Microcontroller Training

Controller: TC1796

www.infineon.com/microcontroller

MICROCONTROLLERS

Never stop thinking

Agenda

- Introduction
- Preparation
- Guided Tour
  - Exercise 1: hello, world
  - Exercise 2: LED
  - Exercise 3: Saturation
  - Exercise 4: Interrupts
  - Exercise 5: ASC
  - Exercise 6: Programming the flash
  - Exercise 7: Puls Width Modulation
  - Exercise 8: Pipeline
  - Exercise 9: CAN
  - Exercise 10: Peripherial Control Processor

Introduction

Training Overview

Intended Audience

Field Application Engineers, Application Engineers

Description

This course provides you with detailed information about Infineons 32-bit microcontroller architecture. You will be able to explain and demonstrate the advantages of the TriCore architecture.

Prerequisites

- Basic understanding of microcontroller and C programming
- W2K or XP Notebook w/ CD drive
- 700MB free disk space, 512MB RAM
- 1 parallel port, 1 serial port
- Write access to the BIOS
- Administrator rights

Estimated Learning Time

6 hours

Introduction

TC1796 Market & Applications

Target Market #1: Industrial
Applications:
- High-End Drives
- Programmable Logic Controller (PLC)
- Programmable Automation Controller (PAC)
- Industrial Control

Target Market #2: Automotive
Applications:
- Engine control
- Transmission control
- By-wire systems

TriCore Architecture

The Infineon TriCore™ architecture combines the best of three worlds: RISC, DSP and µ-Controller together in a single core to offers maximum system performance for embedded real-time applications.

Introduction

TriCore Features and Benefits

TriCore Features	TriCore Benefits
➢ High Performance Core with 4-stage pipeline and triple issue super-scalar implementation. HW supported context switch in 2 clock cycles	➢ Optimized chip-size to performance ratio for real-time critical embedded systems
➢ Local Memory Bus (LMB) with 64 bits data and separated busses used for program and data	➢ Separated instruction and data busses speed up the system performance and avoids arbitration conflicts
➢ Mixed 16-/32-bit instruction format	➢ Optimized code density for embedded flash memory usage
➢ Multi-master interrupt system with HW controlled context switch, 255 priority levels and fast interrupt response time	➢ Sophisticated interrupt system with up to 255 HW arbitrated sources and very fast response times is optimized for real-time sensitive embedded applications
➢ Powerful MAC unit supports circular buffer; no data overflow faults due to saturating arithmetic and bit-reverse addressing modes for DSP algorithms	➢ Given scalability approach due to MCU and DSP function merged in one core. Only one tool set for development
➢ Single precision IEEE-754 Floating Point Unit (FPU) with integrated interrupt capability for exception handling	➢ Balanced data precision and performance requirement

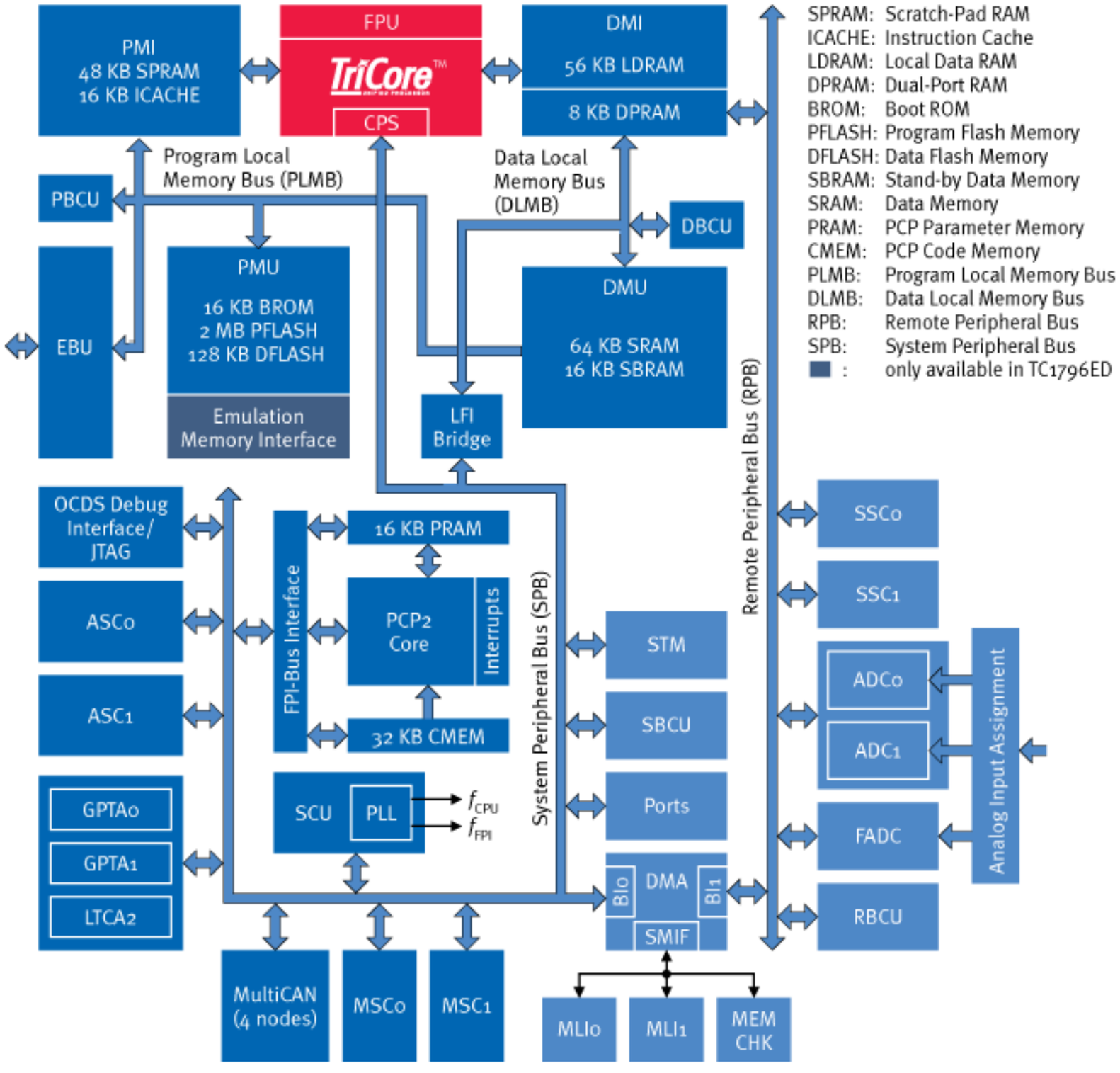
Introduction

TC1796 Features and Benefits

TC1796 Features	TC1796 Benefits
➢ 195 MIPS at 150MHz	➢ Debug with fast, industry-standard PC interface. Highest flexibility for gateway or field bus applications
➢ Peripheral Control Processor (PCP)	➢ Dedicated fast interrupt controller that offloads the CPU
➢ 2MB FLASH + 208kB SRAM	➢ Large internal memory
➢ 2 ADC (16 Ch) + 1 FADC (4 Ch)	➢ 8/10/12-bit ADCs with 1.25µs@10bit conversion time 10-bit FADC with 280ns conversion time
➢ 2 General Purpose Timer Arrays (GPTA)	➢ Flexible autonomous module with digital signal filtering and timer functionality to realize complex I/O management
➢ External Bus Unit (EBU)	➢ Flexible memory setting with flash, burst flash, EPROM, SRAM
➢ 4 CAN nodes (CAN) + TTCAN	➢ Field bus to large number of actuator and sensor
➢ Free DSP library	➢ Take advantage of the inherited DSP capabilities. Save time and effort and increased time to market without extra cost.
➢ Real Time Operating Systems PXROS, OSE, Euros, µC/OS-II, ThreadX	➢ Wide choice of deeply embedded real-time operating system
➢ IEC61131.3 Soft-SPS: CoDeSys	➢ The industrial standard compliant system software for automation and control from the major system vendors

Introduction

TC1796 Block diagram



Introduction

Industrial TriCore Family Overview

Type	CPU [MHz]	MMU	PCP	RAM[kB]	Flash[MB]	IO-Lines	ADC channels	Timers /Counters	CCU	PWM	EBU	ASC	SSC	USB	MultiCAN	Fast Ethernet	Packaging
TC1100	150	✓	-	144	-	72	-	3xGPT	7	6	✓	2	2	-	-	-	LBGA-208
TC1115	150	✓	-	144	-	72	-	3xGPT	14	12	✓	3	2	-	✓	-	LBGA-208
TC1130	150	✓	-	144	-	72	-	3xGPT	14	12	✓	3	2	✓	✓	✓	LBGA-208
TC1161	66	-	-	48	1	81	36	GPTA+STM	64	96	-	2	1	-	-	-	LQFP-176
TC1162	66	-	-	48	1	81	36	GPTA+STM	64	96	-	2	1	-	✓	-	LQFP-176
TC1163	80	-	✓	64	1	81	36	GPTA+STM	64	96	-	2	2	-	-	-	LQFP-176
TC1164	80	-	✓	64	1	81	36	GPTA+STM	64	96	-	2	2	-	✓	-	LQFP-176
TC1165	80	-	✓	80	1.5	81	36	GPTA+STM	64	96	-	2	2	-	-	-	LQFP-176
TC1166	80	-	✓	80	1.5	81	36	GPTA+STM	64	96	-	2	2	-	✓	-	LQFP-176
TC1796	150	-	✓	208	2	123	44	2xGPTA+STM	112	184	✓	2	2	-	✓	-	BGA-416

Introduction

Web sources

Information

- TriCore Promotion http://www.infineon.com/tricore
- Tools and Software http://www.infineon.com/mc-tools

Freely available software

- Altium Tasking Evaluation Version http://www.tasking.com/tricore
- HighTec GNU + PXROS Evaluation Version http://www.hightec-rt.com/tricore.html
- DAVE http://www.infineon.com/dave
- DSP Library http://www.infineon.com/mc-tools > ‘Software Downloads’
- Stack Depth Analyzer http://www.infineon.com/mc-tools > ‘Software Downloads’

Introduction

Starter Kit

The Starter Kit has been packaged so that you are productive within 30 minutes from unpacking the kit.

Content:

- TC1796 Evaluation Board
- Logic Analyzer Extension Board
- Starter Kit CD
- Power plug for EUR/UK/US

Starter Kit CD

Documentation

- Getting started
- User Manuals
- Architectural Manuals
- Application Notes
- DSP Optimization Guide

Tools

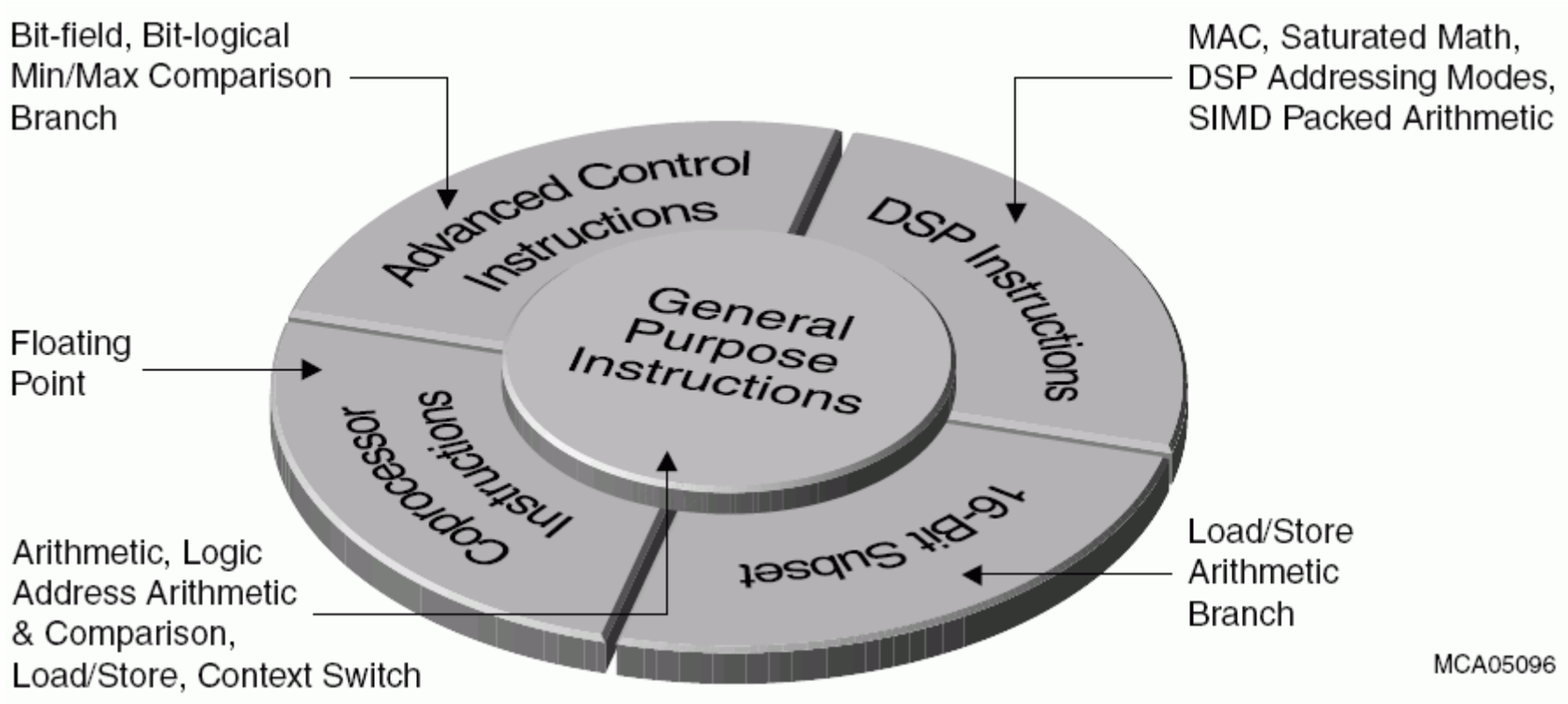
- Compiler: Tasking, GNU
- Debugger: PLS, Lauterbach, Hitex

Software

- RTOS: OSE, PXROS, Euros
- Libraries: TriLib

32-bit TriCore architecture

TriCore is the first combined 32-bit microcontroller/DSP architecture optimized for integrated real-time systems. It combines the outstanding characteristics of three different areas: the signal processing of DSP, real-time microcontroller, and RISC processing power; and it permits the implementation of RISC load-store architectures. It offers an ideal price/performance relationship, because a system requires fewer modules because more functions are integrated on the chip. This family of controllers is equipped with an optimal range of powerful peripherals for a wide spectrum of applications in power train, safety, and vehicle dynamics, driver information and entertainment electronics, and for body and convenience applications.
The instruction set architecture (ISA) supports a global linear 32-bit address space with memory-oriented I/O. The operation of the core is superscalar, i.e. it can execute simultaneously up to three instructions with up to four operations. Furthermore, the ISA can work in conjunction with different system architectures, also with multi-processing architectures. This flexibility at the implementation and system level permits different cost/performance combinations to be created whenever required.



32-bit TriCore architecture

TriCore contains a mixed 16-bit and 32- bit instruction set. Instructions with different instruction lengths can be used alongside each other without changing the operating mode. This substantially reduces the volume of code, so that even faster execution is combined with a reduction in the memory space requirement, system costs and energy consumption. The real-time capability is essentially determined by the interrupt wait and context switching times. Here, the high-performance architecture reduces response times to a minimum by avoiding long multi-cycle instructions and by providing a flexible hardware-supported interrupt scheme. In addition, the architecture supports rapid context switching.
Detailed information about the TriCore architecture with the complete instruction set is contained in the “TriCore Architecture Manual”.

Overview of the features of TriCore architecture

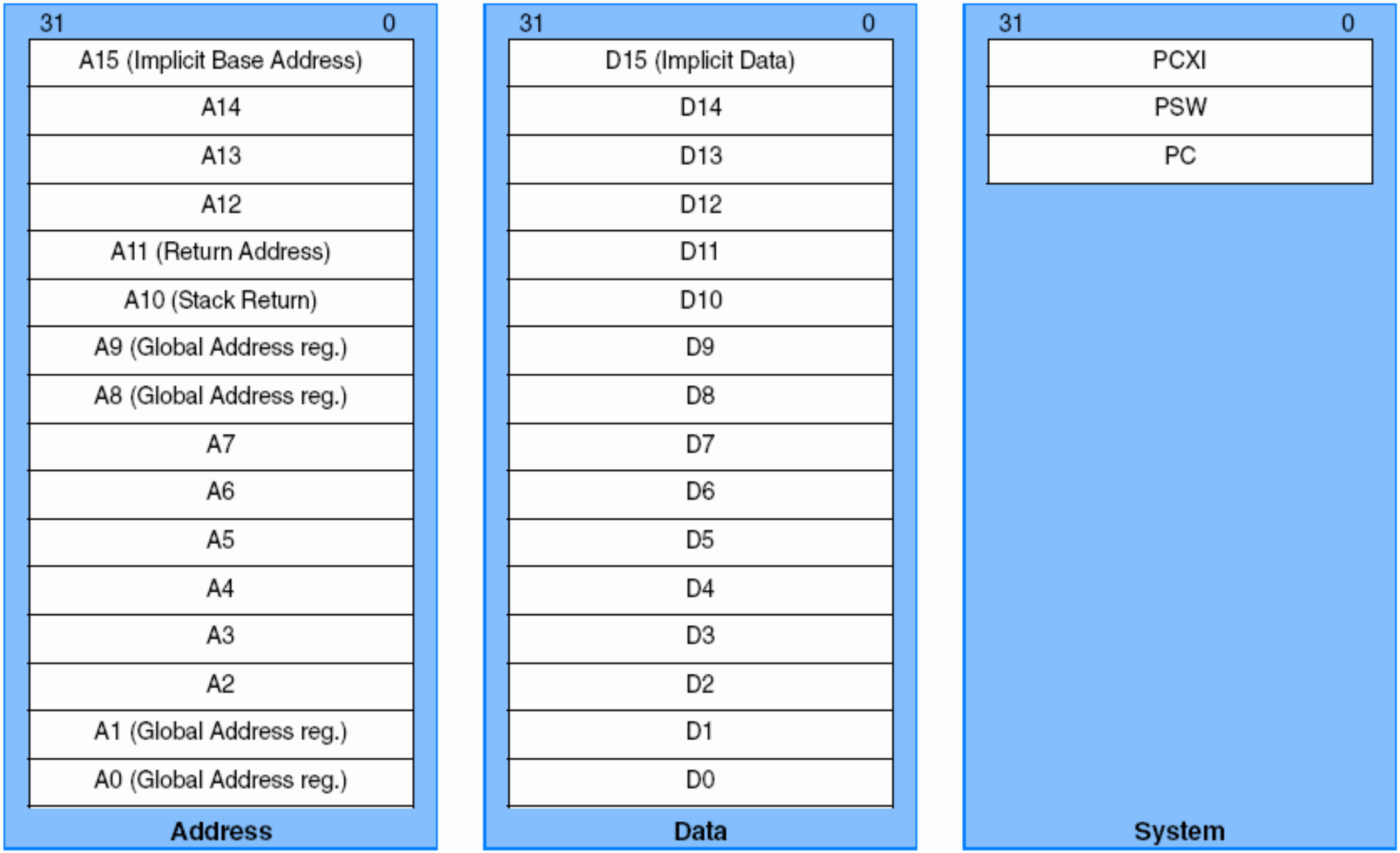
The list below summarizes the basic features of the TriCore architecture:

- 32-bit architecture,
- unified 4-Gbyte data, program, and input/output address space,
- 16-bit/32-bit instructions to reduce code volume,
- low interrupt response times,
- fast, automatic HW context switching,
- multiplication-accumulation unit,
- saturation integer arithmetic,
- bit-operations and bit addressing supported by the architecture and instruction set,
- packed data operations (single instruction multiple data, SIMD),
- zero overhead loop for DSP applications,
- flexible power management,
- byte and bit addressing,
- little endian byte order,
- support for big and little endian byte ordering on the bus interface,
- precise exception states,
- flexible, configurable interrupt management with up to 256 levels.

32-bit TriCore architecture

Program status registers

The TriCore register sets consist of 32 general purpose registers (GPRs), two 32-bit registers with program status information (PCXI and PSW) plus one program counter (PC). PCXI, P8W, and PC are core special function registers (CSPRs).
The 32 general purpose registers ate subdivided into sixteen 32-bit data registers (D0 to D15) and sixteen 32-bit address registers (A0 to A15). Four GPRs perform special functions: D15 serves as an implicit data register, A10 is the stack



pointer (SP), A11 the return address register, and A15 the implicit base address register.
Registers A0 and A1 in the lower address registers, together with A8 and A9 in the upper address registers, are defined as system global registers. These registers are not included in any context partition and are not automatically saved or restored when there is a HW context switch. The operating system uses them, for example, to reduce the system overhead.
The PCXI and PSW registers contain status flags, information about instructions which have been executed, plus protection information.
Important: the register database is split into a lower and an upper context, when there is an automatic switch of one half: saved in a HW-controlled double chained list

Data types

The TriCore instruction set supports boolean operations, bit sequences, characters, fixed point, addresses, signed and unsigned integers, and single-precision floating point numbers.
Most of the instructions process specific data types, while there are others which are suitable for manipulating various data types.

Addressing modes

The addressing modes enable load and store instructions to effect efficient access to simple data elements within data structures such as records, arrays with direct or sequential access, stacks and circular buffers. Simple data elements have a width of 1,8, 16, 32, or 64 bits.
The addressing modes also ensure the efficient compilation of C, easy access to peripheral registers or the efficient implementation of standard DSP data structures (DSP addressing modes such as circular buffers for filters, and bit-reverse addressing for FFT’s). The following seven addressing modes are supported by the TriCore architecture:
- Absolute
- Base + short offset
- Base + long offset
- Pre-increment or pit-decrement
- Post-increment or post-decrement
- Circular
- Bit-reverse

Instruction formats

The TriCore architecture supports both 16-bit and 32-bit instruction formats. All instructions have a 32-bit format. The 16-bit instructions form a subset of them, chosen because of the frequency with which they occur, to reduce the volume of code. Instructions are selected by the compiler, and can be used in parallel alongside one another with no mode change etc.

Tasks and contexts

In this book, the term “task” is used for an independent control procedure. Two types of task must be distinguished: soft-wait-managed user tasks (SMTs) and interrupt service routines (ISRs). Software- managed tasks are produced by the services of a real-tine kernel or operating system, and are chosen for processing under the control of scheduling software. ISRs are chosen for processing by the hardware, as a response to an interrupt. In this architecture, TSR refers only to the code which is called directly by the hardware. Software managed tasks are sometimes referred to as user tasks, on the basis that they are executed in user mode.
Each task can be assigned its own authorization level. These individual rights are mainly enabled/disabled by I/O mode bits in the program status word (PSW). Associated with each task is a set of state elements, known collectively as the task’s context. The term context is used for everything which the processor requires in order to determine the state of the corresponding task and to enable its further execution. This includes the CPU general registers used by the task, the task’s program counter (PC), and its program status information (PCXI and PSW). The TriCore architecture exercises efficient management of the tasks’ contexts by means of the hardware.

Upper and lower contexts

The context is subdivided into the upper context and the lower context
The upper context comprises the upper address registers A10 - A15 and the upper data registers D8 to D15. These registers are designated as non-volatile for the purpose of function calls. The upper context also includes the PCXI and PSW registers.
The lower context comprises the lower address registers A2 to A7, the lower data registers D0 to D7 and the PC.
Both the upper and the lower context are associated with a LINK WORD. The contexts are stored in areas with a fixed size, which are linked together by the link word (see following section).
When an interrupt occurs, the upper context is automatically saved and is restored when the return occurs. If the interrupt service routine (ISR) needs to use more registers than are available in the upper context, then the lower context will be explicitly saved and restored by the ISR.

32-bit TriCore architecture

Contest save areas

The TriCore architecture makes use of linked lists of context save areas (CSAs) with fixed sizes, supporting systems with multiple, linked control threads. A CSA consists of 16 words of on-chip storage facilities, aligned to a 16-word boundary. One individual CSA can hold exactly one upper or lower context. CSAs which are not used are linked by an ‘unused’ list. They are assigned from this unused list as necessary and, when they are no longer required, are returned to the list. The processor hardware controls their assignment and release. They are transparent to the application code. Only the system start code and certain of the operating system’s exception handling routines need to explicitly access the CSA lists and the memory device.

Fast context switching

To increase its performance capability, the TriCore architecture has a uniform context switching mechanism for function calls, interrupts, and traps. In all cases, the upper context is automatically saved and retrieved by the hardware, with the saving and retrieval of the lower context being open to the new task as an option. As a result of the unique memory sub-system design of TriCore, which permits the transfer of up to 16 data words between the processor registers and memory, so the entire context can be saved in a single operation, fast context switching is speeded up even more.

Interrupt system

A service request can be defined as an interrupt request from a peripheral device, a DMA request or an external interrupt. For the sake of simplicity, a service request will be described simply as an interruption. The entry code for the ISR consists of a block within a vector of code blocks. Each code block represents the entry for an interrupt source. A priority number is assigned to each source. All the priority numbers are programmable. The service program uses the priority number to determine the memory location for the entry code block. This prioritization of the service programs permits nested interrupts. A service request can interrupt the processing of an interrupt with lower priority. Interrupt sources with the same priority cannot mutually interrupt each other.

Trap system

A trap is a special form of interrupt for error handling, and is initiated in the event of an exception which falls into one of the eight classes identified below:

- reset
- internal protection
- instruction errors

- context management
- internal bus and peripheral errors
- logically ’true’ signal state for L signal level
- system call
- non-maskable interrupt

The entry code for the trap processing routines comprises a vector of code blocks. Each code block contains the entry point address for one trap. When a trap is triggered, the trap’s identification number (TIN) is stored in data register D15. The trap processing routine uses this TIN to identify the precise reason for the trap. During arbitration the trap with the lowest TIN number takes precedence.

Protection system

The protection system gives the programmer the option to assign access permissions to memory regions, for both data and code. The ability can be used to protect the core system functionality from bugs that may have slipped through testing and from transient hardware faults. In addition, TriCore’s protection system contains the critical features for isolating errors, thus facilitating debugging.

Permission levels

TriCore’s embedded architecture allows each task to be assigned the specific permission level it requires to perform its function. Individual permissions are enabled by means of the I/O mode bits in the program status word (PSW). The three authorization levels am called User-0, User-1, and Supervisor:

- User-O mode is used for tasks which do not access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts.
- User-1 mode is used for tasks which access common unprotected peripheral devices. These accesses usually include read/write accesses to SIO ports and read accesses to timers and most of the I/O status registers. Tasks at this level can disable interrupts.
- Supervisor mode permits read/write accesses to the system registers and all peripheral devices. Tasks at this level can disable interrupts.

Protection model

The memory protection model for the TriCore architecture is based on address ranges, with each address range having its own permission setting. The address ranges and the associated permissions are defined in two to four identical sets of

tables, which are stored in the core SFR (CSFR) space. Each set is referred to as a protection register set (PRS). When the protection system is active, TriCore checks the legality of every load, store, or instruction fetch address before performing the access. In order to be legal, the address must fall within one of the ranges specified in the currently selected PRS and permission for that type of access concerned must be available in the matching range.

Reset system

Various events can force a reset of the TriCore device:
- Power-on reset triggered through an external pin when the power supply for the device is switched on (cold start).
- Hard reset: triggered through an external pin during operation (warm start).
- Soft reset triggered by a software write into a reset request register. This register has a special protection mechanism to prevent unintended accesses. Implementation-specific controls in this register effect either a partial or a complete reset of the device.
- Watchdog timer reset triggered by an error state recognized by a watchdog timer.
- Wake-up reset triggered through an external pin when the device is reactivated from the energy-saving mode.

The core can check, using a reset status register, which of the various triggers has invoked the reset.

Debugging system

TriCore contains mechanisms and resources to support on-chip debugging. which are used by the Debug Control Unit, a module which is located outside the core. Most of the functions and details of the Debug Control Unit are implementation-specific. For this reason, no further description of the unit and its registers is included here. The details will be found in the documentation for the products concerned.

Programming model

This section discusses the following aspects of the TriCore architecture which are of relevance for the software: the data types supported, the formats of the data types in registers and memories, the various addressing modes provided by the architecture, and the memory model.

Data types

The TriCore instruction set supports boolean operations, bit strings, characters, signed fractions, addresses, signed and unsigned integers, and single-precision floating point numbers. Most of the instructions process specific data types, while there are others which are suitable for manipulating various data types.

32-bit TriCore architecture

Boolean expressions

A boolean expression is either TRUE or FALSE. TRUE corresponds to a value of one (1) when it is computed, or non-zero when it is checked; FALSE corresponds to a value of zero (0). Boolean expressions are generated as the result of comparison and logical instructions, and are used as source operands in logical and conditional jump instructions.

Bit strings

A bit string is a packed bit field. Bit strings are generated and used by logical, shift, and bit field instructions.

Characters

A character is an eight-bit value which corresponds to a very short unsigned integer. It does not assume any specific coding.

Signed fractions

The TriCore architecture supports signed fractional 16-bit data for DSP arithmetic. Data values in this format have a single leading sign bit with a value of 0 or -1, followed by an implicit binary point and a fraction. Their values thus lie in the range [-1,1). When they are saved into registers, 16-bit fractional data occupies the 16 most significant bits, and the 16 least significant bits are set to zero.

Addresses

An address is a 32-bit unsigned value.

Signed/unsigned integers

Signed and unsigned integers normally comprise 32 bits. When smaller signed and unsigned integers are loaded from memory into a register they are extended to 32 bits with a sign or zeros. Multiple precision integers are supported by addition and subtraction with carry. During move and masking operations, integers are regarded as bit strings. Multi-precision shifts can be done using a combination of single precision shifts and bit field extracts.

Single precision IEEE-754 floating point numbers

Depending on the particular implementation of the core architecture, IEEE-754 floating point numbers are supported by direct hardware instructions or software emulation.

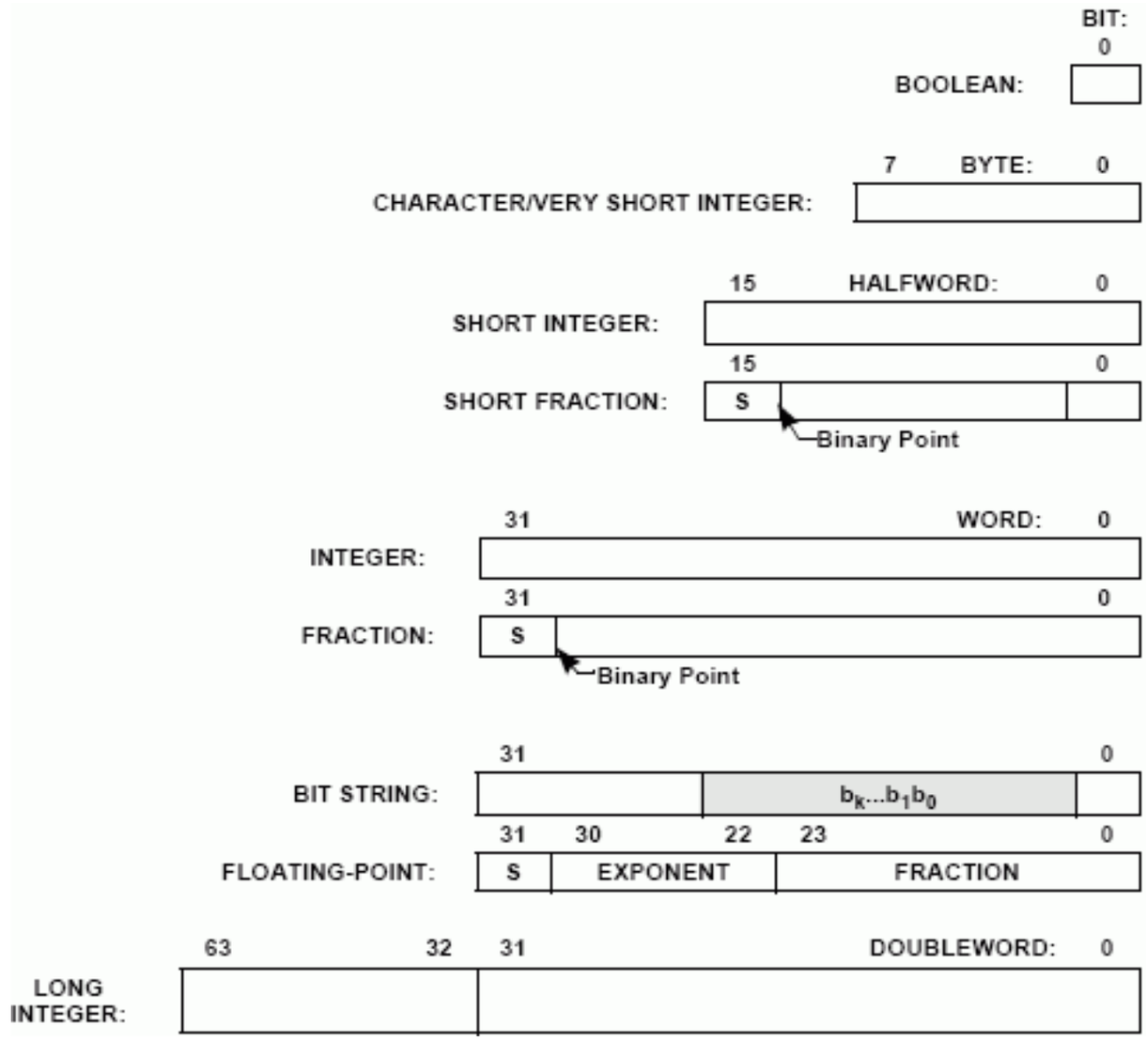
32-bit TriCore architecture

Data formats

All the general purpose registers have a width of 32 bits, and most of the instructions operate on word values (32 bits). If data containing less bits than a word are read out of memory they must be sign-extended or zero-extended before any operations can be applied to the whole word. The required alignments are different for addresses and data. To permit transfers between address registers and memory, addresses (32-bit) must be aligned to a word boundary. For transfers between data registers and memory, a data item can be aligned to any arbitrary half-word boundary, regardless of its size; bytes can be accessed using any valid byte address. The figure shows the data formats which are supported.

The data memory and the CPU registers save data in 'little endian' byte order (i.e. the least significant bytes are stored in the lower addresses).

If the TriCore system is connected to an external big endian device, the bus interface carries out the translation between the big and little endian formats. As already mentioned, bytes must be saved on byte boundaries, and half-words, words and double-words on half-word boundaries.



The memory model

The TriCore architecture permits up to 4 Gbytes of memory to be accessed. The address width is 32 bits. The address space is divided into 16 regions or segments (0 to 15). Each segment comprises 256 Mbytes.

The upper four bits of an address are used for selecting a specific segment The first 16 Kbytes of each segment are accessed using either absolute addressing or absolute bit addressing.

Speculative read accesses are not supported for segments 14 and 15. Accesses to this memory area are only initiated if the core is sure that the access will be successfully completed. These segments are used for peripheral registers (PSFRs)

or for external peripheral devices. FIFOs, peripheral devices with status registers and other devices should be arranged in this address segment, so that they cannot be the subject of speculative read operations which could result in the deletion of data. Accesses cannot be made in the User-0 mode.
Segments 0 to 7 are reserved. Any access to them triggers a trap. For segments 8 to 13 there may be further restrictions in the case of certain products; these are noted in the appropriate product documentation.
Many data accesses are made using addresses which are formed by the addition of an offset value to the contents of a base address register. In these cases, however, the offset value used must not produce an address beyond the segment boundary, as this would trigger a trap. This restriction means that the base address can be used at any tine to determine which segment may be accessed.
The core special function registers (CSFRs) are arranged in one 64-Kbyte area. The base address of this area is product-specific, and is stated in the appropriate product documentation.

Addressing model

Apart from the addressing modes used in the instruction set, the TriCore architecture allows extended addressing modes to be realized using short instruction sequences.

Addressing mode	Address register used	Offset width (bits)
Absolute	None	18
Base + offset	Address register	10/16 (short/long)
Post-op change	Address register	10
Pre-op change	Address register	10
Circular addressing	Address register pair	10
Bit reversed	Address register pair	-

Built-in addressing modes

These addressing modes allow load and store instructions to efficiently access simple data elements with data structures such as records, arrays with direct or sequential access, stack memories, and circular buffers. Simple data elements have a width of 1, 8, 16, 32, or 64 bits. The table gives an overview of the addressing modes which are supported.

32-bit TriCore architecture

The addressing modes have been chosen to permit the efficient compilation of C, easy access to peripheral registers and the efficient implementation of typical DSP data structures (circular buffers for filters, and bit-reversed indexing for FFTs). The instruction formats have been chosen so that they provide the greatest possible space for direct addresses, or offsets in the case of indirect addressing.

Absolute addressing

Absolute addressing is suitable for addressing peripheral registers or global data. It uses an 18-bit constant for the memory address, this being specified in the instruction itself. The complete 32-bit address is formed by using the top 4 bits of the 18-bit constant as the top 4 bits of the 32-hit address, the lower 14 bits being directly copied into the address, and inserting 14 zero-bits between them.

Base+ offset addressing

Addressing by means of a base + offset is useful for accesses to structural elements, local variables (with the stack pointer as the base), and static data (with an address register as the base pointer). Here, the final address is the sum of an address register and an offset value with sign extension. The width of the offset is 10 bits, or for some instructions 16 bits. This allows any required memory location to be addressed.

Addressing with post-modification

This mode of addressing uses the value of an address register as the final address and, after the access, changes the address by the addition of a signed 10-bit offset. The sign can be chosen to move the pointer forward or backward. Both options can be used, for example, for sequential accesses to arrays or to remove (POP) data from a stack memory.

Addressing with pre-modification

This mode of addressing adds a signed 10-bit offset to the value of an address register and uses the result as the final address. The address register is overwritten with the sum which has been formed. The sign can be chosen to move the pointer forward or backward. Both options can be used, for example, for sequential accesses to arrays or to store (PUSH) data into a stack memory.

Circular addressing

The main application of circular addressing is to access data in a circular buffer during filter calculations. In the case of circular addressing, the current state is stored in an address register pair. The even-numbered register contains the base address, the upper half of the odd-numbered register contains the buffer size, and the lower half the buffer index. Here, the final address is the sum of the base address and the index.

After the access has been made, the index is modified by a signed 10-bit offset (contained in the instruction). Provided that this offset is less than the buffer size, the index will automatically wrap-round from one end of the buffer to the other. For example, if the buffer has a size of 50 and the index of 48 is increased by an offset of 4, the index will be given the value 2 (48 + 4 - 50).
A circular buffer is subject to the following restrictions:
- the start of a circular buffer must be aligned to a 64-bit boundary
- the buffer size must be a multiple of the data size, which is implicitly determined by the access instruction. For example, when using a "load word" instruction for the access, the buffer size must be a multiple of 4 bytes, and with "load double word" instructions a multiple of 8 bytes.

Bit-reverse addressing

This mode of addressing is used for FFT algorithms, because with the usual implementations of the FFT the results appear in bit-reversed sequence. For bit-reversed addressing, an address register pair stores the current status. The even-numbered register contains the base address, the lower half of the odd-numbered register contains the array index, and the upper half the modification value. Here, the final address is the sum of the base address and the index. After each access, the modification value is added to the reversed index and the result is again reversed. The result of this with a modification value of 1024, for example, is the following sequence of indices: 0, 1024, 512, 1536, 256, etc.
As a rule, the modification value represents the reversed value of half the array size.
As bit reversal does not represent a bit field operation, and is therefore only programmed with difficulty, this addressing mode permits simple programming and rapid processing.

Extended addressing modes

Special addressing modes which are not directly supported by the architecture can be implemented by short instruction sequences.

Indexed addressing

Using the ADDSC.A instructing, which adds a scaled value to an address register, indexed addressing of byte, half-word, ward, or double-word arrays can be realized (scaling by 1, 2,4 or 8).
To allow indexed addressing of bit fields the instruction ADDSC.AT determines the word in which the required bit or bit field is located. Bits are extracted using the EXTR.U instruction, and are stored using the instruction LDMST (load I modify I store).

32-bit TriCore architecture

PC-relative addressing

For branching and subroutine calls, it is usual to use PC-relative addressing. However, as this mode of addressing for data accesses would reduce the performance of the memory system, the TriCore architecture does not directly support such data access. If PC-relative data access is required, the address of a neighboring label can be loaded into an address register and used as the base address.
In the case of code which is loaded dynamically, the current value of the PC can be determined with the instruction IL (jump and link), which stores the address of the subsequent location in register A11. The return address for the current routine must be saved beforehand.

Extended absolute addressing

Extended absolute addressing is effected by combining two instructions. The instruction LEA (load effective address) loads a 32-bit address into an address register. After execution of the instruction MOVH.A (move high word), the data is addressed using the base address + 16-bit offset.

Core registers

In the TriCore architecture, a set of core special function registers (CSFRs) is defined. These CSFRs control the operation of the core, and provide status information about the core’s operation. The CSFRs are split into the following groups:

- Program state information
- Stack management
- Context management
- Interrupt and trap control
- System control
- Memory protection
- Debug control

32-bit TriCore architecture

The sections below present a summary of these registers. The CSFRs are supplemented by a set of general purpose regis-ten (GPRs). The table includes all the CSFRs and GPRs.

Register name	Description
D0-D15	Data registers
A0-A15	Address registers
PSW	Program status word
PCXI	Previous context information
PC	Program counter
FCX	Pointer to first free location in list
LCX	Pointer to last free location in list
ISP	Interrupt stack pointer
ICR	Interrupt control register
BIV	Base address of the interrupt vector table
BTV	Base address of the trap vector table

Register name	Description
SYSCON	System configuration register
DPRx_0 - DPRx_3	Data segment protection register sets (x=0-3)
CPRx_0 - CPRx_3	Code segment protection register sets (x=0-3)
DPMx_0 - DPMx_3	Data protection mode sets (x=0-3)
CPMx_0 - CPMx_3	Code protection mode sets (x=0-3)
DBGSR	Debug status register
EXEVT	External break input specifier
SWEVT	Software break input specifier
CREVT	CSFR access event specifier
TRnEVT	Specifier for trigger event n (n=0,1)

Accessing the core registers

The core uses the two instructions MFCR and MTCR to access the CSFRs. The instruction MFCR (move from core register) moves the contents of the CSFR which is addressed into a data register MFCR can be executed on any privilege level. The instruction MTCR (move to core register) moves the contents of a data register into the CSFR which is addressed. To pre-vent unauthorized write accesses to the CSFRs, the MTCR instruction can only be executed at the Supervisor privilege level.

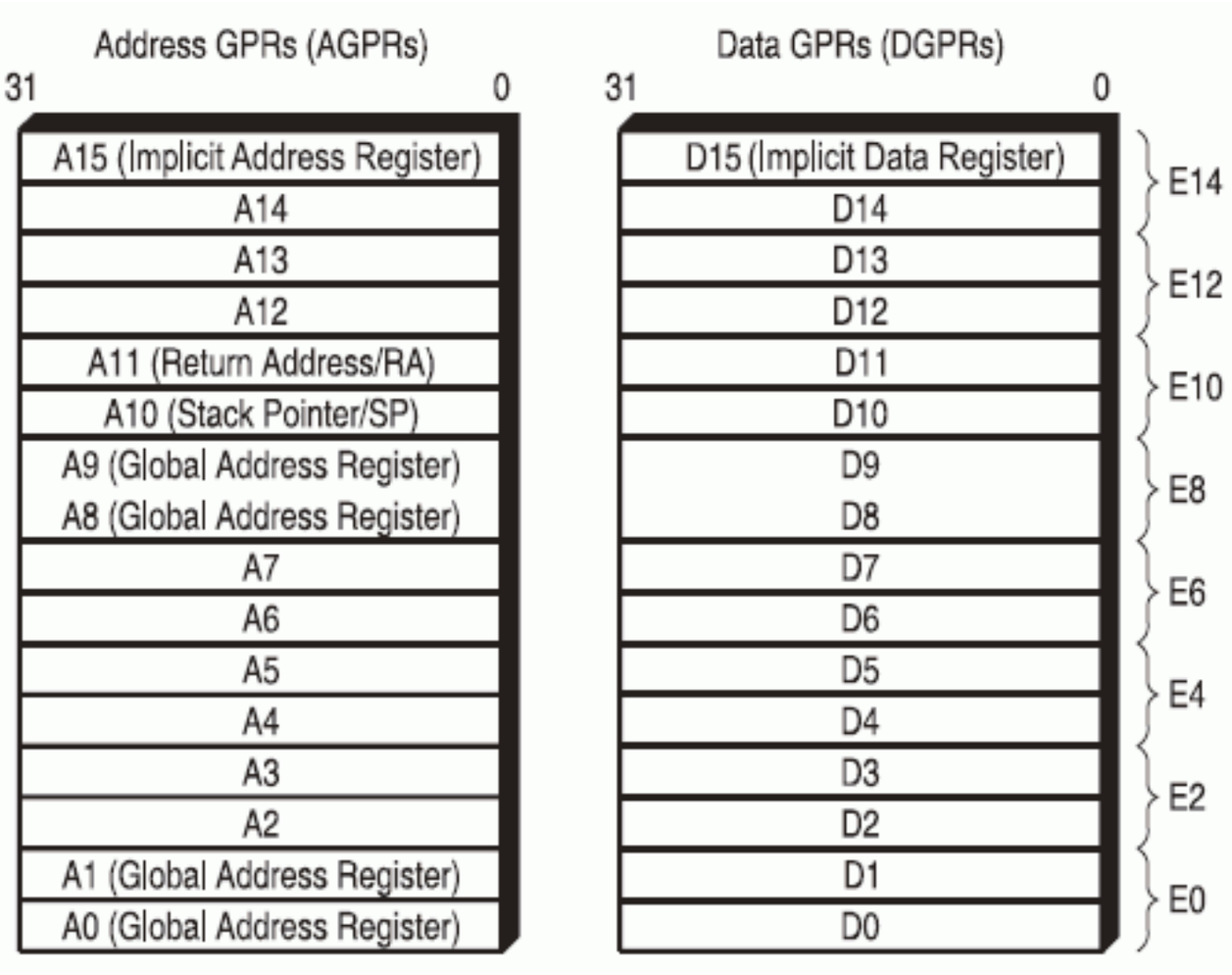
The CSFRs are also mapped into the top of the local code segment in the memory address space. This assignment makes the complete architectural state of the core visible in the address map. This feature ensures efficient support for the debugger and the emulator.

32-bit TriCore architecture

Note: the core may not use this mechanism to access the CSFRs, but must use the instructions MFCR and M'TCR for that purpose.
There are no instruction which allow bit, bit field, or load-modify-store accesses to the CSFRs. The instruction RSTV (reset overflow flags) only resets the overflow flags in the P5W, without changing any other P8W bits. This instruction can be executed at any privilege level.

General purpose registers (GPRs)

The figure shows the general purpose registers. The 32-bit general purpose registers are split evenly into 16 data registers or DGPRs (D0 to D15) and 16 address registers or AGPRs (A0 to A15). The separation of data and address registers facilitates efficient implementations, in which arithmetic and memory operations are performed in parallel. A range of instructions can be used for the interchange of information between data and address registers, in order to generate or derive table indices etc. Two consecutive data registers can be concatenated to form eight extended size registers (E0, E2, E4, E6, E8, E10, E12 and E14), in order to support 64-hit values.
Registers A0, A1, A8, and A9 are defined as system global registers. Their contents are not saved and restored when calls, traps, or interrupts occur. Register A10 is used as the stack pointer (SP); Register A11 is used for saving the return address (RA) for calls and linked jumps, and for saving the return program counter (PC) value during interrupts and traps.



While the 32-bit instructions have unrestricted access to the GPRs, many of the 16-bit instructions implicitly use A15 as their address register and D15 as their data register. This implicit use facilitates the coding of these instructions in 16 bits. To support 64-bit data, these values are stored in an even/odd register pair. In Assembler syntax, these register

pairs are designated either as one pair of 32-bit registers (for example D9/D8) or as one extended 64-bit register (thus, for example, E8 is the concatenation of D9 and D8, where D8 represents the less significant word of E8).
It should be noted that there are no separate floating point registers, but that the data registers are used in performing floating point operations. The saving and retrieval of floating point data is effected automatically, making use of the fast context switching. The GPRs represent an important element in the context of a task. When the context for a task is being saved to or restored from memory, the context is divided into an upper and a lower context. Registers A2 to A7 and D0 to D7 are part of the lower context. Registers A10 to A15 and D8 to D15 belong to the upper context.

Registers for program state information

The PC, PSW, and PCXI registers store and reflect the program state information. When the context for a task is being saved or restored, the contents of these registers form an important part of the procedure, and are saved/restored or modified during this process.
The program counter (PC) contains the address of the instruction which is currently being executed.
The five most significant bits of the PSW contain ALU status flags, which are set or cleared by arithmetic instructions. The remaining bits of the PSW control the permission levels, the protection register sets, and the call depth counter. The PCXI register contains the link to the previous execution context and supports fast interrupts and automatic context switching.

Context management registers

The context management registers comprise three pointers: FCX, PCX and LCX. These pointers handle context management and axe used during the operations to save and restore the context.
Each pointer consists of two fields: a 16- bit offset and a 4-bit segment specifier. A context save area (CSA) is an address range which contains 16 word locations (64 bytes), which is the memory space required to save one upper or one lower context. By incrementing the offset value by one, the effective address is always incremented to the address 16 word locations above the previous address (offset moved 6 bits to the left). The total usable area in each address segment for CSAs amounts to 4 Mbytes, which gives memory space for 64 K of context save areas.
The effective address must point to memory which exists. Otherwise the system behavior is undefined.
The FCX pointer register contains the pointer to the start of the free locations list which always points to an available CSA.

The previous context pointer (PCX) stores the address of the CSA for the previous task. PCX is part of the PCXI register.

The LCX pointer register contains the pointer to the end of the free locations list, and is used to recognize impending CSA list range underflows. If the value of FCX resulting after for an interrupt or a CALL corresponds to the limit value then,

although the context save operation will be completed, the destination address written into the trap vector address will be forced to that for the emptying of the CSA list.

Stack management register

Stack management in the TriCore architecture supports a user stack and an interrupt stack. The management of the stack involves the address register A10, the interrupt stack pointer (ISP), and one PSW bit. The general purpose address register A10 is used as the stack pointer. The initial contents of this register are usually defined by an operating system when a task is generated, where individual tasks can be assigned a private stack area. The interrupt stack pointer (ISP) prevents the interrupt service routines (ISRs) from accessing the private stack areas and possibly interfering with the context of the software-administered tasks. The TriCore architecture implements an automatic switch to the use of the interrupt stack pointer instead of the private stack pointer.

Interrupt and trap control registers

Three CSFRs support interrupt and trap handling: the interrupt control register (ICR), the interrupt vector table pointer (BIV), and the trap vector table pointer (BTV).

The interrupt control register (ICR) contains the current CPU priority number (CCPN), the enable/disable bit for the interrupt system, the pending interrupt priority number (PIPN), and an implementation-specific control for the interrupt arbitration schema. The other two registers contain the base addresses for the interrupt (BIV) and trap vector tables (BTV). When an interrupt is accepted, or when a trap occurs, the pointer to the address in the interrupt/trap vector table is formed by shifting the value of the priority number / trap class 5 bits to the left and then ORing it with the value of the BIV/BTV register. The shift to the left results in a step spacing of 8 words (32 bytes) between the individual entries in the vector tables. The base addresses must be carefully aligned. On the one hand, the addresses must represent even byte addresses (half-word addresses) and on the other hand the base addresses must be aligned to a boundary which is a power of two, because of the simple ORing with the shifted priority values. The appropriate power of two is determined by the number of entries used. For example, the full extent of 256 interrupt entries requires alignment to an 8-Kbyte boundary. For the 8 trap classes (0 to 7), alignment to a 256 byte boundary is sufficient.

System control registers

Three registers provide system control:
the system configuration control register (SYSCON), the local program memory unit control register (PMUCON), and the local data memory unit control register (DMUCON).

Memory protection registers

The TriCore architecture incorporates hardware mechanisms which protect memory areas specified by the user from unauthorized accesses by read, write, or call instructions. Furthermore, the protection hardware can be used to generate signals to the debug unit. TriCore includes register sets in which the address ranges and access rights for a series of memory areas are specified. There are separate register sets for code and data memory.
The 2-bit PRS field in the PSW allows up to four such register sets to be selected in each case (four for data and four for code). The number of register sets provided for memory protection is defined specifically for each implementation of the TriCore architecture.

Data and code segment protection registers

The register pairs DPRx_n/CPRx_n comprise the two-word registers which define the lower and upper boundary addresses of the corresponding memory area. If the lower boundary is greater than the upper boundary, then no range checks will be carried out. If the lower boundary is the same as the upper boundary, then the area is considered to be empty.
When debug signals are being generated, the values in DPRx_n/CPRx_n are regarded as individual addresses, instead of defining a range. Signals are then generated to the debug unit if the address of a memory access corresponds with the contents of one or more of DPRx_n/CPRx_n. For this purpose, a comparison is made with the contents of the upper boundary register.
The 8-bit data/code protection mode registers determine the access permissions and debug signal generation for the data/code protection areas defined by the respective registers.

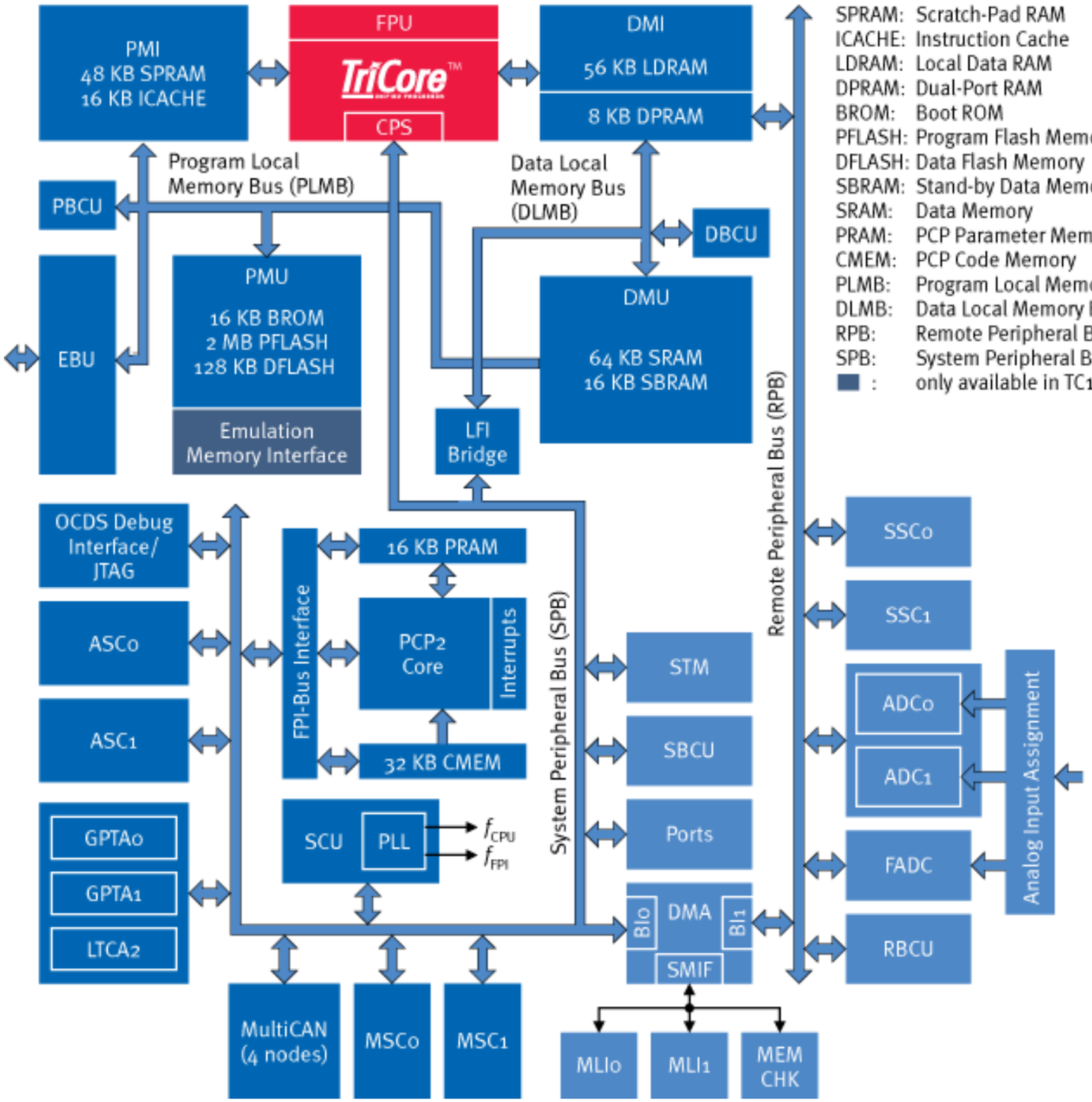
Debug registers

Seven registers have been implemented to support debugging. These registers define the conditions under which a debug event is generated, or what actions are to be initiated when a debug event occurs, and they also supply information about the state of the debug unit. The precise functions of the debug unit depend on the particular implementation. A description of the functions and the associated registers will be found in the appropriate product descriptions.

Block diagrams of 32-bit microcontrollers

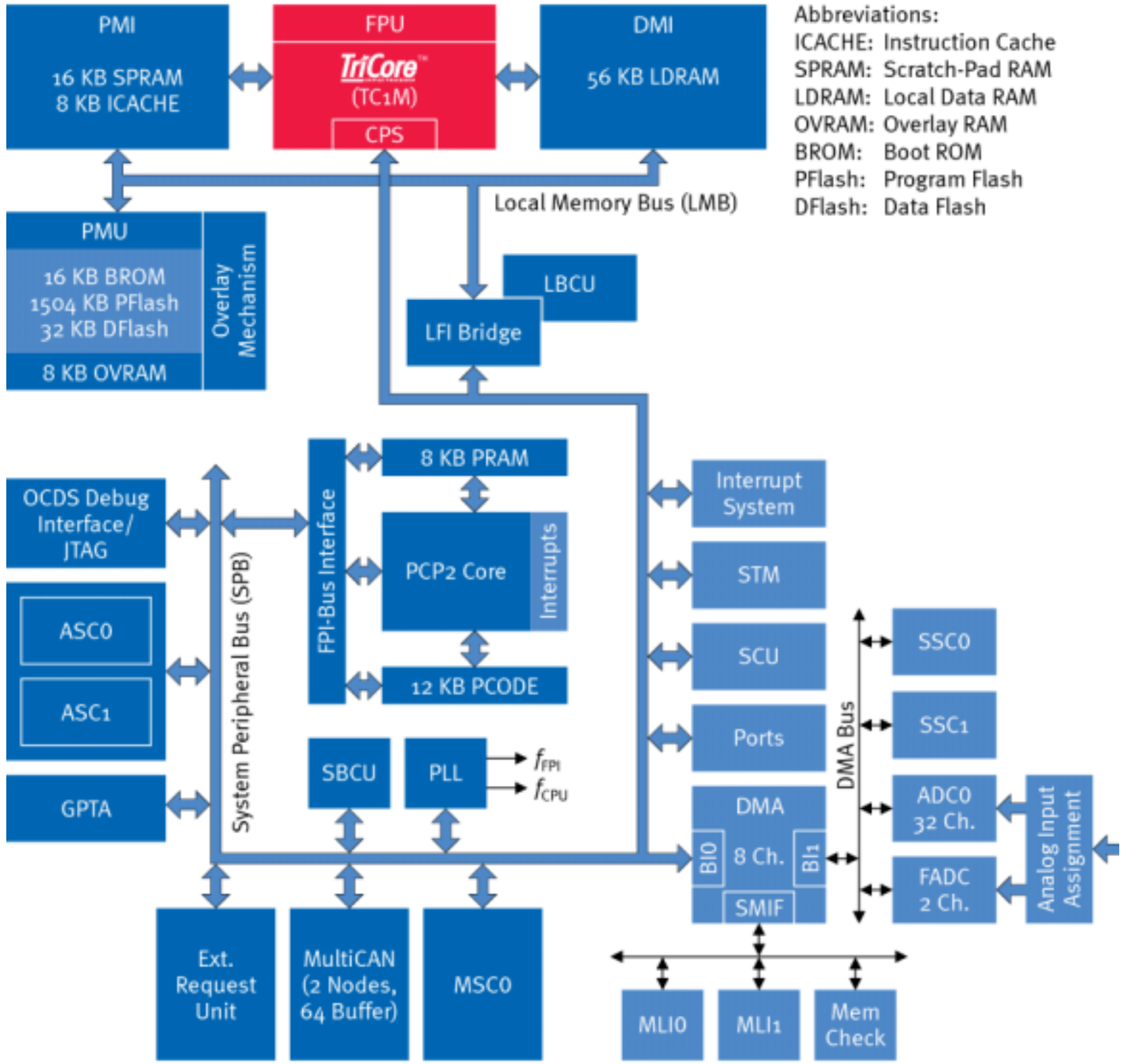
The following figures show block diagrams of three selected 32-bit microcontrollers.

32-bit TriCore architecture



TC1796 functional units

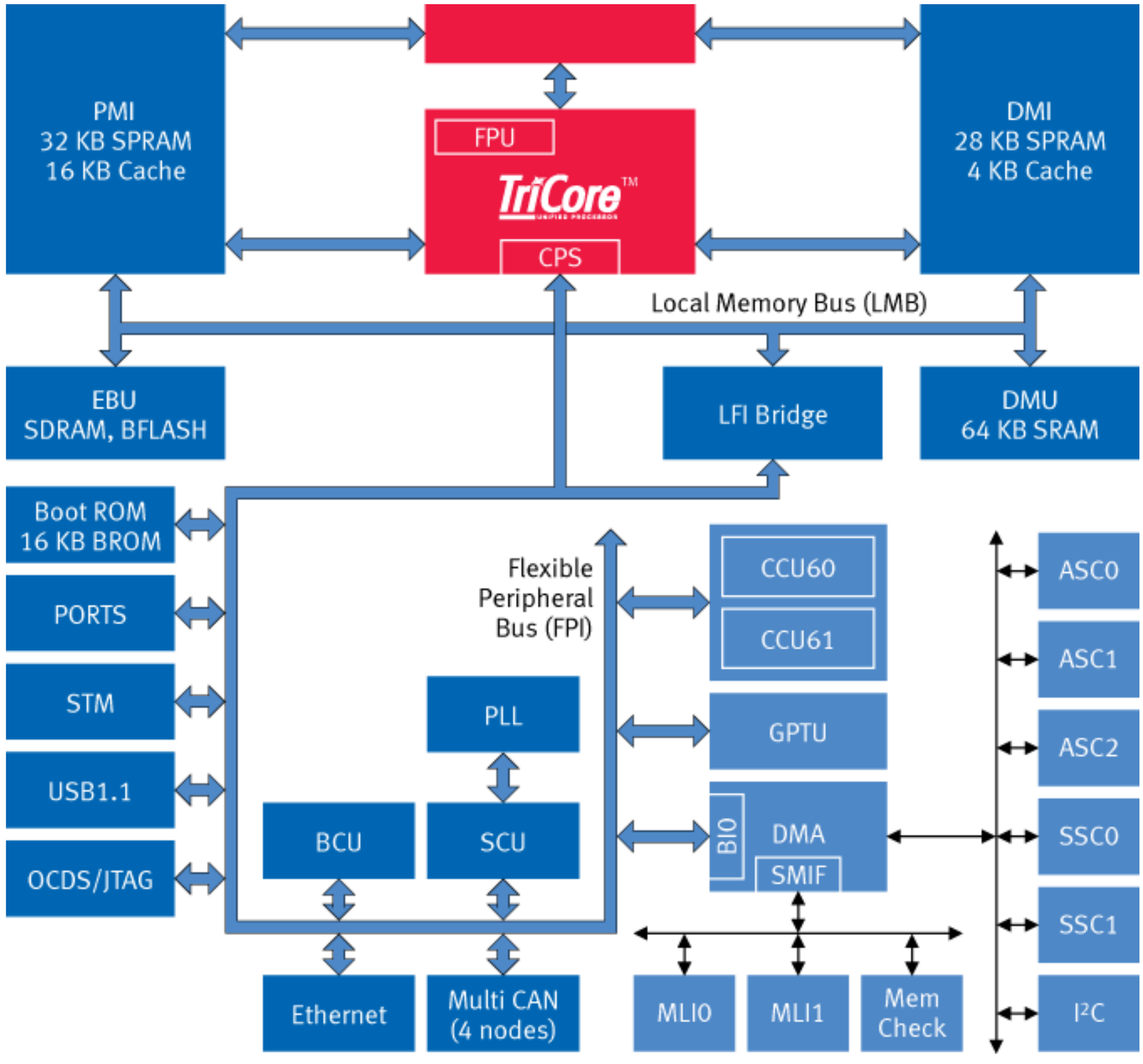
32-bit TriCore architecture



MultiCAN not available in TC1165

TC1166 functional units

32-bit TriCore architecture



TC1130 functional units

Preparation

The 10 exercises of this training are a jump-start to program the Infineon TriCore. The exercises are intended to help you understand the intrinsics of hardware-based and microcontroller programming and to develop the necessary programming and debugging skills.
Starting with a basic set-up the exercises cover the main features of the architecture: instruction set, interrupts, peripherals, flash, pipeline.

hello, world	The classical ’hello, world’.
LED	Use DAvE to generate the on-board LED blink program.
Saturation	How to make use of the instruction set features in C.
Interrupts	Set up interrupt routines easily.
ASC	Configure an on-chip peripheral.
Flash	Program the flash.
PWM	Setup the GPTA for a Puls Width Modulation signal.
Pipeline	Optimize code for the super-scalar architecture.
CAN	Communicate via CAN.
PCP	A first step into the Peripheral Control Processor.

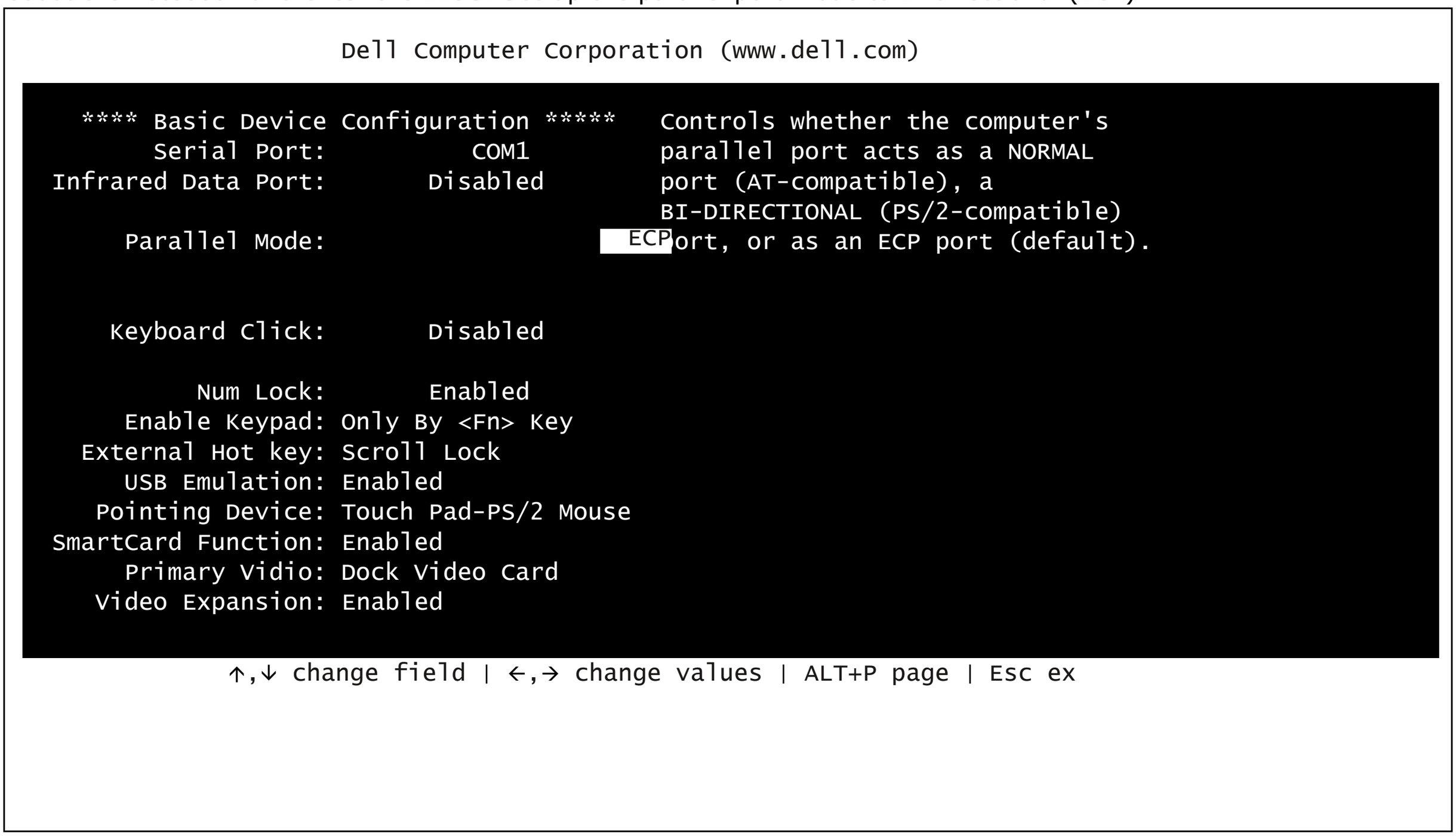
Preparation Steps

- Configure the parallel port in the BIOS of the development host
- Check the TriBoard
- Configure the TriBoard
- Connect the TriBoard
- Install the compiler, debugger and DAvE.

Preparation

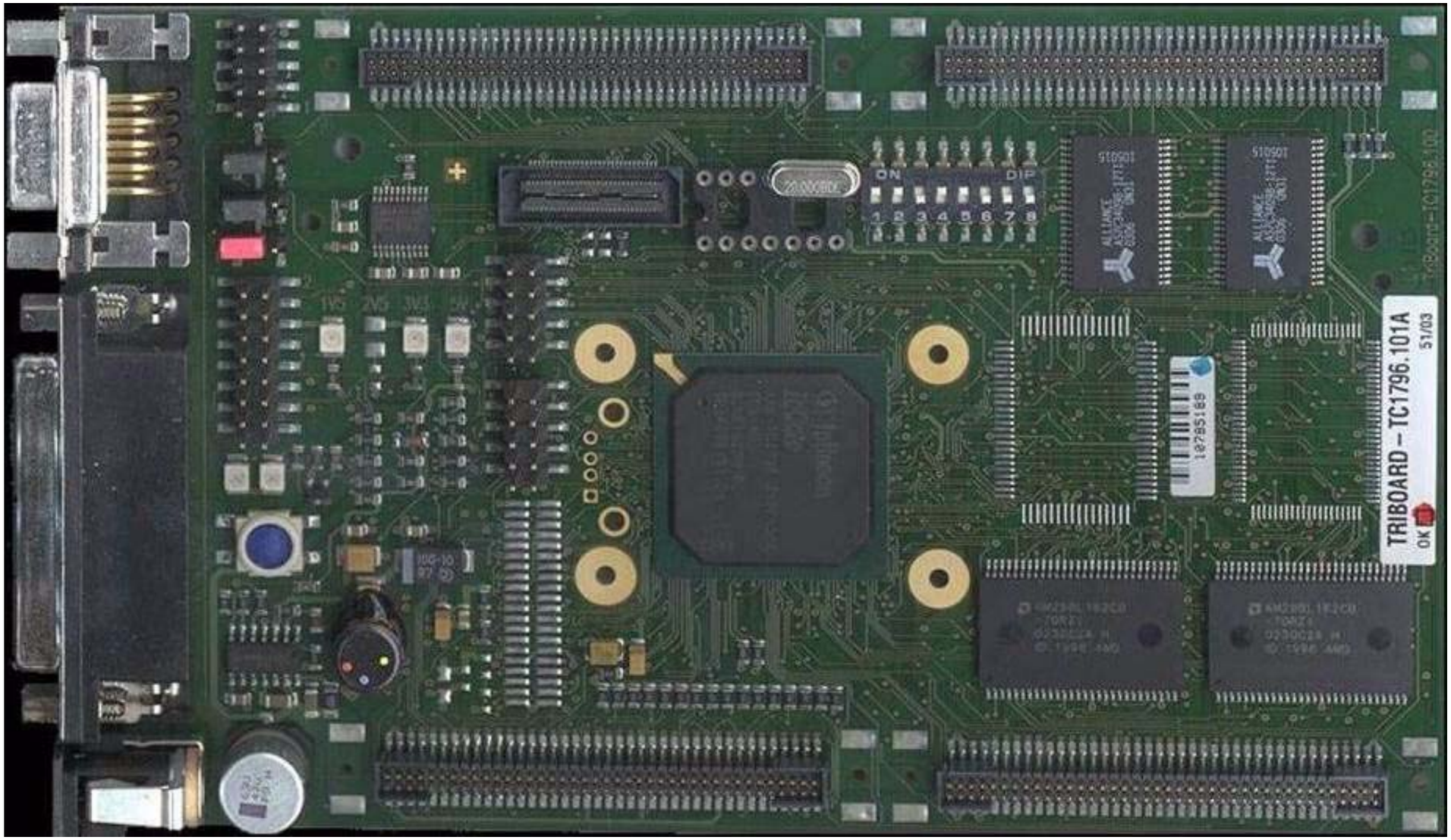
Configure the parallel port

Reboot the notebook and enter the BIOS. Set-up the parallel port mode to Bi-directional (ECP).



