers (z = 0..7)

The channel select bit field of the BACON register selects which ECON register will be used. Together, both registers define how data will be transmitted to each SPI slave device

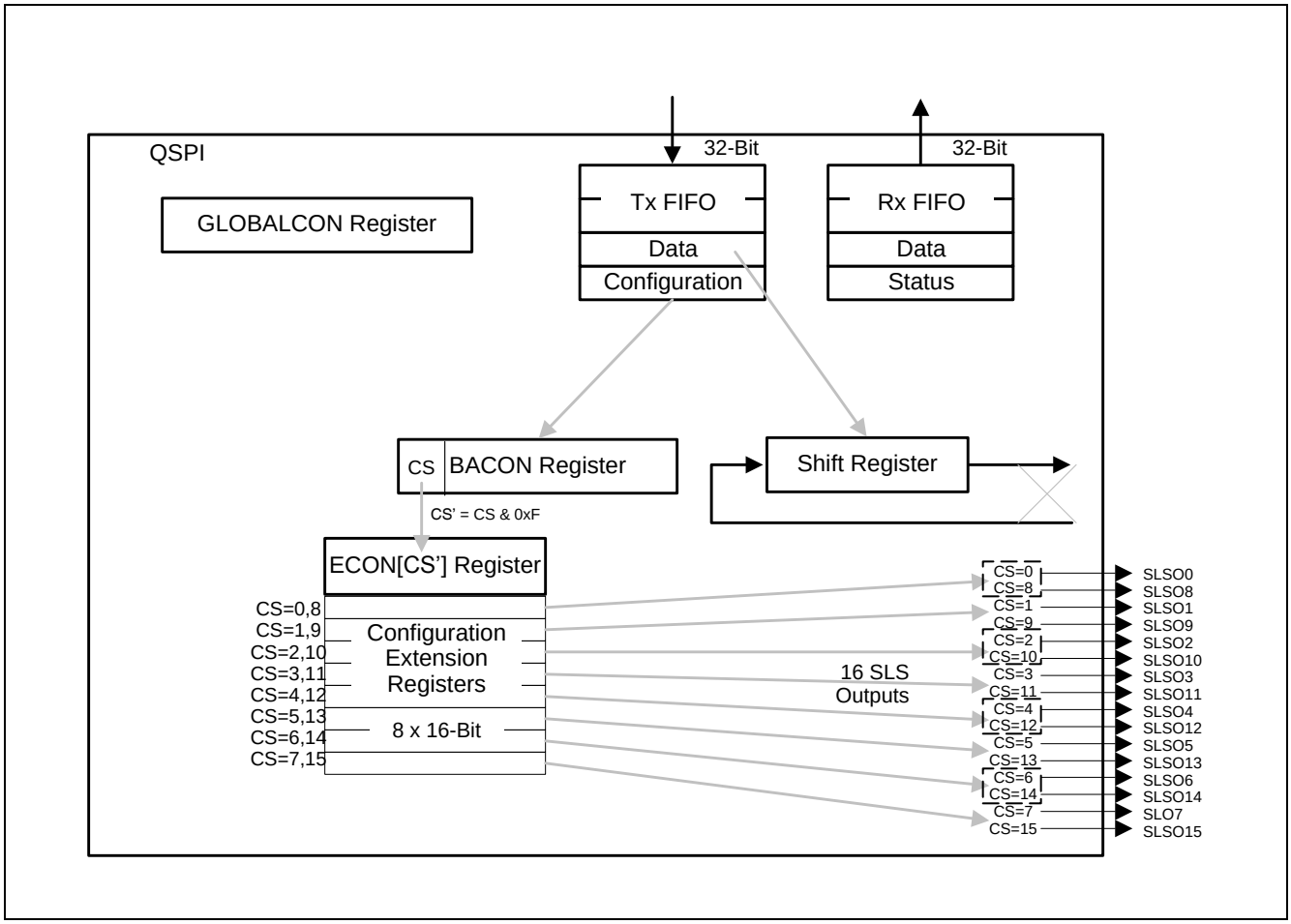


Figure 9 Detailed View of QSPI Internals

The most important thing to know is that the transmit configuration data in the queue contains an image of the BACON SFR. The top four bits represent the slave select channel to which the configuration data relates. Once the BACON image is loaded into the QSPI module the hardware uses these four bits to select a pre-configured ECON SFR which then controls the physical slave select pins.

5.2.1.1 BACON Sfr (BasicCONfiguration)

The BACON register defines basic configuration parameters for the current slave select.

BACON values and transmit data are streamed from the RAM queue to the QSPI Tx FIFO to dynamically change the timing and other characteristics of the SPI data transmitted.

Table 11 BACON Bits

Field	Range	Description	Description
Last	0 or 1	**0 = next data word will not end frame**	**1 = next data word is last of frame, slave select will be disabled**
IDLE, LEAD, TRAIL delays	Timing formulae or 0 to 7 SCLK half-cycles with iLLD API.	Defines idle, leading, and trailing delay times	See section 21.5.4 in TC275 user manual for full details of the calculation of Baud rates and timing delays.
Parity type	0 or 1	0 = even parity	1 = odd parity
User interrupt	0 or 1	1 = generate user interrupt at PT1 (phase transition) event	
MSB	0 or 1	0 = least significant bit transmitted first	1 = most significant bit transmitted first
Byte	0 or 1	0 = data length is in bits	1 = data length is in bytes
Data length	Feb-32	SPI word size in bits or bytes	
Channel select	0-15	Activates corresponding slave select output	1 = data length is in bytes

5.2.1.2 ECONz (ExtendedCONfiguration, 0-7)

The ECON 0..7 registers can be programmed directly. The values are typically static, and programmed just once at initialization. The active ECONz register is selected by the BACON channel select value.

Table 12 ECONz Bits

Field	Range	Description	Description
Q	1..64	**Time quanta divider for the channel**	**See section 21.5.4 in TC275 user manual for full details of the calculation of Baud rates and timing delays.**
A	1..4	Bit segment A width in time quanta units	
B	0..3	Bit segment B width in time quanta units	
C	0..3	Bit segment C width in time quanta units	If B=0, C is set to 1 by hardware
Clock phase	0 or 1	0 = no delay in SPI clock	1 = delay SPI clock by one half cycle
Clock polarity	0 or 1	0 = SPI clock idle low level	1 = SPI clock idle high level
Parity enable	0 or 1	0 = parity disabled	1 = parity enabled

5.2.2 SPI Baud Rates, Signal Timing & Polarity

See section 21.5.4 in TC275 user manual for full details of the calculation of Baud rates and timing delays.

One complete SPI data transaction with all the critical timings highlighted is shown below.

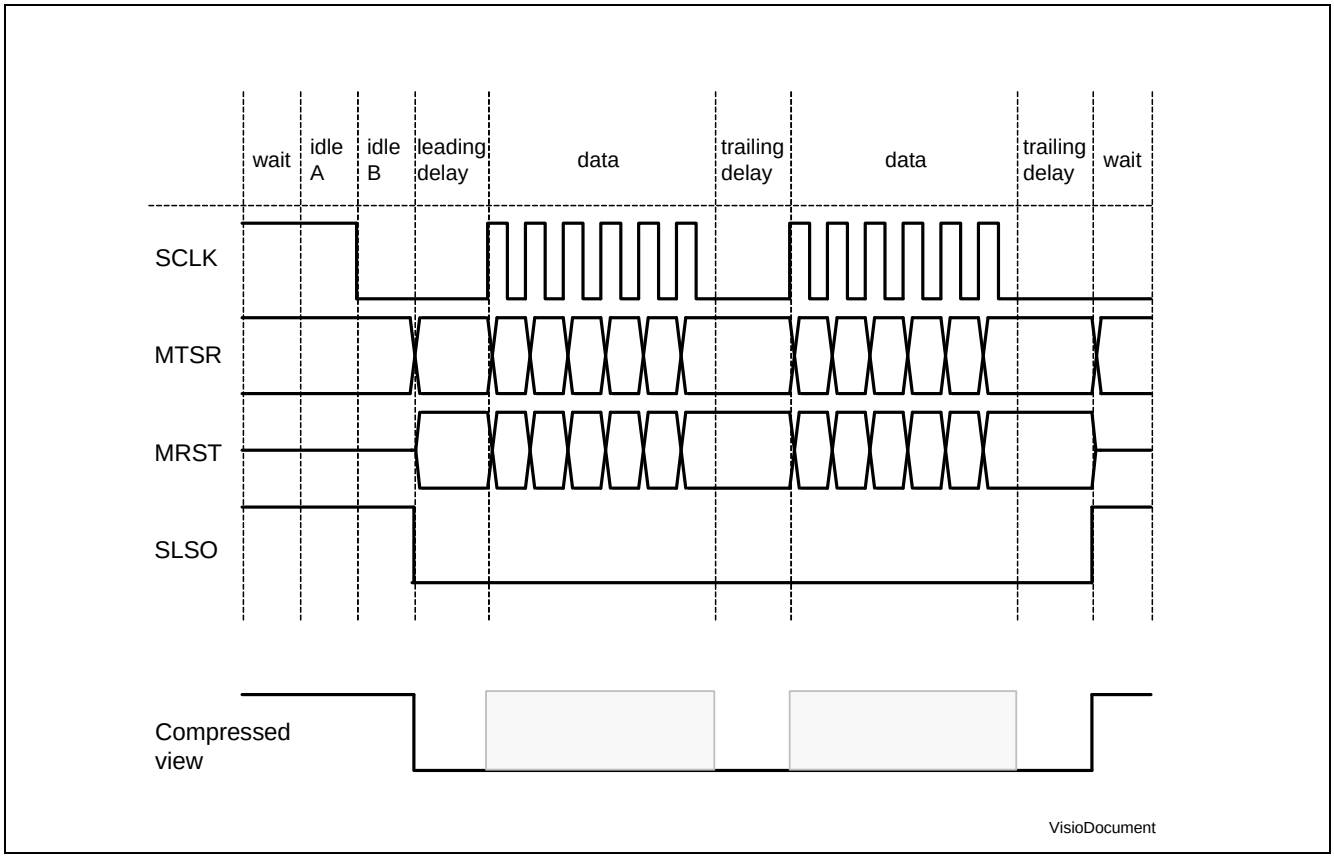


Figure 10 Typical QSPI Transaction

When interfacing the QSPI to a real SPI device, the following critical timings must be obtained from the device's data sheet.

- Idle delay
- Leading delay
- Trailing delay
- Maximum Baud rate

QSPI provides precise control over these delays with configuration bits in the BACON register. The iLLD QSPI configuration functions automatically calculate the necessary BACON and ECONz register contents, based on the timing requirements and Baud rate set by the user. However in the iLLD API for the QSPI, the length of the delays is expressed in units of one half of an SCLK cycle so the user needs to be aware that one SCLK cycle = 1/(Baud Rate) seconds. Thus at a typical 1MBit/s Baud rate, the unit of delay is 0.5us.

You will also need to know the:

Data heading: most or least significant bit first?
Is data shifted on the leading or trailing edge of the SCLK?
The number of bytes per message.

5.2.3 Note about SLSO numbering in TC275 User Manual:

The TC275 UM section 14 covering GPIO refers to the QSPI slave selects as “SLSO36”, “SLSO37”, “SLSO39” etc.. For example, pin Port 00.8:

O	General-purpose output	P00_OUT.P8
	GTM output	TOUT17
	QSPI3 output	SLSO36
	Reserved	–
	Reserved	–
	VADC output	VADCEMUX12
	SENT output	SPC7
	CCU61 output	CC61

This should be read as “SLSO, QSPI3, channel 6” not “SLSO 36”! Thus SLSO110 in the example below for pin Port 10.0 should be read as “SLSO, QSPI1, channel 10”.

O	General-purpose output	P10_OUT.P0
	GTM output	TOUT102
	Reserved	–
	QSPI1 output	SLSO110
	Reserved	–
	VADC output	VADCG6BFL0
	Reserved	–
	Reserved	–

5.3 Using the QSPI Via The iLLD API

The Aurix supports almost every combination of SPI configurations possible. All of these options can be changed in the ILLD API config structures and a full list of options is given later in this chapter. The default configuration is available which will cover the most common eventualities and so only the IO pins will need to be set and the ISR priorities for the module to function. Each slave has its own separate config structure this means that you can change how the data is output for each slave individually rather than changing the master config between transfers. This means that each slave can be individually set for different Baud rates and bus timings.

5.3.1 Basic Steps

The basic steps to use the QSPI ILLDs are as follows:

Instantiate the module, device and pin objects and config structures
Define the Interrupt routines
Use the config init functions to fill the config structures with default values
Change any config options that are required and set the pins.
Initialize the module, channels and install the interrupt routines.

5.3.2 QSPI iLLD Setup

First include the iLLD module:

```
#include <Qspi/SpiMaster/IfxQspi_SpiMaster.h>
```

Next we instantiate the module and device channel:

```
IfxQspi_SpiMaster spi;
IfxQspi_SpiMaster_Channel spiChannel;
```

Next the interrupt priorities are set and then the ISRs are defined:

```
#define IFX_INTPRIO_QSPI0_TX 1
#define IFX_INTPRIO_QSPI0_RX 2
#define IFX_INTPRIO_QSPI0_ER 5

IFX_INTERRUPT(qspi0TxISR, 0, IFX_INTPRIO_QSPI0_TX)
{
IfxQspi_SpiMaster_isrTransmit(&spiMaster);
}
IFX_INTERRUPT(qspi0RxISR, 0, IFX_INTPRIO_QSPI0_RX)
{
IfxQspi_SpiMaster_isrReceive(&spiMaster);
}
IFX_INTERRUPT(qspi0ErISR, 0, IFX_INTPRIO_QSPI0_ER)
{
IfxQspi_SpiMaster_isrError(&spiMaster);
}
001:
```

Next we install the interrupt handlers and enable interrupts globally:

```
/* install interrupt handlers */
IfxCpu_Irq_installInterruptHandler(&qspi0TxISR, IFX_INTPRIO_QSPI0_TX);
IfxCpu_Irq_installInterruptHandler(&qspi0RxISR, IFX_INTPRIO_QSPI0_RX);
IfxCpu_Irq_installInterruptHandler(&qspi0ErISR, IFX_INTPRIO_QSPI0_ER);
IfxCpu_enableInterrupts();
002:
```

Now the module config structure is created and filled with default settings:

```
003:
/* create module config */
IfxQspi_SpiMaster_Config spiMasterConfig;
IfxQspi_SpiMaster_initModuleConfig(&spiMasterConfig, &MODULE_QSPI0);
004:
```

Next the default settings are changed:

```
/* Set the desired mode and maximum baudrate */
spiMasterConfig.base.mode = SpiIf_Mode_master;
spiMasterConfig.base.maximumBaudrate = 10000000;

/* ISR priorities and interrupt target */
spiMasterConfig.base.txPriority = IFX_INTPRIO_QSPI0_TX;
spiMasterConfig.base.rxPriority = IFX_INTPRIO_QSPI0_RX;
spiMasterConfig.base.erPriority = IFX_INTPRIO_QSPI0_ER;
spiMasterConfig.base.isrProvider = (IfxSrc_Tos)IfxCpu_getCoreId();

/* pin configuration */
const IfxQspi_SpiMaster_Pins pins = {
&IfxQspi0_SCLK_P20_11_OUT, IfxPort_OutputMode_pushPull,  /* SCLK */
&IfxQspi0_MTSR_P20_14_OUT, IfxPort_OutputMode_pushPull,  /* MTSR */
&IfxQspi0_MRSTA_P20_12_IN, IfxPort_InputMode_pullDown,   /* MRST */
IfxPort_PadDriver_cmosAutomotiveSpeed3 /* pad driver mode */
};
spiMasterConfig.pins = &pins;
```

Then the module is initialized:

```
/* initialize module */
IfxQspi_SpiMaster spi;
/* defined globally */
IfxQspi_SpiMaster_initModule(&spi, &spiMasterConfig);
```

Now the slave channel config is setup:

```
/* create channel config */
IfxQspi_SpiMaster_ChannelConfig spiMasterChannelConfig;
IfxQspi_SpiMaster_initChannelConfig(&spiMasterChannelConfig, &spi);
```

Then the default settings are changed:

```
/* set the baudrate for this channel */
spiMasterChannelConfig.base.baudrate = 5000000;
/* select pin configuration */
const IfxQspi_SpiMaster_Output slsOutput = {
&IfxQspi0_SLSO7_P33_5_OUT,
IfxPort_OutputMode_pushPull,
IfxPort_PadDriver_cmosAutomotiveSpeed1
};
spiMasterChannelConfig.sls.output = (IfxQspi_SpiMaster_Output)slsOutput;
```

Finally the channel is initialized:

```
/* initialize channel */
IfxQspi_SpiMaster_initChannel(&spiChannel, &spiMasterChannelConfig);
```

That concludes the setup for the QSPI peripheral.

5.3.3 QSPI Example

The example applications include a full example of the QSPI working with 2 SPI EEPROM chips on the same QSPI module but using different slave selects.

5.3.4 Important functions

5.3.4.1 Qspi_Transfer

This function is the main function used for transferring data over the QSPI interface.

```
IfxQspi_SpiMaster_exchange(&spiChannel, &Txdata[0], &RxData[0], 1);
```

This function takes 4 arguments:

1. The QSPI channel over which the data should be transmitted.
2. A pointer to the data which is to be transmitted over the QSPI interface.
3. A pointer to the data which is to be received from the QSPI interface.
4. The number of bytes to be transmitted over the QSPI interface.

5.3.4.2 Qspi getStatus

This function returns the current state of the bus and so can be used to delay the program until the SPI bus is ready. As shown in the example below.

```
while( IfxQspi_SpiMaster_getStatus(&spiEEPROM0Channel) ==
                                                SpiIf_Status_busy ) {;}
```

This function takes 1 argument:

```
The SPI channel to be checked.
```

This function returns:

```
SpiIf_Status_ok
SpiIf_Status_busy
SpiIf_Status_unknown
```

5.4 QSPI Config Options

5.4.1 Changing The QSPI Configuration Directly

Important QSPI settings can be changed directly in the user's initialization code:

```
/* set the baudrate for this channel */
spiMasterChannelConfig.base.baudrate = 100000;
spiMasterChannelConfig.base.mode.dataHeading = 1; /* MSB first */
spiMasterChannelConfig.base.mode.shiftClock = 1;
spiMasterChannelConfig.base.mode.dataWidth = 8;
spiMasterChannelConfig.base.mode.enabled = 1;
spiMasterChannelConfig.base.mode.clockPolarity =
SpiIf_ClockPolarity_idleLow;

/* Really conservative timings to cope with any SPI device */
spiMasterChannelConfig.base.mode.csTrailDelay = SpiIf_SlsoTiming_7;
spiMasterChannelConfig.base.mode.csLeadDelay = SpiIf_SlsoTiming_7;
spiMasterChannelConfig.base.mode.autoCS = 1; /* Automatic Chip select */
```

This configuration is then written into the QSPI module with:

```
/* Initialize channel */
IfxQspi_SpiMaster_initChannel(&spiChannel, &spiMasterChannelConfig);
```

5.4.2 Module Config Options

Table 13 QSPI Driver Configuration Options

Config option	Default Value	Value Range
allowSleepMode	FALSE	Boolean
pauseOnBaudrateSpikeErrors	FALSE	Boolean
pauseRunTransition	IfxQspi_PauseRunTransition_pause	
enabledInterrupts.tx	TRUE	Boolean
enabledInterrupts.rx	TRUE	Boolean
enabledInterrupts.pt1	FALSE	Boolean
enabledInterrupts.pt2	FALSE	Boolean
enabledInterrupts.usr	FALSE	Boolean
txFifoThreshold	IfxQspi_TxFifoInt_1	
rxFifoThreshold	IfxQspi_RxFifoInt_0	
pins	NULL_PTR	pinStructure
dma.rxDmaChannelId	IfxDma_ChannelId_none	
dma.txDmachannelId	IfxDma_ChannelId_none	
dma.rxDmaChannelPriority	0	
dma.txDmaChannelPriority	0	
dma.useDma	FALSE	Boolean
rxPriority	None	
txPriority	None	
erPriority	None	
isrProvider	None	

5.4.3 Channel Config Options

Table 14 QSPI Driver Channel Configuration Options

Config Option	Default
baudrate	0
mode.enabled	1
mode.autoCS	1
mode.loopback	0
mode.clockPolarity	SpiIf_ClockPolarity_idleLow
mode.shiftClock	SpiIf_ShiftClock_shiftTransmitDataOnLeadingEdge
mode.dataHeading	SpiIf_DataHeading_msbFirst
mode.dataWidth	8
mode.csActiveLevel	Ifx_ActiveState_low
mode.csLeadDelay	SpiIf_SlsoTiming_0
mode.csTrailDelay	SpiIf_SlsoTiming_0
mode.csInactiveDelay	SpiIf_SlsoTiming_0
mode.parityCheck	0

Config Option	Default
mode.parityMode	Ifx_ParityMode_even
errorChecks.baudrate	0
errorChecks.phase	0
errorChecks.receive	0
errorChecks.transmit	0

5.4.4 Pin Structure

The pins to be used for a QSPI channel are defined via a structure with a predefined typedef and macros.

```
const IfxQspi_SpiMaster_Pins pins = {
      &IfxQspi1_SCLK_P10_2_OUT, IfxPort_OutputMode_pushPull,     // SCLK
      &IfxQspi1_MTSR_P10_3_OUT, IfxPort_OutputMode_pushPull,     // MTSR
      &IfxQspi1_MRSTA_P10_1_IN, IfxPort_InputMode_pullDown,      // MRST
      IfxPort_PadDriver_cmosAutomotiveSpeed3 // pad driver mode
      };
```

The possible pins that support QSPI are given in the following tables. Useful macros and structures for accessing the QSP in ‘C’ may be found in the iLLD programming examples:

```
.\0_Src\4_McHal\Tricore\_PinMap\IfxQspi_PinMap.c
.\0_Src\4_McHal\Tricore\_PinMap\IfxQspi_PinMap.h
.
```

Table 15 QSPI0 Port Pins

MRSTA	MTSRA	SCLKA	SLSIA	SLSOx	SLSOx	SLSOx
P20.12	P20.14	P20.11	P20.13	P20.8	P11.10	
P22.9	P22.10	P22.8	P20.9	P22.11	P11.11	
P22.6	P22.5	P22.7		P23.6	P11.2	
P20.12	P20.12	P20.11		P22.4	P20.10	
P22.6	P20.14	P20.13		P15.0	P33.5	
P22.9	P22.10	P22.7		P20.9	P20.6	
	P22.5	P22.8		P20.13	P20.3	

Table 16 QSPI1 Port Pins

MRSTA	MTSRA	SCLKA	SLSIA	SLSOx	SLSOx	SLSOx
P10.1	P10.3	P10.2	P11.10	P20.8	P33.10	P11.10
P11.3	P11.9	P11.6		P10.0	P33.5	P11.11
P10.1	P10.4	P10.2		P20.9	P10.4	P11.2
P10.6	P10.1	P11.6		P20.13	P10.5	
P11.3	P10.3					
	P10.4					
	P11.9					

Table 17 QSPI2 Port Pins

MRST	MTSR	SCLK	SLSIx	SLSOx	SLSOx
P15.4	P15.5	P15.3	P15.2	P15.2	P33.13
P15.7	P15.6	P15.8	P15.1	P34.3	P20.10
P21.2	P34.5	P33.14		P33.15	P20.6
P21.3	P13.2	P13.0		P32.6	P20.3
P34.4	P13.3	P13.1		P14.2	
P15.2	P15.5	P15.3		P14.6	
P15.4	P15.6	P15.6		P14.3	
P15.7	P34.5	P15.8		P14.7	
P34.4		P33.14		P15.1	

Note: P21.3 only to be used for differential mode where it is the “MRSTP” signal.

