Application Note, V 1.0, April 2005

AP32087

TC1766

Micro Link Interface: Quick Start.

Microcontrollers

Never stop thinking.

Revision History:	2005-04	**V 1.0**
Previous Version:	-	

Page	Subjects (major changes since last revision)

Controller Area Network (CAN): License of Robert Bosch GmbH

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Edition 2005-04

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

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

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

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

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

Table of Contents **Page**

1 Scope

The goal of this document is to provide practical information on how to configure and program two TC1766 microcontrollers in order to establish a Micro Link Interface (MLI) link and execute basic read and write transfers.

Some programming examples are included with this application note. The code has been created using DAvE 2.1[1)] and Tasking 2.2r1[2)]. It is strongly recommended to uses those versions when using the code delivered with this application note.

Two TriBoards for TC1766 and two standard PCs are needed to run the application described in this document.

Please note that the code given in this application note shall be used for demonstration purpose only. It aims at giving an example on how to build a functional MLI link. It is not optimized nor is its robustness guaranteed.

Section 2 gives an overview of the MLI interface. Readers already familiar with the MLI may want to skip this section. Section 3 gives explanations on how to set-up a basic MLI link between two microcontrollers, from a functional point of view. In section 4, step-by-step explanations are given in order to build physically the connection. This includes hardware set-up, initialization of the microcontrollers, programming of the start-up procedures and of basic read and write operations. In section 5, some explanations are given on the ready-to-use files provided with this application note.

1) A compatible DavE DIP file is included in the package containing this document.
2) For more information about Tasking Tool Chain and the latest patches, please visit www.tasking.com

2 Introduction to the Micro Link Interface

MLI is a serial high speed link (up to 40 Mbaud for TC1766), which is based on a principle similar to the standard serial protocol. Due to its symmetrical structure, it supports full-duplex transmission. It only requires four signal lines in each data direction (downstream = transmit and upstream = receive). For each data transfer direction, there is one clock line (CLK), one data line (DATA) and two handshake lines (READY, VALID). One MLI transmitter might be connected to up to four scalable MLI receivers sharing the same Data and Clock line. An individual addressing of receivers is done by independent sets of handshake lines.

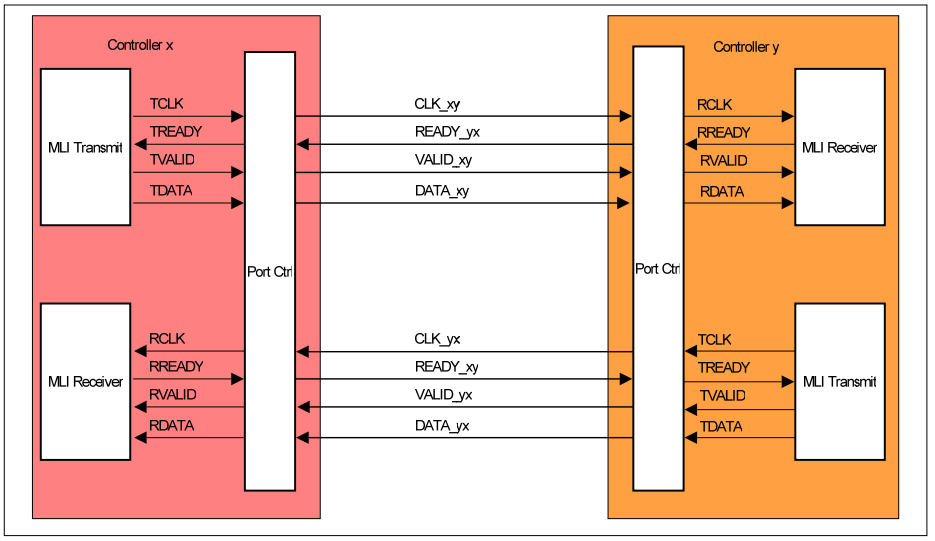


Figure 1 MLI Transmitter – Receiver connection.

The MLI interface has been developed to meet the following application targets:

- Data and program exchanging without intervention of CPU or PCP between microcontrollers of the AUDO-NG family. The MLI is connected to the system bus and can do data move operations independently from the CPU(s).
- The internal architecture of the block allows the communication between controllers in different clock domains.
- The read mode enables the desired data to be read from the other controller.
- Resources sharing between controllers.

- Capability of triggering interrupts in the receiving controller by sending a command.

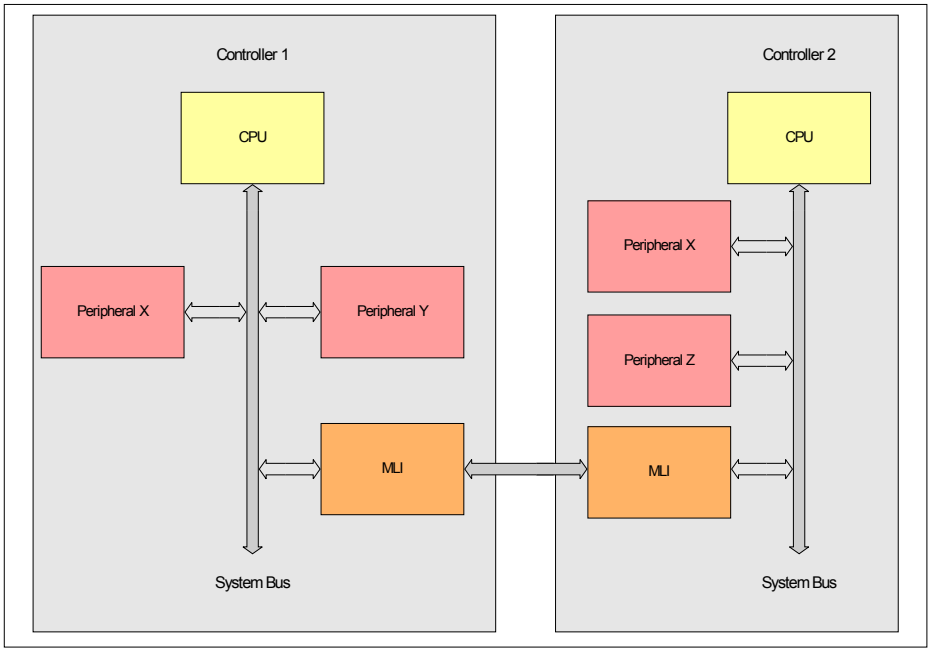


Figure 2 MLI in a microcontroller

MLI I/O-pins are CMOS compliant, allowing microcontrollers from the AUDO-NG family to be mounted closely together on the same PCB. This target doesn't necessarily require cost-extensive LVDS drivers for better EMC behavior. Usage of CMOS MLI I/O drivers instead of LVDS drivers also has a beneficial impact on the absolute current consumption and requires less interface pins. Nevertheless there might be applications, where LVDS drivers are useful for MLI signals; e.g. for electronic valve train, where the ECU for the valves and the ECU for engine control are separated and need to communicate via longer MLI cable (up to more than 1 meter might occur). As a different cable length for the connection leads to a changing loop delay for transmitted or received messages, the timing of the MLI handshake signals can be adapted via programming during the startup procedure.

The internal architecture of the MLI interface supports different clock domains for the transmitter or the receiver module. As the MLI interface is able to act as bus master on the system bus of the controller itself, it autonomously acts like a DMA controller and

therefore might work in parallel to the CPU of the system. As a result, the MLI significantly reduces the CPU load for data transfer tasks. Remote control of peripherals located in the “other” controller is offered as a feature by this behavior; so calculation power or peripherals located in different sub-controller systems might be shared via MLI.

MLI connection is not necessarily restricted to a controller-to-controller connection. Other products, such as smart companion devices (ASSP) can also be connected easily. The advantage of these devices is their extended voltage range, so that they could incorporate e.g. a 5V analog sensor interface or other analog and digital data preconditioning circuits.

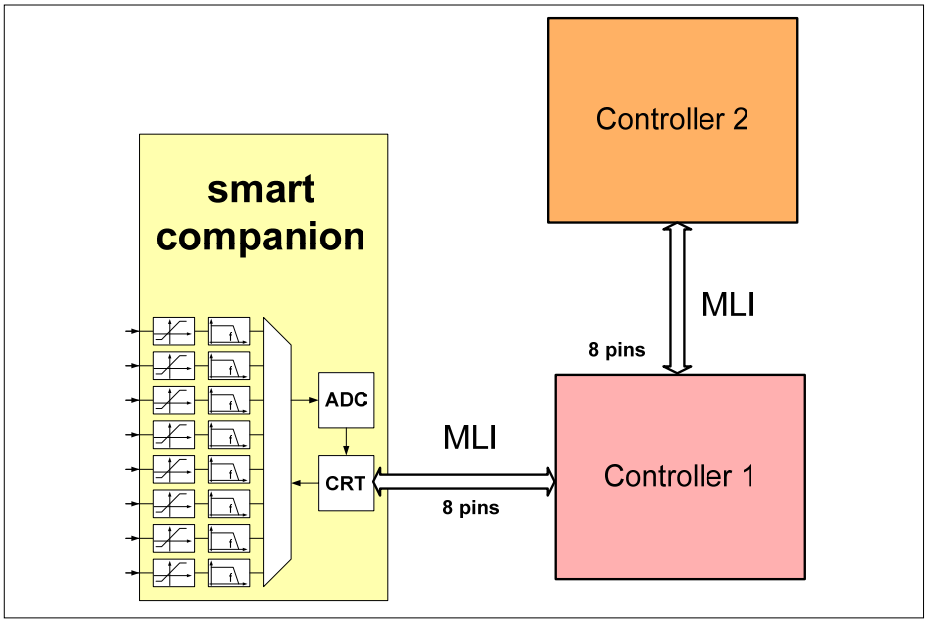


Figure 3 Smart companion device with MLI connection.

General Description of MLI

The communication between two participants is based on a pipe structure. A pipe may be seen as a logical connection between a transmitter and a receiver. In each MLI module, 4 independent pipes are available. The pipes point to address areas in the receiver, starting at programmable base addresses. The MLI transmitter only sends a short offset relative to the base address instead of the full 32-bit address. Each pipe

defines a buffer in the receiver's address map (defined by the base address, the offset and the length of the offset).

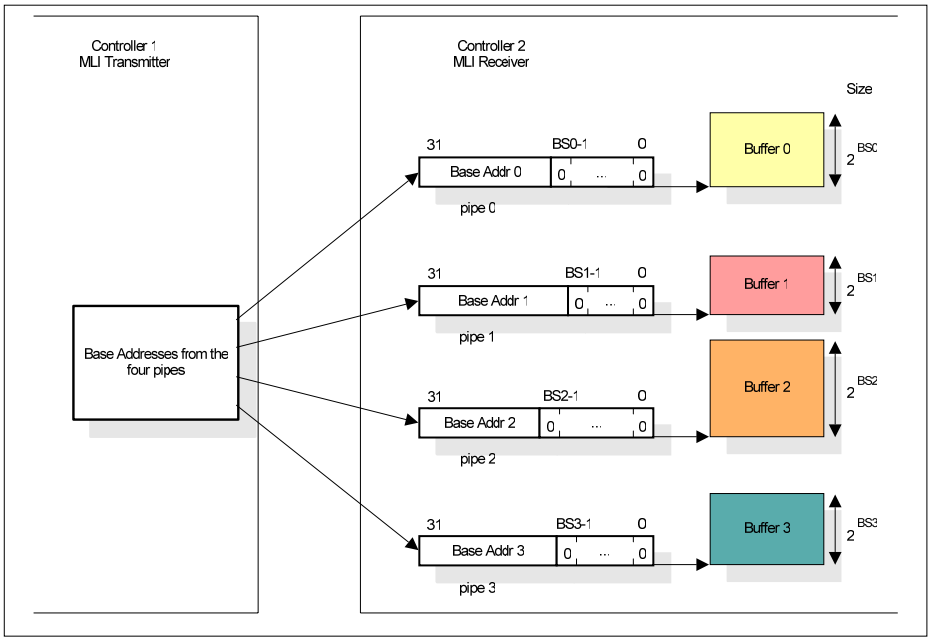


Figure 4 MLI pipe structure.