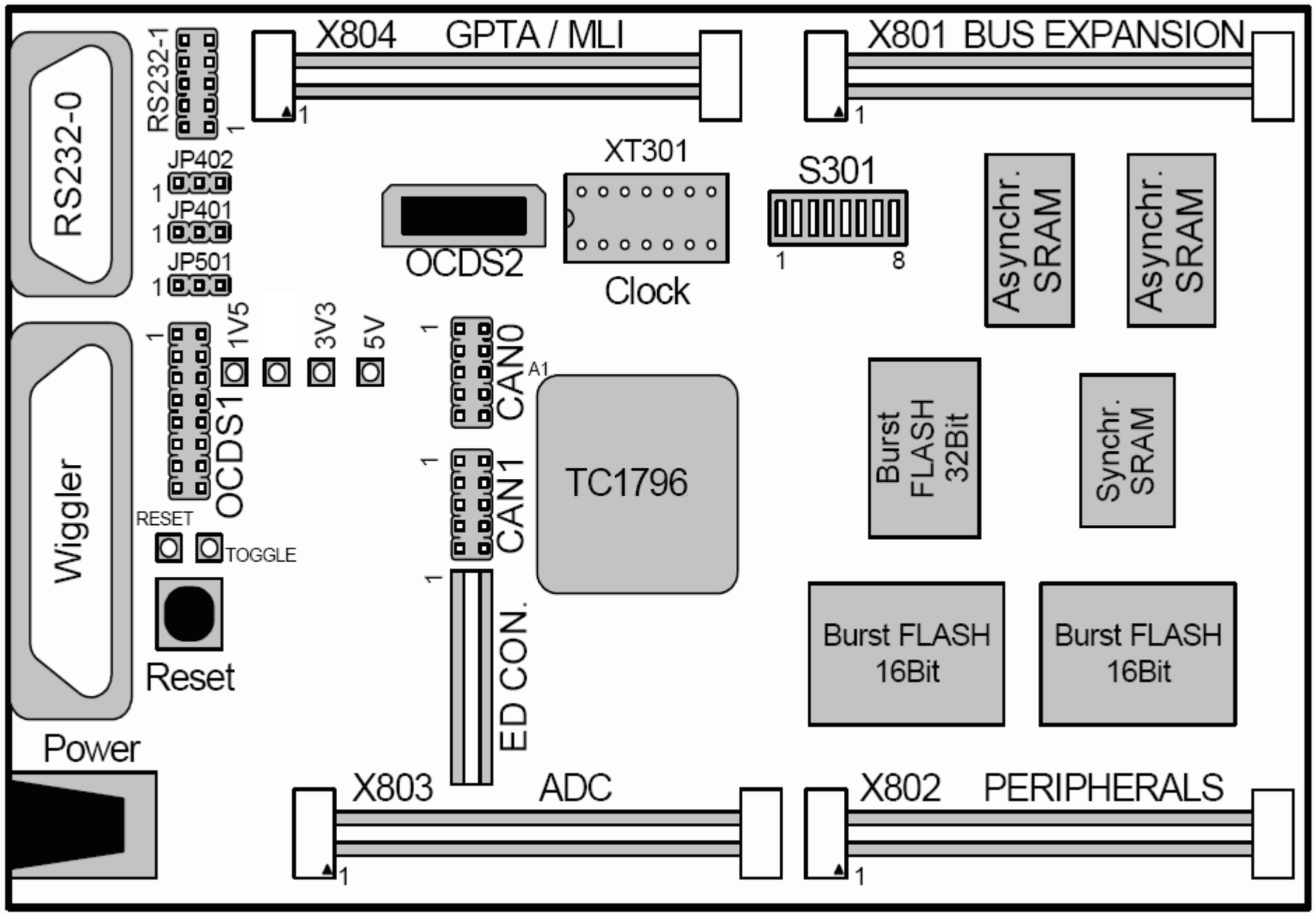
Configure the TriBoard

- Picture



Preparation

■ Schematics



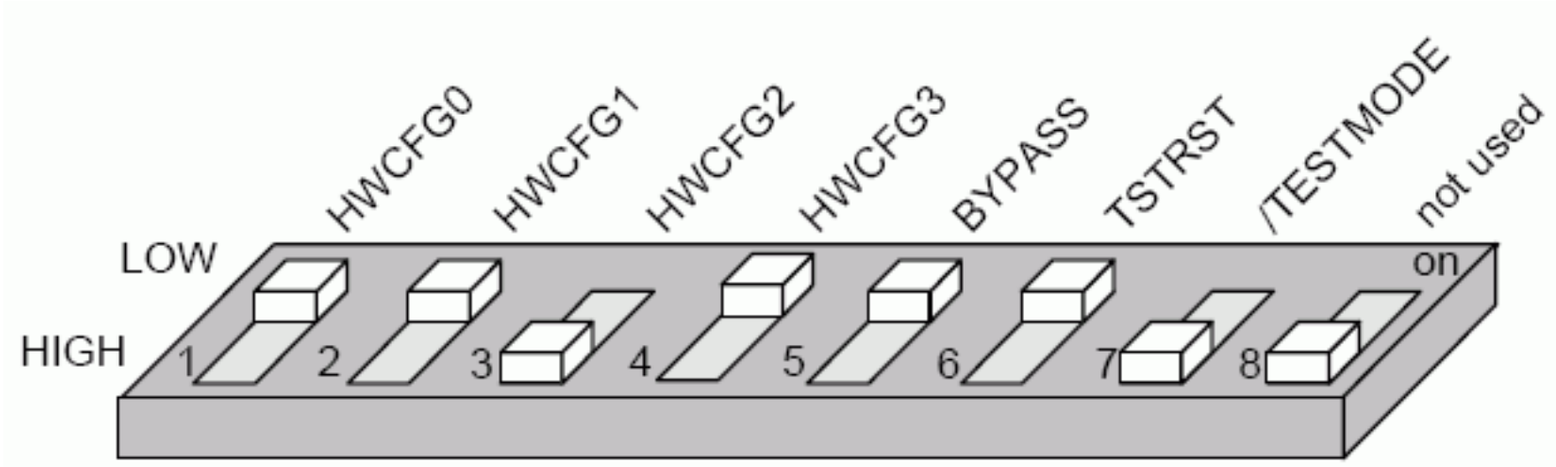
■ Connect Pin 1 and 2 of JP501 to use the on-board wiggler.

Note: The shadowed line indicates the default setting

Setting	On Board Wiggler
1 - 2	Enable On Board Wiggler

Preparation

■ Set the DIP switch S301[1..8] to {ON,ON,OFF,ON,ON,ON,OFF,OFF} to set HWCFG[3..0] = 0100



/BRK_IN	HWCFG[3...0]	Type of Boot	PC Start value
1	0000	Serial boot from ASC to PMI scratchpad, run loaded program	0xD4000000
1	0001	Serial boot from CAN to PMI scratchpad, run loaded program	0xD4000000
1	0010	Start from internal flash	0xA0000000
1	0011	Alternate Bootmode from internal flash	from Header or 0xD4000000
1	0100	External memory, EBU as master	0xA1000000
1	0101	Alternate Bootmode from External memory, EBU as master	from Header or 0xD4000000
1	0110	External memory, EBU as slave	0xA1000000
1	0111	Alternate Bootmode from External memory, EBU as slave	from Header or 0xD4000000
1	1XXX	reserved; don't use this combination	-
0	1000	go to external emulator space	0xDE000000
0	0000	put chip in tristate (deep sleep)	-
0	all others	reserved; don't use this combination	-

Preparation

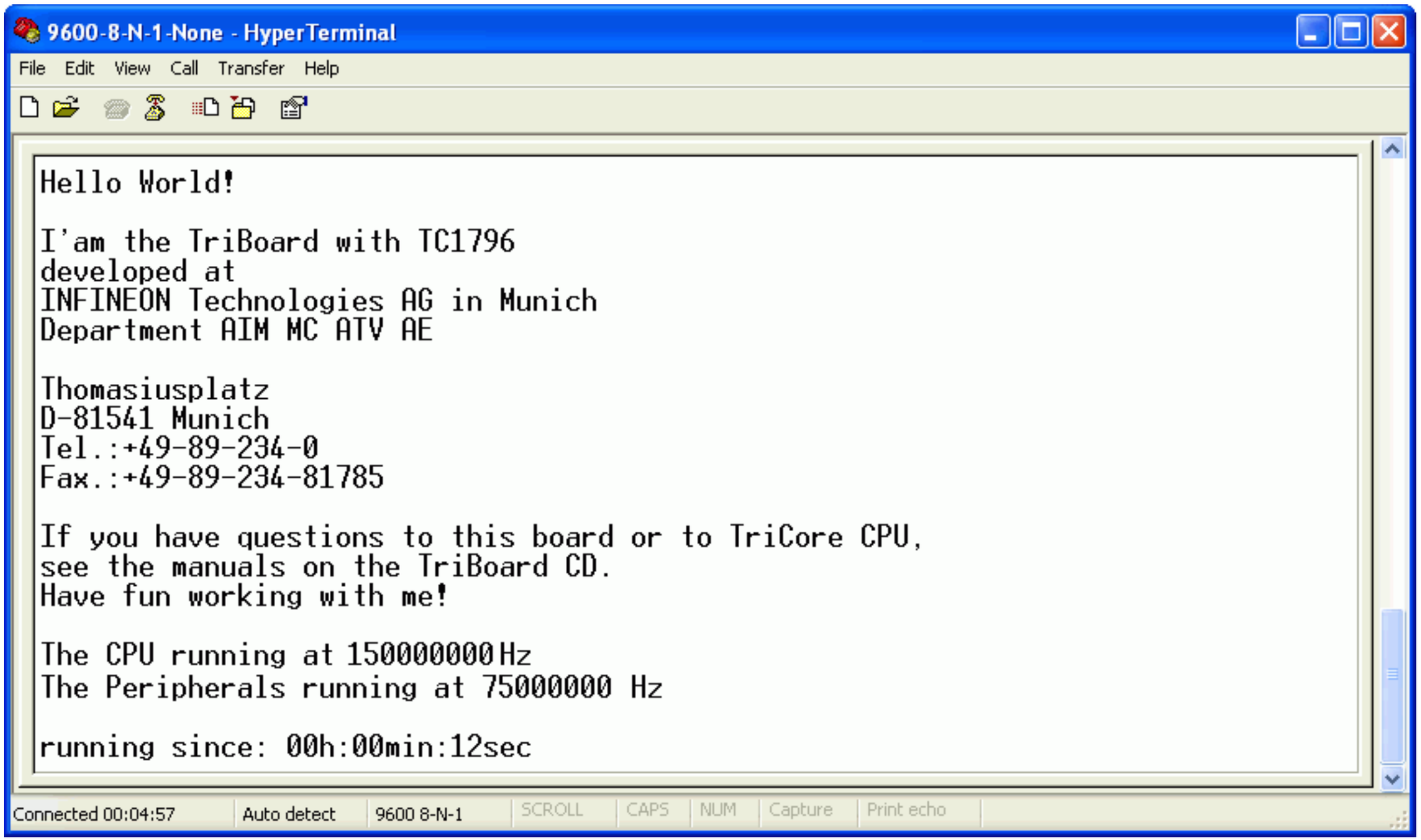
Check the TriBoard

- Make sure the TriBoard is equipped with an TC1796.
- Make sure that the MHz oscillator on board is properly connected.
- Connect the serial port ASC0 of the TriBoard to your PC using a serial cable.
- Connect a power supply Output connector:
  Hollow plug 5.0mm/2.1mm, Polarity +o)-
  Output voltage: 5.5V - 60V DC (7.5V, 500mA recommended)

Preparation

5V, 3.3V and 1.5V will be generated internally. Three green LEDs indicate the correct status of the internally generated voltages

- Open the Microsoft HyperTerminal and set-up a COM port connection with 9600, 8, N, 1, None.
- Press the reset button on the Triboard. Every Triboard comes flashed with an ‘hello, world’ program.



Preparation

Connect the TriBoard

- Connect the TriBoard to your PC using a parallel cable. The on-board wiggler is used to connect a PC to the TriBoard OCDS Level1 via the LPT port.

Install the Tasking Compiler

- Start the from the Starter Kit CD at `<CD>:\tools\altium\setup.exe`
- Follow the Instruction on the screen.

Install the PLS debugger

- Start the from the Starter Kit CD at `<CD>:\tools\pls\setup.exe`
- Follow the Instruction on the screen.

Install Infineon DAVE

- Start the from the Starter Kit CD at <CD>:\tools\infineon\dave\setup.exe
- Follow the Instruction on the screen. If your asked to install Acrobat 4 click **No**.
- Start DAvE and install the TC1130 support by choosing **View > Setup Wizard** from the View menu. Select **I want to install products from the Dave's web site** and navigate to `<CD>:\tools\infineon\dave\products`

Exercise 1: hello, world

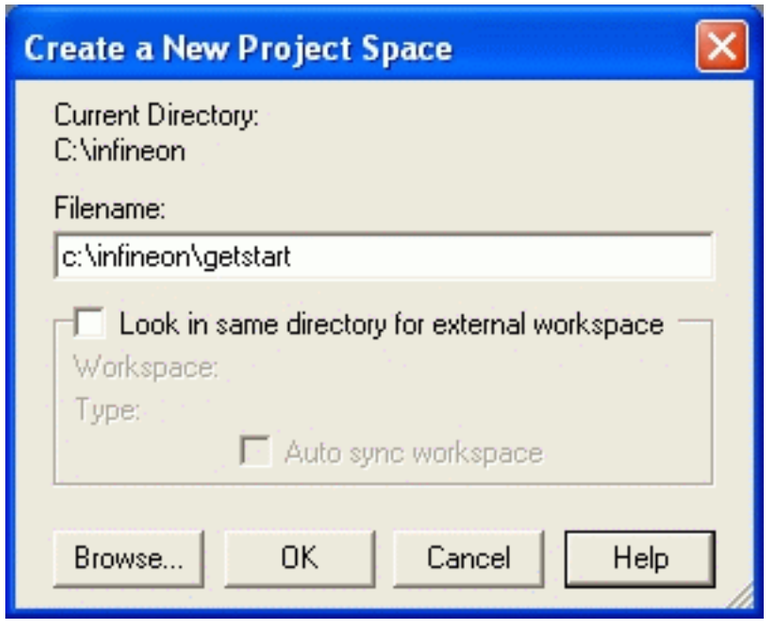
hello, world

In the first exercise you make yourself familiar how to build an 'hello, world' application. You will download and run the application on the TriBoard. The application writes the string 'hello, world' to stdout. Using the *Tasking* stdio library and the *CrossView Pro* debugger these streams are redirected via JTAG to a terminal window inside the debugger. This feature is very useful during the development and most debugger vendors offer a solution with similar functionality.

> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
>
> *Print the words*
>
> *hello, world*
>
> Programming in C, Brian W. Kernighan and Dennis M. Ritchie

1. Create a new Tasking project space

Launch *Tasking* and choose **File > New Project Space...**. In the **Filename** field enter `c:\infineon\getstart`. Click **OK**.

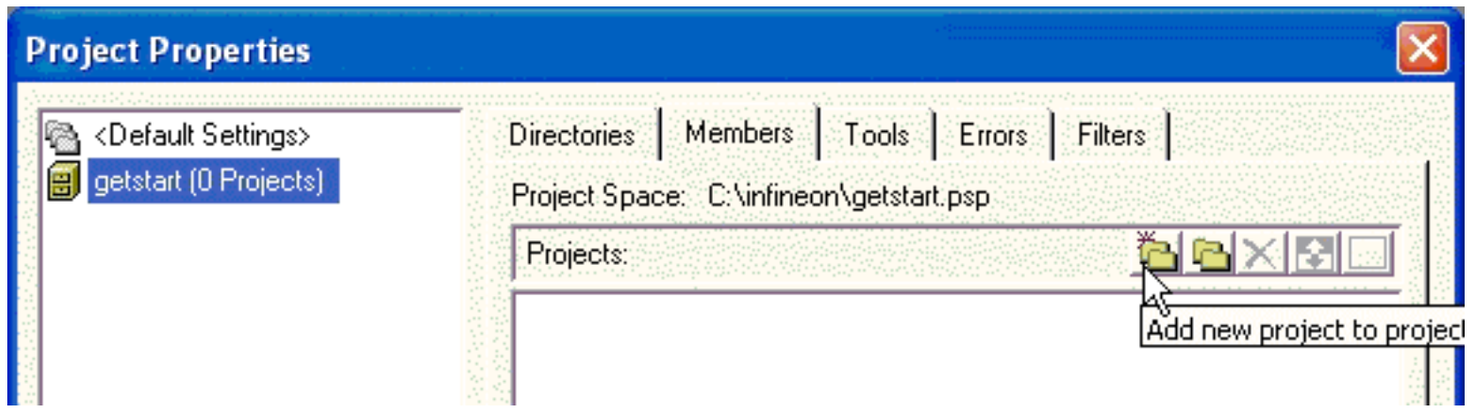


The **Project Properties** dialog appears.

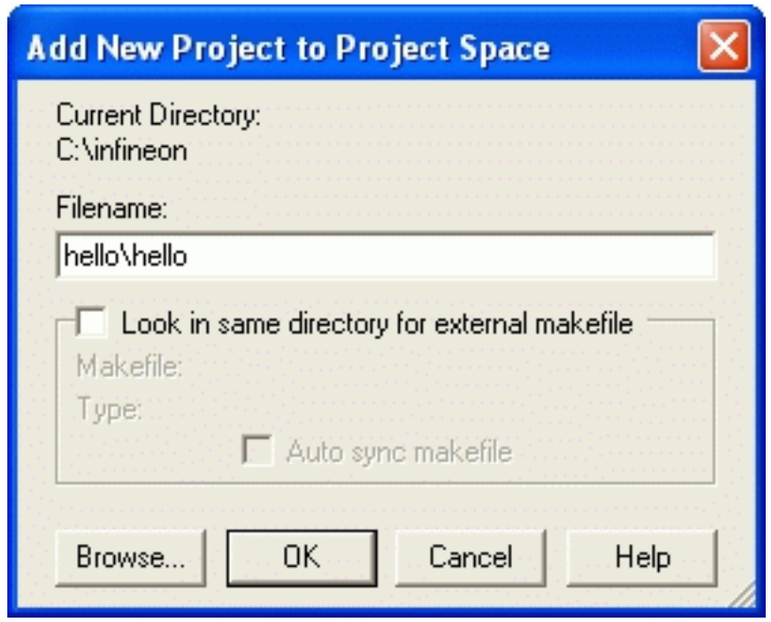
Exercise 1: hello, world

2. Create a new Tasking project

In the **Project Properties** dialog click the **New Project** icon to add a new project.



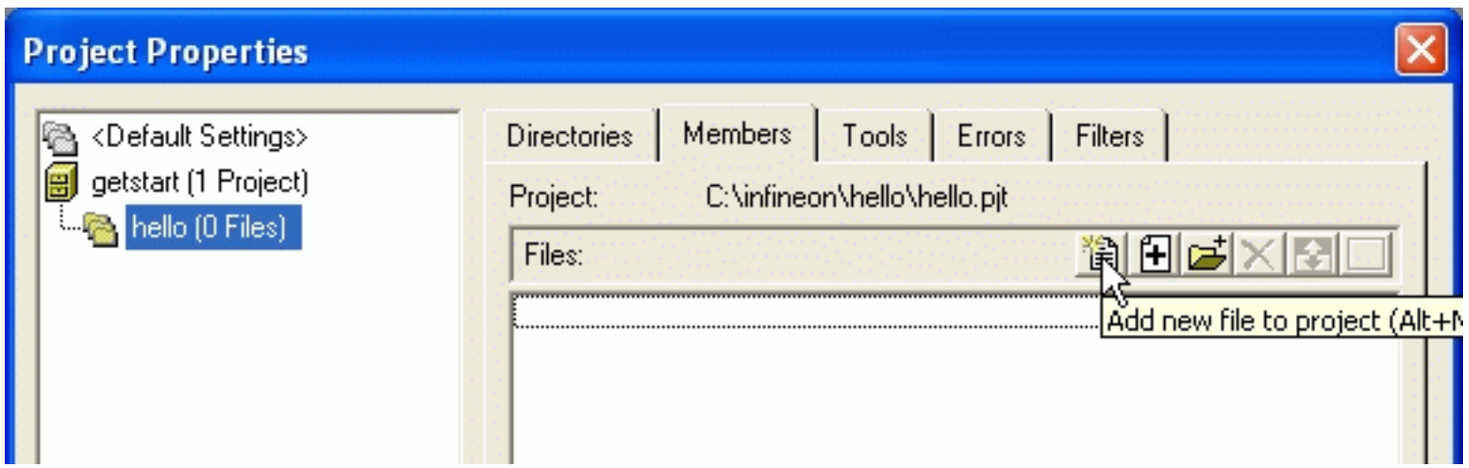
The **Add new Project to Project Space** dialog appears. In the **Filename** field enter `hello\hello` to create the new project in a subdirectory. Click **OK**.



Exercise 1: hello, world

3. Add a new source file

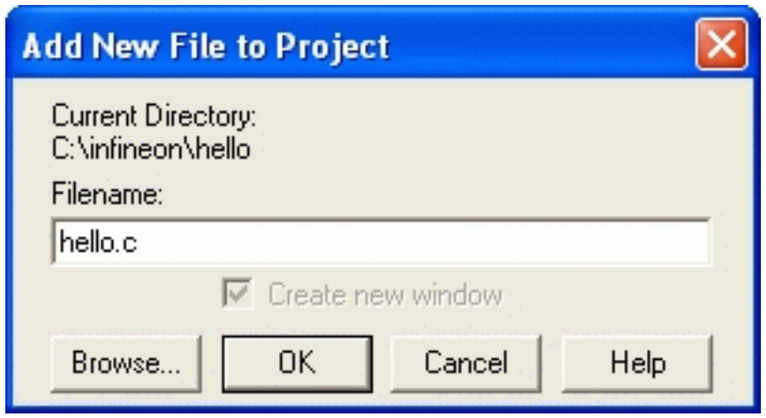
In the **Project Properties** dialog click the **New File** icon  to add a new file.

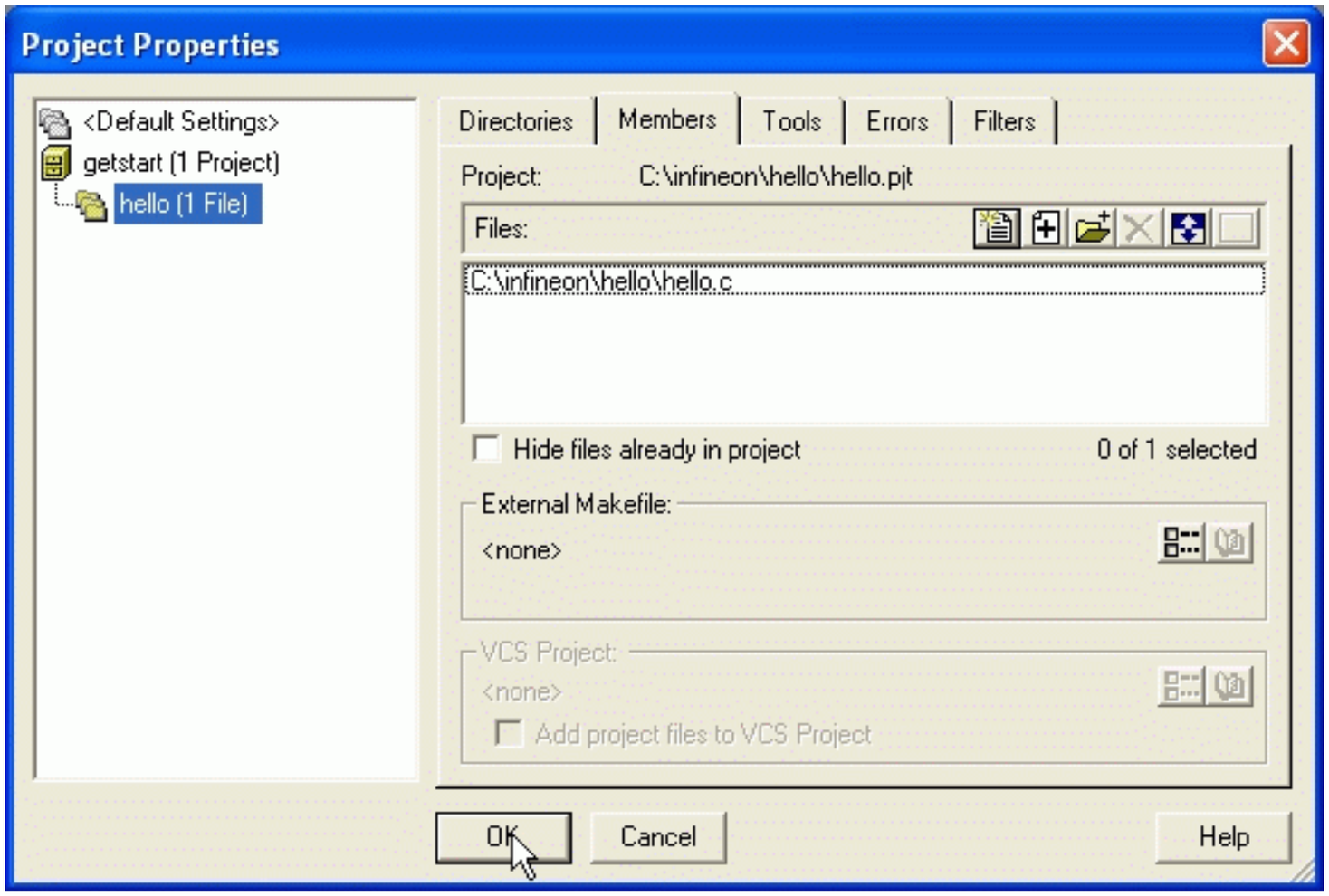


The **Add new File to Project** dialog appears.

Exercise 1: hello, world

In the **Filename** field enter `hello.c` to create and add a new source file. Click **OK**.



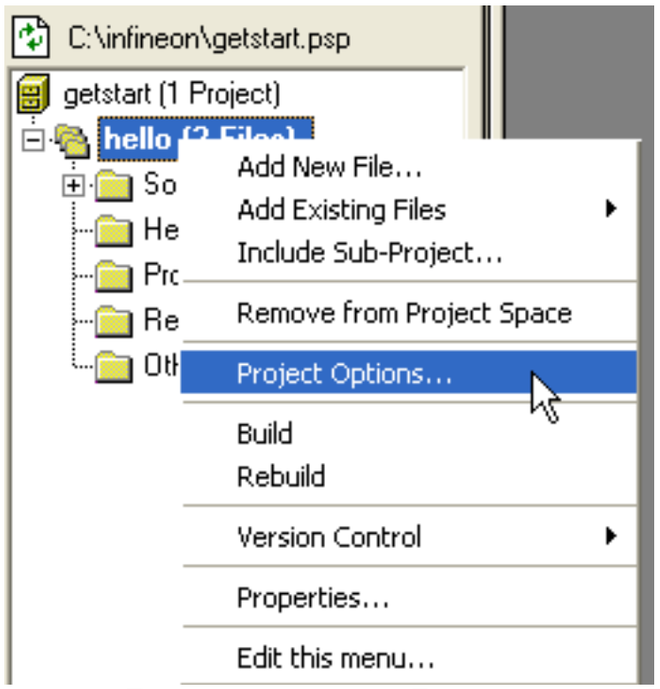


Save and close the **Project Properties** dialog by clicking **OK**.

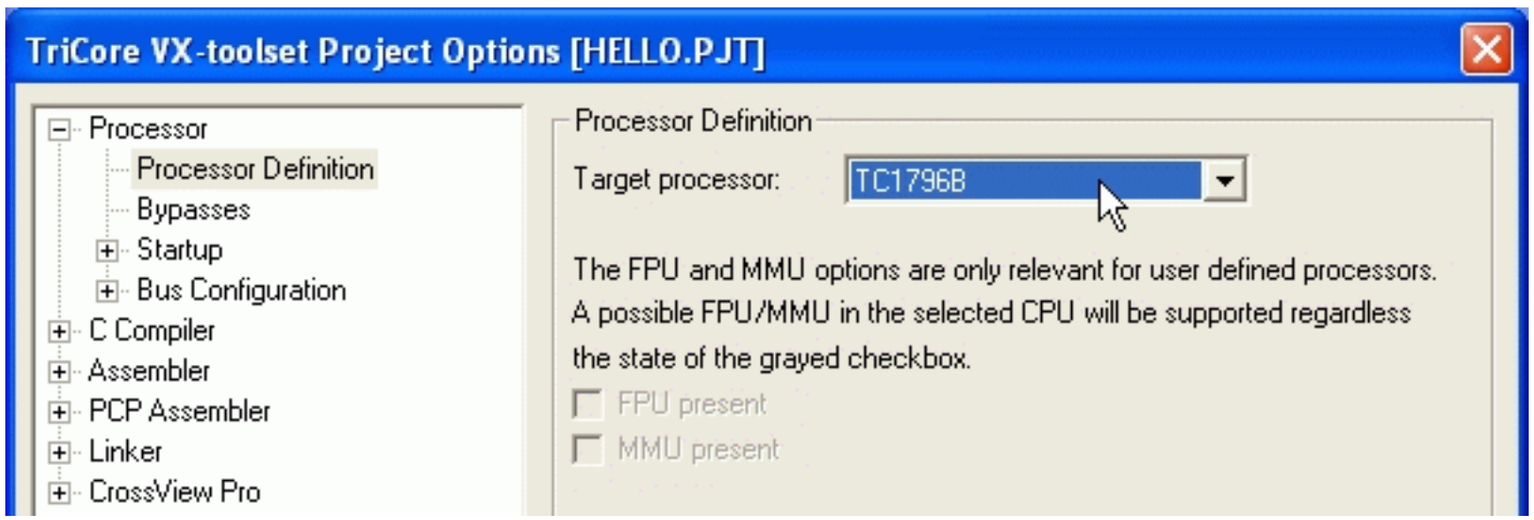
Exercise 1: hello, world

4. Set up the Project Options

Choose **Project > Project Options...** to open the **Project Options** dialog.

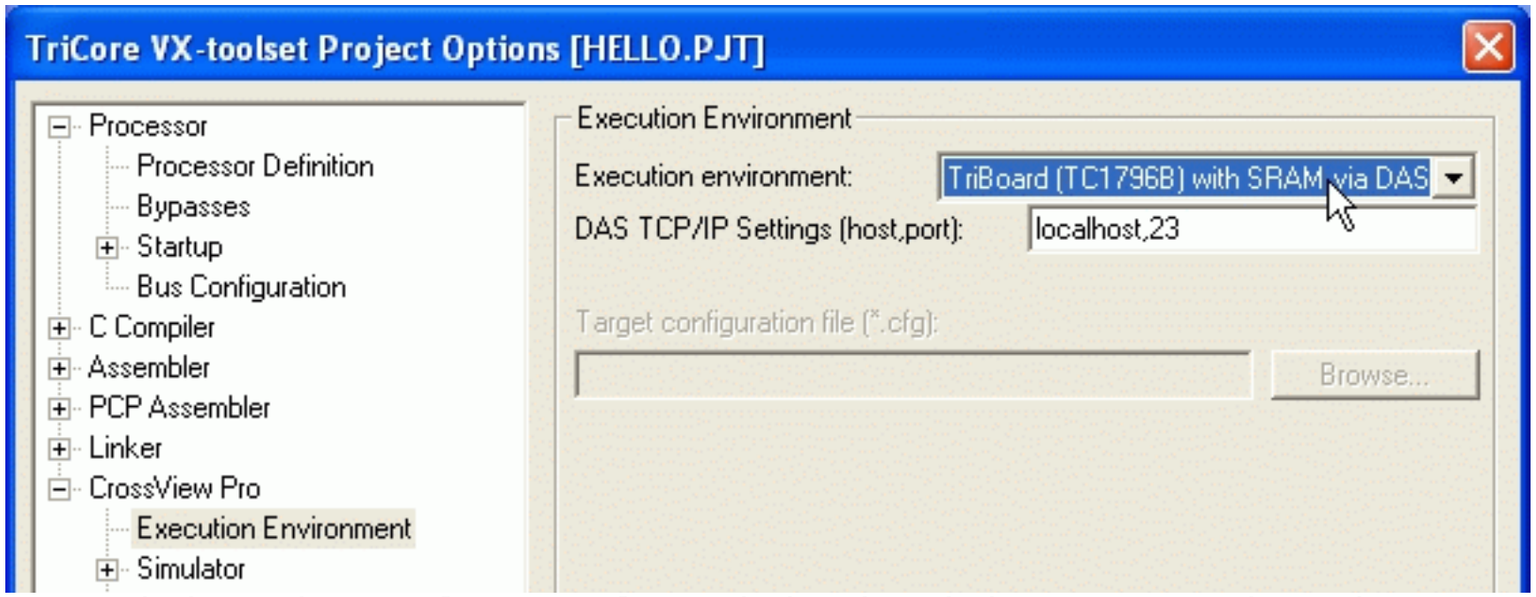


Choose **Processor > Processor Definition** from the left tree. Choose Target Processor: **TC1796B** from the drop down list.

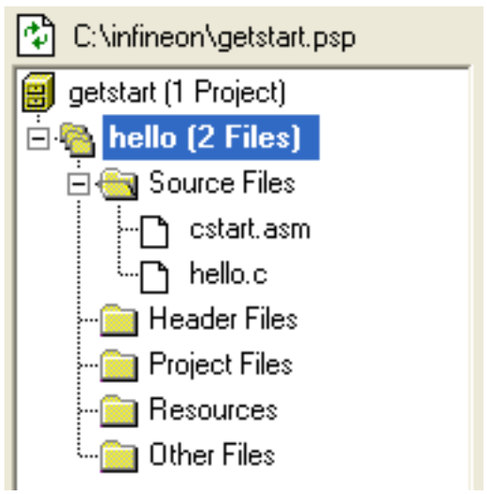


Exercise 1: hello, world

Choose **CrossView Pro > Execution Environment** from the left tree. Choose Execution environment: **TriBoard (TC1796B) via DAS** to enable debugging via *Infineon's* device access server (DAS).



Save and close the **Project Options** dialog by clicking OK. Look at the project window. Your project has 2 source files: `cstart.asm` and `hello.c`. The start-up code `cstart.asm` is copied automatically to any project from the *Tasking* installation directory.



Exercise 1: hello, world

5. Add the application code

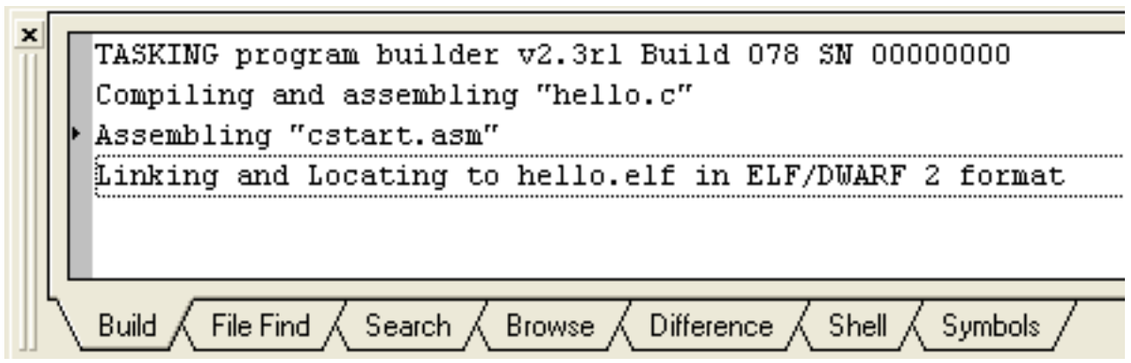
Open the `hello.c` document window and write the 'hello, world' application code.

```
#include <stdio.h>
int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

Choose **File > Save** to save your work.

6. Build the application

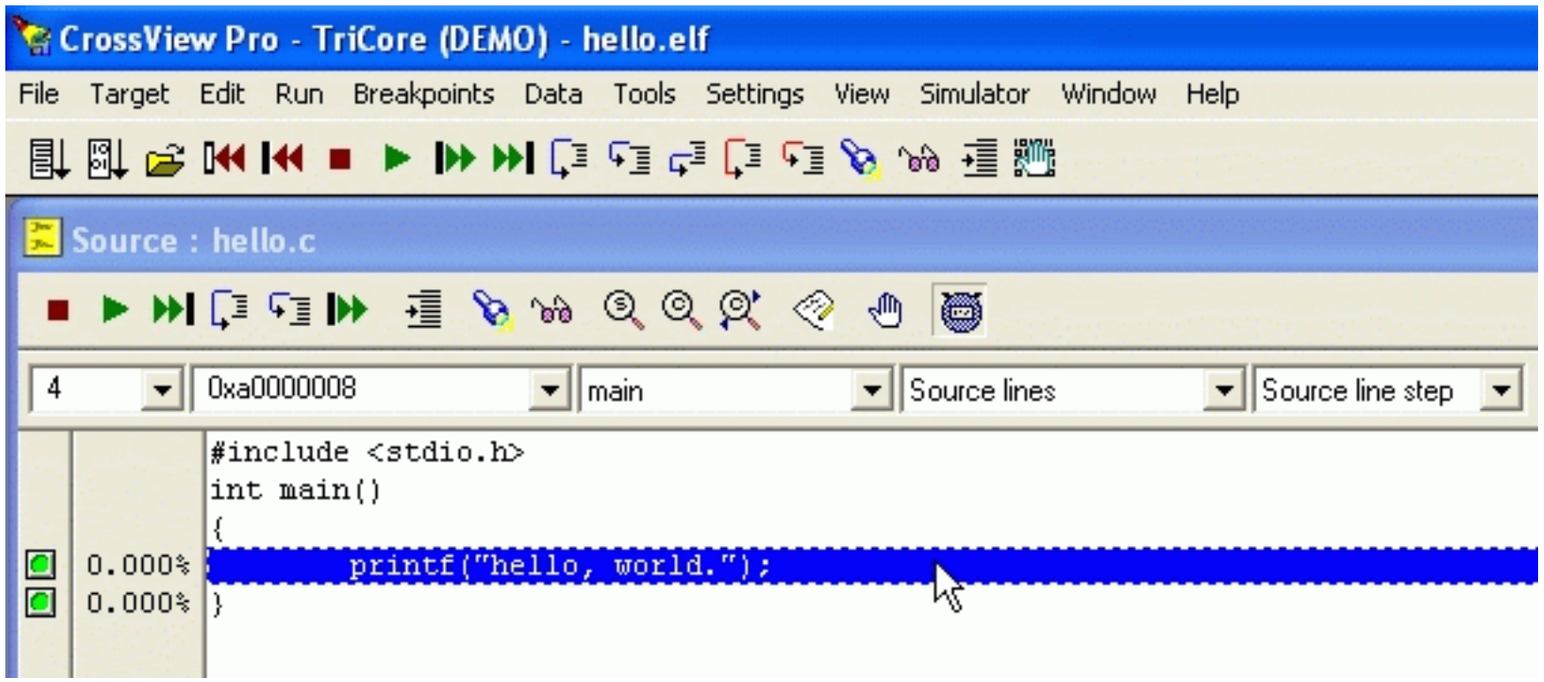
Choose **Build > Build**. The generated messages from the build process can be inspected in the **Build** tab of the output window. If the output window is not shown, choose **Window > Output** and click on the Build tab. The Build process finishes successfully generating the `hello.elf` file and can be downloaded to the target.



Exercise 1: hello, world

7. Download the application

Once the files have been compiled, assembled, linked, located and formatted they can be executed by *CrossView Pro*. To execute *CrossView Pro* choose **Build > Debug**. *CrossView Pro* will automatically download the compiled file for debugging.



8. Run the application

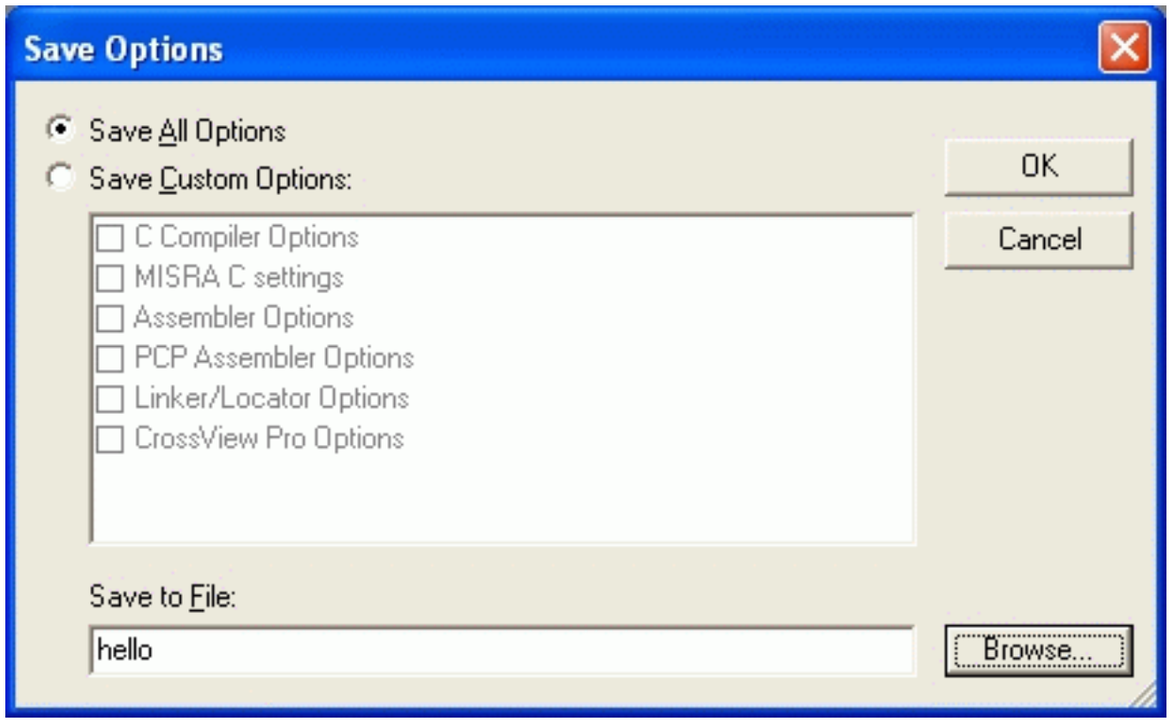
Choose **Run > Run** to execute the application. A Terminal window appears, displaying the string 'hello, world'.

Choose **Run > Reset Application** and try stepping through the source code typing F10. Choose **File > Exit** to exit *CrossView Pro*.

Exercise 1: hello, world

9. Save the project options

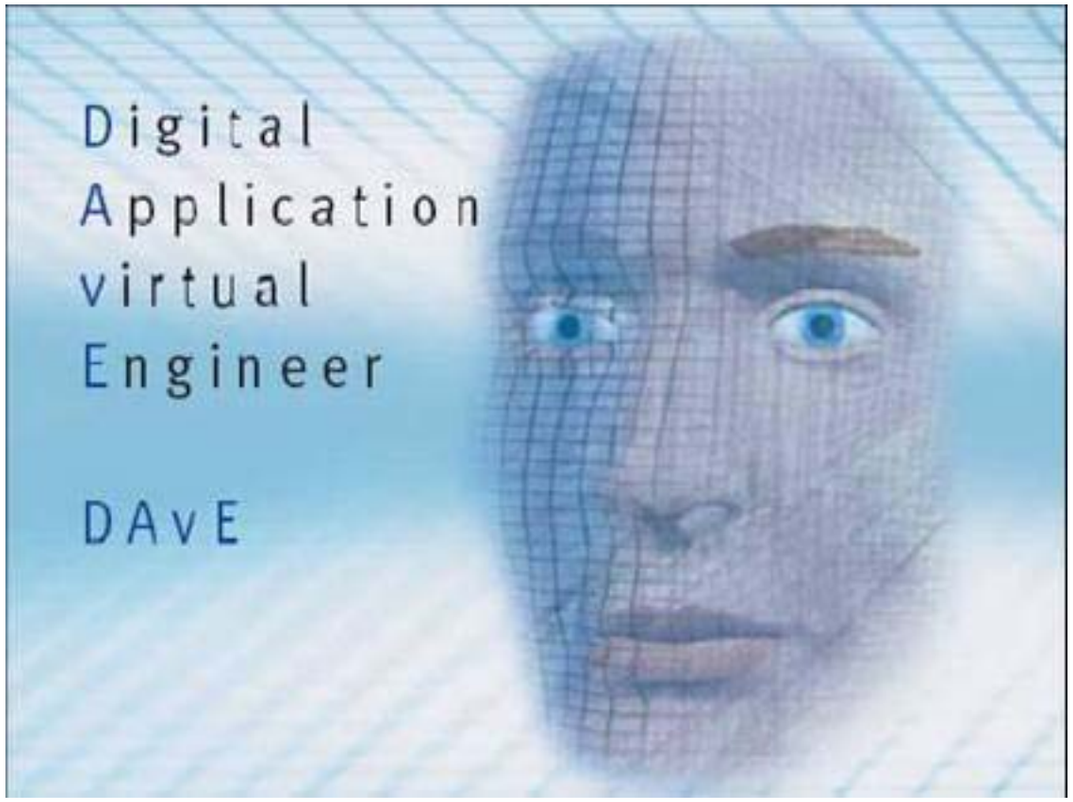
Save the project options for next exercise in a file by choosing **Project > Save Options...**. In the **Save to File** field, enter `hello` and click **OK**.



Exercise 2: LED blink

LED blink

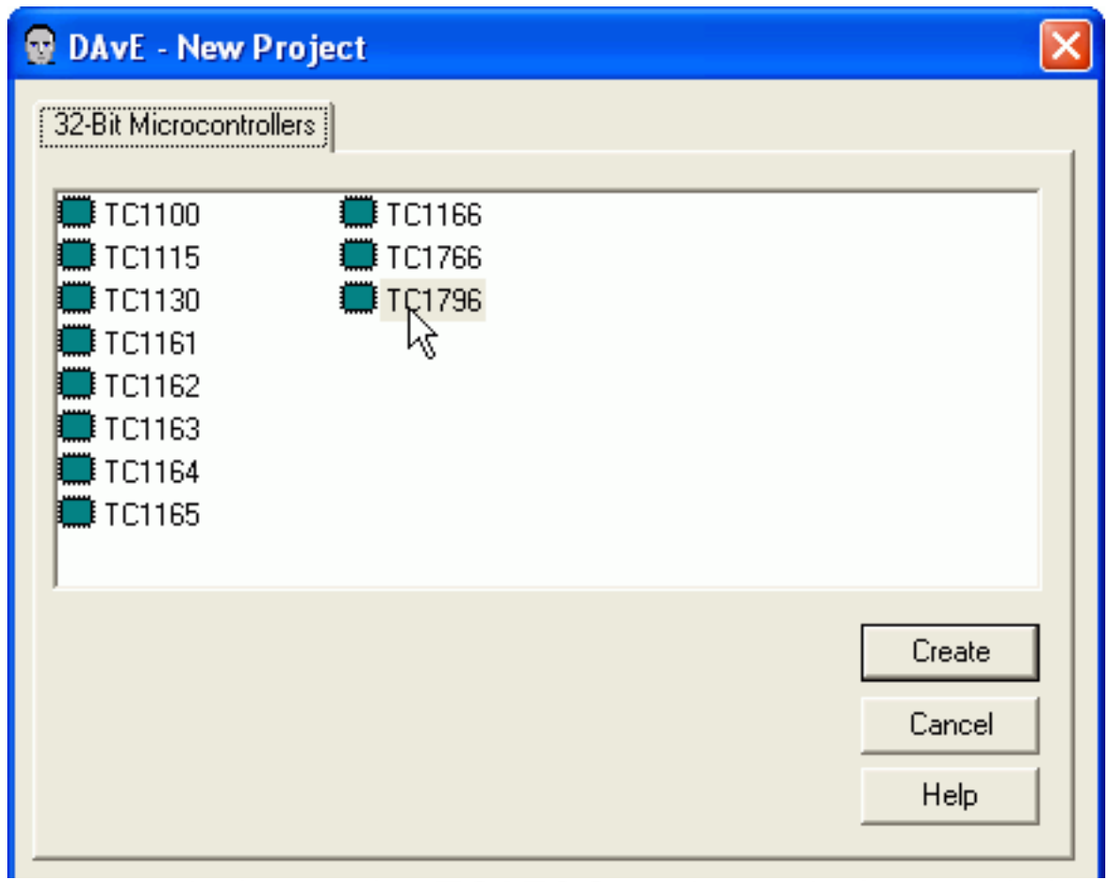
A 'hello, world' application is often impossible to write on embedded devices because stdio is not supported. Embedded programers therefore often starts with writing a simple application that toggles an led on the board. In this exercise we will use the tool that help you to configure the initialization code for *Infineon* microcontroller: *DAvE*, the Digital Application virtual Engineer. *DAvE* is available for free for any *Infineon* 8-, 16- and 32-bit microcontroller. *DAvE* generates an application framework based on the configuration you made. *DAvE* is also very helpful in understanding the microcontroller because while you set up the configuration on a higher abstraction level you are still able to inspect the changes that were made on register level. *DAvE* has also a powerful link to the users manual and lets you jump from a configuration dialog directly to the appropriate information in the online manual. In this exercise *DAvE* will be used to configure a port pin that is hard wired to an LED on the TriBoard.



Exercise 2: LED blink

1. Create a new DAvE project

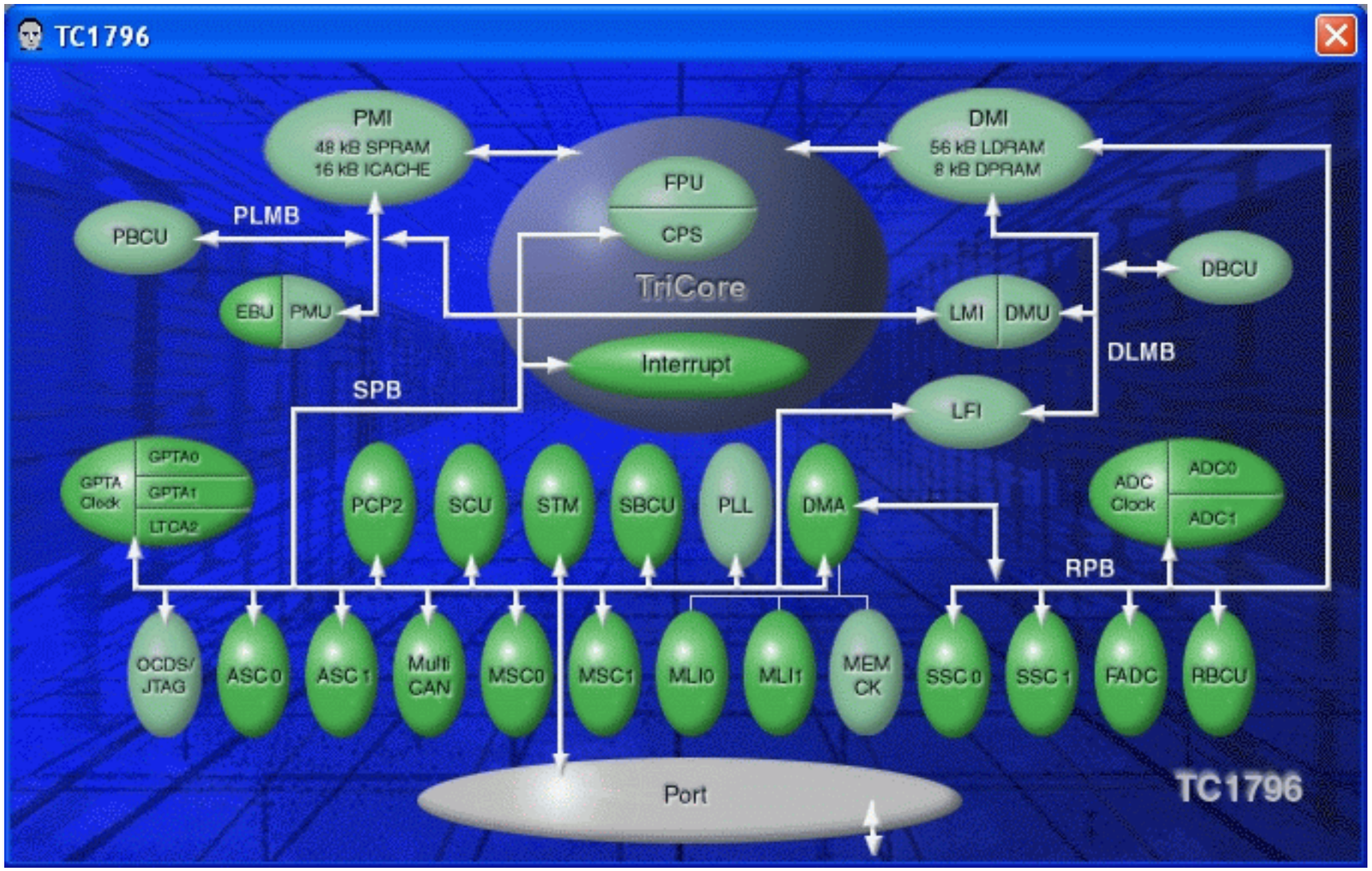
Launch *DAvE* and choose TC1796 from the list of 32-bit microcontrollers. Click **Create**.



Two new windows, the **TC1796** project window and the **Project Settings** dialog appear.

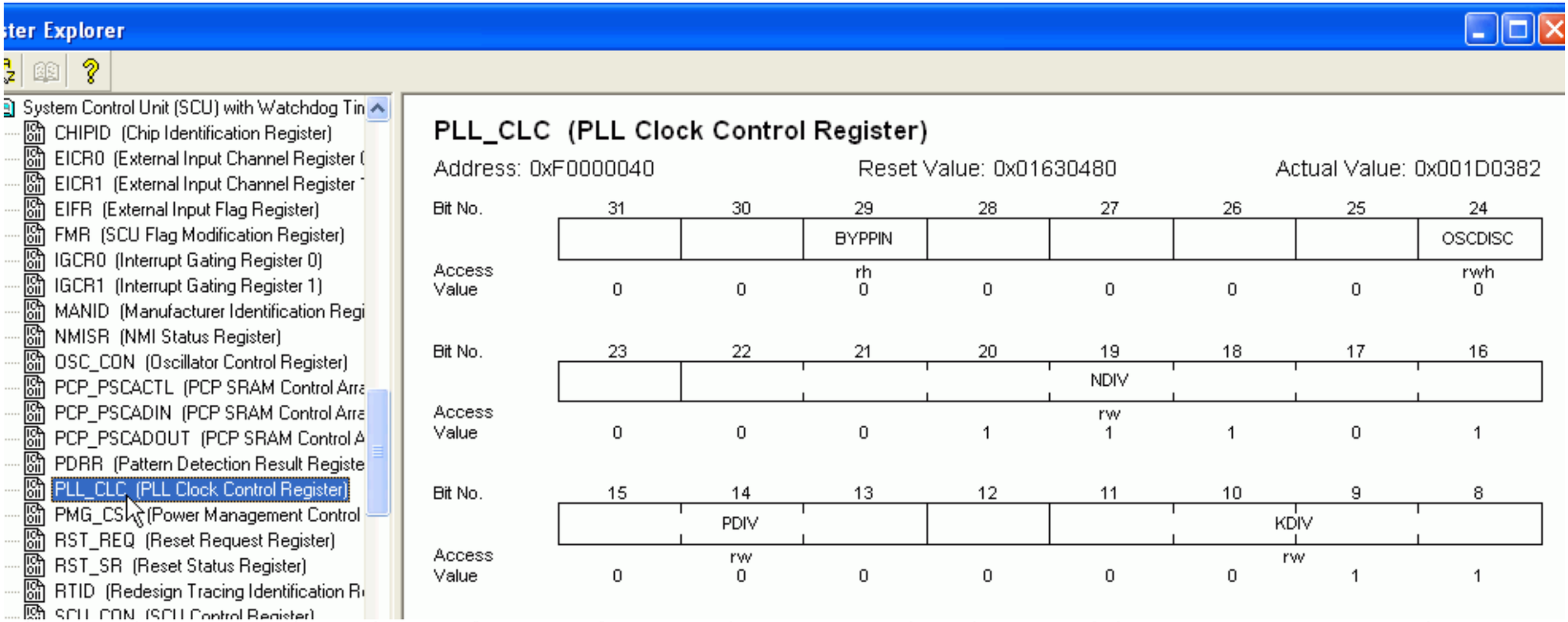
Exercise 2: LED blink

The project window presents an overview of the TC1796 architecture. By clicking on each of the items, the properties of the item can be viewed and modified.



Exercise 2: LED blink

DAvE gives you a complete overview of all registers. Choose **View > Register Explorer** an browse e.g. to **System Control Unit (SCU) > PLL_CLC**.

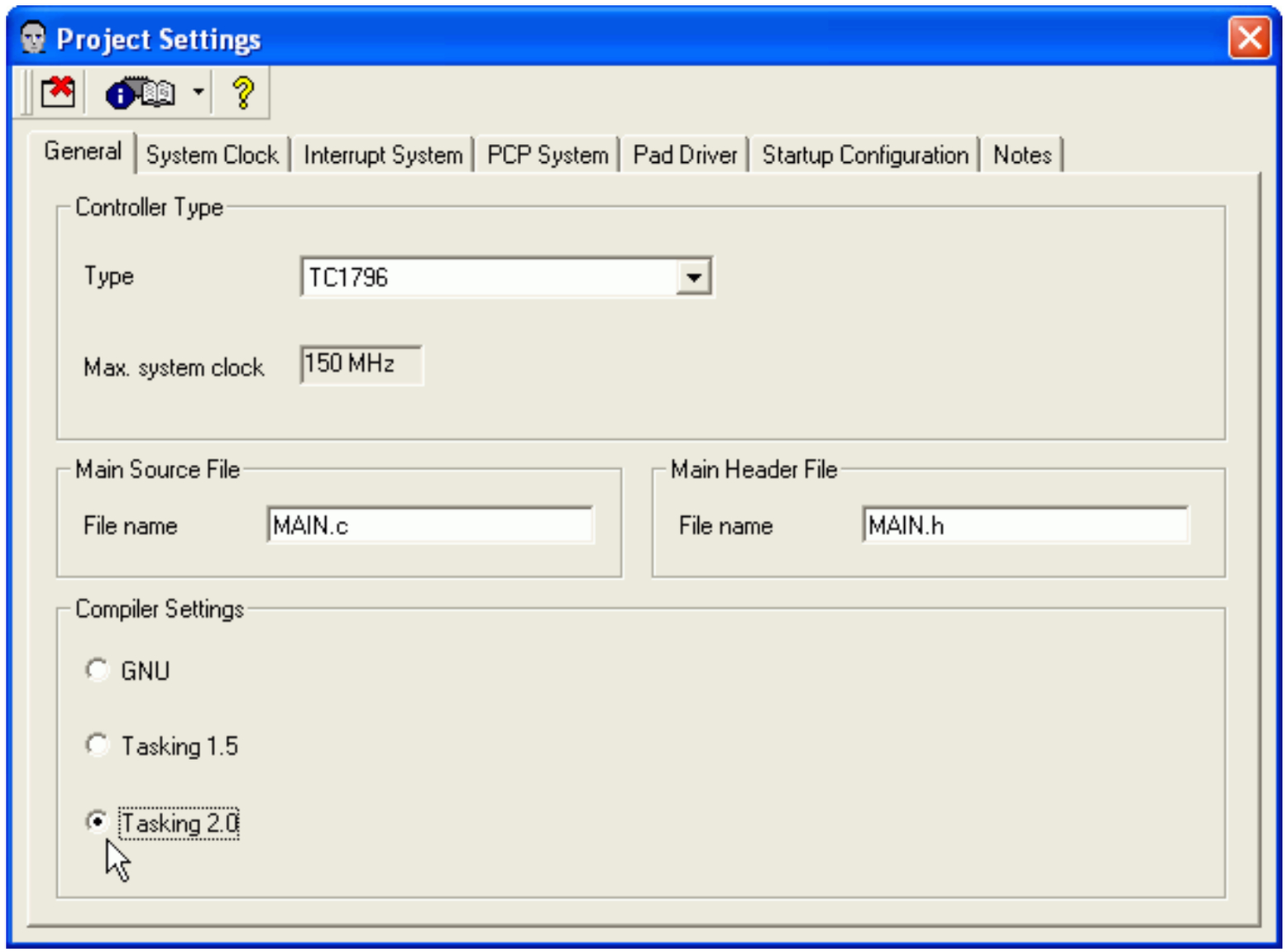


It's a good practice to keep this **Register Explorer** window open, so that changes of the register settings can be seen instantaneously. The bottom left tree shows a list of all registers which has been changed since the last time the **Register Explorer** window was opened. Click the **Documentation** icon 📖 or use the context menu on the tree item to go directly to the online manual explaining the details of the register.

Exercise 2: LED blink

2. Set up the Project Settings

In the **Project Settings** dialog select the Compiler settings *Tasking* 2.0. If the dialog is not open choose **File > Project Settings**.



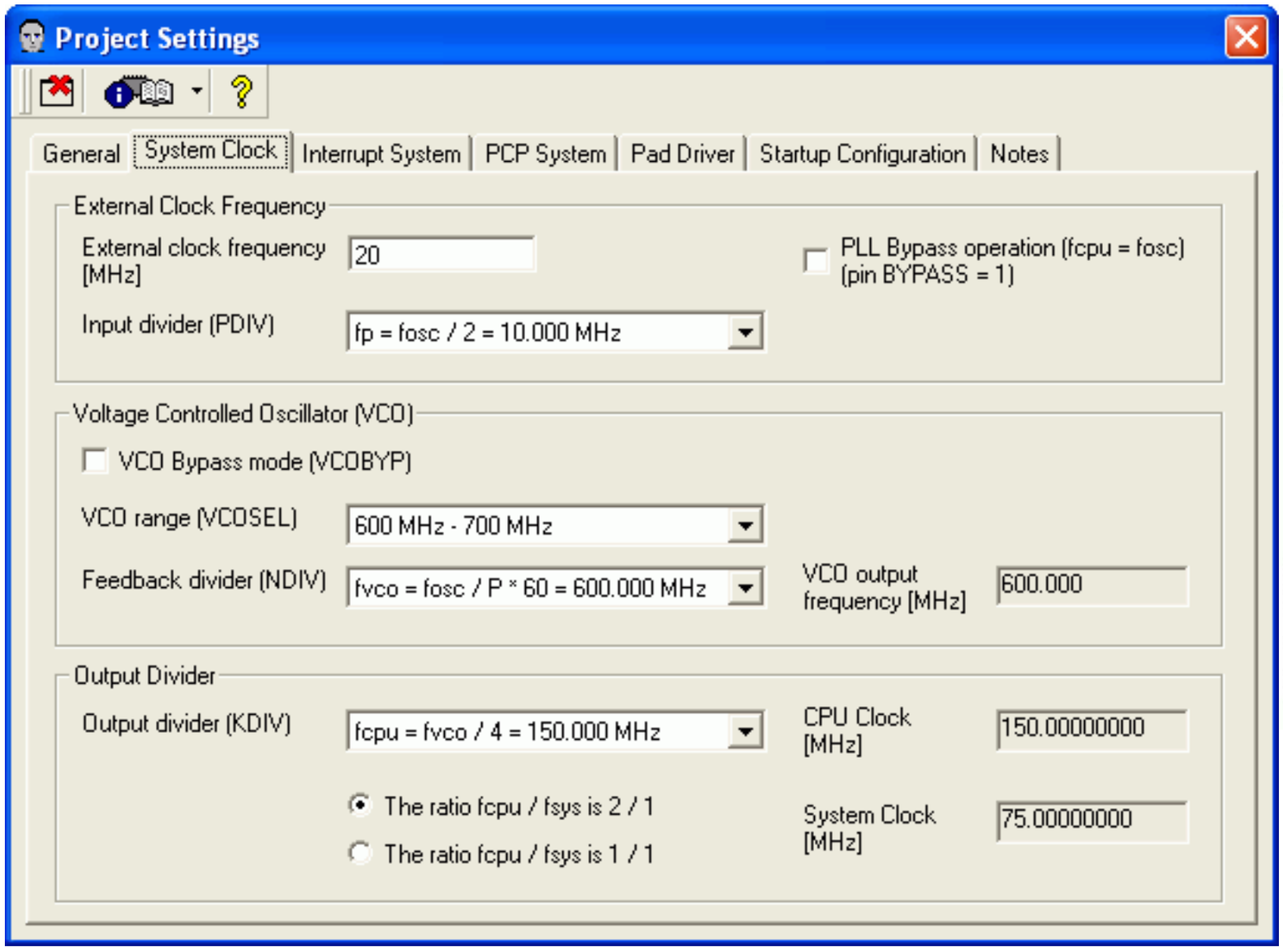
Most of the dialogs in *DAvE* have the same style. A multitab dialog where each page is used to configure a specific part. The dialogs have a small toolbar with a **Close** icon, a **Documentation** icon and a **Help** icon. The documentation menu is context sensitive showing only links to information appropriate for the current dialog.

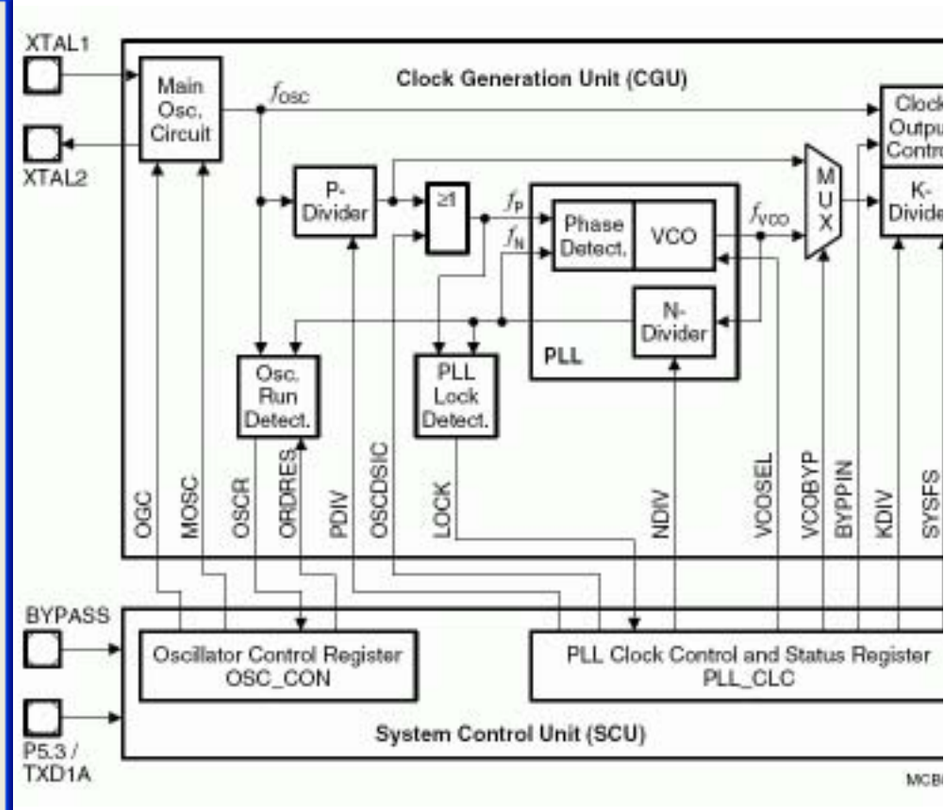
Exercise 2: LED blink

3. Set up the System Clock.

On the **System Clock** page

■ Set the external clock frequency to 20 MHz, PDIV = 2, NDIV = 60, KDIV = 4.





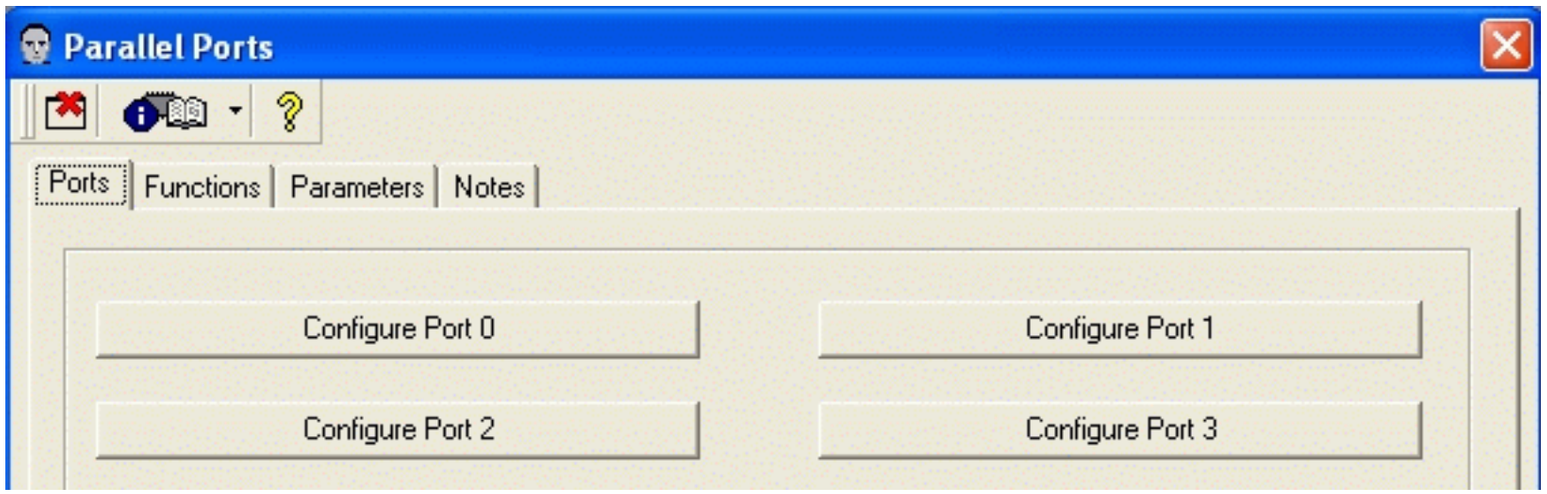
Click the **Close** icon on the dialog toolbar to close the dialog.

Exercise 2: LED blink

4. Configure the Port.

Click on **Port** in the project window to open the **Parallel Ports** properties.

The **Parallel Ports** properties dialog appears. Click on **Configure Port 1**.



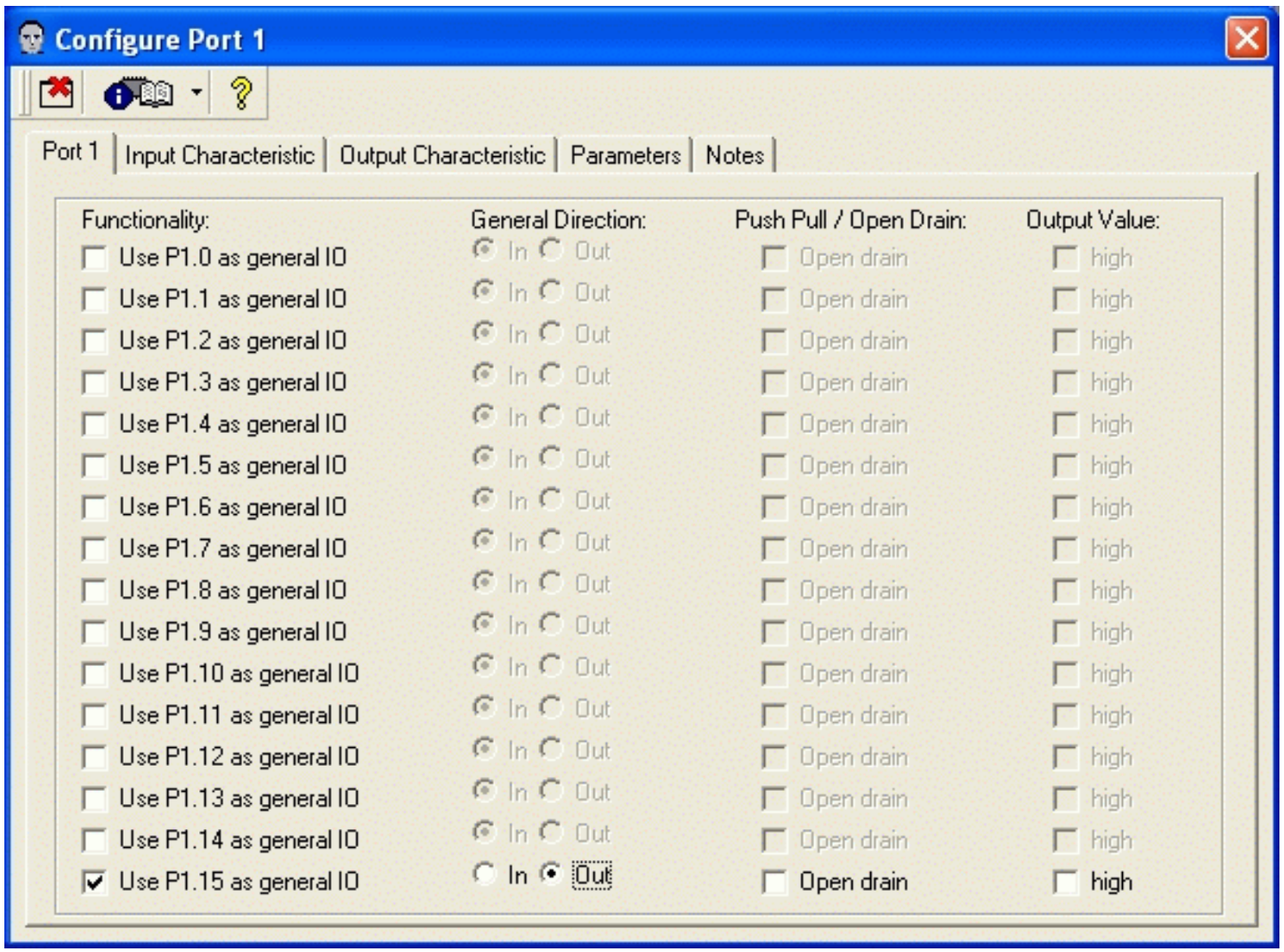
The **Configure Port 1** dialog appears.

5. Configure Port

One of the ports is always connected to an LED on the TC1796 TriBoard. The information can be found in the TriBoard Manual.

Exercise 2: LED blink

On the **Port 1** page
- Check Functionality **Use P1.15 as general IO**,
- Select General Direction **Out**.

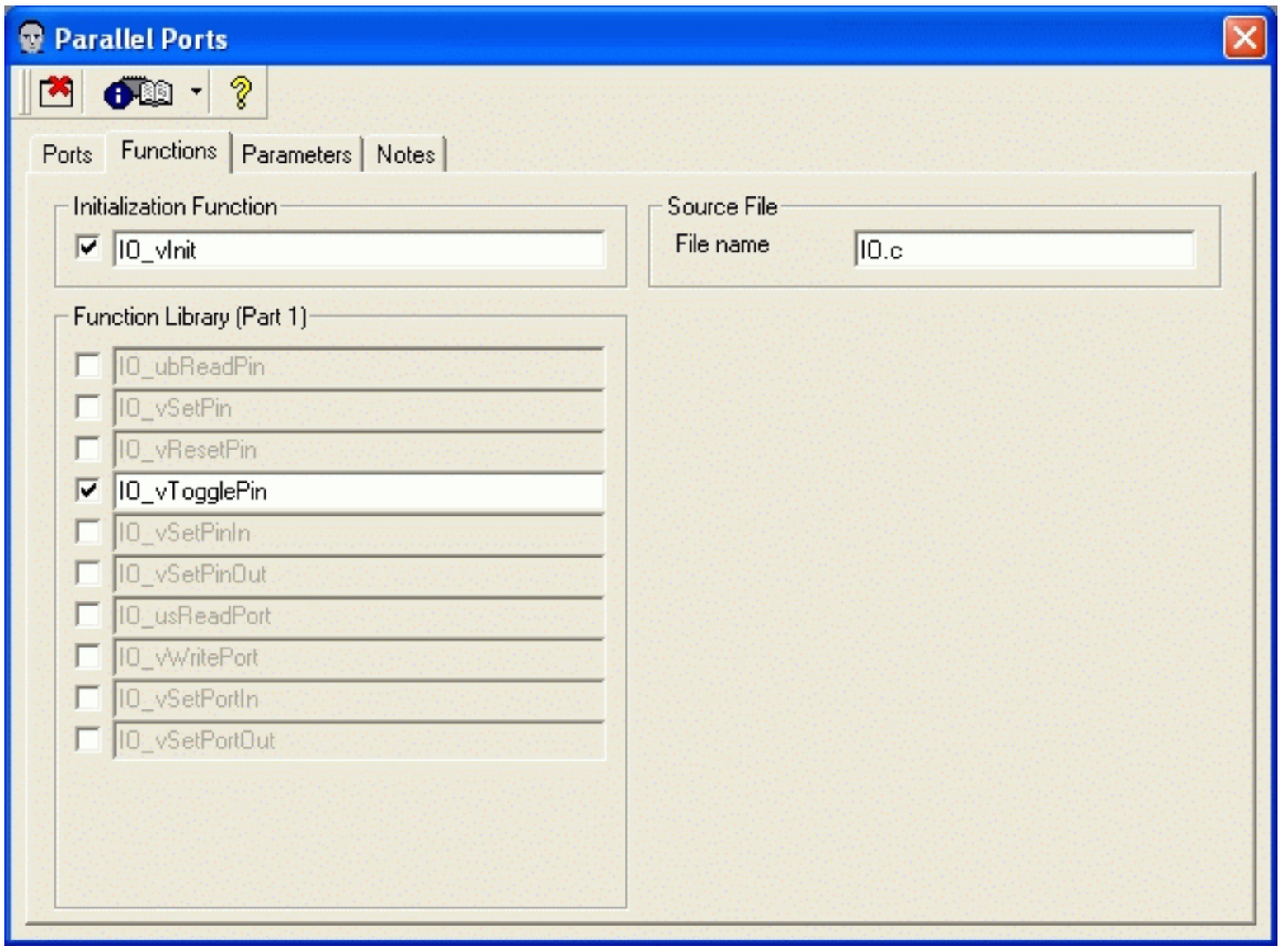


If the **Register Explorer** is still open, register P1_IOCR12 is added to the changed register view. See the changes of byte PC15 in register P1_IOCR12.
Click the **Close** icon on the dialog toolbar to close the dialog.

Exercise 2: LED blink

6. Enable the source code generation for the Port

On the **Functions** page check the module initialization function `IO_vInit` and the `IO_vTogglePin` function.



DAvE has the opportunity to generate only the function you need in your application. This keeps you source code clear and the application code small.
DAvE has a function naming schema: `<module>_<return value><function>`,
where `<module>` is the module name in capital letters, `<return value>` is `v` (void), `ub` (unsigned byte), `uw` (unsigned word), `vi` (void interrupt) etc. and `<function>` is the function name.

Exercise 2: LED blink

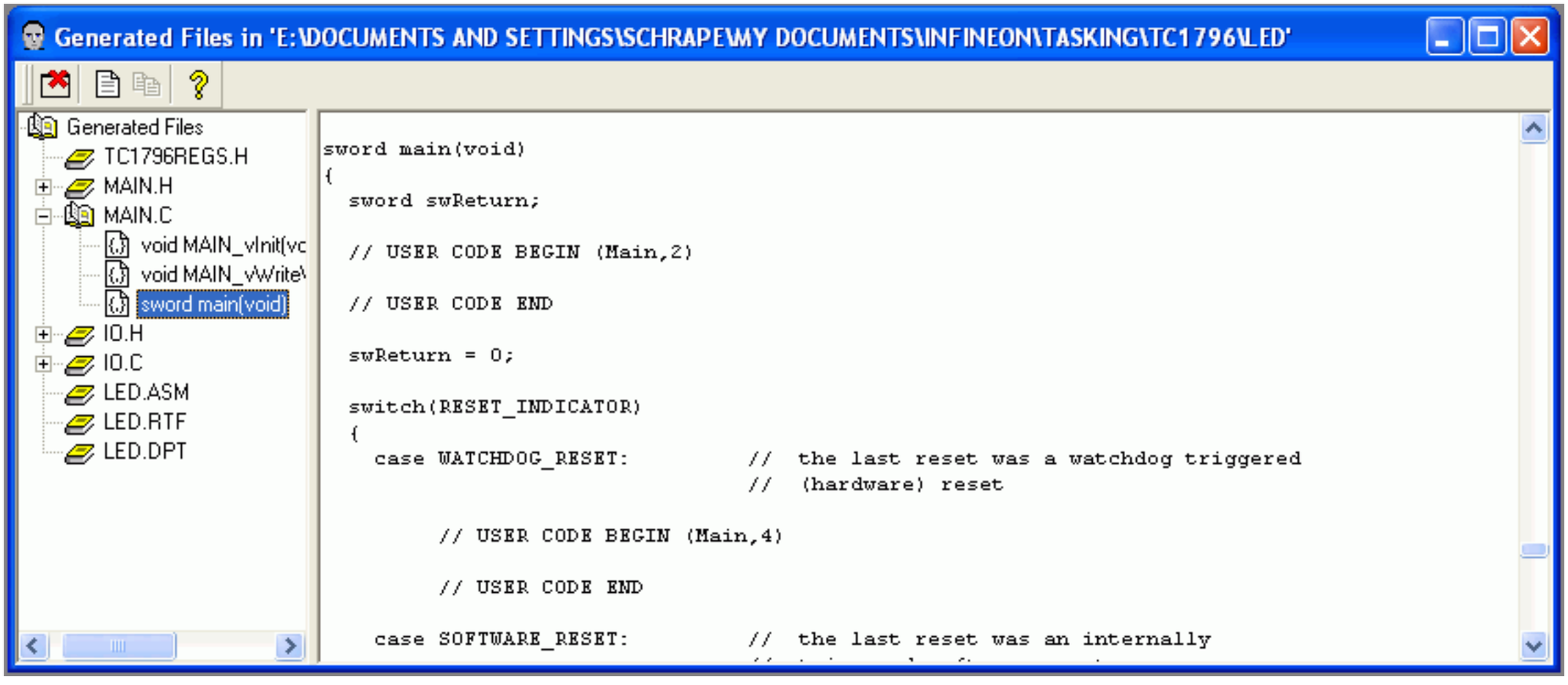
Click the **Close** icon on the dialog toolbar to close the dialog.

7. Save the project

Choose **File > Save** and save the project as `c:\infineon\led\led.dav`.

8. Generate the application framework

Choose **File > Generate Code** to start the code generation process. *DAvE* looks in the project directories for already existing files, reads these files and copies all lines between the comment lines `// USER CODE BEGIN` and `// USER CODE END` to the newly generated files. This superior feature makes *DAvE* to be used continuously during the whole development time.
The generated files can be explored in the new window.



Exercise 2: LED blink

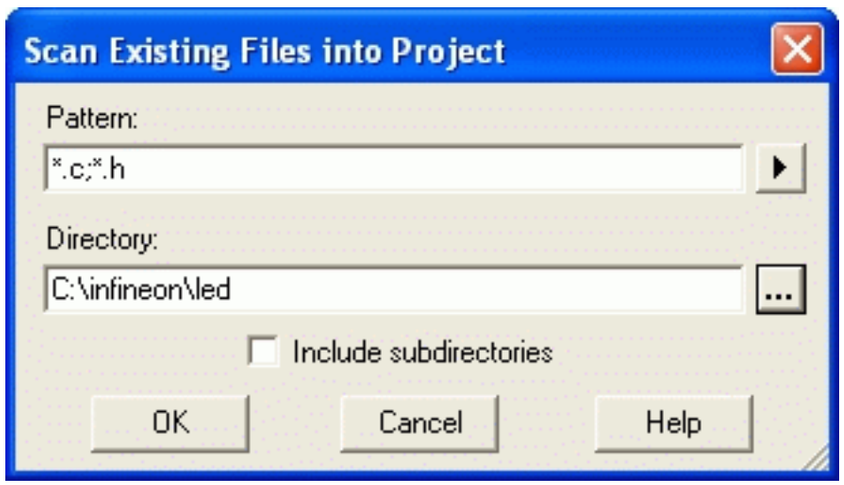
To edit the file click the **Open** icon in the dialog toolbar to open the file with the registered file type viewer of your Windows Explorer. This should be the *Tasking* EDE. Save and Close the *DAvE* project.

9. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\led\led.pjt`.

10. Add the application framework

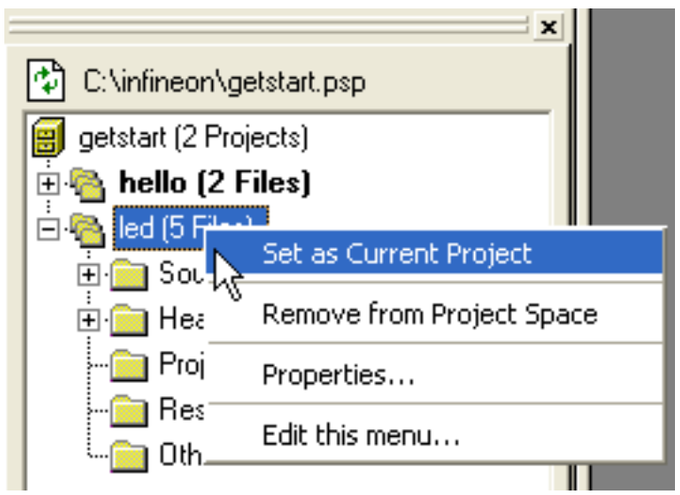
In the **Project Properties** dialog click the **Scan** icon . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.



Exercise 2: LED blink

11. Set current project

Use the context menu in the workspace window to make the led project the current project.



12. Load the project options

Choose **Project > Load Options** and load the option file from the previous exercise. Enter `c:\infineon\hello\hello.opt` in the **Filename** field and click **OK**.

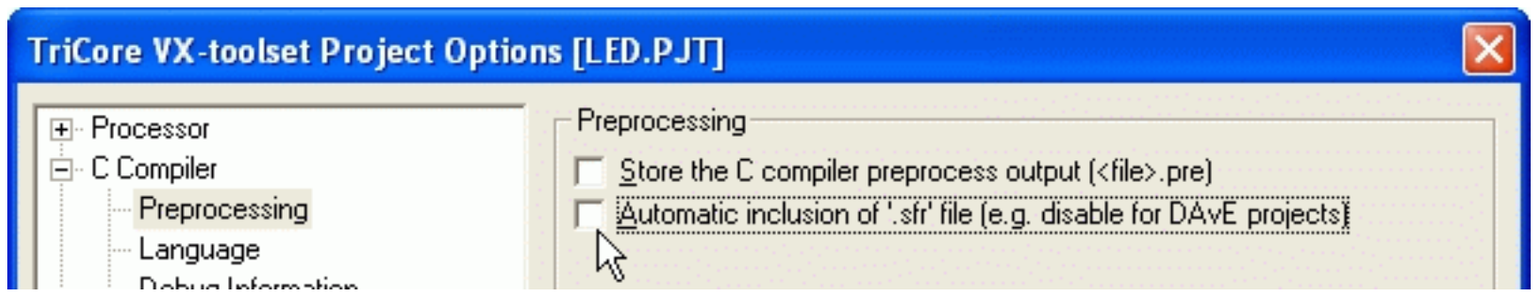
13. Build the application

Click the **Build** icon  on the Build toolbar. The system is issuing a long list of warnings:
DAvE projects and *Tasking* project uses different register definition files. *Tasking* automatically includes a special function register file `regtc1796.sfr` which is part of the *Tasking* installation. *DAvE* projects needs the `TC1796Regs.h` file that was created during the code generation process.

Exercise 2: LED blink

14. Modify the project options

Choose **Project > Project Options...**.
In **C Compiler > Preprocessing** uncheck **Automatic inclusion of '.sfr' file (e.g. disable for DAvE projects)**.



Close and Save the **Project Options** dialog by clicking **OK**.

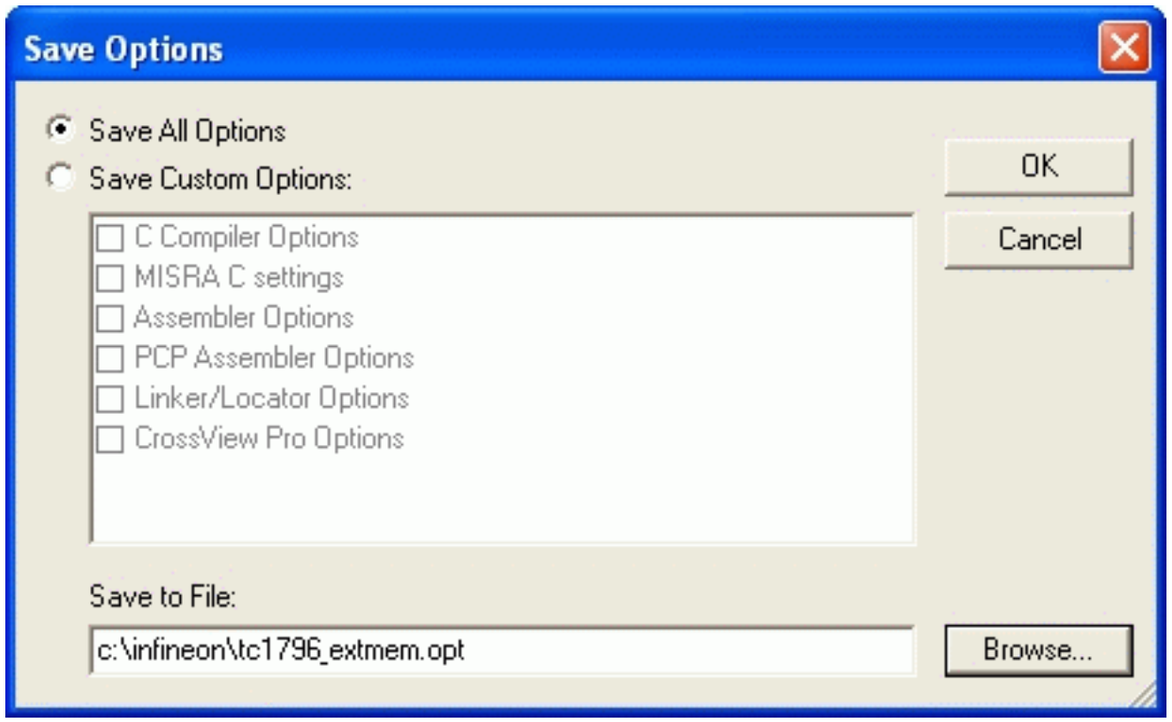
15. Build the application

Click the **Build** icon on the Build toolbar. The Build process finishes successfully.

Exercise 2: LED blink

16. Save the project options

Save the project option in a file by choosing **Project > Save Options** and save the options as `c:\infineon\tc1796_extmem.opt`. These options will be used in later exercises again.



17. Add the user code

Add a delay function implemented as a macro to `MAIN.h` at (MAIN_Header,3)