Table 18 QSPI3 Port Pins

MRST	MTSR	SCLK	SLSI	SLSOx	SLSOx
P02.5	P02.6	P02.7	P02.4	P02.4	P02.8
P10.7	P10.6	P10.8	P01.3	P01.4	P23.4
P01.5	P01.6	P01.7	P33.10	P33.10	P00.8
P33.13	P33.12	P33.11	P22.2	P22.2	P00.9
P22.1	P22.0	P22.3		P23.1	P33.7
P21.2	P22.2	P22.0		P02.0	P10.5
P21.3	P22.3	P22.1		P33.9	P01.3
P01.5	P01.6	P01.7		P02.1	
P02.5	P02.6	P02.7		P33.8	
P10.7	P10.6	P10.8		P02.2	
P22.1	P22.0	P22.3		P02.3	
P33.13	P33.12	P33.11		P23.5	

Note: P21.3 only to be used for differential mode where it is the “MRSTP” signal. P21.2 (“MRSTN”) must be used for the MRST in standard QSPI mode.

6 I2C Module

6.1 I2C Protocol

I2C is a simple short distance, bi-directional communication protocol, which uses 2 data lines to transmit data serially to many devices on a common bus. All drivers are open collector so external pull-up resistors are required. There are a huge number of IO, memory and specialist devices that use I2C as the serial interface. It is set up in a master-slave relationship, so that the master always initiates data transfers and the slave will only ever respond when asked to by the master. I2C uses a data line and a clock line. The clock line is used to register when the state of the data line should be read and so keep the master and slave in sync. The clock is only ever driven by the master. Since the data line is connected to many devices, the master needs to be able to distinguish between which devices to send to. This is done by sending an address byte before a data transfer. Each device has a unique address allowing for up to one hundred and twelve 7-bit address devices on a single bus, or up to 1024 devices if the 10-bit mode is used. The 8th bit is used to indicate whether the master wants to write to or read from the slave. Whenever a slave device detects that its own address has been sent by the master, it will generate an ACKNOWLEDGE signal by pulling the data line low for one clock period. This is detected by the master which uses it as a trigger to send more data.

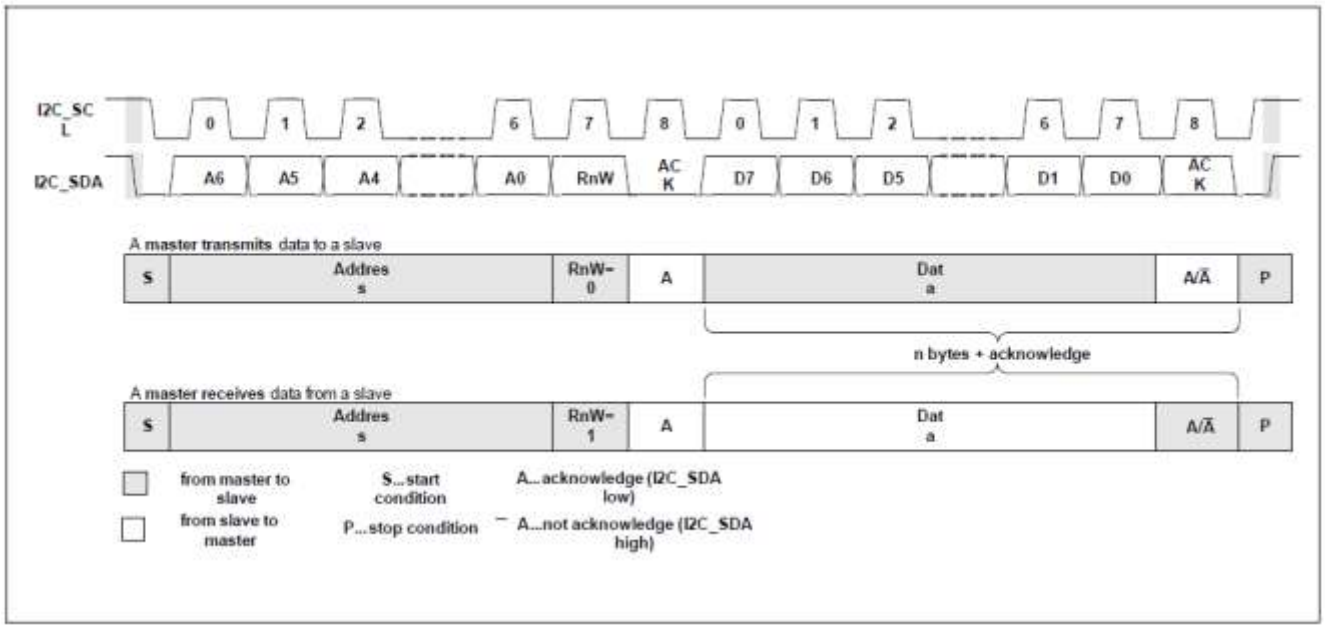


Figure 11 7-Bit Address Mode

In 10-bit address mode, the slave address must be of the form “1,1,1,1, 0, A9, A8” with the R/W bit set of zero. The two most significant bits of the 10-bit address are placed into A9 and A8 respectively to form a device address in the range 0x78 to 0x7B, which is ignored by all 7-bit devices. The 10-bit device must generate an ACKNOWLEDGE, triggering the master to send the remaining 8-bits of the address. These bits are received by the 10-bit device which will then generate the ACKNOWLEDGE. 7-bit devices will not respond to this second byte as it was not preceded by a START condition. The number of addresses supported is 112 for the 7-bit mode and 1024 for 10-bit.

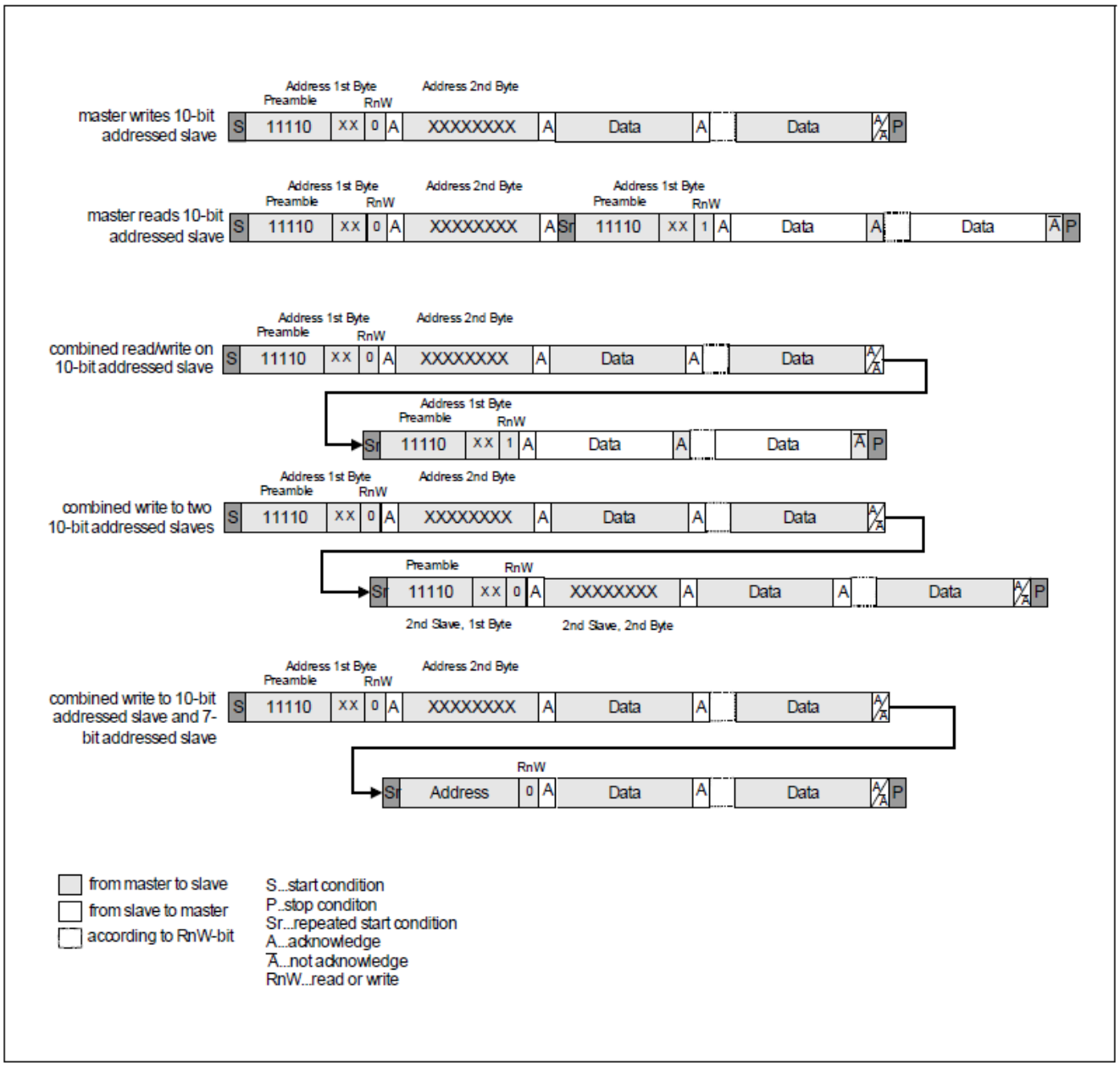


Figure 12 10-Bit Address Mode

Data transmission speeds are normally up to 100kbit/s (standard mode) or 400kbit/s in fast mode. There is no difference in the protocol between these speed ranges. However, the 3.4Mbit/s high speed mode uses a special “master code” in the address position to signal that the bit rate is about to rise.

6.2 Aurix I2C Implementation

6.2.1 I2C Module Main Features Quick View

- Compatible with I2C-bus specification version 2.1
- Master mode, multi-master mode and slave mode supported
- Standard mode (0-100 kbit/s), Fast mode (0-400 kbit/s), High-speed mode (0-3.4 Mbit/s)
- 7-bit and 10-bit I2C-bus addressing supported
- (De)Serialization of the bus data
- Generation/detection of start and stop signal
- Generation/detection of acknowledge signal
- Bus state detection
- Bus access arbitration in multi-master mode
- Recognition of device address in slave mode
- Detection of general call address
- Configurable repeated start in master mode
- I2C-bus signal timing adjustment
- 8 Tx/Rx FIFO stages
- Configurable data alignment (byte, half word, word)
- Configurable sizes for burst, transmit and receive package
- 4 data transfer interrupts (burst, last burst, single, last single)
- Interrupts for address match, general call, master code,
- arbitration lost, not-acknowledge received, transmission end, receive mode

The Aurix I2C module supports the original I2C 7-bit addressing or the 10-bit address mode and is compliant with I2C Specification 2.1, January 2000. It can act as both a master or slave device with data transmission speeds at up to 100kbit/s in standard I2C mode or 400kbit/s in fast mode. The latest 3.4Mbit/s high-speed mode is also possible, but great care must be taken in the selection of the pad driver characteristics and the timing of the SDA and SCLK signals (I2C_TIMCFG). Clock stretching by slow slaves is also possible at the high speeds, but to support this the SCL must be configured as open drain. The number of devices on the bus is limited only by the number of available addresses and a total capacitive load of 400pF.

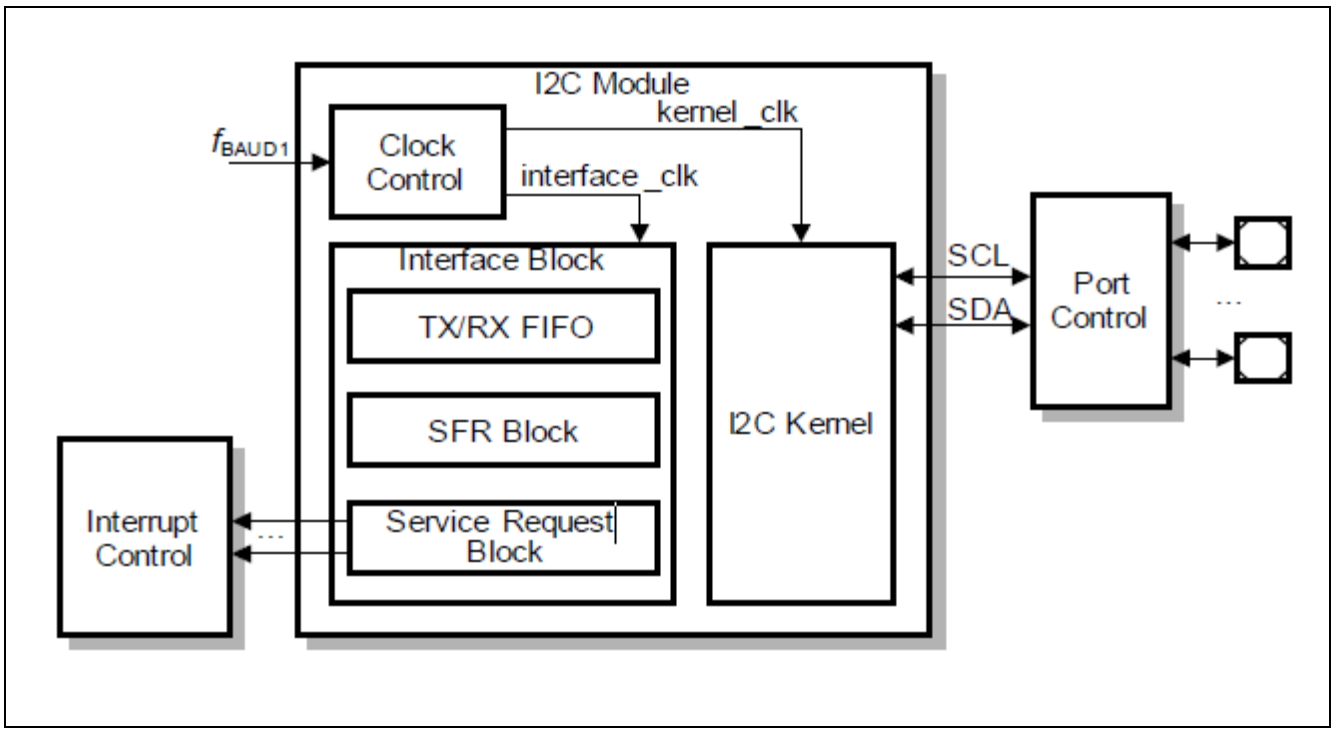


Figure 13 Aurix I2C Module

The I2C module clock is derived from the Aurix system clock via a prescaler. An additional fractional divider generates the desired bit rate. The exact timing of the I2C-bus signals can be adjusted via the TIMCFG SFR but the default values will work for most situations at up to 400kbit/s. These can be changed in the IfxI2c_setBaudrate() function.

```
TIMCFG.B.SDA_DEL_HD_DAT = 0x3F;
TIMCFG.B.FS_SCL_LOW     = 1;
TIMCFG.B.EN_SCL_LOW_LEN = 1;
TIMCFG.B.SCL_LOW_LEN    = 0x20;
```

Here, FS_SCL_LOW = 0 for operations up to and including 100kBit/s and ‘1’ for high speed operation up to 400kBit/s. The values stated here can be used as a starting point, but depending on the I2C slave devices employed, these may need to be changed. For more information on this, please refer to the TC275 user manual:

[1] User Manual Ref: I2C TIMCFG SFR explanation, pages 32-60
[2] User Manual Ref: I2C Baud rate generation, section 32.2.2.1

A FIFO is used for data transfer between the CPU and I2C module during transmission and reception. This decoupling of the transmit and receive registers allows writing and reading of multiple bytes without tying up a CPU.

The I2C module manages the data transmission and reception automatically and the user application is relieved of routine tasks such as detecting ACKNOWLEDGE,

NOTACKNOWLEDGE and generating START, STOP and RESTART conditions, so that the I2C can be viewed as a simple data source/sink.

Six separate interrupt requests can be generated; available for filling or emptying the FIFO, for reacting on certain protocol events and for handling of errors. However, for most common applications (EEPROMs, IO, RTC etc.), these refinements need not be utilized. For safety applications though, the error interrupt should be used to detect problems such as FIFO transmit/receive overflow or underflow. Such errors are not routed to the SMU.

Table 20 I2C Interrupt Sources

Signal Name	Name in Module	Explanation
SRC_I2CmBREQ	BREQ_INT	Burst Data Transfer
SRC_I2CmLBREQ L	LBREQ_INT	Last Burst Data Transfer
SRC_I2CmSREQ	SREQ_INT	Single Data Transfer
SRC_I2CmLSREQ	LSREQ_INT	Last Single Data Transfer
SRC_I2CmERR	ERR_INT	Error
SRC_I2CmP	P_INT	Protocol

Note: At the time of writing, interrupts must be disabled for single data transfers of more than 32 bytes. For most I2C devices, this should not be an issue.

To allow application software to react to the progress of I2C transactions, the protocol interrupt is triggered by 7 sources. This makes detecting common bus states easy. These states are listed below.

Table 21 I2C Bus Event Interrupt Sources

Protocol Event	Mode	I2C Bus Event
address match	Slave	A master has sent a device address for this slave
general call	Slave	A master has sent a general call transmission (address = 0x00, R/W bit = 0)
arbitration lost	Master	Another master had control of the bus when start of transmission was attempted (SDA = 1 overwritten by SDA = 0 from other master).
not-acknowledge received	Master	Slave device did not acknowledge its address (did not pull SDA low in 9th bit period)
transmission end	Master/slave	All bits have been sent
receive mode	Slave	Master has sent a write command
master code	Slave, 3.4Mbit/s high speed	Master code transmission

Protocol Event	Mode	I2C Bus Event
address match	Slave	A master has sent a device address for this slave
	mode	detected.

External I2C devices are connected to the module via a pair of receive and transmit pins. Three pairs of SCL/SDA pins are possible, that may be taken in any combination from the table below.

Table 22 I2C Module Available Pins

SCL	**P02_5**	**P13_1**	**P15_4**
SDA	**P02_4**	**P13_2**	**P15_5**

In a multi-master environment, or where clock-stretching by slow slaves is required, the SCL must be configured as open drain with an external pull-up resistor, typically 4K7 on 5V pins and 3K3 on 3.3V pins. SDA is always configured as open drain with a pull-up resistor. By default the output drivers for both pins are set to strong, sharp edge with automotive hysteresis (see chapter on GPIO Ports).

In most applications where the Aurix will be the only master, no clock stretching is required and there are many I2C devices, it is advisable to explicitly set the SCL pin to push-pull mode so that no external pull-up resistor is needed. This also true where the fast or high speed modes are used.

Note: When an Aurix pin is used as an input only, internal pull-up and pull-down resistors can be selected, as well as no pull device at all. In cases like I2C SDA line where the pin is both an output and input, the open drain output mode must be selected, but as a result the internal pull-up option cannot be used. An external resistor must therefore be fitted.

6.3 Using I2C Via The iLLD

The basic steps required to use the I2C are:

5. Selection of I2C module via output port multiplexer
6. Configuration of output port function with open drain
7. Configuration of pad driver characteristics
8. Selection of I2C input lines

These steps are now illustrated using the I2C iLLD.