In addition to the offset (its bit width defines the buffer size), the MLI transmitter sends a reference to the pipe in use. When the MLI receiver obtains this data it elaborates the absolute target address by simply concatenating the received offset to the base address of the selected pipe.

A data write access to a pipe in controller 1 leads to an automatic transfer from the MLI transmitter to the MLI receiver on controller 2. This transfer includes the written data, the offset address and the pipe number. The received information becomes available in the MLI receiver. The CPU of controller 2 can read it under SW control or the MLI can autonomously write it to the given address. In order to avoid write actions to safety-critical address areas, an access protection scheme has been added.

A read access to a pipe transfers a request to the MLI receiver on controller 2. If enabled, the MLI executes the read operation autonomously and the requested data will be sent back to the MLI on controller 1 (by the MLI transmitter on controller 2 to the MLI receiver of controller 1). When this information is available in the MLI module of

controller 1, an interrupt can be generated and the CPU (or a DMA, etc.) of controller 1 can read the requested data.

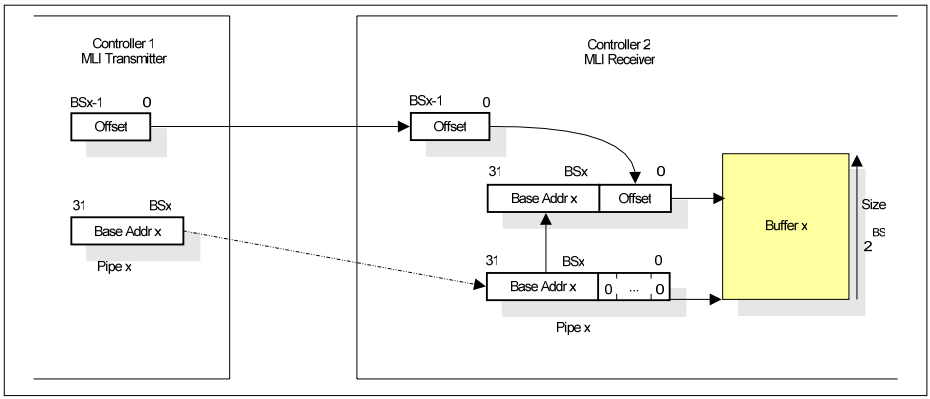


Figure 5 Target address generation.

The kernel MLI includes an optimized mode to transfer data blocks. Whenever the MLI transmitter detects that the new address and the previous one follow a predictable scheme, it will send just the data reducing this way the number of transferred bits.

If the complete autonomous feature set of MLI connection is enabled, data transfers between two participants can take place without any CPU action for data handling. The transmitting MLI stores the write access to its pipes, does the complete encoding and transfers the complete move action to the receiving MLI. There, the address and the data are reconstructed and the write action is executed on the system bus. As a result, a MLI module can be seen as a fully autonomous module transferring data moves between the system buses of independent controllers.

3 How to set – up an MLI connection

3.1 Goal

The goal of this application note is to set-up an MLI link between two microcontrollers MC1 and MC2, as depicted in Fig. 6.

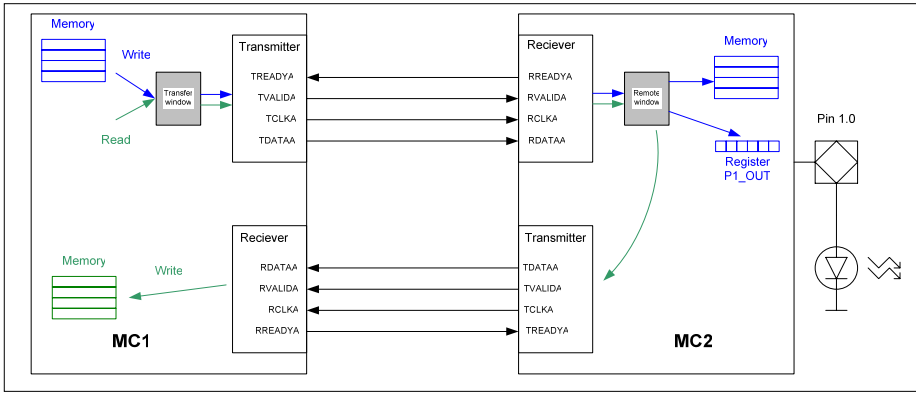


Figure 6 MLI link between two controllers.

Both microcontrollers run at 80 MHz CPU frequency.

The MLI links runs at 10 Mbaud/s.

Specifically, the following actions will be performed via the MLI link:

- MC1 writes to the register P1_OUT of MC2. The effect is that the LED on the TriBoard of MC2 is turned on.
- MC1 writes six data words to the address space of MC2 (address 0xd000a000, 0xd000a004,…, 0xd000a014).
- MC1 reads some 6 words in the memory space of MC2 (address 0xd0008000, 0xd0008004,…, 0xd0008014), and stores them in its own memory space at address 0xd0006000, 0xd0006004,…, 0xd0006014 respectively.

3.2 Functional description

This section describes from a functional point of view the different steps necessary to create the MLI link. The principles developed here are general and can be used in most of the cases. However, practical implementations may differ from the description below, depending on the specific applications requirements.

3.2.1 Start-up

When the two controllers are powered on and their respective MLI is statically initialized (module enabled, pin assignment, etc.), some procedures are needed to initiate a transmission.

At the beginning, all MLI Service Request Nodes (SRN) of MC1 are disabled. They will be enabled later on. This solution has been chosen here in order to avoid unwanted servicing of routines initiated by dummy frames.

On the contrary for MC2, at the beginning, one SRN is enabled (interrupt on received command frames on pipe 3). All other SRN are disabled.

MC1 is used as local controller during the transmission, MC2 as the remote controller. Please note that during the start-up procedures described below, MC2 is also used briefly as local controller, and MC1 as remote.

The initialization procedures are divided into 3 steps:

First step: MC1 initiates the start-up procedures and stands-by.

Second step: getting MC2 ready for communication.

Third step: getting MC1 ready for communication.

3.2.1.1 Start-up procedure: first step.

During this step, MC1 first configures both local transmitter (MC1) and remote receiver (MC2) so that the parity error signaling is performed correctly; MC1 also configures the base address of the remote window and initiates some start-up procedures in MC2. A flow chart describing step 1 is depicted Fig. 8.

First, the local controller MC1 has to configure its transmitter and the remote receiver (of MC2) so that parity error signaling is performed correctly. This is performed according to the set-up procedure described in the User's Manual [1)]. For the sake of

[1)] User's Manual, Peripheral Units, V0.2, Dec 2004, section 21.1.9.

clarity, in order to avoid mismatch between this specific procedure and the rest of the procedures described here (which are also start-up procedures), the “set-up procedure” described in the User’s Manual will be referred as “parity error start-up procedure”, or PESP.

PESP

A flow chart describing the PESP is depicted Fig. 7.

MC1 sends a dummy frame to MC2 (in this case, a command frame on pipe 0). It waits for the transfer to complete and then measures how many cycles have elapsed between the beginning of the transfer and the moment when the signal Ready toggles from Low to High. The measurement is done by reading the bit field MLI0_TSTATR.RDC.

This value represents the overall loop delay, as defined in the PESP description. In the case of this example, the value RDC+1 is written to bit field MLI0_TCR.MDP.

MC1 then sends a command frame on pipe 1 to write on the remote controller bit field MLI0_RCR.DPE. In the case of this example, MLI0_RCR.DPE is chosen to be MLI0_TCR.MDP + 2.

As defined in the user’s manual, these settings need to be tested. First, a dummy frame with parity error is sent (by setting bit MLI0_TCR.TP to 1) and the software checks if the error is detected by the transmitter (by checking bit MLI0_TSTATR.PE). If not, special actions must be taken and the start-up procedure must be restarted from the beginning.

If an error is correctly detected, then MC1 sends a dummy frame with no parity error. The software checks if no error occurs (by checking bits MLI0_TSTATR.PE and MLI0_TSTATR.NAE). If an error is detected, special actions must be taken and the start-up procedure must be restarted from the beginning. If not, this finishes PESP.

End of step 1.

Once the PESP is correctly executed, MC1 enables interrupts on received command frames on pipe 3 (interrupt source DMA_MLI0SRC1, SRPN = 2 in this example). This pipe will be used by MC2 to tell MC1 that it is ready for communication.

The MC1 sends then four copy base address frames, in order to configure the remote window of MC2.

Once this is done, it sends a command frame (pipe 3 / code 0 in this case) to the remote controller and stands-by, until the MC2 sends a command frame on pipe 3.

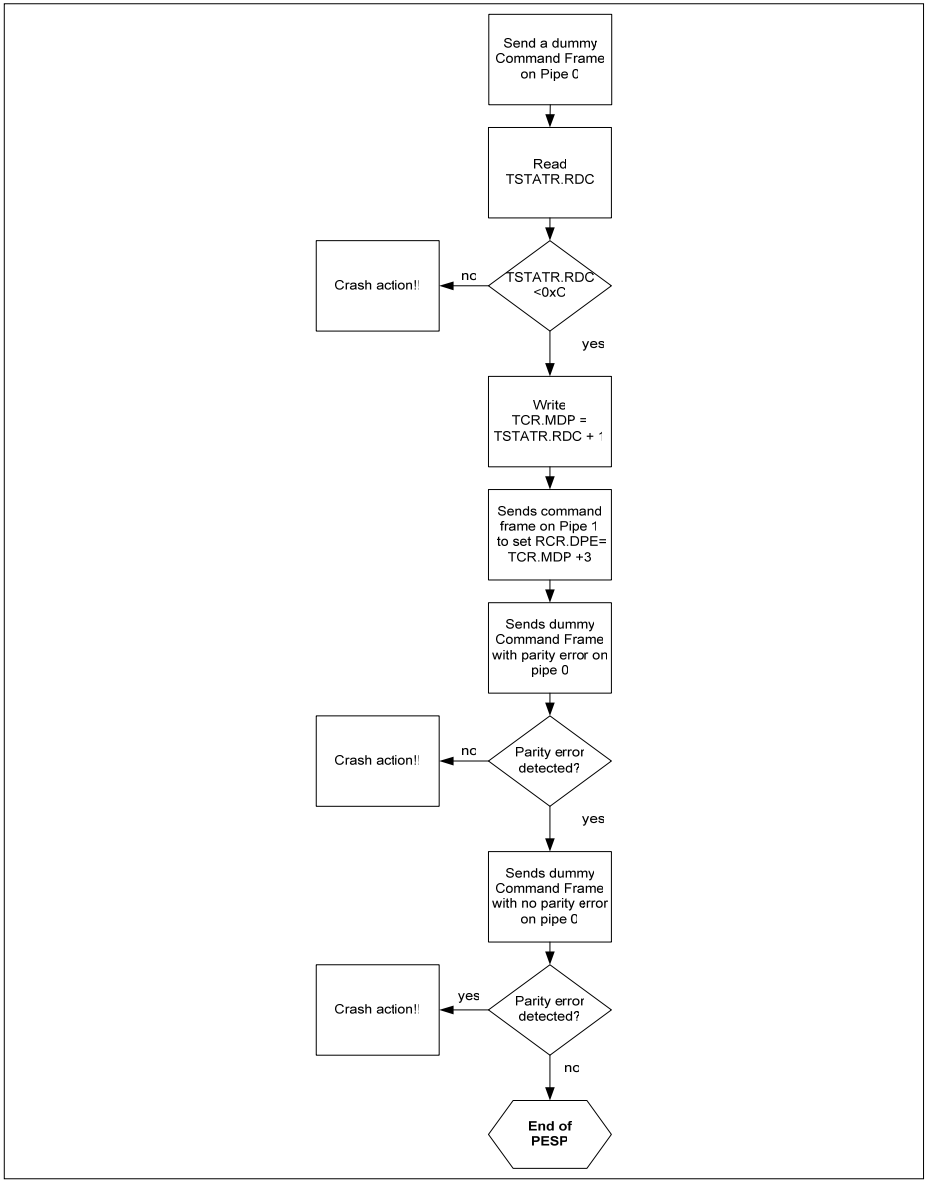


Figure 7 Parity Error Signaling Procedure (PESP).