```
// USER CODE BEGIN (MAIN_Header,3)
#define delay( _milliseconds )  \
  {for (unsigned int ii=0; ii < (_milliseconds * 150000/2); ii++) { __nop(); __nop(); }}
// USER CODE END
```

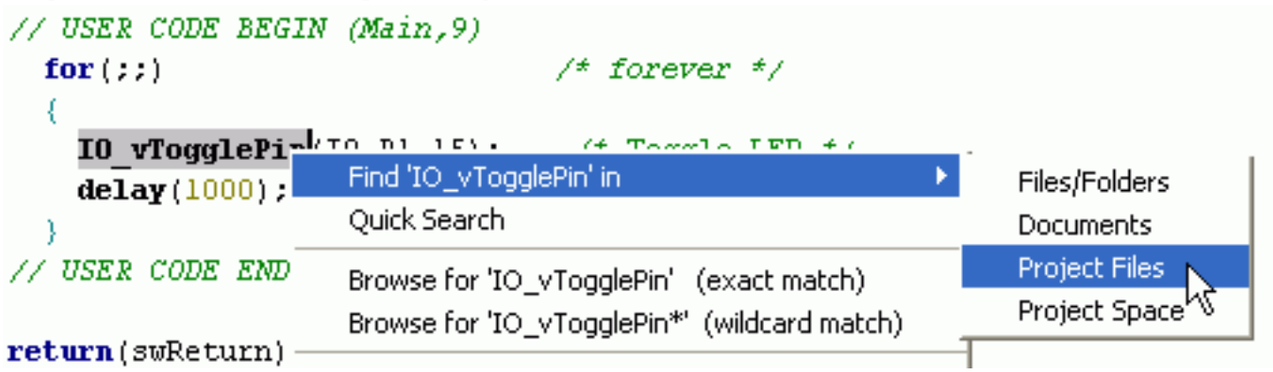
The two `__nop()` ("No operations") are necessary to halt the program during the execution of the delay functions. Some debugger has problems with too short loops.

Exercise 2: LED blink

Add the following code to `MAIN.c` at (Main,9)

```
// USER CODE BEGIN (Main,9)
for(;;) {
  IO_vTogglePin(IO_P1_15);    // Toggle LED
  delay(1000);                // 1 second delay
}
// USER CODE END
```

To browse through your source code, e.g. searching for the definition of `IO_vTogglePin` select the function with the mouse and click the right mouse button. The context menu offers you a quick way to look for the selected element in all Project Files or Project Space.



18. Build the application

Click the **Build** icon on the Build toolbar. The Build process finishes successfully.

19. Debug the application

Click the **Debug** icon on the Build toolbar to open the *CrossView Pro* debugger.

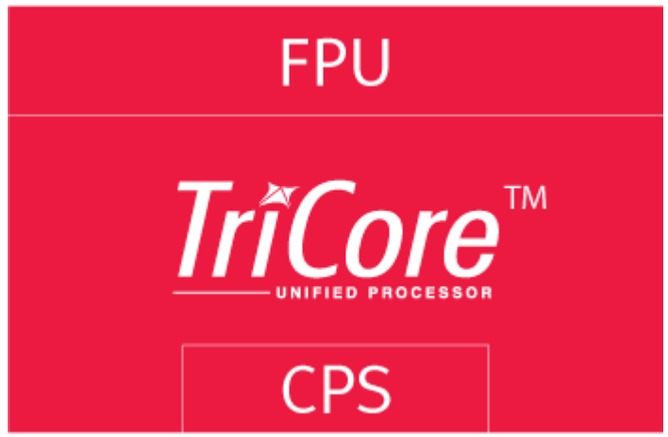
20. Run the application

Click the **Run** icon on the *CrossView Pro* toolbar. See the LED blinking on the TriBoard.
Choose **File > Exit**. A dialog appears. Confirm **Exit**.

Exercise 3: Saturation

Saturation

The TriCore™ Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computational power of a DSP, and the high-performance/price feature of a RISC load/store architecture onto a compact, reprogrammable core. One of the key features is a saturating integer arithmetic. With the *Tasking* compiler this feature is accessible as a C language extensions. This exercise shows how saturating integer arithmetic can be used to dramatically improve execution speed. To learn more about the instruction set, you should read the TriCore™ Architectural Manual.
Infineon also offers a highly optimized DSP library called *TriLib* for free.



Exercise 3: Saturation

Paris, 19 July 1996

ARIANE 5
Flight 501 Failure
Report by the Inquiry Board

FOREWORD
On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.
...
2. ANALYSIS OF THE FAILURE
...

- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

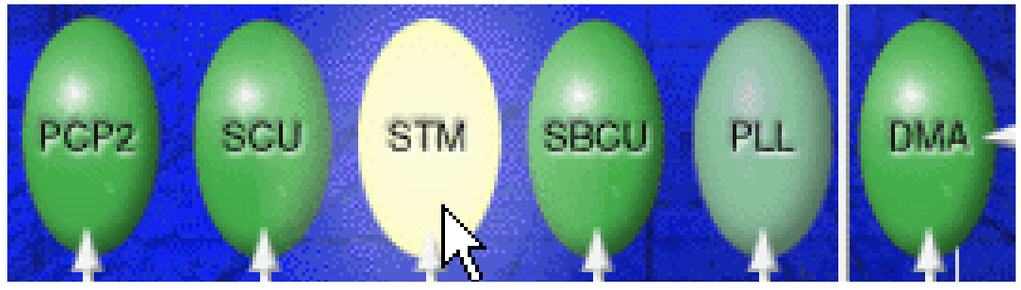
Exercise 3: Saturation

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\sat`. Copy the `led.dav` file from the previous exercise to the new directory and rename the file to `sat.dav`.

2. Open the System Timer Properties

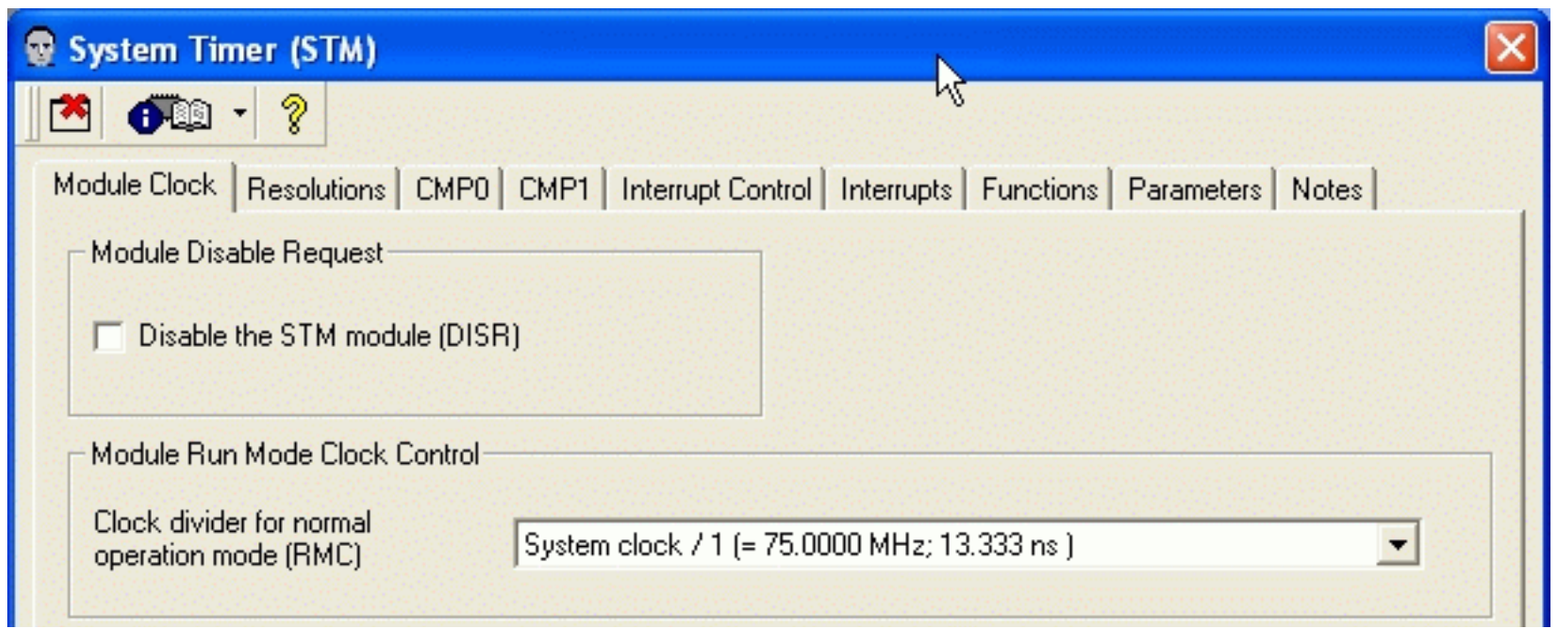
Start *DAvE* and open the `sat.dav` project file. Click on **STM** in the project window to open the STM properties.



3. Configure the System Timer (STM)

On the **Module Clock** page

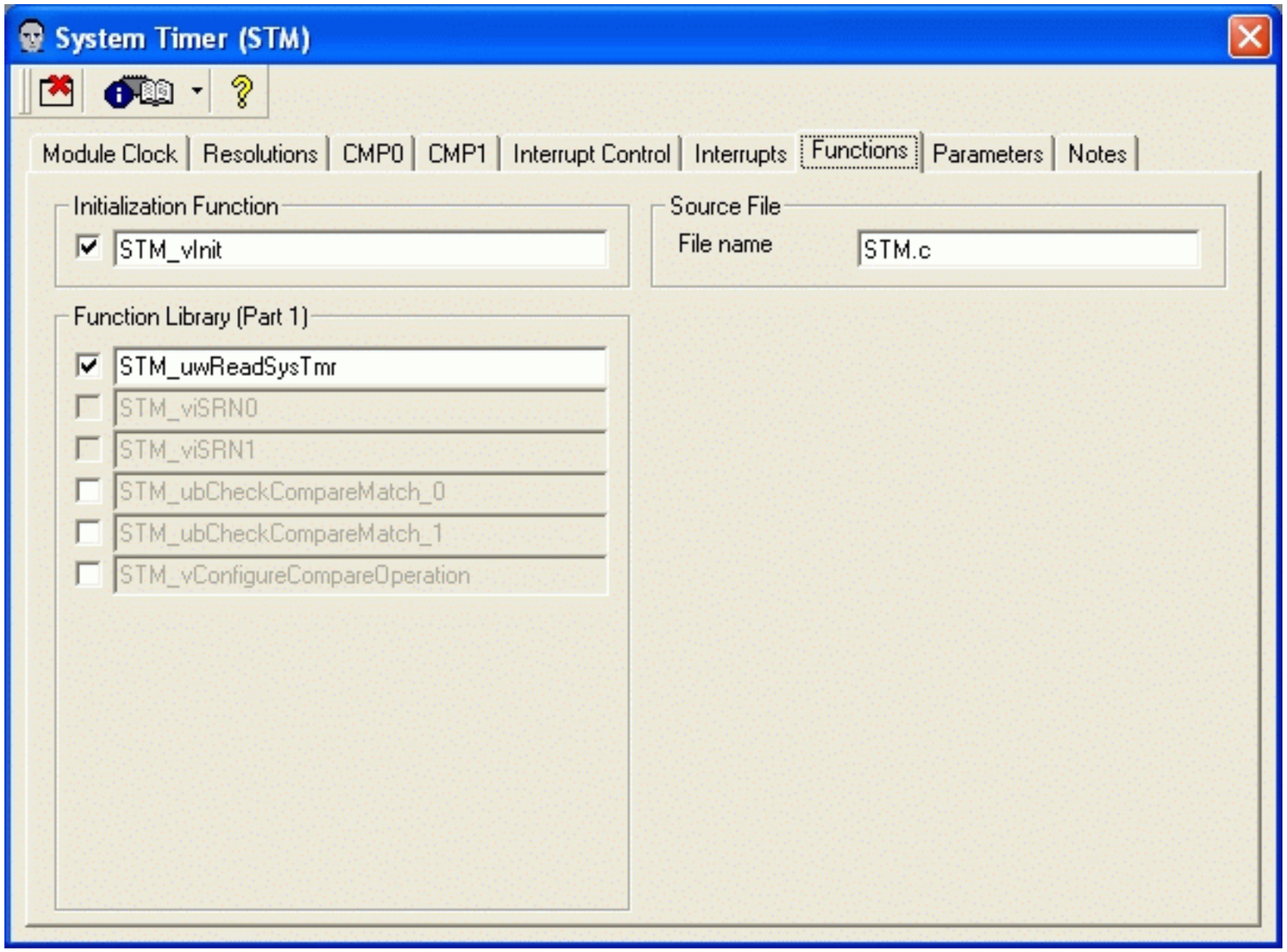
- Uncheck **Disable the STM module (DISR)**
- Choose Module Run Mode Clock Control **System clock / 1**.



Exercise 3: Saturation

On the **Functions** page

■ Check the initialization function `STM_vInit` and the read timer function `STM_uwReadSysTmr`.

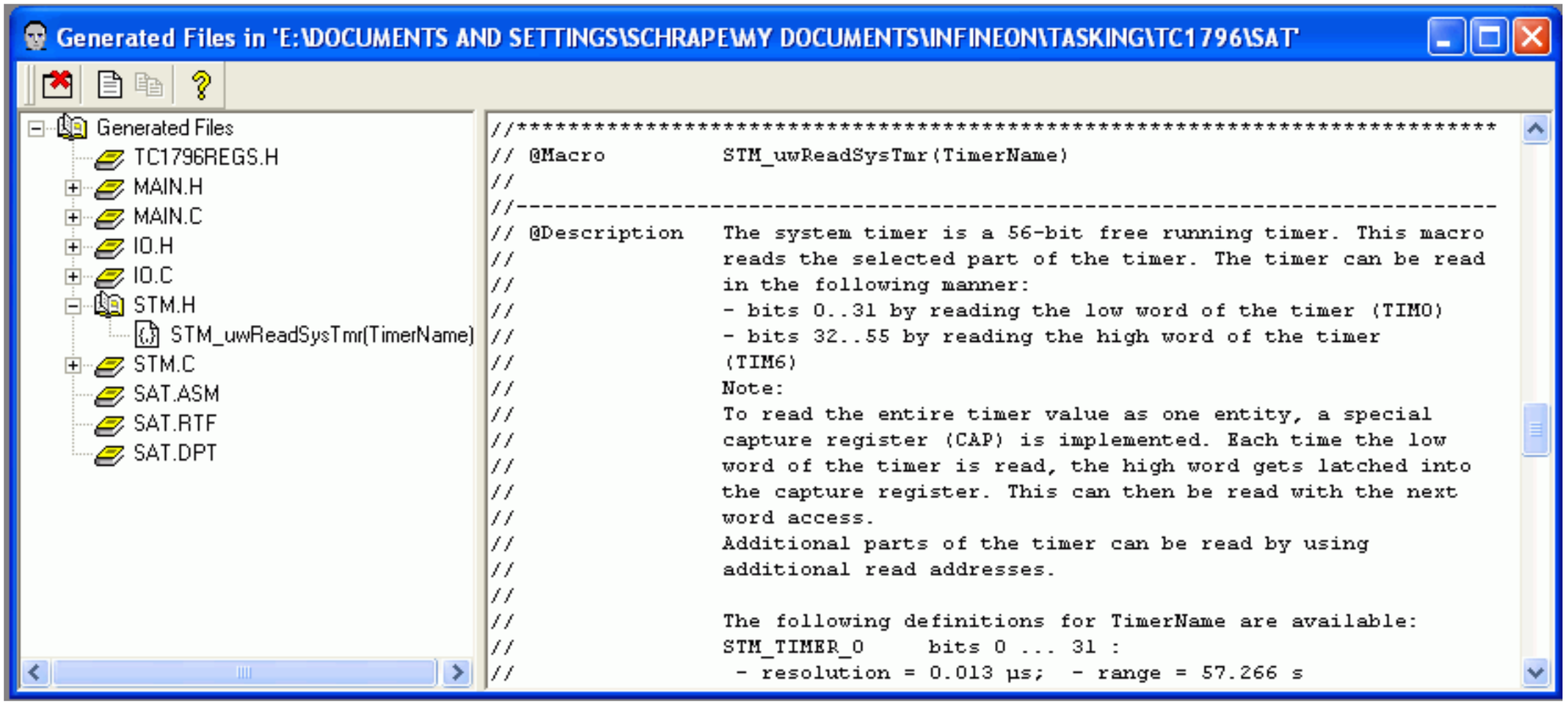


Click the **Close** icon on the dialog toolbar to close the dialog.

Exercise 3: Saturation

4. Generate the application framework

Click the **Generate Code** icon  on the application toolbar to start the code generation process.



Save and Close the *DAvE* project.

5. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\sat\sat.pjt`.

Exercise 3: Saturation

6. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

7. Set current project

Use the context menu in the workspace window to make the sat project the current project.

8. Load the project options

This exercise uses the internal memory. The appropriate option file can be found in the `tc1796.zip` package. Copy the `tc1796_intmem.opt` to `c:\infineon`. For a description see Appendix C. Choose **Project > Load Options**. In the **File-name** field enter `tc1796_intmem.opt`.

9. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

10. Add the user code

Three different solution to handle saturation during an addition are considered.
Solution 1 is an ANSI C, plausibility check.

```
long x1,x2,y;

y = x1 + x2;
if ((x1 >= 0) && (x2 >= 0) && (y < 0))
  y = LONG_MAX;
else if ((x1 < 0) && (x2 < 0) && (y >= 0))
  y = LONG_MIN;
```

Exercise 3: Saturation

Solution 2 is ANSI C but uses a 64-bit word width.

```
long long dummy;

dummy = (long long)x1 + (long long)x2;
if (dummy > LONG_MAX)
  y = LONG_MAX;
else if (dummy < LONG_MIN)
  y = LONG_MIN;
else
  y = (long) dummy;
```

Solution 3 uses a *Tasking* language extension that directly maps to a TriCore™ instruction.

```
y = (__sat long)x1 + (__sat long)x2;
```

Exercise 3: Saturation

The execution speed of each solution is measured with the system timer function `STM_uwReadSysTmr`. The result is printed to a *CrossView Pro* terminal window. Add the following code to `MAIN.c` at (Main,9).

```
// USER CODE BEGIN (Main,9)
unsigned long dt,i;
volatile long x1, x2, y;
const int N = 1000;       // Each calculation is done N times
                          // to measure the execution time precisely
x1 = 6; x2 = LONG_MAX;    // Initialize x1, x2 to numbers so that
                          // the sum exceeds LONG_MAX

// Solution 1: ANSI C, plausibility check
dt = STM_uwReadSysTmr(STM_TIMER_0);
for (i=0; i<N; i++) {
  y = x1+x2;
  if ((x1>=0) && (x2>=0) && (y<0))
    y = LONG_MAX;
  else if ((x1<0)&&(x2<0)&&(y>=0))
    y = LONG_MIN;
}
dt = STM_uwReadSysTmr(STM_TIMER_0) - dt;
printf("Solution 1: %.1f CPU cycles\n", 2.*dt/N);
```

Exercise 3: Saturation

```
// Solution 2: ANSI C, increase of word width
long long dummy;
dt = STM_uwReadSysTmr(STM_TIMER_0);
for (i=0;i<N;i++) {
  dummy = (long long)x1+(long long)x2;
  if (dummy > LONG_MAX)
    y = LONG_MAX;
  else if (dummy < LONG_MIN)
    y = LONG_MIN;
  else
    y = (long) dummy;
}
dt = STM_uwReadSysTmr(STM_TIMER_0) - dt;
printf("Solution 2: %.1f CPU cycles\n", 2.*dt/N);
```

```
// Solution 3: Language extension, saturation type
dt = STM_uwReadSysTmr(STM_TIMER_0);
for (i=0;i<N;i++) {
  y = (__sat long)x1 + (__sat long)x2;
}
dt = STM_uwReadSysTmr(STM_TIMER_0) - dt;
printf("Solution 3: %.1f CPU cycles\n", 2.*dt/N);
// USER CODE END
```

`LONG_MAX`, `LONG_MIN` are macros of the C standard library. Include `limits.h` and `stdio.h` in `MAIN.c` at (MAIN_General,2).

11. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

12. Debug the application

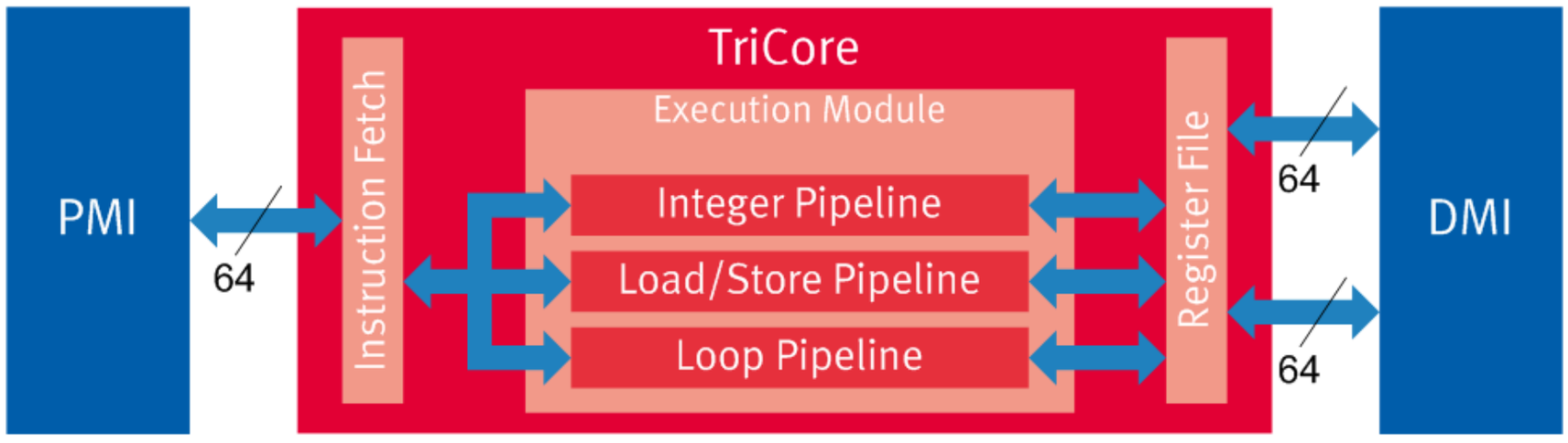
Click the **Debug** icon  on the Build toolbar to open the *CrossView Pro* debugger.

Exercise 3: Saturation

13. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar. The *CrossView* terminal window will show the result:

Solution 2 executes only slightly faster than solution 1, but with the language extension used in solution 3 the execution speed is much faster. The 5 instructions used in solution 3 - two loads, add with saturation, store and loop - take up just 3 cycles because of parallel execution using the 3 pipelines in the TriCore™.



Exercise 4: Interrupts

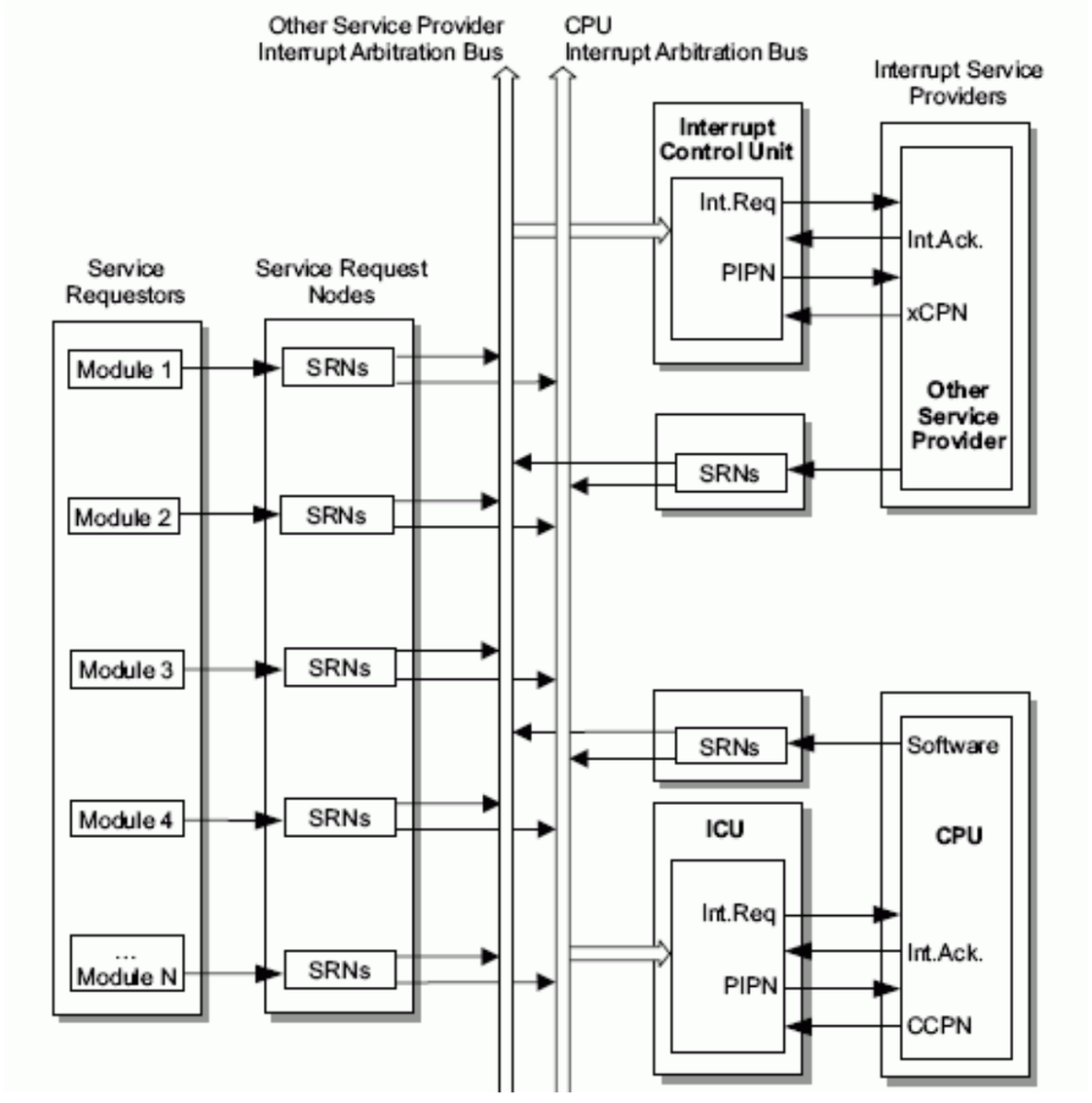
Interrupts

A key feature of the TriCore™ architecture is its powerful and flexible interrupt system. The interrupt system is built around programmable Service Request Nodes (SRNs). Conventional architectures handle service requests by loading a new Program Status (PS) from a vector table in data memory. With the TriCore™ architecture however, service requests jump to vectors in code memory. This procedure reduces response time for service requests. The first instructions of the Interrupt Service Routine (ISR) execute at least three cycles earlier than they would otherwise. This exercise shows that programming interrupt services with TriCore™ using *DAvE* is quite simple.

Exercise 4: Interrupts

Features

- In a TriCore™ system, multiple sources such as peripherals or external inputs can generate an interrupt signal to the CPU or another provider to request for service.
- Each interrupt or service request from a module connects to a Service Request Node, containing a Service Request Control Register (SRC). Interrupt arbitration busses connect the SRNs with the interrupt control units of the service providers. These control units handle the interrupt arbitration and communication with the service provider.
- A peripheral can have several service request lines, with each one of them connecting to its own individual SRN.
- The 8-bit Service Request Priority Number (SRPN) of a service request, indicates its priority with respect to other sources requesting an interrupt to the same service provider, and to the priority of the service provider itself.
- The interrupt arbitration scheme permits up to 255 sources to be active at one time.
- The Interrupt Control Unit (ICU) manages the interrupt system and arbitrates incoming interrupt requests to find the one with the highest priority and to determine whether or not to interrupt the service provider.
- Several conditions could block the CPU from immediately responding to the interrupt request:
  - The interrupt system is globally disabled
  - The current CPU priority CCPN, is equal to or higher than the Pending Interrupt Priority Number (PIPN).
  - The CPU is in the process of entering an interrupt or trap service routine.
  - The CPU is operating on non-interruptible trap services, a multi-cycle instruction.
- The interrupt response time for a system with e.g. 15 interrupts is 9 cycles. 2 arbitration are 1 priority check are done in parallel to CPU operation. 2 context save, 1 acknowlege to ICU, 3 for fetching and jumping to the first instruction.



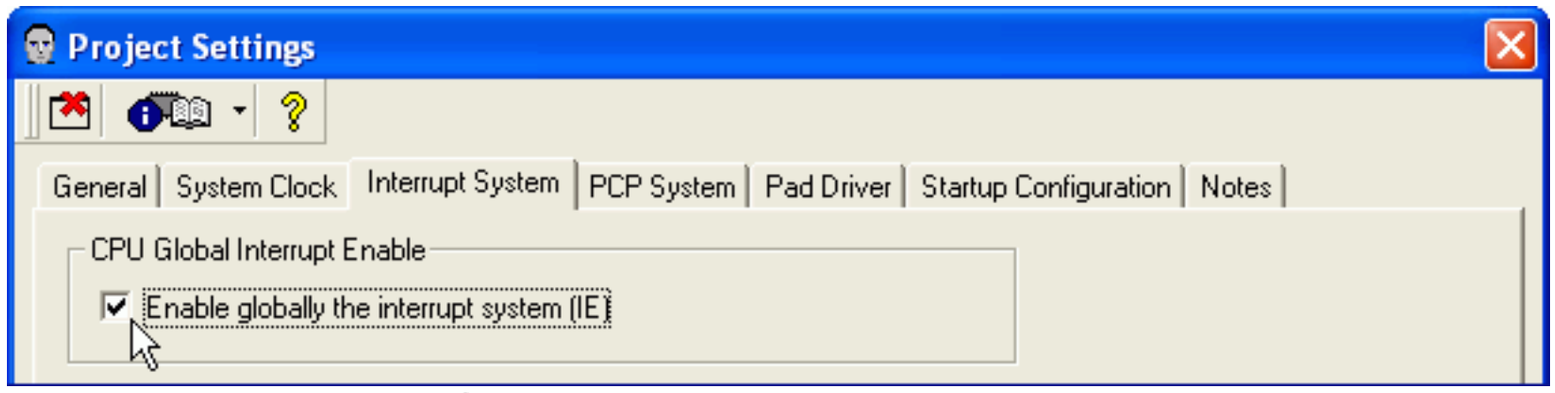
Exercise 4: Interrupts

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\isr`. Copy the `sat.dav` file from the previous exercise to the new directory and rename the file to `isr.dav`.

2. Modify the Project Properties

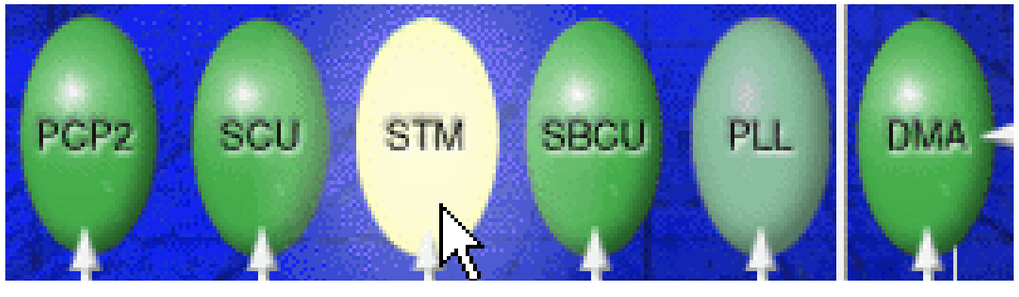
Start *DAvE* and open the `isr.dav` project file. Choose **File > Project Settings**.
On the **Interrupt System** page
■ Check **Enable globally the Interrupt System (IE)**.



Click the **Close** icon on the dialog toolbar to close the dialog.

3. Open System Timer Properties

Click on **STM** in the project window to open the STM properties.

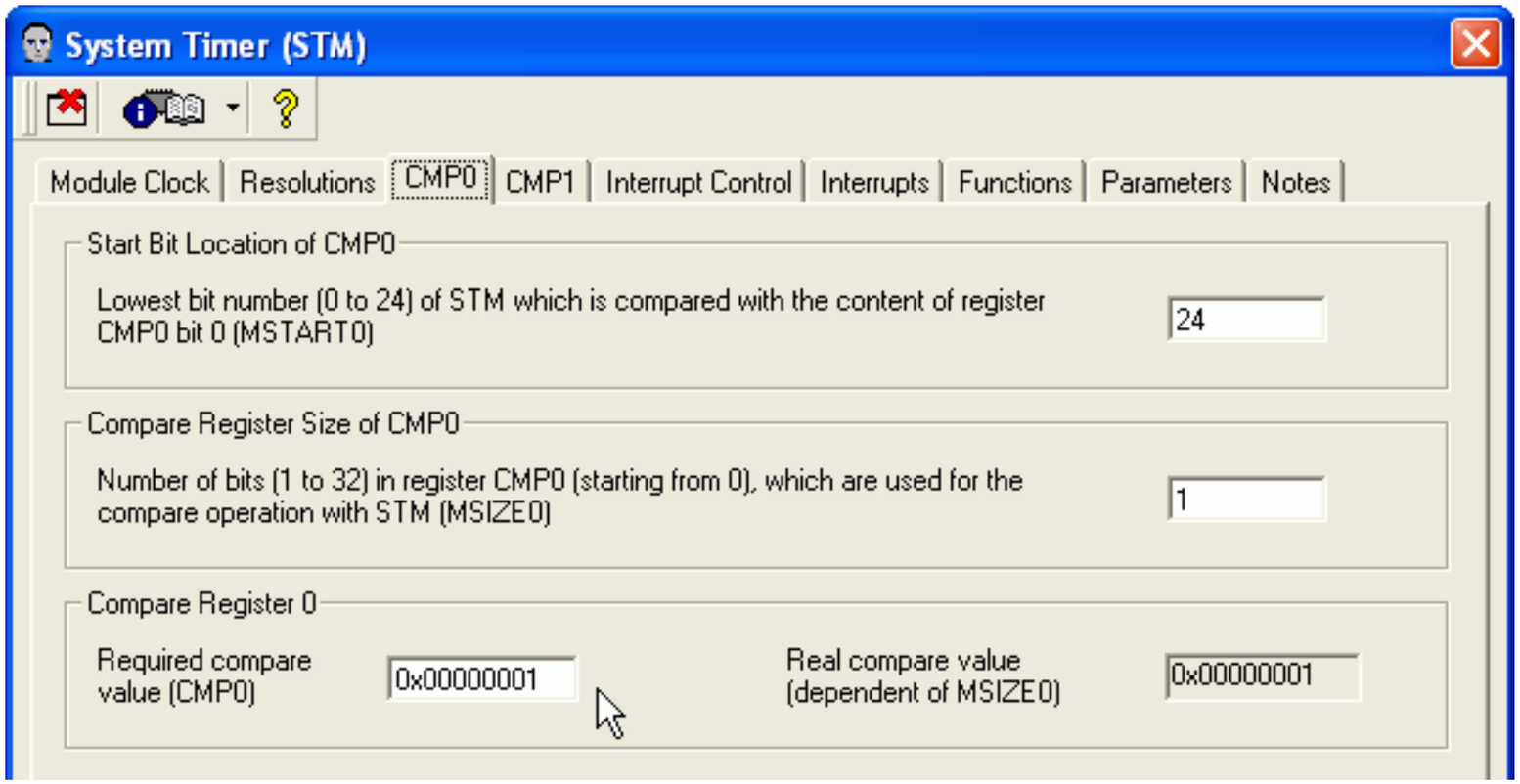


Exercise 4: Interrupts

4. Modify the System Timer (STM) configuration

On the **CMP0** page

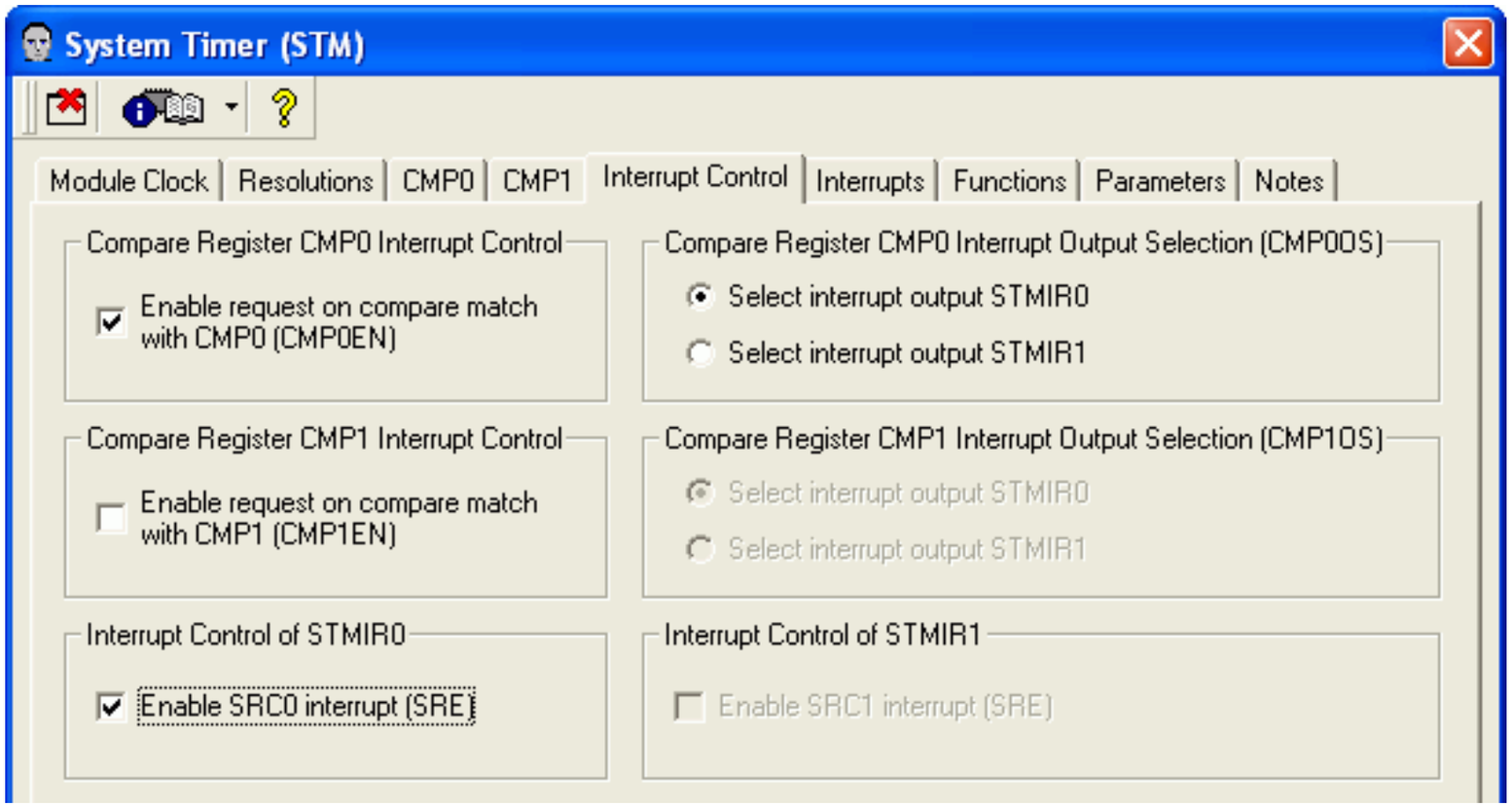
- At **Start Bit Location of CMP0** enter 24 and type **Return**,
- At **Compare Register Size of CMP0** enter 1 and type **Return**,
- At **Compare Register 0** enter 1 and type **Return**. The free running timer sets bit 24 every 2^{24} timer ticks, thus issuing a compare event every ~0.223 s ($=2^{24}/f_{STM}$). Decimal numbers entered in the text field are converted automatically to hexadecimal.

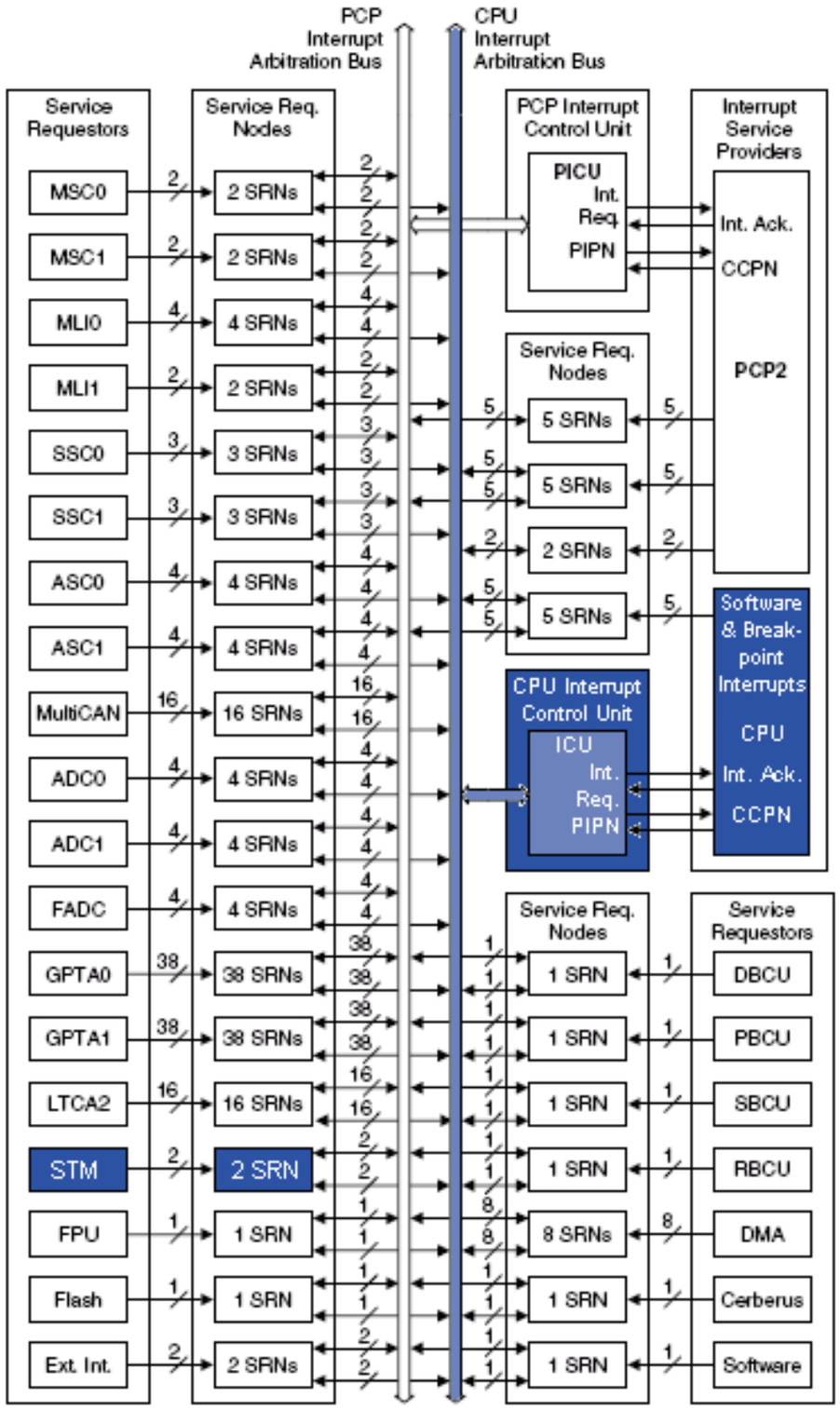


Exercise 4: Interrupts

On the **Interrupt Control** page

- Check **Enable request on compare match with CMP0 (CMP0EN)**,
- Check **Enable SRC0 interrupt (SRE)**,
- Select **Select interrupt output STMIR0**.

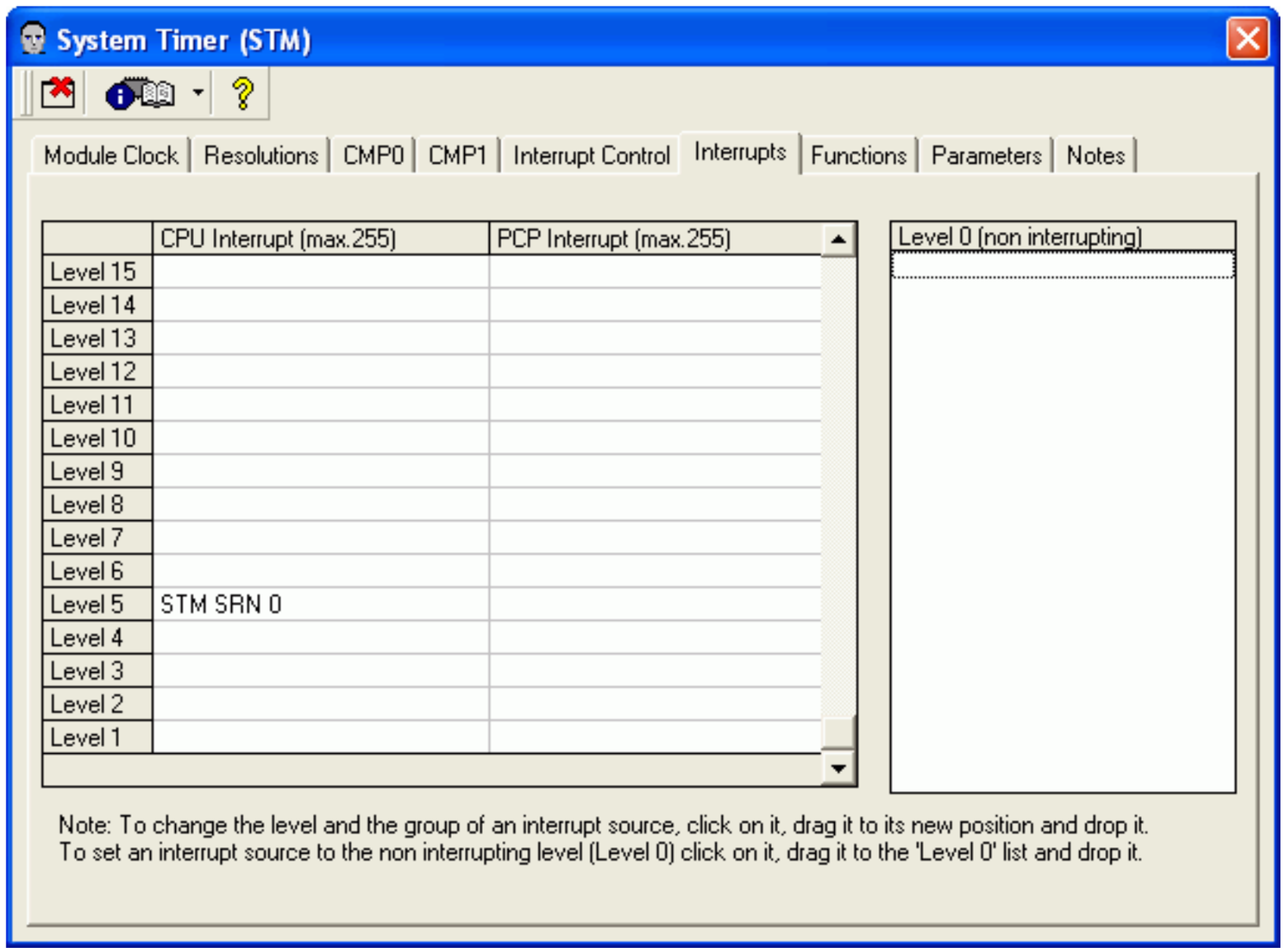




Exercise 4: Interrupts

5. Set up System Timer Interrupt

On the **Interrupts** page drag the **STM SRN 0** interrupt from the right list to the CPU Interrupt level 5.



Click the **Close** icon on the dialog toolbar to close the dialog.

Exercise 4: Interrupts

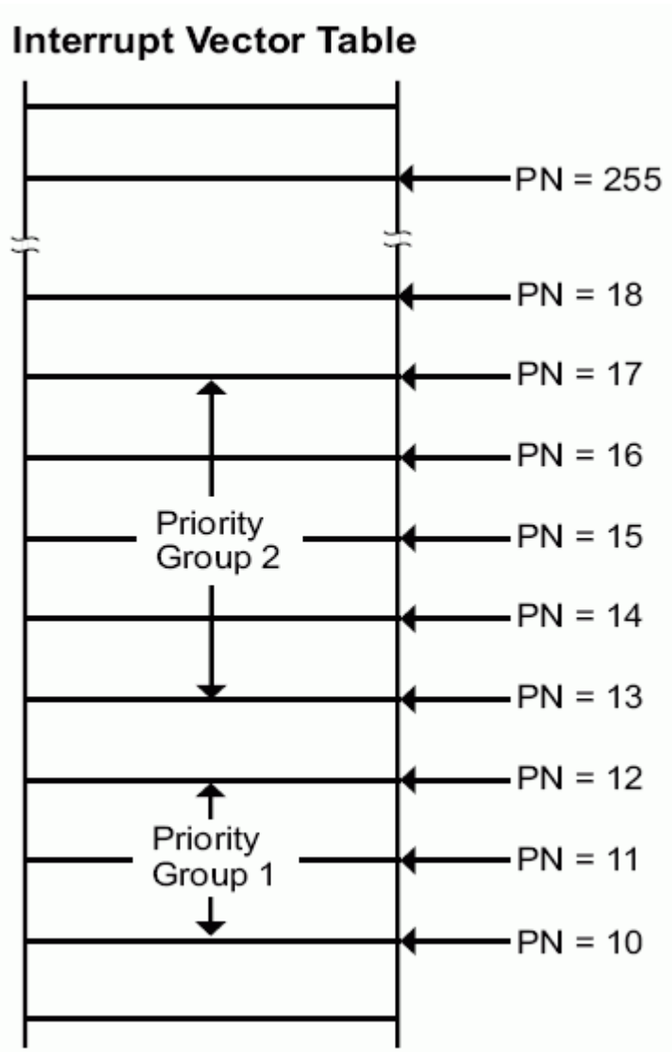
The maximum number of interrupt sources is 255. Programmable options range from one priority level with 255 sources, up to 255 priority levels with one source each. During the execution of an interrupt service routine, the system blocks the CPU from taking further interrupt requests. You can immediately re-enable the system to accept interrupt requests:

```
__interrupt(vector) __enable_ isr( void )
```

The compiler generates an enable instruction as first instruction in the routine. The enable instruction sets the interrupt enable bit (ICR.IE) in the interrupt control register. The function qualifier `__bisr_()` also re-enables the system to accept interrupt requests. In addition, the current CPU priority number (CCPN) in the interrupt control register is set:

```
__interrupt(vector) __bisr_(CCPN) isr( void )
```

The argument CCPN is a number between 0 and 255. The system accepts all interrupt requests that have a higher pending interrupt priority number (PIPN) than the current CPU priority number. So, if the CPU priority number is set to 0, the system accepts all interrupts. If it is set to 255, no interrupts are accepted.
The compiler generates a bisr instruction as first instruction in the routine. The bisr instruction sets the interrupt enable bit (ICR.IE) and the current CPU priority number (ICR.CCPN) in the interrupt control register.
If less priority level are sufficient than the number of arbitration cycles can be reduced which also reduce the interrupt latency. See the **File > Project Settings** page **Interrupt System**.



6. Generate the application framework

Click on the **Generate Code** icon  on the application toolbar to start the code generation process. Save and close the *DAvE* project.

7. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\isr\isr.pjt`.

8. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

9. Set current project

Use the context menu in the workspace window to make the isr project the current project.

10. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter `c:\infineon\tc1796_extmem.opt`.

11. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

Exercise 4: Interrupts

12. Add the user code

Add an endless loop in `MAIN.c` at (Main,9) and add the following code to the interrupt routine in `STM.c`.