Firstly you will need to instantiate the module handler. This should be done globally.

```
static IfxI2c_I2c i2c;
```

You will also need to instantiate an I2C device handler. You will need to do this for each device you will have on the I2C bus. This allows you to have different config settings for each device on the bus.

```
static IfxI2c_I2c_Device i2cDev; /* slave device handle
```

Next the config structures need to be instantiated and then filled with default values.

```
/* create config structure */
IfxI2c_I2c_Config config;
/* fill structure with default values and Module address
IfxI2c_I2c_initConfig(&config, &MODULE_I2C0);
```

Next you need create a pin structure containing the pins to be used for the data and the clock.

```
/* configure pins */
const IfxI2c_Pins pins = {
&IfxI2c0_SCL_P02_5_INOUT,
&IfxI2c0_SDA_P02_4_INOUT,
IfxPort_PadDriver_cmosAutomotiveSpeed1
};
```

Change some of the default values such as the baudrate and set the pins setting to the pin structure we created earlier.

```
config.pins = &pins;
config.baudrate = 400000; /* 400 kHz */
```

Now we initialize the module.

```
/* initialize module */
IfxI2c_I2c_initModule(&i2c, &config);
```

For greatest flexibility, the default pin configuration sets both SCL and SDA to open drain mode so external pull-up resistors must be fitted. Where the Aurix is the sole master, the SCL is best set to be a push-pull output by adding an additional line.

```
/* Force SCL to be a push-pull output rather than default */
/* open drain (Pxx_IOCR.PCy = 0x15 not 0x1D) */
IfxPort_setPinMode (&MODULE_P02, 5, IfxPort_OutputIdx_alt5);
```

Now we need to setup each of the I2C devices. Starting with instantiating the device config and filling it with default values.

```
/* create device config */
IfxI2c_I2c_deviceConfig i2cDeviceConfig;
/* fill structure with default values and i2c Handler
IfxI2c_I2c_initDeviceConfig(&i2cDeviceConfig, &i2c);
```

Next we set the address of the device.

```
/* set device specifig values */
i2cDeviceConfig.deviceAdddress = 0xa0;
```

Finally we initialize the device handle.

```
/* initialize the i2c device handle */
IfxI2c_I2c_initDevice(&i2cDeviceConfig,&i2cDev);
```

That completes the setup of the I2C module. To add more devices all you have to do is repeat the device setup.

To use the I2C all you need are 2 simple functions which read or write to the bus.

```
IfxI2c_I2c_write(&i2cDev, data, size)
IfxI2c_I2c_read(&i2cDev, data, size)
```

Both of these functions return status values indicating whether the read/write was successful and if not what went wrong. These are:

```
  IfxI2c_I2c_Status_ok = 0,
  IfxI2c_I2c_Status_nak = 1,
  IfxI2c_I2c_Status_al = 2,
  IfxI2c_I2c_Status_busNotFree = 3,
  IfxI2c_I2c_Status_error = 4
```

This can be helpful for error handling and debugging purposes.

6.4 I2C Working Example

An example of the I2C module driving a MCP27017 port expander and a 24LC256 EEPROM on the same bus can be found at:

```
.\ExampleCode\BaseFramework_TC27xC_iLLD01010_I2C_EEPROM
```

6.5 I2C iLLD Key Functions

6.5.1 Read from I2C device

Prototype

```
IFX_EXTERN IfxI2c_I2c_Status IfxI2c_I2c_read(
                      IfxI2c_I2c_Device *  i2cDevice,
                       uint8 *  data,
                       Ifx_SizeT  size
                     )
```

Usage

```
/* Read byte back from EEPROM */
/* Send Address to read (this will hang until ACK is received */
/* after programming timeout 5ms) */
while(IfxI2c_I2c_write(&i2cDev, I2c_Buffer, 2) == IfxI2c_I2c_Status_nak)
{ ; }

/* read device data to data array */
while(IfxI2c_I2c_read(&i2cDev, I2c_RxTxBuffer, 1) == IfxI2c_I2c_Status_nak)
{ ; }
```

6.5.2 Write To I2C Device

Prototype

```
IFX_EXTERN IfxI2c_I2c_Status IfxI2c_I2c_write  (
                      IfxI2c_I2c_Device *  i2cDevice,
                       uint8 *  data,
                       Ifx_SizeT  size
                     )
```

Usage

```
/* read device data to data array */
while(IfxI2c_I2c_write(&i2cDev, I2c_Buffer, 3) == IfxI2c_I2c_Status_nak)
{ ; }
```

Note: The I2c read and write functions send the device address and check for an ACK to make sure that the bus is free before sending the device address again followed by the "size"

bytes received as parameters. This can be seen on a bus analyser as two sequential device address sends.. If no ACK is received the function returns with `IfxI2c_I2c_Status_nak`.

6.6 10-Bit Address Devices

10 bit address has a prefix in the high part of the address (0xF0). So, in the low level drivers you can set the device address in the initialisation. Thus for 7 bit addressing, this is just the address. Then once initialised, you just set up the payload and call the write function. What I did for 10 bit addressing was to set the device address to the high part of the address and the prefix...

```
// Set the slave device for a 10 bit address
i2cDevConfig.deviceAddress = (ADDRESS >> 7)|(TENBITPREFIX);
```

And then add the low part of the address to the start of the payload before it is written.

```
void I2CWRITE2BYTES (uint16 Address, uint8 Command, uint16 Data) {
   uint8 TxDat[4]; /* To write 2 bytes of data, 6 frames are required.
                      The high part of the address (inc 11110) + start
                      condition
                      The low part of the address
                      The command
                      The 2 data bytes
                      The stop condition
                      Currently the assumption is that the low level driver
                      function will generate the start condition and the
                      stop so the data payload needs to be 4 bytes long –
                      low address, command, data1 and data2
                      */

   TxDat[0] = (uint8)Address;
   TxDat[1] = Command;
   TxDat[2] = Data>>8;
   TxDat[3] = (uint8)Data;

   while(IfxI2c_I2c_write(&i2cDev, TxDat, 4) ==
               IfxI2c_I2c_Status_nak)   /* Keep writing until the slave
                                           responds */
     { ; }
```

6.7 I2C iLLD Configuration Options

6.7.1 I2C Module Configuration

Table 23 I2C iLLD Pin Configuration

Config Options	Default Value	Usage

Pins	NULL_PTR	`const IfxI2c_Pins pins = {`
		`  &IfxI2c0_SCL_P02_5_INOUT, /*Clock Pin */`
		`  &IfxI2c0_SDA_P02_4_INOUT, /*Data pin */`
		`  IfxPort_PadDriver_cmosAutomotiveSpeed1};`
		
		`moduleConfig.pins = &pins;`

Baudrate	400000	`i2cModuleConfig.baudrate = 400000U;`

6.7.2 I2C Device Configuration

Table 24 I2C Device Configuration

Config Options	Default Value	Usage
Device Address	0xff	i2cDeviceConfig.deviceAddress = 0xFFU;

6.8 I2C Pin Locations

6.8.1 I2C Module 0 Pins

Table 25 I2C Module 0 Pins

Data Line	Pin	Name
Data(SDA)	P02.4	IfxI2c0_SDA_P02_4_INOUT
	P13.2	IfxI2c0_SDA_P13_2_INOUT
	P15.5	IfxI2c0_SDA_P15_5_INOUT
Clock(SCL)	P02.5	IfxI2c0_SCL_P02_5_INOUT
	P13.1	IfxI2c0_SCL_P13_2_INOUT
	P15.4	IfxI2c0_SCL_P15_4_INOUT

7 Aurix Timers

7.1 GPT120

GPT120 consists of two timer blocks, GPT1 and GPT2. These can run independently to generate PWM or measure pulse, or as a combined unit to implement an encoder input function.

7.1.1 Using GPT1

GPT1 consists of three 16-bit timers (Timer2, Timer3 & Timer4) plus a number of I/O pins. It can be used to make period measurements, generate pulsetrains or PWM and is based on a maximum input frequency of 100MHz. The T2IN, T3IN and T4IN input pins can be used as clock sources for their respective timers. T2IN and T4IN are also able to trigger a capture of the free running timer 3. If the timing functions are not required, the GPT1 input pins, T2IN, T4IN and T3IN, can be used to generate interrupts. There are dozens of different ways of using GPT1 but what follows are typical applications

7.1.2 Using GPT2

This is a block of two 16-bit timers (Timers 5 and 6) that can operate at up to 100MHz on a 200MHz CPU. The input clock can be a pre-scaled version of the CPU clock, or an external input signal, up to 25MHz. Like GPT2, the timers can be concatenated to produce a 32-bit timer. However it can do some clever tricks, such as the multiplication of an input frequency applied to the CAPIN pin or period measurement with zero CPU intervention.

Some typical GPT2 applications are:

- Timebase generation*****
- Pulse generation
- Time-between-edge measurements*****
- Two-channel software UART*****
- TV line capture and buffering*****
- Pulse position modulation receiver for TV remote control*

7.1.3 GPT120 Encoder Mode

GPT12 has a special mode for interfacing to incremental (quadrature) encoders where it is able to autonomously measure the speed and direction of two pulsestreams with a 90 degree offset between them. The two signals “A” and “B” must be applied to T3INx (P2.6 or P10.4) and T3EUD (P2.7 or P10.7) with an optional mechanical zero position signal connected to T4INx (P2.8 or P10.8).

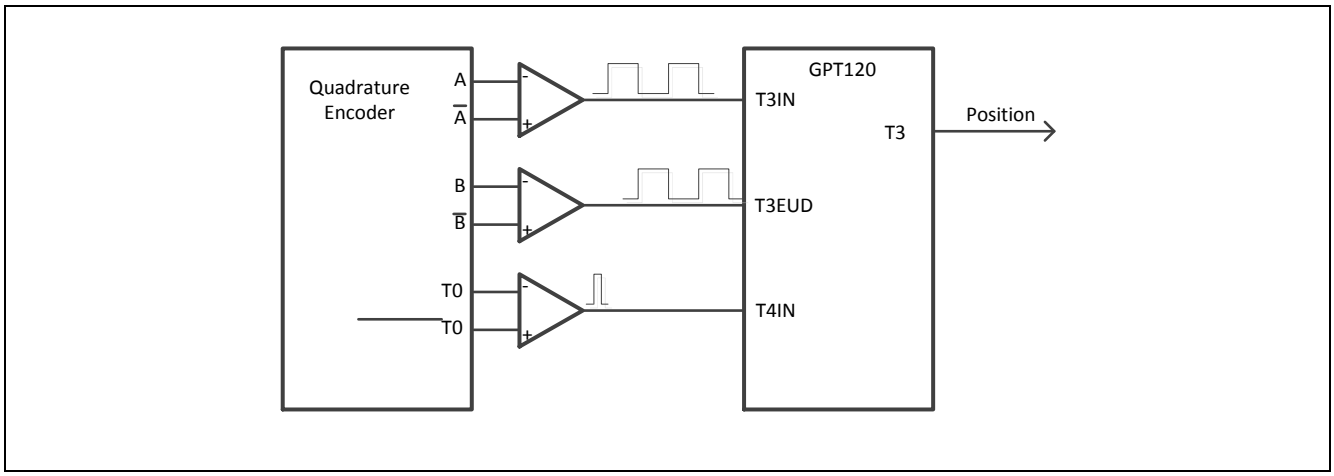


Figure 14GPT120 Encoder Connections

The absolute position is held in Timer3. The speed and direction can be deduced by a simple calculation between two complete input periods.

7.1.4 GPT120 Port Pins

Port Pin	Function	Port Pin	Function
00_7	T2INA	10_7	T3EUDB
00_8	T2EUDA	10_8	T4INB
00_9	T4EUDA	13_2	CAPINA
02_6	T3INA	20_0	T6EUDA
02_7	T3EUDA	20_3	T6INA
02_8	T4INA	21_6	T5EUDA
10_0	T6EUDB	21_6	T3OUT
10_1	T5EUDB	21_7	T5INA
10_2	T6INB	21_7	T6OUT
10_3	T5INB	33_5	T4EUDB
10_4	T3INB	33_6	T2EUDB
10_5	T6OUT	33_7	T2INB
10_6	T3OUT		

7.2 Generic Timer Module (GTM) Introduction

The Generic Timer Module (GTM) is the main source of pulse generation and measurement functions containing over 200 IO channels. It is designed primarily for automotive powertrain control and electric motor drives. Unlike conventional timer blocks, time-processing units, CAPCOM units etc. it can work in both the time and angle domains without restriction. This is particularly useful for mechanical control systems, motor commutation etc.

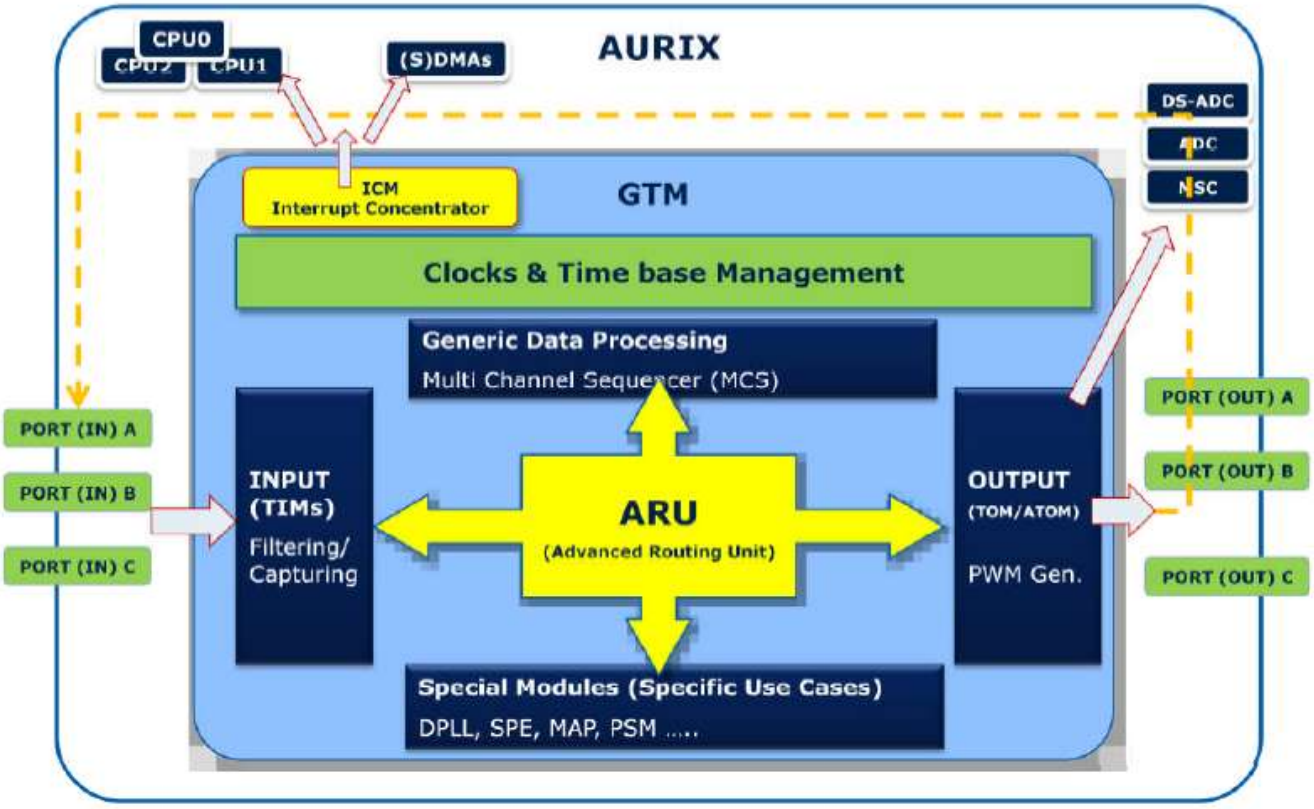


Figure 15Simplified GTM Overview

The GTM has around 3000 SFRs but fortunately you do not need to know all of these to realize useful functions! The configurable interconnections between timers allow a significant reduction in interrupt activity in the CPU as the routing of timing data between input and output timer blocks can be automated. For advanced applications it has its own internal Assembler-based calculation cores (MCS) that allow both basic data movement and calculation tasks to be off-loaded from the CPU.

It is enormously powerful and the culmination of 25 years of meeting the needs of high-end automotive control systems. However it can and indeed has been successfully applied to more general industrial applications, particularly in the field of motor control.

The GTM's major functional units are:

Table 26 Major GTM Blocks

Module	Basic	Intermediate	Advanced
Advanced Routing Unit (ARU)		X	
Broadcast Module (BRC)			X
First In First Out Module (FIFO)			X
AEI-to-FIFO Data Interface (AFD)			X
FIF0-to-ARU Interface (F2A)			X
Clock Management Unit (CMU)	X		
Timebase Unit (TBU)	X		
Timer Input Module (TIM)	X		X
Timer Output Module (TOM)	X		
ARU-connected Timer Output Module (ATOM)		X	
Dead-time Module (DTM)	X		
Multi-Channel Sequencer (MCS)			X
TIM0 Input Mapping Module (MAP)		X	
Digital PLL (DPLL)		X	
Sensor Pattern Evaluation Module (SPE)		X	
Interrupt Concentrator Module (ICM)	X		

However for most applications, only those listed in the "Basic" column are required. These include:

- Time-based motor control, SVM, FOC
- Angle-based motor control, SRM
- PWM
- DAC/ Waveform generation
- Power Inverters
- Software UARTS
- Encoder inputs
- Resolver exciters
- BLDC/PMSM
- CAPCOM replacement (C166 migration)
- Pulse detection/Duty ratio measurement
- Event-driven ADC triggering
- Timebase generation

In reality, the following internal GTM blocks are the most important:

TIM: Timer Input Module (16-bit, 32 channels)
TOM: Timer Output Module (16-bit, 48 channels)
ATOM: ARU-connected Timer Output Module (24-bit, 40 channels)
TBU: Timebase unit (24-bit, 3 channels)
DPLL: Digital Phase Locked Loop

Only a basic overview can be given here, but the capabilities of the GTM are so great that it is very likely that it can meet the requirements of almost any application, using a number of

different approaches. Therefore please contact your Infineon FAE or local Preferred Design House for advice and assistance in applying the GTM to your application.

7.3 Common Use Cases

This section sets out some common uses for the GTM. It is by no means exhaustive and the examples given use only a very small part of the GTM's capabilities. However they do serve to show the basic operating modes.

7.3.1 CAPCOM Replacement

7.3.1.1 Compare

The popular 16-bit Infineon microcontrollers featured the "CAPCOM" unit to make timing measurements and generate pulses on up to 16 channels. It could capture the times of input pulses and toggle outputs when a timer became equal to a compare register. Interrupts could be generated on both events. The CAPCOMs have two timers used as a timebase against which the captures and compares are made. The CAPCOM could be used for PWM generation, timebase generation, duty ratio measurement etc.. Some devices had 2 CAPCOMs, giving 32 channels in total.

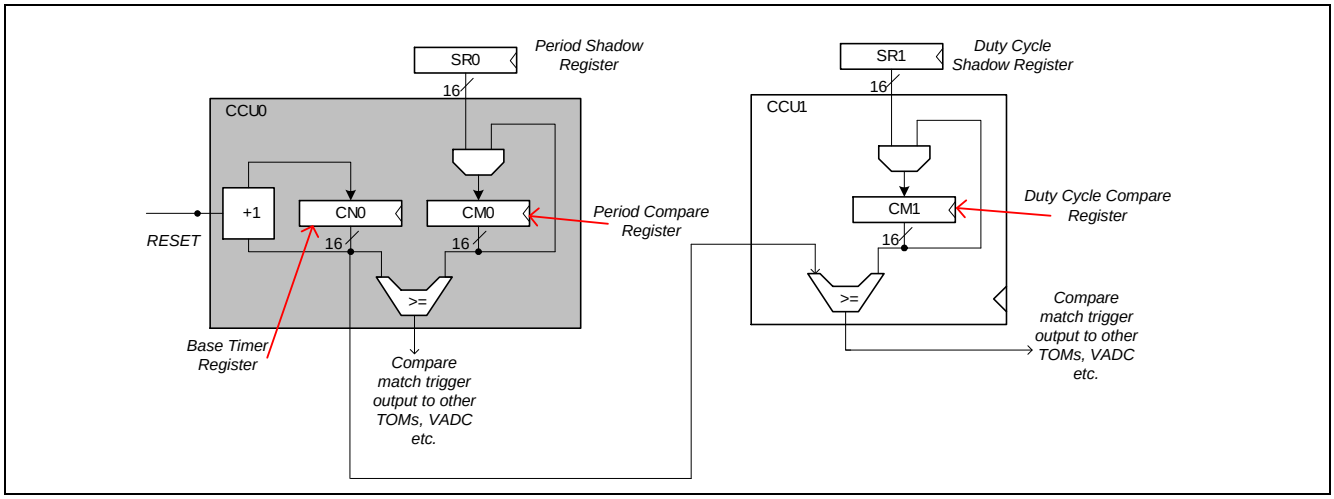


Figure 16Timer Output Module (TOM) Simplified Internal Layout

In the AURIX, the GTM can duplicate this functionality using the array of Timer Output Module (TOM) or ATOM channels. Each of these is a miniature single channel CAPCOM unit in its own right with a base timer (CN0), a period compare register (CM0) and a duty ratio compare register (CM1) that can be attached to a GPIO pin. The counter CN0 input clock is derived from the GTM's Clock Management Unit (CMU). This is running with a maximum of 100MHz so the timers run at 0.01us/count.