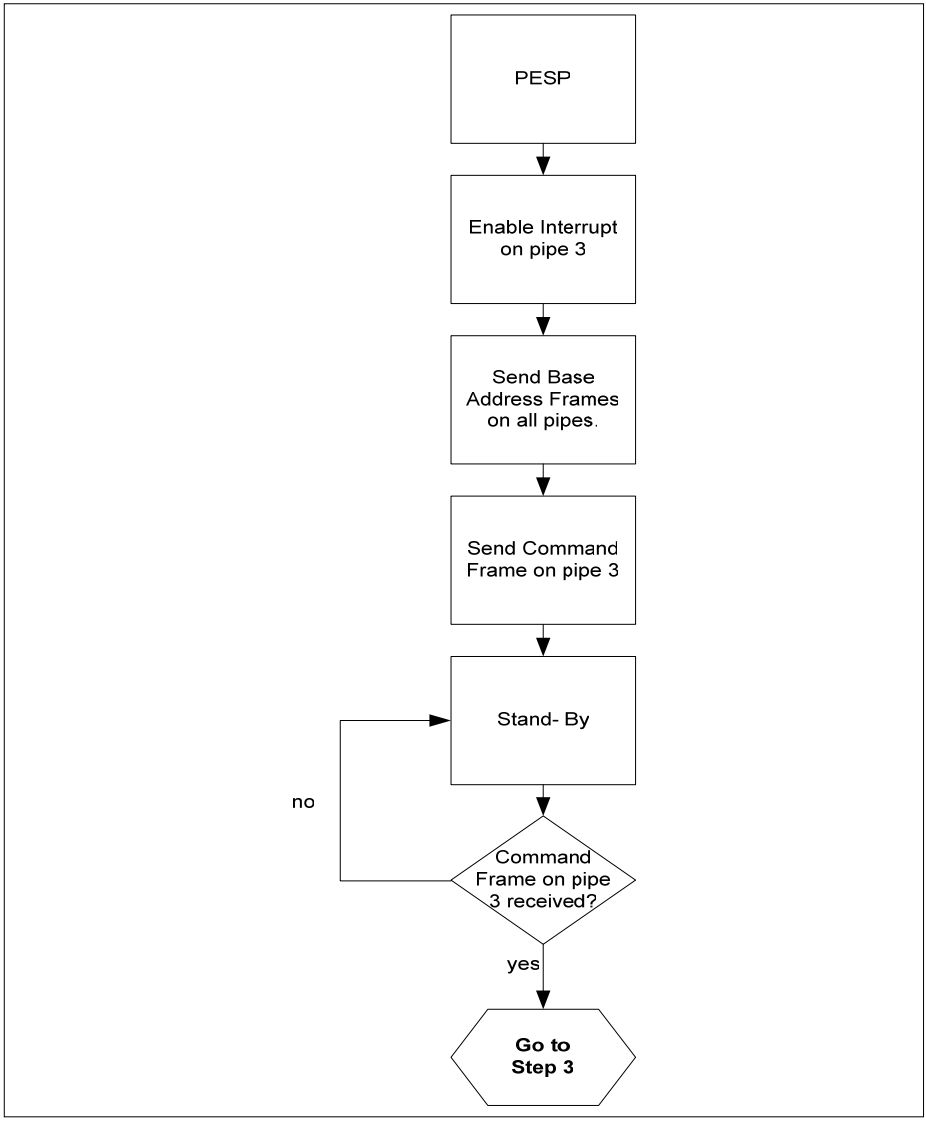


Figure 8 Start-up procedure (MC1): step 1.

3.2.1.2 Start-up procedure: second step.

This step is needed for two reasons. First, it enables MC2 to know when to clear all its interrupt flags and its error flags generated by the frames received in the first step. Secondly, it lets MC1 know when MC2 is ready for transmission. A flow chart describing step 2 is depicted Fig. 9.

When MC2 receives a command frame on pipe 3, an interrupt routine is started. During this routine, MC2 will operate as the local controller.

First, a PESP, as described in the User's Manual (and similar to the one described in step one) is performed. It configures the transmitter of MC2 and the receiver of MC1 for parity error signaling.

Once the PESP is correctly executed, MC2 clears all transmit and receive errors bits (MLI0_TRSTAR.CNAE, MLI0_TRSTAR.CTPE, MLI0_RCR.PE). It also clears all the interrupt flags on the receive side (by writing to register MLI0_RIER).

It then enables the automatic move engine (by setting bit MLI0_RCR.MOD).

MC2 sends a command frame to MC1 on pipe 3, in order to indicate it is ready for transmission.

Finally, it clears transmit interrupt flags (by writing to register MLI0_TIER).

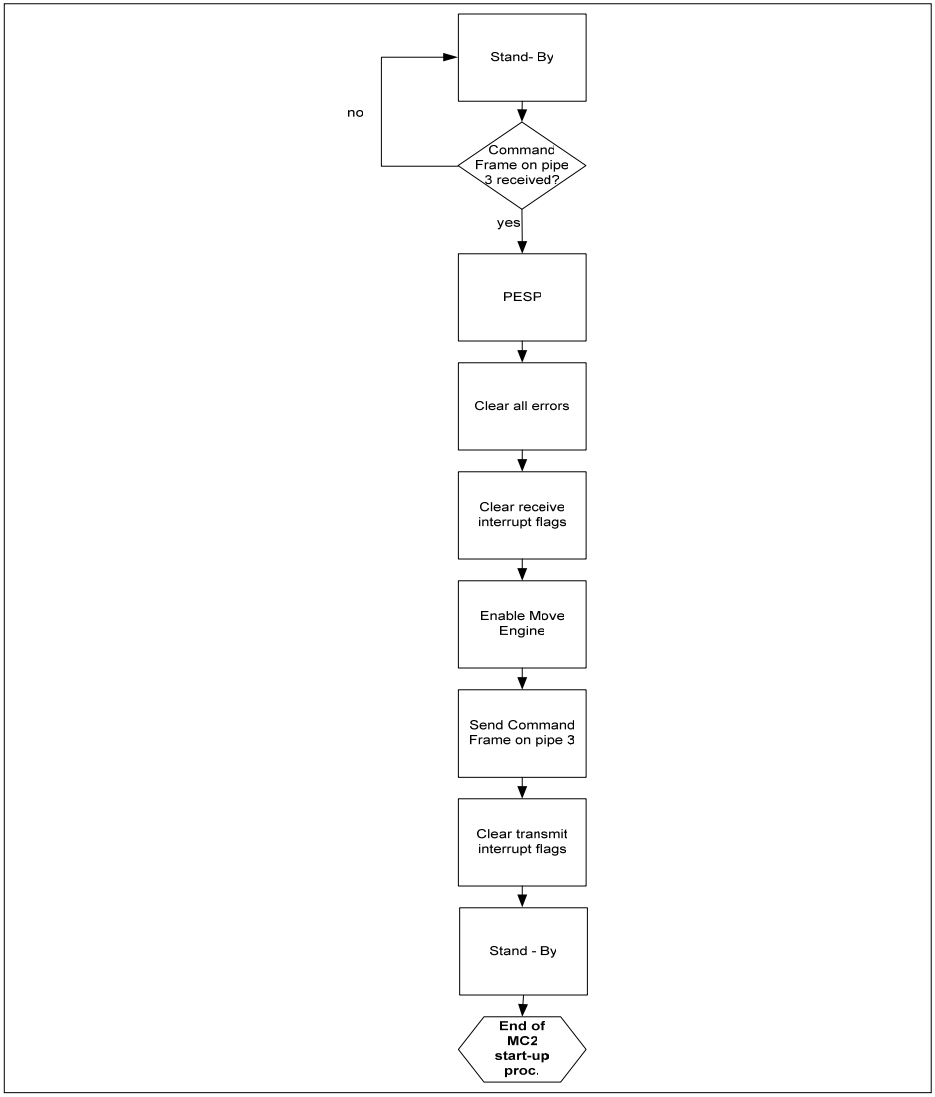


Figure 9 Start-up procedure (MC2): step 2.

3.2.1.3 Start-up procedure: third step.

This step finalizes the start-up of MC1. When MC1 acknowledges the command frame on pipe 3, it will initiate an interrupt routine, where several actions are performed. A flow chart describing step 3 is depicted Fig. 10.

First, all errors flags are cleared (bits MLI0_TRSTAR.CNAE, MLI0_TRSTAR.CTPE, MLI0_RCR.PE).Then the interrupt flags (registers MLI0_RIER and MLI0_TIER) are reset. Interrupts are besides enabled on the reception of normal frames (interrupt source DMA_MLI0SRC0, SRPN = 1 in this example). This interrupt source is used when answer frames are received.

Optimized frame mode is then enabled (by setting MLI0_TCR.NO)

Finally a flag will be set, which will trigger the read / write operations (in this example, the flag is a global variable called MLI_Remote_Ready).

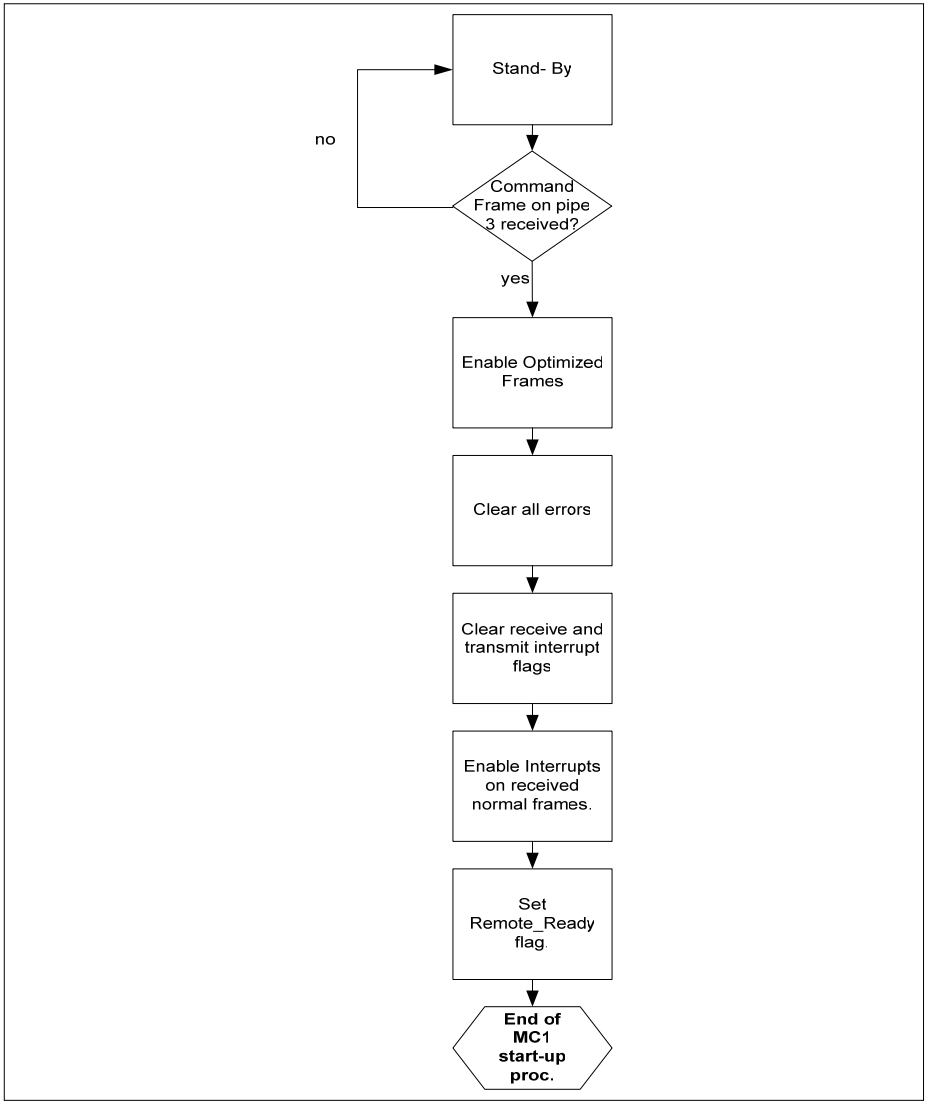


Figure 10 Start-up procedure (MC1): step 3.