```
void __interrupts (STM_INT0) STM_viSRN0(void)
{
  // USER CODE BEGIN (SRN0,2)
  // USER CODE END

  if(STM_ICR_CMP0IR == 1)  // if compare match of CMP0 is pending
  {
    // USER CODE BEGIN (SRN0,3)
    IO_vTogglePin(IO_P1_15);
    // USER CODE END
    STM_ISRR_CMP0IRR = 1;  // clear request bit of CMP0
  }

  if(STM_ICR_CMP1IR == 1)  // if compare match of CMP1 is pending
  {
    // USER CODE BEGIN (SRN0,4)
    // USER CODE END
    STM_ISRR_CMP1IRR = 1;  // clear request bit of CMP1
  }

  // USER CODE BEGIN (SRN0,5)
  // USER CODE END

} //  End of function STM_viSRN0
```

13. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

Exercise 4: Interrupts

14. Debug the application

Click the **Debug** icon  on the Build toolbar to open the *CrossView Pro* debugger.

15. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar. See the LED blinking. Exit *CrossView*.

Exercise 5: ASC

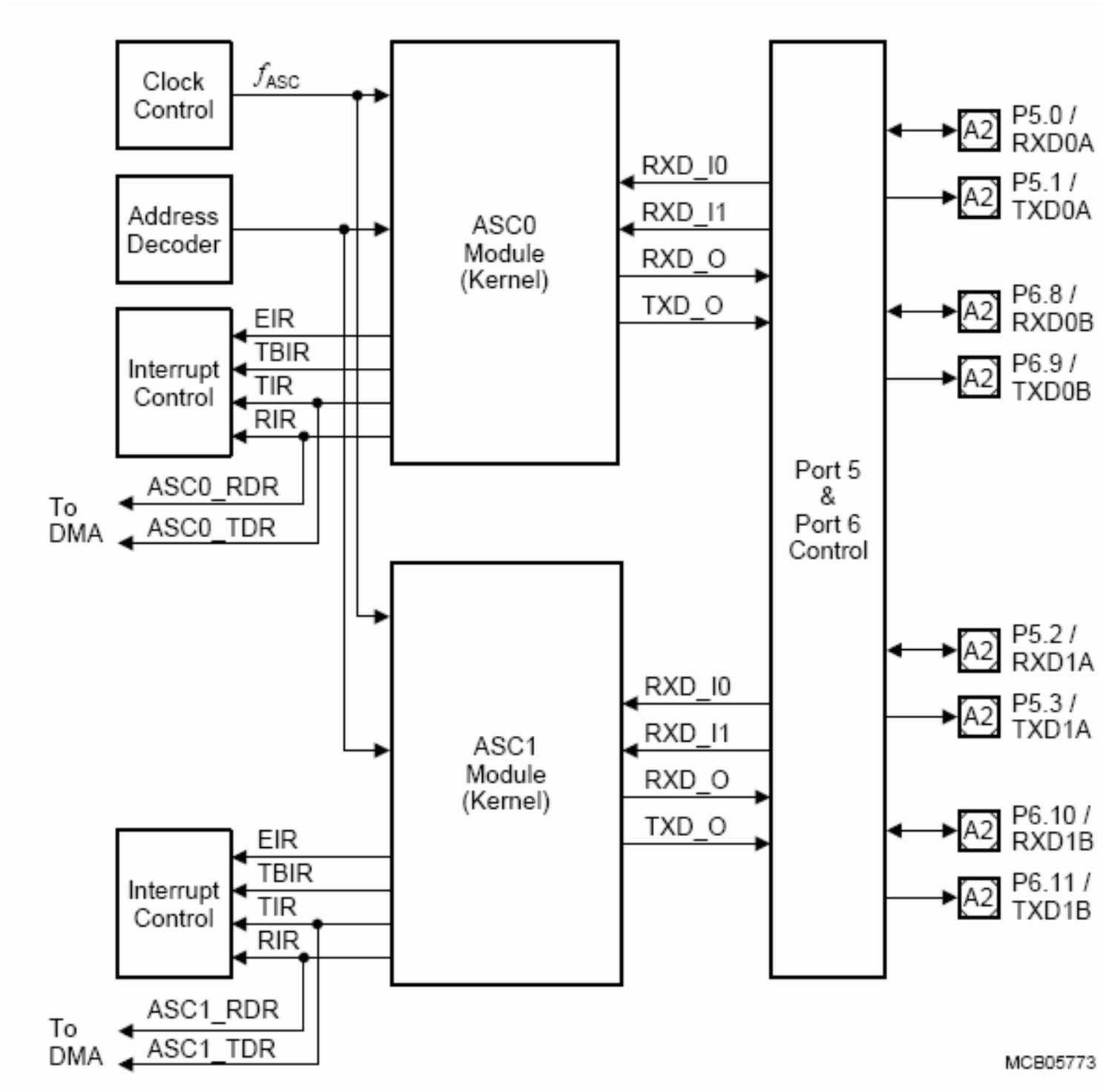
ASC

In this exercise you will develop an application that writes a strings to the serial port.
The Asynchronous/Synchronous Serial Interfaces provide serial communication between the TC1796 and other microcontrollers, microprocessors, or external peripherals. The ASC supports full-duplex asynchronous communication and half-duplex synchronous communication. In Synchronous Mode, data is transmitted or received synchronous to a shift clock which is generated by the ASC internally. In Asynchronous Mode, 8-bit or 9-bit data transfer, parity generation, and the number of stop bits can be selected. Parity, framing, and overrun error detection are provided to increase the reliability of data transfers. Transmission and reception of data are double-buffered. For multiprocessor communication, a mechanism is included to distinguish address bytes from data bytes. Testing is supported by a loop-back option. A 13-bit baud rate generator provides the ASC with a separate serial clock signal which can be very accurately adjusted by a prescaler implemented as a fractional divider.
Each ASC module, ASC0 and ASC1, communicates with the external world via two I/O lines. The RXD line is the receive data input signal (in Synchronous Mode also output). TXD is the transmit output signal. In the TC1796, the two I/O lines of each ASC can be alternatively switched to different pairs of GPIO lines. Clock control, address decoding, and interrupt service request control are managed outside the ASC module kernel.

Exercise 5: ASC

Features

- Full-duplex asynchronous operating modes
  - 8-bit or 9-bit data frames, LSB first
  - Parity bit generation/checking
  - One or two stop bits
  - Baud rate: 4.69 MBit/s to 1.12 Bit/s @ 75MHz
  - Multiprocessor mode for automatic address/data byte detection
  - Loop-back capability
- Half-duplex 8-bit synchronous operating mode
  - Baud rate: 9.38 MBit/s to 763 Bit/s @ 75 MHz
- Double buffered transmitter/receiver
- Interrupt generation
  - On a transmit buffer empty condition
  - On a transmit last bit of a frame condition
  - On a receive buffer full condition
  - On an error condition



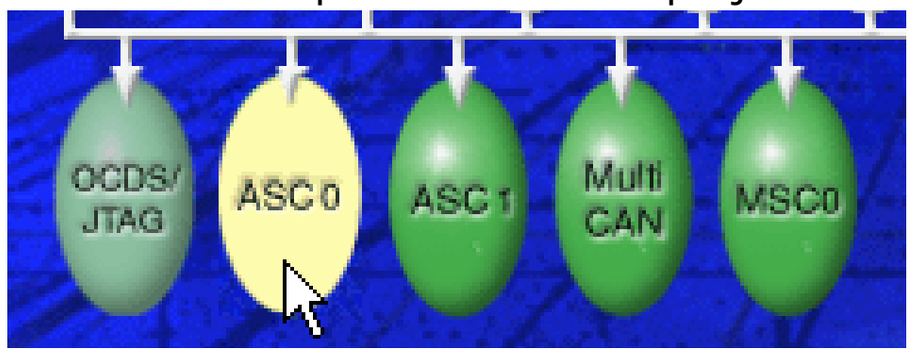
Exercise 5: ASC

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\asc`. Copy the `isr.dav` file from the previous exercise to the new directory and rename the file to `asc.dav`.

2. Open the ASC0 properties

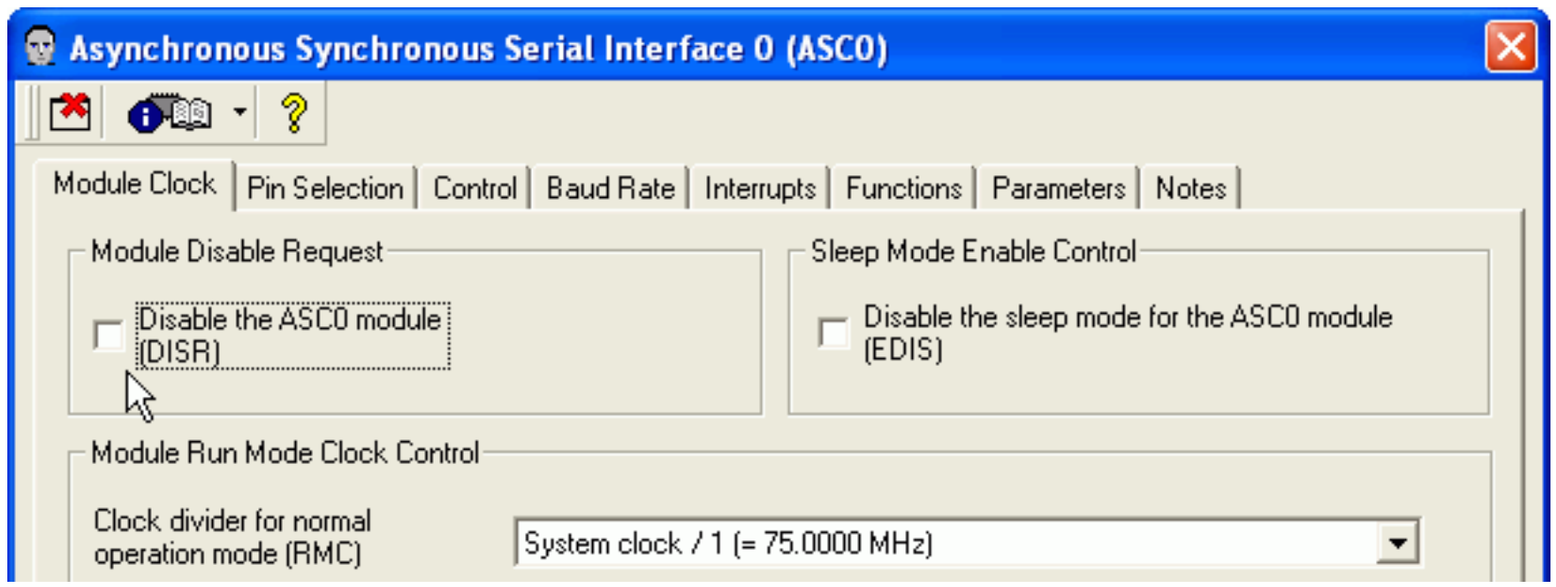
Start *DAvE* and open the `asc.dav` project file. Click **ASC0** in the project window to open the ASC0 properties.



3. Set up the ASC0 properties

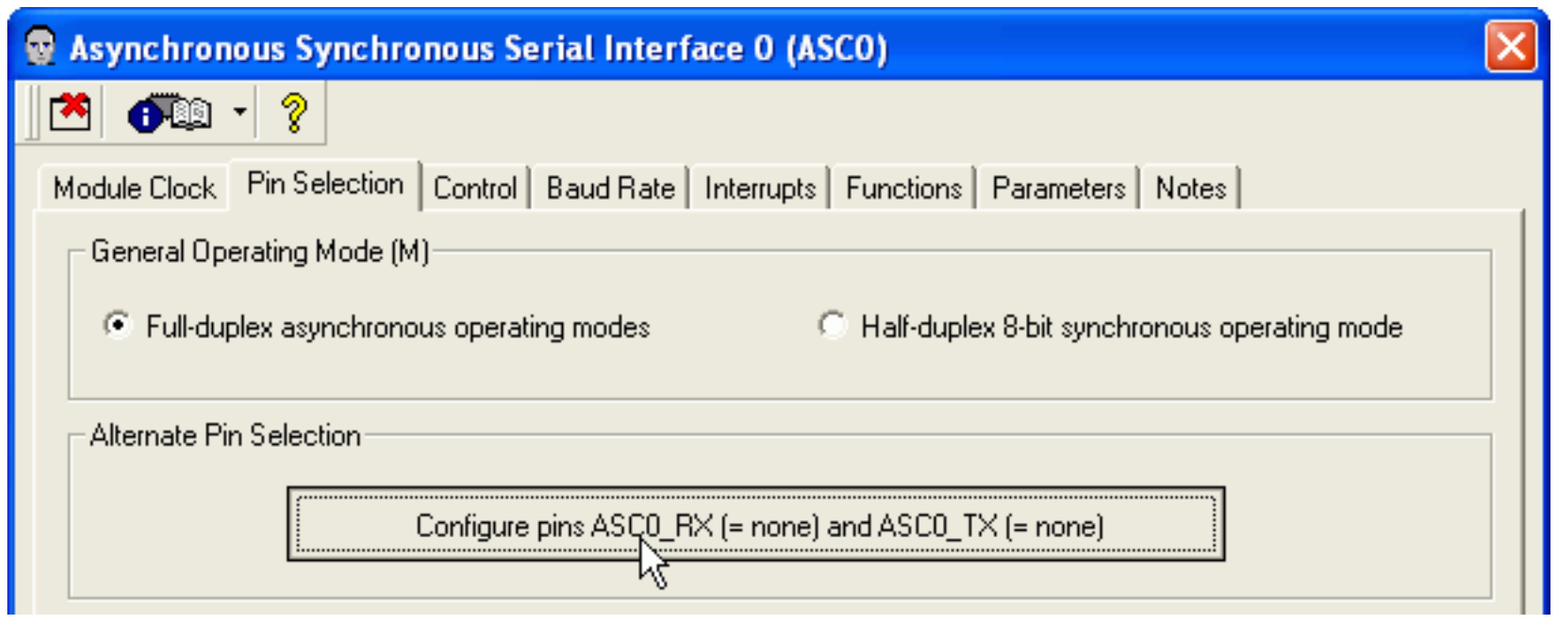
On the **Module Clock** page

- Uncheck **Disable the ASC0 module (DISR)**,
- Choose Module Run Mode Clock Control **System clock / 1**.



Exercise 5: ASC

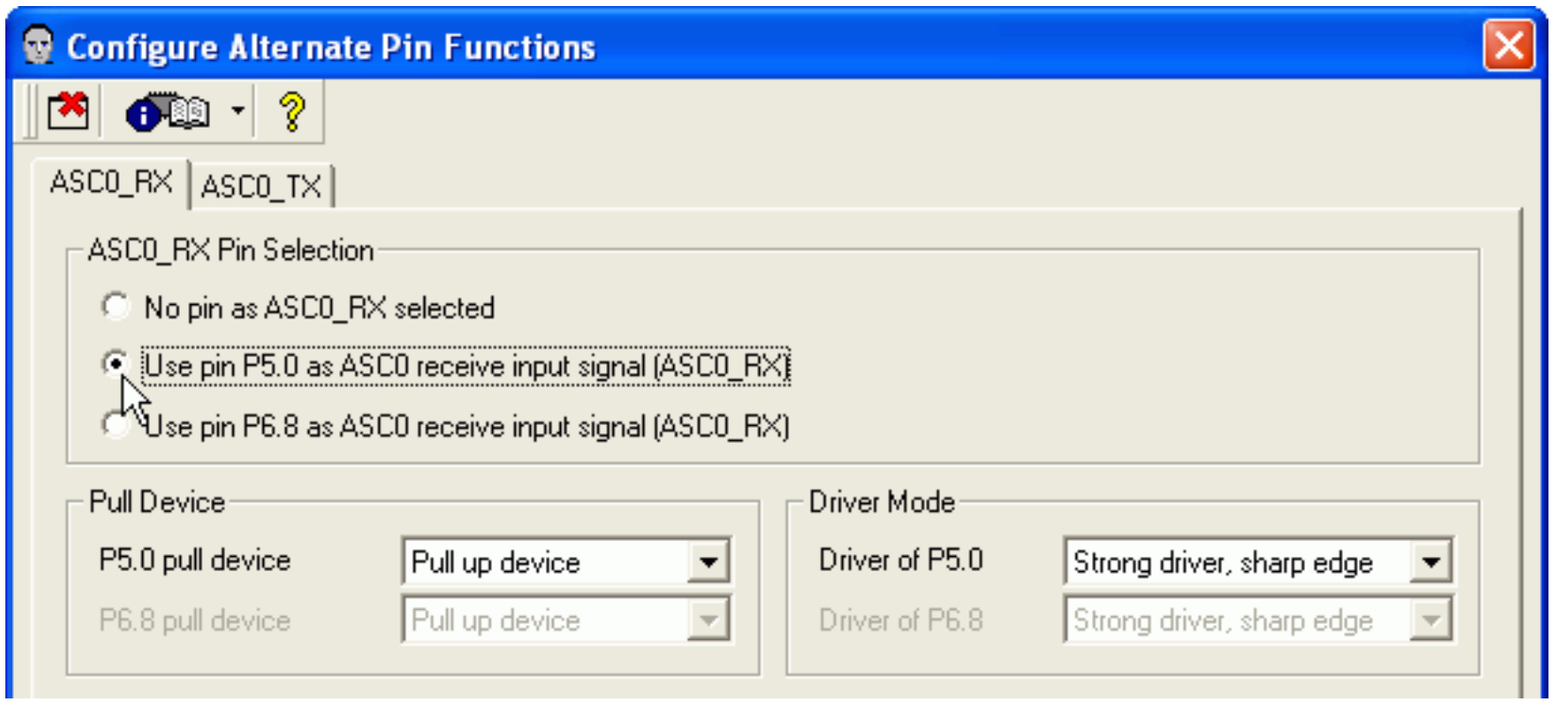
On the **Pin Selection** page click **Configure pins ASC0_RX and ASC0_TX**.



Exercise 5: ASC

On the **ASC0_RX** page

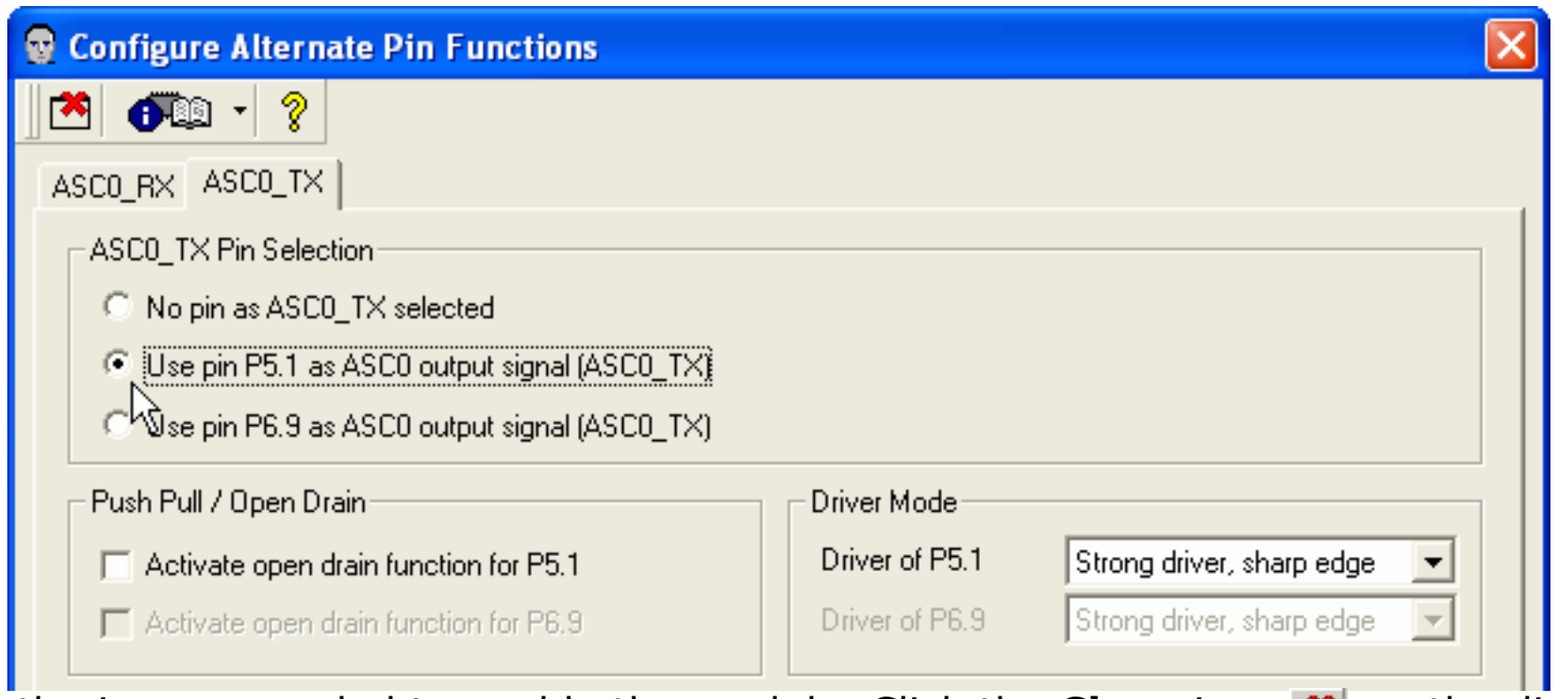
■ Select **Use pin P5.0 as ASC0 receive input signal (ASC0_RX)**.



Exercise 5: ASC

On the **ASC0_TX** page

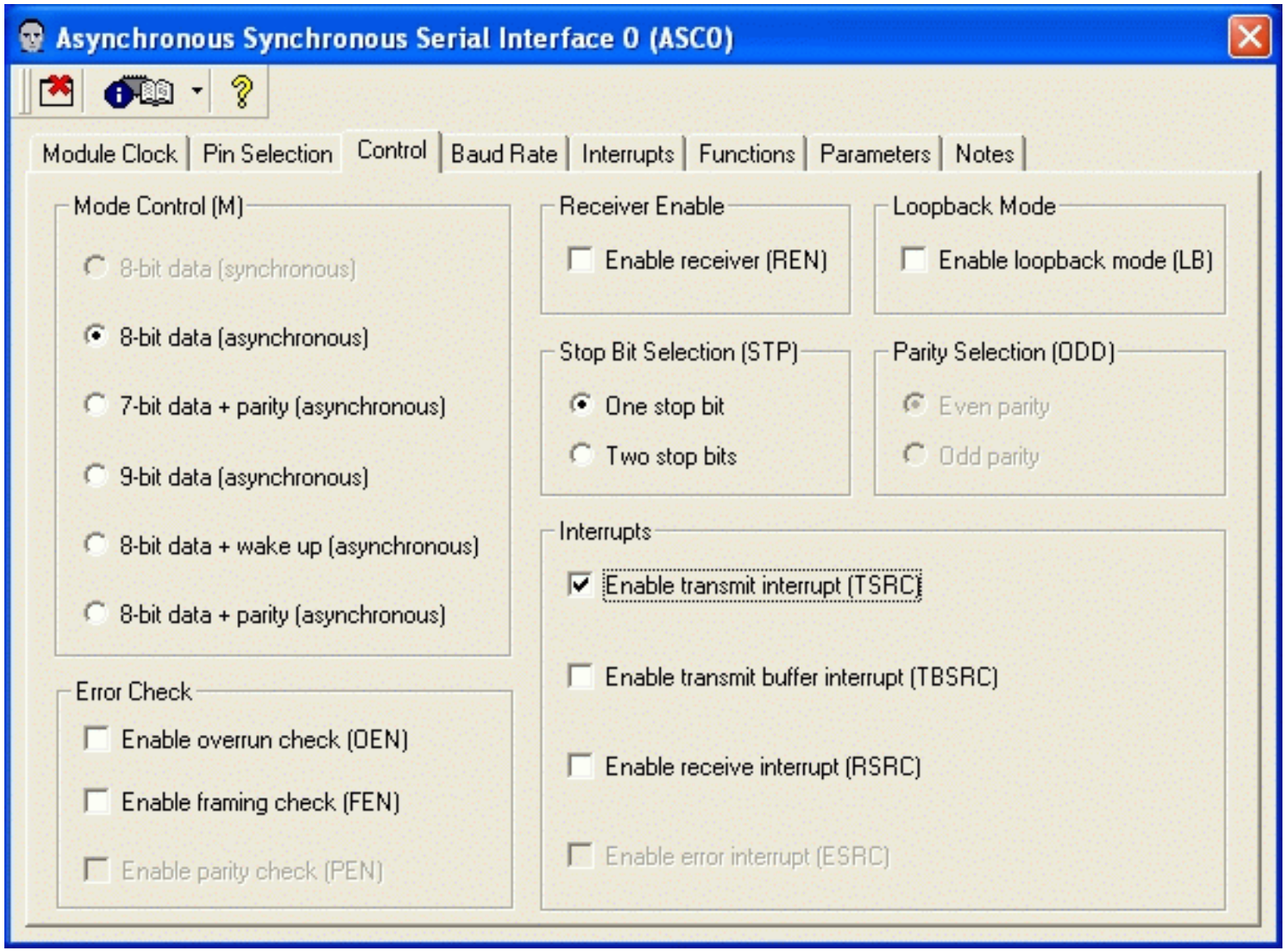
■ Select **Use pin P5.1 as ASC0 output signal (ASC0_TX)**.



Both pins are needed to enable the module. Click the **Close** icon on the dialog toolbar to close the dialog.

Exercise 5: ASC

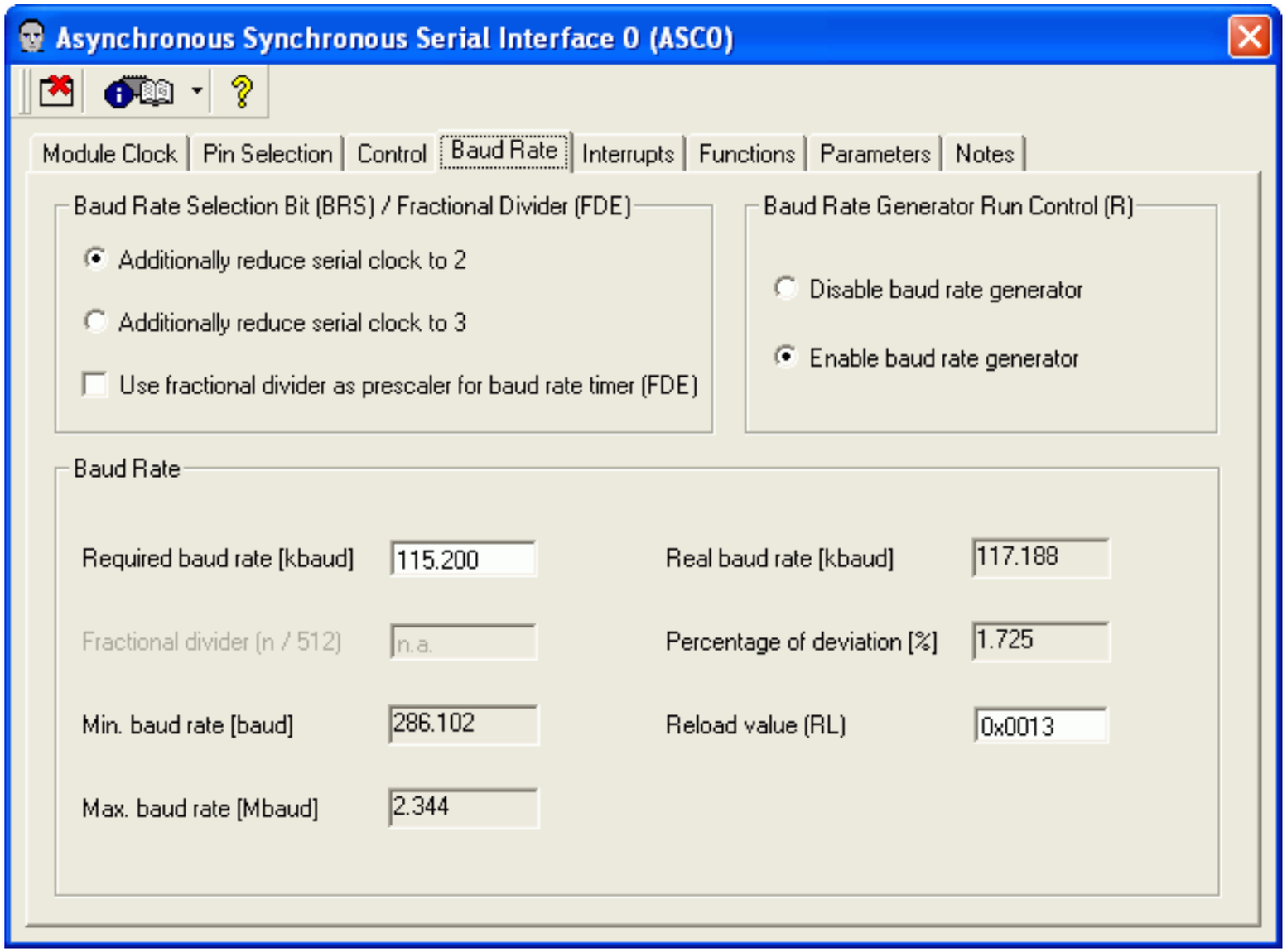
On the **Control** page
- Select Mode Control (M) **8-bit data (asynchronous)**,
- Select Stop Bit Selection (STP) **One Stop bit**,
- Check Interrupts **Enable transmit interrupts (TSRC)**.



Exercise 5: ASC

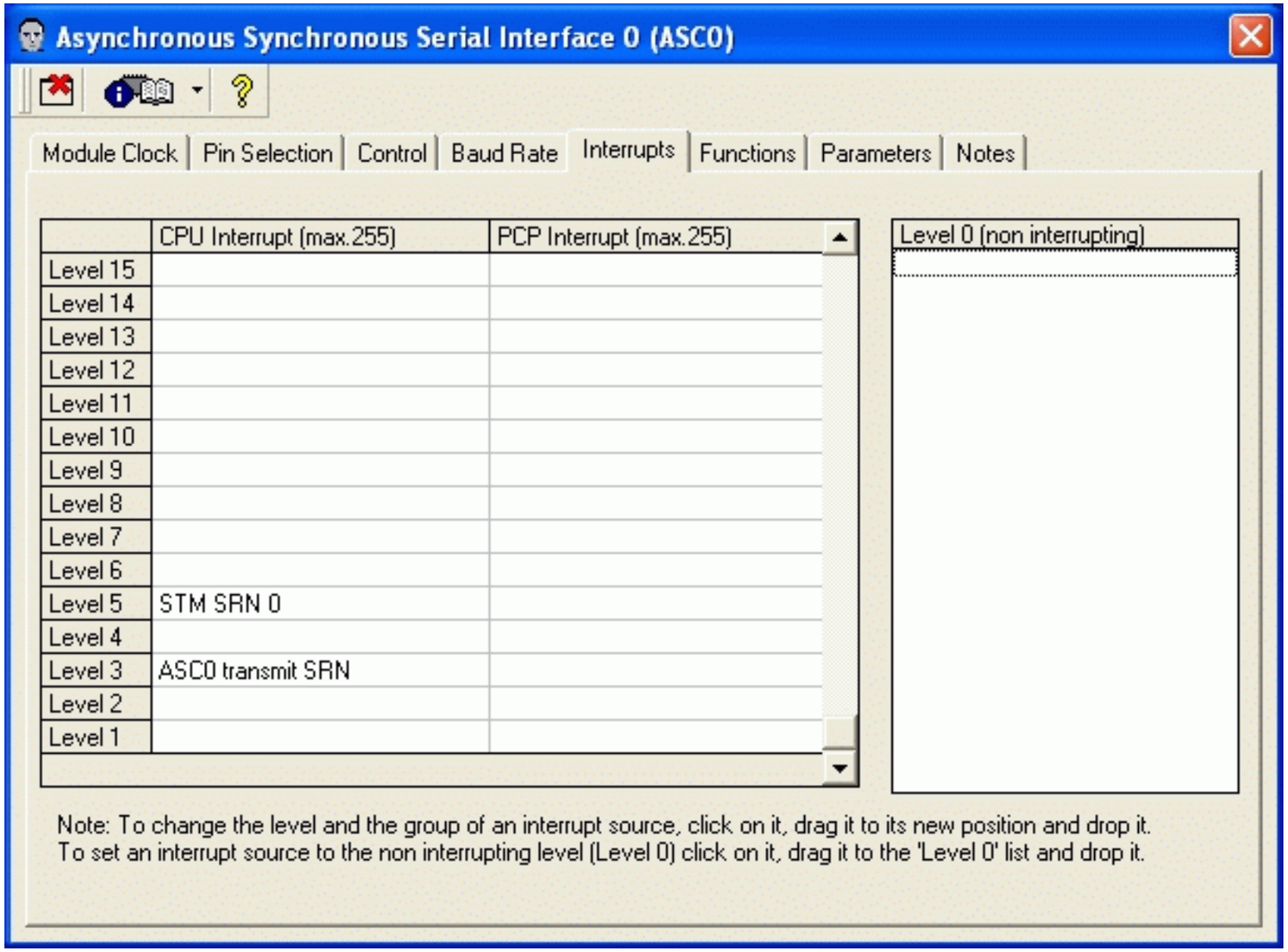
On the **Baud Rate** page

- Select Baud Rate Generator Run Control (R) **Enable baud rate generator**,
- At Baud Rate enter 115.000 as **Required Baud rate [kbaud]**.



Exercise 5: ASC

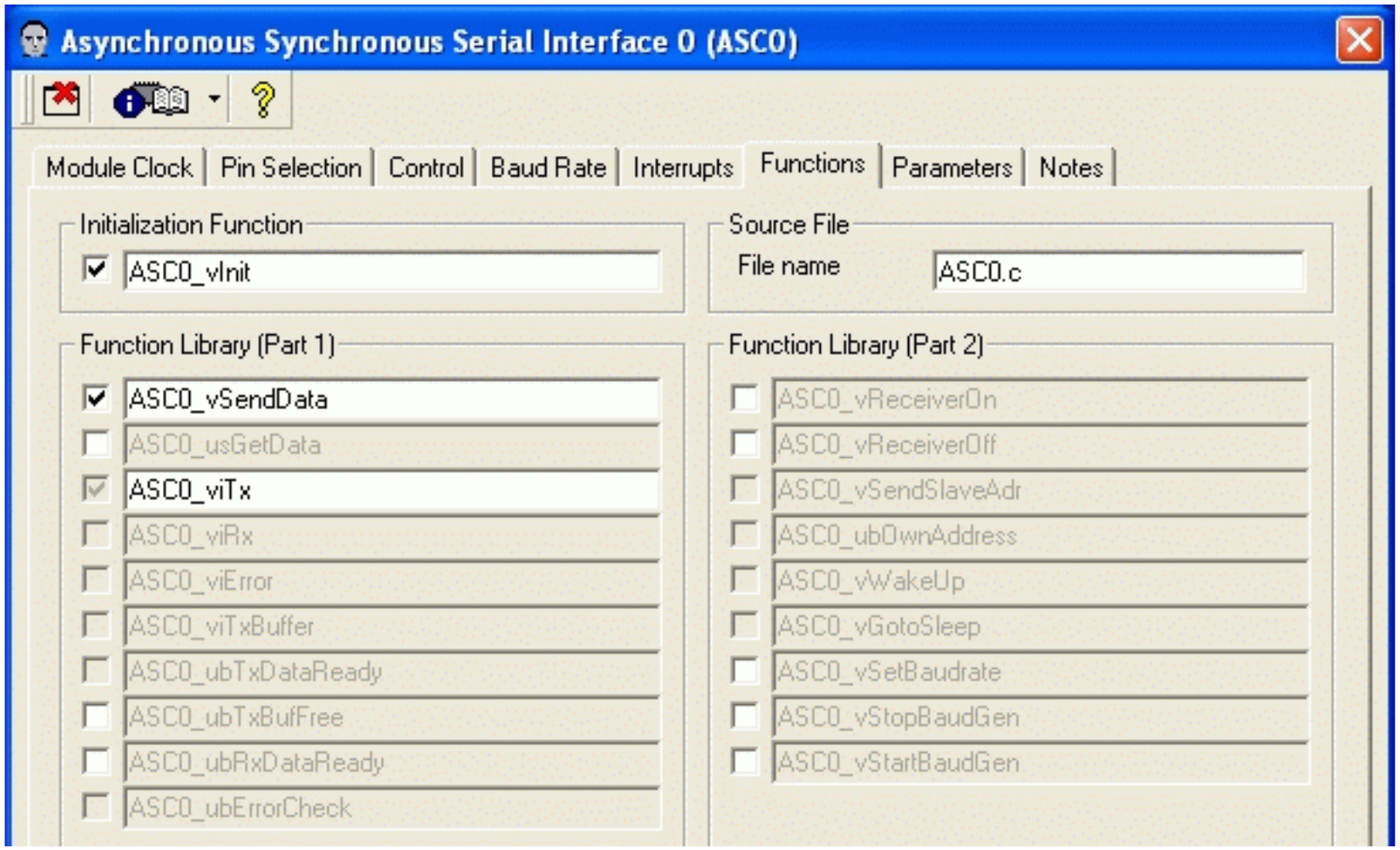
On the **Interrupts** page drag the **ASC0 transmit SRN** interrupt from the right list to the CPU interrupt level 3.



Exercise 5: ASC

On the **Functions** page

■ Check the `ASC0_vInit` and `ASC0_vSendData` functions.



4. Generate the application framework

Click on the **Generate Code** icon ⚡ on the application toolbar to start the code generation process. Save and close the *DAvE* project.

5. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\asc\asc.pjt`.

Exercise 5: ASC

6. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

7. Set current project

Use the context menu in the workspace window to make the asc project the current project.

8. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter `c:\infineon\tc1796_extmem.opt`.

9. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

Exercise 5: ASC

10. Add the user code

The `ASC0_Write` function in this exercise is implemented using a global pointer to a string send buffer. The `ASC0_Write` function sets the global pointer and requests an interrupt. The interrupt routine sends on byte of the string and increments the pointer. Add the following code to `ASC0.c`.

```
// USER CODE BEGIN (Tx,1)
static char *ss = "";                   // output pointer
// USER CODE END
...
void __interrupt(ASC0_TINT) ASC0_viTx(void)
{
  // USER CODE BEGIN (Tx,2)
  if (*ss)
    ASC0_vSendData(*ss++);              // send until string end
  // USER CODE END
} //  End of function ASC0_viTx
...
// USER CODE BEGIN (ASC0_General,10)
void ASC0_vWrite(char *s)
{
  while(*ss)                            // wait if busy
    ;
  ss = s;
  ASC0_TSRC_SETR = 1;                   // Set interrupt request
}
```

Add a prototype of `ASC0_vWrite` to `ASC0.h` at (ASC0_Header,8). Add an endless loop in `MAIN.c` at (Main,9).

Exercise 5: ASC

In `STM.c` at (SRN0,2) add code to the interrupt routine:

```
void __interrupt(STM_INT0) STM_viSRN0(void)
{
  ...
  if(STM_ICR_CMP0IR == 1)  // if compare match of CMP0 is pending
  {
    // USER CODE BEGIN (SRN0,3)
    static unsigned long i = 0;
    unsigned long sec;
    char s[50];

    IO_vTogglePin(IO_P1_15);
    sec = i*((1<<24)/75000000.);
    i++;
    sprintf(s, "TriBoard running since: "
           "%02ldh:%02ldmin:%02ldsec\r",
           sec/3600, (sec%3600)/60, sec%60);
    ASC0_vWrite(s);
    // USER CODE END
    STM_ISRR_CMP0IRR = 1;      // clear request bit of CMP0
  }
...
} //  End of function STM_viSRN0
```

11. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

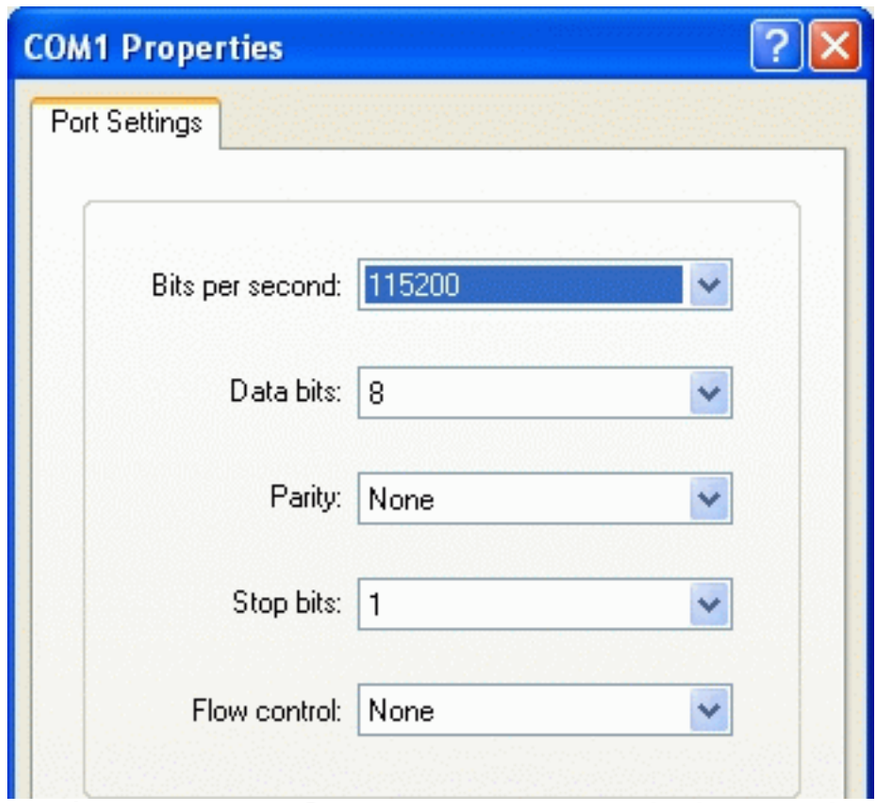
12. Debug the application

Click the **Debug** icon  on the Build toolbar to open the *CrossView Pro* debugger.

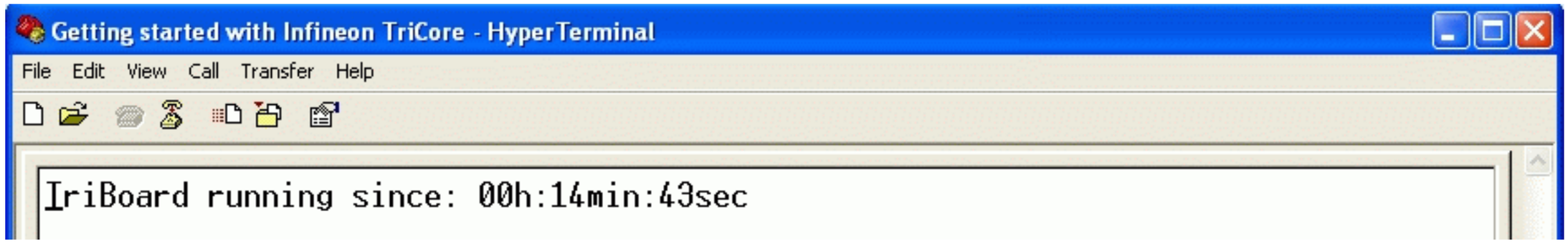
Exercise 5: ASC

13. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar. Open a HyperTerminal window. Set the COM parameter to 115200 8-N-1-None.



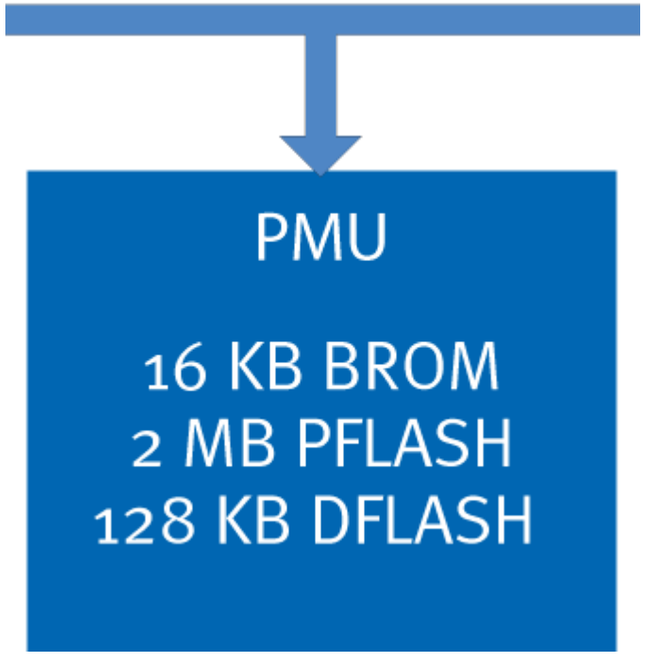
See the output. On some system is it required to reset the application once. Choose **Run > Reset Target System**.



Exercise 6: Programming the flash

Programming the flash

The TC1796 TriBoard comes with 2 MByte on-chip flash memory. In this exercise we learn how to program the flash. We are using UDE Flash/OTP Memory Programming which is an add-in to the *PLS UDE* debugger environment. If you prefer to use the *Infineon* Memtool read the instructions in the *Appendix A*. If you prefer to use the *Hitex* Debugger read the instructions in the *Appendix B*.



Exercise 6: Programming the flash

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\fls`. Copy the `asc.dav` file from the previous exercise to the new directory and rename the file to `fls.dav`.

2. Generate the application framework

Start *DAvE* and open the `fls.dav` project file. Click on the **Generate Code** icon  on the application toolbar to start the code generation process. Save and close the *DAvE* project.

3. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\fls\fls.pjt`.

4. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

5. Set current project

Use the context menu in the workspace window to make the fls project the current project.

6. Load the project options

Copy the preconfigured option file `tc1796_intflash.opt` from the `tc1796.zip` archive to `c:\infineon`. Choose **Project > Load Options**. In the **Filename** field enter `c:\infineon\tc1796_intflash.opt`.

Exercise 6: Programming the flash

7. Add the user code

At (Main,9) in `MAIN.c` add

```
// USER CODE BEGIN (Main,9)
ASC0_vWrite("\n\r\n\r"
            "Infineon Technologies AG\n\r"
            "www.infineon.com/microcontroller\n\r"
            "TriBoard: TC1796\n\r"
            "CPU clock: 150 MHz\n\r"
            "System clock: 75 MHz\n\r");
for (;;)  // forever
  ;
// USER CODE END
```

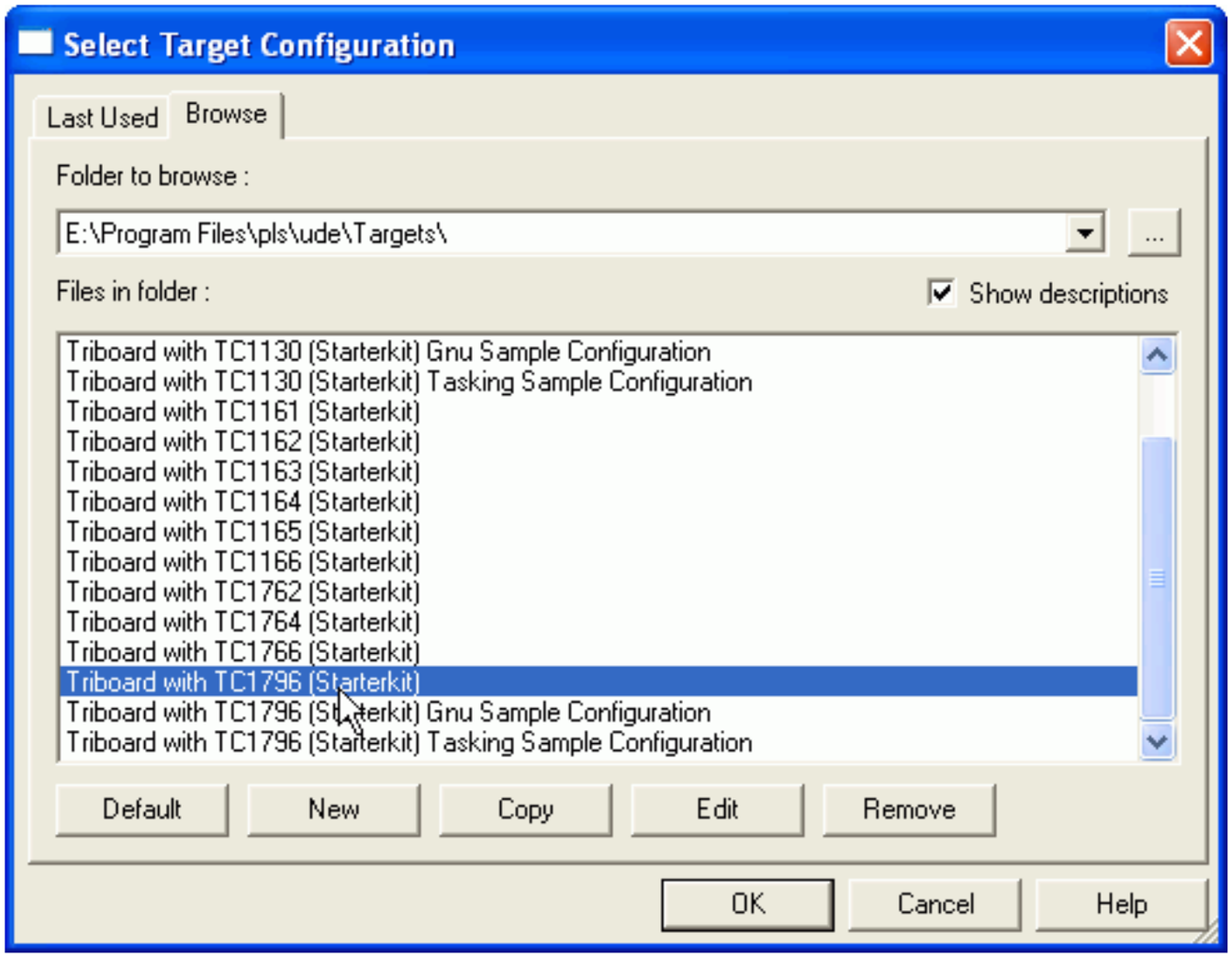
Include the `stdio.h` to `STM.c` and add in the interrupt routine at (SRN0,3)

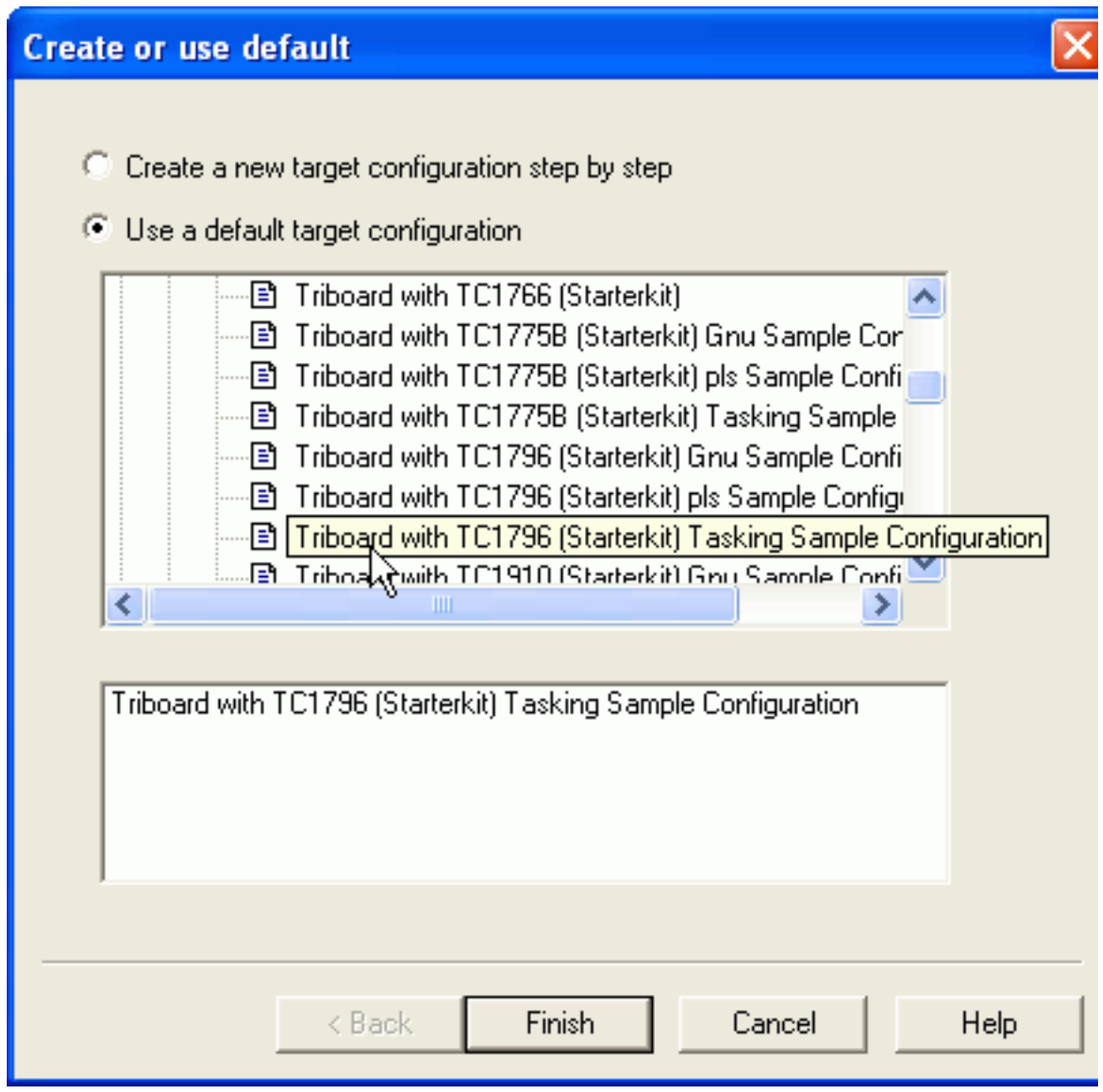
```
// USER CODE BEGIN (SRN0,3)
static unsigned long i = 0;
unsigned long sec;
char s[50];