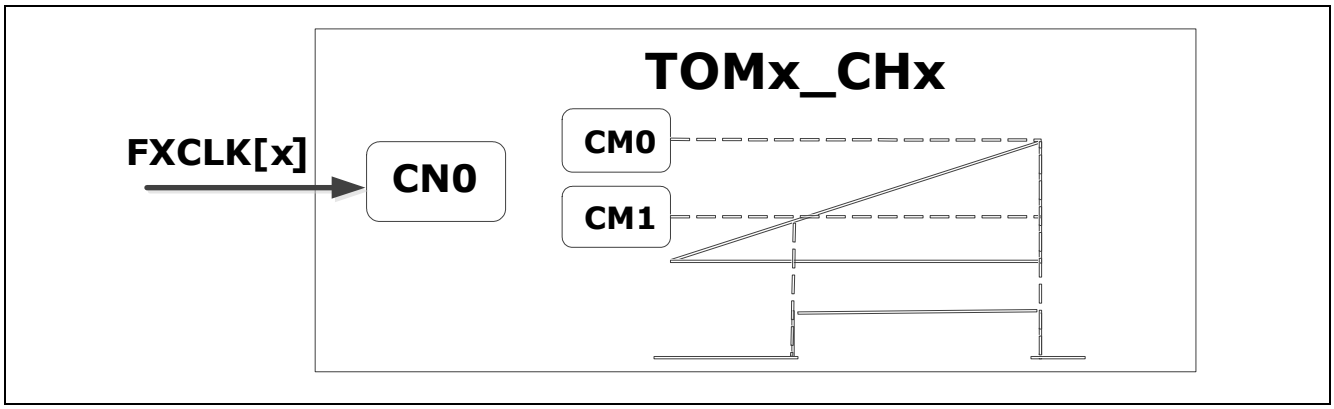


Figure 17Simple Edge-Aligned PWM

For PWM on say 3 channels, the first TOM is setup to define the period of the PWM. Its internal timer CN0 is configured to reset after a number of counts set by the special compare register CM0 that represents the PWM period. This then generates a trigger signal that the succeeding TOM groups can use to reset their own internal CN0 timers. Thus all TOMs are effectively using the same PWM period. The duty ratios required for each TOM can then be programmed individually into their CM1 duty ratio compare registers, eventually causing a pin toggle.

If required, interrupts may be triggered on the first TOM's duty ratio compare register CM1. Often this is set to half the CM0 period value, so that an interrupt can be triggered on the PWM centre point to allow reloading of the individual TOM's CM1 duty ratio registers. However, with the Aurix there is the new possibility of triggering the VADC to make a conversion. There are also shadow registers for the duty ratio compare to make sure new duty ratios are only applied during the next period and to overcome problems with very short or very long on times.

This is a very similar overall functionality to the CAPCOM unit, even though the underlying hardware is completely different. Moreover, as is often the case with the GTM, there are other approaches besides that used above.

Up to 47 channels of centre-aligned PWM can be realized with the TOM on the TC27x. (The first channel of TOM group 0 is used to generate the common period).

7.3.1.2 Capture

The capture functionality can be realized using the Timer Input Modules (TIM). Like the TOMs, each TIM channel is a miniature single-channel CAPCOM unit with its own base timer (CNT) in the Signal Measurement Unit (SMU) and a capture register (CNTS). Where a number of incoming signals need to be measured against the same timebase, then any one of the three global GTM timebase timers in the Time Base Unit (TBU) can be used as a basis.

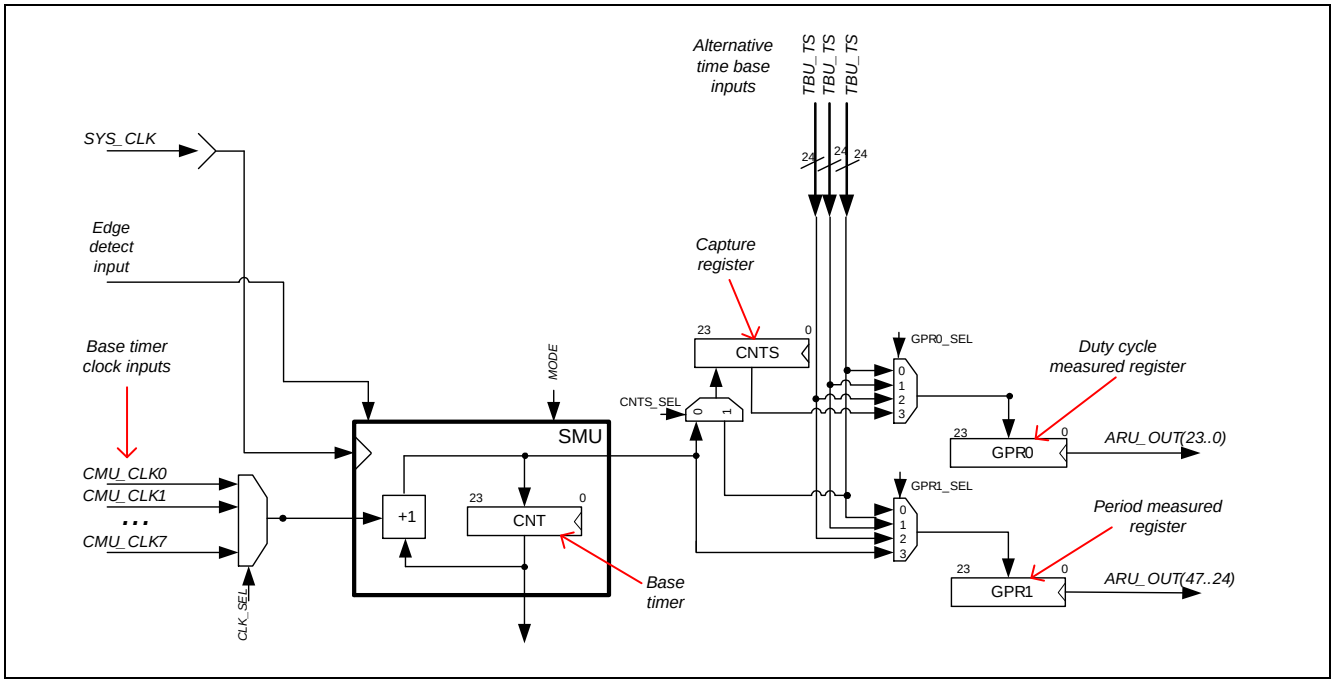


Figure 18 Simplified Timer Input Module (TIM) Channel Internal Layout

The TIMs' Timed Input Event Mode will allow the time at which a pin change took place to be captured into register GPR1 and an interrupt to be raised. In a CAPCOM this value would often be used as the basis for scheduling an output event in a compare register, but in the context of the GTM, there are other and better ways to achieve this.

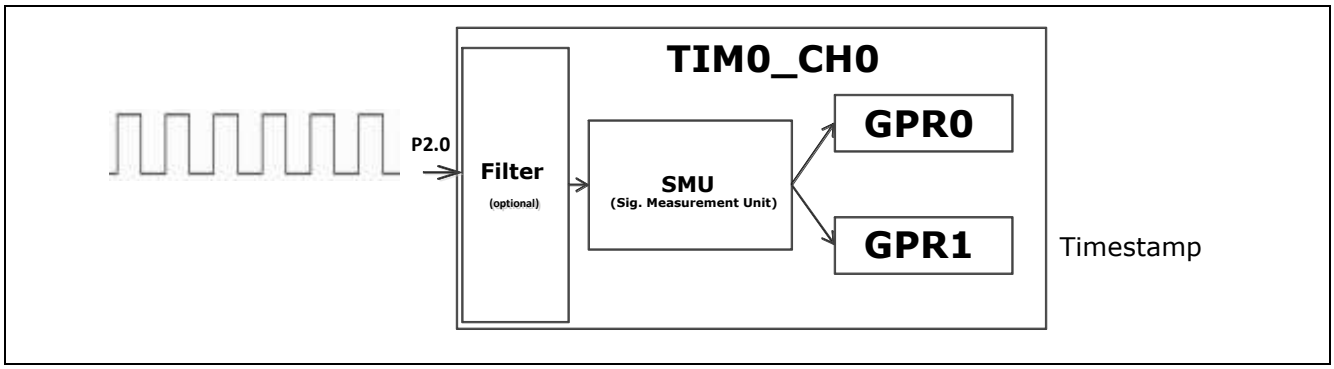


Figure 19TIM In Timed Input Event Mode

Where the objective is to make a duty ratio measurement or count pulses on a pin, then the TIMs' dedicated PWM Measurement and Input Prescaler modes make more sense.

7.3.1.3 TIM PWM Measurement Mode

The duty cycle (high time) and the period of the input waveform are automatically captured into the GPR0 and GPR1 registers respectively, with no user intervention.

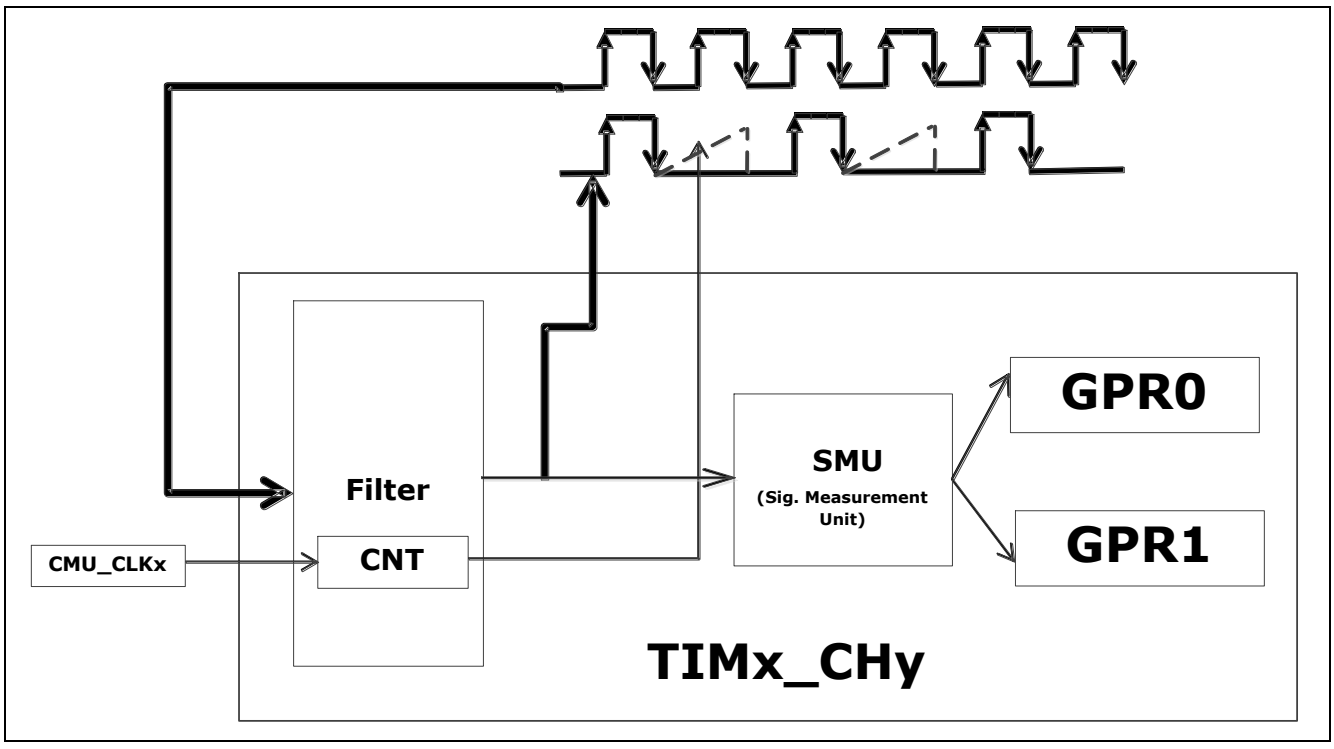


Figure 20PWM Measurement Mode

Technical Note: The DSL bit in the TIM control register defines the polarity of the PWM signal to be measured. When measurement of pulse high time and period is requested (PWM with a high level duty cycle, DSL=1), the channel starts measuring after the first rising edge is detected by the filter. If a PWM with a low level duty cycle should be measured (DSL = 0), the channel waits for a falling edge until measurement is started.

7.3.1.4 TIM Input Prescaler Mode

An interrupt is generated after a specific user defined number of edges (both rising/falling) set by the CNTS register.

7.3.1.5 TIM Pulse Integration Mode

The total high time (or low time) of a series of pulses is measured. Unlike CAPCOMs though, the TIMs have configurable filters that optionally remove glitches from input pulse trains automatically.

7.3.2 Digital To Analog Conversion (DAC)

The GTM can perform digital to analog conversion on every timer output channel using a simple PWM approach with an external lowpass RC filter.

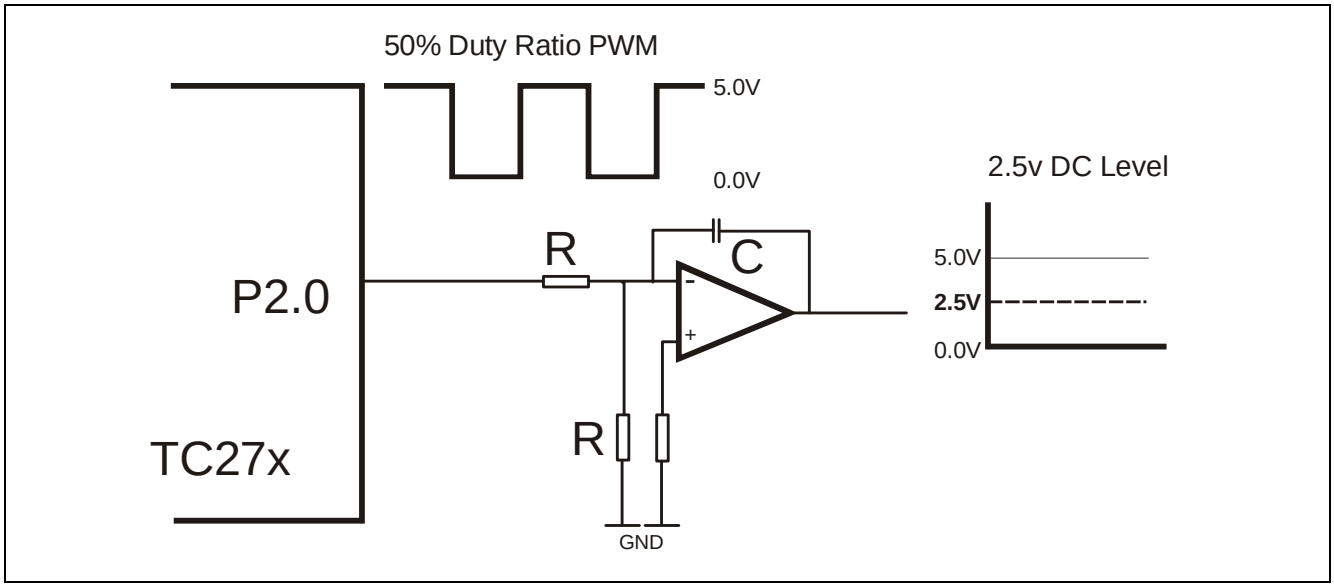


Figure 21 Digital to Analog Conversion with PWM

In most cases, edge-aligned PWM is adequate, as the load is non-inductive, being just a simple RC filter. The resolution of the DAC created is related to the underlying PWM frequency. For an 8-bit DAC, this is around 400kHz. For 16-bit resolution, the PWM frequency is 1.56kHz.

Table 27 Typical DAC Resolutions Possible Using GTM

DAC Resolution	PWM Frequency (approx.)
8 bits	400kHz
10 bits	100kHz
12 bits	25kHz
14 bits	6.25kHz
16 bits	1.56kHz Please add also for 24-bit

Updating the DAC is best done using the CM1 shadow compare registers, as the new value is only transferred at the end of the period. Thus new values can be written when convenient, rather than having to wait for the end of a PWM cycle.

To achieve a complete independent cyclical output, a combination of PSM and ATOM can be used. Then no interaction from a CPU or DMA is needed.

7.3.3 Motor Control

There are numerous ways to implement motor control with the GTM and all common motor configurations can be supported. There are a number of dedicated modules in the GTM to assist such as the Sensor Pattern Evaluation unit (SPE) for interfacing to Hall sensors.

7.3.3.1 BLDC Block Commutation/Six-Step

This is the most basic motor control strategy and is easy to implement in the GTM, mainly due to the SPE unit. The SPE is able to recognize a user-defined Hall sensor input sequence e.g.

```
100b, 110b, 010b, 011b, 001b, 101b, 100b
```

When applied to three nominated TIMs. From this it deduces the direction of rotation and triggers six TOMs (three low side and three high side) to generate PWM in such a manner as to cause motor rotation, all without CPU intervention. The SPE is also able to detect broken wires, illegal sequences and states (000, 111) and trigger an error interrupt as a result. Two complete motor subsystems can be controlled this way.

Current control is through a simple VADC measurement of voltage across a shunt in the DC link with the conversion taking place at a fixed rotor angle.

7.3.3.2 PMSM

Space-Vector Modulation (SVM) is commonly used where a near-sinusoidal waveform must be generated on three channels, with associated complementary outputs. The PWM generated by the TOMs is now centre-aligned with a near-sinusoidal envelope.

For the phase voltage, measurement is often made in the ground path across two shunts, where it is essential that the VADC triggering is locked to timing of the PWM. This is achieved by using the trigger output from TOMs, generating the PWM as a trigger source for

two VADC modules where simultaneous analog conversions are made (the current for the third phase being calculated).

7.3.4 Linking The GTM To Other AURIX Peripherals

As the source of timing events, the GTM is a good way to trigger other peripherals on the AURIX. The most important of these is the VADC. In applications like motor control, the GTM will be generating the PWM and it is common to need to synchronize ADC readings to specific parts of the PWM waveform, to make simultaneous voltage and current measurements for example. In other situations it may be required to read a sequence of ADC channels at the centre point of the PWM.

As well as generating pin changes, TOMs can generate internal trigger signals that can be routed to the VADC, CAN, CCU6, PSI, DSADC, MSC and some others. The MSC (Micro Second Channel) connection is particularly useful, as it allows events generated by a TOM to be connected via a very high speed serial link to an MSC-compatible output device (e.g. TLE8718) that generates a physical signal. This allows the number of physical pins that the GTM can drive to be greatly extended beyond what the CPU package supports.

It should be noted that not all TOMs can be used as triggers for other peripherals and this could have an indirect influence on the pinout of the CPU and hence the PCB layout. For example, TOMx channel 0 cannot generate ADC triggers, so if this is being used as the period generator for centre-aligned PWM, you will need to use another TOM to generate the VADC trigger. Refer to the GTM implementation chapter in the AURIX UM for the trigger connections.

7.3.5 Digital Phase Locked-Loop (DPLL)

Traditionally, most schemes for synchronizing to shafts, generating PWM and controlling rotating machines has been based on detecting the time of a marker on the shaft and then scheduling an output event to a power device (for example) that will then influence either the position of the shaft or the speed of rotation. The commonest example of this is in engine controls and motors drives. What is really happening is that the position of the shaft in the future is being predicted, based on its speed of rotation over the previous 6 degrees, 10 degrees, 90 degrees or whatever the angular distance between the markers happens to be. The angular velocity is converted into a number of counts of a CPU timer, running at say 500ns/bit. The angle of the output event is then used to multiply this count to generate a new time, which is then added to the timestamp recorded at the last input marker and loaded into a compare register.

What is happening here is that we are moving from the angle domain into the time domain and then back again. This requires real time multiplication and division operations, which rely on the CPU. It works well in many situations, but has some limitations. Where the output events have to be at specific angular positions, compensating for acceleration can become difficult and cause a high interrupt loading on the CPU, as calculations are repeated rapidly.

The DPLL largely removes the need for CPU intervention in the scheduling of output events. It takes the raw input marker pulse stream and multiplies their frequency in hardware. Thus 10 degree markers can be multiplied by 10 to get 1 degree markers. Now the compare registers creating the output events can work in terms of degrees of rotation, rather than being relative to a timer and the required angle is simply loaded into the register. When the angle counter emitted by the DPLL equals the angle in the compare register, the event occurs.

In the automotive field, such an approach is widely used and this is the main reason the GTM contains the DPLL. However, it has interesting applications in motor control, particularly for switched-reluctance drives where the control of the output drivers is almost entirely angle-based. By removing all the angle to time to angle calculations, system design is considerably simplified and CPU load reduced.

7.3.6 PCB Layout Issues

It is recommended to not make a final AURIX pin allocation for the PCB until you have finalized the GTM configuration, as the TIM and TOM to port pin mapping has certain rules, special features and the distribution of timer modules within a port may not be as expected – see the red items in the table below.