3.2.2 Read and Write operations.

At this point, all errors or interrupt flags generated in both controllers by the start-up procedure have been cleared. The parity error signaling has been checked and is functional, and the remote window of MC2 has been configured. Besides, both controllers are now ready to communicate with each other. MC1 can now start read and write operations.

The write and read operations are triggered by the fact the flag MLI_Remote_Ready is set to 1.

Write operation 1.

The local controller MC1 sends a Write frame on pipe 3. It writes the value 0x00000000 to register P1_OUT of MC2. The effect is that the LED on the remote TriBoard is switched on as soon as the transfer is completed.

Write operation 2.

The local controller MC1 sends 6 Write frames on pipe 1 in a row. In a row means here that it sends frame 0, waits for bit MLI0_TRSTATR.DV1 to be set to 1 and then to get cleared, then it sends frame 1, etc. It writes the words 0xaaaa0000, 0xaaaa0001, …, 0xaaaa0005 respectively to the following memory locations in the memory space of MC2: 0xd000a000, 0xd000a004,…, 0xd000a014.

Note: Actually, the write operation is not performed directly on the DMI memory but on ist image. That is why, in the code, the base address of pipe 1 is configured to be 0xe8408000 (and not 0xd0008000.,

On the remote controller side, the transfer is handled automatically by the move engine.

Read operation.

The local controller MC1 sends 6 Read frames on pipe 0 in a row. In a row means here that it sends frame 0, waits for bit MLI0_TRSTATR.RP0 to be set to 1 and then to get cleared, then it sends frame 1, etc. It reads the words 0xffff0000, 0xffff0001, …, 0xffff0005 respectively to the following memory locations in the memory space of MC2: 0xd0008000, 0xd0008004,…, 0xd0008014.

On the remote controller side, since the move engine is activated, it will automatically pass the wanted data to its transmit buffer as soon as a read frame is received. The remote controller sends then the corresponding answer frames to MC2.

When MC1 receives the answer frame, an interrupt request is generated and the CPU services the routine. The data word of answer frame 0 (0xffff0000) is written to 0xd0006000, the data word of frame 1 (0xffff0001) to 0xd0006004, etc.

4 Practical Implementation

The following items are necessary to realize the set-up described below:

- Two TriBoard Evaluation board for TC1766.
- Two TriBoard Logic Analyzer Extension Board.
- Tasking Tool Chain (TriCore Compiler, Assembler, Linker/Locator, CrossView Pro Debugger) version 2.2r1.

Note: The Quick Start may not work with a demo version of the Tasking Tool Chain. Please contact Tasking a full featured version for demo purpose (time limited). For more information, please visit www.tasking.com.

- DAvE, the Digital Application Engineer, version 2.1. Please install the DIP file for TC1766 included in the package containing this document.
- 2 standard PC (with Windows NT, XP or Windows 2000).

4.1 Hardware connection

The required MLI connection between the two controllers is described in Table 1:

Local Controller		Remote Controller	
Signal	**Pin**	**Signal**	**Pin**
TCLKA	P2.0	RCLKA	P2.4
TREADYA	P2.1	RREADYA	P2.5
TVALIDA	P2.2	RVALIDA	P2.6
TDATAA	P2.3	RDATAA	P2.7
RCLKA	P2.4	TCLKA	P2.0
RREADYA	P2.5	TREADYA	P2.1
RVALIDA	P2.6	TVALIDA	P2.2
RDATAA	P2.7	TDATAA	P2.3

Table 1 Physical MLI connections between remote and local controllers

For example, pin P2.0 of the local controller is connected to pin P2.4 of the remote controller, etc. All the other pins will not be used and thus may remain open. The connection between the two TriBoards is depicted Fig. 11.

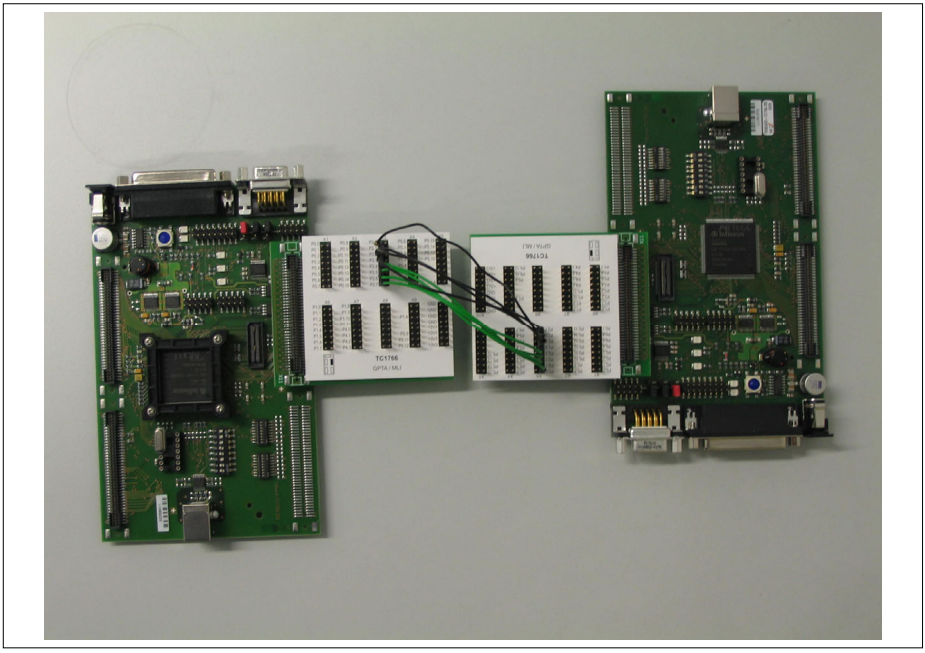


Figure 11 Hardware connection overview.

Note: Make sure that the following resistances (0 Ohm) are removed from both TriBoards: R531, R532, R533, R534, R535 and R536 (please refer to TriBoard manual).

4.2 Setting up MC1

All the operations below shall be performed on the PC connected to MC1.

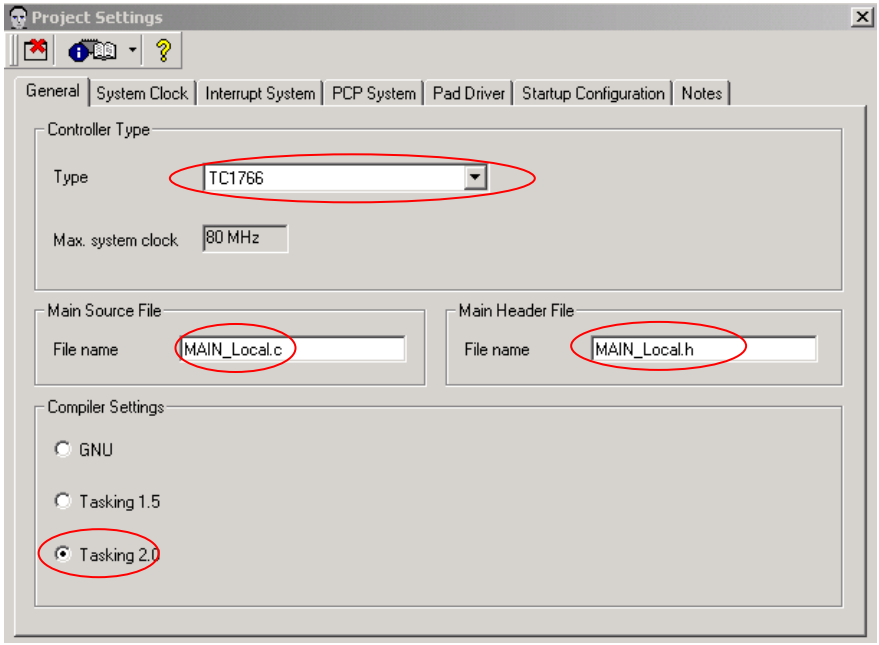
4.2.1 Configuration of the local controller

The local controller MC1 can be configured using DAvE (v2.1).

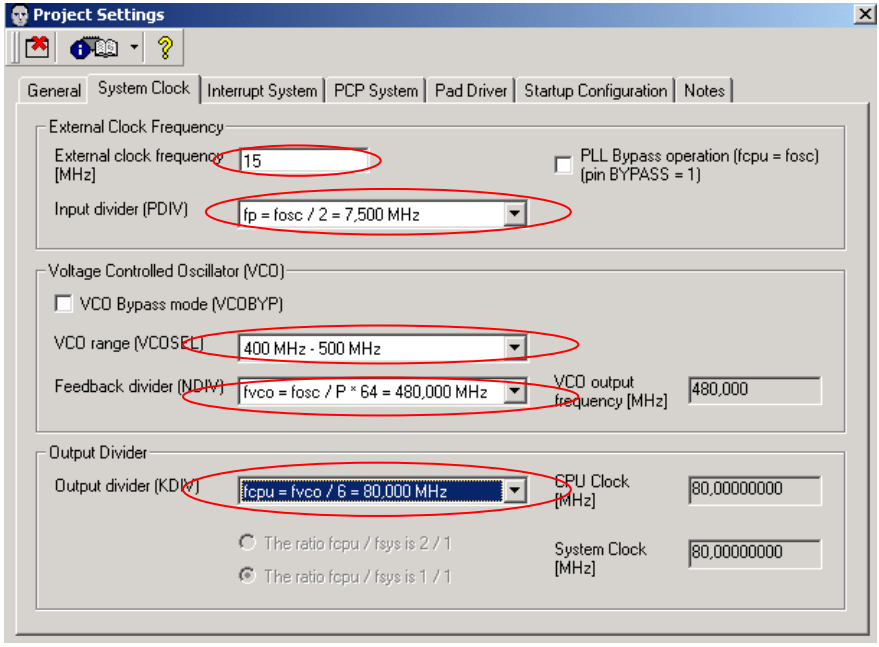
Open a new project for TC1766.