IO_vTogglePin(IO_P1_15);
sec = i*((1<<24)/75000000.);
i++;
sprintf(s, "TriBoard running since: "
       "%02ldh:%02ldmin:%02ldsec\r",
       sec/3600, (sec%3600)/60, sec%60);
ASC0_vWrite(s);
// USER CODE END
```

Exercise 6: Programming the flash

8. Install the flash programming tool

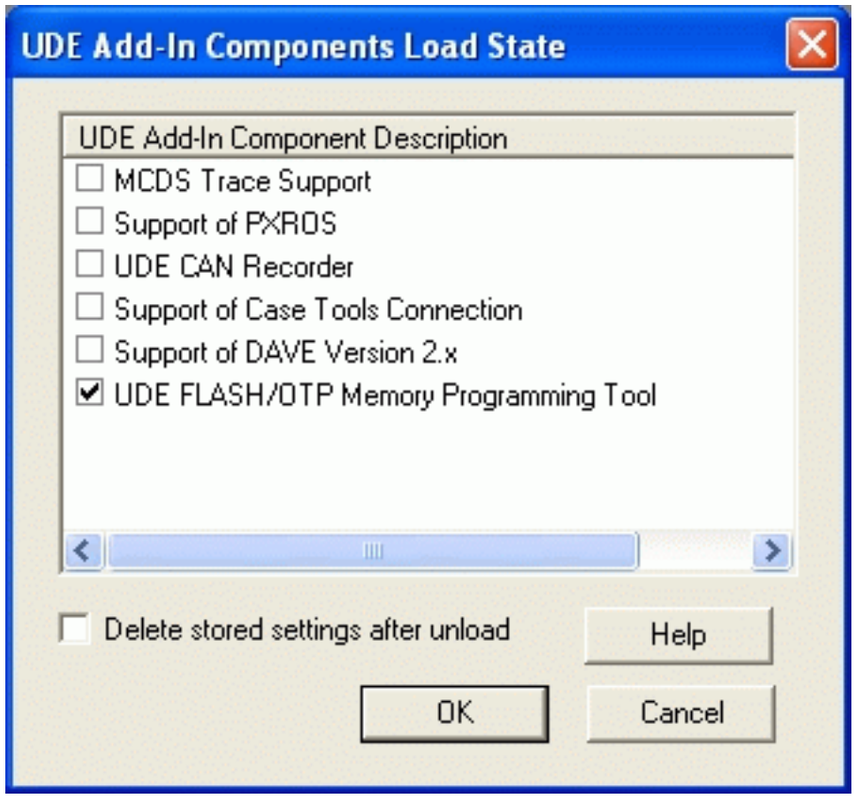
Start the PLS UDE and create a new workspace file `fls.wsp` with the TC1796 target configuration. If the configuration is not in the list click **New** to get the default TriCore > Starter Kit configuration.





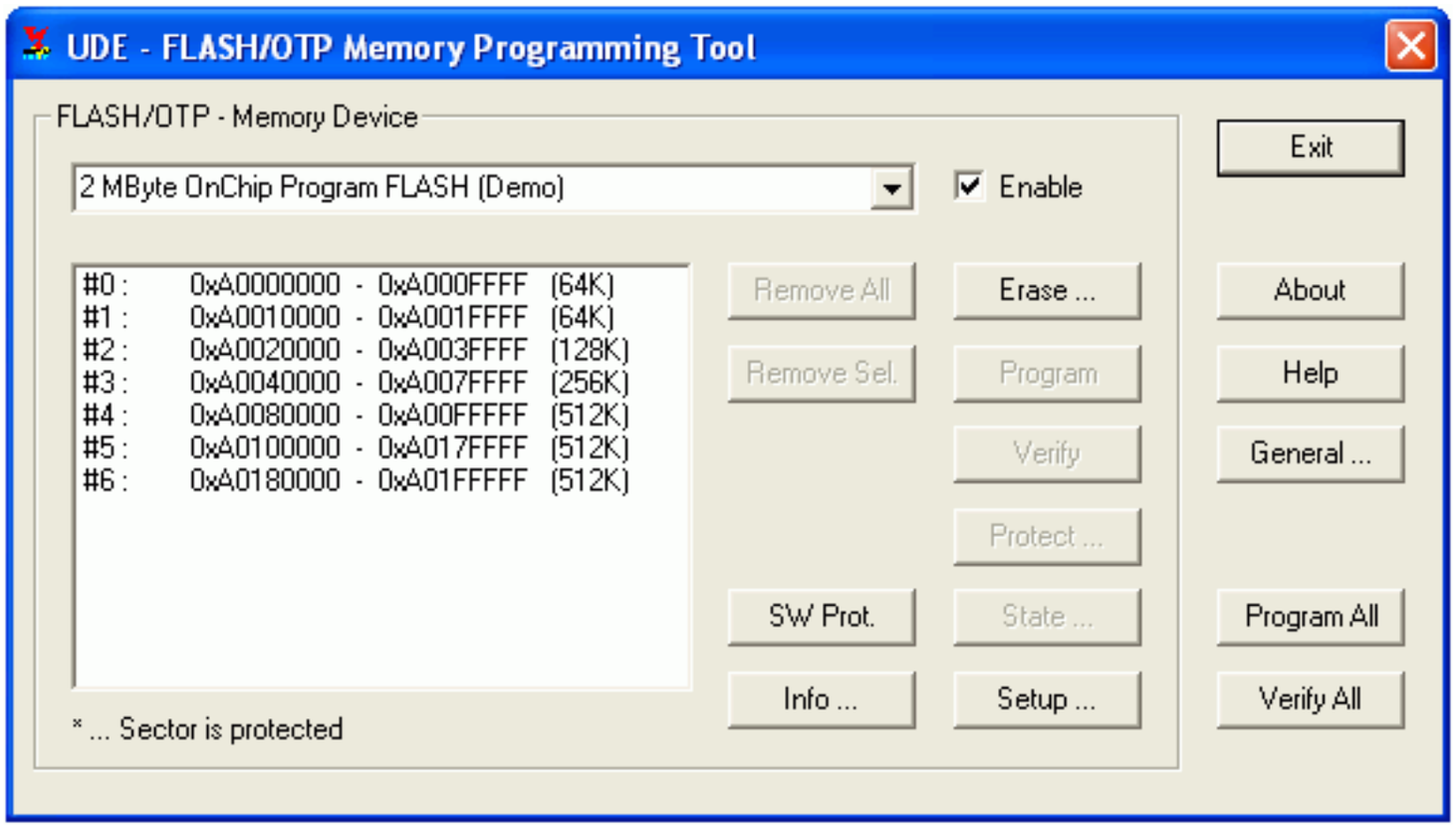
Exercise 6: Programming the flash

Choose **Config > Add-in Components** and load the **UDE Flash/OTP Memory Programming Tool**.



Exercise 6: Programming the flash

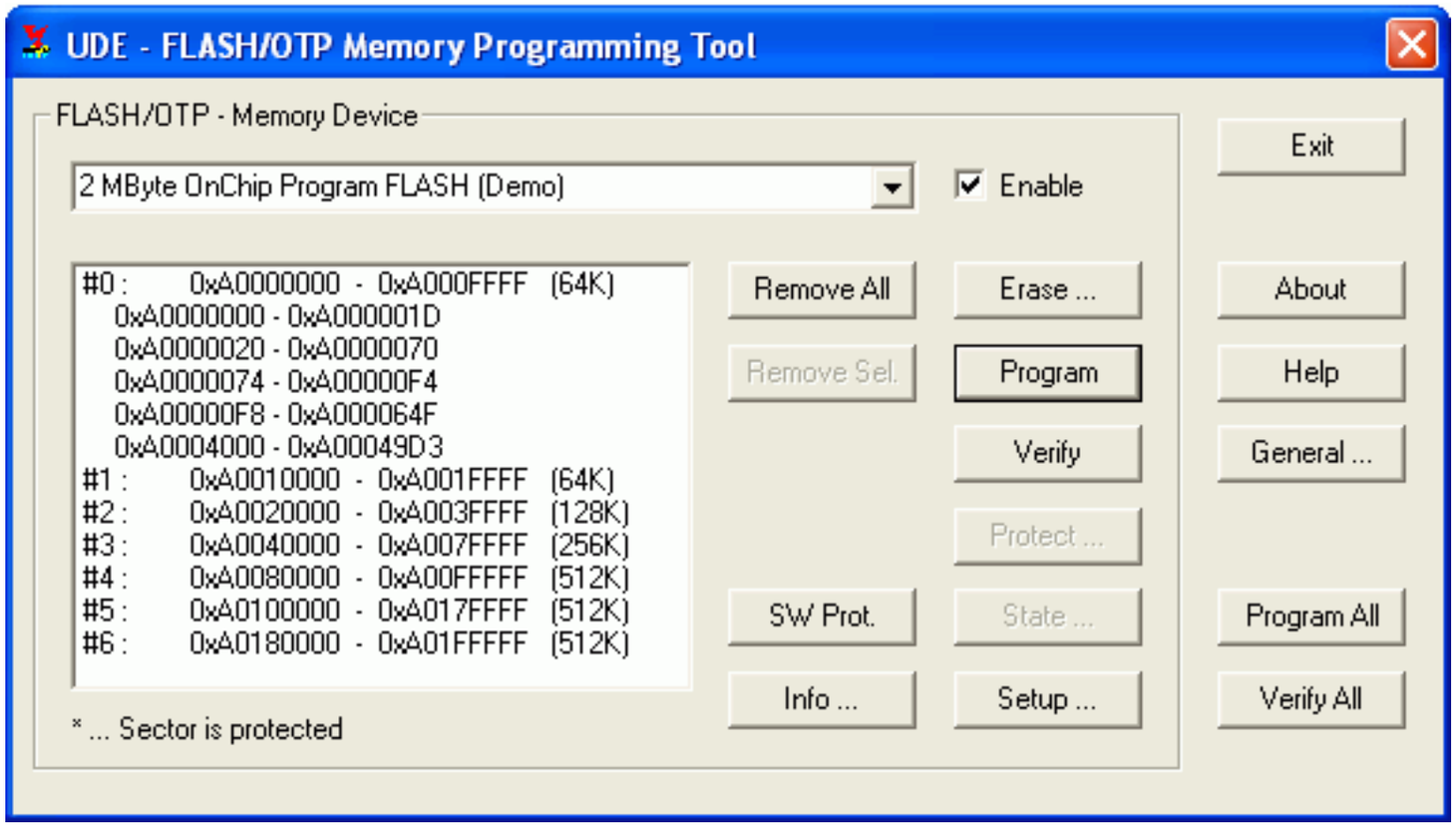
Choose **Tool > FLASH Programming...** and enable the flash. Click **Exit**.



Exercise 6: Programming the flash

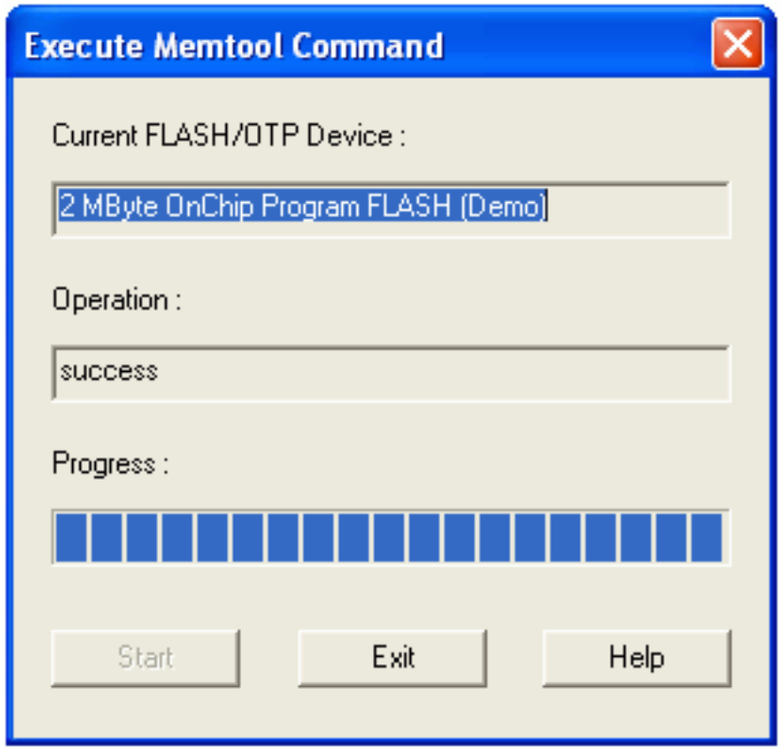
9. Program to flash

Choose **File > Load Program** and open `fls.elf`. Click **Program** to start programming the flash.



Exercise 6: Programming the flash

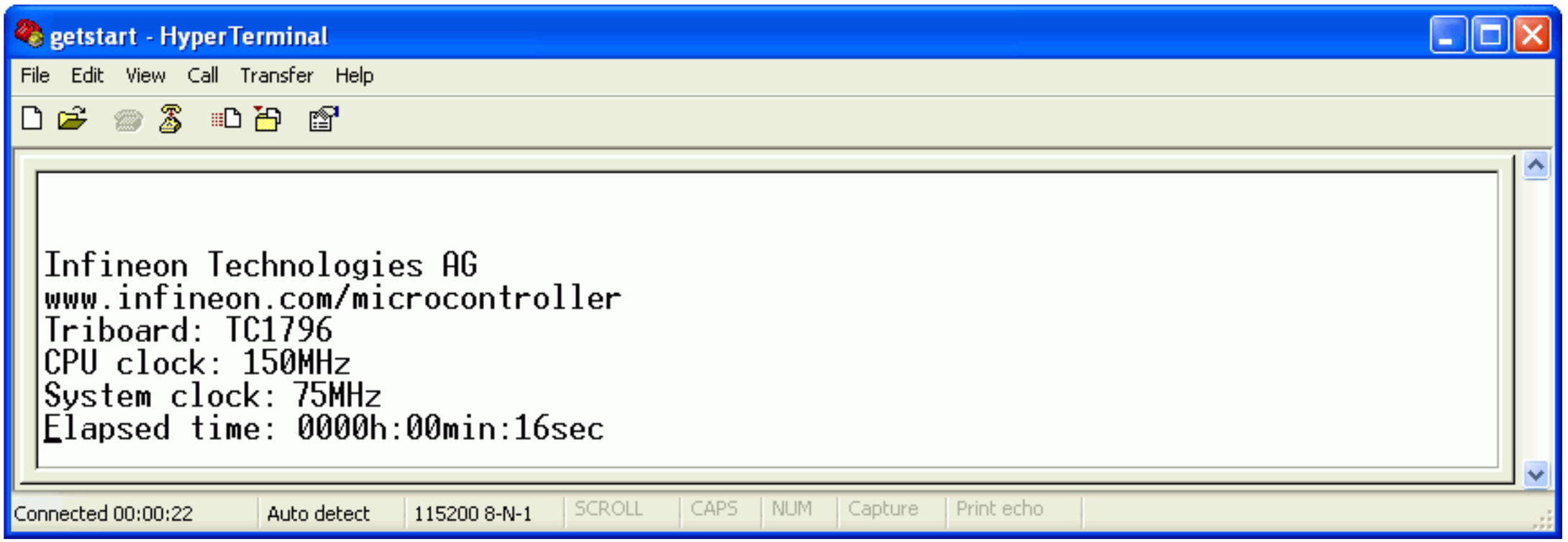
A process dialog appears. Wait until the operation succeeds and **Exit** the dialog.



Exercise 6: Programming the flash

10. Verify the changed program

Open a HyperTerminal window and setup a COM port connection to 115200 8-N-1-None. Press the reset button on the Tri-Board and see the changes.

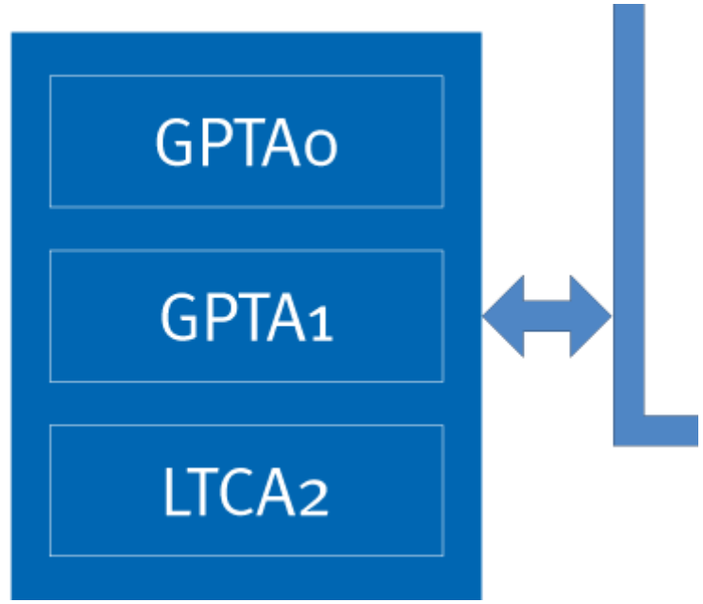


Exercise 7: PWM

PWM

In this exercise you will develop an application that generates a sinusodial complementary 3-phase PWM (Puls Width Modulation) signal with an update frequency of 20KHz using the General Purpose Timer Array (GPTA).

The GPTA provides a set of timer, compare and capture functionalities that can be flexibly combined to form signal measurement and signal generation units. They are optimized for tasks typical of engine, gearbox, and electrical motor control applications, but can also be used to generate simple and complex signal waveforms needed in other industrial applications.
The TC1796 contains two General Purpose Timer Arrays (GPTA0 and GPTA1) with identical functionality, plus an additional Local Timer Cell Array (LTCA2).



Exercise 7: PWM

GPTA Features

Each of the General Purpose Timer Arrays (GPTA0 and GPTA1) provides a set of hardware modules required for high-speed digital signal processing:

Clock Generation Unit

- Filter and Prescaler Cells (FPC) support input noise filtering and prescaler operation.
- Phase Discrimination Logic units (PDL) decode the direction information output by a rotation tracking system.
- Duty Cycle Measurement Cells (DCM) provide pulse-width measurement capabilities.
- A Digital Phase Locked Loop unit (PLL) generates a programmable number of GPTA module clock ticks during an input signal's period.

Signal Generation Unit

- Global Timer units (GT) driven by various clock sources are implemented to operate as a time base for the associated Global Timer Cells.
- Global Timer Cells (GTC) can be programmed to capture the contents of a Global Timer on an external or internal event. A GTC may also be used to control an external port pin depending on the result of an internal compare operation. GTCs can be logically concatenated to provide a common external port pin with a complex signal waveform.
- Local Timer Cells (LTC) operating in Timer, Capture, or Compare Mode may also be logically tied together to drive a common external port pin with a complex signal waveform. LTCs, enabled in Timer Mode or Capture Mode, can be clocked or triggered by various external or internal events.

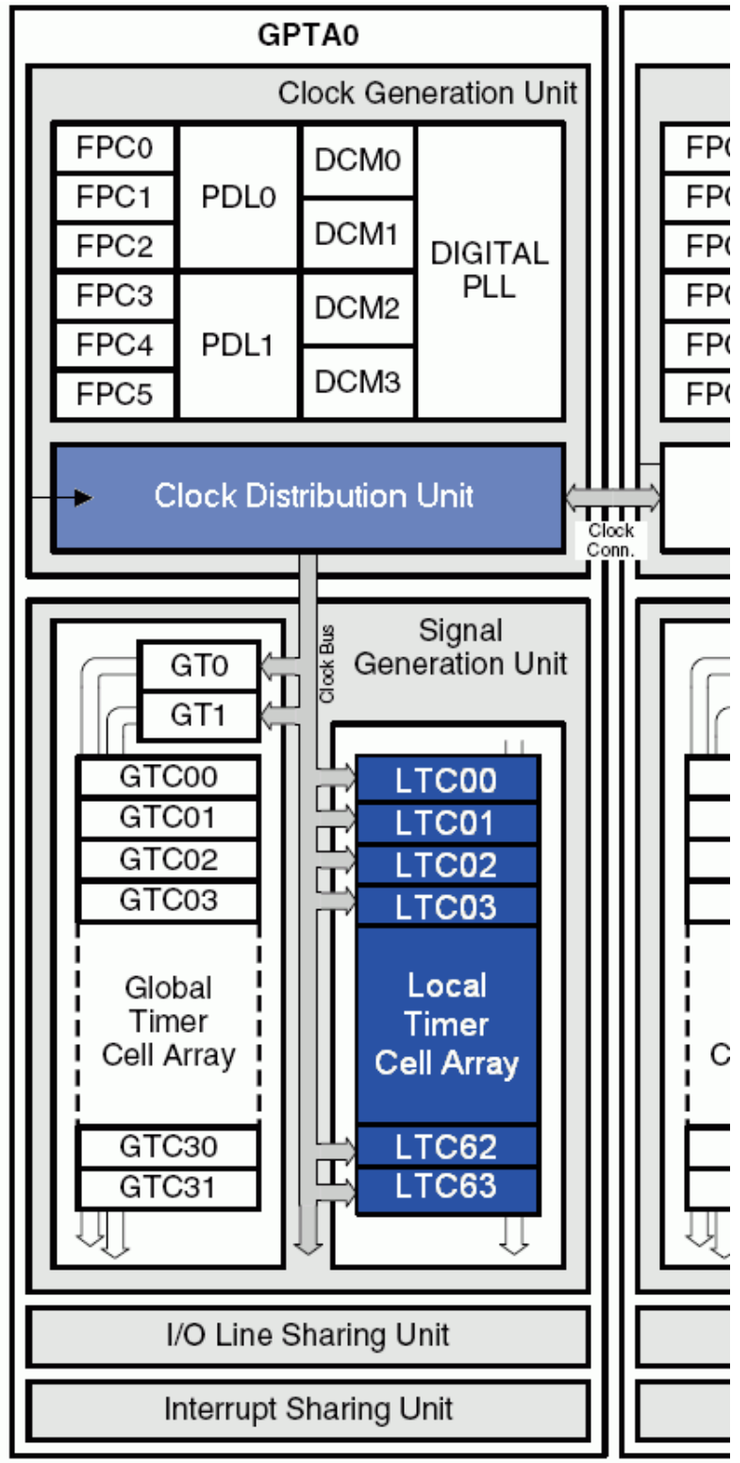
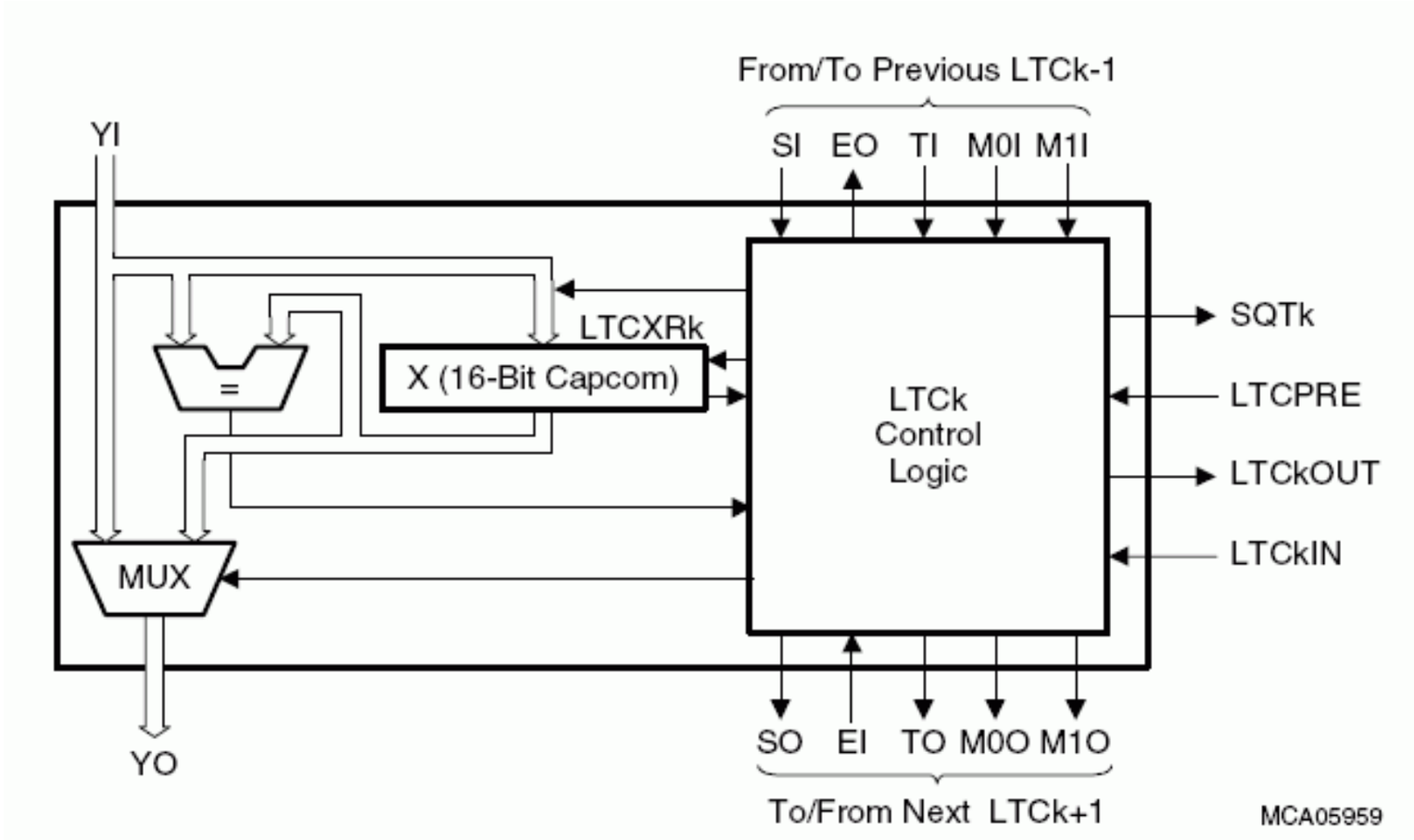


Fig.1 GPTA Block Diagram

Exercise 7: PWM

LTC Functionality

- Local Timer Cell (LTC)
- 64 independent units
- Three basic operating modes (Timer, Capture and Compare) for 63 units
- Special compare modes for one unit
- 16-bit data width
- f_{GPTA} maximum resolution
- $f_{GPTA}/2$ maximum input signal frequency



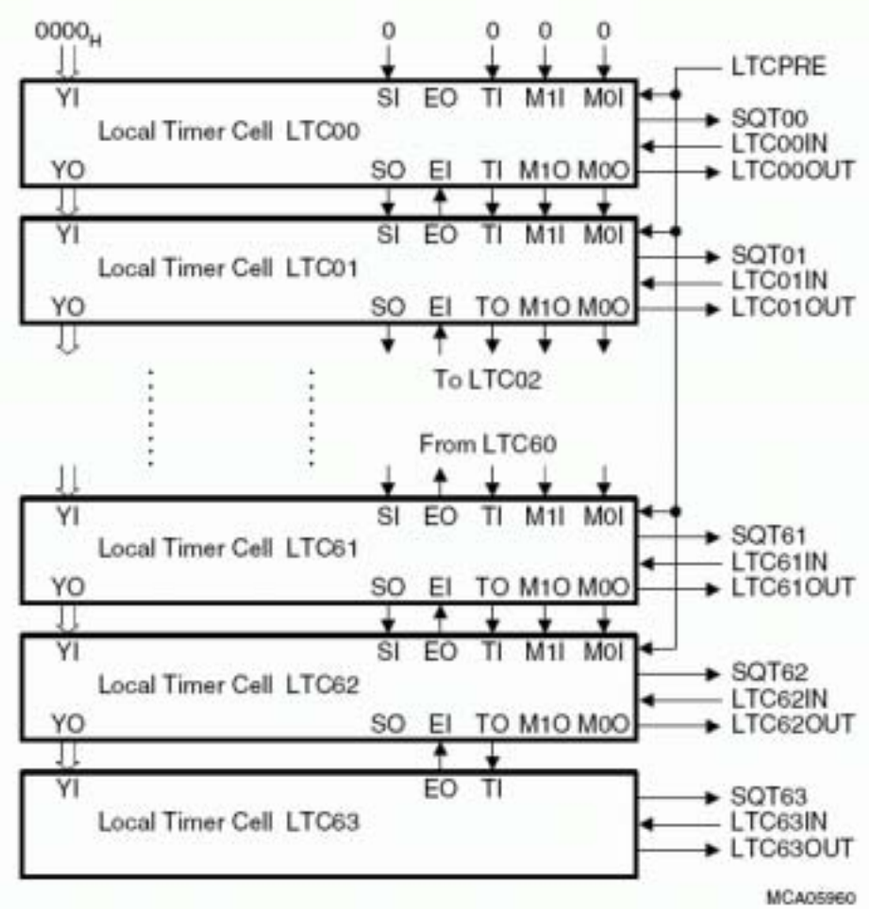


Fig.2 Architecture of Local Timer Cells and Interconnections between LTCs

Exercise 7: PWM

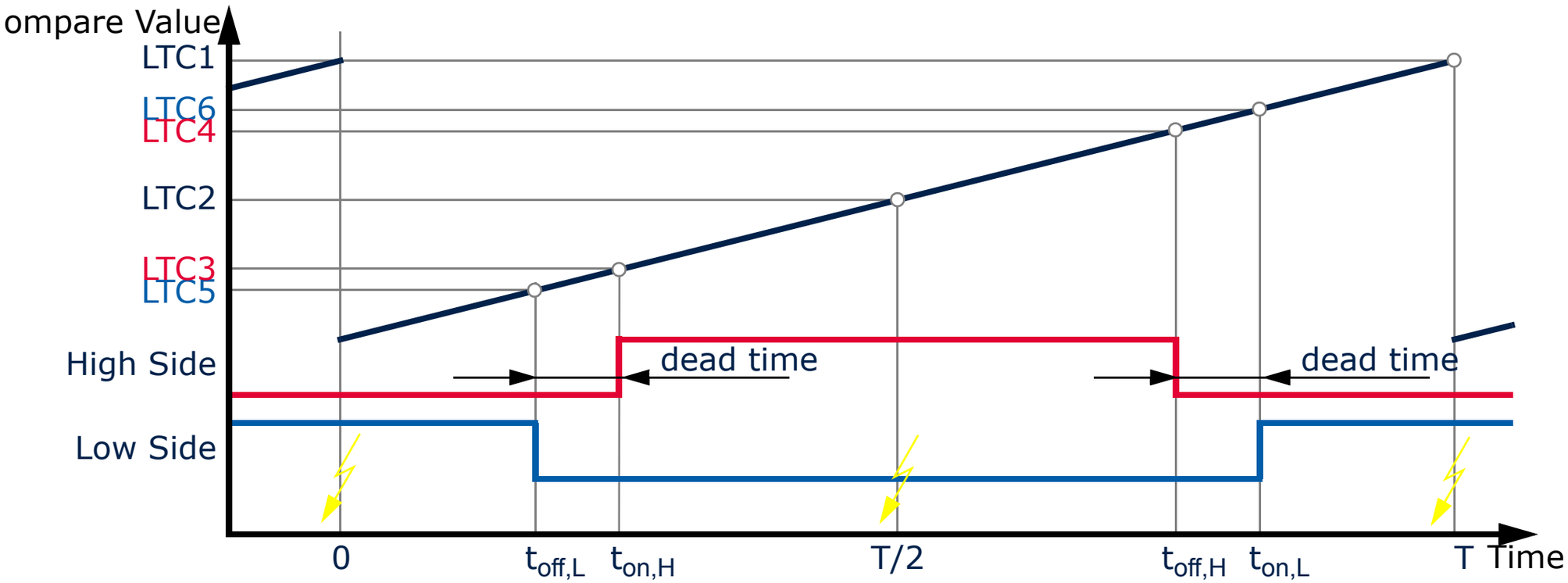


Fig.3 PWM Generation with Local Timer Cells

	Value	Cell	Output Control Mode	Port
Reset Timer	-	LTC0	-	-
Compare Period	T	LTC1	Interrupt: adjust $t_{off,H}$, $t_{on,L}$	-
Compare Mid Period	T/2	LTC2	Interrupt: adjust $t_{on,H}$, $t_{off,L}$	-
Compare High side On	$t_{on,H}$	LTC3	Set output by a local event	-
Compare High side Off	$t_{off,H}$	LTC4	Reset output by a local event or copy the previous cell action	P2.8
Compare Low Side Off	$t_{on,L}$	LTC5	Reset output by a local event	-
Compare Low Side On	$t_{on,L}$	LTC6	Set output by a local event or copy the previous cell action	P2.9

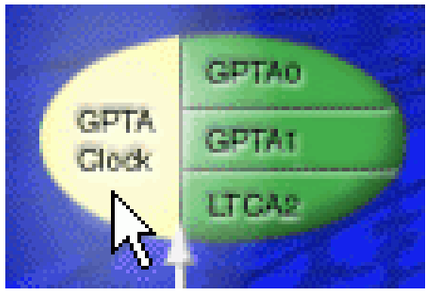
Exercise 7: PWM

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\pwm`. Copy the `isr.dav` file from the previous exercise to the new directory and rename the file to `pwm.dav`.

2. Open the GPTA clock properties

Click on **GPTA Clock** in the project window to open the GPTA Clock properties.

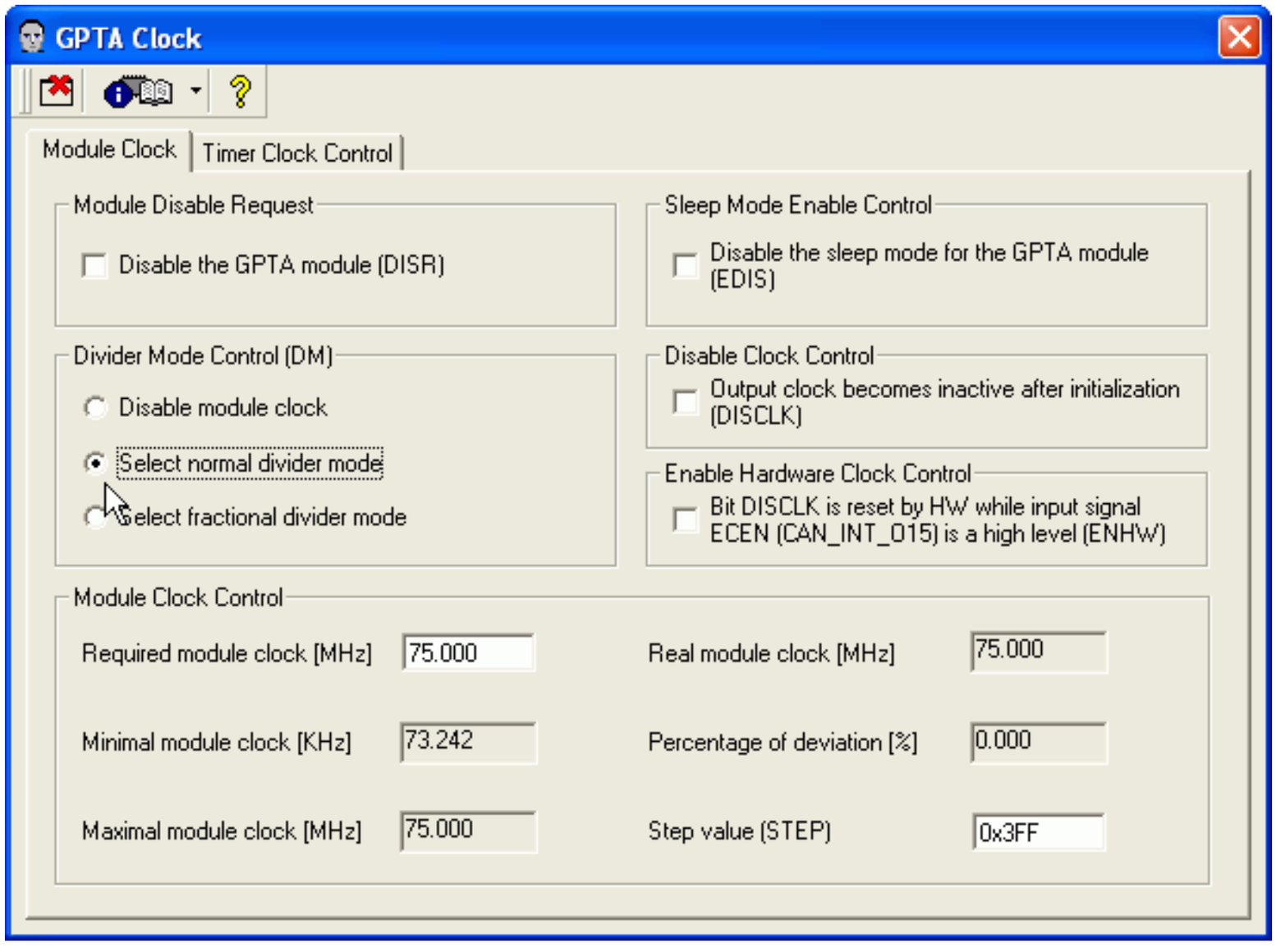


Exercise 7: PWM

3. Set up the GPTA clock properties

On the **Module Clock** page

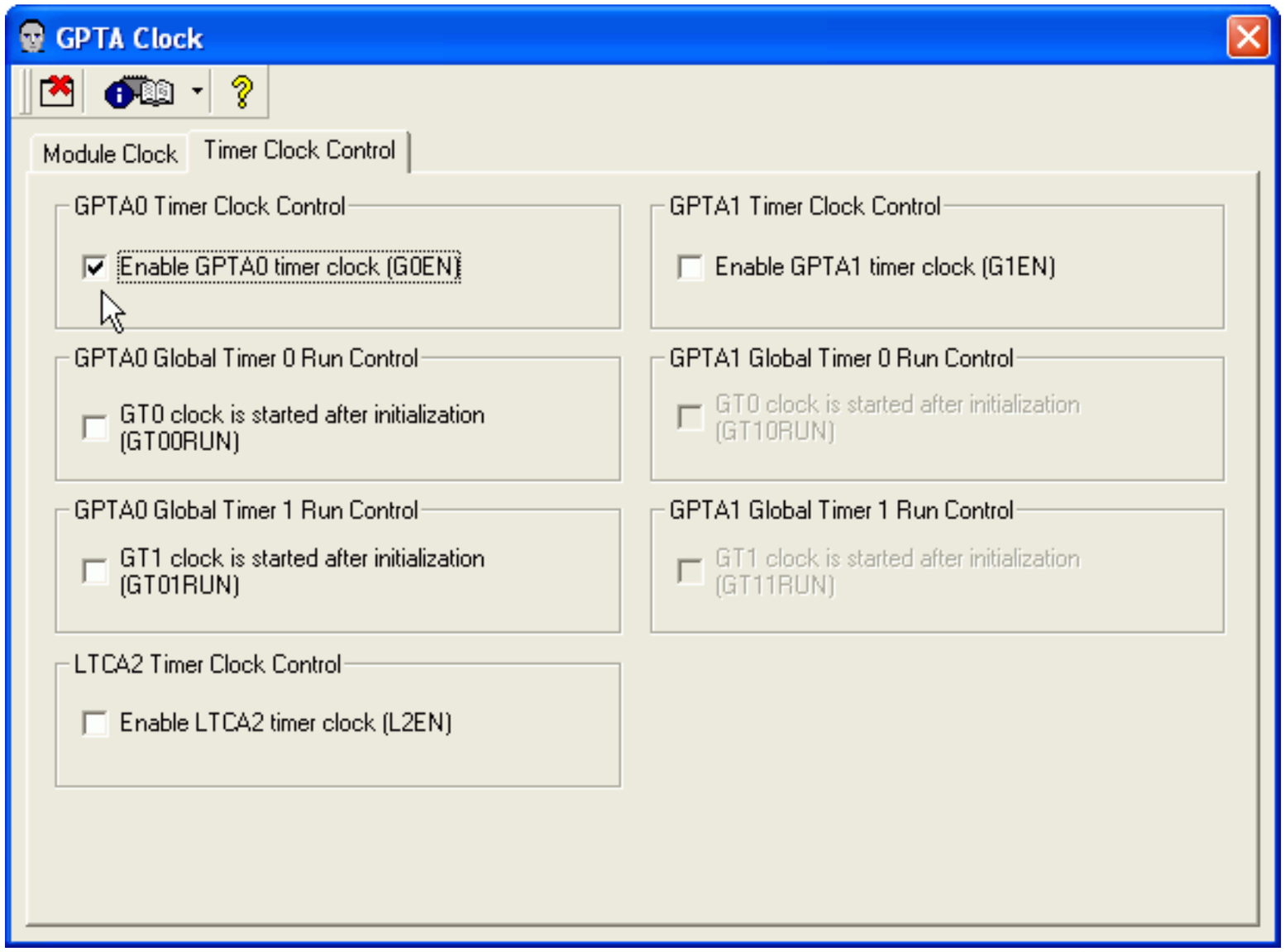
- Select **Select normal divider mode**.
- Enter **Required module clock [MHz]** to f_{GPTA} = 75 MHz.



Exercise 7: PWM

On the **Timer Clock Control**

■ Check **Enable GPTA0 timer clock (G0EN)**.

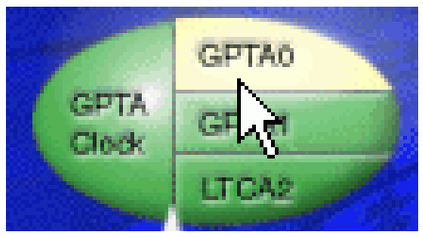


The GPTA clock is completely configured. Click the **Close** icon on the dialog toolbar to close the **GPTA Clock** dialog.

Exercise 7: PWM

4. Open the GPTA0 properties

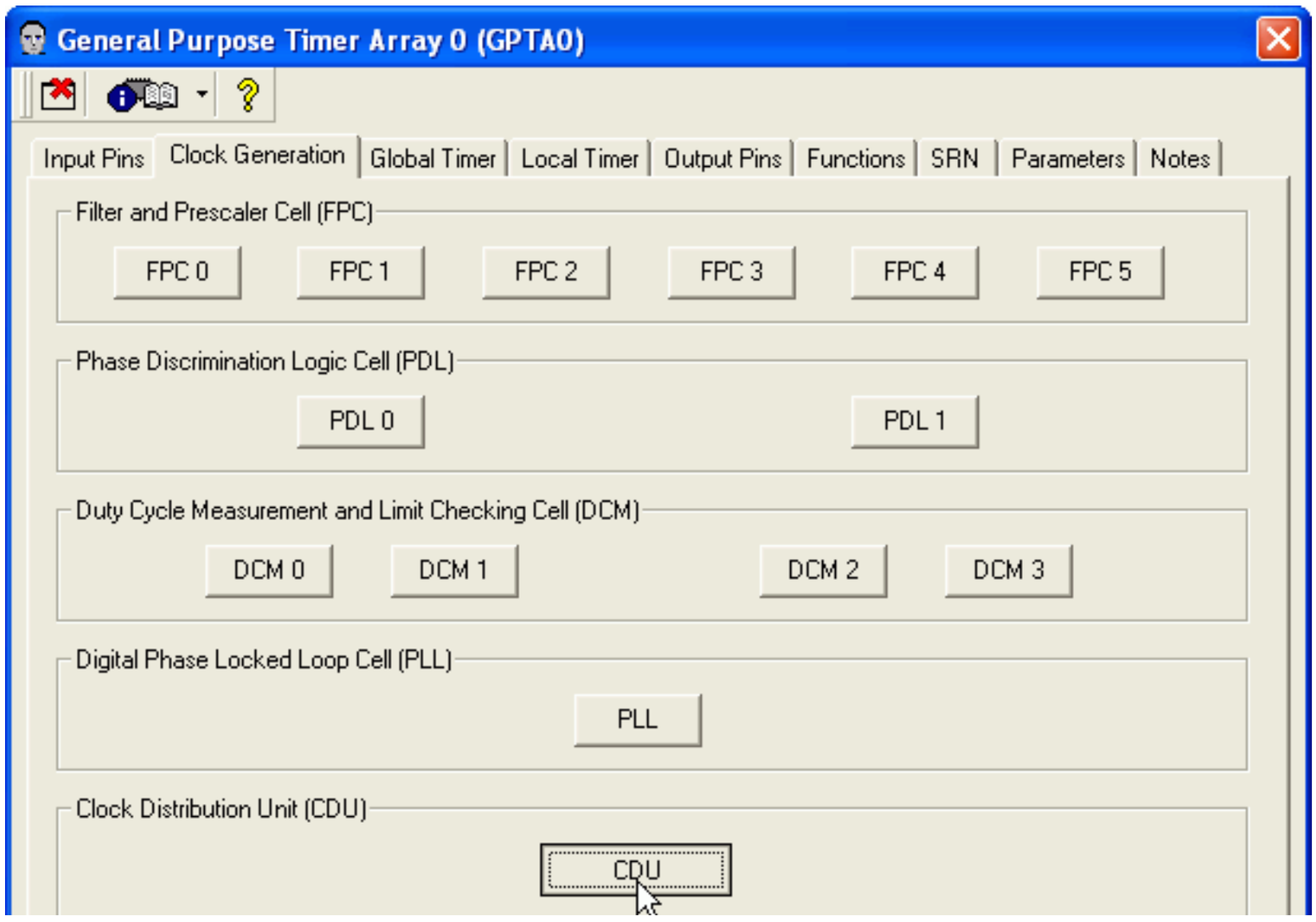
Click on **GPTA0** in the project window to open the GPTA0 properties.



Exercise 7: PWM

5. Set up the GPTA0 properties

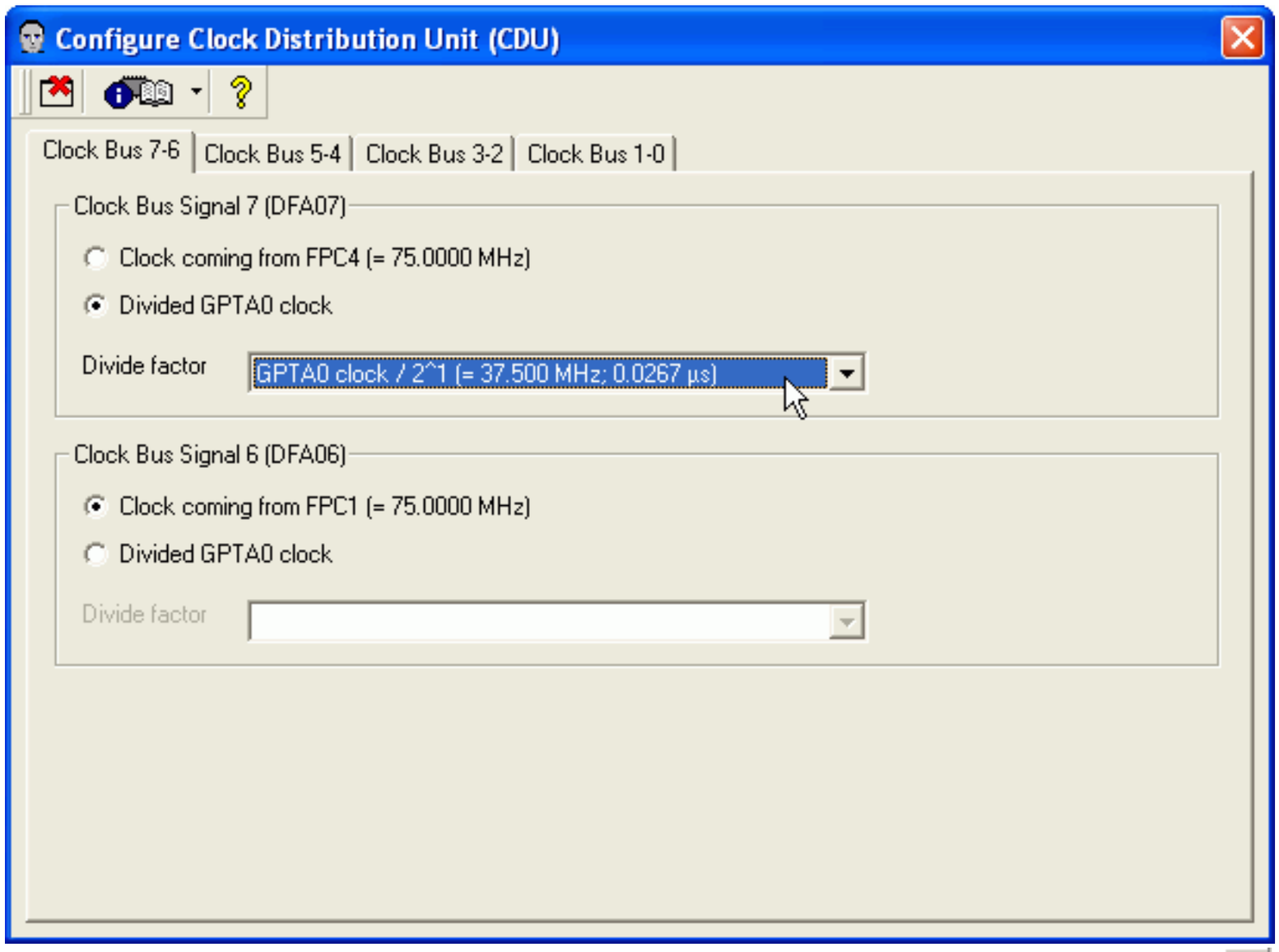
On the **Clock Generation** page click **CDU** to open the Clock Distribution Unit properties.



Exercise 7: PWM

On the **Clock Bus 7-6** page

- At **Clock Bus Signal 7 (DFA07)** select **Divided GPTA0 clock** and choose the Divide factor **GPTA0 clock / 2^1**. The maximum input frequency of the local timer cells is $f_{GPTA}/2$.

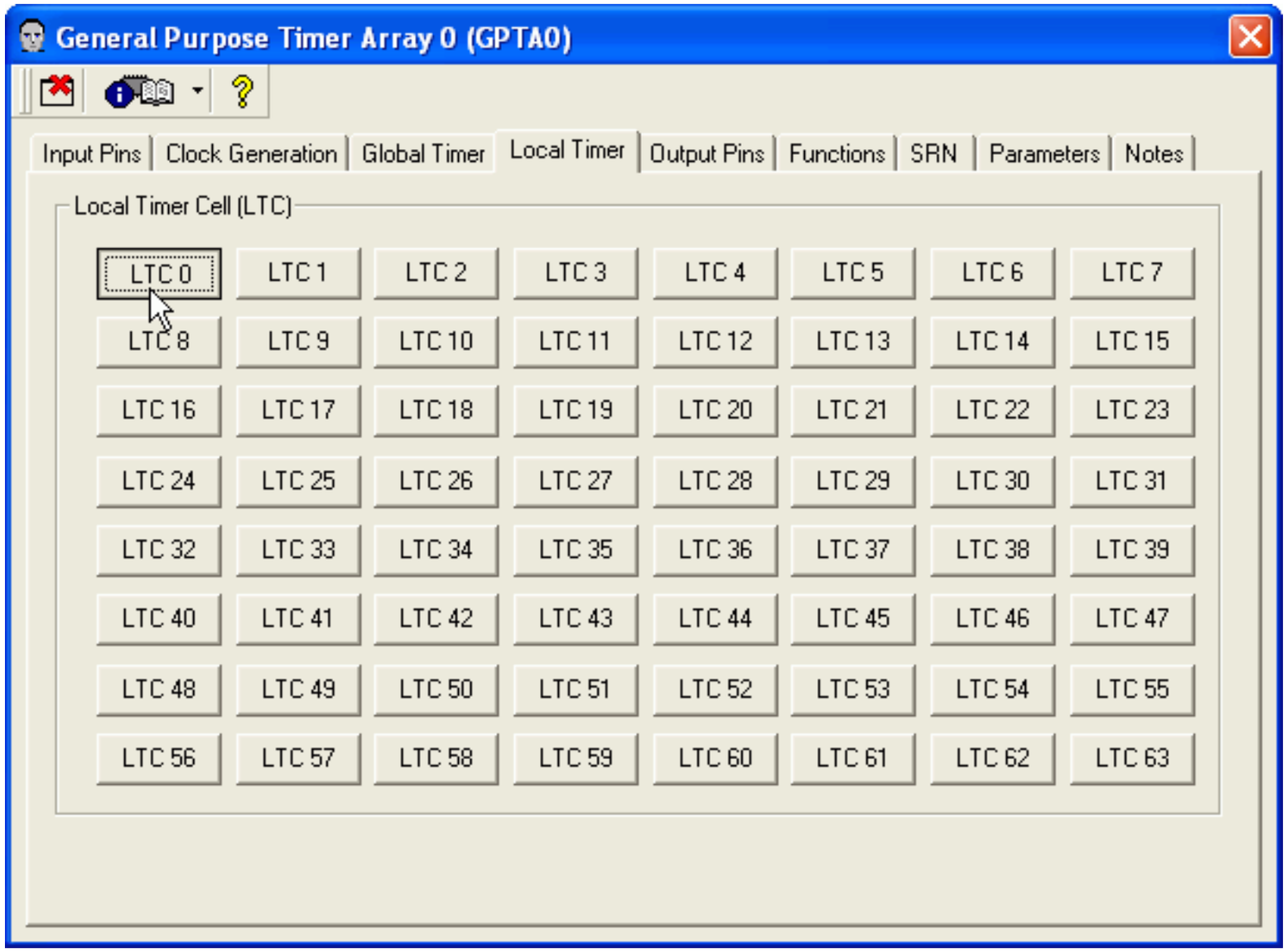


The Clock Distribution Unit is completely configured. Click the **Close** icon on the dialog toolbar to close the **Configure Clock Distribution Unit (CDU)** dialog.

Exercise 7: PWM

6. Set up the GPTA0 properties. Local Timer

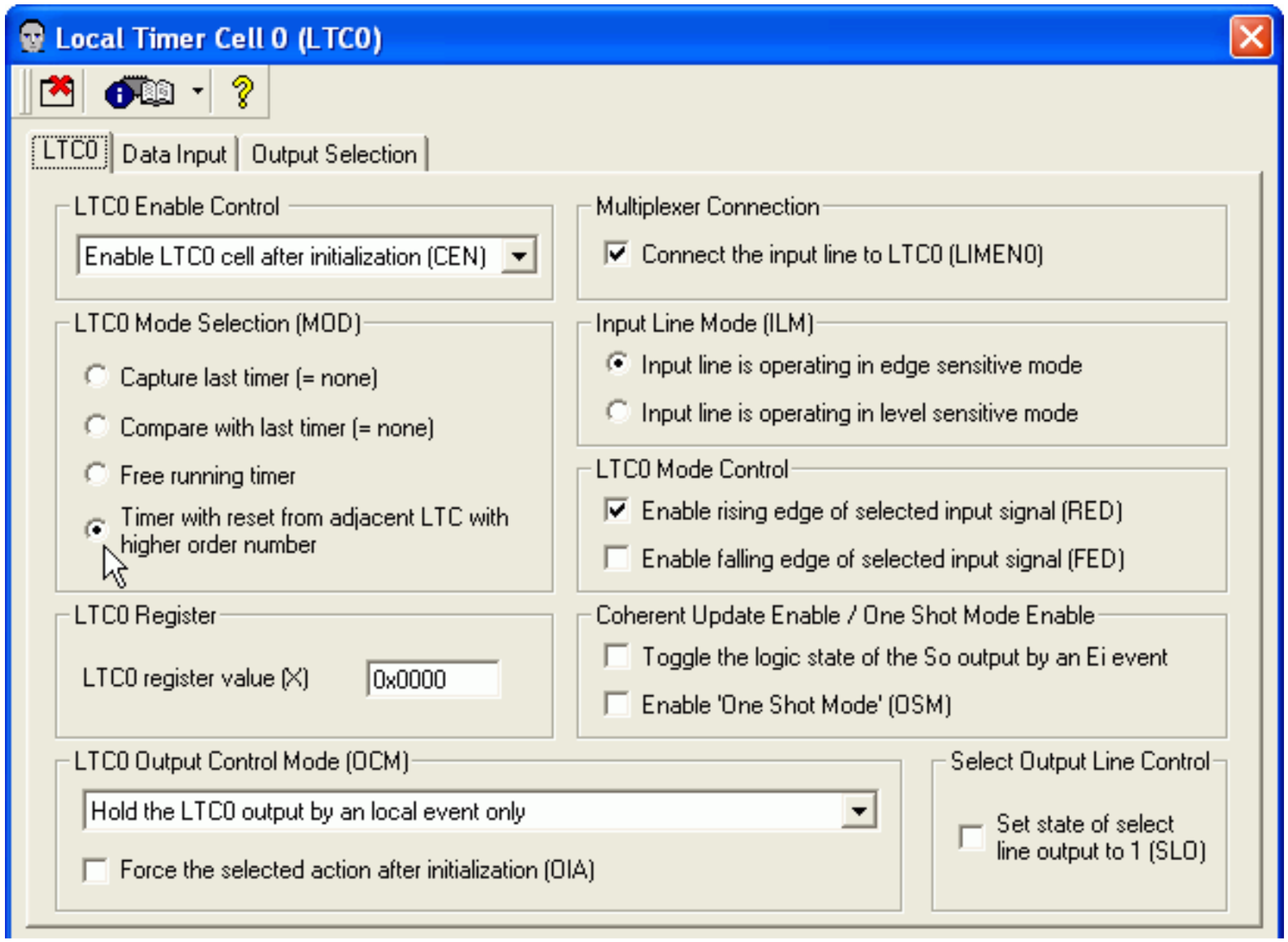
On the **Local Timer** page click **LTC0** to open the Local Timer Cell 0 properties.



Exercise 7: PWM

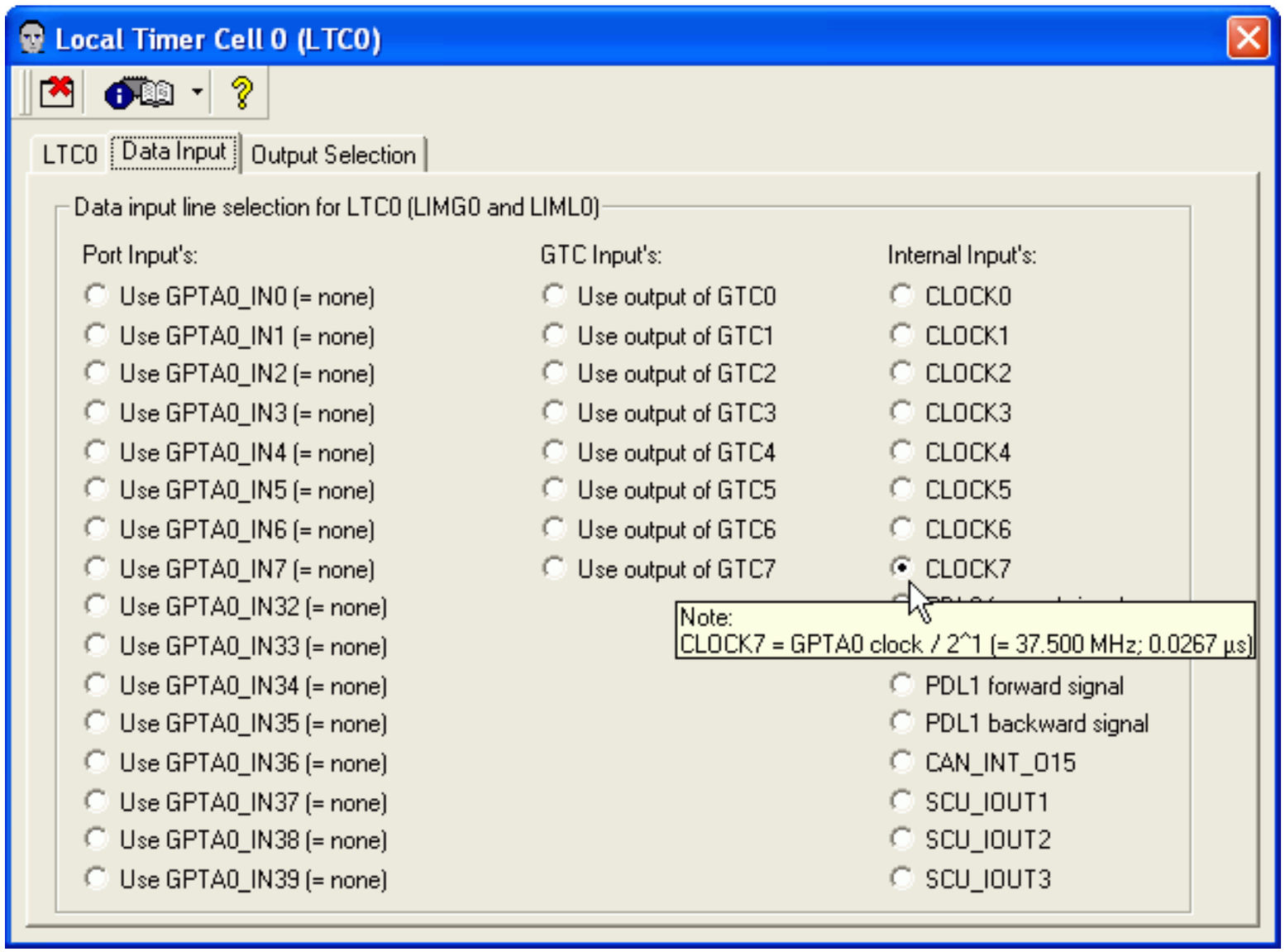
On the **LTC0** page configure the cell as reset timer:

- Choose **Enable LTC0 cell after initialization (CEN)**,
- Select **Timer with reset from adjacent LTC with higher order number**. When the timer value in LTC0 matches the compare value stored in LTC1, the Event Out signal from LTC1 is activated and passed upstream to LTC0 as the Event In signal which then resets the timer to the value of $FFFF_h$.
- Check **Connect the input line to LTC0 (LIMEN0)**. The cell will be connected to the clock bus line 7 which we configured in step 5.
- Check **Enable rising edge of selected input signal (RED)**.



Exercise 7: PWM

On the **Data Input** page of LTC0 and select the internal input **Clock7** as data input line for LTC0.

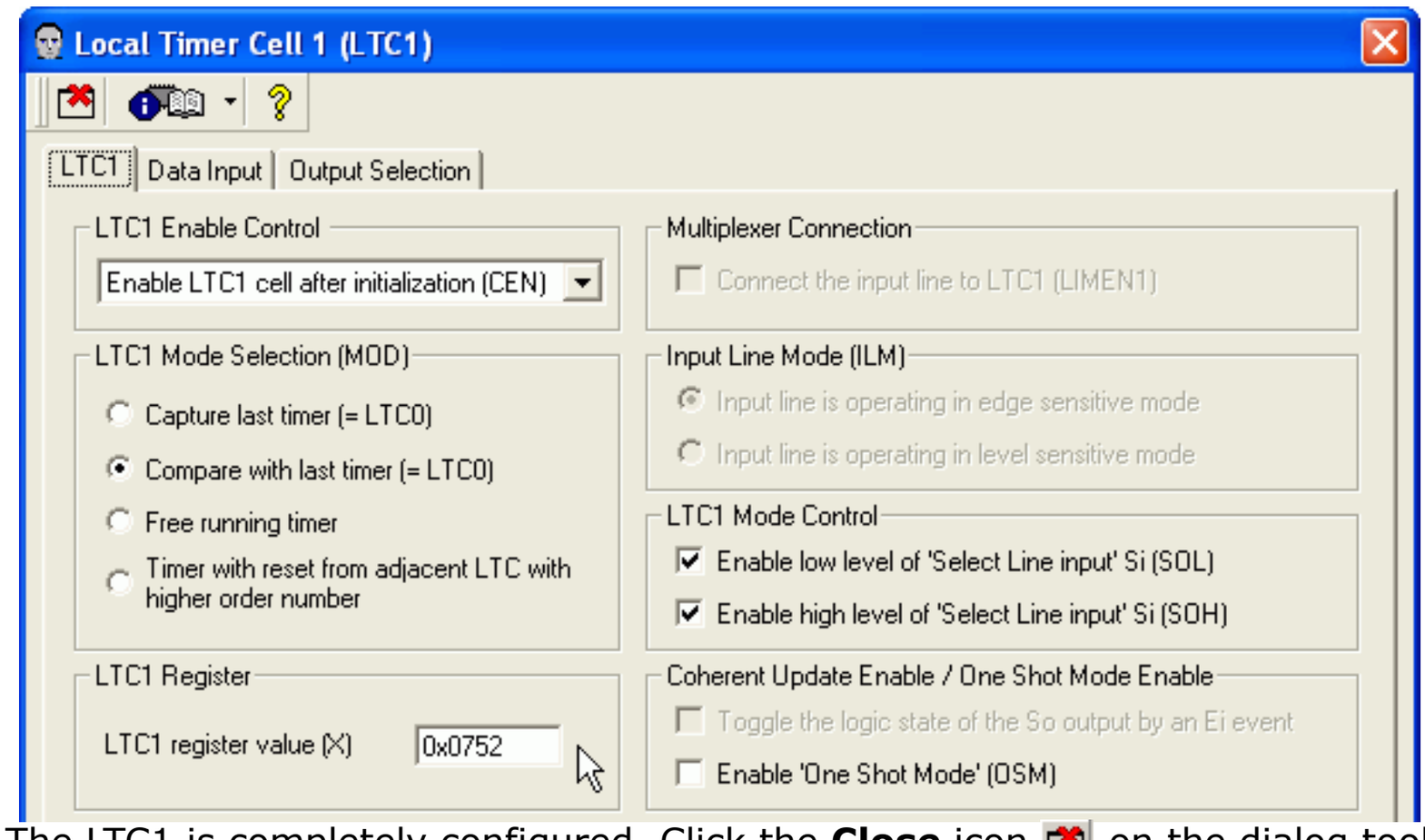


The LTC0 is completely configured as reset timer running at $f_{GPTA}/2$ = 37.5MHz. Click the **Close** icon on the dialog tool-bar to close the **Local Timer Cell 0 (LTC0)** dialog.

Exercise 7: PWM

Click **LTC1** on the **Local Timer** page. On the **LTC1** page configure the cell to compare mode:

- Choose **Enable LTC1 cell after initialization (CEN)**,
- Select **Compare with last timer (=LTC0)**,
- Check **Enable low level of 'Select Line input' Si (SOL)** and **Enable high level of 'Select Line input' Si (SOH)** to enable Compare mode in all cases,
- Set the **LTC1 register value (X)** to the period value. Since the timer LTC0 resets to -1 ($FFFF_h$) and then counts through 0 up to (and including) the compare value, the compare value used to generate a periodic interval is somewhat different than what might be expected. To generate a periodic interval of P clocks, the compare value must be set to (P-2). To get a symmetric PWM P must be even. P = 37.5MHz/20KHz = 1875. The period P will be choosen as the next higher even value 1876. The register is set to 1874 = 752_h. Enter **0x0752** in the text field and type **Return**.

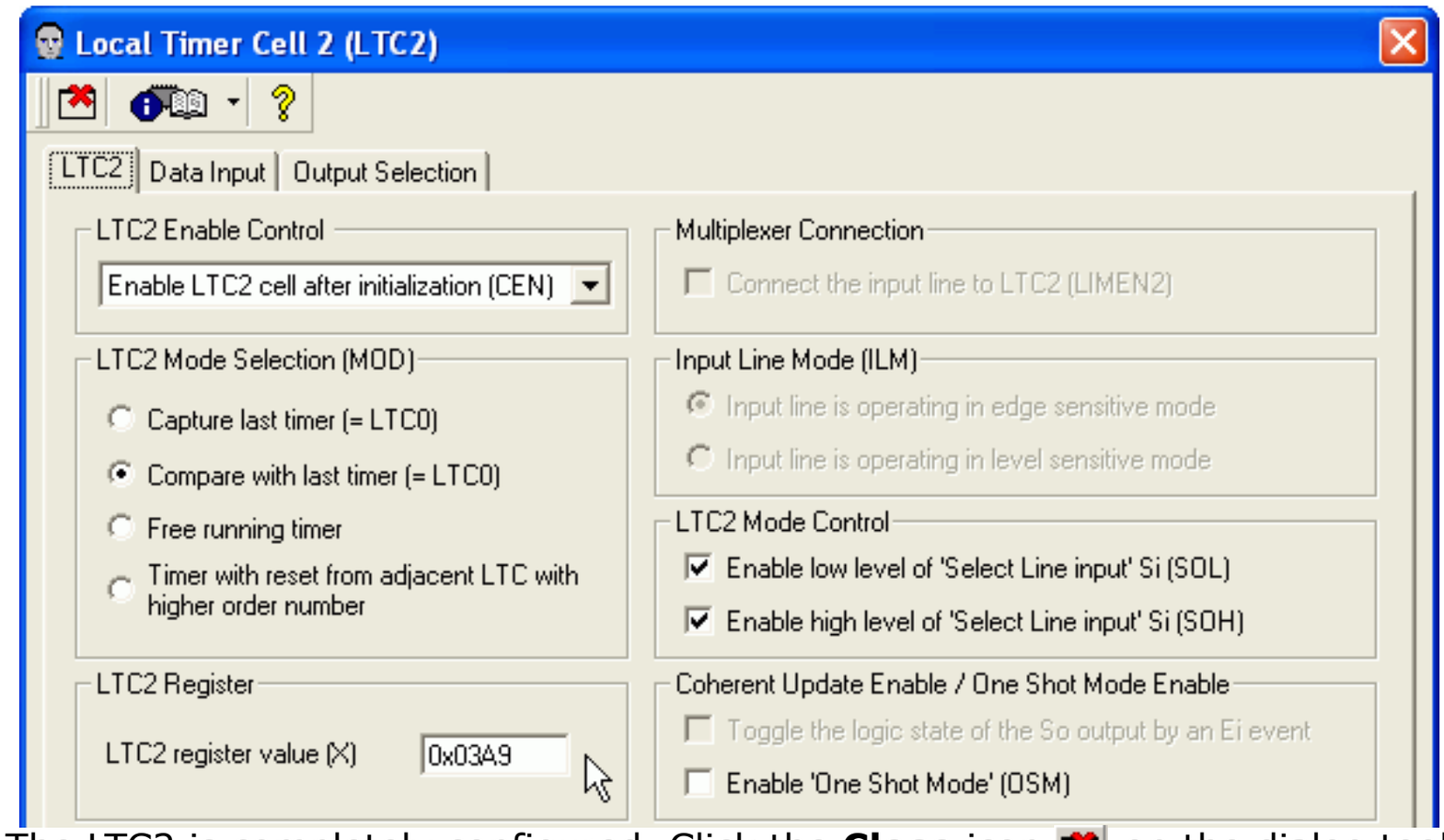


The LTC1 is completely configured. Click the **Close** icon on the dialog toolbar to close the **LTC1** dialog.

Exercise 7: PWM

Click **LTC2** on the **Local Timer** page. On the **LTC2** page and configure the cell to compare mode:

- Choose **Enable LTC2 cell after initialization (CEN)**,
- Select **Compare with last timer (=LTC0)**,
- Check **Enable low level of 'Select Line input' Si (SOL)** and **Enable high level of 'Select Line input' Si (SOH)** to enable Compare mode in all cases,
- Set the **LTC2 register value** to the mid period value -1+1876/2 = 937 = $3A9_h$. Enter **0x03A9** in the text field and type **Return**.

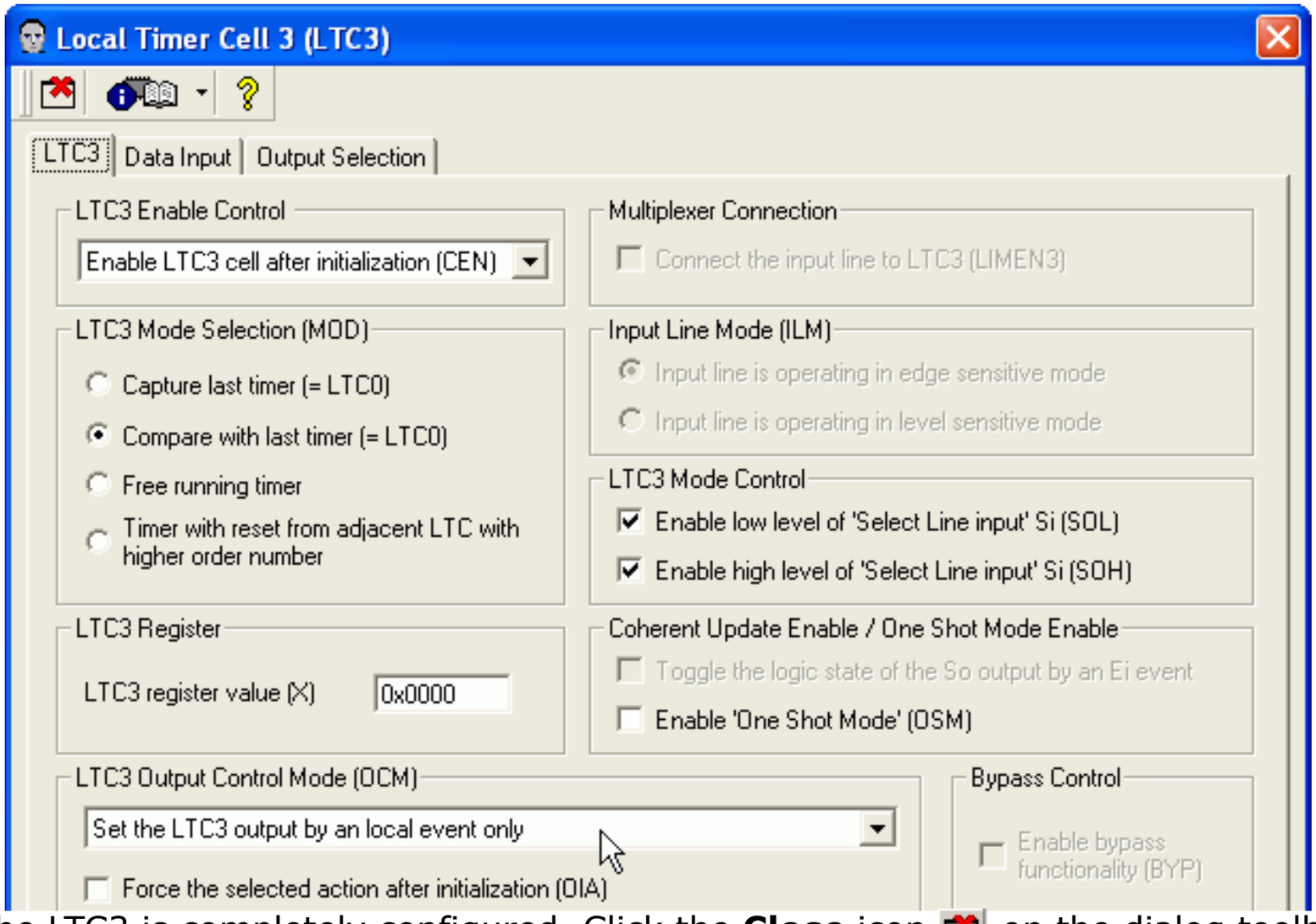


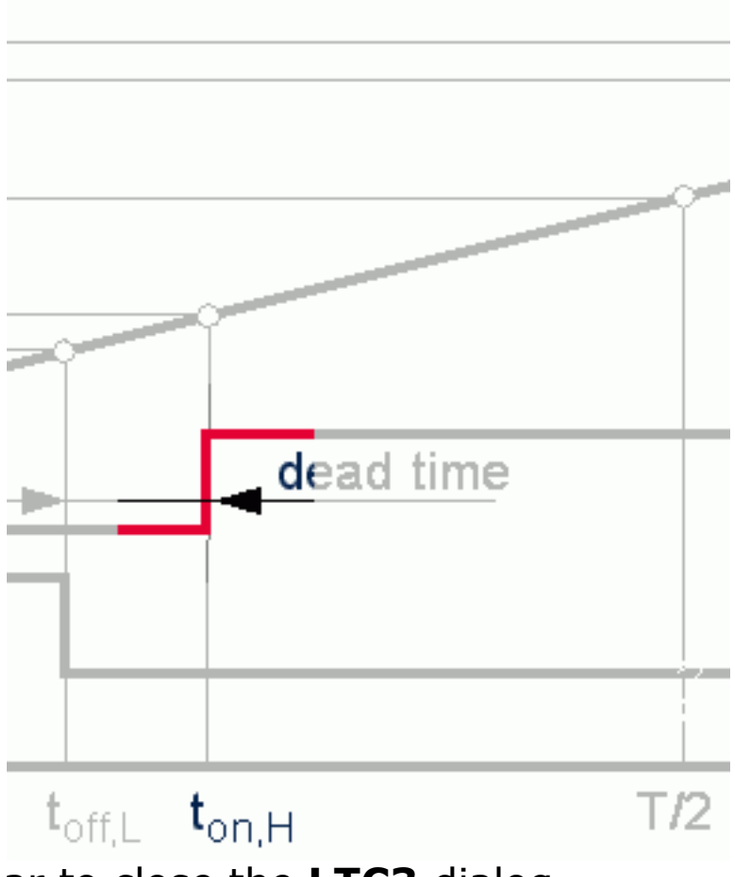
The LTC2 is completely configured. Click the **Close** icon on the dialog toolbar to close the **LTC2** dialog.

Exercise 7: PWM

To configure the phase U using LTC3/LTC4 and LTC5/LTC6 click **LTC3** on the **Local Timer** page.
On the **LTC3** page configure the cell to compare mode:

- Choose **Enable LTC3 cell after initialization (CEN)**,
- Select **Compare with last timer (=LTC0)**,
- Check **Enable low level of 'Select Line input' Si (SOL)** and **Enable high level of 'Select Line input' Si (SOH)** to enable Compare mode in all cases,
- Choose **Set the LTC3 output by an local event only** as the **Output Control Mode** to set the output to high on a compare match. The **LTC3 register value** will be calculated in the mid period interrupt routine.



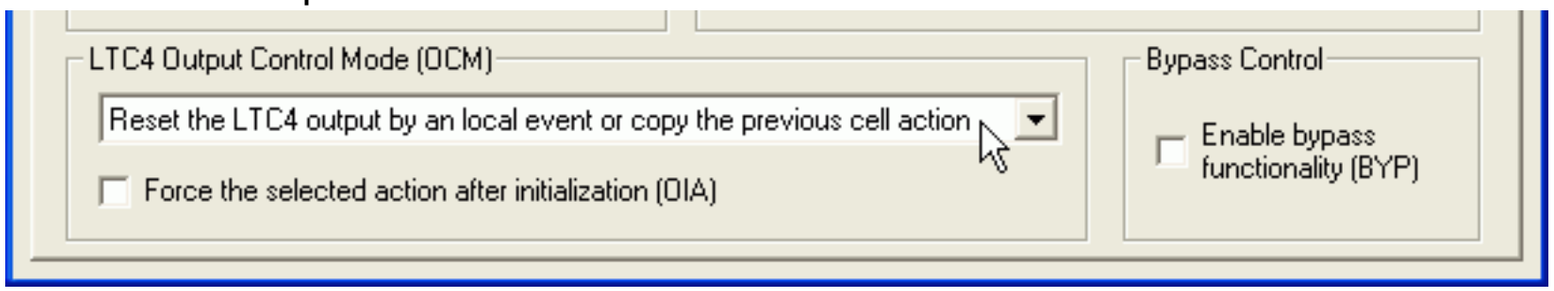


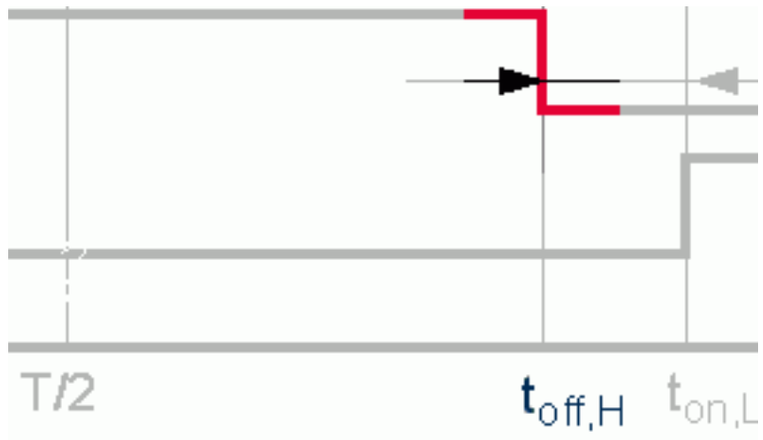
The LTC3 is completely configured. Click the **Close** icon on the dialog toolbar to close the **LTC3** dialog.

Exercise 7: PWM

Open the LTC4 to LTC6 and configure them also in compare mode

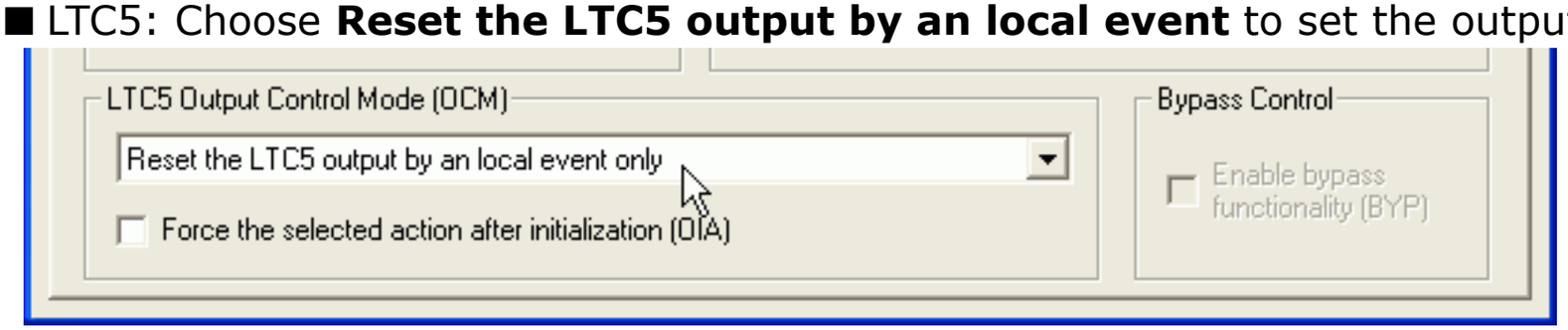
- LTC4: Choose **Reset the LTC4 output by an local event or copy the previous cell action** to set the output to passive on a compare match.

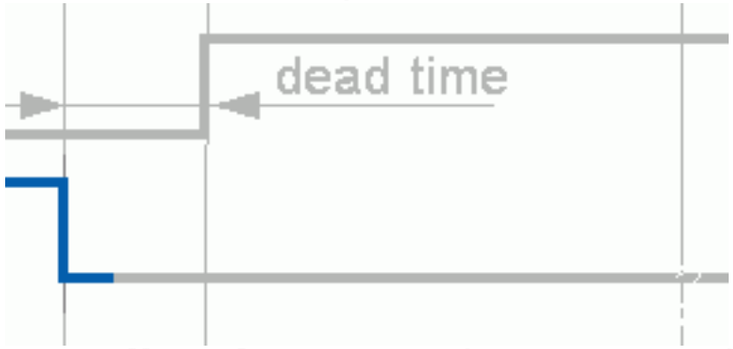




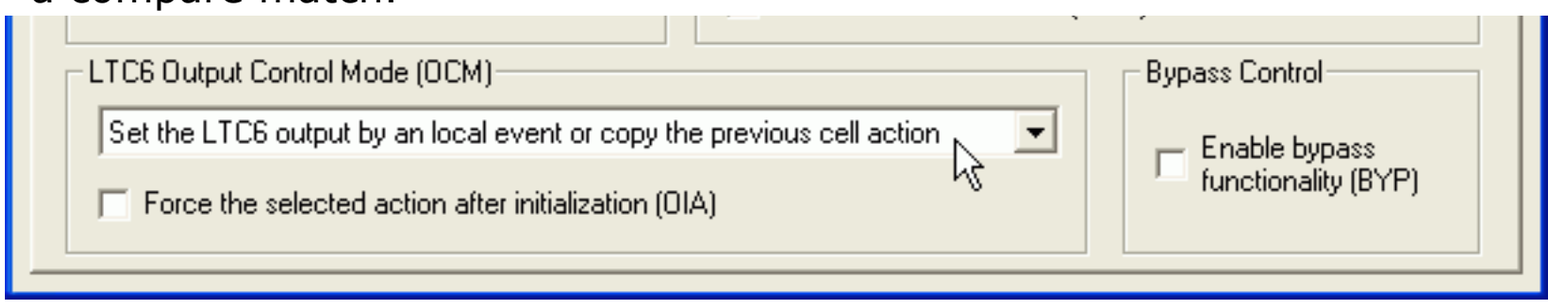
To configure the low side of phase U using LTC5/LTC6 configure the **Output Control Mode**:

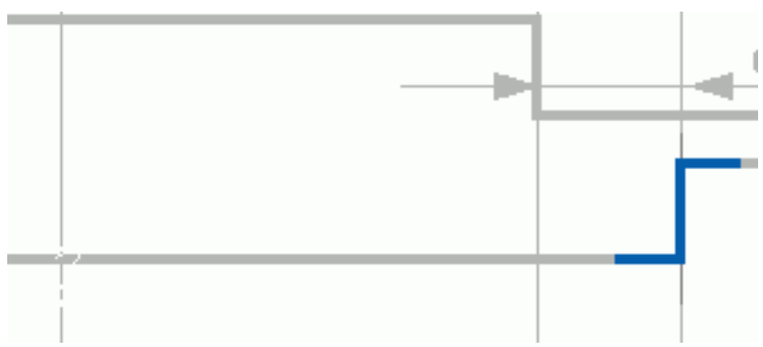
- LTC5: Choose **Reset the LTC5 output by an local event** to set the output to low on a compare match,





- LTC6: Choose **Set the LTC6 output by an local event or copy the previous cell action** to set the output to high on a compare match.



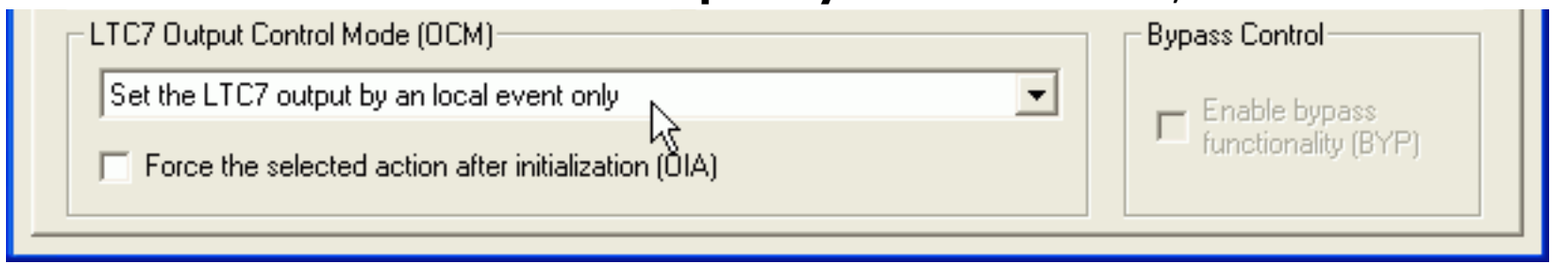


The high side and the low side of phase U are configured. Close the **LTC3** to **LTC6** dialogs.

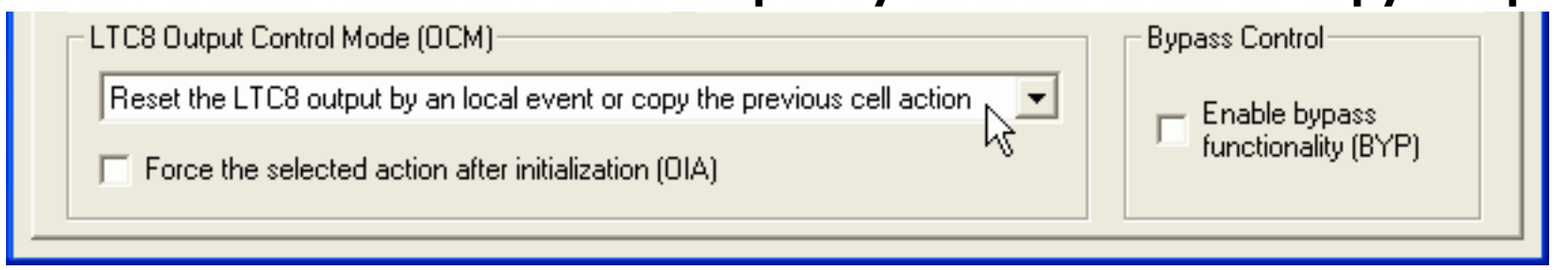
Exercise 7: PWM

For phase V open cell LTC7 to LTC10 and configure them also in compare mode:

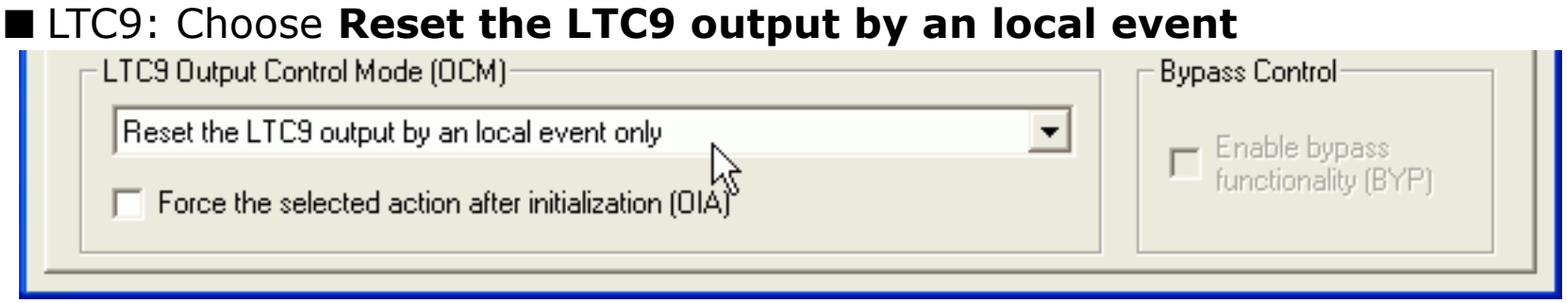
- LTC7: Choose **Set the LTC7 output by an local event**,



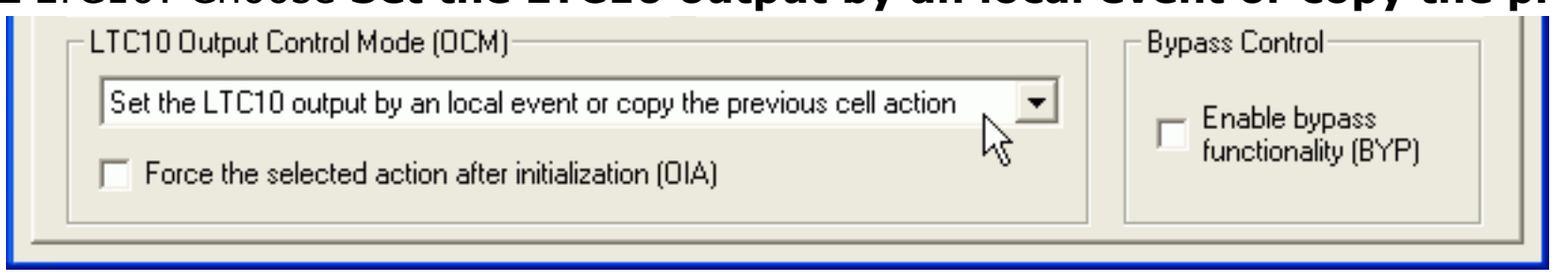
- LTC8: Choose **Reset the LTC8 output by an local event or copy the previous cell action**,



- LTC9: Choose **Reset the LTC9 output by an local event**



- LTC10: Choose **Set the LTC10 output by an local event or copy the previous cell action**.

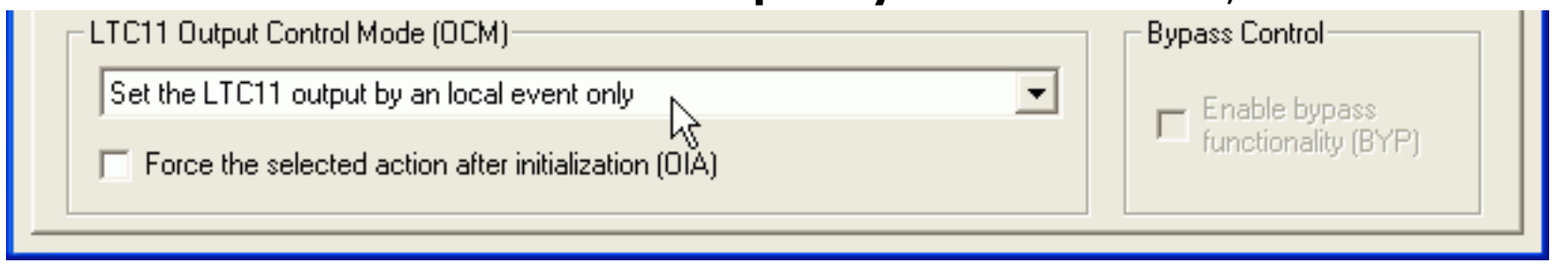


The high side (LTC7/LTC8) and the low side (LTC9/LTC10) of phase V are configured. Close the **LTC7** to **LTC10** dialogs.

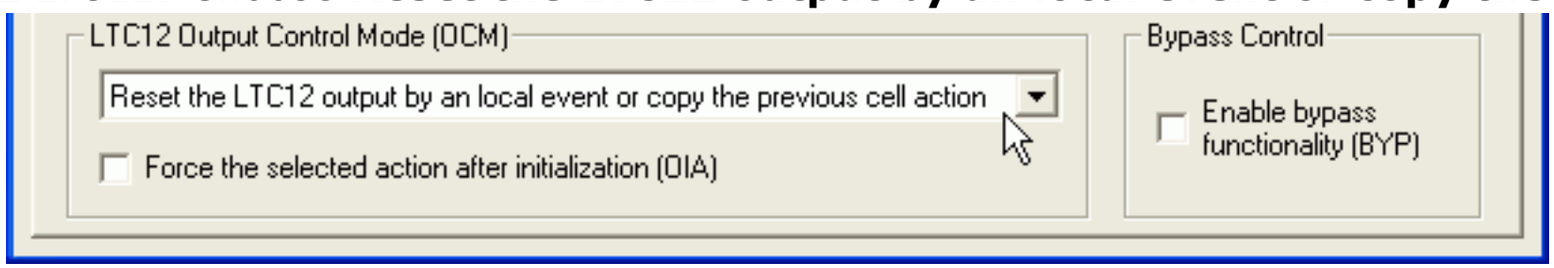
Exercise 7: PWM

For phase W open cell LTC11 to LTC14 and configure them also in compare mode:

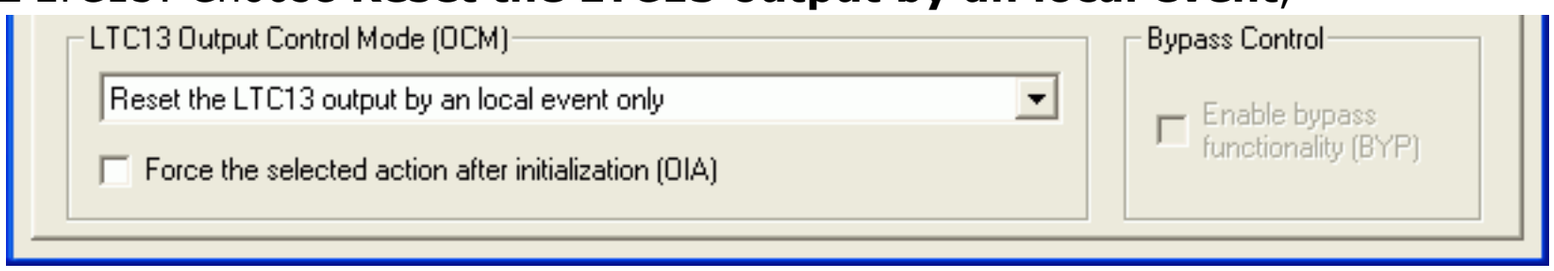
- LTC11: Choose **Set the LTC11 output by an local event**,



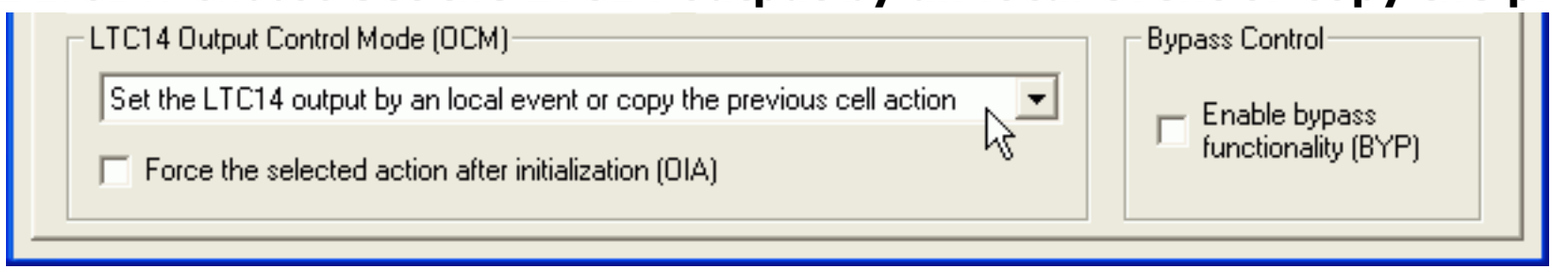
- LTC12: Choose **Reset the LTC12 output by an local event or copy the previous cell action**,



- LTC13: Choose **Reset the LTC13 output by an local event**,



- LTC14: Choose **Set the LTC14 output by an local event or copy the previous cell action**.

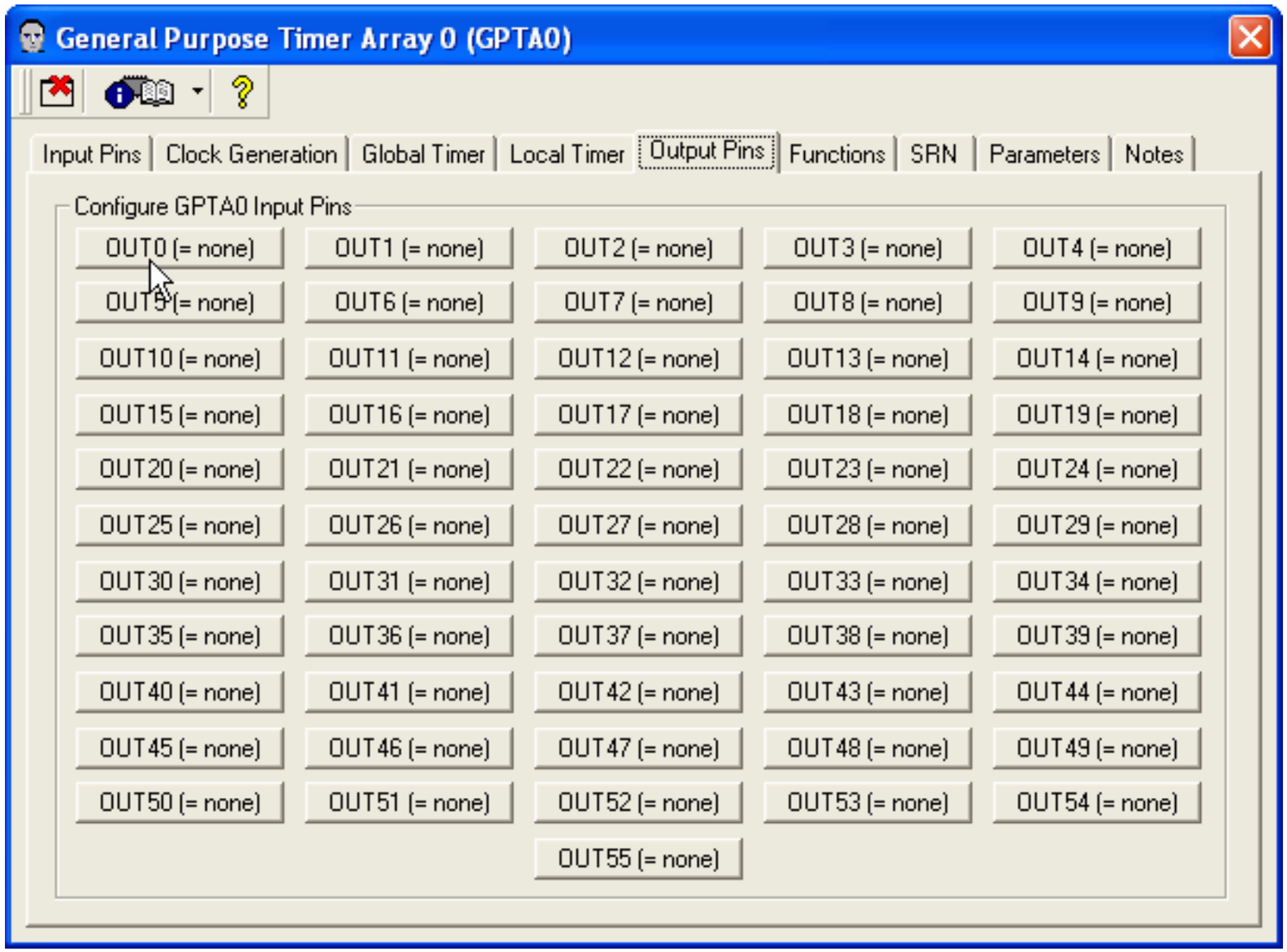


The high side (LTC11/LTC12) and the low side (LTC13/LTC10) of phase W are configured. Close the **LTC11** to **LTC14** dialogs.

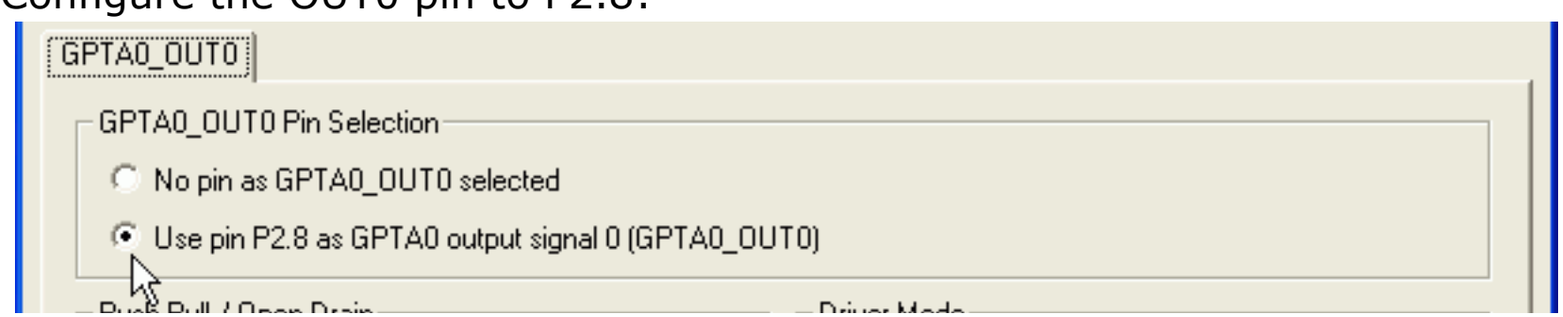
To connect the data out of the cells to the port the **Output Pins** has to be configured.

Exercise 7: PWM

On the **Output Pins** page open the **OUT0**, **OUT1**, **OUT8**, **OUT9**, **OUT10**, **OUT11** properties.

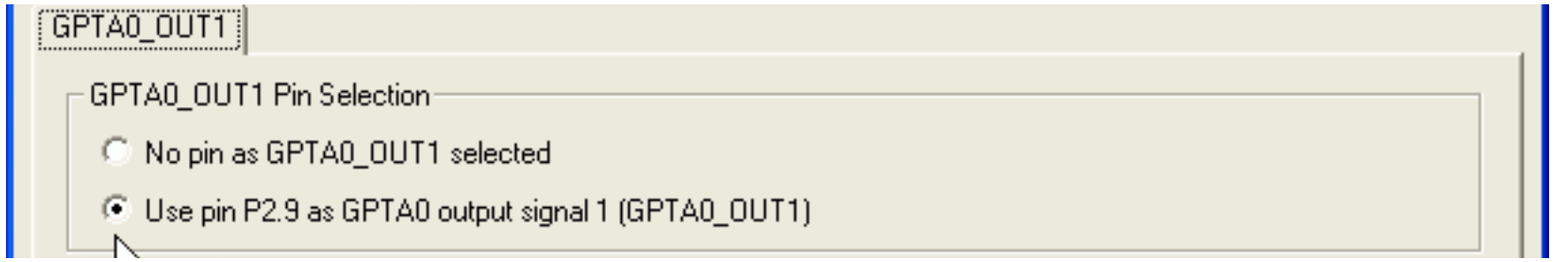


Configure the OUT0 pin to P2.8:

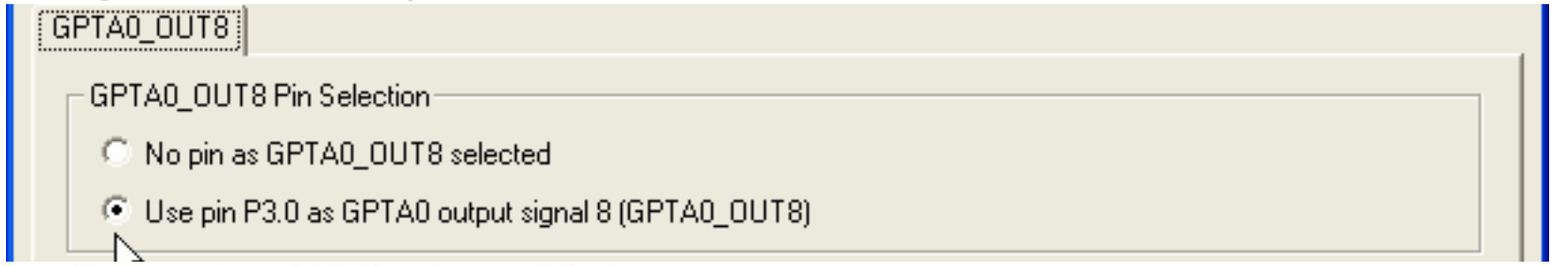


Exercise 7: PWM

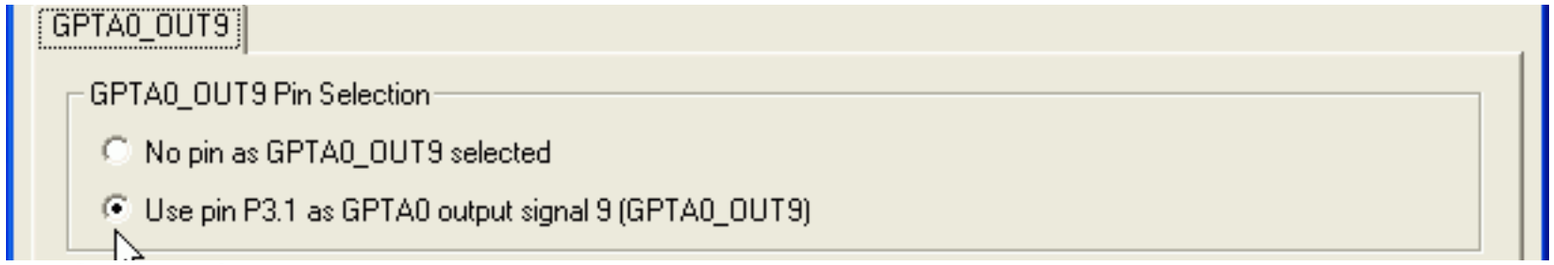
Configure the OUT1 pin to P2.9:



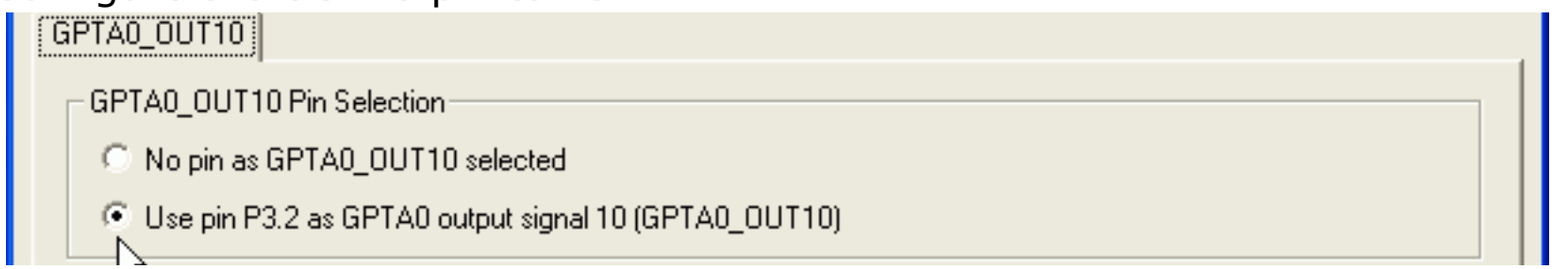
Configure the OUT8 pin to P3.0:



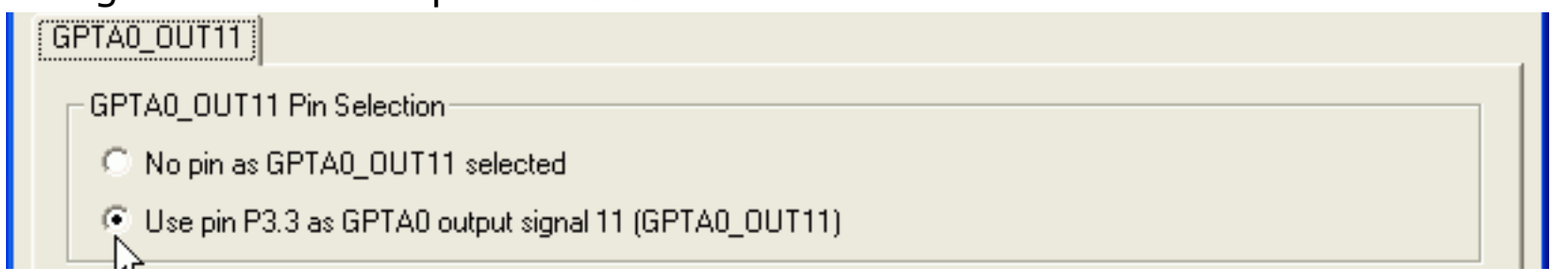
Configure the OUT9 pin to P3.1:



Configure the OUT10 pin to P3.2:

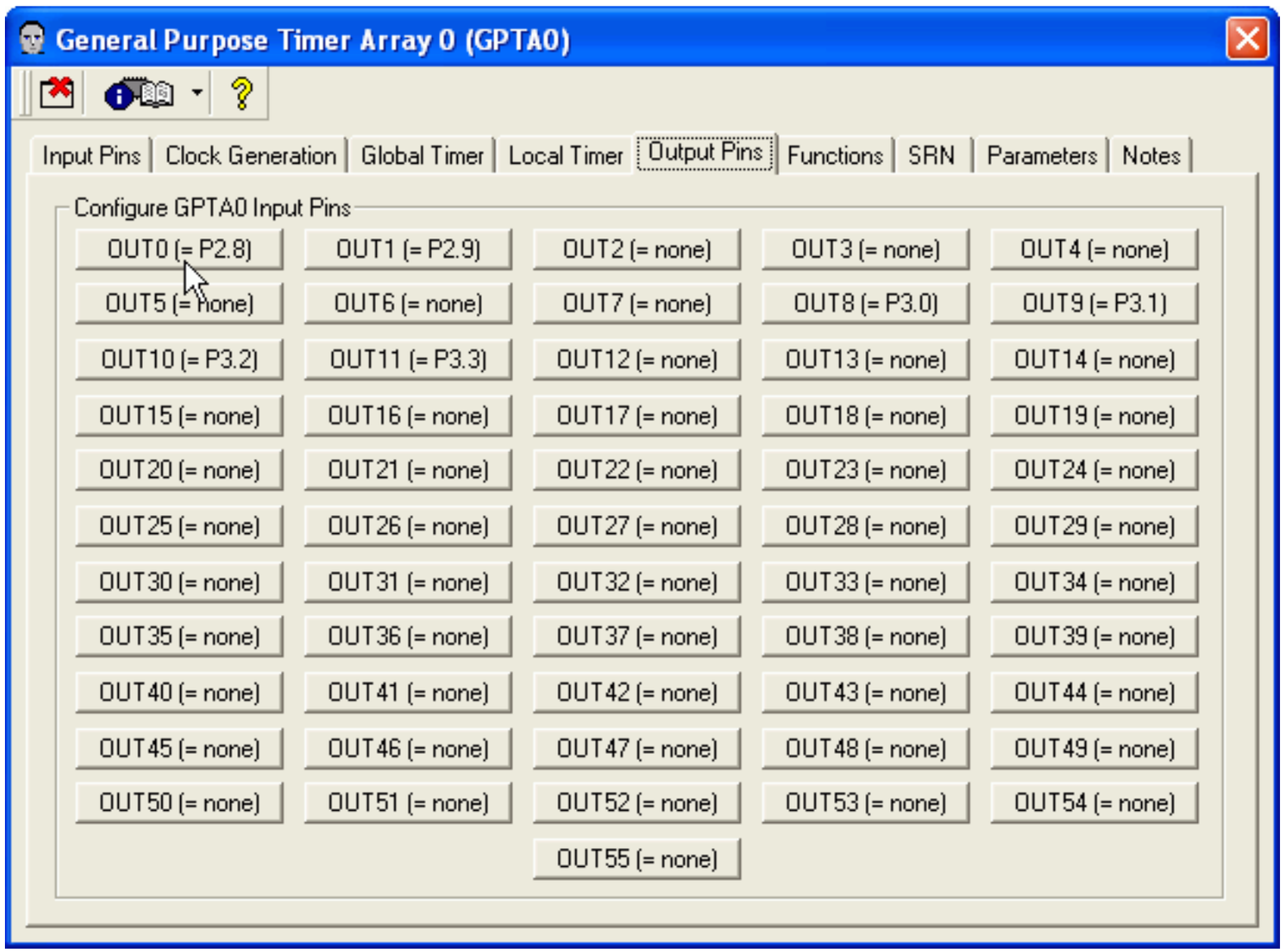


Configure the OUT11 pin to P3.3:



Exercise 7: PWM

Close the 6 **Configure Alternate Pin Functions** dialogs.
The **Output Pins** page displays the complete configuration of the output pins.

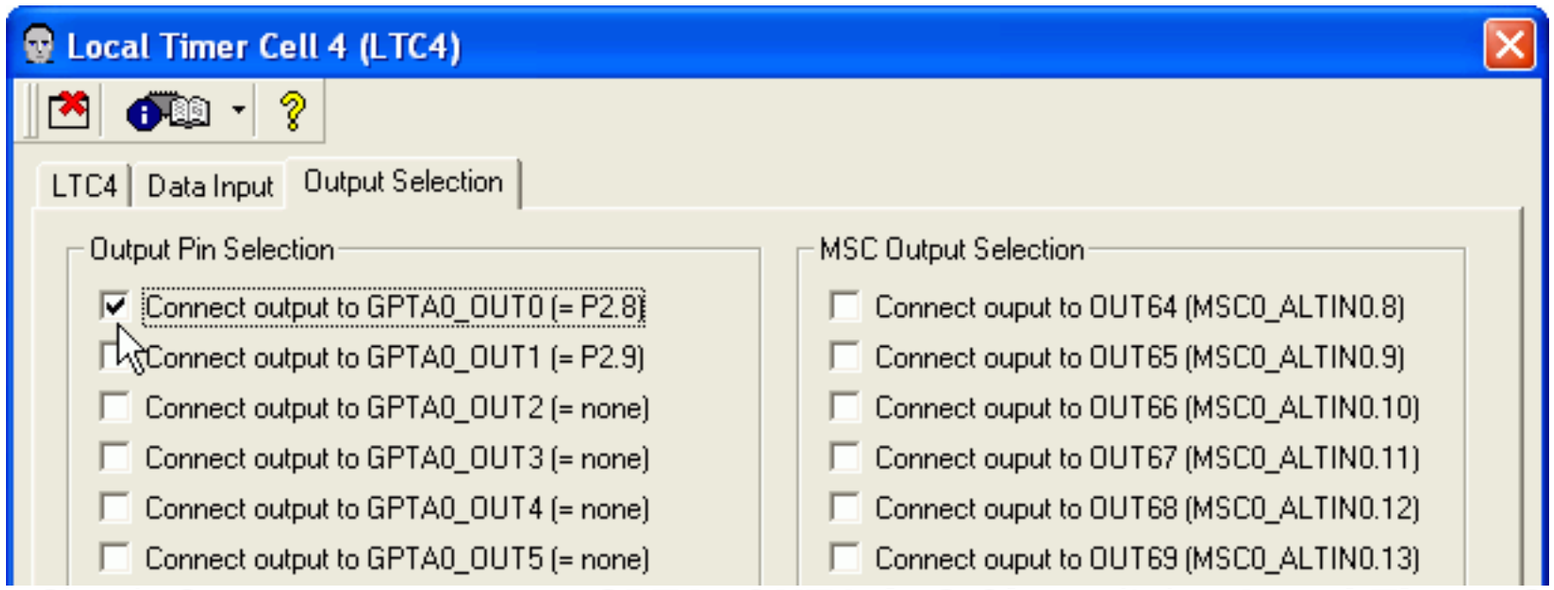


Select the **Local Timer** page again and click **LTC4**, **LTC6**, **LTC8**, **LTC10**, **LTC12**, **LTC14**.

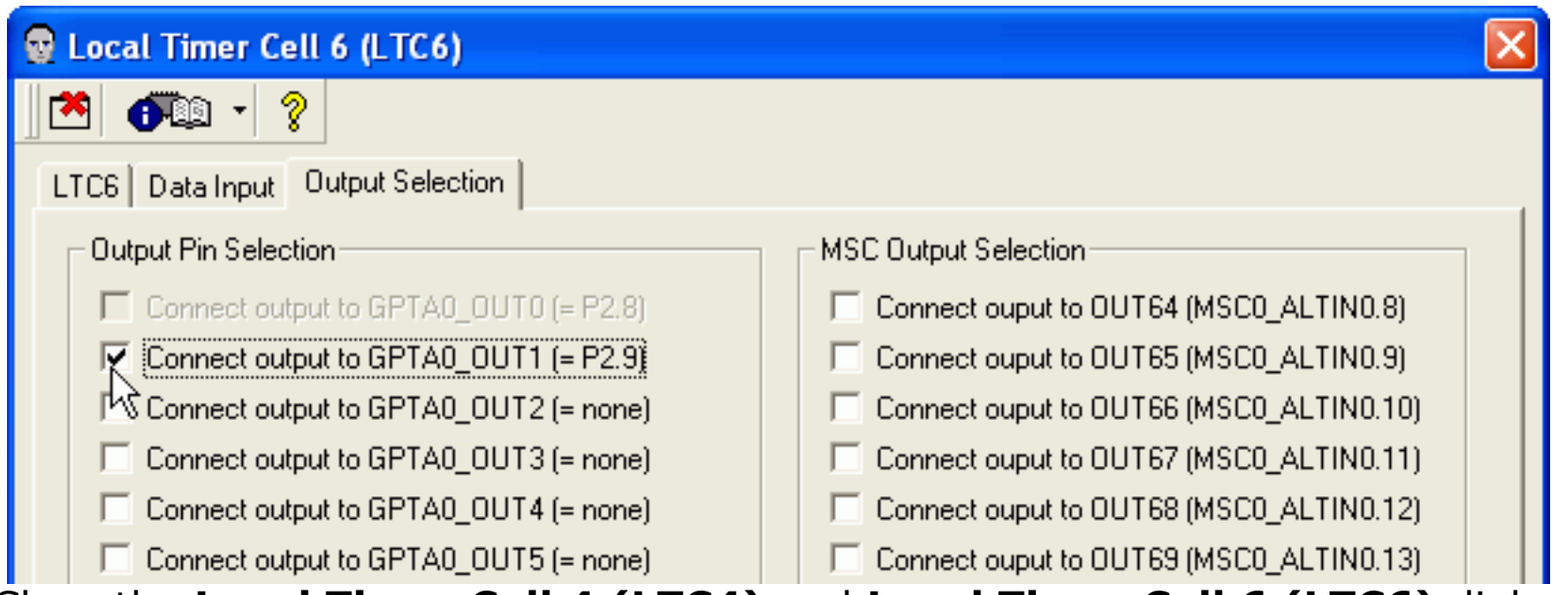
Exercise 7: PWM

On the **Output Selection** page of each dialog

■ Check **Connect output to GPTA_OUT0 (=2.8)** on dialog **Local Timer Cell 4** for the high side of phase U.



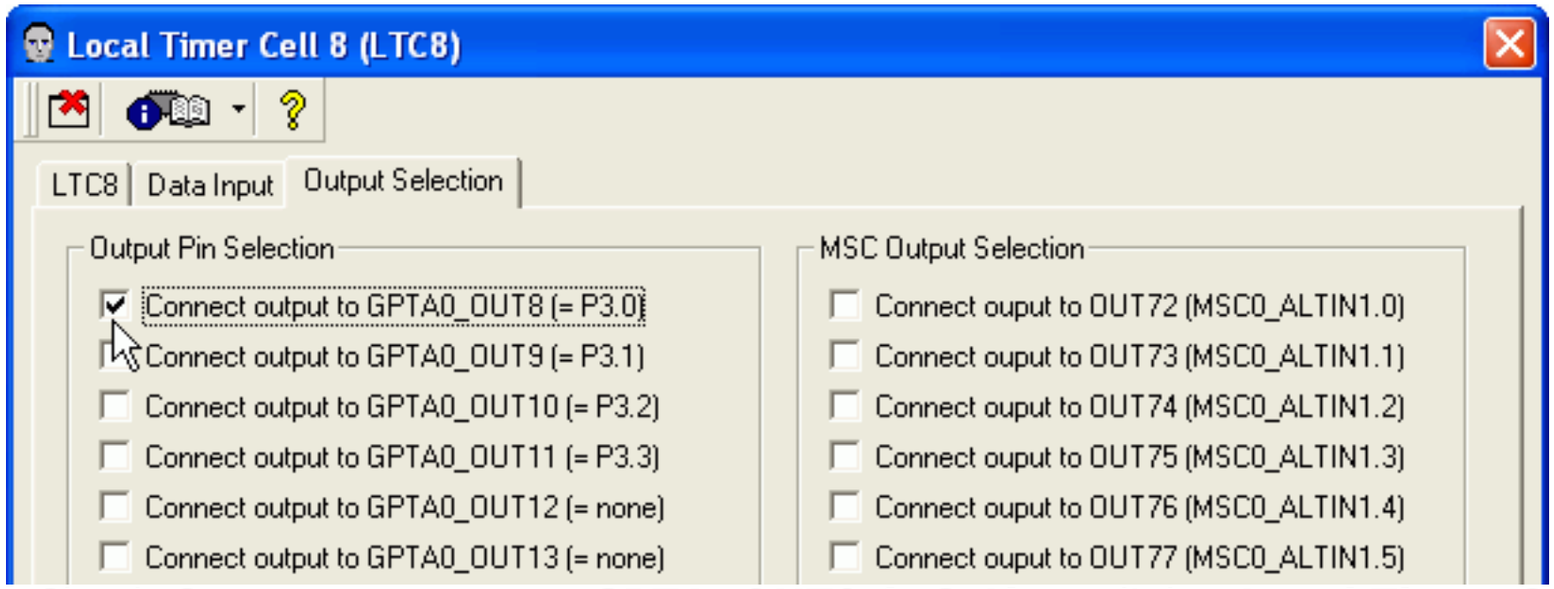
■ Check **Connect output to GPTA_OUT1 (=2.9)** on dialog **Local Timer Cell 6** for the low side of phase U.



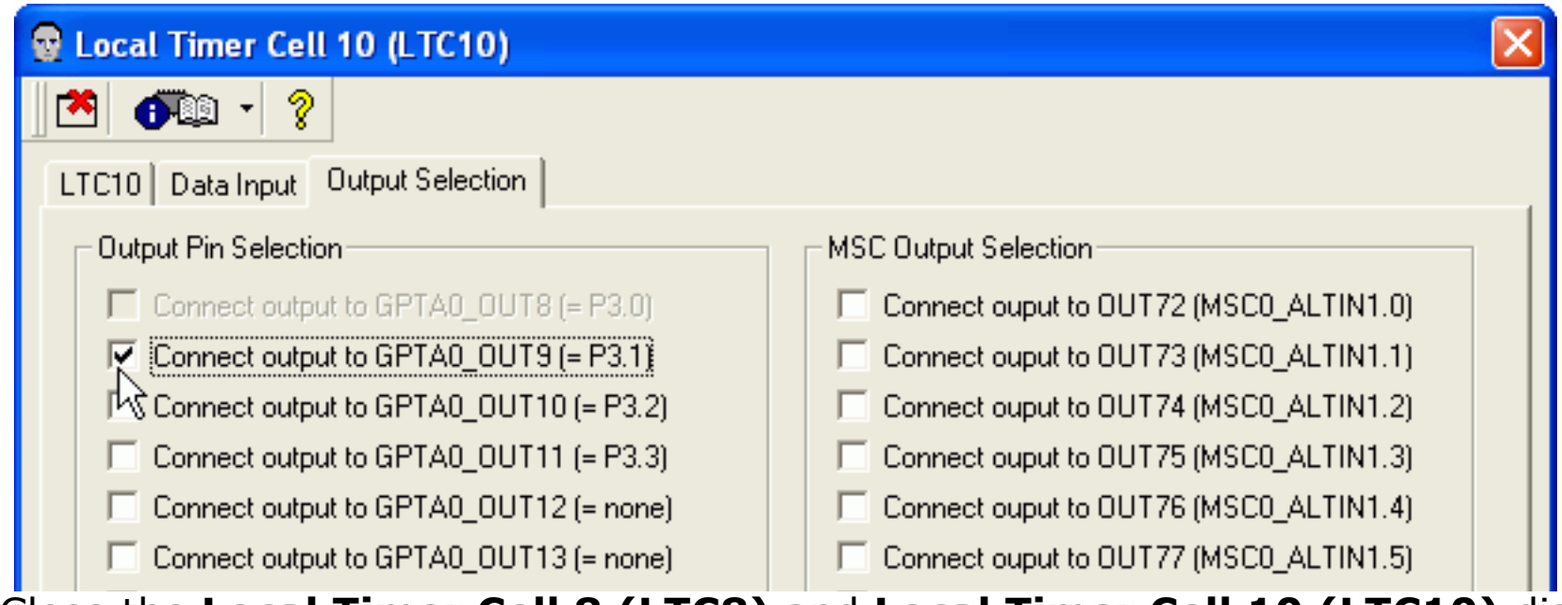
Close the **Local Timer Cell 4 (LTC4)** and **Local Timer Cell 6 (LTC6)** dialogs.

Exercise 7: PWM

■ Check **Connect output to GPTA_OUT8 (=3.0)** on dialog **Local Timer Cell 8** for the high side of phase V.



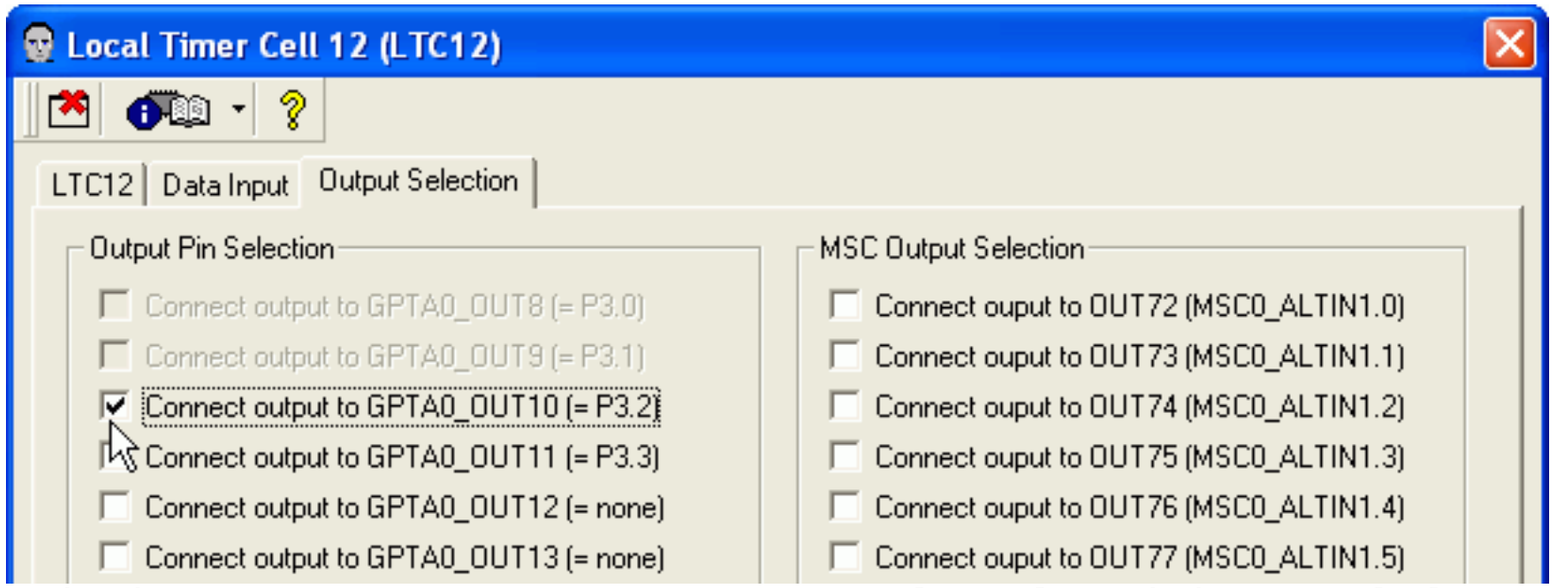
■ Check **Connect output to GPTA_OUT9 (=3.1)** on dialog **Local Timer Cell 10** for the low side of phase V.



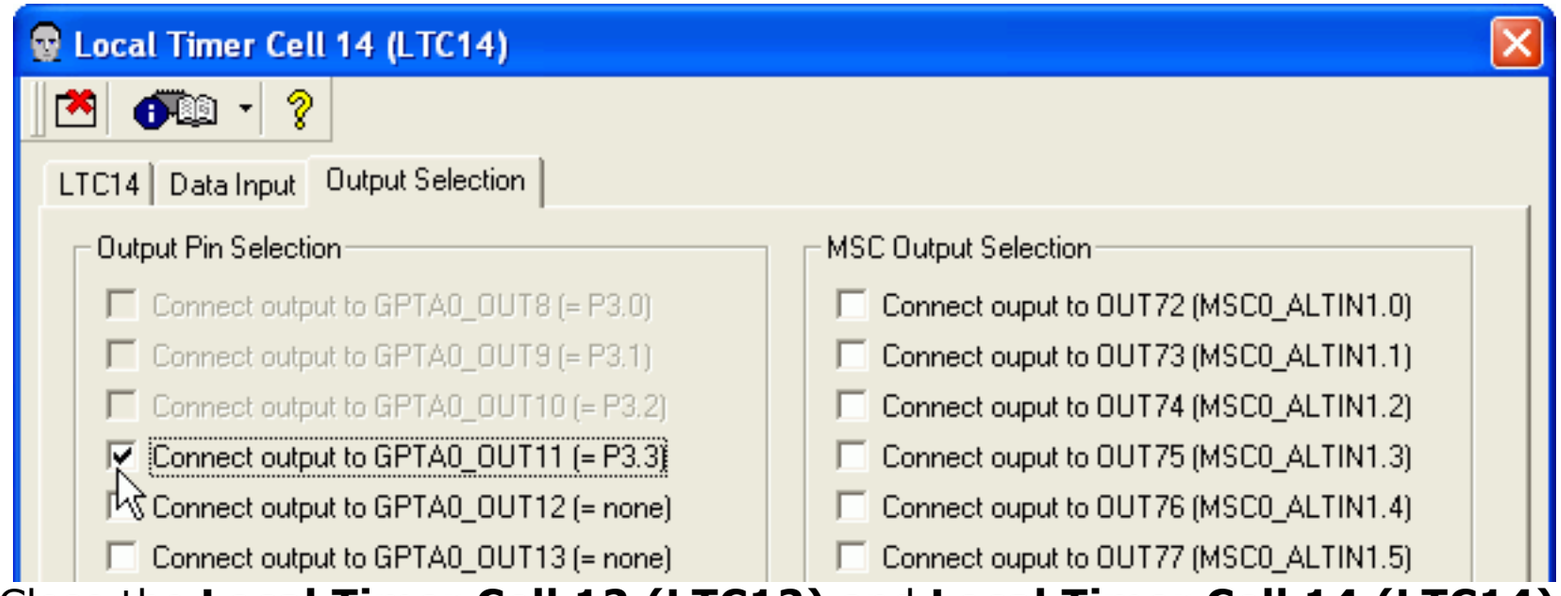
Close the **Local Timer Cell 8 (LTC8)** and **Local Timer Cell 10 (LTC10)** dialogs.

Exercise 7: PWM

■ Check **Connect output to GPTA_OUT10 (=3.2)** on dialog **Local Timer Cell 12** for the high side of phase W.



■ Check **Connect output to GPTA_OUT11 (=3.3)** on dialog **Local Timer Cell 14** for the low side of phase W.

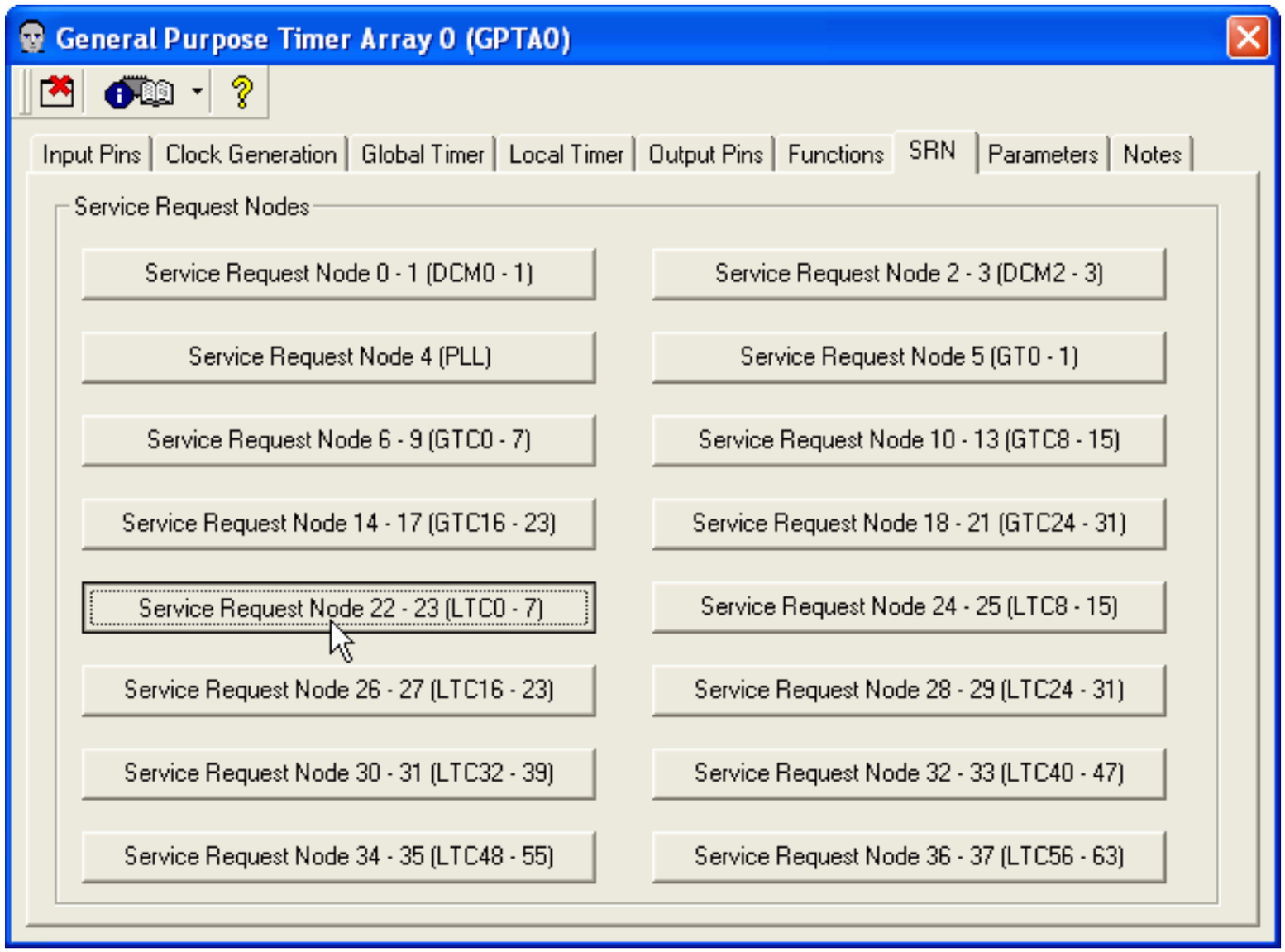


Close the **Local Timer Cell 12 (LTC12)** and **Local Timer Cell 14 (LTC14)** dialogs.

Exercise 7: PWM

7. Set up the GPTA0 properties. Service Request Node.

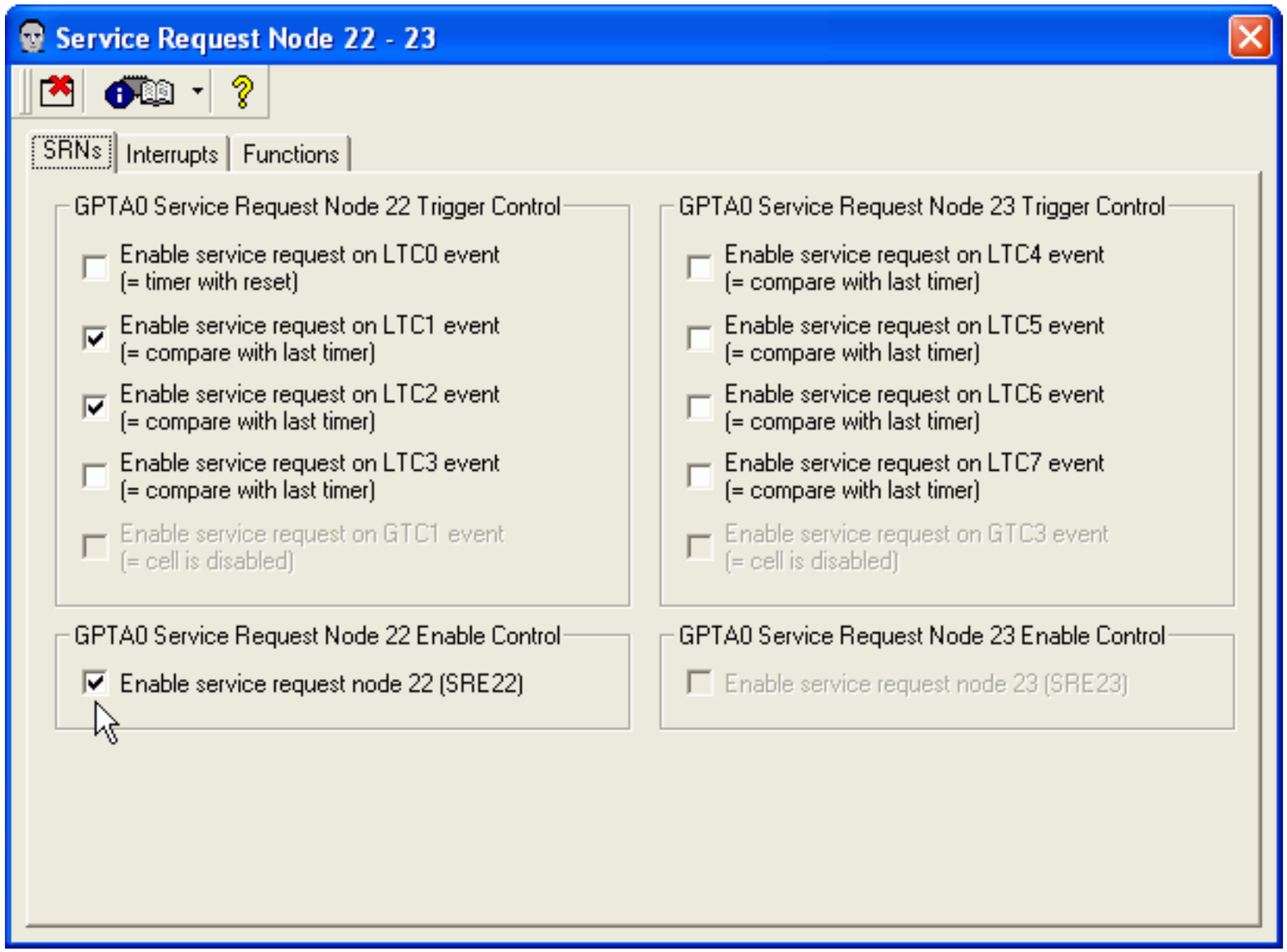
LTC1 and LTC2 are configured in period and mid period compare mode. To generate an interrupt on a compare match, the appropriate Service Request Node (SRN) has to be configured.
On the **SRN** page click **Service Request Node 22-23 (LTC0-7)**.



Exercise 7: PWM

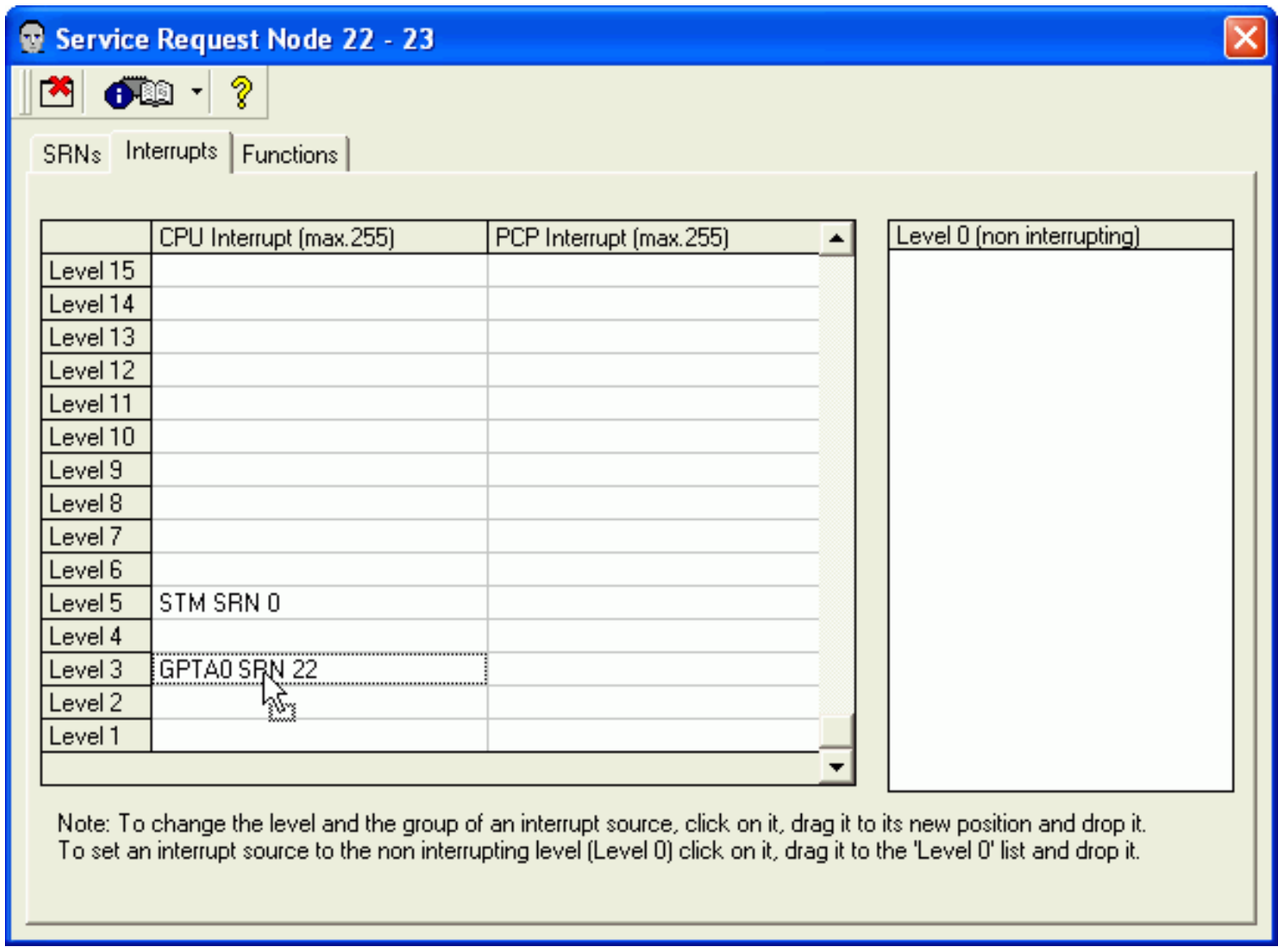
On the **SRNs** page enable the interrupt generation on the period and on the mid period compare match.

- Check **Enable service request on LTC1 event (=compare with last timer)**,
- Check **Enable service request on LTC2 event (=compare with last timer)**,
- Check **Enable service request node 22 (SRE22)**.



Exercise 7: PWM

On the **Interrupts** page and drag **GPTA0 SRN 22** from Level 0 to the CPU Interrupt level 3.

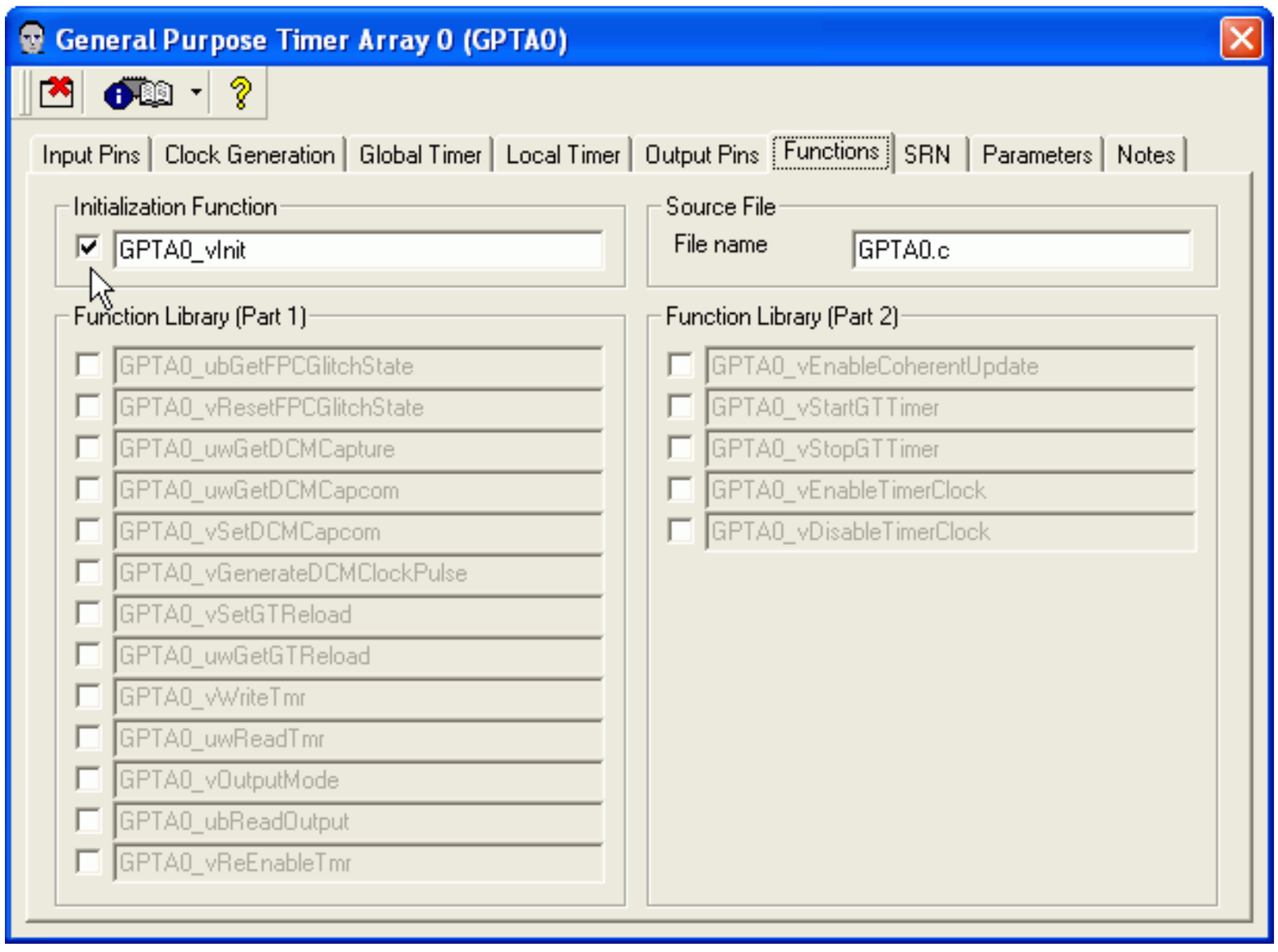


Close the **Service Request Node 22-23** dialog.

Exercise 7: PWM

On the **Functions** page

■ Check the `GPTA0_vInit` function.

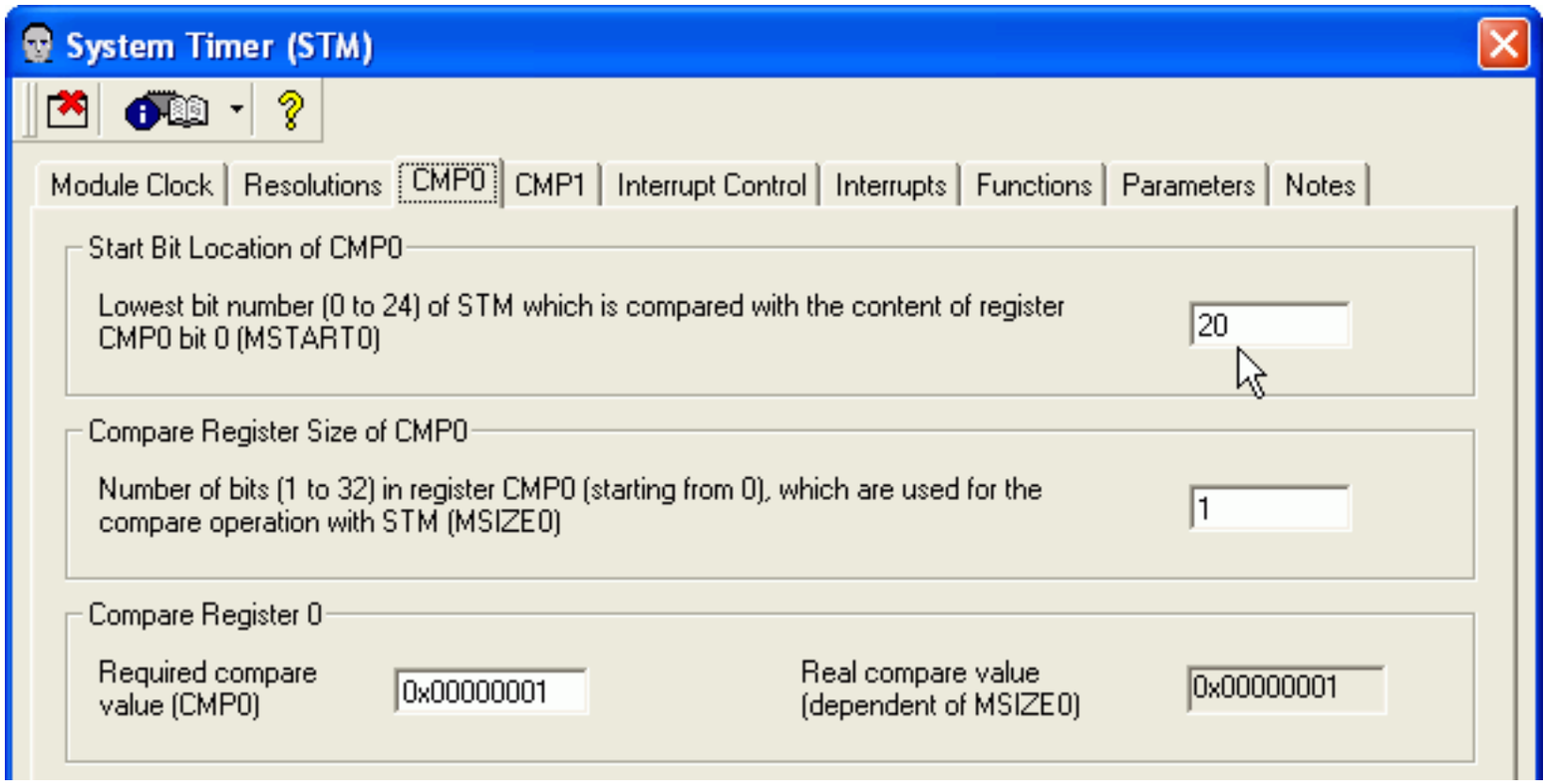


The GPTA is completely configured to generate a complementary 3-phase PWM signal. Click the **Close** icon on the dialog toolbar to close the **General Purpose Timer Array 0 (GPTA0)** dialog.

8. Modify the STM properties.

Open the System timer properties and select the **CMP0** page

- Change the **Start Bit Location of CMP0** to 20



The system timer interrupt is used to update the PWM values.

9. Generate the application framework

Click on the **Generate Code** icon  on the application toolbar to start the code generation process. Save and close the *DAvE* project.

10. Add a new project to the Tasking Workspace

Open the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\pwm\pwm.pjt`.

Exercise 7: PWM

11. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

12. Set current project

Use the context menu in the workspace window to make the pwm project the current project.

13. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter `c:\infineon\tc1796_intmem.opt`.

14. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

15. Add the user code

Add the following code to `GPTA0.c`. Include the standard math library at (GPTA0_General,2).