Table 28 Port 00 TIM, TOM and ATOM Relationship (Extract from TC275 UM Table 25-67)

Port	Input	Output	Input Timer Mapped		Output Timer Mapped			
			A	B	A	B	C	D
P00.0	TIN9	TOUT9	TIM2_0	TIM3_0	TOM0_8	TOM1_0	ATOM 0_0	ATOM 1_0
P00.1	TIN10	TOUT10	TIM2_1	TIM3_1	TOM0_9	TOM1_1	ATOM 0_1	ATOM 1_1
P00.2	TIN11	TOUT11	TIM2_1	TIM3_1	TOM0_9	TOM1_1	ATOM 0_1	ATOM 1_1
P00.3	TIN12	TOUT12	TIM2_2	TIM3_2	TOM0_10	TOM1_2	ATOM 0_2	ATOM 1_2
P00.4	TIN13	TOUT13	TIM2_3	TIM3_3	TOM0_11	TOM1_3	ATOM 0_3	ATOM 1_3
P00.5	TIN14	TOUT14	TIM2_4	TIM3_4	TOM0_12	TOM1_4	ATOM 0_4	ATOM 1_4
P00.6	TIN15	TOUT15	TIM2_5	TIM3_5	TOM0_13	TOM1_5	ATOM 0_5	ATOM 1_5
P00.7	TIN16	TOUT16	TIM2_6	TIM3_6	TOM0_14	TOM1_6	ATOM 0_6	ATOM 1_6
P00.8	TIN17	TOUT17	TIM2_7	TIM3_7	TOM0_15	TOM1_7	ATOM 0_7	ATOM 1_7
P00.9	TIN18	TOUT18	TIM0_0	TIM1_0	TOM0_0	TOM1_0	ATOM 2_0	ATOM 3_0
P00.10	TIN19	TOUT19	TIM0_1	TIM1_1	TOM0_1	TOM1_1	ATOM 2_1	ATOM 3_1
P00.11	TIN20	TOUT20	TIM0_2	TIM1_2	TOM0_2	TOM1_2	ATOM 2_2	ATOM 3_2
P00.12	TIN21	TOUT21	TIM0_3	TIM1_3	TOM0_3	TOM1_3	ATOM 2_3	ATOM 3_3

Some examples of where GTM to pin mapping has constraints are:

- TIM0 has special functions related to the DPLL and crankshaft and camshaft position detection which TIM1/2/3 do not have.
- Also, TOMx channel 15 has a specific PCM feature.
- With centre-aligned PWM generation, it is most common that TOMx channel 0 will be used as the centre and period event generator. If not, then it will be the lowest TOM in the group being used to generate PWM.

7.4 Getting The GTM Running

7.4.1 Timebase Generation

A basic GTM function is to implement a timebase that generates a periodic interrupt. How to do this step-by-step, using the Infineon iLLDs to create a 1ms interrupt, is given below.

7.4.1.1 Basic Steps

Connect the system clock to the GTM to enable it

```
Ifx_GTM *gtm = &MODULE_GTM;
IfxGtm_enable(gtm);
```

Get the current GTM clock frequency (System Peripheral Bus).

```
g_GtmTomTimer.info.gtmFreq = IfxGtm_Cmu_getModuleFrequency(gtm);
```

Set up the GTM's Clock Management Unit (CMU).

```
IfxGtm_Cmu_setGclkFrequency(gtm, g_GtmTomTimer.info.gtmFreq);
g_GtmTomTimer.info.gtmGclkFreq = IfxGtm_Cmu_getGclkFrequency(gtm);
```

Set the required frequency of the interrupt to be generated. Here it is 1000Hz.

```
timerConfig.base.frequency = 1000;
```

Set the priority of the interrupt to be generated.

```
timerConfig.base.isrPriority = ISR_PRIORITY(INTERRUPT_TIMER_1MS);
```

Set the source of the interrupt.

```
timerConfig.base.isrProvider = ISR_PROVIDER(INTERRUPT_TIMER_1MS);
```

Make a check to see whether the requested interrupt frequency is feasible.

```
timerConfig.base.minResolution = (1.0/timerConfig.base.frequency)/1000;
```

Disable the TOM output trigger as there are no other TOMs sending triggers to this TOM.

```
timerConfig.base.trigger.enabled = FALSE;
```

Set which TOM to use (0, 1, 2, 3) and which channel. Here it is TOM1, channel 1.

```
    timerConfig.tom = IfxGtm_Tom_1;
```

```
        timerConfig.timerChannel = IfxGtm_Tom_Ch_0;
```

Set which clock source the TOM should use.

```
        timerConfig.clock  = IfxGtm_Tom_Ch_ClkSrc_cmuFxclk1;
```

Write the configuration into the GTM:

```
        IfxGtm_Tom_Timer_init(
&g_GtmTomTimer.drivers.timerOneMs, &timerConfig);
```

Start the TOM.

```
        IfxGtm_Tom_Timer_run(&g_GtmTomTimer.drivers.timerOneMs);
```

The interrupt function will now be called every 1ms.

```
void ISR_Timer_1ms(void)
{
   IfxCpu_enableInterrupts();
   IfxGtm_Tom_Timer_acknowledgeTimerIrq(&g_GtmTomTimer.drivers.timerOneMs);
   g_GtmTomTimer.isrCounter.slotOneMs++;
}
```

7.4.2 Simple Centre-Aligned PWM

This example creates a centre-aligned PWM on two pins with low-side and complementary high side waveforms, along with the necessary deadtime offset to prevent damage to power drivers. This uses TOM1 channel 0 as the timebase, TOM1 channel 1 on Port 00.1 as the output and TOM1 channel 2 on Port 00.3 as the complementary output. It is the basis of many motor control strategies using the GTM.

7.4.2.1 Basic Steps

The first part of the configuration is identical to the timebase example above as it also uses TOM1 channel 0. However, it then goes on to set up two more TOMs to generate the PWM outputs.

Connect the system clock to the GTM to enable it.

```
Ifx_GTM *gtm = &MODULE_GTM;
IfxGtm_enable(gtm);
```

Get the current GTM clock frequency (System Peripheral Bus).

```
g_GtmTomTimer.info.gtmFreq = IfxGtm_Cmu_getModuleFrequency(gtm);
```

Set up the GTM's Clock Management Unit (CMU).

```
IfxGtm_Cmu_setGclkFrequency(gtm, g_GtmTomTimer.info.gtmFreq);
g_GtmTomTimer.info.gtmGclkFreq = IfxGtm_Cmu_getGclkFrequency(gtm);
```

Set the required frequency of the interrupt to be generated. Unlike before, where it was 1000Hz, we are now using 10000Hz to give a PWM period of 100us.

```
timerConfig.base.frequency = 10000;
```

Set the priority of the interrupt to be generated.

```
timerConfig.base.isrPriority = ISR_PRIORITY(INTERRUPT_TIMER_1MS);
```

Set the source of the interrupt

```
timerConfig.base.isrProvider = ISR_PROVIDER(INTERRUPT_TIMER_1MS);
```

Make a check to see whether the requested interrupt frequency is feasible.

```
timerConfig.base.minResolution = (1.0/timerConfig.base.frequency)/1000;
```

Disable the TOM output trigger as there are no other TOMs sending triggers to this TOM.

```
timerConfig.base.trigger.enabled = FALSE;
```

Set which TOM to use (0, 1, 2) and which channel. Here it is TOM1, channel 1.

```
timerConfig.tom = IfxGtm_Tom_1;
timerConfig.timerChannel = IfxGtm_Tom_Ch_0;
```

Set which clock source the TOM should use.

```
timerConfig.clock  = IfxGtm_Tom_Ch_ClkSrc_cmuFxclk1;
```

Write the configuration into the GTM:

```
IfxGtm_Tom_Timer_init(
&g_GtmTomTimer.drivers.timerOneMs, &timerConfig);
```

Set up Two Complementary PWM Outputs. Nominate the two port pins to be used for the PWM outputs.

```
IfxGtm_Tom_ToutMapP ccx[1] = {&IfxGtm_TOM1_1_TOUT10_P00_1_OUT};
IfxGtm_Tom_ToutMapP coutx[1] = {&IfxGtm_TOM1_2_TOUT12_P00_3_OUT};
…….
pwmHlConfig.ccx = ccx;
pwmHlConfig.coutx = coutx;
```

Indicate the base timer to be used. Here it is the TOM1 channel 0 from the previous steps above.

```
pwmHlConfig.timer = &g_GtmTomPwmHl.drivers.timer;
pwmHlConfig.tom = timerConfig.tom;
```

Set the number of channels of PWM to generate, plus the deadtime offset between them and the minimum permitted pulsewidth. The units for these are expressed in seconds.

```
pwmHlConfig.base.deadtime = 2e-6;
pwmHlConfig.base.minPulse = 1e-6;
pwmHlConfig.base.channelCount = 1;
```

Set up the output driver to be used by the PWM pins.

```
pwmHlConfig.base.outputMode = IfxPort_OutputMode_pushPull;
pwmHlConfig.base.outputDriver = IfxPort_PadDriver_cmosAutomotiveSpeed1;
```

Set the active states of the two pins.

```
pwmHlConfig.base.ccxActiveState   = Ifx_ActiveState_high;
pwmHlConfig.base.coutxActiveState = Ifx_ActiveState_high;
```

Write the configuration into the GTM.

```
IfxGtm_Tom_PwmHl_init(&g_GtmTomPwmHl.drivers.pwm, &pwmHlConfig);
```

Start the TOM1_0.

```
IfxGtm_Tom_Timer_run(&g_GtmTomPwmHl.drivers.timer);
```

The interrupt function will now be called every 100us. It will cycle the PWM outputs between 0, 25, 50, 75 and 100% duty ratios continuously.

Figure 22Centre-Aligned Complementary PWM with 25% and 50% Duty Ratios

7.5 GTM Port Pins

7.5.1 Port Pin To TOM Mapping

Each TOM can be connected to a range of GPIO pins through a multiplexer. There are 3 TOM units in the TC275, numbered TOM0 to TOM2.

Useful macros and structures for accessing the GTM in 'C' may be found in the programming examples:

```
.\0_Src\4_McHal\Tricore\_PinMap\IfxGtm_PinMap.c
.\0_Src\4_McHal\Tricore\_PinMap\IfxGtm_PinMap.h
```

Table 29 Port Pin To TOM Mapping

Port Pin	TOM	Port Pin	TOM	Port Pin	TOM	Port Pin	TOM
P00_0	TOM0_8	P11_0	TOM2_0	P15_3	TOM2_14	P22_5	TOM2_8
P00_0	TOM1_0	P11_1	TOM2_1	P15_4	TOM1_7	P22_6	TOM2_9
P00_1	TOM0_9	P11_10	TOM0_13	P15_4	TOM2_15	P22_7	TOM2_10
P00_1	TOM1_1	P11_10	TOM2_5	P15_5	TOM0_0	P22_8	TOM2_11
P00_10	TOM0_1	P11_11	TOM0_14	P15_5	TOM1_0	P22_9	TOM2_12
P00_10	TOM1_1	P11_11	TOM2_6	P15_6	TOM0_0	P23_0	TOM0_10
P00_11	TOM0_2	P11_12	TOM0_15	P15_6	TOM1_0	P23_0	TOM1_5
P00_11	TOM1_2	P11_12	TOM2_7	P15_7	TOM0_1	P23_1	TOM0_15
P00_12	TOM0_3	P11_13	TOM2_6	P15_7	TOM1_1	P23_1	TOM0_6
P00_12	TOM1_3	P11_14	TOM2_7	P15_8	TOM0_2	P23_2	TOM0_11
P00_2	TOM0_9	P11_15	TOM2_8	P15_8	TOM1_2	P23_2	TOM1_6
P00_2	TOM1_1	P11_2	TOM0_8	P20_0	TOM0_6	P23_3	TOM0_12
P00_3	TOM0_10	P11_2	TOM2_1	P20_0	TOM2_6	P23_3	TOM1_7
P00_3	TOM1_2	P11_3	TOM0_10	P20_1	TOM1_11	P23_4	TOM0_7
P00_4	TOM0_11	P11_3	TOM2_2	P20_1	TOM2_3	P23_4	TOM1_7
P00_4	TOM1_3	P11_4	TOM2_2	P20_10	TOM1_14	P23_5	TOM0_10
P00_5	TOM0_12	P11_5	TOM2_3	P20_10	TOM2_14	P23_5	TOM2_2
P00_5	TOM1_4	P11_6	TOM0_11	P20_11	TOM1_15	P32_0	TOM1_14
P00_6	TOM0_13	P11_6	TOM2_3	P20_11	TOM2_15	P32_0	TOM2_14
P00_6	TOM1_5	P11_7	TOM2_4	P20_12	TOM1_0	P32_2	TOM0_3
P00_7	TOM0_14	P11_8	TOM2_5	P20_12	TOM2_8	P32_2	TOM1_3
P00_7	TOM1_6	P11_9	TOM0_12	P20_13	TOM1_1	P32_3	TOM0_4
P00_8	TOM0_15	P11_9	TOM2_4	P20_13	TOM2_9	P32_3	TOM1_4
P00_8	TOM1_7	P12_0	TOM1_8	P20_14	TOM1_2	P32_4	TOM0_5
P00_9	TOM0_0	P12_1	TOM1_9	P20_14	TOM2_10	P32_4	TOM1_5
P00_9	TOM1_0	P13_0	TOM0_5	P20_3	TOM1_12	P32_6	TOM1_8
P02_0	TOM0_8	P13_0	TOM2_5	P20_3	TOM2_4	P32_7	TOM1_9
P02_0	TOM1_8	P13_1	TOM0_6	P20_6	TOM1_10	P33_0	TOM0_4
P02_1	TOM0_9	P13_1	TOM2_6	P20_6	TOM2_10	P33_0	TOM1_4
P02_1	TOM1_9	P13_2	TOM0_7	P20_7	TOM1_11	P33_1	TOM0_5
P02_2	TOM0_10	P13_2	TOM2_7	P20_7	TOM2_11	P33_1	TOM1_5
P02_2	TOM1_10	P13_3	TOM0_8	P20_8	TOM1_7	P33_10	TOM0_0
P02_3	TOM0_11	P13_3	TOM2_0	P20_8	TOM2_7	P33_10	TOM1_0
P02_3	TOM1_11	P14_0	TOM0_3	P20_9	TOM1_13	P33_11	TOM0_2
P02_4	TOM0_12	P14_0	TOM1_3	P20_9	TOM2_13	P33_11	TOM1_2
P02_4	TOM1_12	P14_1	TOM0_4	P21_0	TOM0_8	P33_12	TOM1_12
P02_5	TOM0_13	P14_1	TOM1_4	P21_0	TOM2_8	P33_12	TOM2_12
P02_5	TOM1_13	P14_10	TOM0_4	P21_1	TOM0_9	P33_13	TOM1_13
P02_6	TOM0_14	P14_10	TOM2_4	P21_1	TOM2_9	P33_13	TOM2_13
P02_6	TOM1_14	P14_2	TOM0_5	P21_2	TOM0_0	P33_14	TOM1_10

Port Pin	TOM	Port Pin	TOM	Port Pin	TOM	Port Pin	TOM
P02_7	TOM0_15	P14_2	TOM1_5	P21_2	TOM2_0	P33_15	TOM1_11
P02_7	TOM1_15	P14_3	TOM0_6	P21_3	TOM0_1	P33_2	TOM0_6
P02_8	TOM0_8	P14_3	TOM1_6	P21_3	TOM2_1	P33_2	TOM1_6
P02_8	TOM1_0	P14_4	TOM0_7	P21_4	TOM0_2	P33_3	TOM0_7
P10_0	TOM0_4	P14_4	TOM1_7	P21_4	TOM2_2	P33_3	TOM1_7
P10_0	TOM2_12	P14_5	TOM0_0	P21_5	TOM0_3	P33_4	TOM0_0
P10_1	TOM0_1	P14_5	TOM1_0	P21_5	TOM2_3	P33_4	TOM1_0
P10_1	TOM2_9	P14_6	TOM0_1	P21_6	TOM0_4	P33_5	TOM0_1
P10_2	TOM0_2	P14_6	TOM1_1	P21_6	TOM2_4	P33_5	TOM1_1
P10_2	TOM2_10	P14_7	TOM0_0	P21_7	TOM0_5	P33_6	TOM0_2
P10_3	TOM0_3	P14_7	TOM2_0	P21_7	TOM2_5	P33_6	TOM1_2
P10_3	TOM2_11	P14_8	TOM0_2	P22_0	TOM0_9	P33_7	TOM0_3
P10_4	TOM0_6	P14_8	TOM2_2	P22_0	TOM2_1	P33_7	TOM1_3
P10_4	TOM2_6	P14_9	TOM0_3	P22_1	TOM0_8	P33_8	TOM0_4
P10_5	TOM0_2	P14_9	TOM2_3	P22_1	TOM2_0	P33_8	TOM1_4
P10_5	TOM2_10	P15_0	TOM1_3	P22_10	TOM2_13	P33_9	TOM0_1
P10_6	TOM0_3	P15_0	TOM2_11	P22_11	TOM2_14	P33_9	TOM1_1
P10_6	TOM2_11	P15_1	TOM1_4	P22_2	TOM0_11	P34_1	TOM1_13
P10_7	TOM0_0	P15_1	TOM2_12	P22_2	TOM2_3	P34_2	TOM1_14
P10_7	TOM2_8	P15_2	TOM1_5	P22_3	TOM0_12	P34_3	TOM1_15
P10_8	TOM0_5	P15_2	TOM2_13	P22_3	TOM2_4	P34_4	TOM2_15
P10_8	TOM2_13	P15_3	TOM1_6	P22_4	TOM2_7	P34_5	TOM1_15

7.5.2 Port Pin To TIM Mapping

Each TIM can be connected to a range of GPIO pins through a multiplexer. There are 4 TIM units in the TC275, numbered TIM0 to TIM3.

Port Pin	TIM	Port Pin	TIM	Port Pin	TIM	Port Pin	TIM
P00_0	TIM2_0	P10_6	TIM1_3	P15_2	TIM3_5	P22_6	TIM3_2
P00_0	TIM3_0	P10_7	TIM0_0	P15_3	TIM2_6	P22_7	TIM3_3
P00_1	TIM2_1	P10_7	TIM1_0	P15_3	TIM3_6	P22_8	TIM3_4
P00_1	TIM3_1	P10_8	TIM0_5	P15_4	TIM2_7	P22_9	TIM3_5
P00_10	TIM0_1	P10_8	TIM1_5	P15_4	TIM3_7	P23_0	TIM0_5
P00_10	TIM1_1	P11_0	TIM2_0	P15_5	TIM2_0	P23_0	TIM1_5
P00_11	TIM0_2	P11_1	TIM2_1	P15_5	TIM3_0	P23_1	TIM0_6
P00_11	TIM1_2	P11_10	TIM2_5	P15_6	TIM0_0	P23_1	TIM1_6
P00_12	TIM0_3	P11_10	TIM3_5	P15_6	TIM1_0	P23_2	TIM0_6
P00_12	TIM1_3	P11_11	TIM2_6	P15_7	TIM0_1	P23_2	TIM1_6
P00_2	TIM2_1	P11_11	TIM3_6	P15_7	TIM1_1	P23_3	TIM0_7
P00_2	TIM3_1	P11_12	TIM2_7	P15_8	TIM0_2	P23_3	TIM1_7
P00_3	TIM2_2	P11_12	TIM3_7	P15_8	TIM1_2	P23_4	TIM0_7
P00_3	TIM3_2	P11_13	TIM2_6	P20_0	TIM0_6	P23_4	TIM1_7
P00_4	TIM2_3	P11_14	TIM2_7	P20_0	TIM1_6	P23_5	TIM0_2
P00_4	TIM3_3	P11_15	TIM0_7	P20_1	TIM2_3	P23_5	TIM1_2
P00_5	TIM2_4	P11_2	TIM2_1	P20_1	TIM3_3	P23_6	TIM1_2
P00_5	TIM3_4	P11_2	TIM3_1	P20_10	TIM2_6	P23_7	TIM1_3
P00_6	TIM2_5	P11_3	TIM2_2	P20_10	TIM3_6	P32_0	TIM2_2
P00_6	TIM3_5	P11_3	TIM3_2	P20_11	TIM2_7	P32_0	TIM3_2
P00_7	TIM2_6	P11_4	TIM2_2	P20_11	TIM3_7	P32_2	TIM0_3
P00_7	TIM3_6	P11_5	TIM2_3	P20_12	TIM2_0	P32_2	TIM1_3
P00_8	TIM2_7	P11_6	TIM2_3	P20_12	TIM3_0	P32_3	TIM0_4
P00_8	TIM3_7	P11_6	TIM3_3	P20_13	TIM2_1	P32_3	TIM1_4
P00_9	TIM0_0	P11_7	TIM2_4	P20_13	TIM3_1	P32_4	TIM0_5
P00_9	TIM1_0	P11_8	TIM2_5	P20_14	TIM2_2	P32_4	TIM1_5