Project settings:

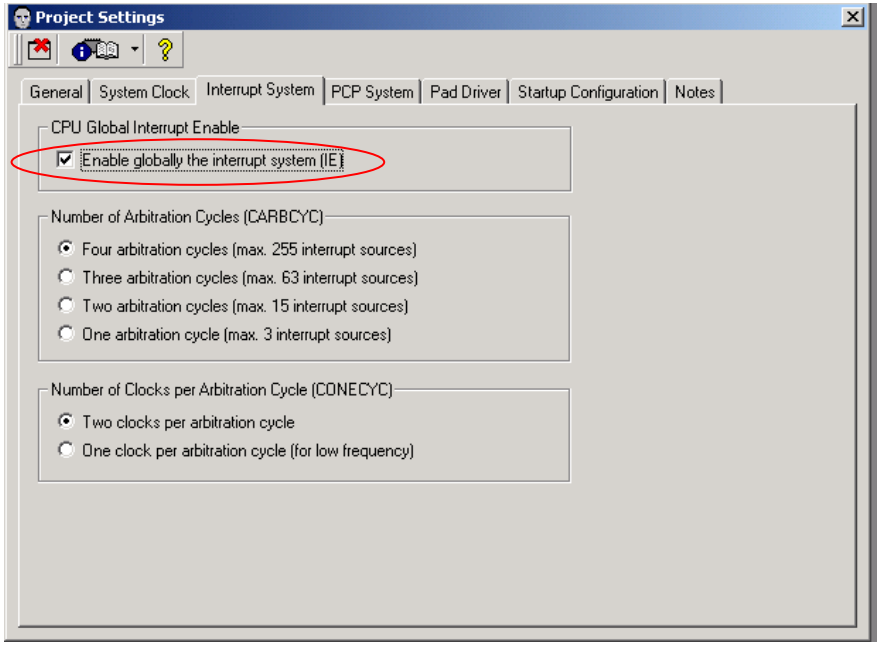
1. General Settings.
- Rename the Main Source File into Main_Local.c.
- Rename the Main Header File into Main_Local.h.
- Select Tasking 2.0.



2. System Clock
- Change external clock frequency to 15 MHz .
- Change input divider PDIV to 2.
- Change VCOSEL to 400MHz – 500 MHz.
- Change feedback divider NDIV to 64.
- Change output divider KDIV to 6.

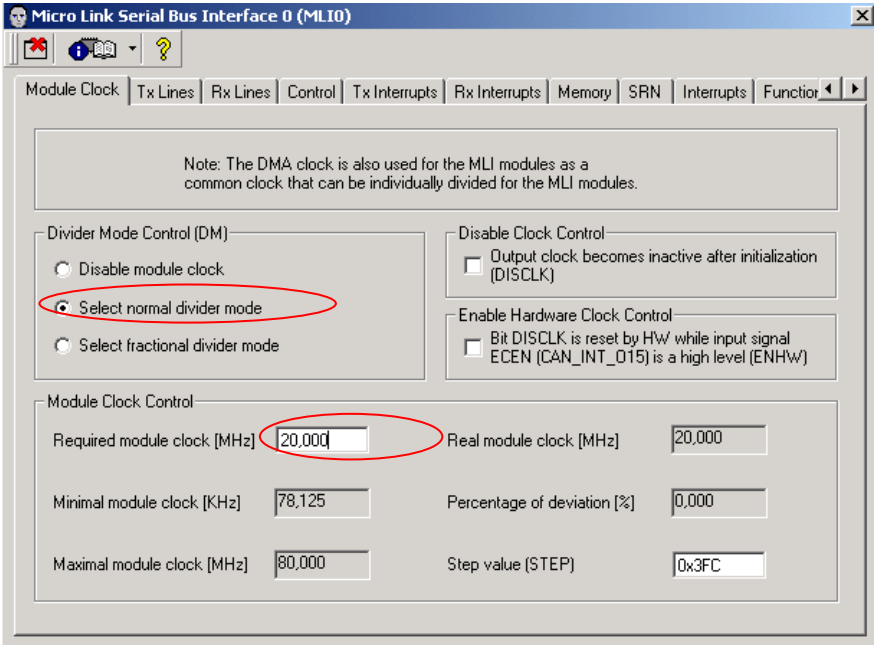


3. Interrupt system
- Enable the Interrupt System globally.

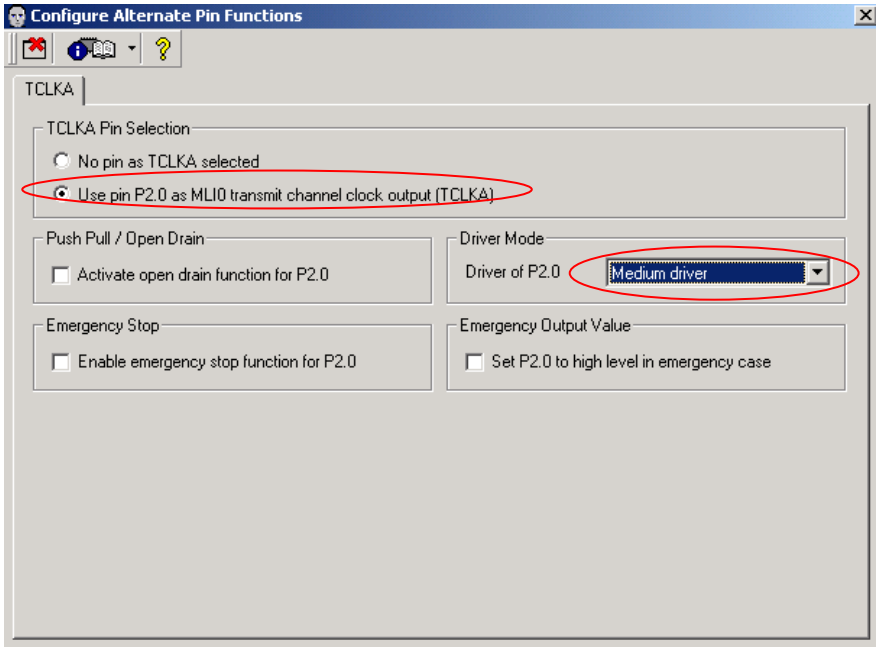


MLI0:

4. Module clock
- Change Divider Mode Control to “Select normal divider mode”.
- Change “Required Module Clock” to 20 MHz.



5. Tx Lines / TCLKA
- TCLKA pin selection: Use pin P2.0.
- Driver Mode: Medium Driver.

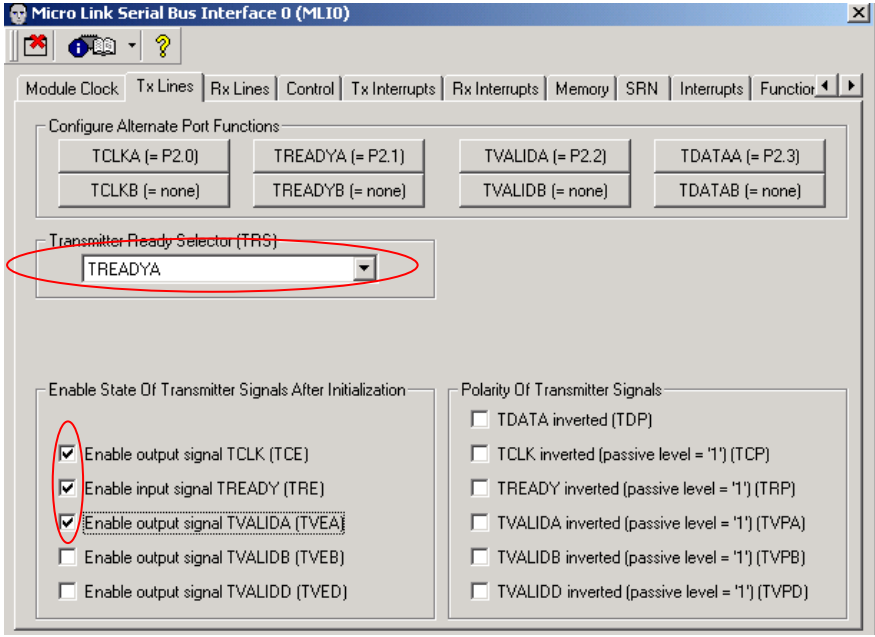


6. Tx Lines / TREADYA
- TREADYA pin selection: Use pin P2.1.
- Driver Mode: Medium Driver.

7. Tx Lines / TVALIDA
- TVALIDA pin selection: Use pin P2.2.

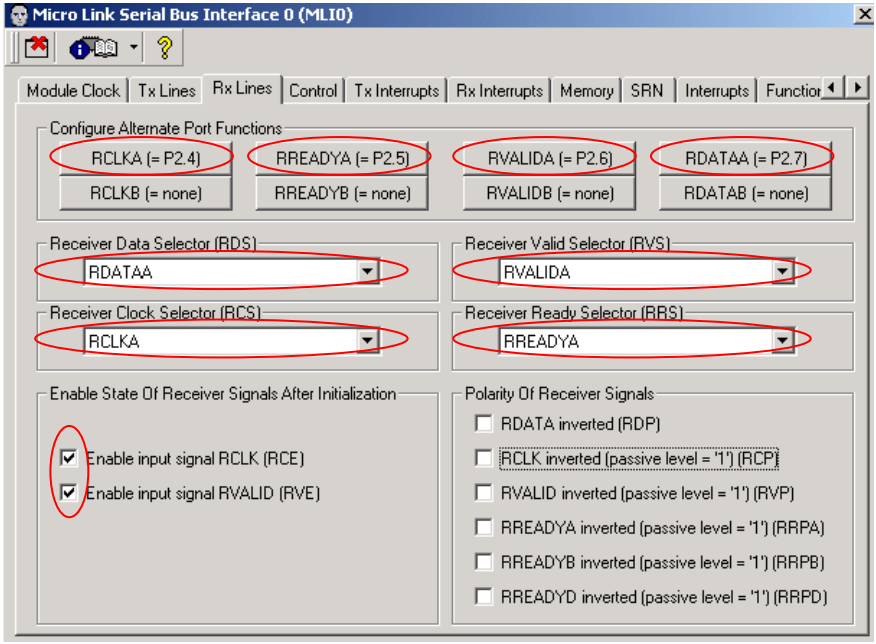
8. Tx Lines / TDATAA
- TDATAA pin selection: Use pin P2.3.

9. Tx Lines
- Change Transmitter Ready Selector to TREADYA.
- Enable output signal TCLK, input signal TREADY, output signal TVALIDA.

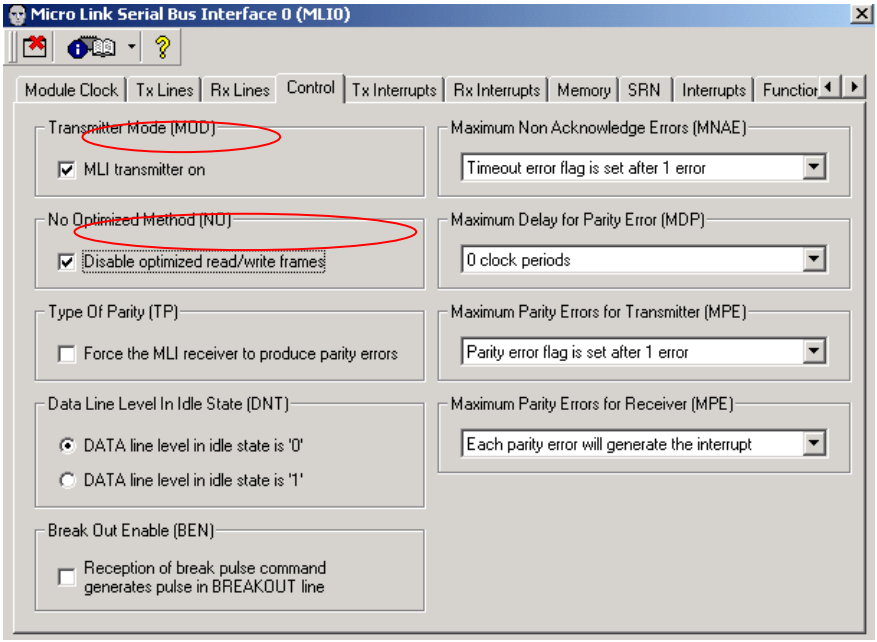


10. Rx Lines

- Configure Alternate Port Function to RCLKA=P2.4, RREADYA=P2.5, RVALIDA=P2.6, RDATA=P2.7.
- Change Receiver Data Selector to RDATAA.
- Change Receiver Clock Selector to RCLKA.
- Change Receiver Ready Selector to RREADYA.
- Change Receiver Valid Selector to RVALIDA.
- Enable input signals RCLK and RVALID.