```
// USER CODE BEGIN (GPTA0_General,2)
#include <math.h>
// USER CODE END
```

Define the following constants at (GPTA0_General,4).

```
// USER CODE BEGIN (GPTA0_General,4)

#define LTC_FREQ                37500000  // = 37.5MHz
#define PWM_FREQ                20000     // = 20.0kHz
#define DEADTIME                1E-6      // = 1µs

#define DEADTIME_CNTS           ((short) (LTC_FREQ * DEADTIME))
#define PWM_PERIOD_CENTER_CNTS  ((short) ((LTC_FREQ / PWM_FREQ) / 2)) // same as GPTA0_LTCXR02

// USER CODE END
```

Exercise 7: PWM

Declare the following global variables at (GPTA0_General,7). For the sinus a look up table is used to save time. Three circular buffers pointers are used to step through the elements of the array. Incrementing a circular buffer pointer that points to the last element results in a pointer to the first element.

```
// USER CODE BEGIN (GPTA0_General,7)
#define SINE_STEPS 360
short  __near __circ duty[SINE_STEPS];  // duty lookup table in a circular buffer
short  __near __circ *dutyU, *dutyV, *dutyW;
const float pi = 3.14159;
// USER CODE END
```

In the `GPTA_vInit` function disable the interrupt for LTC1 and LTC2 at (Init, 3). The PWM will be started by the first STM interrupt.

```
// USER CODE BEGIN (Init,3)
GPTA0_LTCCTR01_REN = 0;    // disable period interrupt
GPTA0_LTCCTR02_REN = 0;    // disable mid period interrupt
// USER CODE END
```

Fill the duty lookup table and initialize the circular buffer pointer at (Init, 4)

```
// USER CODE BEGIN (Init,4)
for(int i=0;i<SINE_STEPS;i++)
  duty[i] =  (PWM_PERIOD_CENTER_CNTS + 1) * 0.5*(sinf(i * 2*pi/SINE_STEPS) + 1);

dutyU = &duty[0];                // set to sin(0°)
dutyV = &duty[SINE_STEPS/3];     // set to sin(120°)
dutyW = &duty[-SINE_STEPS/3];    // set to sin(240°)
// USER CODE END
```

Exercise 7: PWM

Configure the period and mid period interrupts at `GPTA0_viSRN22` in file `GPTA.c`. Add the code for the period and mid period interrupt at (SRN22, 2). At zero percent duty the low side must be set to active, i.e. $t_{on,L}$ should be ignored. This is done by setting $t_{on,L}$ to -2 (=$FFFE_H$), a value that is never reached.

```
// USER CODE BEGIN (SRN22,2)
unsigned int n;
if (GPTA0_LTCCTR01_REN)
{
  GPTA0_LTCCTR01_REN = 0;  // disable period interrupt
  STM_ICR_CMP0EN = 1;  // enable the stm interrupt

  //  update duty U
  n = PWM_PERIOD_CENTER_CNTS + *dutyU;
  GPTA0_LTCXR04 = n;
  GPTA0_LTCXR06 = n + DEADTIME_CNTS;
  //update duty V
  n = PWM_PERIOD_CENTER_CNTS + *dutyV;
  GPTA0_LTCXR08 = n;
  GPTA0_LTCXR10 = n + DEADTIME_CNTS;
  //update duty W
  n = PWM_PERIOD_CENTER_CNTS + *dutyW;
  GPTA0_LTCXR12 = n;
  GPTA0_LTCXR14 = n + DEADTIME_CNTS;

}
...
```

Exercise 7: PWM

```
else
{
  GPTA0_LTCCTR02_REN = 0;  // disable mid period interrupt
  GPTA0_LTCCTR01_REN = 1;  // enable  period interrupt

  // update duty U
  n = PWM_PERIOD_CENTER_CNTS - *dutyU;
  GPTA0_LTCXR03 = n;
  GPTA0_LTCXR05 = (n == PWM_PERIOD_CENTER_CNTS) ? 0xfffe :
                          ((n < DEADTIME_CNTS) ? 0xffff : n - DEADTIME_CNTS);
  // update duty V
  n = PWM_PERIOD_CENTER_CNTS - *dutyV;
  GPTA0_LTCXR07 = n;
  GPTA0_LTCXR09 = (n == PWM_PERIOD_CENTER_CNTS) ? 0xfffe :
                          ((n < DEADTIME_CNTS) ? 0xffff : n - DEADTIME_CNTS);
  // update duty W
  n = PWM_PERIOD_CENTER_CNTS -  *dutyW;
  GPTA0_LTCXR11 = n;
  GPTA0_LTCXR13 = (n == PWM_PERIOD_CENTER_CNTS) ? 0xfffe :
                          ((n < DEADTIME_CNTS) ? 0xffff : n - DEADTIME_CNTS);
  }
  // USER CODE END
```

Exercise 7: PWM

Enable the mid period interrupt in file `MAIN.c` before the forever loop at (Main,9):

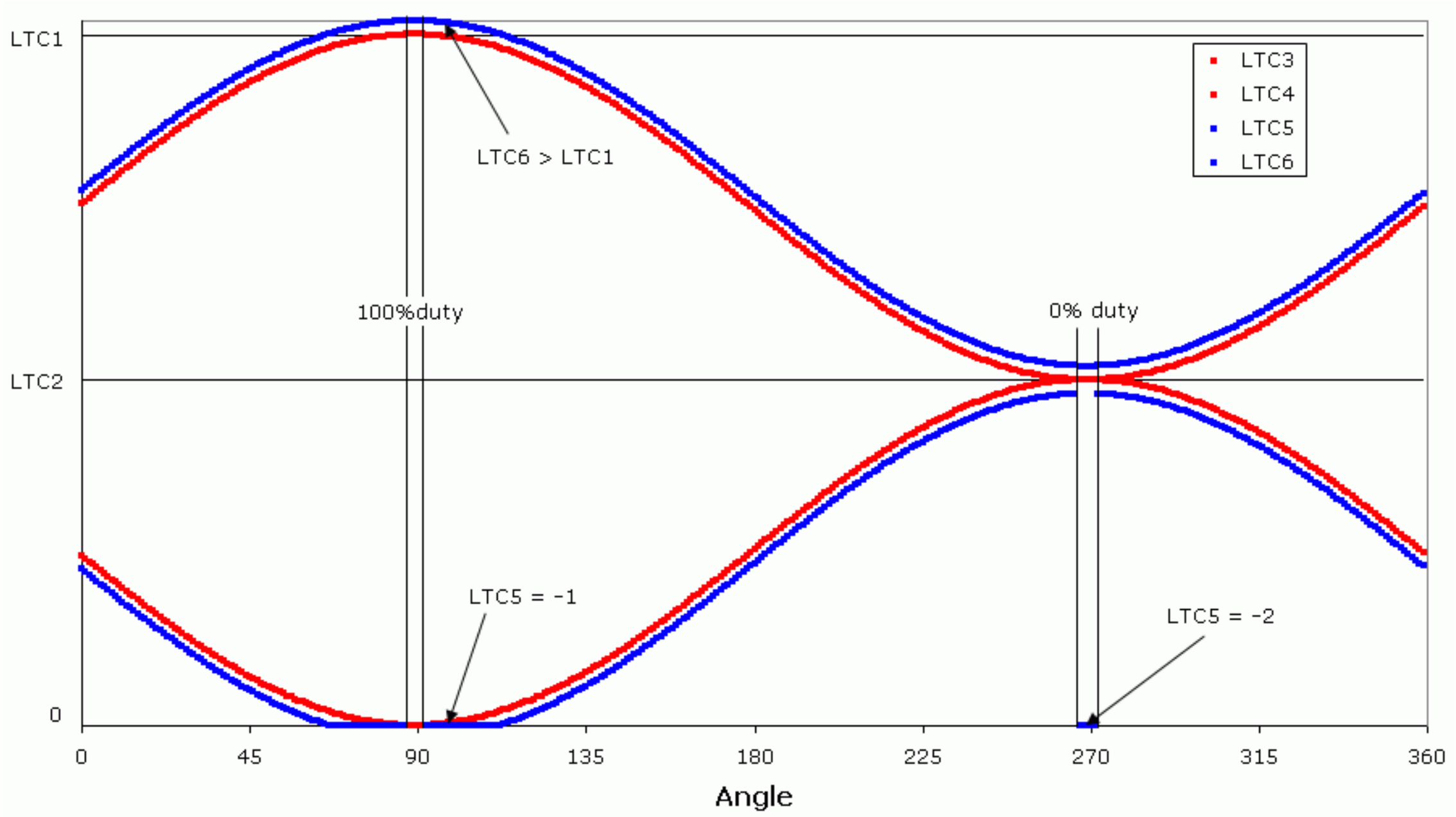
```
// USER CODE BEGIN (Main,9)
STM_ICR_CMP0EN = 1;  // enable the stm interrupt
for(;;)  // forever
    ;
// USER CODE END
```

In file `STM.c` define the `*dutyU, *dutyV, *dutyW` as `extern` and add to function `STM_viSRN0` (SRN0,3):

```
// USER CODE BEGIN (SRN0,3)
STM_ICR_CMP0EN = 0;          // disable the stm interrupt
GPTA0_LTCCTR02_REN = 1;      // enable mid period interrupt
dutyU++; dutyV++; dutyW++;   // circular step through the sinus lookup table
// USER CODE END
```

Exercise 7: PWM

At 0% duty LTC3 and LTC4 have the same compare value and the output is following the action request of the higher cell LTC4, e.g. the high side is set to passive. LTC5 is -2 and therefor ignored, so that the low side is set to active.
At 100% duty LTC5 has a compare value of -1 and is set immediately. LTC6 is larger than the period in LTC1, so that a reset is never happen. I.e. The low side is set to passive.



Exercise 7: PWM

16. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

17. Debug the application

Click the **Debug** icon  on the Build toolbar to open the *CrossView Pro* debugger.

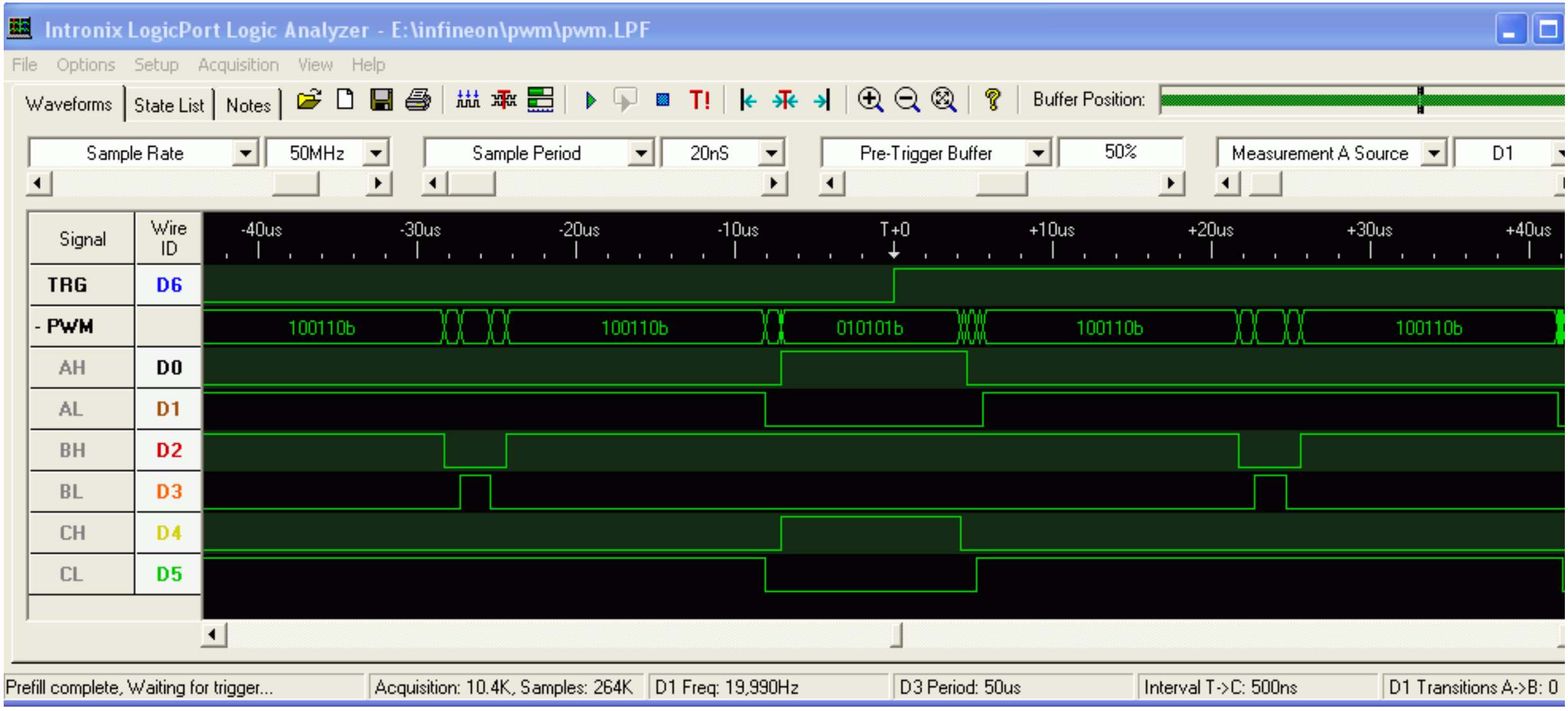
18. Connect an Oscilloscope

Connect the pins P2.8, P2.9, P3.0, P3.1, P3.2, P3.3 to a logic analyzer. Connect P2.10 to the trigger input.

Exercise 7: PWM

19. Run the application

Click the **Run** icon ▶ on the *CrossView Pro* toolbar and see the logic analyzer output.
The cursor is adjusted to measure the dead time.

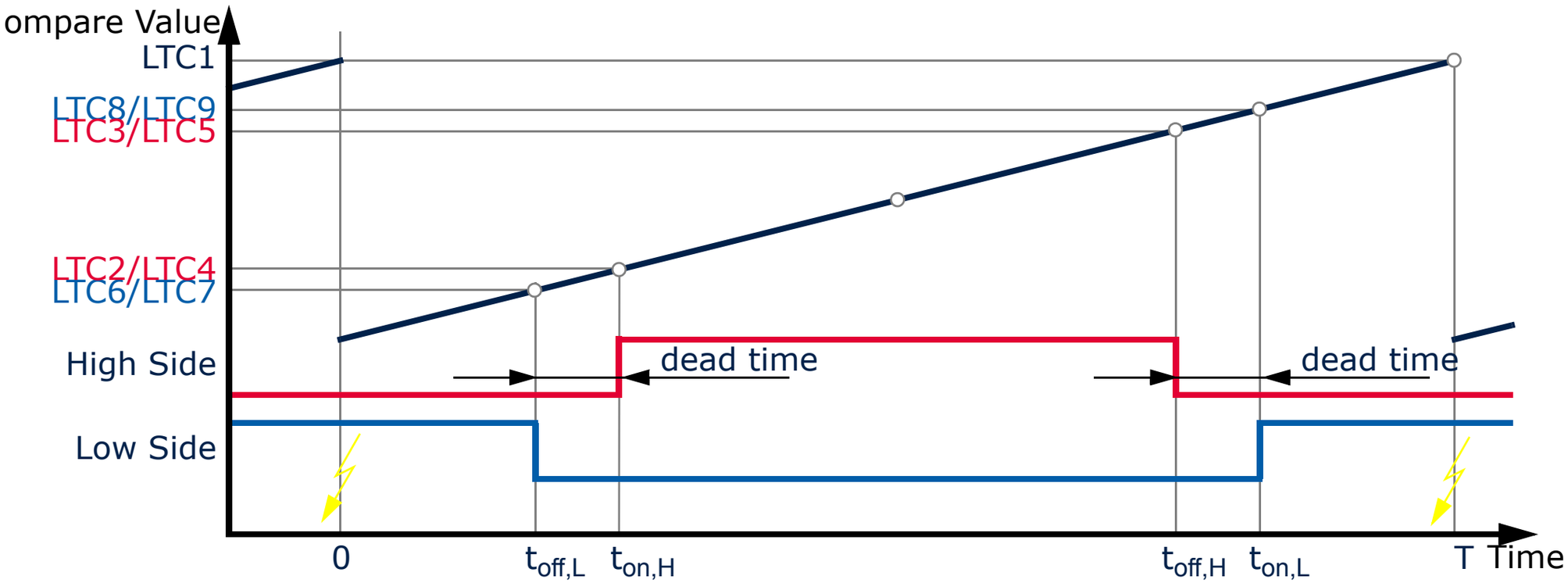


Exercise 7: PWM

PWM with coherent update

The exercise above uses 2 interrupts to set-up the t_{on} and t_{off} values of the low and high side. This system setup might be critical when a interrupt is delayed by more than half a period and the interrupt routine set the new values in the deadtime period. Then only one side will be set to a new value and this might cause a short.
Therefor in systems with heavy interrupt load another configuration is needed. Instead of using shadow registers which are not available within the LTC a second set of LTCs can be set-up and the reset timer switches by HW from on set to the other. This is called coherent update (CUD). The exercise solutions contains a project pwm_cud which implements such a solution. This solution uses twice as much LTCs for each PWM, so that for a complementary 3-phase PWM 26 cells are used instead of 15.

Exercise 7: PWM

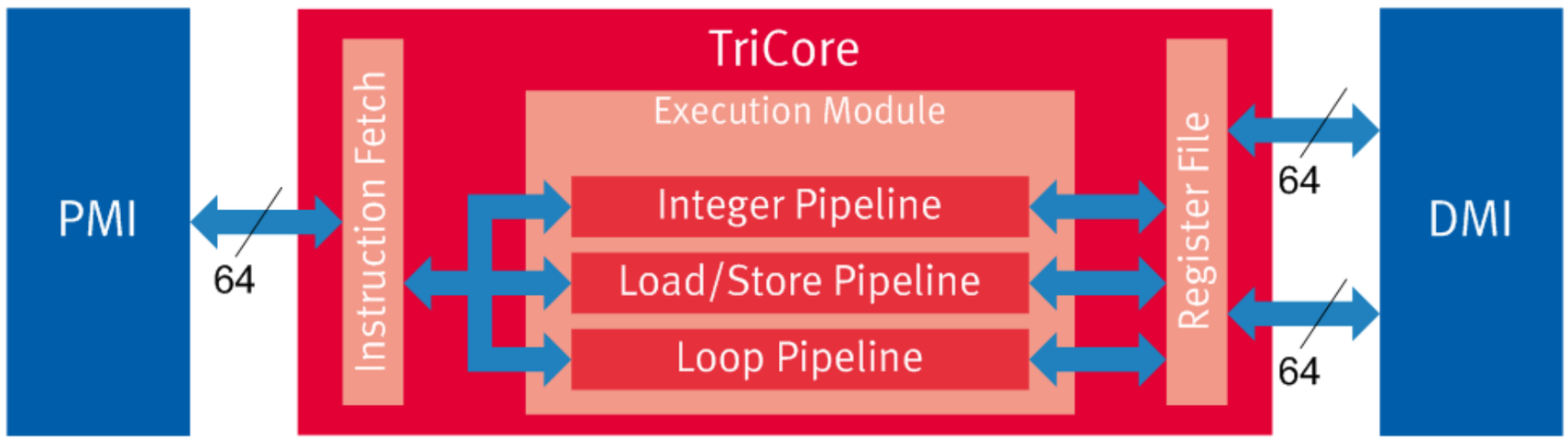


	Cell	SOL	SOH	Output Control Mode	Port
Reset Timer	LTC0		-	Toggle select line (SL) by HW on reset, Interrupt on reset to set-up $t_{on,H}$,$t_{off,H}$,$t_{on,L}$,$t_{on,L}$	-
Compare Period	LTC1	1	1	-	-
Compare High Side On	LTC2	1	0	Set output by a local event	-
Compare High Side Off	LTC3	1	0	Reset output by a local event or copy the previous cell action	-
Compare High Side On	LTC4	0	1	Set output by a local event or copy the previous cell action	-
Compare High Side Off	LTC5	0	1	Reset output by a local event or copy the previous cell action	P2.8
Compare Low Side Off	LTC6	1	0	Reset output by a local event	-
Compare Low Side On	LTC7	1	0	Set output by a local event or copy the previous cell action	-
Compare Low Side Off	LTC4	0	1	Reset output by a local event or copy the previous cell action	-
Compare Low Side On	LTC5	0	1	Set output by a local event or copy the previous cell action	P3.0

Exercise 8: Pipelining

Pipelining

In this exercise you will develop an application that uses efficiently the super-scalar pipeline TriCore™ architecture. The implementation is done in assembler, so that the user learns to write and optimize a small routine in assembler. The exercise shows that with a basic understanding of the TriCore™ architecture and the TriCore™ instruction set algorithm execution speed can be considerable increased.



Exercise 8: Pipelining

Software pipelining means starting an equation before the previous equation has finished. This is achieved with knowledge of the TriCore pipelining rules.

Example: Calculate the sum of an array with N+1 elements.

$$sum = \sum_{i=0}^{N} X_i$$

C Implementation

```
int sum0(int* X, int N)
{
  int sum = 0;

  for(int i=0;i<=N;i++)
      sum += X[i];

  return sum;
}
```

Exercise 8: Pipelining

Generated assembler output by the C compiler.