Port Pin	TIM	Port Pin	TIM	Port Pin	TIM	Port Pin	TIM
P01_3	TIM0_5	P11_9	TIM2_4	P20_14	TIM3_2	P32_5	TIM3_5
P01_4	TIM0_6	P11_9	TIM3_4	P20_3	TIM2_4	P32_6	TIM3_6
P01_5	TIM2_3	P12_0	TIM3_0	P20_3	TIM3_4	P32_7	TIM3_7
P01_6	TIM2_5	P12_1	TIM3_1	P20_6	TIM2_6	P33_0	TIM0_4
P01_7	TIM2_7	P13_0	TIM2_5	P20_6	TIM3_6	P33_0	TIM1_4
P02_0	TIM0_0	P13_0	TIM3_5	P20_7	TIM2_7	P33_1	TIM0_5
P02_0	TIM1_0	P13_1	TIM2_6	P20_7	TIM3_7	P33_1	TIM1_5
P02_1	TIM0_1	P13_1	TIM3_6	P20_8	TIM0_7	P33_10	TIM0_0
P02_1	TIM1_1	P13_2	TIM2_7	P20_8	TIM1_7	P33_10	TIM1_0
P02_10	TIM0_3	P13_2	TIM3_7	P20_9	TIM2_5	P33_11	TIM0_2
P02_11	TIM0_7	P13_3	TIM2_0	P20_9	TIM3_5	P33_11	TIM1_2
P02_2	TIM0_2	P13_3	TIM3_0	P21_0	TIM2_4	P33_12	TIM2_0
P02_2	TIM1_2	P14_0	TIM0_3	P21_0	TIM3_4	P33_12	TIM3_0
P02_3	TIM0_3	P14_0	TIM1_3	P21_1	TIM2_5	P33_13	TIM2_1
P02_3	TIM1_3	P14_1	TIM0_4	P21_1	TIM3_5	P33_13	TIM3_1
P02_4	TIM0_4	P14_1	TIM1_4	P21_2	TIM0_0	P33_14	TIM2_0
P02_4	TIM1_4	P14_10	TIM2_4	P21_2	TIM1_0	P33_15	TIM2_1
P02_5	TIM0_5	P14_10	TIM3_4	P21_3	TIM0_1	P33_2	TIM0_6
P02_5	TIM1_5	P14_2	TIM0_5	P21_3	TIM1_1	P33_2	TIM1_6
P02_6	TIM0_6	P14_2	TIM1_5	P21_4	TIM0_2	P33_3	TIM0_7
P02_6	TIM1_6	P14_3	TIM0_6	P21_4	TIM1_2	P33_3	TIM1_7
P02_7	TIM0_7	P14_3	TIM1_6	P21_5	TIM0_3	P33_4	TIM0_0
P02_7	TIM1_7	P14_4	TIM0_7	P21_5	TIM1_3	P33_4	TIM1_0
P02_8	TIM2_0	P14_4	TIM1_7	P21_6	TIM0_4	P33_5	TIM0_1
P02_8	TIM3_0	P14_5	TIM0_0	P21_6	TIM1_4	P33_5	TIM1_1
P02_9	TIM0_2	P14_5	TIM1_0	P21_7	TIM0_5	P33_6	TIM0_2
P10_0	TIM0_4	P14_6	TIM0_1	P21_7	TIM1_5	P33_6	TIM1_2
P10_0	TIM1_4	P14_6	TIM1_1	P22_0	TIM0_1	P33_7	TIM0_3
P10_1	TIM0_1	P14_7	TIM0_0	P22_0	TIM1_1	P33_7	TIM1_3
P10_1	TIM1_1	P14_7	TIM1_0	P22_1	TIM0_0	P33_8	TIM0_4
P10_2	TIM0_2	P14_8	TIM2_2	P22_1	TIM1_0	P33_8	TIM1_4
P10_2	TIM1_2	P14_8	TIM3_2	P22_10	TIM3_6	P33_9	TIM0_1
P10_3	TIM0_3	P14_9	TIM2_3	P22_11	TIM3_7	P33_9	TIM1_1
P10_3	TIM1_3	P14_9	TIM3_3	P22_2	TIM0_3	P34_1	TIM2_3
P10_4	TIM0_6	P15_0	TIM2_3	P22_2	TIM1_3	P34_2	TIM2_4
P10_4	TIM1_6	P15_0	TIM3_3	P22_3	TIM0_4	P34_3	TIM2_5
P10_5	TIM0_2	P15_1	TIM2_4	P22_3	TIM1_4	P34_4	TIM2_6
P10_5	TIM1_2	P15_1	TIM3_4	P22_4	TIM3_0	P34_5	TIM2_7
P10_6	TIM0_3	P15_2	TIM2_5	P22_5	TIM3_1		

8 VADC

8.1 VADC Introduction

The VADC allows for high speed conversions from analogue voltage levels to a digital representation. This therefore allows you to measure voltages from code. Converting analogue to digital has many uses, such as measuring potentiometers, analogue accelerometers and frequencies. The VADC consists of separate Successive Approximation Register converters (SAR-ADC), each with their own sample & hold units, arranged in “Groups”.

It can be triggered by other Aurix peripherals such as the ERU, CCU6, GPT12, GTM and others. For example, in the latter case, the ADC can be triggered to take up to 4 simultaneous readings at the centre point of a PWM pulse train being use to drive a motor so that multiple currents and voltages can be captured at a single point.

8.1.1 VADC Resolution

It supports conversion resolutions of 8, 10 and 12-bit and when used in 12-bit calibrated mode, the Total Unadjusted Error (TUE) is +/-4LSB with an Effective Number Of Bits (ENOB) of 10. Uncalibrated, it is +/-6 LSB with an ENOB of 9.5. The conversion times are typically 980ns for 12-bit calibrated mode, reducing with the number of bits. The conversion time is guaranteed to be deterministic.

8.1.2 VAREF

Both 5V and 3V3 operation is possible through a flexible analogue reference voltage (VAREF) scheme. In addition, input channel CH0 can be used as an alternate reference voltage input to allow both 5V and 3.3V based conversions on the same VADC module.

There are various built-in signal processing modes, which implement anti-aliasing or decimation filters based on the accumulation of up to 4 conversion results, as well as standard FIR (3 coefficients) and IIR (2 coefficients) filter types. These are completely independent of any CPU core.

8.1.3 Safety Aspects

System safety is addressed through three main functions:

- Broken Wire Detection function that checks the connection from the sensor to the input pin.
- Insertion of a strong pull-down device to a channel to show a change in conversion value.
- Validation of the operation of the internal analogue input multiplexer.
- Validation of the operation of the Analogue/Digital converter itself.

Failures detected in these areas are reported to the SMU.

8.1.4 VADC Conversion Request Sources

The VADC makes conversions based on 4 different request sources. The trigger may be automatic, or from an external pin, or another Aurix peripheral. These are:

- Queued request
- Scan Request Source
- Background Scan Request Source
- Synchronization Request Source

8.1.4.1 Queued Request

The VADC converts all the selected channels in a user-defined order. The same channel may be selected for multiple conversions, so that it can achieve a higher effective sampling rate than other channels.

8.1.4.2 Scan Request Source

The VADC converts all the selected channels, starting from the highest numbered channel to the lowest one. This can be made to repeat.

8.1.4.3 Background Scan Request Source

The VADC converts all the selected channels, starting from the highest numbered channel to the lowest one. The channels participating in the scan can come from any enabled VADC module.

8.1.4.4 Synchronization Request Source

Here up to 4 channels, each taken from an independent SAR-ADC group, are converted simultaneously.

8.1.5 VADC Calibration

The VADC has the ability to be calibrated to improve the absolute accuracy of conversions.

Calibration automatically compensates for deviations caused by the manufacturing process, operating temperature and voltage variations. An initial start-up calibration is required once after a reset for all SAR converters. Conversions may be started after the initial calibration sequence. After that, post-calibration cycles will compensate for the effects of drifting parameters. The post-calibration cycles can be disabled.

8.1.6 VADC Configuration

Several parameters can be configured that control the conversion of each analogue input channel.

Channel Parameters - the sample time for a particular channel and the data width of the result are defined via input classes. Each channel can select one of two classes from its own group, or one of two global classes.

Reference selection - Standard reference (VAREF) or CH0.

Alias Feature - conversion requests for channels CH0 and/or CH1 can be redirected to other channels.

Results storage – conversion results can be stored in one of 16 local result registers or in a common results register. Result location can be channel-specific or source-specific. Results can be stored left- or right-aligned. The exact position depends also on the result width and data accumulation mode.

Channel event handling - channel events can be generated whenever a new result value becomes available. The event generation can be restricted to values that lie inside or outside of a user-configurable band. An interrupt can also be generated.

Conversions result data can be pre-processed to a certain extent before storing in a result location and subsequent access by the CPU or a DMA channel.

8.1.7 External Multiplexer Control

The number of analogue channel inputs to the Aurix may be extended by using an external “1 of 8” analogue multiplexer. Here one normal VADC channel is shared between 8 extended channels connected the multiplexer. The select inputs to this are connected to any 3 of the EMUXxx pins in the table below.

The control of the EMUX pins is automatic and the settling time required between a channel change is automatically compensated for by using a different sampling time for the real VADC being driven by the multiplexer.

Table 30 External VADC Multiplexer Control Pins

GPIO Pin	EMUX
P02.6	EMUX00
P33.3	EMUX00
P02.7	EMUX01
P33.2	EMUX01
P02.8	EMUX02
P33.1	EMUX02
P00.6	EMUX10
P33.6	EMUX10
P00.7	EMUX11
P33.5	EMUX11
P00.8	EMUX12
P33.4	EMUX12

Gray code can be used to drive the select lines to avoid intermediate multiplexer switching.

Table 31 EMUX Control Signal Coding

Channel	0	1	2	3	4	5	6	7
Binary	000B	001B	010B	011B	100B	101B	110B	111B
Gray	0	1	11	10	110	111	101	100

8.2 VADC Pin Allocation

The VADC uses the following pins on the TC275. This is in addition to the external multiplexer pins given above.

Table 32 TC275 VADC Pin Allocation

TC275 Pin	Pin Name	SAR Channel	TC275 Pin	Pin Name	SAR Channel
24			**46**	ADC	AN20 VADC2.4
25			**47**	ADC	AN19 VADC2.3
26	VAREF2		**48**	ADC	AN18 VADC2.2
27	VAGND2		**49**	ADC	AN17 VADC2.1
28	ADC	AN47 VADC5.7	**50**	ADC	AN16 VADC2.0
29	ADC	AN46 VADC5.6	**51**	VAGND1	
30	ADC	AN45 VADC5.5	**52**	VAREF1	
31	ADC	AN44 VADC5.4	**53**	VSSM	
32	ADC, P40.9	AN39 VADC4.7	**54**	VDDM	
33	ADC, P40.8	AN38 VADC4.6	**55**	ADC	AN13 VADC1.5
34	ADC, P40.7	AN37 VADC4.5	**56**	ADC	AN12 VADC1.4
35	ADC, P40.6	AN36 VADC4.4	**57**	ADC	AN11 VADC1.3
36	ADC	AN35 VADC4.3	**58**	ADC	AN10 VADC1.2
37	ADC, P40.5	AN33 VADC4.1	**59**	ADC	AN8 VADC1.0
38	ADC, P40.4	AN32 VADC4.0	**60**	ADC	AN7 VADC0.7
39	ADC	AN29 VADC3.5	**61**	ADC	AN6 VADC0.6
40	ADC	AN28 VADC3.4	**62**	ADC	AN5 VADC0.5
41	ADC, P40.3	AN27 VADC3.3	**63**	ADC	AN4 VADC0.4
42	ADC, P40.2	AN26 VADC3.2	**64**	ADC	AN3 VADC0.3
43	ADC, P40.1	AN25 VADC3.1	**65**	ADC	AN2 VADC0.2
44	ADC, P40.0	AN24 VADC3.0	**66**	ADC	AN1 VADC0.1
45	ADC	AN21 VADC2.5	**67**	ADC	AN0 VADC0.0

8.3 Using the VADC ILLD Functions

8.3.1 Setting up the VADC

First the VADC files are included:

```
#include <Vadc/Adc/IfxVadc_Adc.h>
```

Secondly the VADC handle is instantiated:

```
IfxVadc_Adc vadc;
```

Then the VADC module config structure is instantiated and it is filled with default values:

```
IfxVadc_Adc_Config adcConfig;
IfxVadc_Adc_initModuleConfig(&adcConfig, &MODULE_VADC);
```

Then the module is initialized:

```
IfxVadc_Adc_initModule(&vadc, &adcConfig);
```

Now the group config structure is instantiated and filled with default values:

```
IfxVadc_Adc_GroupConfig adcGroupConfig;
IfxVadc_Adc_initGroupConfig(&adcGroupConfig, &vadc);
```

Next the default values are changed for the group config:

```
/* change group (default is GroupId_0, change to GroupId_3) */
adcGroupConfig.groupId = IfxVadc_GroupId_3;
/* IMPORTANT: usually we use the same group as master! */
adcGroupConfig.master = adcGroupConfig.groupId;
/* enable all arbiter request sources */
/* enable Queue mode */
adcGroupConfig.arbiter.requestSlotQueueEnabled = TRUE;
/* enable Scan mode  */
adcGroupConfig.arbiter.requestSlotScanEnabled = TRUE;
/* enable Background scan  */
adcGroupConfig.arbiter.requestSlotBackgroundScanEnabled = TRUE;
/* enable all gates in "always" mode (no edge detection) */
adcGroupConfig.queueRequest.triggerConfig.gatingMode                    =
IfxVadc_GatingMode_always;
adcGroupConfig.scanRequest.triggerConfig.gatingMode                     =
IfxVadc_GatingMode_always;
adcGroupConfig.backgroundScanRequest.triggerConfig.gatingMode           =
IfxVadc_GatingMode_always;
```

The group is then instantiated and initialized:

```
IfxVadc_Adc_Group adcGroup;
IfxVadc_Adc_initGroup(&adcGroup, &adcGroupConfig);
```

Now the ADC channel config structure is instantiated

8.3.2 ILLD Important Functions

8.3.2.1 Add to queue

Adds an A to D conversion to the queue.

```
IfxVadc_Adc_addToQueue(&adcChannel[chnIx], IFXVADC_QUEUE_REFILL);
```

The add to queue function takes the following arguments:

```
adcChannel: The channel on which the conversion will be performed. Defines
whether a conversion in the queue will be copied to the top of the queue
after conversion so that it will be converted again.
```

8.3.2.2 Start Queue

Starts the queued VADC conversions in the group.

```
IfxVadc_Adc_startQueue(&adcGroup);
```

The start queue function takes the following arguments:

```
adcGroup: The group on which to start the queued conversions.
```

8.3.2.3 Get Result

Get the result of a conversion on a particular channel.

```
IfxVadc_Adc_getResult(&adcChannel[chnIx]);
```

The get result function takes the following arguments:

```
adcChannel: The channel to get the completed conversion from.
```

9.

The get result function returns the following:

```
The value of the adc channel.
```

8.3.2.4 Clear Queue

Remove all queued conversions.

```
IfxVadc_Adc_clearQueue(&adcGroup);
```

The clear queue function takes the following arguments:

```
adcGroup: The group from which you want to clear all of the queued
conversions.
```

8.3.2.5 Set Background Scan

Set the ADC to continuously perform conversions on a particular channel, converting selected channels on a group from the higher number channel to the lower one.

```
IfxVadc_Adc_setBackgroundScan(&vadc, &adcGroup, channels, mask);
```

The Set Background Scan function takes the following arguments:

```
adcGroup: The adcGroup to perform the scan on
channels: Enables/disables the VADC channel conversion.
mask: Sets which channels should be modified
continuous: Boolean. Set to true to continuously run conversions.
```

8.3.2.6 Set Background Scan

Set the ADC to continuously perform conversions on ungrouped channels with a low priority.

The set background scan function takes the following arguments:

```
Vadc: The Vadc module
adcGroup: The group on which the conversions are to be done.
Channels: Enables/disables each channel for background scans.
Mask: Sets which channels should be modified
```

8.3.2.7 Start Background Scan

Starts all of the ADC Background Scan.

```
IfxVadc_Adc_startBackgroundScan(&vadc, continuous);
```

The Start Background Scan function takes the following arguments:

```
Vadc: The Vadc module to start the background scan on.
Continuous: Boolean. Set to True to enable continuous conversions.
```

8.4 iLLD Config Options

8.4.1 Channel Config Options

Config Option	Default
channelId	IfxVadc_ChannelId_0
group	NULL_PTR
inputClass	IfxVadc_InputClasses_group0
reference	IfxVadc_ChannelReference_standard
resultRegister	IfxVadc_ChannelResult_0
globalResultUsage	FALSE
lowerBoundary	IfxVadc_BoundarySelection_group0
upperBoundary	IfxVadc_BoundarySelection_group0
boundaryMode	IfxVadc_BoundaryExtension_standard
limitCheck	IfxVadc_LimitCheck_noCheck
synchonize	FALSE
backgroundChannel	FALSE
rightAlignedStorage	FALSE
resultPriority	0x00
resultSrcNr	IfxVadc_SrcNr_group0
resultServProvider	IfxSrc_Tos_cpu0
channelPriority	0x00
channelSrcNr	IfxVadc_SrcNr_group0
channelServProvider	IfxSrc_Tos_cpu0

8.4.2 Group Config Options

Config Option	Default
.arbiter	
.arbiterRoundLength	IfxVadc_ArbitrationRounds_4_slots,
.requestSlotQueueEnabled	FALSE,
.requestSlotScanEnabled	FALSE,
.requestSlotBackgroundScanEnabled	FALSE,
.backgroundScanRequest	
.autoBackgroundScanEnabled	FALSE,
.triggerConfig.gatingMode	IfxVadc_GatingMode_disabled,
.triggerConfig.triggerMode	IfxVadc_TriggerMode_noExternalTrigger,
.triggerConfig.gatingSource	IfxVadc_GatingSource_0,
.triggerConfig.triggerSource	IfxVadc_TriggerSource_0,
.requestSlotPrio	IfxVadc_RequestSlotPriority_low,
.requestSlotStartMode	IfxVadc_RequestSlotStartMode_waitForStart,

Config Option	Default
.scanRequest	
.autoscanEnabled	FALSE,
.triggerConfig.gatingMode	IfxVadc_GatingMode_disabled,
.triggerConfig.gatingSource	IfxVadc_GatingSource_0,
.triggerConfig.triggerMode	IfxVadc_TriggerMode_noExternalTrigger,
.triggerConfig.triggerSource	IfxVadc_TriggerSource_0,
.requestSlotPrio	IfxVadc_RequestSlotPriority_low,
.requestSlotStartMode	IfxVadc_RequestSlotStartMode_waitForStart,
.queueRequest	
.flushQueueAfterInit	TRUE,
.triggerConfig.gatingMode	IfxVadc_GatingMode_disabled,
.triggerConfig.gatingSource	IfxVadc_GatingSource_0,
.triggerConfig.triggerMode	IfxVadc_TriggerMode_noExternalTrigger,
.triggerConfig.triggerSource	IfxVadc_TriggerSource_0,
.requestSlotPrio	IfxVadc_RequestSlotPriority_low,
.requestSlotStartMode	IfxVadc_RequestSlotStartMode_waitForStart,
.inputClass[0].resolution	IfxVadc_ChannelResolution_12bit,
.inputClass[0].sampleTime	1.00E-06s
.inputClass[1].resolution	IfxVadc_ChannelResolution_12bit,
.inputClass[1].sampleTime	1.00E-06s
***config**	IfxVadc_Adc_defaultGroupConfig
config->groupId	IfxVadc_GroupId_0
config->module	vadc
config->master	config->groupId
config->disablePostCalibration	FALSE

9 Micro Second Channel (MSC)

9.1 Introduction to Aurix MSC

The Micro Second Channel (MSC) is a high speed serial interface which is designed to connect external multi-channel power devices to the microcontroller. The serial data transmission capability minimizes the number of pins required for connection. The TC27x MSC module contains two MSC serial interfaces, MSC0 and MSC1 and each is able to connect to up to four external power devices. Thus the MSC is an advanced form of IO expansion.

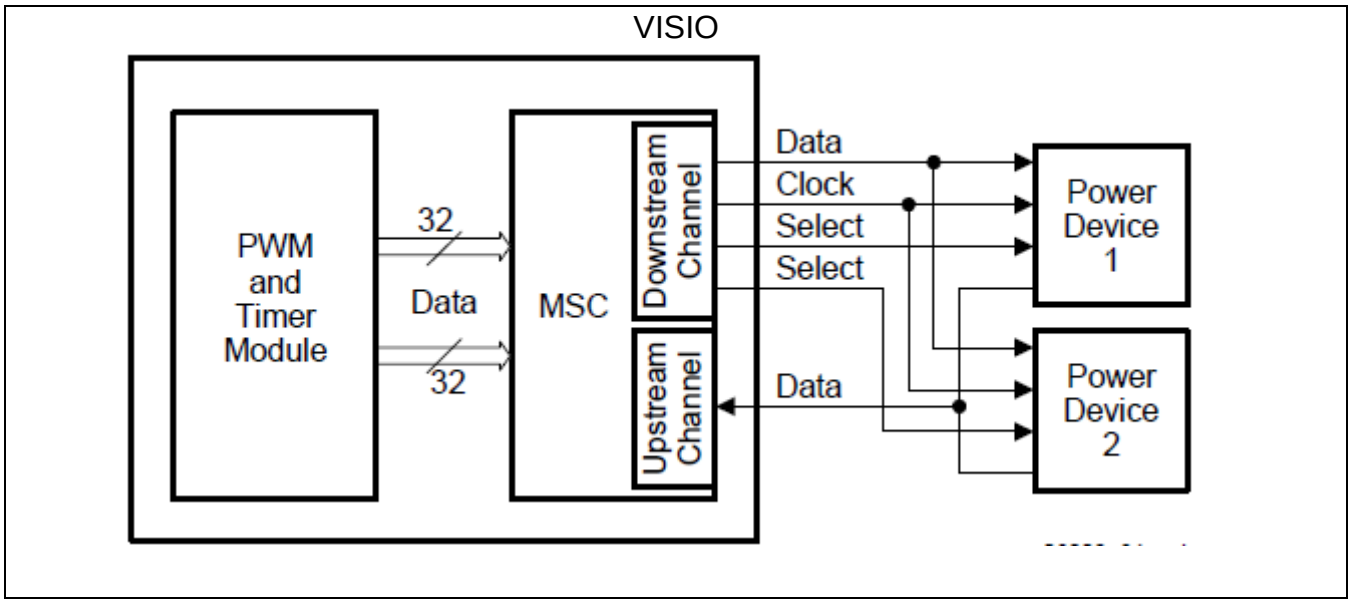


Figure 23IO Expansion With The MSC.