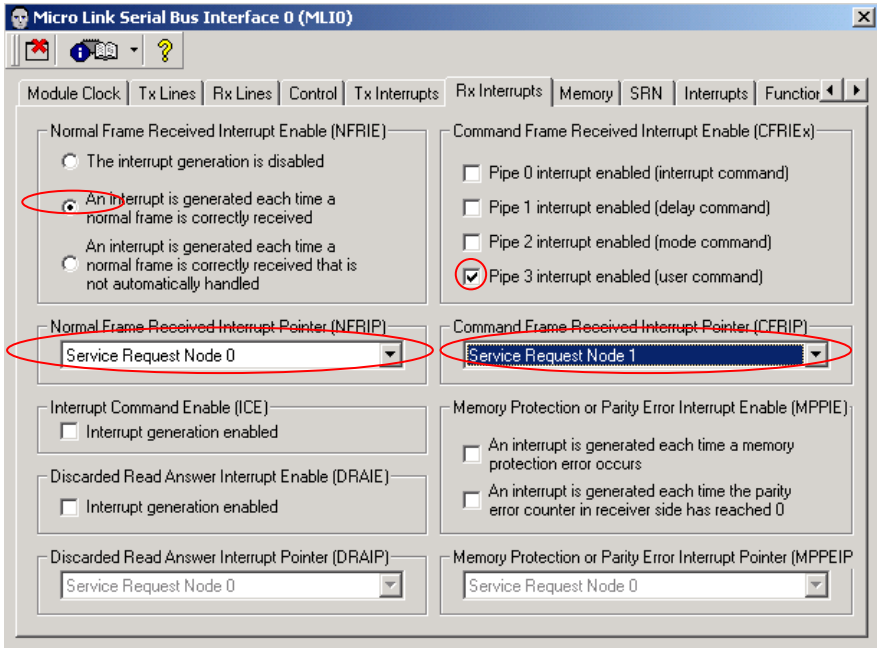


11. Control
- Select MLI Transmitter ON.
- Disable optimized frames.

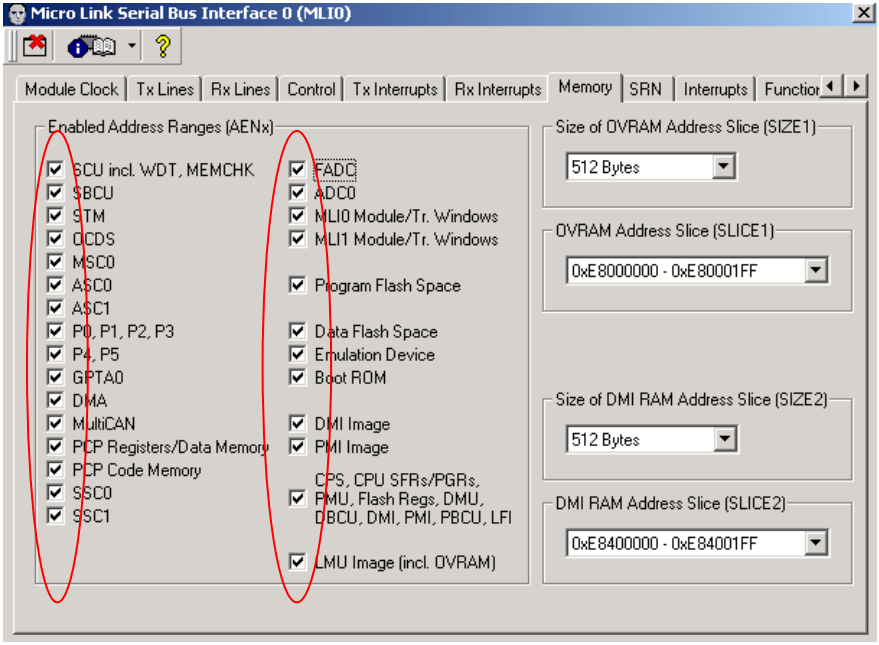


12. Rx Interrupt

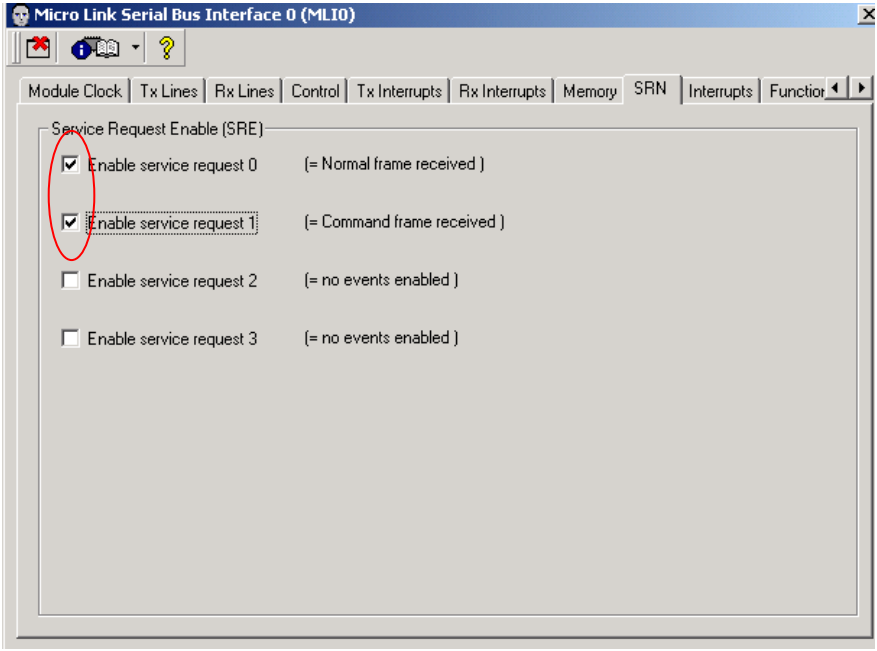
- Normal Frame Received Interrupt Enable: an interrupt is generated each time a normal frame is correctly received.
- Normal Frame Received Interrupt Pointer: select Service Request Node 0.
- Command Frame Received Interrupt Enable: enable interrupts for pipe 3.
- Command Frame Received Interrupt Pointer: select Service Request Node 1.



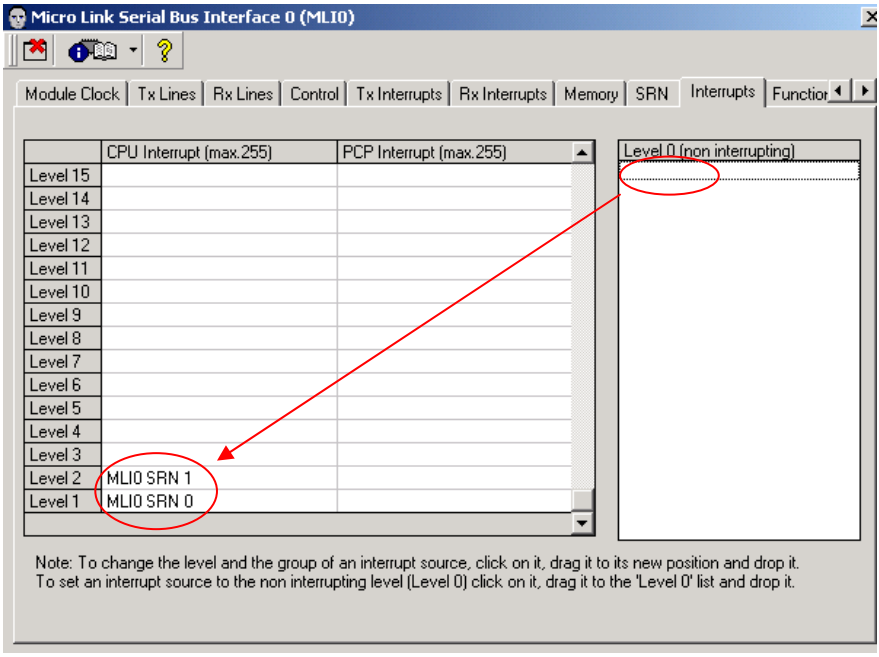
13. Memory
- Enable all address ranges.



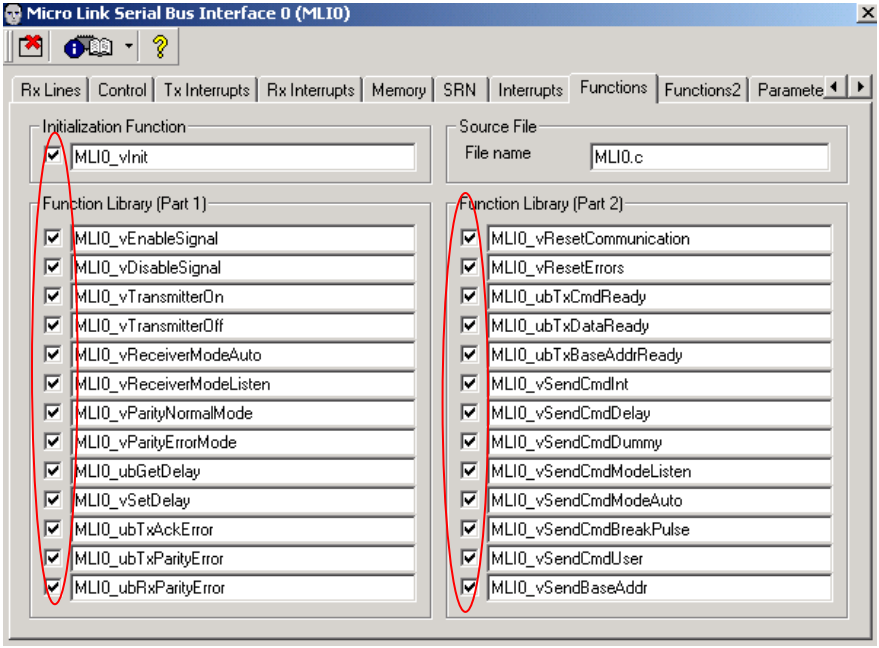
14. SRN
- Enable Service Request 0 and 1



15. Interrupts
- Drag and drop SRN0 to CPU – Level1
- Drag and drop SRN1 to CPU – Level2



16. Functions
- Select all



17. Saving and generating code

Now, the configuration of the Local controller is finished. Create a new folder on your hard drive (for example C:\MLI_TC1766_QuickStart\MLI_Local \) and save there DAvE project (for example local.dav).

Code can now be generated with DAvE. The following files will be created:
- TC1766REGS.H
- MAIN_LOCAL.H
- MAIN_LOCAL.C
- MLI0.H
- MLI0.C

4.2.2 Setting up Tasking environment.

18. Start Tasking v2.2.r1

19. Create a new project space
- 'File' -> 'New Project Space'
- Enter a path (for example C:\MLI_TC1766_QuickStart\) and a name (for example TC1766).