```
sum0:   .type  func          ; Set symbol type in the ELF symbol table
; main.c      107  {
; main.c      108    long i;
; main.c      109    long sum = 0;
  mov16  d2,#0               ; Initialize the sum saved in d2 to 0. d2 is the return register
; main.c      110    for (i=0;i<N;i++)
  mov16  d15,d2              ; Intialize the loop counter i to 0. d15 form the upper context is used
  j L_2                      ; Jump to L_2
L_3:
; main.c      111    {
; main.c      112    sum += X[i];
  ld16.w d0,[a4+]            ; a4 is the 1st pointer argument. Load X[i] to register d0 and increment
                             ; the pointer to the X array
  add16  d2,d0               ; Add X[i] to the sum
  add16  d15,#1              ; Add 1 to the loop counter i
L_2:
  jge  d4,d15,L_3            ; d4 is the 1st data argument. Jump to L_3 if N >= i
; main.c      113    }
; main.c      114    return sum;
; main.c      115  }
  ret16                       ; return sum in d2
```

The generated code does not take use of the super-scalar pipeline architecture of the TriCore™ and the usage of a jump great equal **`jge`** is time consuming because it takes two cycles to execute.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
Integer Pipeline	-	**`add`**	**`add`**	**`jge`**	**`jge`**
Load/Store Pipeline	**`ld`**	-	-	-	-
Loop Pipeline	-	-	-	-	-

The code can be improved by using the loop instruction which is executed in the loop pipeline.

Exercise 8: Pipelining

Assembler Implementation

```
.define     Xptr    'a4'    ; 1st address arg
.define     N       'd4'    ; 1st data arg
.define     sum     'd2'    ; return value
.define     x       'd12'   ; upper context register
.define     LC      'a15'   ; register used for loop counter

sum1:
  mov    sum, #0            ; Initialize the sum saved to 0.
  mov.a  LC,N               ; Intialize the loop counter LC to N
sum1loop:
  ld.w   x,[Xptr+]          ; Load X[i] to x and increment the pointer to the X array
  add    sum,x              ; Add x to the sum
  loop   LC,sum1loop        ; Jump to sum1loop if the loop counter is greater
                            ; zero, decrement the counter
  ret                       ; return sum in d2
```

	Cycle 1	Cycle 2
Integer Pipeline	-	`add`
Load/Store Pipeline	`ld`	-
Loop Pipeline	-	`loop`

Comparing the C and the ASM implementation, introducting the loop instruction replaces two instructions. The zero over-head loop saves 3 CPU cycles, because it is executed in parallel. But still the pipelines are not optimal filled. The add instruction has to wait for the load instruction.

Exercise 8: Pipelining

Optimized Assembler Implementation

An optimized implementation decouples the add and load instruction in the loop.

```
.define     Xptr    'a4'    ; 1st address arg
.define     N       'd4'    ; 1st data arg
.define     sum     'd2'    ; return value
.define     x       'd12'   ; upper context register
.define     LC      'a15'   ; register used for loop counter

sum2:
  mov    sum,#0              ; Initialize the sum saved in d2 to 0
  mov.a  LC,N                ; Intialize the loop counter LC to 0
  ld.w   x,[Xptr+]4;         ; Load X[0] to x and increment the pointer to the X array
sum2loop:
  add    sum,x               ; Add x to the sum
  ld.w   x,[Xptr+]4          ; Load X[i] to x and increment the pointer to the X array
  loop   LC,sum2loop         ; Jump to sum2loop if the loop counter is greater
                             ; zero, decrement the counter
  ret                       ; return sum in d2
```

	Cycle 1
Integer Pipeline	**add**
Load/Store Pipeline	**ld**
Loop Pipeline	**loop**

The optimized ASM solution completely fills the three pipelines and doubles the execution speed.

Exercise 8: Pipelining

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\pipe`. Copy the `sat.dav` file from the previous exercise to the new directory and rename the file to `pipe.dav`.

2. Generate the application framework

Start *DAvE* and open the `pipe.dav` project file. Click on the **Generate Code** icon  on the application toolbar to start the code generation process. Save and close the *DAvE* project.

3. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\pipe\pipe.pjt`.

4. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

5. Set current project

Use the context menu in the workspace window to make the pipe project the current project.

6. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter `c:\infineon\tc1796_intmem.opt`.

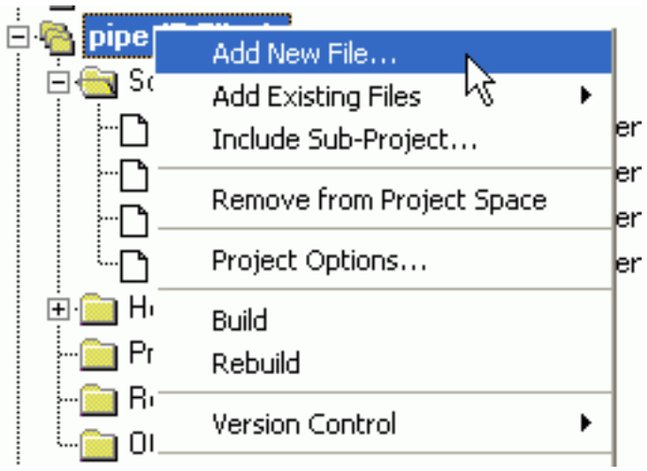
7. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

Exercise 8: Pipelining

8. Add the user code

Right click on the project and choose **Add New file**. Add a new file named `pipe.asm` to the project.



Add the following code at the beginning of the `pipe.asm` file.

```
    .global  sum1
    .global  sum2
    .sdecl   ".text.pipe",CODE
    .sect    ".text.pipe"
```

and implement the different solution described on page 10-2, 10-5 and 10-6.

Exercise 8: Pipelining

Measure the execution time like it is done in the saturation exercise using the `STM_uwReadSysTmr` function.

```
// USER CODE BEGIN (Main,9)
unsigned long dt;
long X[100];

for(int i=0; i<100; i++)                    // fill the array
    X[i]=1;

__isync();                                  // this clears the pipeline
dt = STM_uwReadSysTmr(STM_TIMER_0);
s = sum0(X,99);
dt = STM_uwReadSysTmr(STM_TIMER_0) - dt;
printf("sum solution 1: %.1f cycles %ld\n", 2.*dt, s);
...
```

9. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar to run the application.

As expected the execution time of solution 3 is twice as fast as solution 2. The C Implementation is much slower. It uses 3 more cycles than solution 2 because of an additional add and an jump greater equal instruction which in this case is a 2 cycle instruction.
Due to the tripple issue architecture a 150MHz device can execute up to 450MIPS and taking the PCP into account the TC1796 has more than 500MIPS.

Exercise 8: Pipelining

10. Single-handed exercise (20min): Implement and optimize a squarediff algorithm.

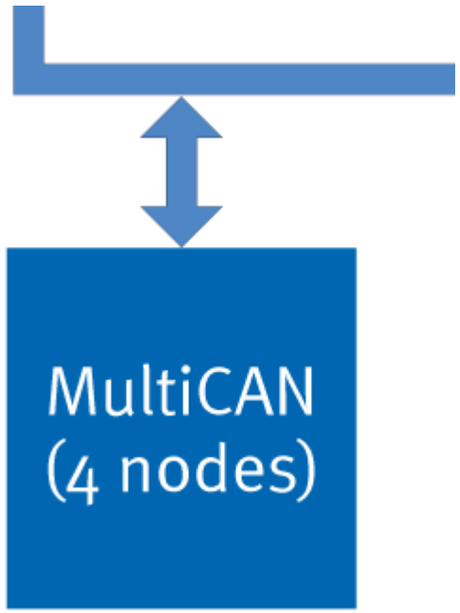
$$sum = \sum_{i=1}^{N} (X_i - Y_i)^2$$

Hint: Start with a C implementation. Analyze the assembler output and then write an optimized ASM implementation.

Exercise 9: CAN

CAN

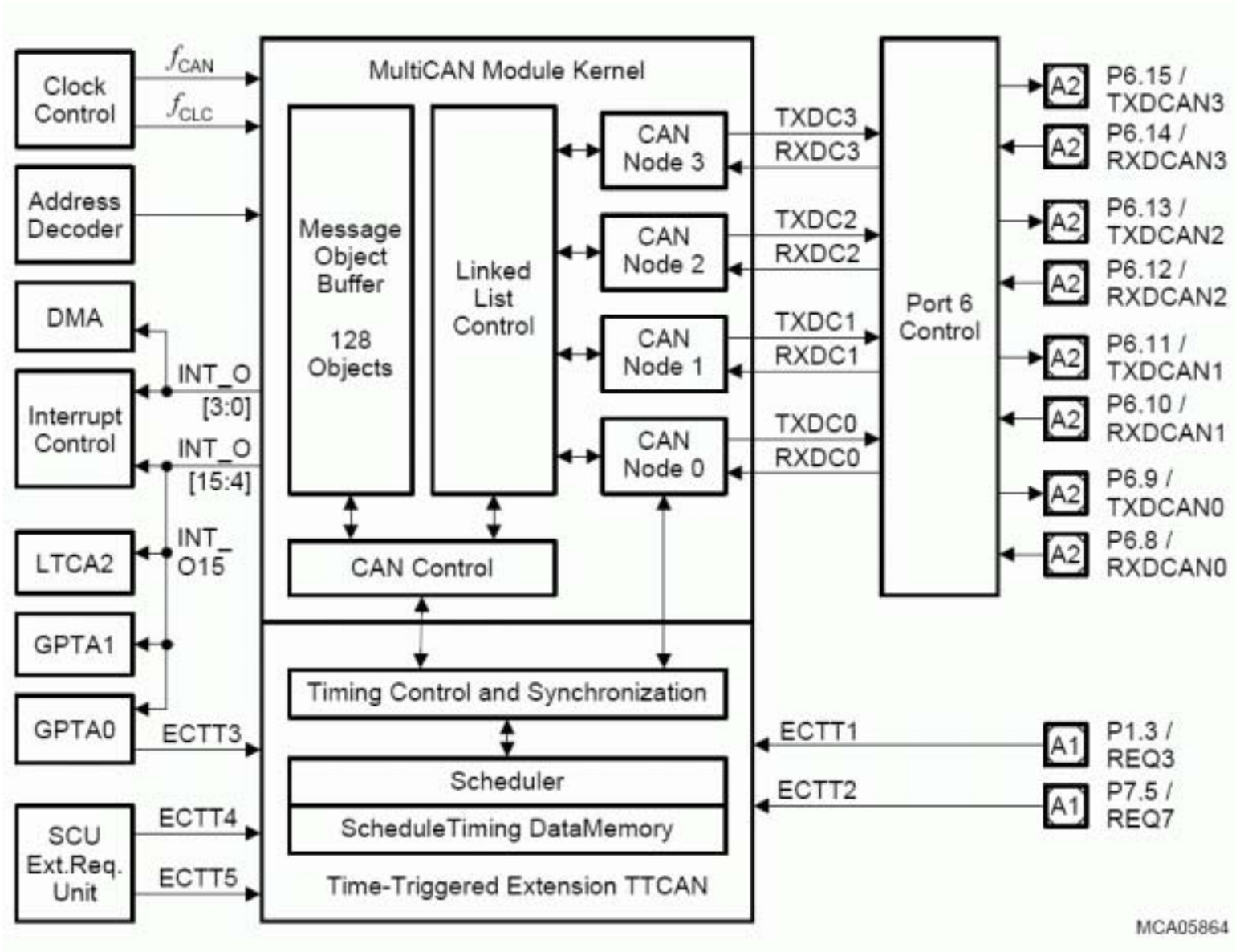
This exercise develops an application that sends messages between two CAN nodes. The TC1796 MultiCAN module has 4 independent CAN nodes to connect the microcontroller to different, separated CAN networks which might operate at different baurate and might run different higher level protocols.
This exercise uses the internal CAN bus so that no cabling is necessary to connect the two nodes, but the exercise can easily modified to send messages between two TriBoards.



Exercise 9: CAN

CAN Features

- CAN V2.0 B conform (compliant to ISO 11898)
- 4 CAN nodes
- 128 message objects
- Dedicated control registers for each CAN node
- Data transfer rate up to 1MBit/s
- Flexible and powerful message transfer control and error handling capabilities
- Message objects can be individually
  - Assigned to one of the four CAN nodes
  - Configured as transmit or receive object
  - Configured as message buffer
  - Configured to handle 11-bit or 29-bit identifiers
  - Provided with programmable filter mask
  - Monitored via a frame counter
  - Configured for Remote Monitoring Mode
- Automatic Gateway Mode support
- 16 programmable interrupt nodes
- CAN bus analyzer mode
- Time-Triggered Extension (TTCAN)

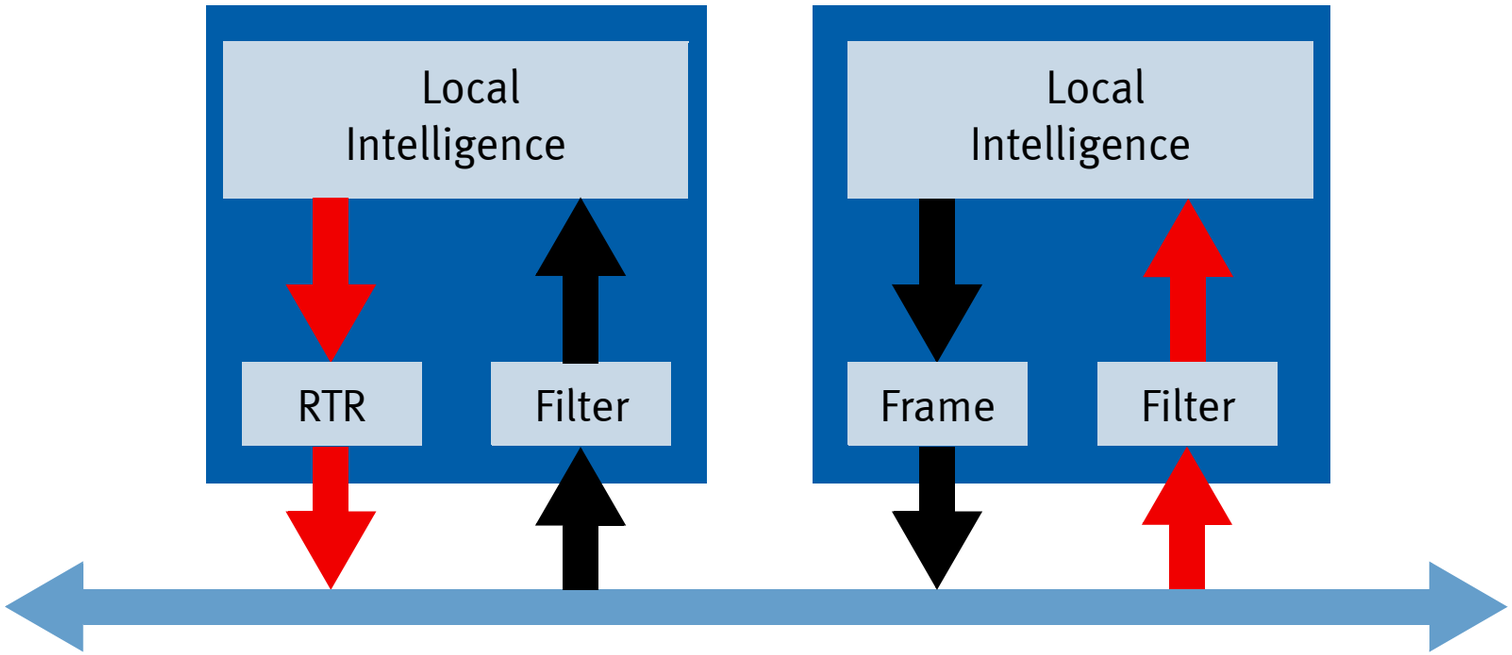


Exercise 9: CAN

Remote request

CAN Node 0
(Requester + Consumer)

CAN Node 1
(Producer)



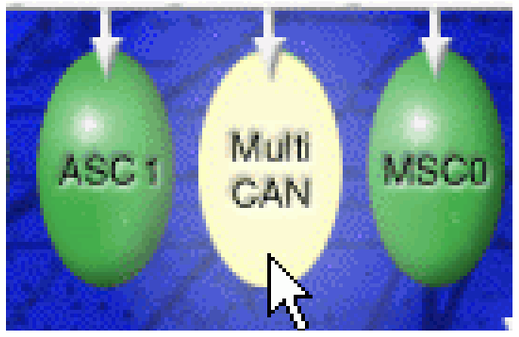
Exercise 9: CAN

1. Create a DAvE project

Open the *Windows Explorer* and create a new directory `c:\infineon\can`. Copy the `isr.dav` file from the previous exercise to the new directory and rename the file to `can.dav`.

2. Open the MultiCAN properties

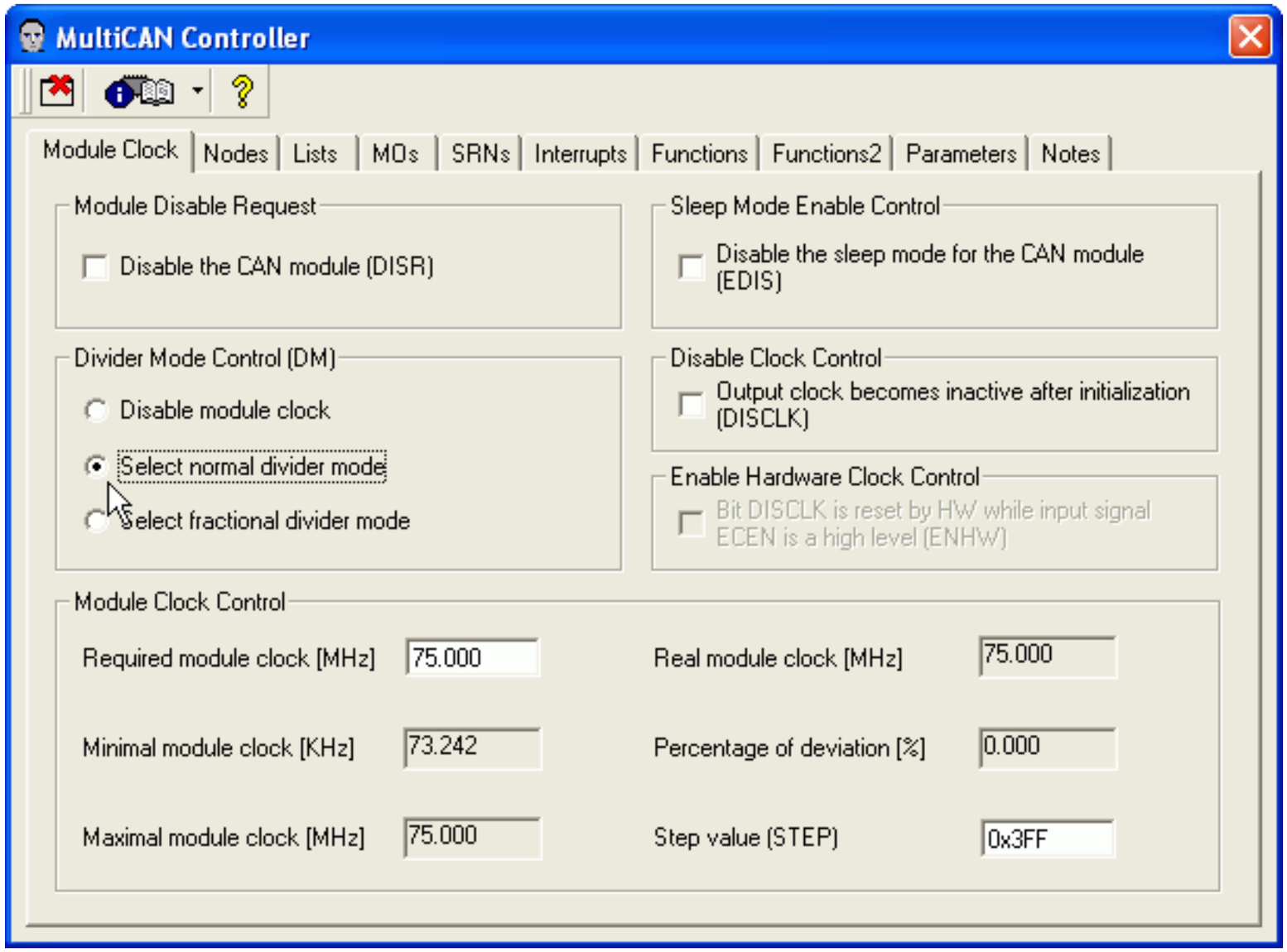
Click on **MultiCAN** in the project window to open the MultiCAN properties.



Exercise 9: CAN

3. Set up the MultiCAN properties: Module Clock

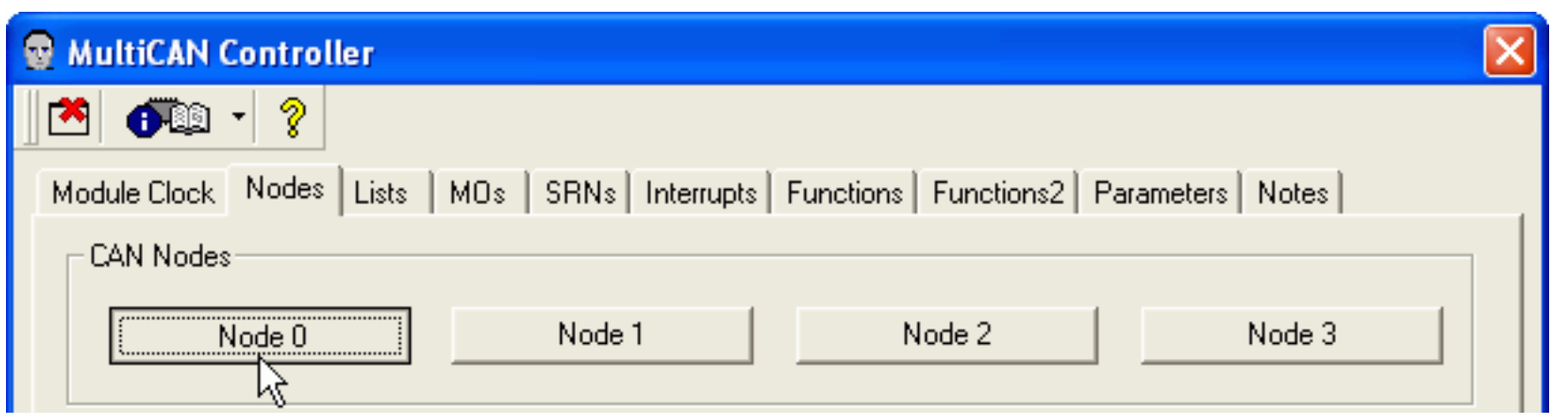
Enable the CAN module and select **Select normal divider mode**, so that the CAN module is running at 75MHz.



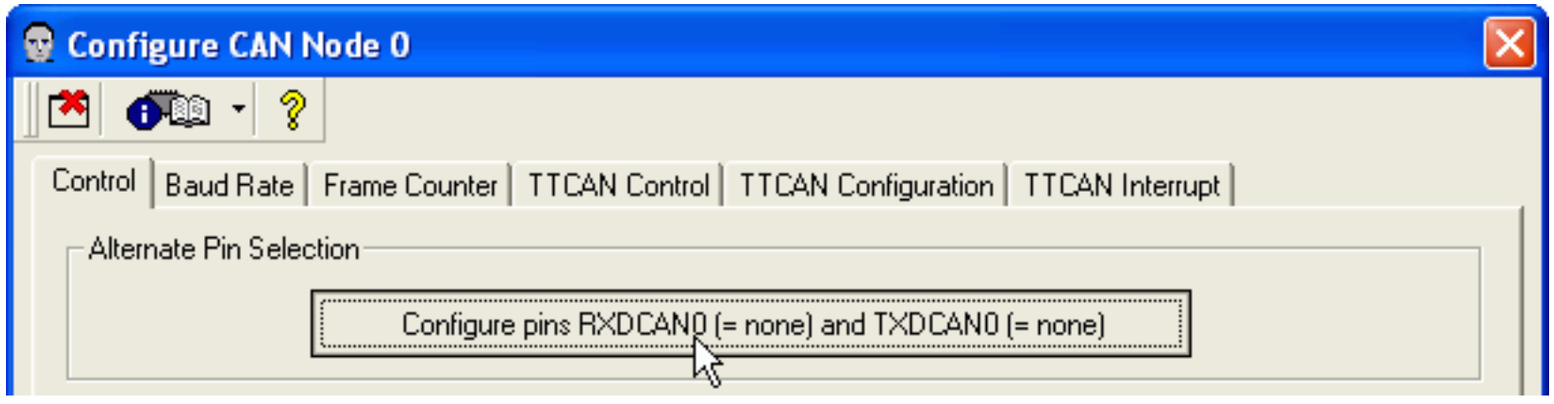
Exercise 9: CAN

4. Set up the MultiCAN properties: CAN node 0

In this exercise, CAN node 0 is used to transmit a remote frame to node 1 and receives a data frame from node 1.
Both nodes are connected to the internal CAN bus, as to avoid cabling (Loop Back Mode).
On the **Nodes** page click **Node 0** to configure the CAN Node 0.

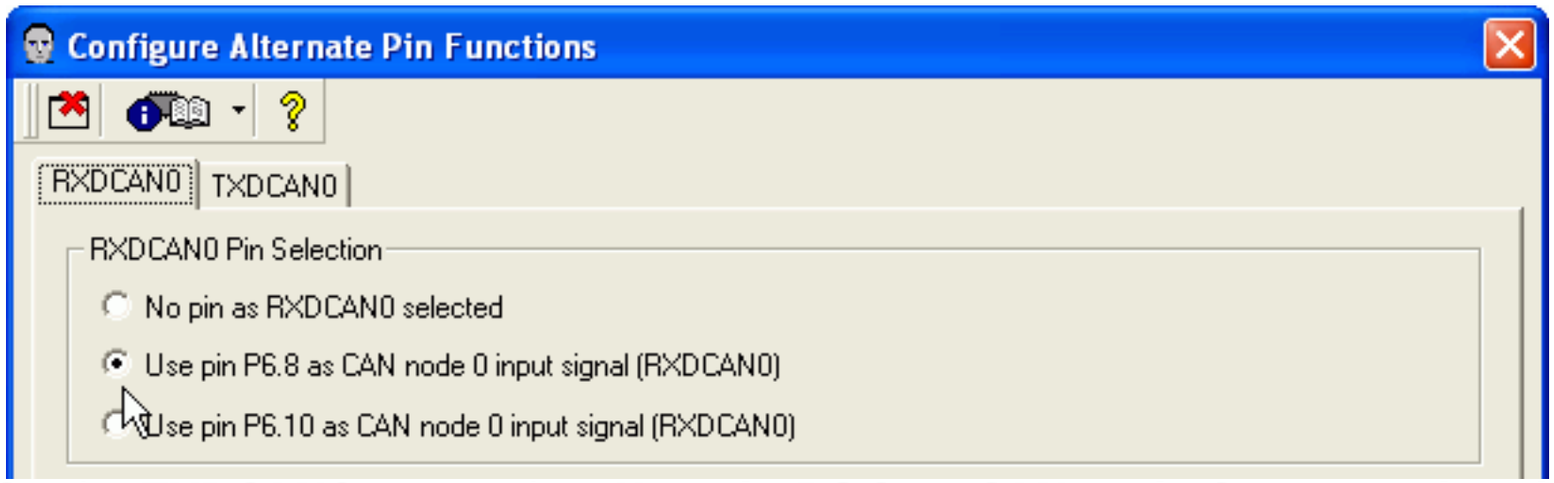


Although we are connecting the CAN nodes internally using the Loop-Back-Mode (LBM) input and output pins has to be configured.
Click Alternate Pin Selection **Configure pins**.

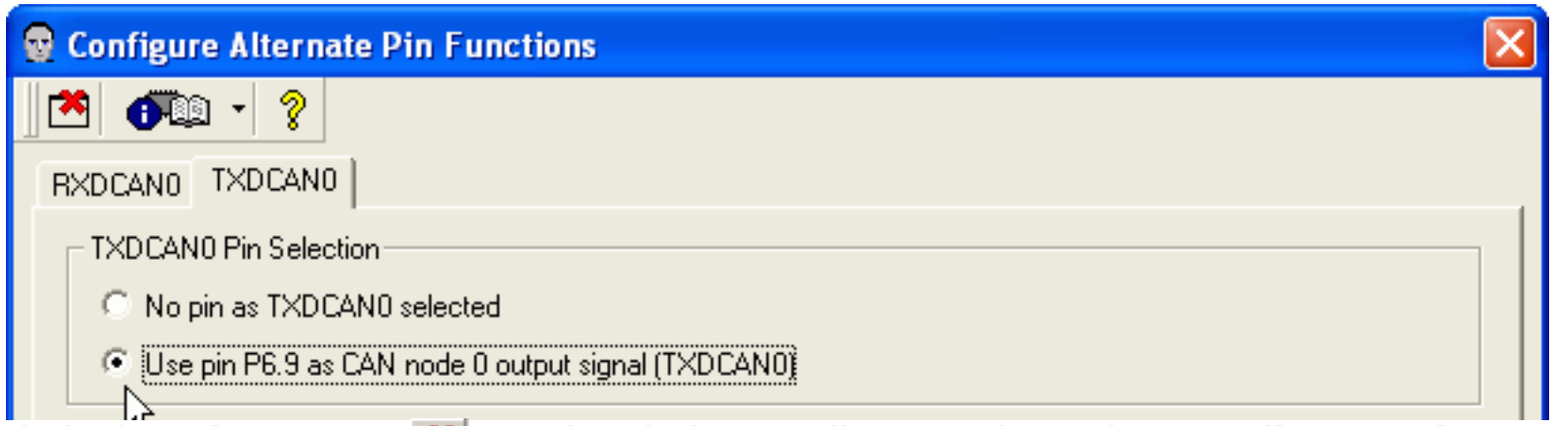


Exercise 9: CAN

On the **RXDCAN0** page select **Use pin P6.8 as CAN node 0 input signal (RXDCAN0)**.



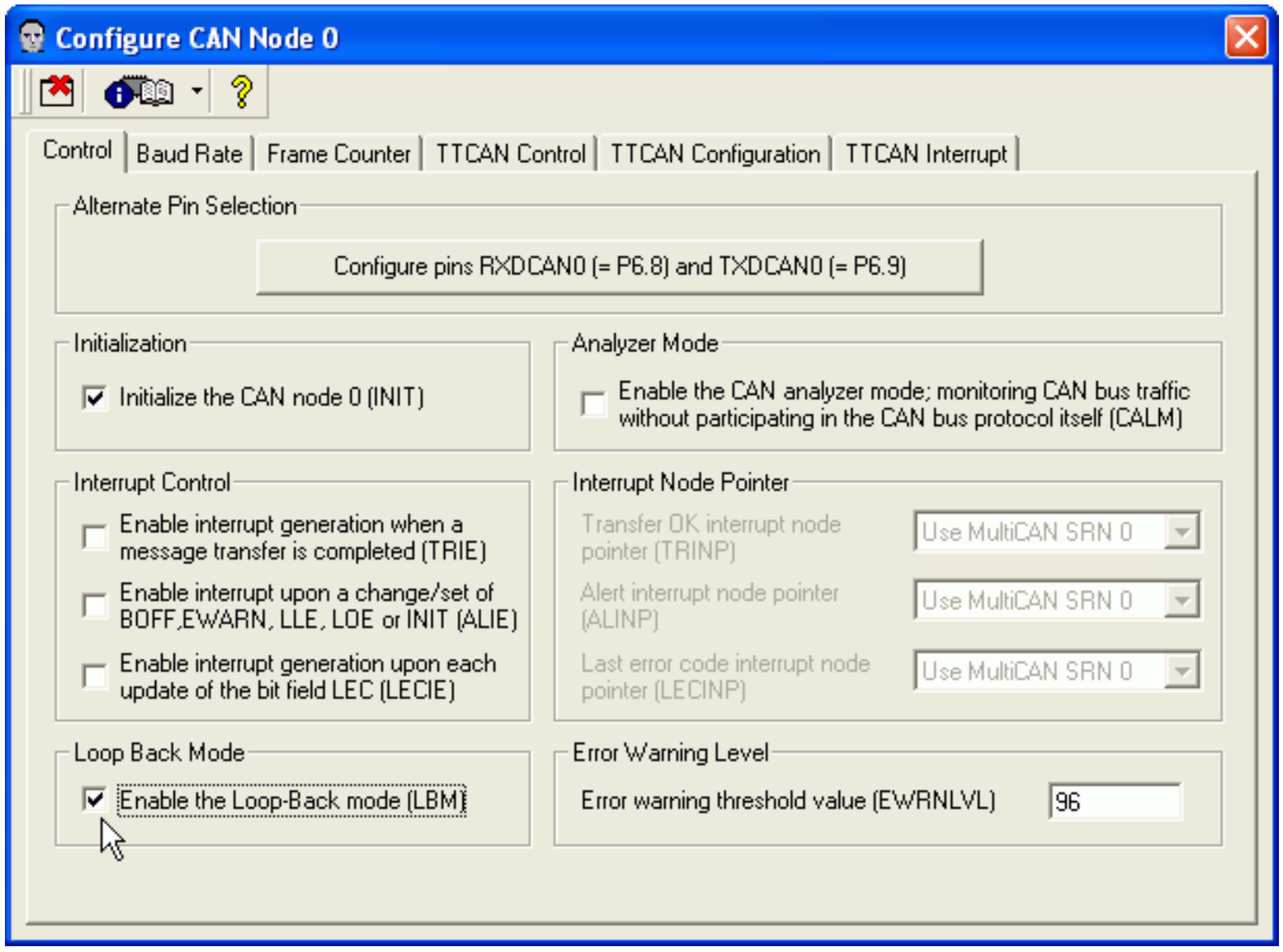
On the **TXDCAN0** page select **Use pin P6.9 as CAN node 0 output signal (TXDCAN0)**.



Click the **Close** icon  on the dialog toolbar to close the **Configure Alternate Pin Functions** dialog.

Exercise 9: CAN

On the **Control** page
- Check **Initialize the CAN node 0 (INIT)**,
- Check **Enable the Loop-Back mode (LBM)** to connect the node to the internal CAN bus.

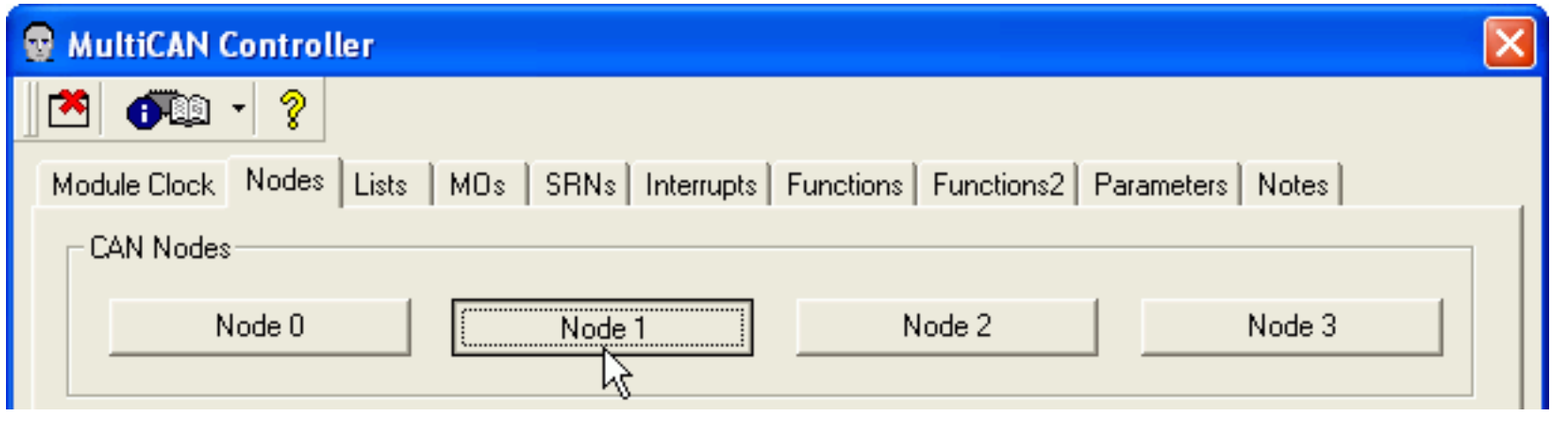


Node 0 is configured.
Click the **Close** icon on the dialog toolbar to close the **Configure CAN Node 0** dialog and return to the **MultiCAN** page.

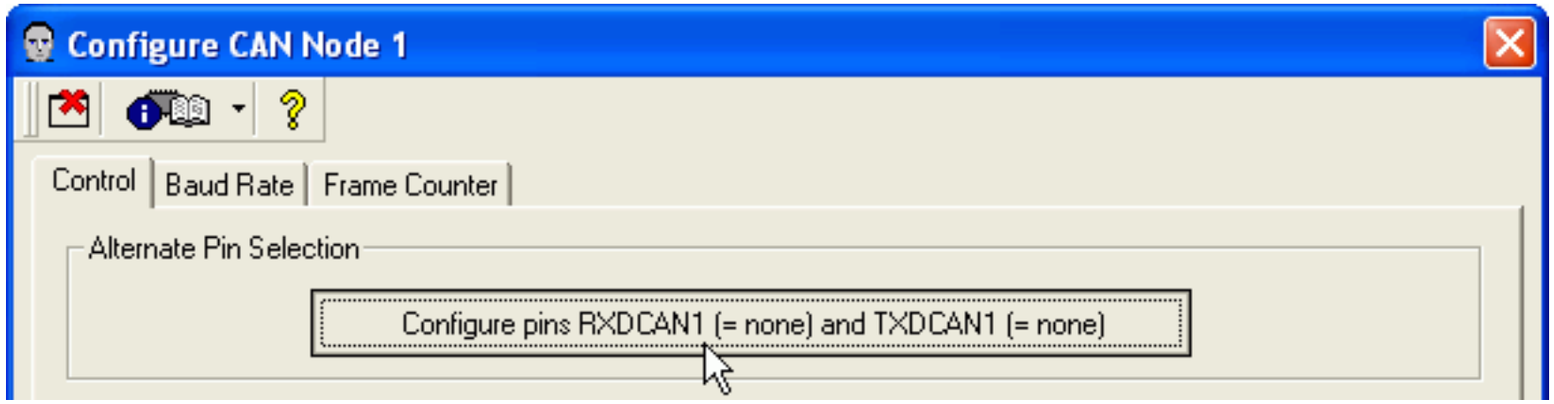
Exercise 9: CAN

5. Set up the MultiCAN properties: CAN node 1

CAN node 1 is used to receive remote frames and to transmit data frames.
On the **Nodes** page and click **Node 1**.

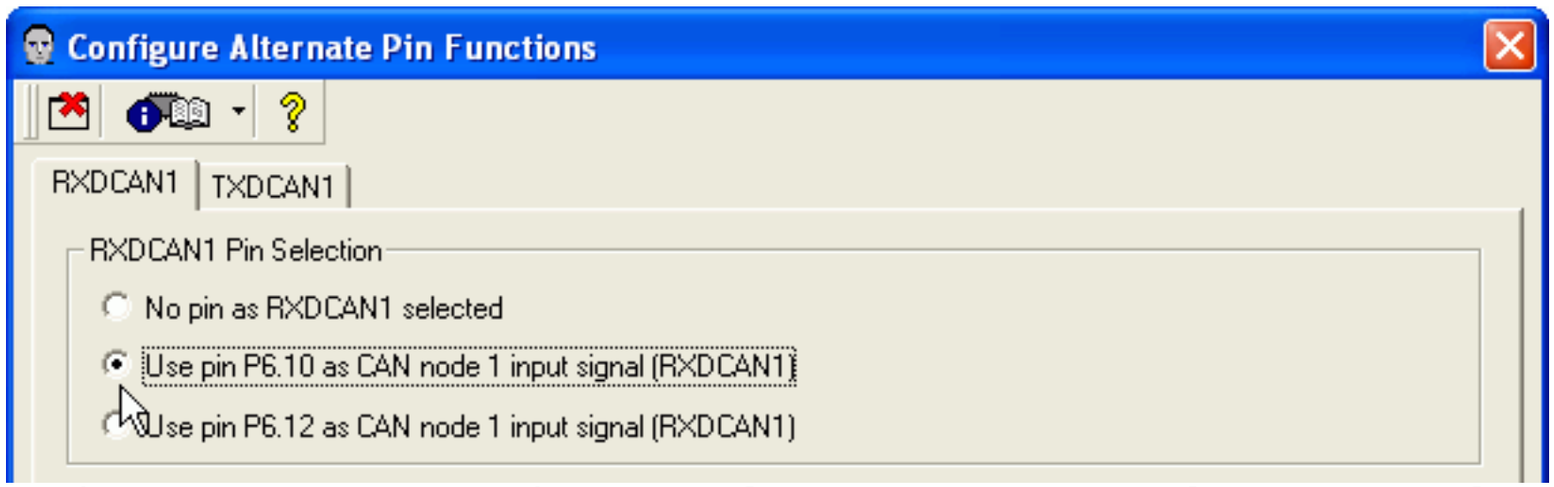


Click Alternate Pin Selection **Configure pins**.

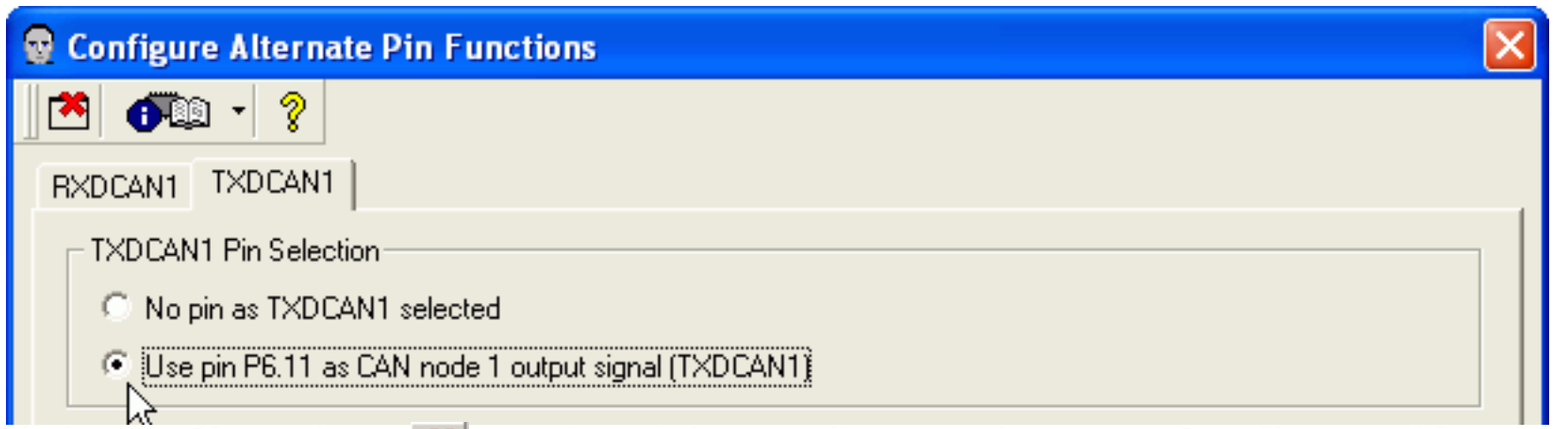


Exercise 9: CAN

On the **RXDCAN0** page select **Use pin P6.10 as CAN node 1 input signal (RXDCAN1)**.



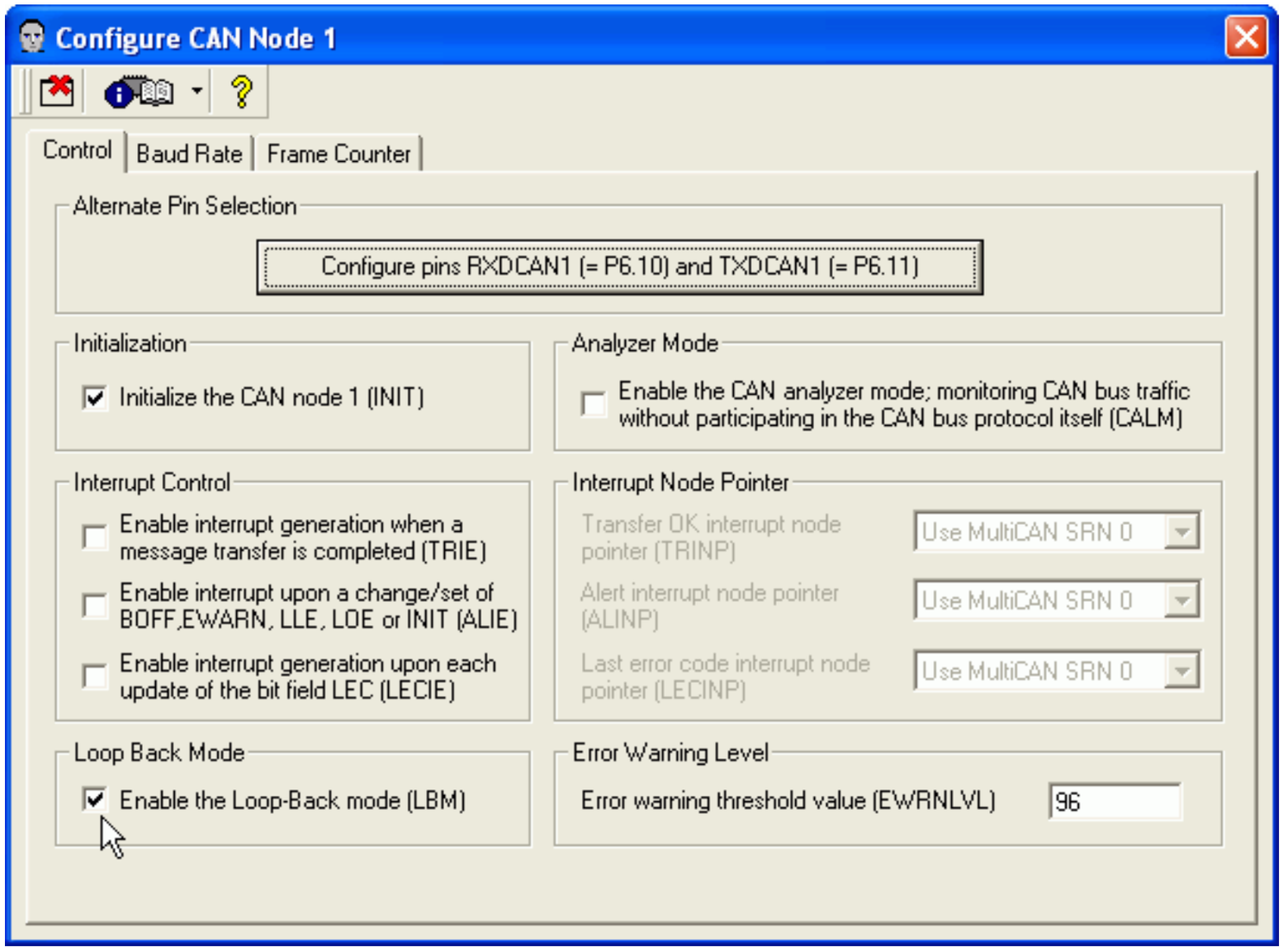
On the **RXDCAN0** page select **Use pin P6.11 as CAN node 1 output signal (TXCAN1)**.



Click the **Close** icon  on the dialog toolbar to close the **Configure Alternate Pin Functions** dialog.

Exercise 9: CAN

On the **Control** page
- Check **Initialize the CAN node 1 (INIT)**,
- Check **Enable the Loop-Back mode (LBM)** to connect the node to the internal CAN bus.

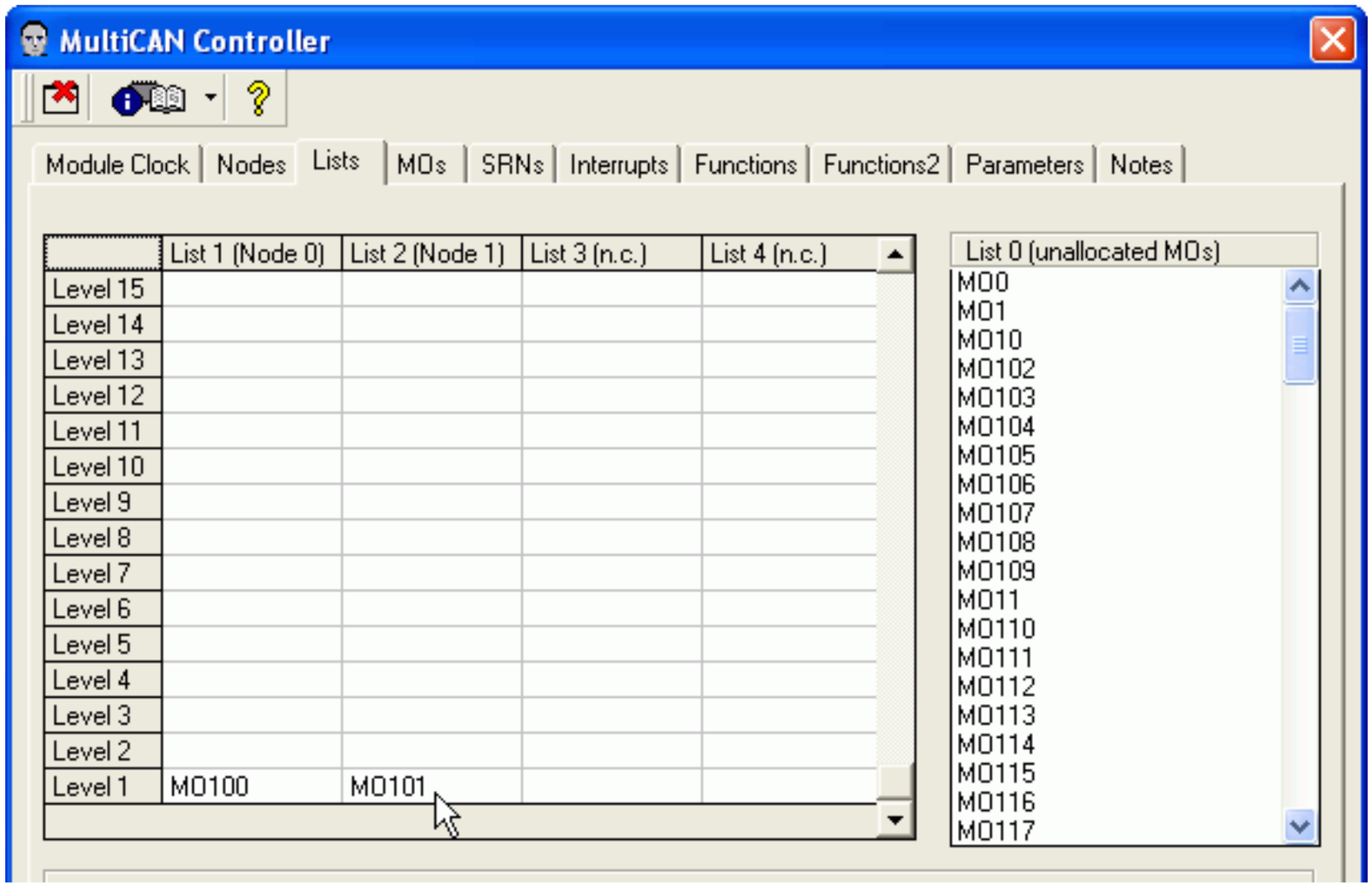


Node 1 is configured.
Click the **Close** icon on the dialog toolbar to close the **Configure CAN Node 1** dialog and return to the MultiCAN page.

6. Set up the MultiCAN properties: Message objects

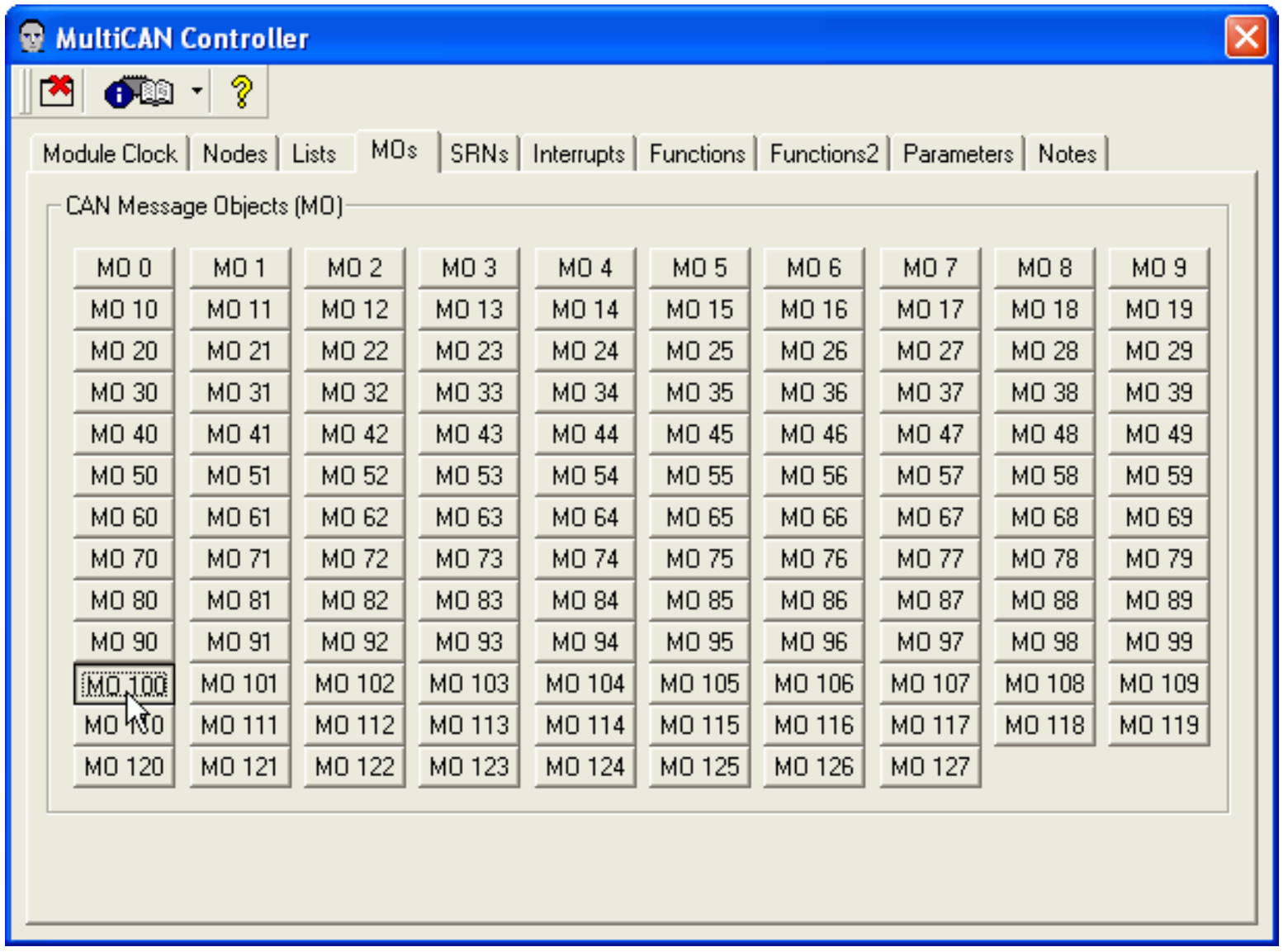
The next step is the configuration of two message objects. The MultiCAN module includes a total of 128 message objects. Each message object can be allocated to one (and only one) CAN node and handles messages with a certain ID. This means on the other hand, that each CAN node has a list of message objects (between 0 and 128). The order of each list may be used for prioritizing messages.
In this exercise, node 0 sends a message to node 1 and receives an answer from node 1.
On the **Lists** page

- Drag MO100 from **List 0 (unallocated MOs)** to **List 1 (Node 0)** level 1,
- Drag MO101 from **List 0 (unallocated MOs)** to **List 2 (Node 1)** level 1.



Exercise 9: CAN

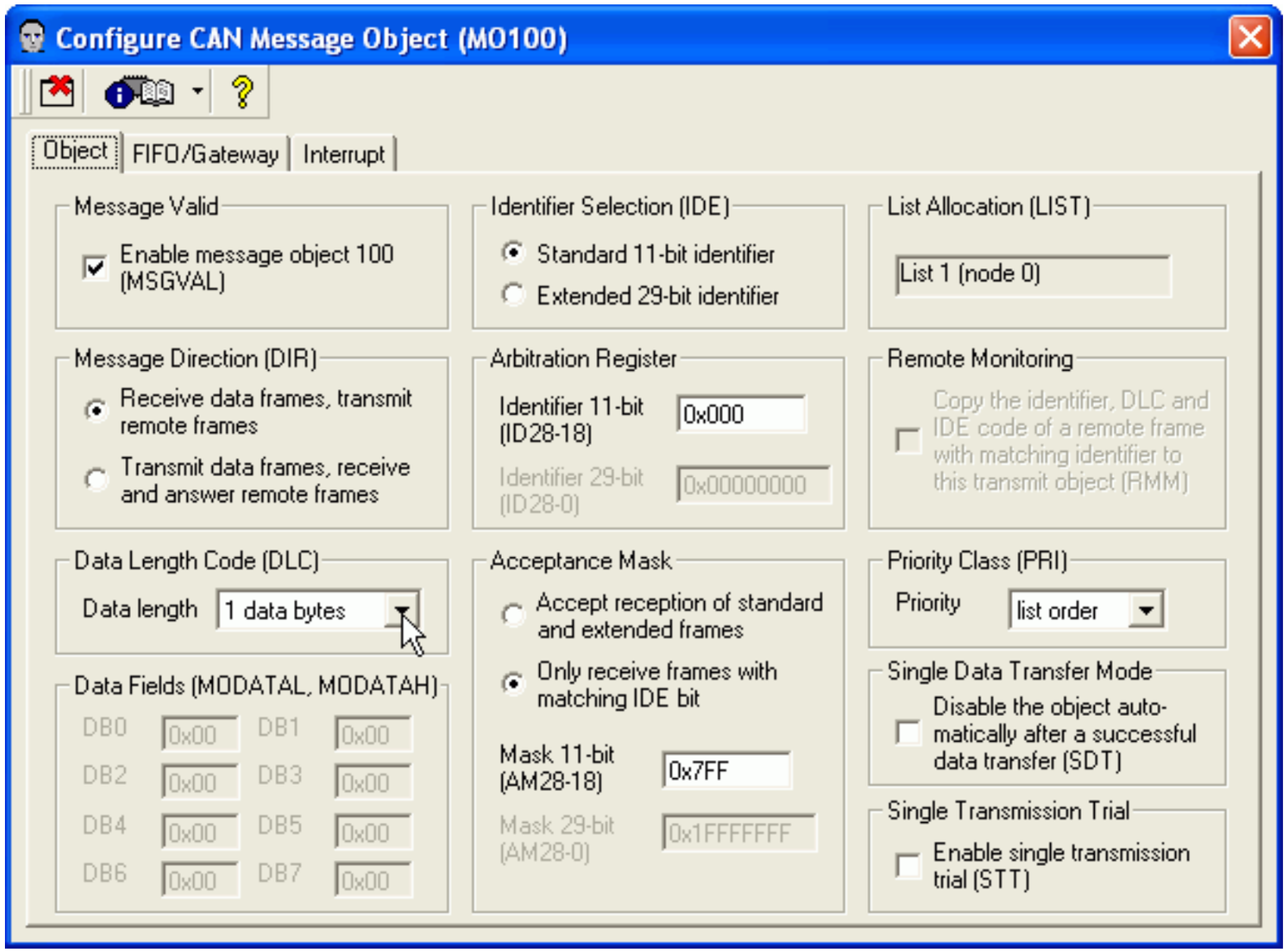
On the **MOs** page click **MO100** to configure the message object attached to node 0.



Exercise 9: CAN

On the **Object** page

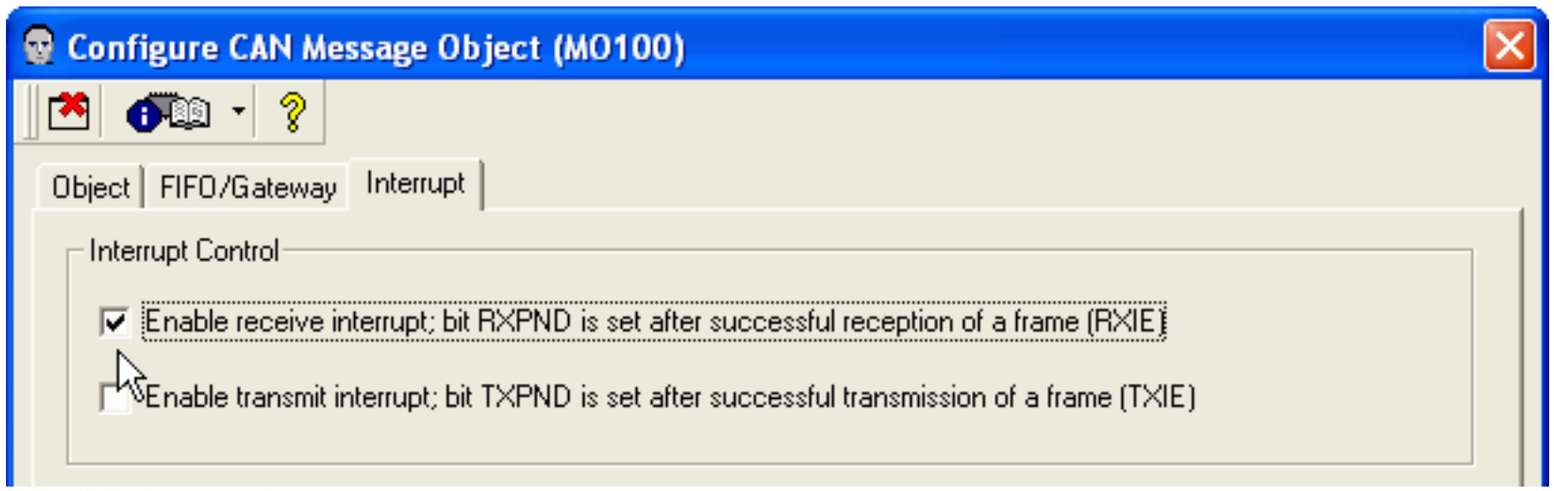
- Check **Enable message object 100 (MSGVAL)**,
- Select Message Direction (DIR) **Receive data frames, transmit remote frames**,
- Choose **1 data bytes** as Data length.



Exercise 9: CAN

On the **Interrupt** page

■ Check **Enable receive interrupt; bit RXPND is set after successful reception of a frame (RXIE)**.



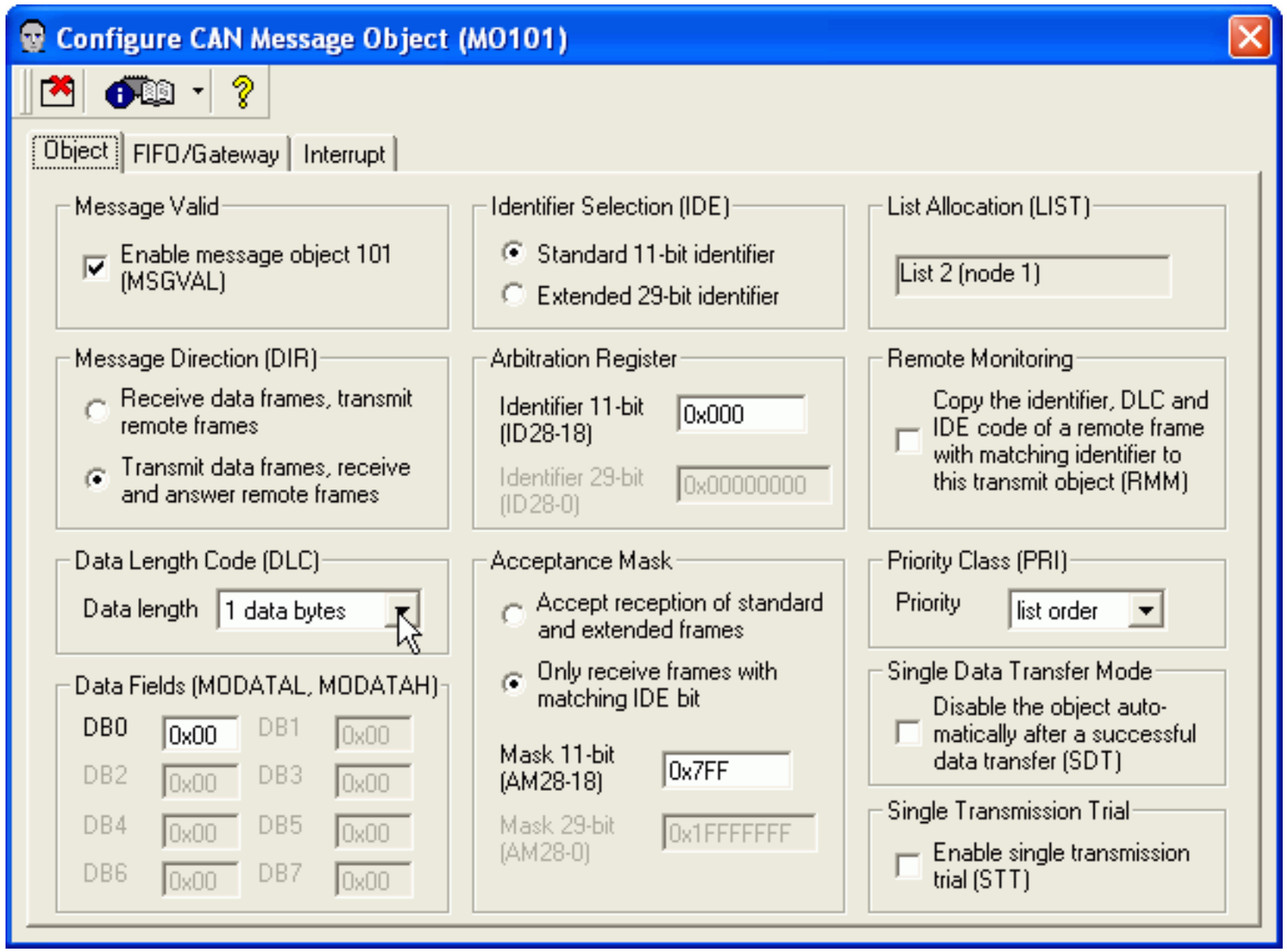
M0100 is configured.

Click the **Close** icon on the dialog toolbar to close the **Configure CAN Message Object (MO100)** dialog.

Exercise 9: CAN

On the **MOs** page click **M0101** to configure the message object attached to node 1.
On the **Object** page

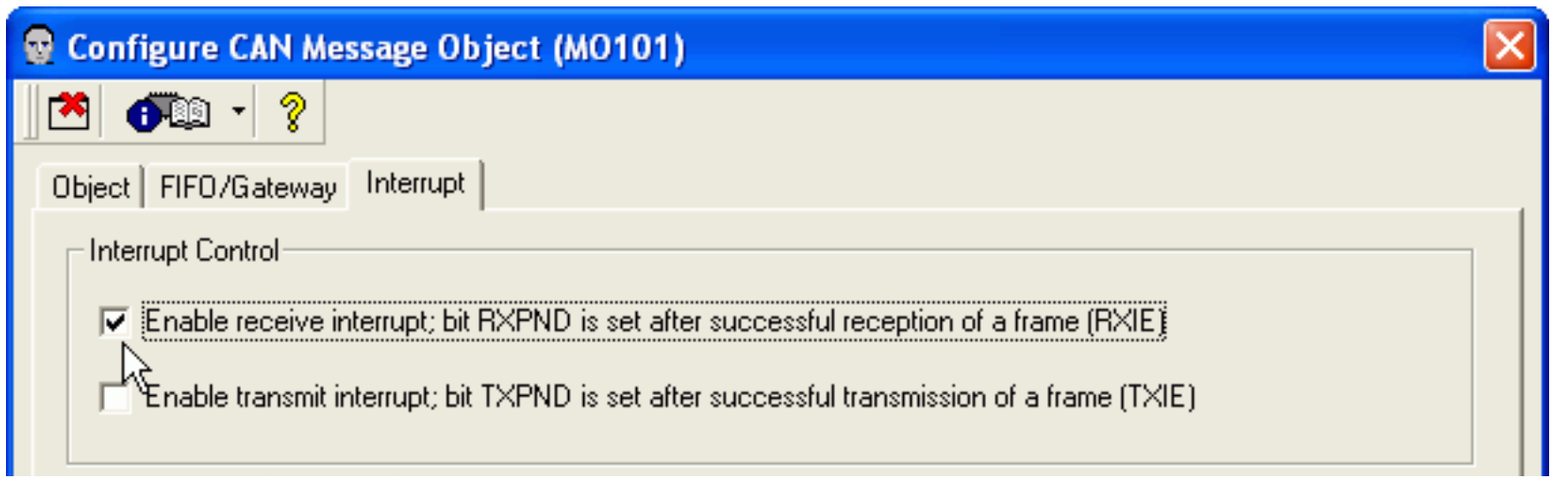
- Check **Enable message object 101 (MSGVAL)**,
- Select Message Direction (DIR) **Transmit data frames, receive and answer remote frames**,
- Choose **1 data bytes** as Data length.



Exercise 9: CAN

On the **Interrupt** page

■ Check **Enable receive interrupt; bit RXPND is set after successful reception of a frame (RXIE)**.



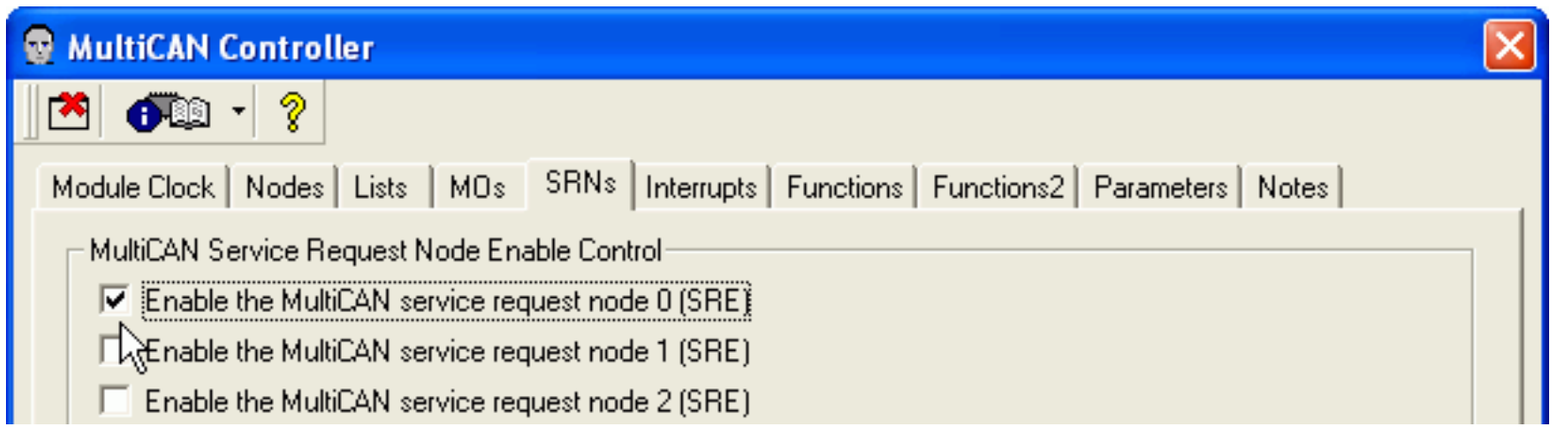
M0101 is configured.

Click the **Close** icon on the dialog toolbar to close the **Configure CAN Message Object (MO101)** dialog.

Exercise 9: CAN

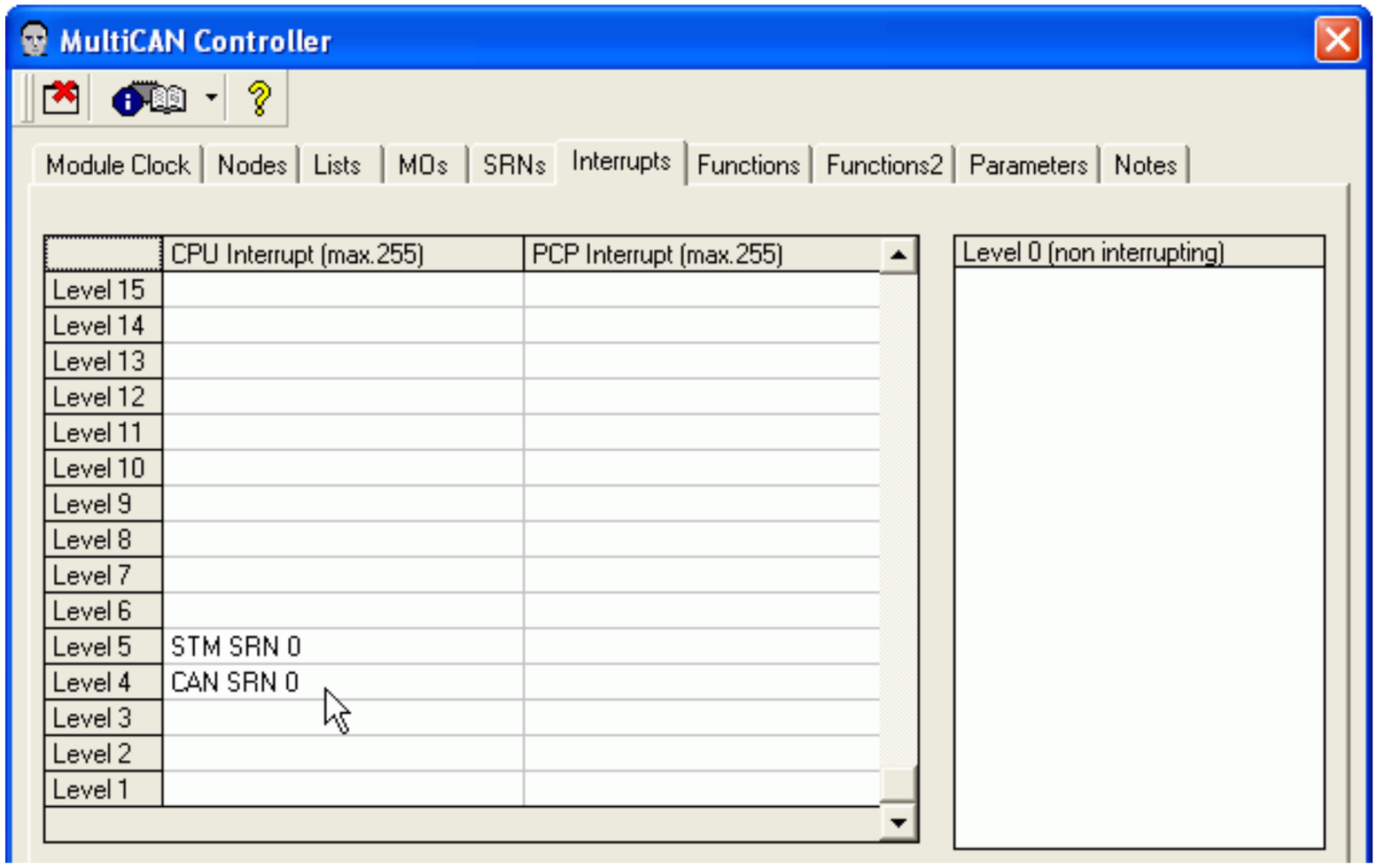
On the **SRN** page

■ Check **Enable the MultiCAN service request node 0 (SRE)**.



Exercise 9: CAN

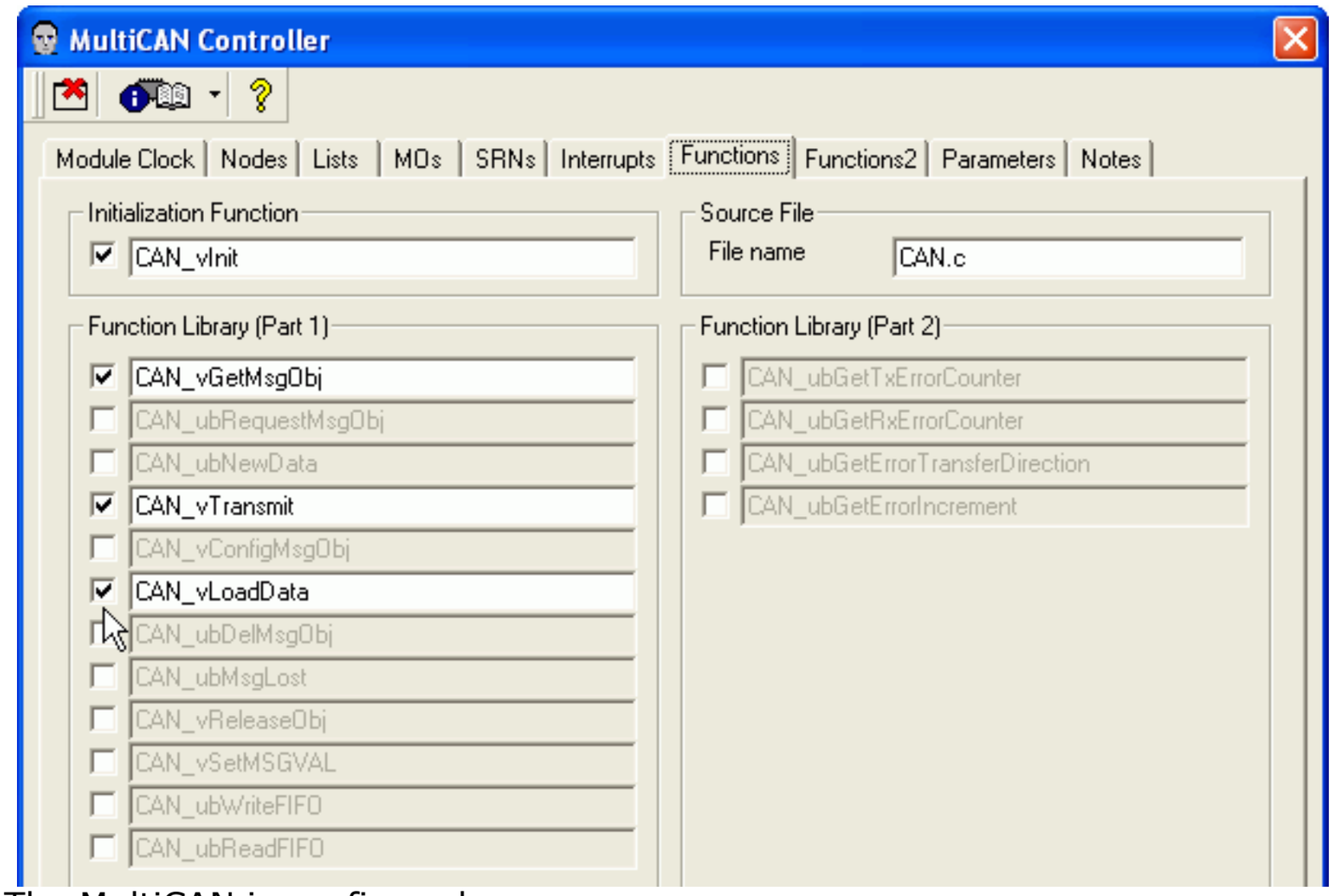
On the **Interrupts** page drag the **CAN SRN 0** from the Level 0 list to CPU Interrupt Level 4.



Exercise 9: CAN

On the **Functions** page

■ Check the `CAN_vInit`, `CAN_vGetMsgObj`, `CAN_vTransmit` and `CAN_vLoadData` functions.



The MultiCAN is configured.

Click the **Close** icon on the dialog toolbar to close the **MultiCAN Controller** dialog.

7. Generate the application framework

Click the **Generate code** icon on the application toolbar to start the code generation process. Save and close the *DAvE* project.

8. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\can\can.pjt`.

9. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

10. Set current project

Use the context menu in the workspace window to make the can project the current project.

11. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter `tc1796_extmem.opt`.

12. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

13. Add the user code

Add an endless loop which transmits message object 100 every 2 seconds to file `MAIN.c` at (Main,9).