One of the unique features of the MSC is the ability to route event triggers from Aurix peripherals such as the GTM out to external power drivers. As an example, a PWM waveform generated from a TOM (Timer Output Module) can be sent to a power switching pin on an MSC-connected device serially rather than directly driving a pin directly on the AURIX package. This allows the output capabilities of the AURX to be considerably extended.

Command information or parallel data information (coming from the timer units) is sent out to the power device via a high-speed synchronous serial data stream (downstream channel). The MSC receives data and status back from the power device via a low-speed asynchronous serial data stream (upstream channel).

The MSC communicates with the outside world via nine I/O lines. Eight output lines are required for the serial communication of the downstream channel (clock, data, and enable signals). One out of eight input lines SDI[7:0] is used as serial data input signal for the upstream channel. The source of the serial data to be transmitted by the downstream channel can be MSC register contents or data that is provided at the ALTINL/ALTINH input lines. These input lines are typically connected to other on-chip peripheral units (for example with a timer unit like the GTM).

Table 33 GTM to MSC Connections

GTM Output	MSC Input
GTM_MSC0ALTINL[15:0]	MSC0_ALTINL[15:0]
GTM_MSC0ALTINH[15:0]	MSC0_ALTINLE[15:0]
GTM_MSC0ALTINH[15:0]	MSC0_ALTINH[15:0]
GTM_MSC0ALTINL[15:0]	MSC0_ALTINHE[15:0]
GTM_MSC1ALTINL[15:0]	MSC1_ALTINL[15:0]
GTM_MSC1ALTINH[15:0]	MSC1_ALTINLE[15:0]
GTM_MSC1ALTINH[15:0]	MSC1_ALTINH[15:0]
GTM_MSC1ALTINL[15:0]	MSC1_ALTINHE[15:0]

For supporting differential output drivers, the serial clock output FCL and the serial data output SO are available in both polarities, indicated by the signal name suffix “P” and “N”.

An emergency stop input signal makes it possible to set bits of the serial data stream to dedicated values in emergency case.

Clock control, address decoding, and interrupt service request control are managed outside the MSC module kernel. Service request outputs are able to trigger an interrupt or a DMA request.

The MSC downstream channel uses three types of frame formats for operation:

- Command Frames
- Data Frames
- Passive Time Frames

The MSC module can operate in two modes: Standard (up to 32 data bits) or Extended (up to 64 data bits). The mode is selected by using the bit DSCE.EXEN.

The downstream channel of the MSC makes it possible to select between two transmission modes:

- Triggered Mode
- Data Repetition Mode

In Triggered Mode, command frames or data frames are sent out as a result of a software event. In Data Repetition Mode, data frames are sent out continuously without any software interaction. In the time gap between two consecutive data frames, passive time frames can be inserted.

9.2 Clock Control and Baudrate

In TC27x, the MSC module is provided with the following clock signals:

- f_{CLC} - This is the module clock that is used inside the MSC kernel for control purposes such as clocking of control logic and register operations. The frequency of f_{CLC} is always identical to the system clock frequency f_{SPB}. The clock control register MSC_CLC makes it possible to enable/disable f_{CLC} under certain conditions.

- f_{MSC} - This clock is the module clock that is used inside the MSC for baud rate generation of the serial upstream and downstream channel. The fractional divider register MSC_FDR controls the frequency of f_{MSC} and makes it possible to enable/disable it independent of f_{CLC}.

- f_{MSCA} – This clock is the downstream clock which is used together with the ABRA (Asynchronous Baud Rate Adjustment) block. In this case, the upstream baud rate can be fine-tuned with the fractional divider, and the downstream channel frequency is configured with an n-divider.

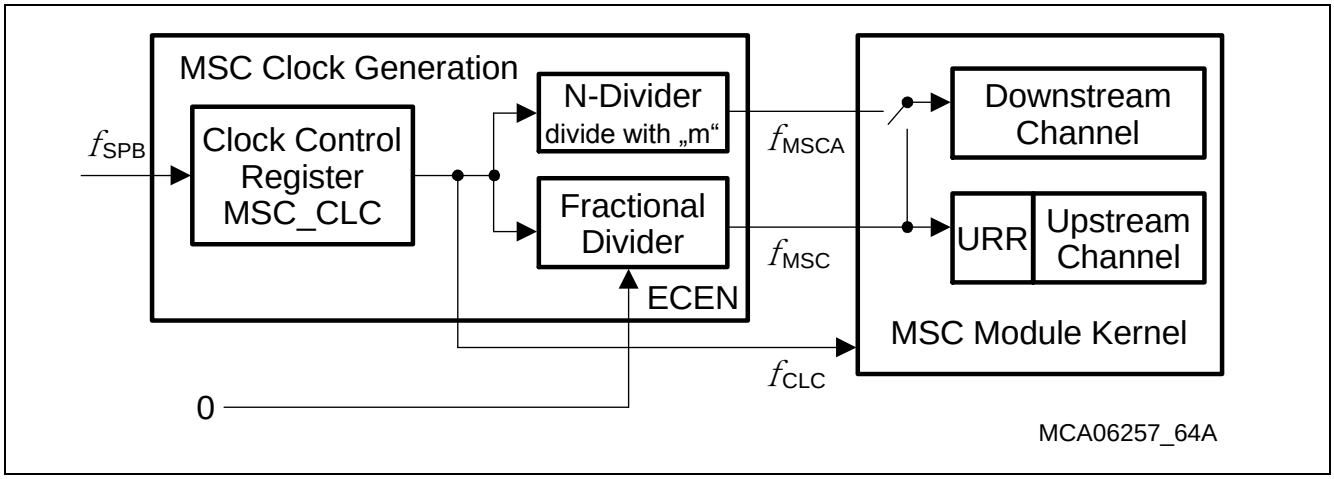


Figure 24MSC Module Clock Generation

There is an optional ABRA (Asynchronous Baud Rate Adjustment) block, which allows the use of asynchronous frequencies (f_{ERAY}) for generating the baud rate, as well as f_{SRI} and f_{SPB}. With f_{ERAY}, it is possible to configure the maximum baud rate of 40 MBaud, as well as 20MBaud, something which is not possible with the f_{SPB}.

This ABRA block is more or less a “simple” Synchronization FIFO, which can be used or bypassed, as required.

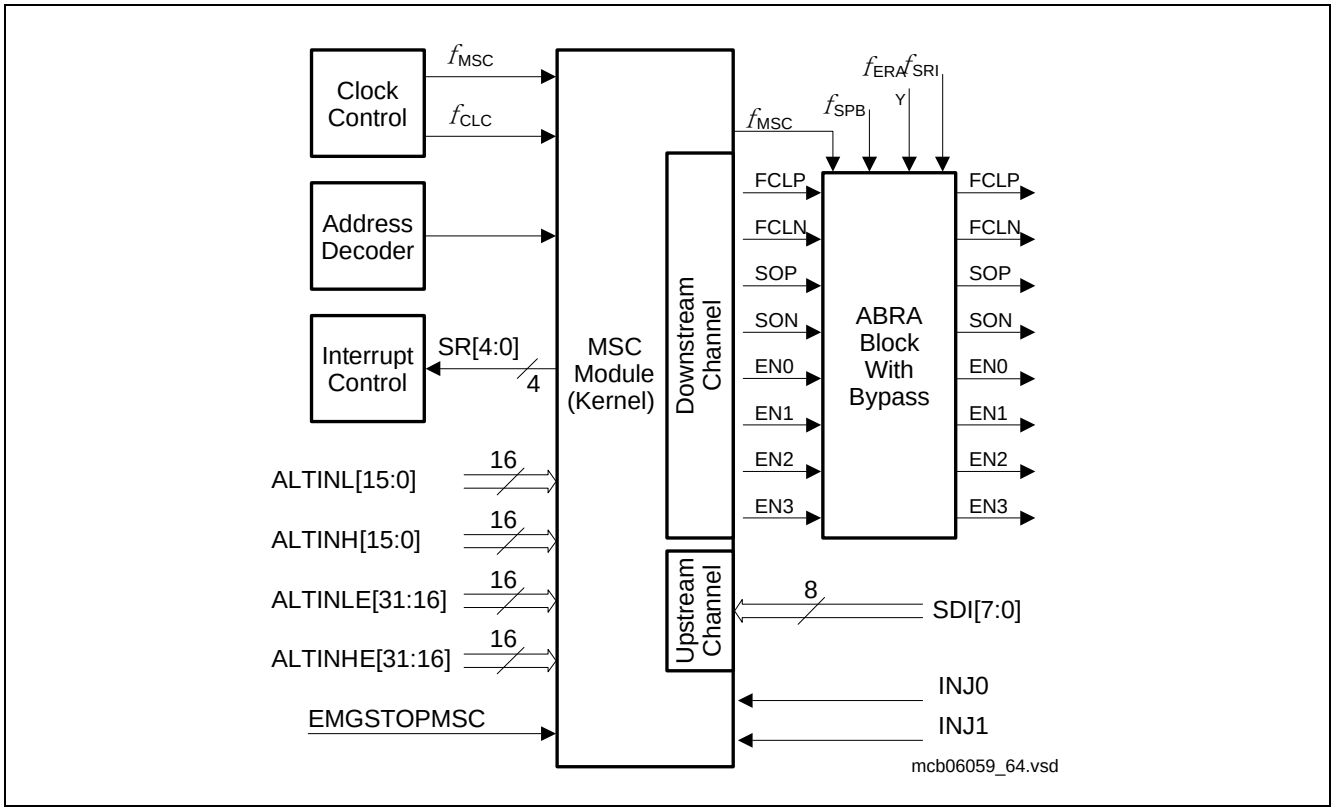


Figure 25Block Diagram of the MSC Interface

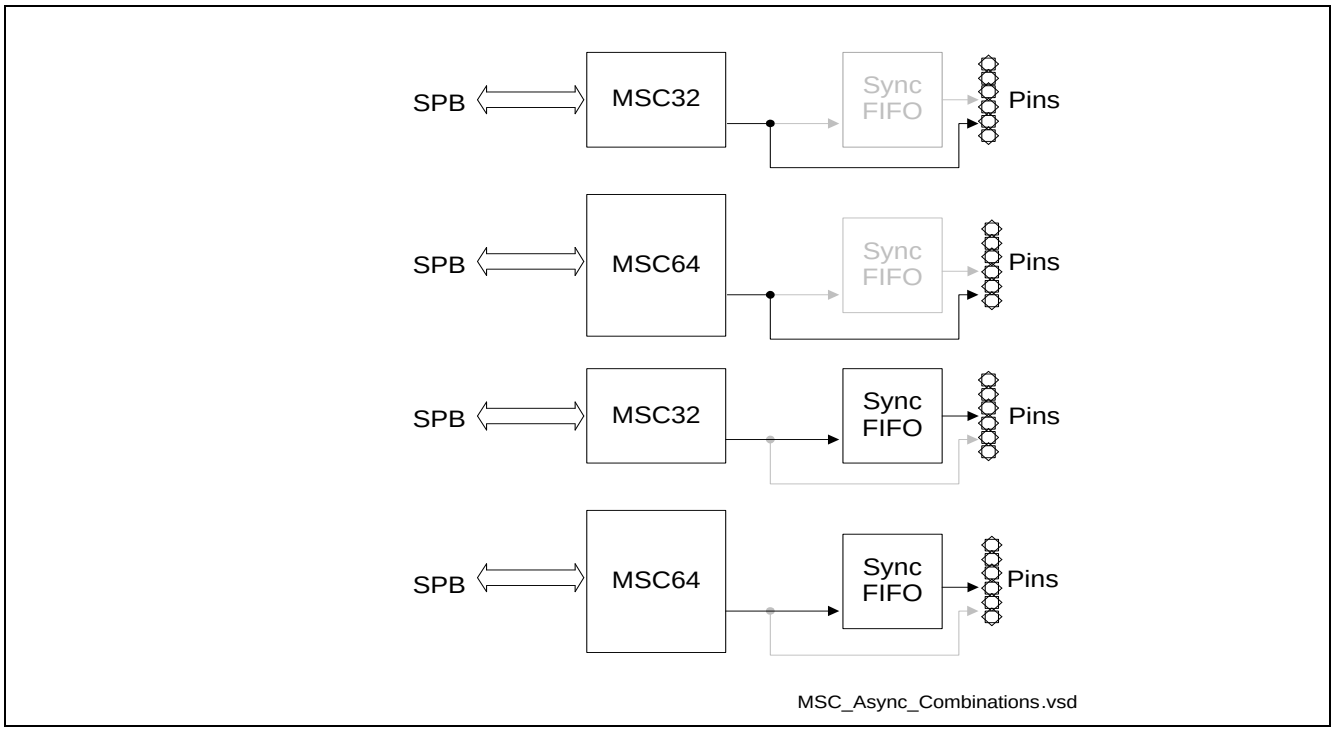


Figure 26MSC Use-Case Combinations

When the ABRA block is not used, the additional N-Divider (dividing with the division factor named “m”) will not be used. That is, when the ABRA block is bypassed, only the fractional divider will be used. In this case, the Baudrate calculations are as given below:

Downstream Baudrate in Normal Divider Mode = f_{SPB} / (2 * (1024 - MSC.FDR.STEP))

Downstream Baudrate in Fractional Divider Mode = f_{SPB} * MSC.FDR.STEP / 2048

Upstream Baudrate in Normal Divider Mode = f_{SPB} / (DF * (1024 - MSC.FDR.STEP))

Upstream Baudrate in Fractional Divider Mode = f_{SPB} * MSC.FDR.STEP / (DF * 1024)

9.3 MSC interface

9.3.1 Interfacing MSC with TLE8718SA

TLE8718SA is a low-side switch power device from Infineon, which can be interfaced with the microcontroller and controlled via the MSC.

Table 34 Interfacing TC27x to TLE8718SA

TC27x	Comment	TLE8718SA
MSCx.ENy	Slave Select	SSY
MSCx.FCLN	Differential Clock Signal N	FCLN
MSCx.FCLP	Differential Clock Signal P	FCLP
MSCx.SON	Differential Downstream Serial Data N	SIN
MSCx.SOP	Differential Downstream Serial Data P	SIP
MSCx.SDIy	Upstream Serial Data	SDO
Digital Output Pin	TLE8718SA Reset input	RST
Digital Output Pin	Disable Pin for OUT5…OUT10	DIS5_10

9.3.2 TC27x MSC port pin mapping

The MSC peripheral may be accessed via the pins given in the table below. Useful macros and structures for accessing the MSC in 'C' may be found in the programming examples:

```
.\0_Src\4_McHal\Tricore\_PinMap\IfxMsc_PinMap.c
.\0_Src\4_McHal\Tricore\_PinMap\IfxMsc_PinMap.h
```

Table 35 Port pin mapping

Signal	MSC0	MSC1
EN0	P10.2 P10.3 P10.4 P11.11 P14.10 P15.5	P23.4 P32.4
EN1	P10.1 P11.2 P13.0 P14.9 P15.3	P23.5
EN2	P10.2 P10.3 P10.4 P11.11 P14.10 P15.5	P23.4 P32.4
EN3	P10.1 P11.2 P13.0 P14.9 P15.3	P23.5
FCLP	P11.6 P13.1 P13.2	P22.1
FCLN	P13.0	P22.0
FCLND	P13.0	P22.0
SOP	P11.9 P13.3	P22.3
SON	P13.2	P22.2
SOND	P13.2	P22.2
SDI0	P11.10	P23.1
SDI1	P10.2	P02.3
SDI2	P14.3	P32.4
SDI3	P11.3	
INJ0	P00.0	P23.3
INJ1	P10.5	P33.13

9.4 Getting the MSC running using the iLLD

In this example, MSC1 is used to communicate with TLE8718SA - Smart 18-Channel Lowside Switch from Infineon.

9.4.1 Port pin selection/configuration

Choose and configure the various port pins required for the MSC communication.

For example:

```
static const IfxMsc_Msc_Io IfxMsc_PinMap[IFXMSC_NUM_MODULES] = {
  {
    {&IfxMsc0_FCLP_P13_1_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_FCLN_P13_0_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_SOP_P13_3_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_SON_P13_2_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_EN0_P10_2_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_EN1_P11_2_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc0_SDI0_P11_10_IN, IfxPort_InputMode_pullUp},
    {&IfxMsc0_INJ0_P00_0_IN, IfxPort_InputMode_pullUp},
    {&IfxMsc0_INJ1_P10_5_IN, IfxPort_InputMode_pullUp},
    IfxPort_PadDriver_cmosAutomotiveSpeed3
  },
  {
    {&IfxMsc1_FCLP_P22_1_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_FCLN_P22_0_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_SOP_P22_3_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_SON_P22_2_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_EN0_P23_4_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_EN1_P23_5_OUT, IfxPort_OutputMode_pushPull},
    {&IfxMsc1_SDI0_P23_1_IN, IfxPort_InputMode_pullUp},
    {&IfxMsc1_INJ0_P23_3_IN, IfxPort_InputMode_pullUp},
    {&IfxMsc1_INJ1_P33_13_IN, IfxPort_InputMode_pullUp},
    IfxPort_PadDriver_cmosAutomotiveSpeed3
  }
};
```

9.4.2 Create and update MSC module configuration

Create a default MSC configuration and then update it with the required configuration.

```
IfxMsc_Msc_Config mscConfig;
IfxMsc_Msc_initModuleConfig(&mscConfig, &MODULE_MSC0);

/* TLE8718SA specific configurations */
/* clock is enabled always */
mscConfig.outputControlConfig.fclClockControl                                    =
IfxMsc_FclClockControlEnabled_always;
/* Insert low level selection bit for SRL */
mscConfig.downstreamConfig.srlActivePhaseSelection                               =
IfxMsc_ActivePhaseSelection_lowLevel;
/* Upstream rate is fMSC/128 */
mscConfig.upstreamConfig.upstreamChannelReceivingRate                            =
IfxMsc_UpstreamChannelReceivingRate_128;
/* Command frame length is 16 bits */
mscConfig.downstreamConfig.commandFrameLength                                    =
IfxMsc_CommandFrameLength_16;
/* As data length is 16 bits, No need for SRH bits */
mscConfig.downstreamConfig.srhDataFrameLength = IfxMsc_DataFrameLength_0;
```

9.4.3 Initialize MSC as per the configuration

Initialize the MSC as per the configuration made.

```
  /* initialize MSCs */
  for (i = 0; i < IFXMSC_NUM_MODULES; ++i)
  {
    /* init module pointer */
    mscConfig.msc = (Ifx_MSC *)IfxMsc_cfg_indexMap[i].module;

    /* IO Config */
    mscConfig.io = IfxMsc_PinMap[i];

    /* initialize module */
    IfxMsc_Msc_initModule(&g_MscBasic.msc[i], &mscConfig);
  }
```

9.4.4 TLE8718SA specific initialization

```
/* Update MSC1_FDR (0xF000270C) */
   passwd = IfxScuWdt_getCpuWatchdogPassword();
   IfxScuWdt_clearCpuEndinit(passwd);
   FDR = *((Ifx_MSC_FDR *)0xF000270Cu);
   FDR.B.STEP = 1020;
   *((Ifx_MSC_FDR *)0xF000270Cu) = FDR;
   IfxScuWdt_setCpuEndinit(passwd);

/* P00.2 - DIS5_10 */
   IfxPort_setPinModeOutput(&MODULE_P00,   2,   IfxPort_OutputMode_pushPull,
IfxPort_OutputIdx_general);
   IfxPort_setPinHigh(&MODULE_P00,2);

/* P00.2 - RST – Pull it low and then high to ensure TLE8718SA is reset */
   IfxPort_setPinModeOutput(&MODULE_P00, 4, IfxPort_OutputMode_pushPull,
                            IfxPort_OutputIdx_general);
   IfxPort_setPinLow(&MODULE_P00,4);
   while(MscDemo_Counter < 0x100000U){MscDemo_Counter++;}
   IfxPort_setPinHigh(&MODULE_P00,4);

/* 8 cycles delay for MSC interface of TLE8718SA to be functional */
 __nop();
 __nop();
 __nop();
 __nop();
 __nop();
 __nop();
 __nop();
 __nop();

/* MSC1 command: WR_START */
   IfxMsc_Msc_sendCommand(&g_MscBasic.msc[1], 0xB);
```

9.4.5 Transmission of data via high speed serial downstream channel

Send the required data via the downstream channel.

```
/* In the triggered mode of transmission, it should be ensured to transmit
successive data frames before the expiry of tMSC_mon timeout by TLE8718SA*/
/* Note:
   -  Only MSCdataL (16-bit data) is applicable for TLE8718SA
   -  if the data bit is '1', TLE8718SA turns OFF the corresponding stage
   -  if the data bit is '0', TLE8718SA turns ON the corresponding stage
*/
   IfxMsc_Msc_sendData(&g_MscBasic.msc[1], MSCdataL, MSCdataH);
```

9.4.6 Other important iLLD functions

9.4.6.1 Send Data Extension

Sends downstream data extension (64-bit)

```
IfxMsc_Msc_sendDataExtension(&g_MscBasic.msc[1], MscData, MscDataExt);
```

This function takes the following arguments:

```
&g_MscBasic.msc[1]: Pointer to MSC1 handle
MscData: 32-bit MSC data
MscDataExt: 32-bit MSC data extension
```

10.