20. Create a new project
- Right - click once on the project space TC1766 (window on the left).
- Select 'Add New Project'.
- Enter a path (for example C:\ MLI_TC1766_QuickStart \MLI_Local) and a name (for example MLI_Local).
- Click OK

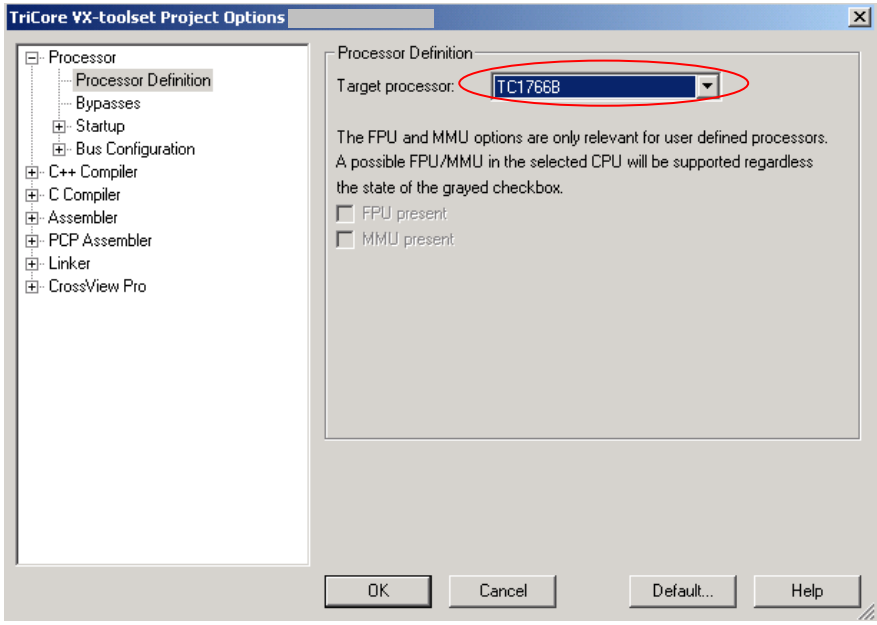
21. Add DAvE generated files to the project
- Right - click once on the project MLI_Local (window on the left).
- Select 'Add existing files'.
- Add TC1766REGS.H, MAIN_LOCAL.H, MAIN_LOCAL.C, MLI0.H, MLI0.C
- Click OK

22. Add two new files to the projects
- Right - click once on the project MLI_Local (window on the left).
- Select 'Add new files'.
- Add "MLIO_Config_Local.c"
- Click OK
- Repeat the previous steps and add "MLIO_Config_Local.h".

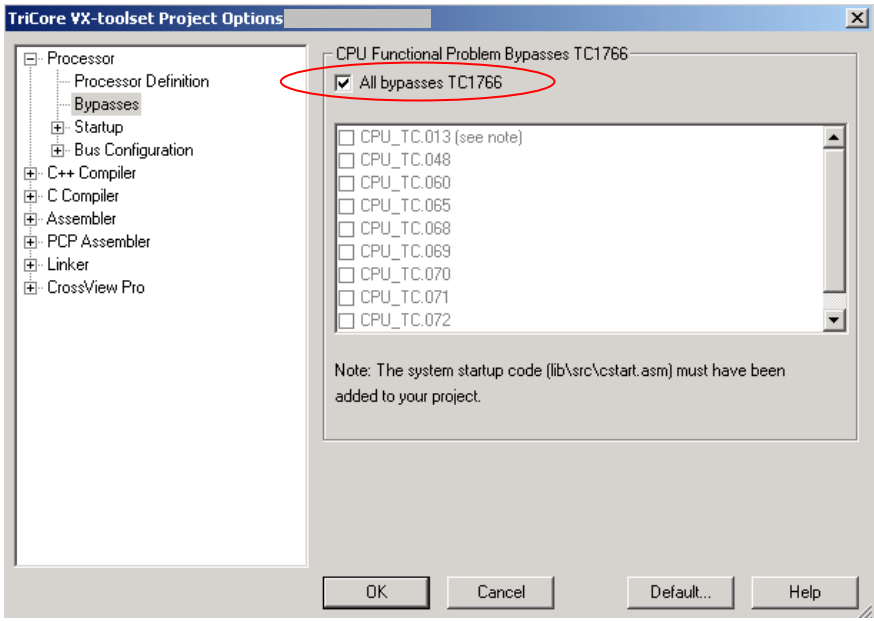
Setting up the project options

23. Open the project option dialog box
- 'Project' -> 'Project Options'

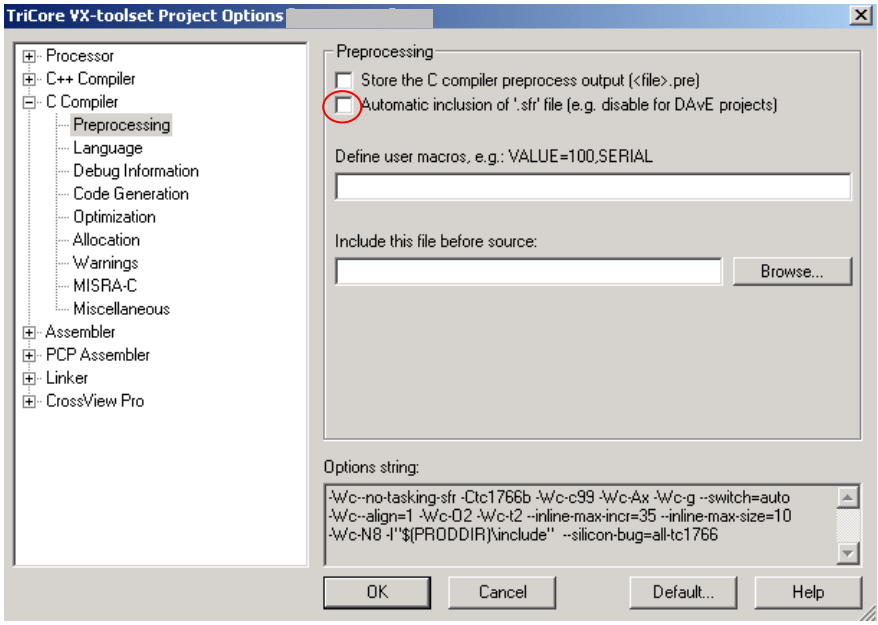
24. Processor -> Processor Definition
- Select TC1766



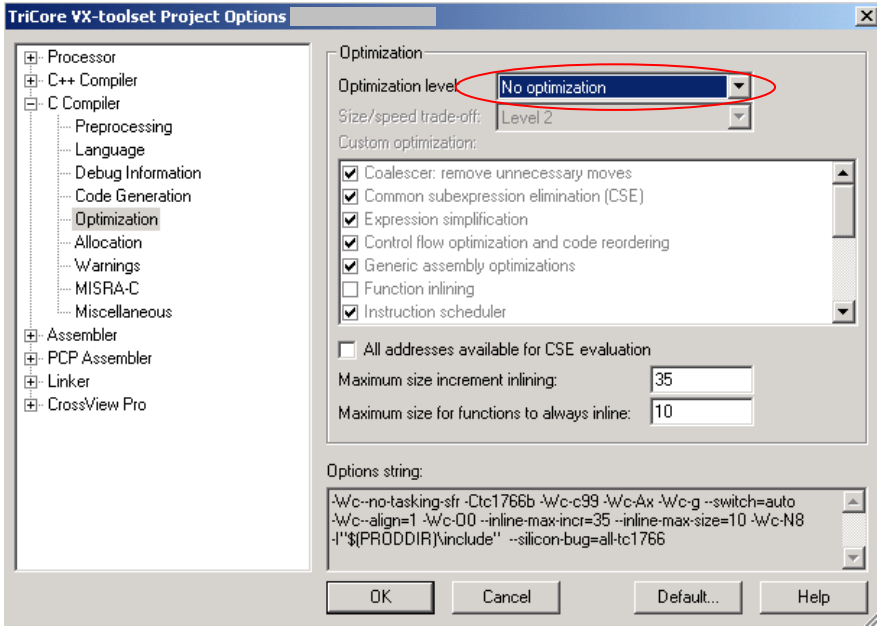
25. Processor -> Bypasses
- Select All bypasses.



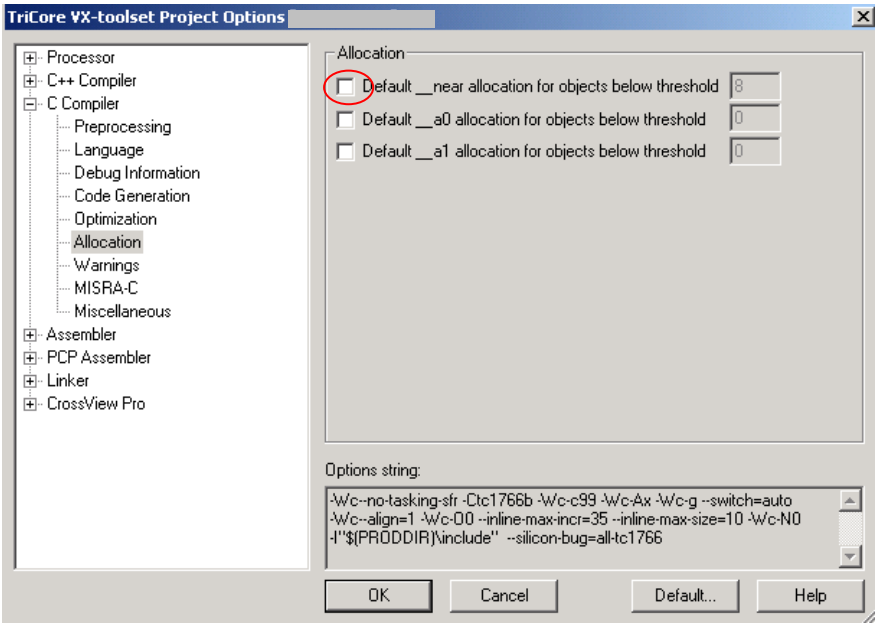
26. C Compiler -> Preprocessing
- Disable automatic inclusion of .sfr files.



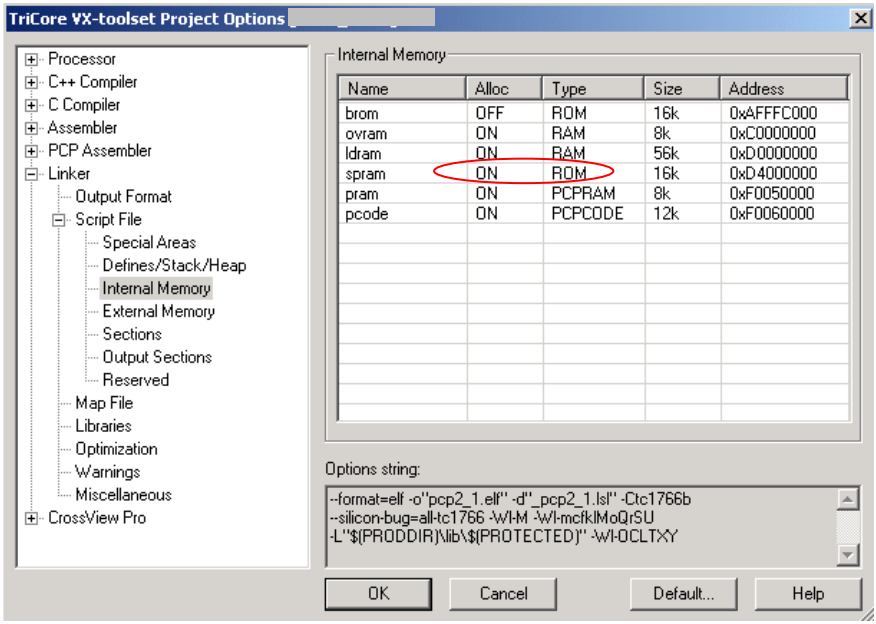
27. C Compiler -> Optimization
- Select no optimization.