```
// USER CODE BEGIN (Main,9)
for(;;){
  printf("Node 0: transmit remote frame\n");
  CAN_vTransmit(100);              // Transmit message object 100 of node 0
  delay(2000);
}
// USER CODE END
```

Exercise 9: CAN

In file `CAN.c`, add the following code to the interrupt routine `CAN_viSRN0` at (SRN0_OBJ100,2), which is called upon successful reception of a CAN data frame on node 0.

```
// USER CODE BEGIN (SRN0_OBJ100,2)
CAN_SWObj msg;
CAN_vGetMsgObj(100, &msg);
printf("Node 0: receive data frame, data = %d\n\n", (int)msg.ubData[0]);
// USER CODE END
```

In the file `CAN.c`, add the following code to the interrupt routine `CAN_viSRN0` at (SRN0_OBJ101,4), which is called upon successful reception of a CAN remote frame on node 1.

```
// USER CODE BEGIN (SRN0_OBJ101,4)
CAN_SWObj msg;
CAN_vGetMsgObj(101, &msg);
printf("Node 1: receive remote frame, transmit data frame, data = %d\n",
       (int)msg.ubData[0]);
// USER CODE END
```

In the file `STM.c`, add the following code to the interrupt routine `STM_viSRN0` at (SRN0,3). The STM interrupt is called every second and modifies the data value of message object 101.

```
// USER CODE BEGIN (SRN0,3)
static ubyte c = 5;
CAN_vLoadData(101, &c);  // Load new data to message object 101
c++;
// USER CODE END
```

14. Build the application

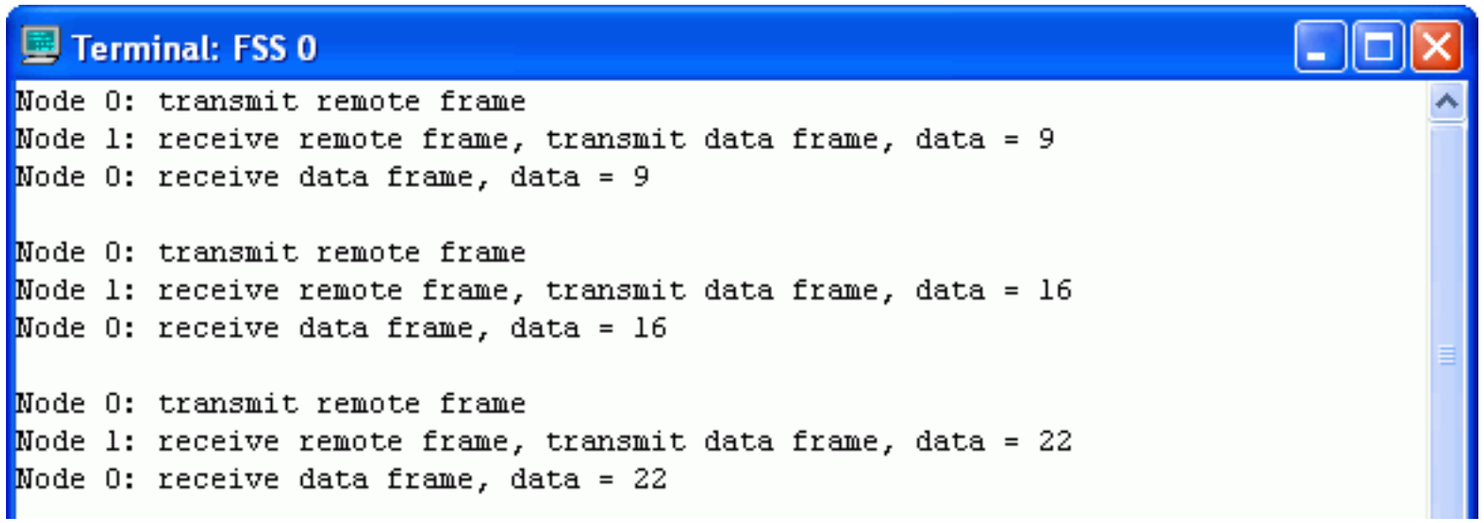
Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

15. Debug the application

Click the **Debug** icon  on the Build toolbar to open the *CrossView Pro* debugger.

16. Run the application

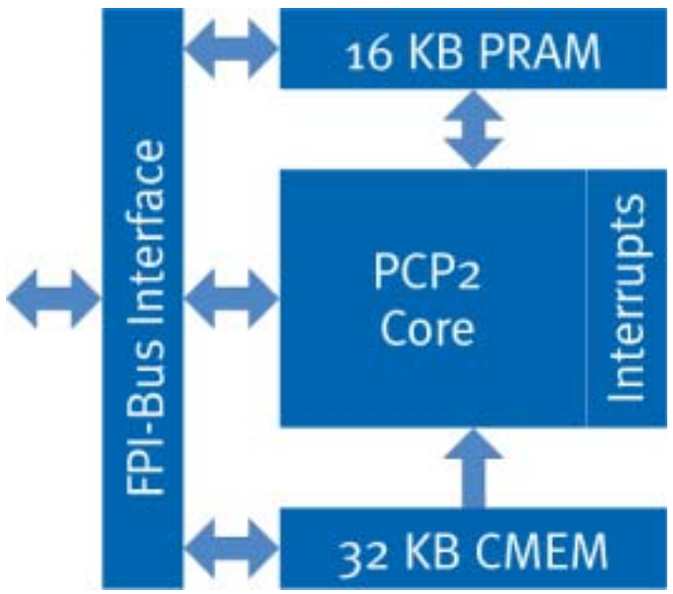
Click the **Run** icon  on the *CrossView Pro* toolbar and see terminal output.



Exercise 10: Peripheral Control Processor

Peripheral Control Processor

The PCP2 is a freely programmable, single cycle, 32-bit RISC processing unit with its own code and data memory unit used as an interrupt service providerd.



PCP Benefits

- PCP provides programmable improved peripheral intelligence instead of static implementation
- Off-loads TriCore™ from interrupt tasks and allows a parallel execution
- PCP fits best for real-time critical tasks due to very fast interrupt response.
- Handles fast, interrupt-driven routines for peripheral control Intelligent data pre-conditioning e.g. building up mean value of several concurrently sampled ADC values
- DMA data transportation like queued memory or FIFO structures e.g. coming from serial communication interfaces like CAN, SPI or ASC off-loads TriCore™ and realizes partially peripheral functionality in SW
- Complex state machines can easily be code

Exercise 10: Peripheral Control Processor

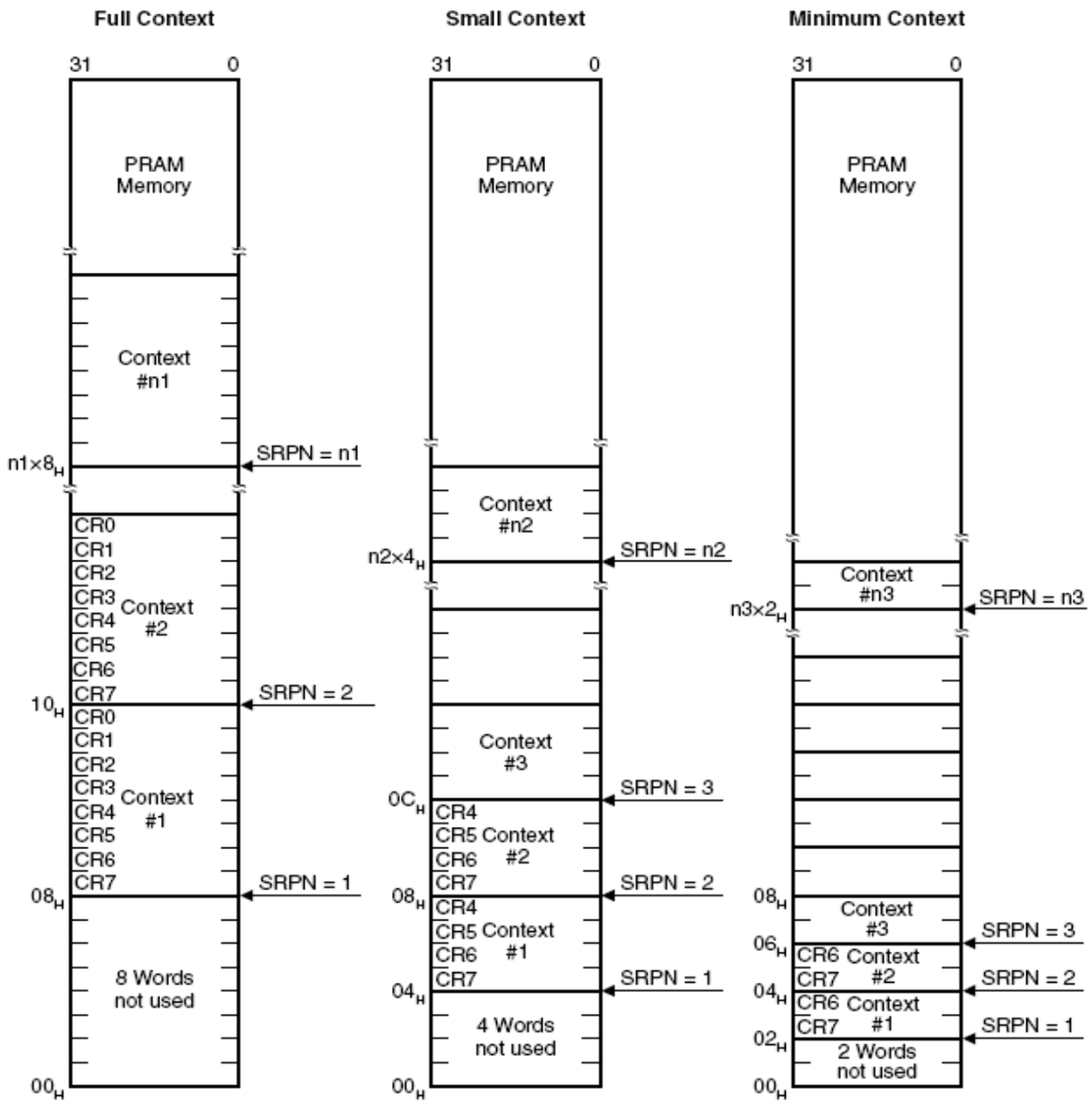


Fig.1 Context Storage in PRAM

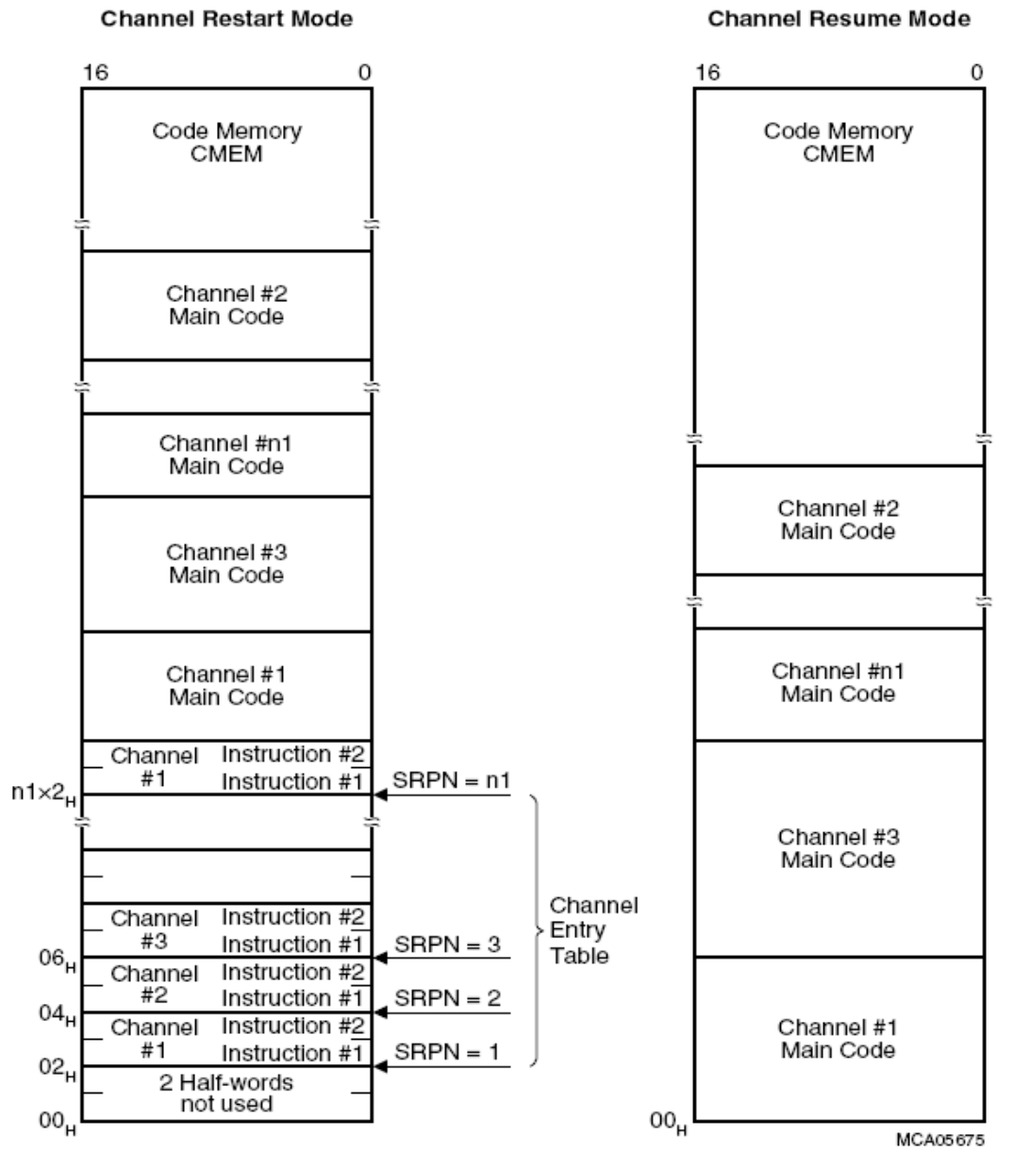
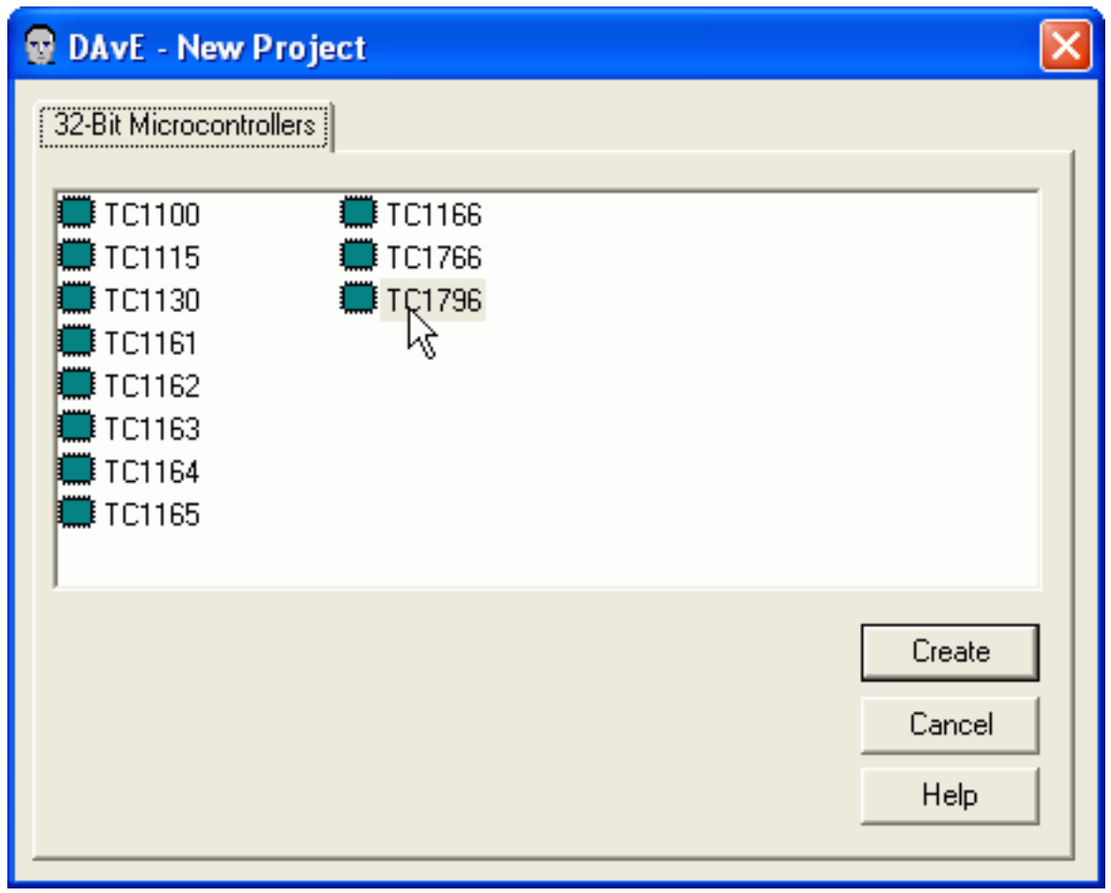


Fig.2 CMEM organization in rest resume mode (right)

Exercise 10: Peripheral Control Processor

1. Create a DAvE project

Launch *DAvE* and choose TC1796 from the list of 32-bit microcontrollers. Click **Create**.



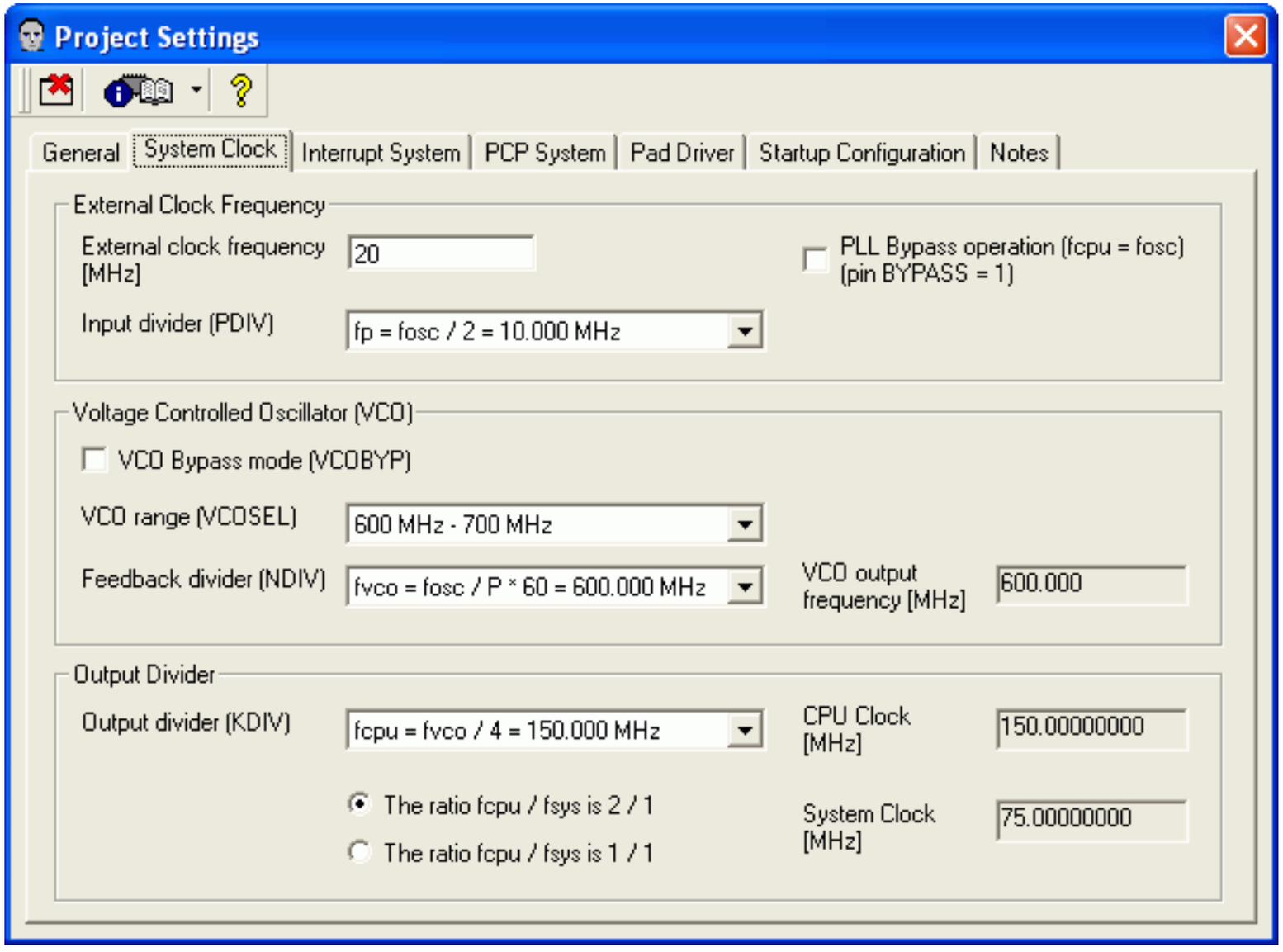
Two new windows, the **TC1796** project window and the **Project Settings** dialog appears.

2. Set up the Project Settings

In the **Project Setting**s dialog select the Compiler settings *Tasking* 2.0. If the dialog is not open choose **File > Project Settings**.

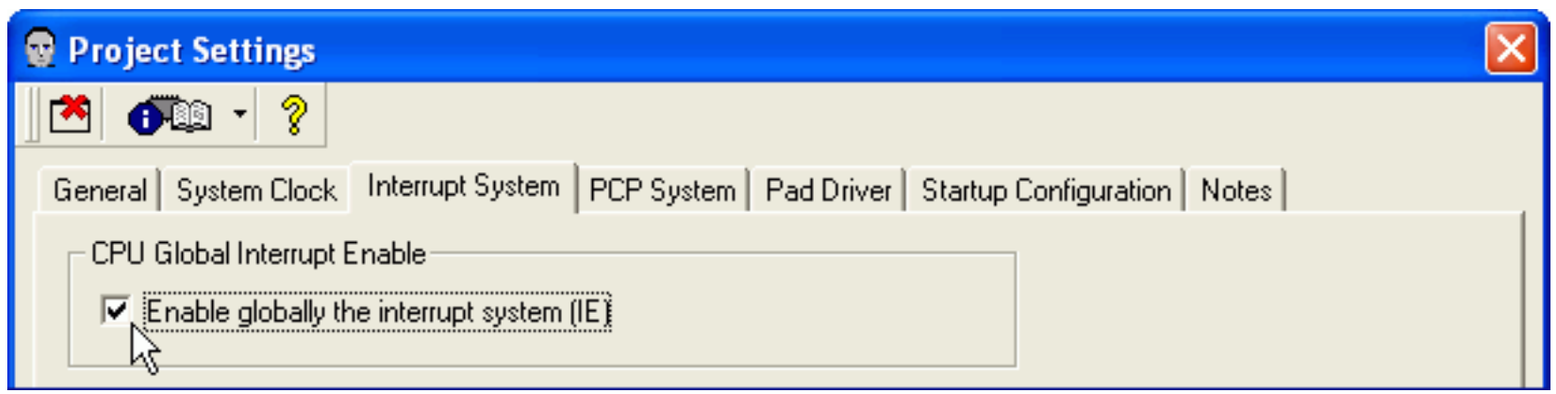
Exercise 10: Peripheral Control Processor

On the **System Clock** page set the external clock frequency to 20 MHz, PDIV = 2, NDIV = 60, KDIV = 4.



Exercise 10: Peripheral Control Processor

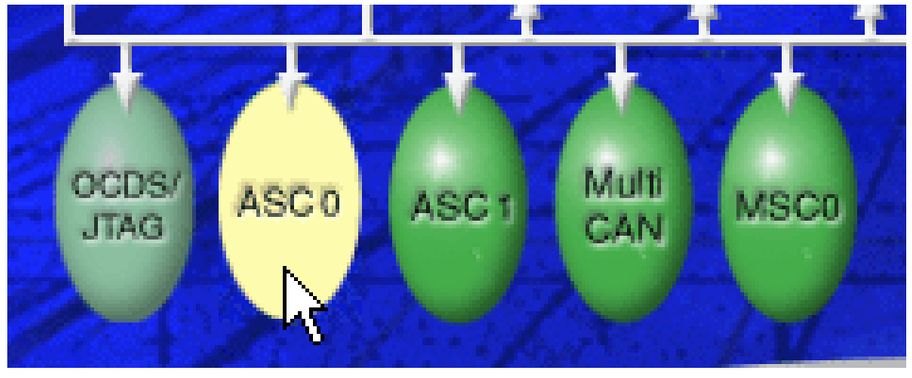
On the **Interrupt System** page check **Enable globally the Interrupt System (IE)**.



Click the **Close** icon on the dialog toolbar to close the dialog.

3. Open the ASC0 properties

Click on **ASC0** in the project window to open the ASC0 properties.

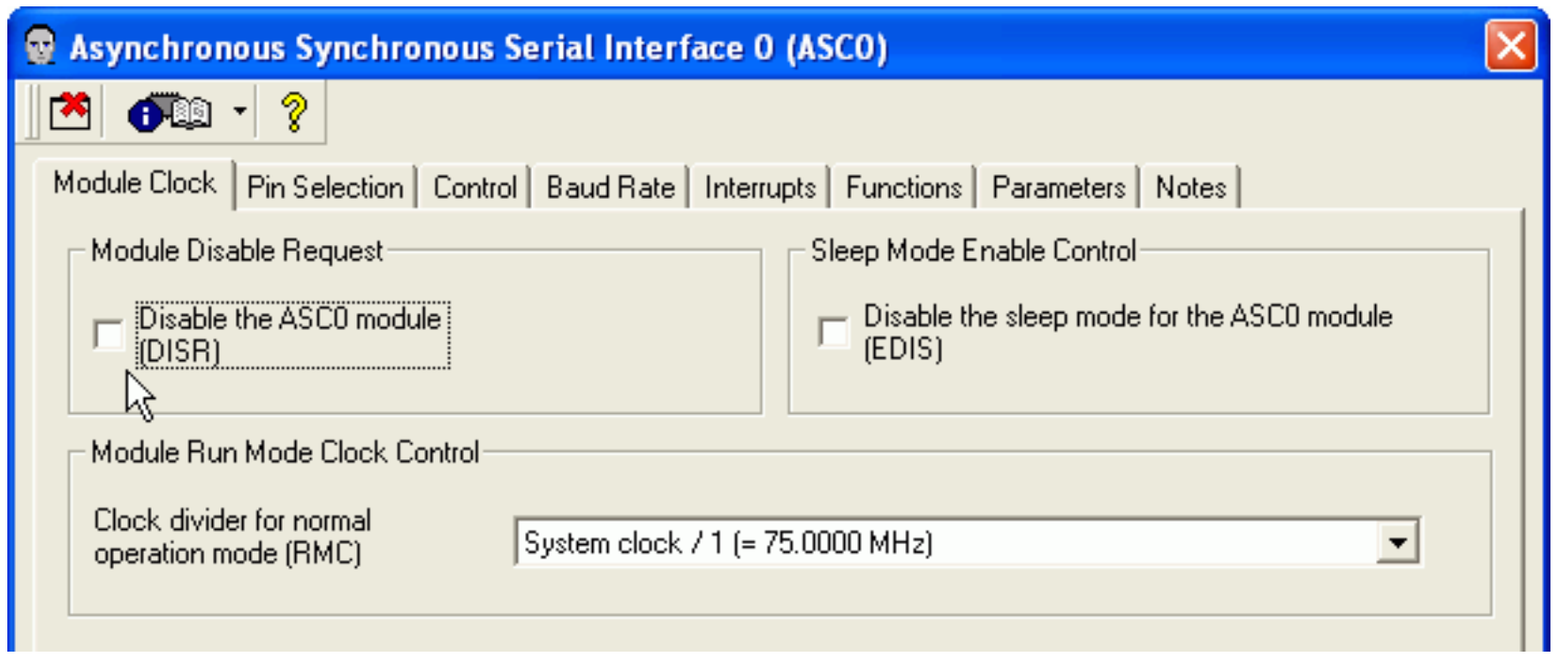


Exercise 10: Peripheral Control Processor

4. Set up the ASC0 properties

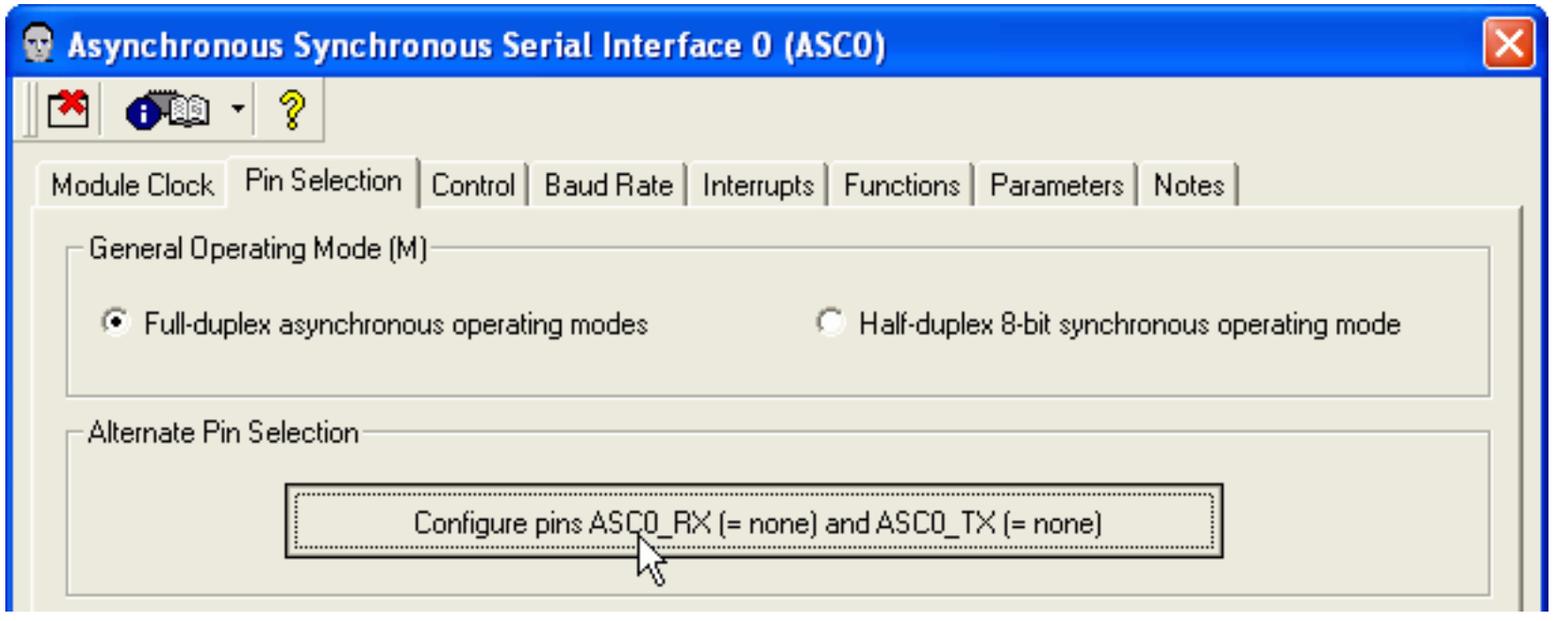
On the **Module Clock** page

- Uncheck **Disable the ASC0 module (DISR)**,
- Choose Module Run Mode Clock Control **System clock / 1**.

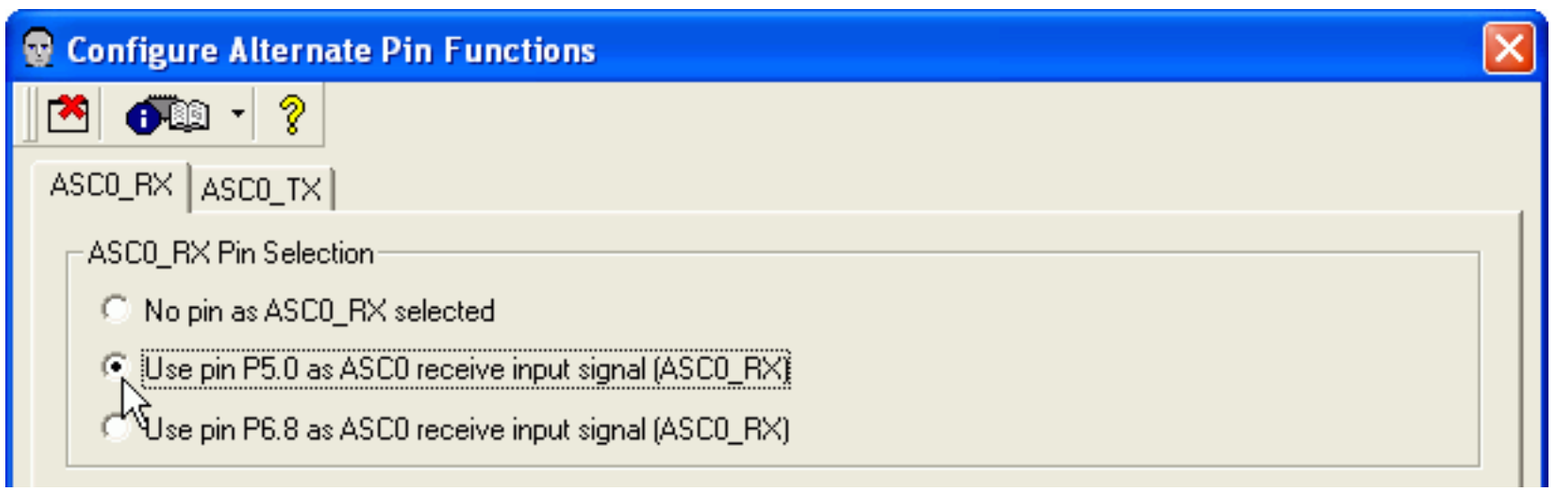


Exercise 10: Peripheral Control Processor

On the **Pin Selection** page click **Configure pins**.

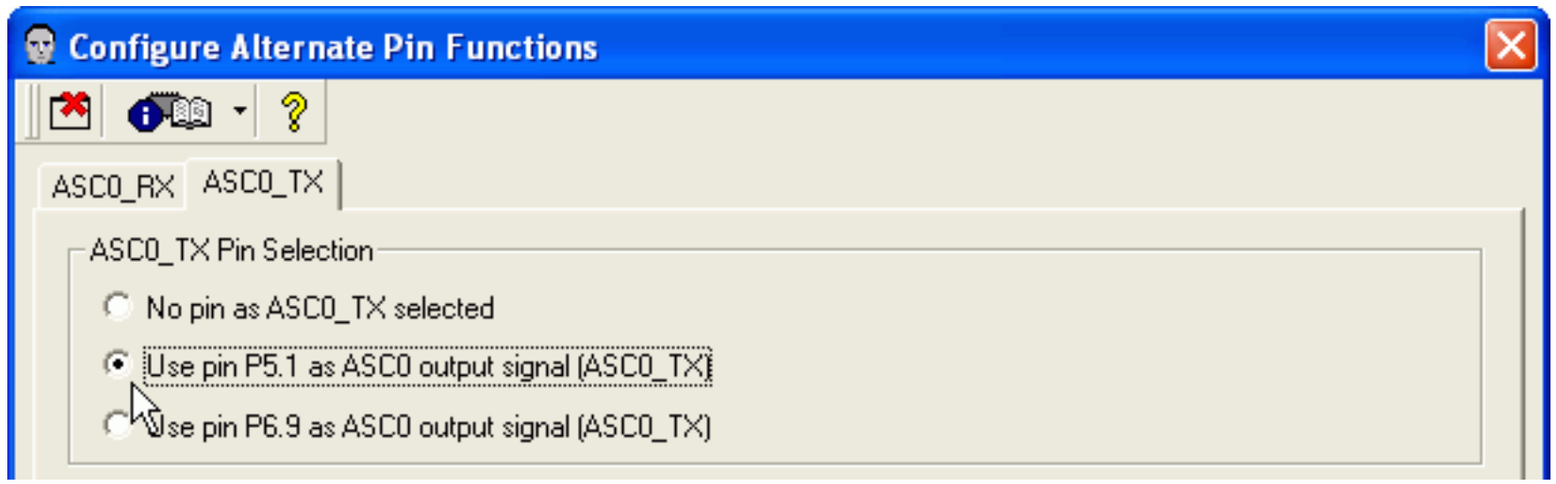


On the **ASC0_RX** page select **Use pin P5.0 as ASC0 receive input signal (ASC0_RX)**.



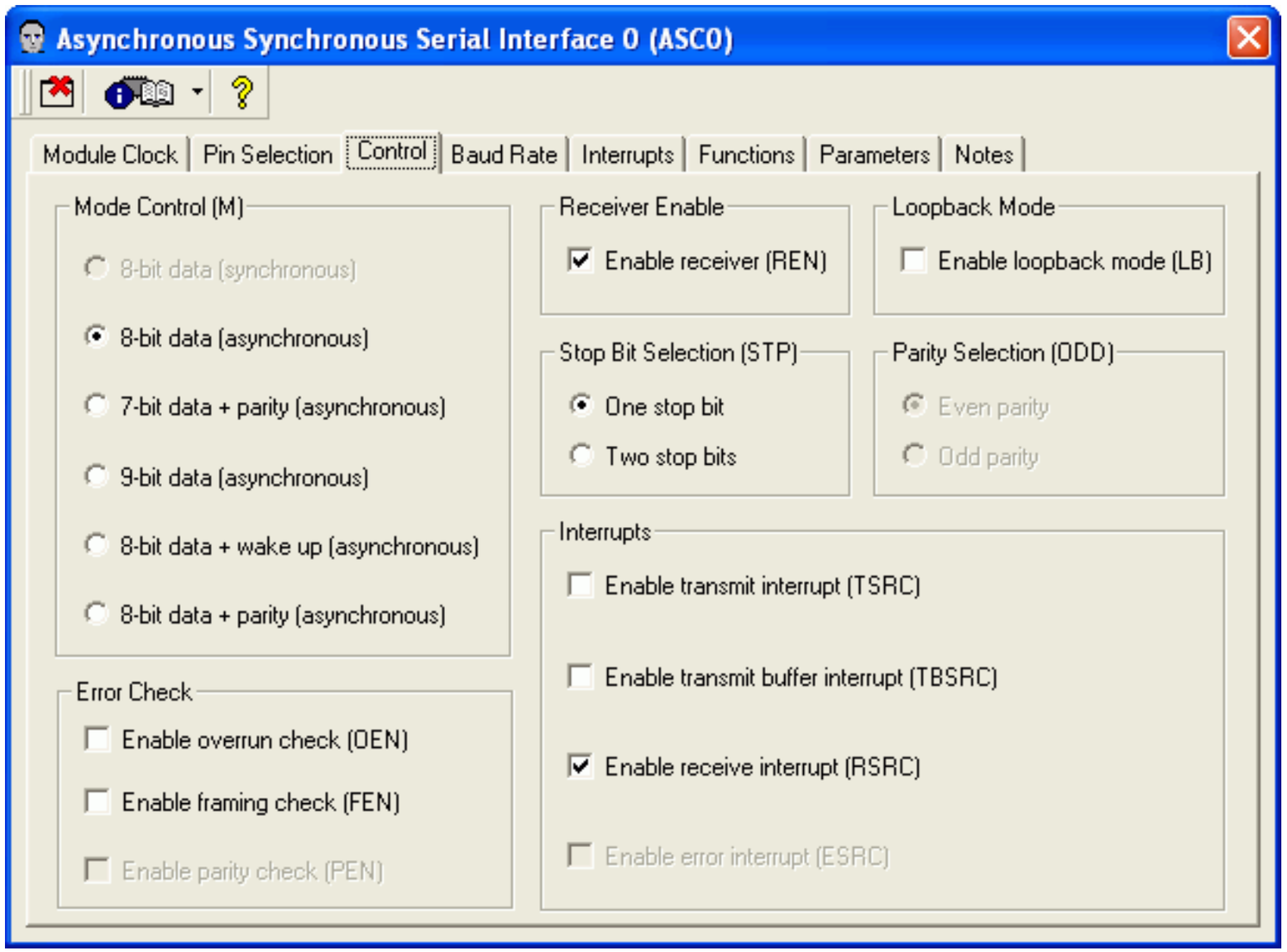
Exercise 10: Peripheral Control Processor

On **ASC0_TX** page select **Use pin P5.1 as ASC0 output signal (ASC0_TX)**.



Exercise 10: Peripheral Control Processor

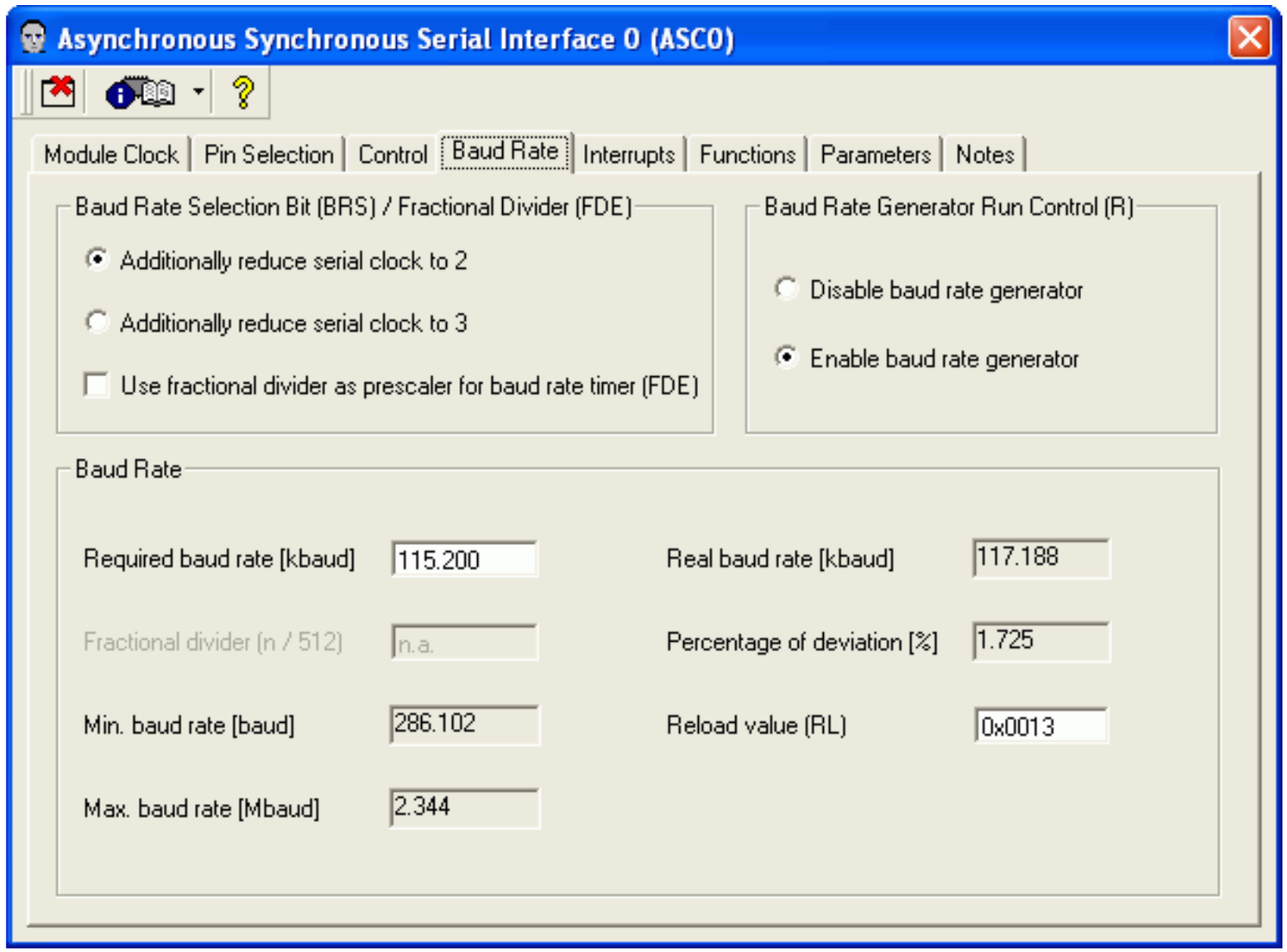
On the **Control** page
- Select Mode Control (M) **8-bit data (asynchronous)**,
- Select Stop Bit Selection (STP) **One Stop bit**,
- Check Interrupts **Enable receive interrupts (TSRC)**.



Exercise 10: Peripheral Control Processor

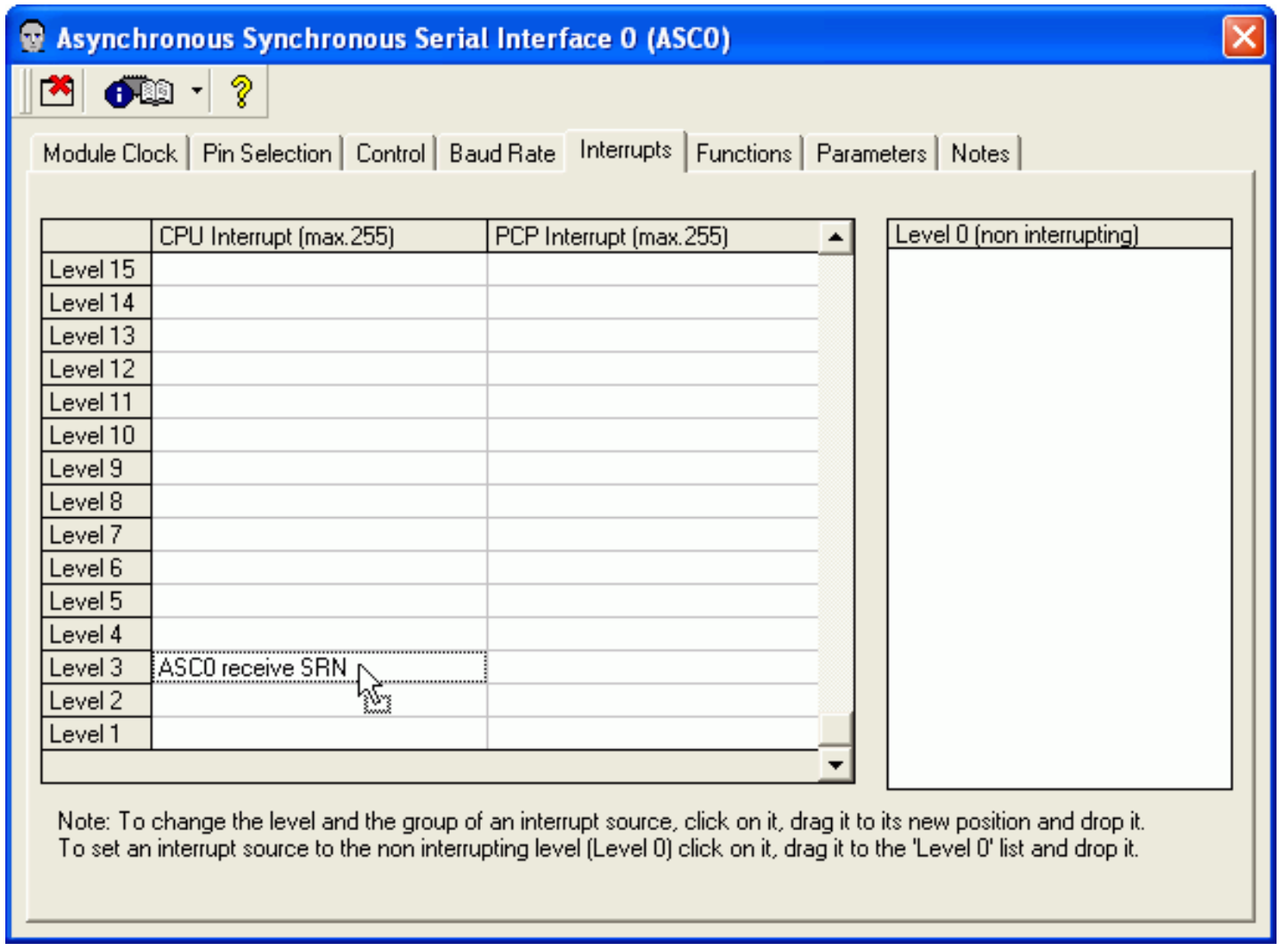
On the **Baud Rate** page

- Select Baud Rate Generator Run Control (R) **Enable baud rate generator**,
- Enter as **Required baud rate**: 115.200 . Type **Return**.



Exercise 10: Peripheral Control Processor

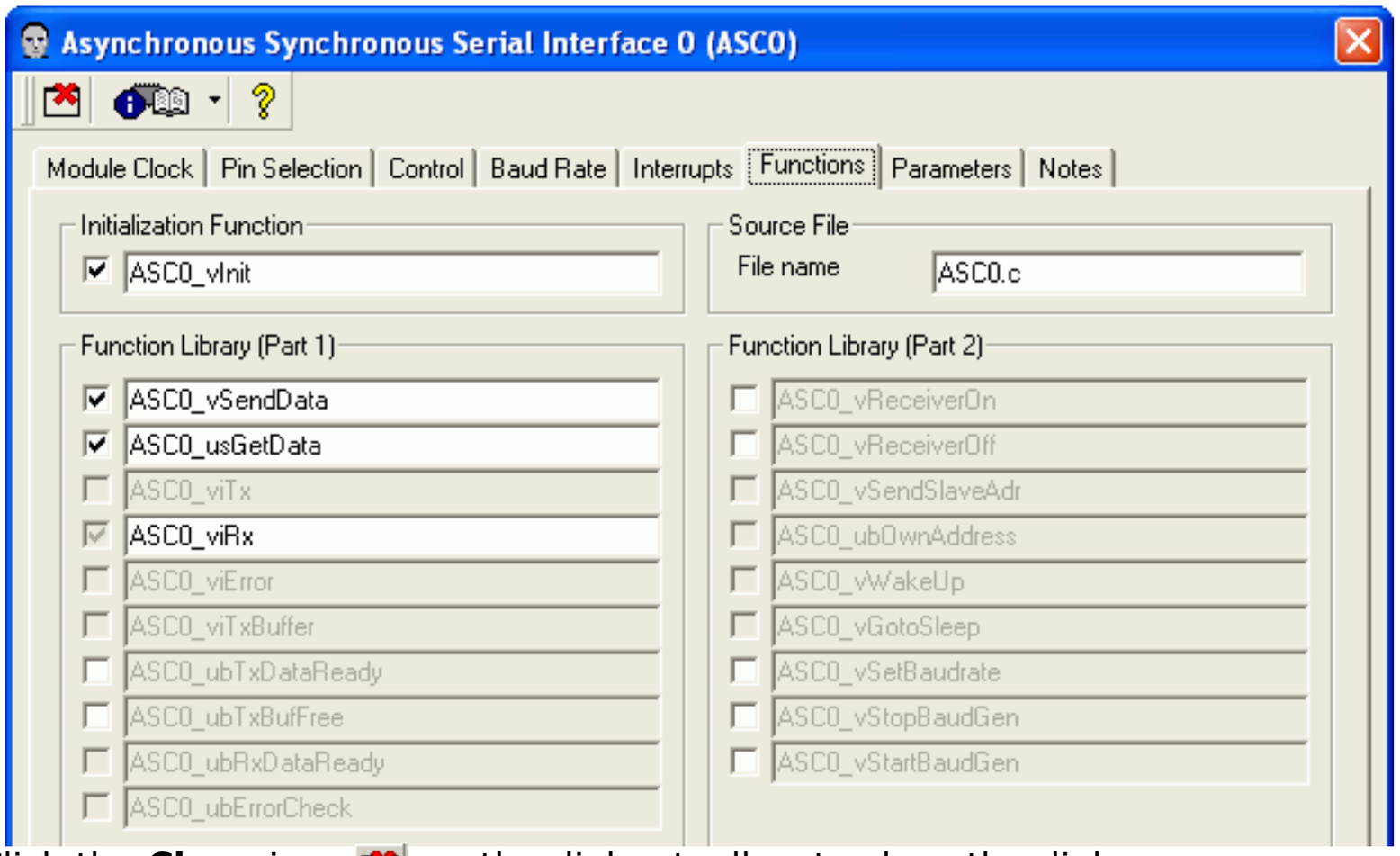
On the **Interrupts** page drag the **ASC0 transmit SRN** interrupt from the right list to the CPU interrupt level 3.



Exercise 10: Peripheral Control Processor

On the **Functions** page

■ Check the `ASC0_vInit`, `ASC0_vSendData` and `ASC0_vGetData` functions.



Click the **Close** icon on the dialog toolbar to close the dialog.

5. Save the project

Choose **File > Save** and save the project as `c:\infineon\pcp\pcp.dav`.

6. Generate the application framework

Choose **File > Generate Code** to start the code generation.

Exercise 10: Peripheral Control Processor

7. Add a new project to the Tasking Workspace

Switch to the *Tasking* EDE. Choose **File > Configure Project Space... > Add new project** and add a new project `c:\infineon\pcp\pcp.pjt`.

8. Add the application framework

In the **Project Properties** dialog click the **Scan** icon  . A dialog appears. In the Pattern field, enter `*.c;*.h`. This will select all generated files of the application framework. Select the project directory and click **OK**.

9. Set current project

Use the context menu in the workspace window to make the pcp project the current project.

10. Load the project options

Choose **Project > Load Options**. In the **Filename** field enter the `c:\infineon\tc1796_intmem.opt`. The option file can be found in the `tc1796.zip` package.

11. Add the user code

Add an endless loop to `MAIN.c` at (Main,9)

```
// USER CODE BEGIN (Main,9)
for(;;)                          // forever
  ;
// USER CODE END
```

Modify the interrupt routine in `ASC0.c`

```
void _interrupt (ASC0_RINT) ASC0_viRx(void)
{
//  USER CODE BEGIN (Rx, 2)
  ASC0_vSendData(ASC0_usGetData()); // echo any character received
//  USER CODE END
} //  End of function ASC0_viRx
```

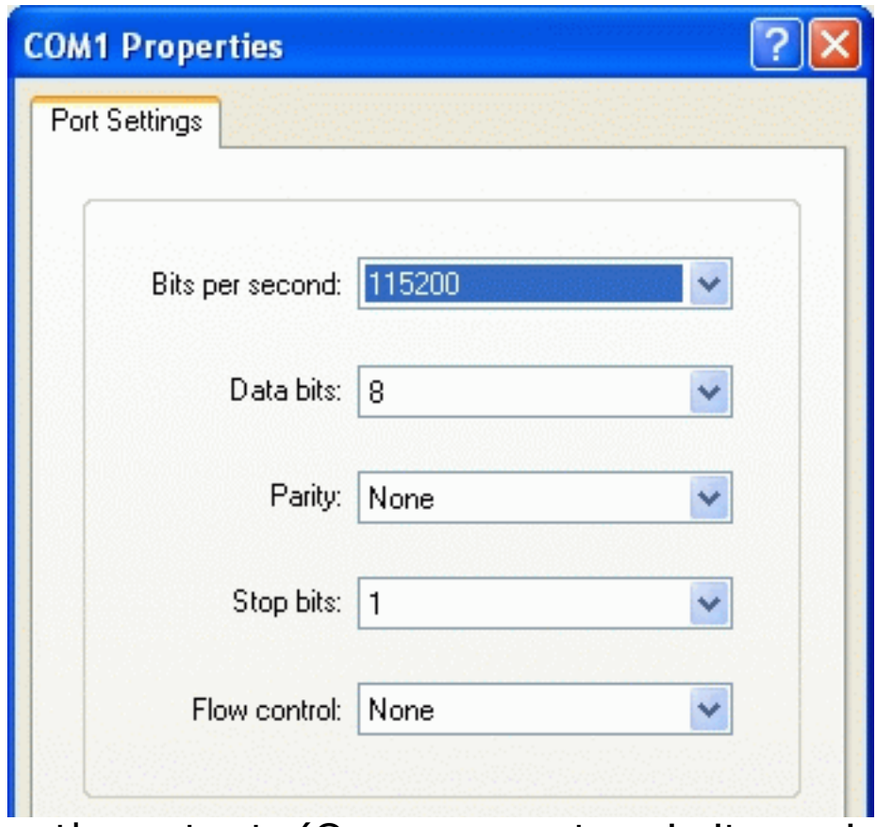
Exercise 10: Peripheral Control Processor

12. Build the application

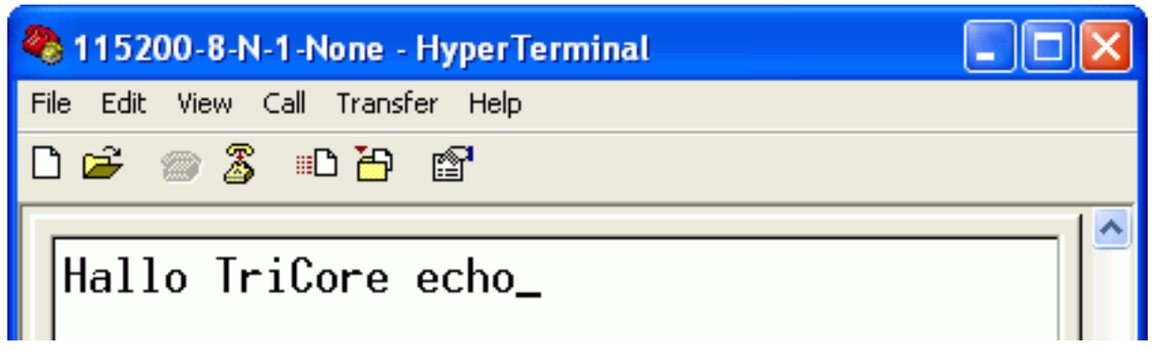
Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

13. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar. Open a HyperTerminal window. Set the COM parameter to 115200 8-N-1-None.



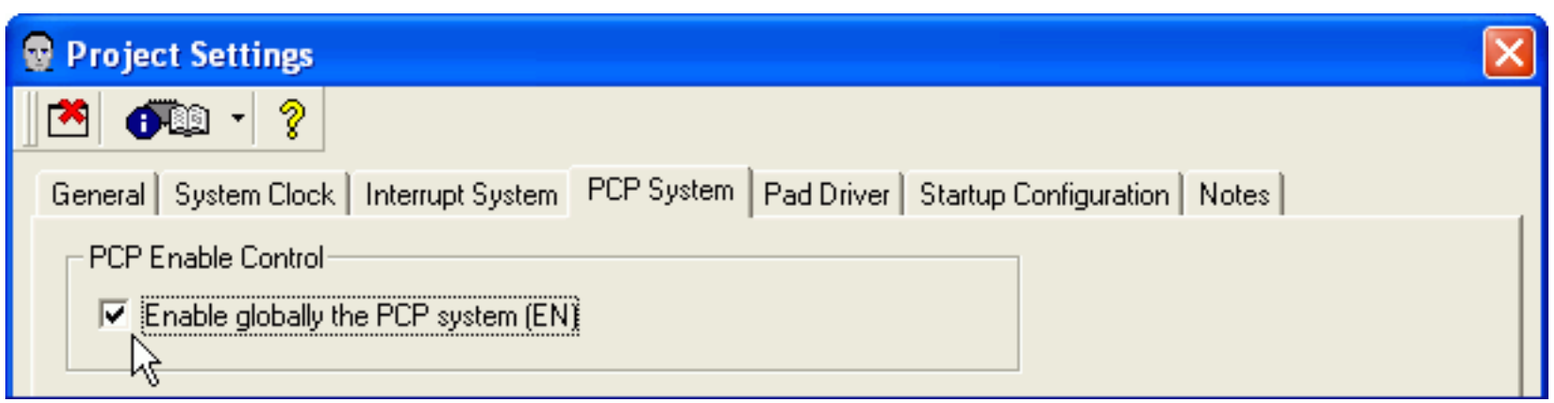
See the output. (On some system is it required to reset the application once. Choose **Run > Reset Target System**.)



Exercise 10: Peripheral Control Processor

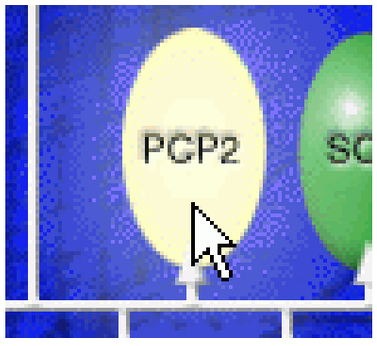
14. Modify the DAvE project.

Switch to *DAvE* to modify the project and choose **File > Project Settings.**
On the **PCP System** page
■ Check **Enable globally the PCP system**. This also automatically enables the PCP module.



15. Open the PCP properties

Click on **PCP** in the project window to open the PCP properties.

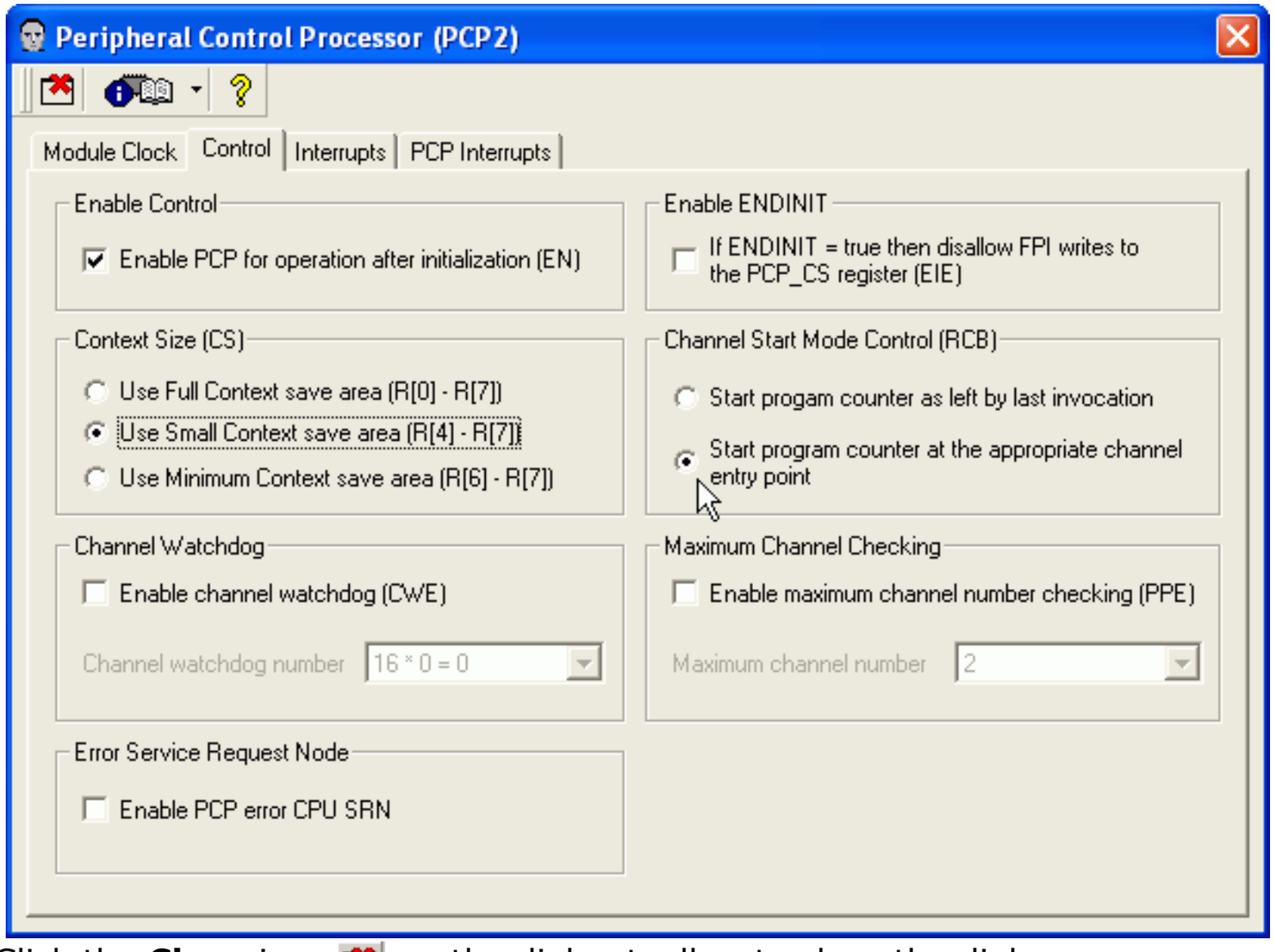


Exercise 10: Peripheral Control Processor

16. Set up the PCP properties

On the **Control** page

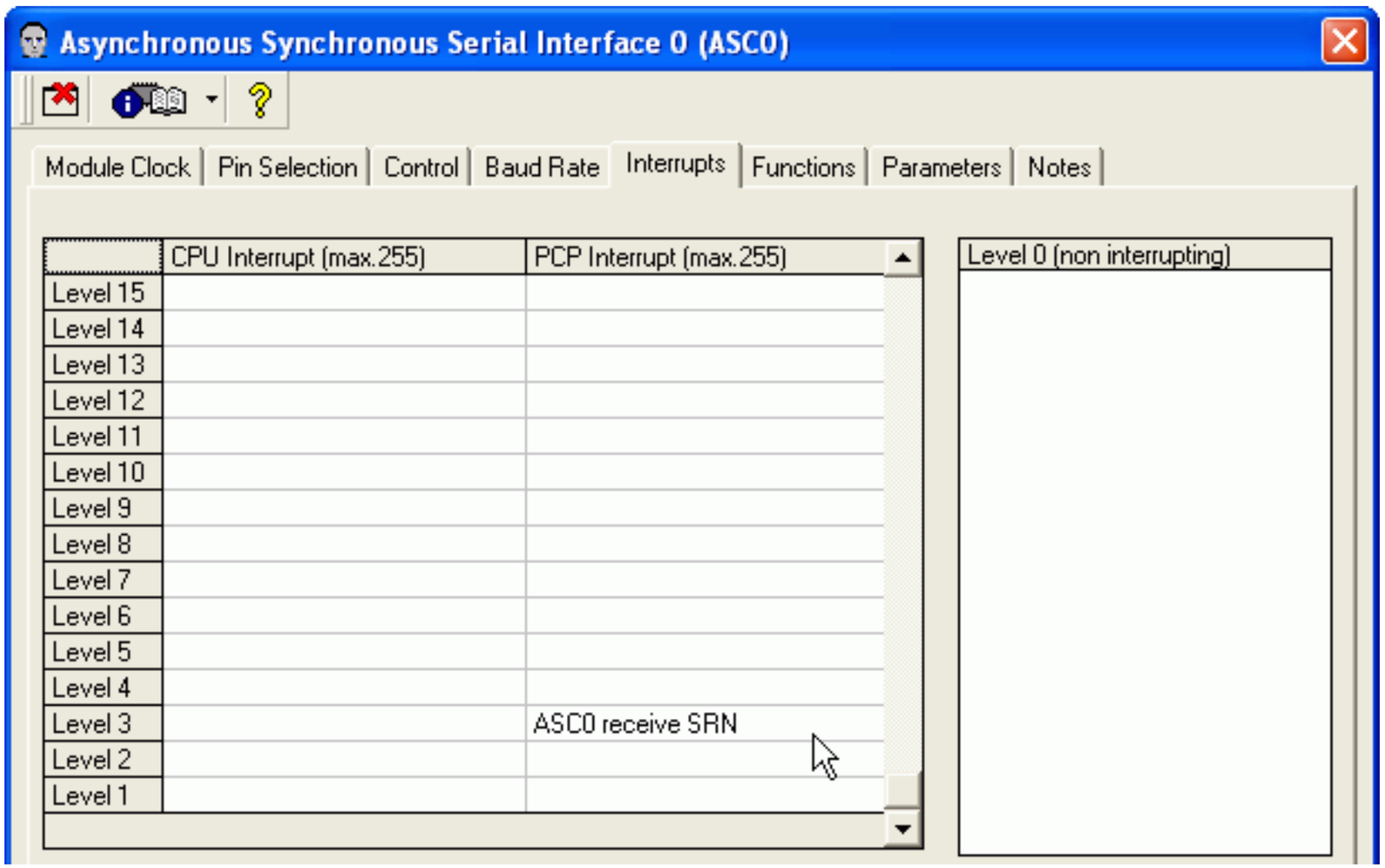
- Check **Enable PCP for operation after initialization (EN)**,
- Select Context Size **Use Small Context save area (R[4]-R[7])**,
- Select Channel Start Mode Control (RCB) **Start program counter at the appropriate channel entry point**.



Click the **Close** icon on the dialog toolbar to close the dialog.

17. Modify the ASC0 properties

Open the ASC0 properties. On the **Interrupts** page drag the interrupt from the CPU column to the PCP column.



18. Save the project

Choose **File > Save** and save the project.

19. Generate the application framework

Choose **File > Generate Code** to start the code generation.

Exercise 10: Peripheral Control Processor

20. Modify the Tasking project

Add a new file `pcp.pcp` to the *Tasking* project.

```
; PCP Context Save Area
  .org 3*4, DATA, INIT    ;3 channels (0,1,2) not used. 4 words for small context
ch3_context:
  .word 0x00000040          ;R[7] CEN=1 channel enable set, DPTR=0
  .word 0x03000000          ;R[6] current priority CPPN=3
  .word ASC0_TBUF           ;R[5] intitialize to ASC0_TBUF
  .word ASC0_RBUF           ;R[4] intitialize to ASC0_RBUF

; Entry table for the PCP vector Mode start of channel 3 (SPRN 3)
  .org 3*2, CODE, INIT      ;3 channels (0,1,2) not used. 2 words for each vector
ch3:
  jc.a  ch3_start, cc_UC
ch3_start:
  ld.f  R1, [R4], SIZE=8    ;load receive buffer from FPI bus address to R1
  st.f  R1, [R5], SIZE=8    ;store R1 to transmit buffer at FPI bus address
  exit  EC=0, ST=0, INT=0, EP=0, cc_UC ;exit w/ no counter action, not channel stop,
                                       ;no interrupt, set PC to channel start
```

21. Build the application

Click the **Build** icon  on the Build toolbar. The Build process finishes successfully.

Exercise 10: Peripheral Control Processor

22. Run the application

Click the **Run** icon  on the *CrossView Pro* toolbar. Open a HyperTerminal window. Set the COM parameter to 115200 8-N-1-None and try out the terminal