9.4.6.2 Receive Data

Receives MSC upstream data

```
IfxMsc_Msc_receiveData(&g_MscBasic.msc[1], upstreamIdx);
```

This function takes the following arguments:

```
&g_MscBasic.msc[1]: Pointer to MSC1 handle
upstreamIdx: Index of the upstream data register
```

This function returns:

```
Upstream data
```

9.4.6.3 Set Command Target

Sets the target for Command

```
IfxMsc_Msc_setCommandTarget(&g_MscBasic.msc[1], enX);
```

This function takes the following arguments:

```
&g_MscBasic.msc[1]: Pointer to MSC1 handle
enX: Target to be selected
```

9.4.6.4 Set Data Target

Sets the target for Data

```
IfxMsc_Msc_setDataTarget(&g_MscBasic.msc[1], enXHigh, enXLow);
```

This function takes the following arguments:

```
&g_MscBasic.msc[1]: Pointer to MSC1 handle
enXHigh: High target to be selected
enXLow: Low target to be selected
```

11.

9.5 iLLD Config Options

9.5.1 Clock Config Options

Config Option	Default
baudrate	3125000
dividerMode	IfxMsc_DividerMode_normal
Step	0

9.5.2 Upstream Config Options

Config Option	Default
upstreamChannelFrameType	IfxMsc_UpstreamChannelFrameType_12bit
upstreamChannelReceivingRate	IfxMsc_UpstreamChannelReceivingRate_16
Parity	IfxMsc_Parity_even
serviceRequestDelay	IfxMsc_ServiceRequestDelay_noDelay
upstreamTimeoutPrescaler	IfxMsc_UpstreamTimeoutPrescaler_32768
upstreamTimeoutValue	IfxMsc_UpstreamTimeoutValue_16

9.5.3 Interrupt Config Options

Config Option	Default
dataFrameInterruptNode	IfxMsc_DataFrameInterruptNode_SR0
dataFrameInterrupt	IfxMsc_DataFrameInterrupt_disabled
commandFrameInterruptNode	IfxMsc_CommandFrameInterruptNode_SR0
commandFrameInterrupt	IfxMsc_CommandFrameInterrupt_disabled
timeFrameInterruptNode	IfxMsc_TimeFrameInterruptNode_SR0
timeFrameInterrupt	IfxMsc_TimeFrameInterrupt_disabled
receiveDataInterruptNode	IfxMsc_ReceiveDataInterruptNode_SR0
receiveDataInterrupt	IfxMsc_ReceiveDataInterrupt_disabled
upstreamTimeoutInterruptNode	IfxMsc_UpstreamTimeoutInterruptNode_SR0
upstreamTimeoutInterrupt	IfxMsc_UpstreamTimeoutInterrupt_disabled
overflowInterruptNode	IfxMsc_OverflowInterruptNode_SR0
overflowInterrupt	IfxMsc_OverflowInterrupt_disabled
underflowInterruptNode	IfxMsc_UnderflowInterruptNode_SR0
underflowInterrupt	IfxMsc_UnderflowInterrupt_disabled

9.5.4 Output Control Config Options

Config Option	Default
fclpPolarity	IfxMsc_FclLinePolarity_nonInverted
sopPolarity	IfxMsc_SoLinePolarity_nonInverted
cslpPolarity	IfxMsc_ChipSelectActiveState_low
sdiLinePolarity	IfxMsc_SdiLinePolarity_likeSi

fclClockControl	IfxMsc_FclClockControlEnabled_activePhaseOnly

9.5.5 Downstream Config Options

Config Option	Default
transmissionMode	IfxMsc_TransmissionMode_triggered
srlDataFrameLength	IfxMsc_DataFrameLength_16
srhDataFrameLength	IfxMsc_DataFrameLength_16
srlActivePhaseSelection	IfxMsc_ActivePhaseSelection_none
srhActivePhaseSelection	IfxMsc_ActivePhaseSelection_none
commandFrameLength	IfxMsc_CommandFrameLength_32
dataFramePassivePhaseLength	IfxMsc_DataFramePassivePhaseLength_2
passiveTimeFrameCount	IfxMsc_PassiveTimeFrameCount_0
externalSignalInjectionPin0	IfxMsc_ExternalSignalInjection_disabled
injectionPositionPin0	IfxMsc_ExternalBitInjectionPosition_0
externalSignalInjectionPin1	IfxMsc_ExternalSignalInjection_disabled
injectionPositionPin1	IfxMsc_ExternalBitInjectionPosition_0
commandDataCommandReceptionMode	IfxMsc_CommandDataCommandRepetitionMode_disabled
downstreamDataSourcesLow	0
downstreamDataSourcesHigh	0
emergencyStopEnableBits	0

9.5.6 Downstream Extension Config Options

Config Option	Default
Extension	IfxMsc_Extension_disabled
srlBitsShiftedAtDataFramesExtension	IfxMsc_MsbBitDataExtension_notPresent
srhBitsShiftedAtDataFramesExtension	IfxMsc_MsbBitDataExtension_notPresent
downstreamExtensionDataSourcesLow	0
downstreamExtensionDataSourcesHigh	0
emergencyStopExtensionEnableBits	0
dataFrameExtensionPassivePhaseLength	IfxMsc_DataFrameExtensionPassivePhaseLength_0
controlFrameExtensionPassivePhaseLength	IfxMsc_ControlFrameExtensionPassivePhaseLength_0
nDividerDownstream	IfxMsc_NDividerDownstream_1

9.5.7 ABRA Config Options

Config Option	Default
abraDownstreamBlockBaudrate	500000
lowPhaseOfShiftClock	IfxMsc_ShiftClockPhaseDuration_1
highPhaseOfShiftClock	IfxMsc_ShiftClockPhaseDuration_1
clockSelectAbra	IfxMsc_ClockSelect_fspb
nDividerAbra	IfxMsc_NDividerAbra_1
abraBlockBypass	IfxMsc_AsynchronousBlock_bypassed

9.5.8 I/O Config Options

Config Option	Default
Fclp.pin	NULL_PTR
Fclp.mode	IfxPort_OutputMode_pushPull
Fcln.pin	NULL_PTR
Fcln.mode	IfxPort_OutputMode_pushPull
Sop.pin	NULL_PTR
Sop.mode	IfxPort_OutputMode_pushPull
Son.pin	NULL_PTR
Son.mode	IfxPort_OutputMode_pushPull
En0.pin	NULL_PTR
En0.mode	IfxPort_OutputMode_pushPull
En1.pin	NULL_PTR
En1.mode	IfxPort_OutputMode_pushPull
Sdi.pin	NULL_PTR
Sdi.mode	IfxPort_InputMode_pullUp
Inj0.pin	NULL_PTR
Inj0.mode	IfxPort_InputMode_pullUp
Inj1.pin	NULL_PTR
Inj1.mode	IfxPort_InputMode_pullUp
pinDriver	IfxPort_PadDriver_cmosAutomotiveSpeed3

10 On-Chip FLASH Programming

The program FLASH can be programmed via JTAG, DAP or CAN and ASC bootstrap loaders. During development, the most commonly used programming method is a debugger, e.g. PLS UDE, whereby the executable in an ELF file format is downloaded prior debugging. This uses the JTAG or DAP interface to the Aurix's on-chip debug resources. Where no debugger is present, the free Infineon MEMTOOL is often used. This is a conventional JTAG/DAP FLASH programming tool for PCs, which loads an Intel-format .HEX file into the FLASH. MEMTOOL is normally used as a Windows application, but it can be controlled from a batch file where necessary.

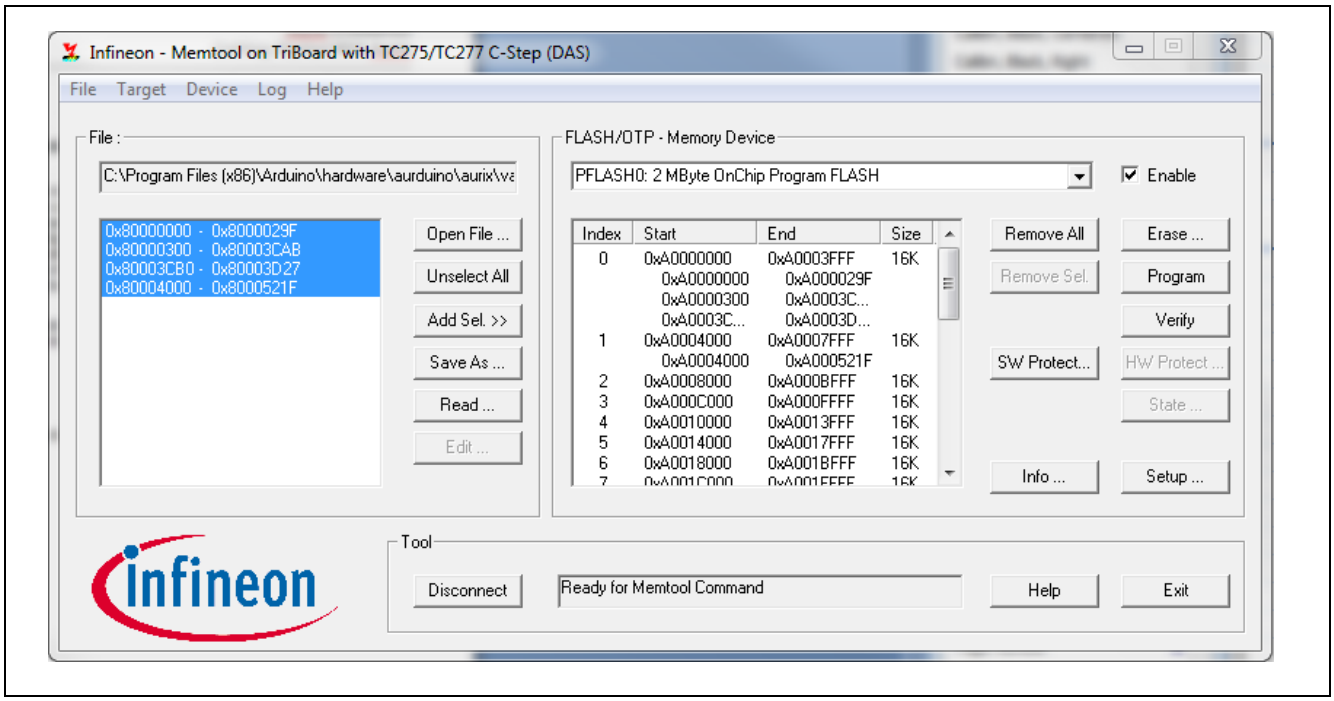


Figure 27 Infineon MEMTOOL FLASH Programming Tool

It can also be incorporated into the Eclipse IDE to give a single-click program download.

MEMTOOL is also available as a commercial application from PLS in Germany, in which form it can be used for production line programming, often being called as a .DLL from a test environment of a complete board tester.

The program FLASH can also be programmed through the Aurix's serial and CAN bootstrap loader modes, although the tools for doing this are chargeable. This is similar to the situation where FLASH programming is to be carried out by an user-defined bootloader installed in the PFLASH. Here too there are ready-made drivers in source-code form available from Infineon and Hitex to handle the programming algorithms and low-level access to the on-chip Flash controller.

11 On- Chip Debug Support (OCDS)

11.1 On-Chip Debug Support (OCDS)

Like the TriCore (AUDOMAX), there is a powerful emulator-like debugger built into the Aurix that is accessed through either a conventional JTAG port or the newer “Device Access Port” or DAP. The JTAG uses 5 basic pins TDI, TDO, TRST, TMS and TCK whereas DAP can use as few as 2. The same range of pins is used for these two interfaces. JTAG is limited to 4 Mbyte/s read accesses whereas even the 2-pin DAP can achieve 15MByte/s. For most Aurix variants the DAP option is recommended.

Table 36 JTAG/DAP Options

	Block R/W [MByte/s]	Pins	Pins	Pins	Pins
JTAG	4.5 W 4.0 R	TCK	TMS	TDO	TFI
DAP	15 W 15 R	DAP0	DAP1	GPIO or TGI/O3	GPIO or TGI/O2
DAP 3PU 3 Pin Unidir	15 W 15 R	DAP0	DAP1	DAP2	GPIO or TGI/O2
DAP WM 3 pin Wide Mode	30 W 30 R	DAP0	DAP1	DAP2	GPIO or TGI/O2

The DAP 3-pin wide mode can reach 30MByte/s. Unlike JTAG, DAP has a built-in CRC for error checking. Unused JTAG pins in the slower and narrower DAP operating modes can be used as trigger pins that are driven by events such as read or write, occurring to specific addresses in the Aurix.

For basic debugging such as FLASH download, single stepping, run-to-breakpoint etc. JTAG is adequate but if the special Aurix emulation devices (“ED”) are used with their 768k (or larger) debug/trace RAM blocks, then DAP is definitely preferable. In addition, some devices such as the TC26x and TC23x only support DAP.

Significantly the “3-wire unidir” mode uses 3 pins, but as there is no bi-directional usage, LVDS can be used to connect to external trace tools. This allows very high transfer speeds over long distances such as might be useful where the Aurix is physically hard to access, as it is installed in a large machine.

11.2 Debugging Via CAN

An alternative means of accessing the OCDS is using the “debug over CAN” mode. The Aurix DXCPL pins P14.0 (TXDCAN1) and P14.1 (RXDCAN1) are able to recognise a particular bit sequence which will enable access to the OCDS via CAN. A suitable debugger (e.g. PLS UAD2) with a CAN interface can then allow normal debug functions to be used.

11.3 Aurix Emulation Devices

The Aurix devices are available in two forms. The “normal” type which can be used for development and series production “TP” suffix and a special development/debug version with a “TE” suffix in the part number. The two types differ in the size of the on-chip emulation/debug RAM and the capabilities of the debug unit. These are known as the “Emulation Device” or “ED” variants that have effectively a full in-circuit emulator with trace built in -- the “TE” marking means that it is an “ED” device. Externally the standard and ED parts are identical, so boards destined for development use can be built with ED parts and then be connected to a suitable external debugger unit (e.g. PLS UAD2 with special extension licence).

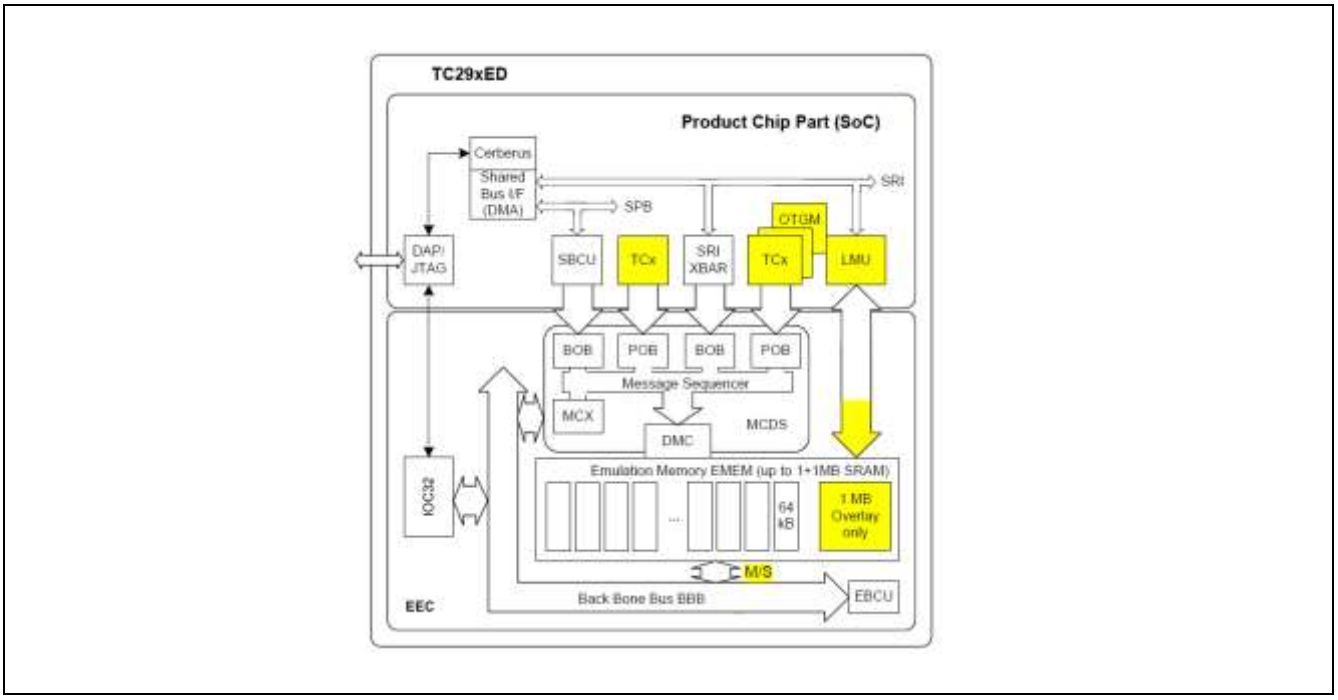


Figure 28 TC29xED Emulation Device Internal Structure

The ED trace system uses the 1024k EMEM RAM (TC27x) block to hold a real time recording of program execution with no impact on real time behaviour. Any two CPUs can be traced at once. The information captured by the trace is filtered on the input, so that only useful data is recorded. This massively reduces the quantity of data that has to be serially transferred through the DAP to an external PC-based tool that actually displays the traced information, so that the monitoring can be continuous and gapless.

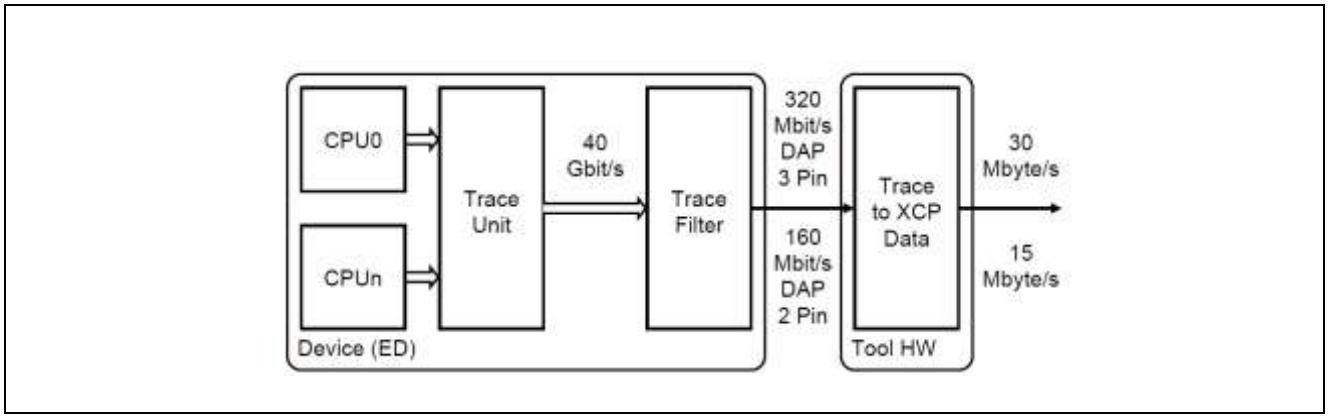
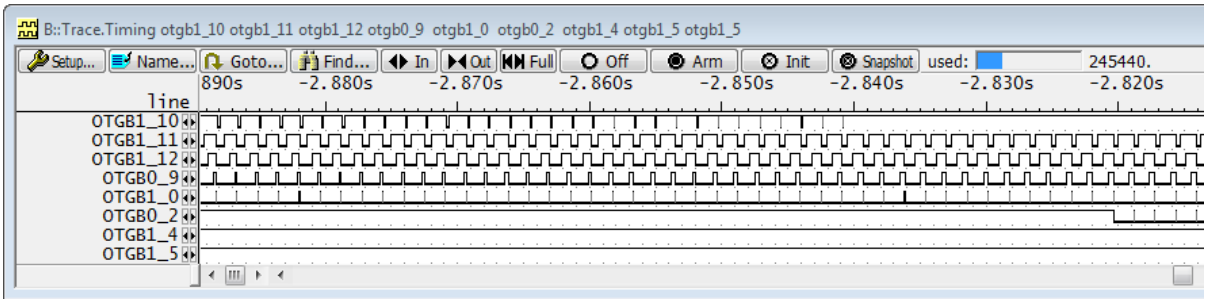


Figure 29

Figure 30ED Trace Overview

In situations where the Aurix-based system is physically inaccessible, e.g installed in a car, the CAN interface can be used to access the trace system. The PLS UAD debugger series is equipped with a CAN port for this (and other) reasons.

The trace contents are not limited to instruction execution and data read/writes. Using the logic analyser function, the operation of peripherals can be recorded. For example, IO pins being used for PWM from the GTM can be traced.



There are also specific triggers and filters for the MultiCAN module, so that the MCDS can be used as a CAN traffic analyser but with no external hardware.

By recording just the entry and exits of major functions, a real time performance analysis is possible over long periods of execution. This is the “compact function trace”. To keep the data rate within the 30Mbyte/s limit of DAP, very small leaf functions (< 10 cycles) are not recorded. With larger leaf functions of between 10 cycles and 30 cycles, just the call to the function is recorded. This helps to prevent normally hidden compiler-generated functions from stealing DAP bandwidth.

The MCDS solves one well known challenge in debugging and that is tracing through a power-on reset, such that events leading up to and away from a complete device reset can be recorded. The MCDS delays the reset so that an end-of-trace marker can be written into the trace RAM before the reset actually takes place.

w w w . h i t e x . c o . u k

Published by Hitex (UK) Ltd.