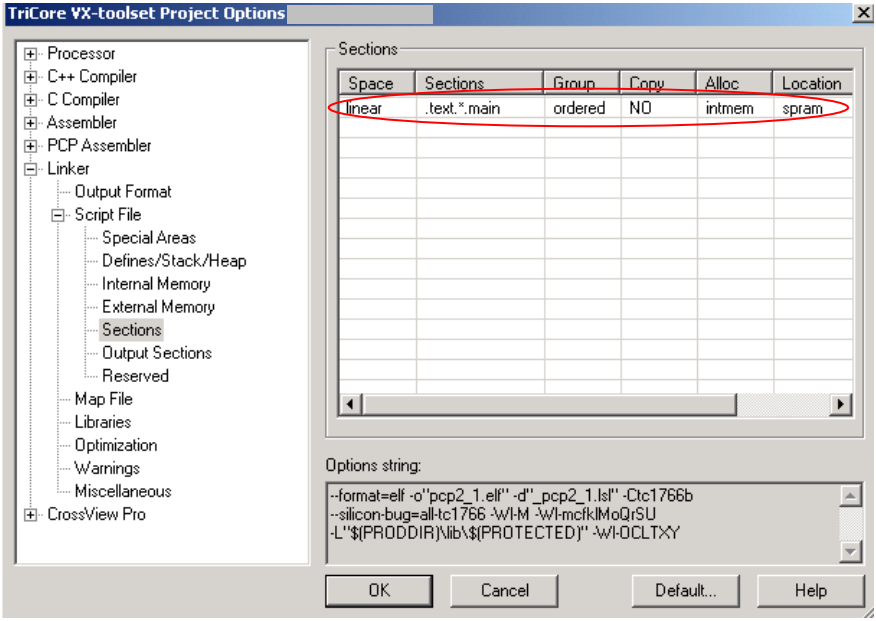
28. C Compiler -> Allocation.
- Disable Default __near allocation.



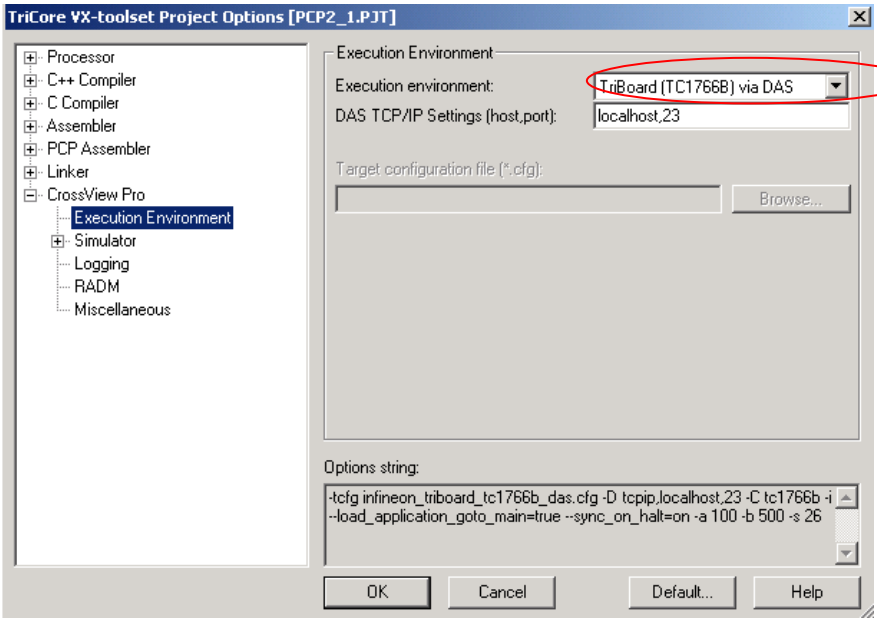
29. Linker -> Script file -> internal memory SPRAM.
- Alloc: select ON.
- Type: select ROM



30. Linker -> Script file -> Sections.
- Space: select linear.
- Sections: type .text.*.main
- Group: select order.
- Copy: select NO.
- Alloc: select intmem
- Location: select spram.



31. CrossView Pro -> Execution environment
- Execution environment: Select TriBoard TC1766 with SRAM.



4.2.3 Programming of the Local controller

In addition to the code automatically generated by DAvE, the following code shall be added. It is recommended to add this code in the dedicated area between the two comments:

```
// USER CODE BEGIN
//add code
// USER CODE END
```

All the files containing this code are attached and can be used directly. Comments are also included there.

4.2.3.1 MLI0.C

Section Imported Global Variables

```
// USER CODE BEGIN (MLI0_General,6)
extern uword volatile *target_read_store;
// USER CODE END
```

void MLI0_vInit(void)

- End of the routine:

```
// USER CODE BEGIN (Init,3)
MLI0_RCR = 0x00010000;
// USER CODE END
```

void INTERRUPT (MLI0_INT0) MLI0_viSRN0(void)

- Beginning of the routine:

```
// USER CODE BEGIN (SRN0,2)
uword volatile *p1;
uword volatile dummy;
// USER CODE END
```

- Case 3:

```
// USER CODE BEGIN (SRN0,133)
p1 = target_read_store;
*p1 = MLI0_RDATAR;
dummy =*p1;
//USER CODE END
```

- End of the routine:

```
// USER CODE BEGIN (SRN0,13)
while(MLI0_RISR & MLI0_RISR_NFRI);
// USER CODE END
```

- void INTERRUPT (MLI0_INT1) MLI0_viSRN1(void)

```
// USER CODE BEGIN (SRN1,17)
Initiate_Transmission();
while(MLI0_RISR & MLI0_RISR_CFRI3!=0);
// USER CODE END
```

4.2.3.2 MLI0_Config_Local.H

```
// Start of file
#define remote_pipe0_base 0xe8408000
#define remote_pipe1_base 0xe840a000
#define remote_pipe2_base 0xe840c000
#define remote_pipe3_base 0xf0000d00

#define MLI0_ubTxAllDataReady() ((ubyte) ((MLI0_TRSTATR & \
 (MLI0_TRSTATR_DV0 | MLI0_TRSTATR_DV1 | MLI0_TRSTATR_DV2 | \
 MLI0_TRSTATR_DV3)) == 0))

#define MLI0_ubTxAllPendingReadReady() ((ubyte) ((MLI0_TRSTATR & \
(MLI0_TRSTATR_RP0 | MLI0_TRSTATR_RP1 | MLI0_TRSTATR_RP2 | \
MLI0_TRSTATR_RP3)) == 0))

#define wait_bf        while(MLI0_ubTxBaseAddrReady()==0)
#define wait_bf_neg    while(MLI0_ubTxBaseAddrReady()!=0)
#define wait_cf        while(MLI0_ubTxCmdReady()==0)
#define wait_cf_neg    while(MLI0_ubTxCmdReady()!=0)
#define wait_df        while(MLI0_ubTxAllDataReady()==0)
#define wait_df_neg    while(MLI0_ubTxAllDataReady()!=0)
#define wait_rf        while(MLI0_ubTxAllPendingReadReady()==0)
#define wait_rf_neg    while(MLI0_ubTxAllPendingReadReady()!=0)
```

```
#define NOP             __asm("nop \n")
#define wait_states 10000

void MLIO_config_local_n_standby(void);
void MLIO_startup_procedure(void);
void Initiate_Transmission(void);
void wait(int i);
// End of file
```

4.2.3.3 MLI0_Config_Local.C

```
// Start of file
#include "MAIN_Local.h"

extern uword volatile MLI_Remote_Ready;
uword volatile pe1_flag, pe2_flag;



void MLIO_config_local_n_standby(void)
{
MLIO_startup_procedure();

MLI0_RIER = 0x00000020;
DMA_MLI0SRC1 = 0x00004002;
DMA_MLI0SRC1 = 0x00001002;

MLI0_vSendBaseAddr(0, remote_pipe0_base, 12);
wait_bf_neg;
wait_bf;
MLI0_vSendBaseAddr(1, remote_pipe1_base, 12);
wait_bf_neg;
wait_bf;
MLI0_vSendBaseAddr(2, remote_pipe2_base, 12);
wait_bf_neg;
wait_bf;
MLI0_vSendBaseAddr(3, remote_pipe3_base, 12);
wait_bf_neg;
wait_bf;
MLI0_vSendCmdUser(0);
wait_cf_neg;
wait_cf;
}
```

```
void Initiate_Transmission(void)
{

MLI0_TCR = MLI0_TCR & ~MLI0_TCR_NO;
MLI0_vResetErrors();
MLI0_TIER = MLI0_TIER | 0x03FF0000;
while(MLI0_TISR!=0);
MLI0_RIER = MLI0_RIER | 0x03FF0000;
while(MLI0_RISR!=0);
MLI0_RIER    =  0x00000021;
DMA_MLI0SRC0 = 0x00004001;
DMA_MLI0SRC0 = 0x00001001;
MLI_Remote_Ready=0x00000001;
}



void MLI0_startup_procedure(void)
{

int k=0;
unsigned int line_delay;
line_delay = 0;
pe1_flag =0;
pe2_flag =0x00000001;
MLI0_TCR |= MLI0_TCR_RTY ;
MLI0_vResetCommunication();
MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
MLI0_vResetCommunication();
line_delay = MLI0_ubGetDelay()+1;
if (line_delay < 0xC)
       MLI0_vSetDelay(line_delay);
else
       NOP;
MLI0_vSendCmdDelay(line_delay+3);
MLI0_SCR = MLI0_SCR_CCV1;
wait(wait_states);

MLI0_SCR = MLI0_SCR_CTPE | MLI0_SCR_CNAE;
while((MLI0_TSTATR & MLI0_TSTATR_NAE) & (MLI0_TSTATR & MLI0_TSTATR_PE)
!=0);

MLI0_vParityErrorMode();
```

```
MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
pe1_flag= (MLI0_TSTATR & MLI0_TSTATR_PE);
if (pe1_flag==0)
   NOP;

MLI0_vParityNormalMode();
MLI0_SCR = MLI0_SCR_CTPE | MLI0_SCR_CNAE;
while((MLI0_TSTATR & MLI0_TSTATR_NAE) & (MLI0_TSTATR & MLI0_TSTATR_PE)
!=0);

MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
pe2_flag= (MLI0_TSTATR & MLI0_TSTATR_PE) | (MLI0_TSTATR &
MLI0_TSTATR_NAE);
if (pe2_flag!=0)
   NOP;
}



void wait(int i)
{
int j;
for (j=0; j<=i;j++)
        {
        NOP;
        }
}

//End of file
```

4.2.3.4 Main_Local.H

At the end of the file

```
// USER CODE BEGIN (MAIN_Header,10)
#include "MLI0_Config_Local.h"
// USER CODE END
```

4.2.3.5 Main_Local.C

Section Global Variables

```
// USER CODE BEGIN (MAIN_General,7)
uword volatile *target_write, *target_read, *target_read_store;
uword volatile MLI_Remote_Ready;
// USER CODE END
```

Main function

```
// USER CODE BEGIN (Main,9)
int i;
uword volatile xread;

DMA_MLI0SRC0 = 0x00000001;
DMA_MLI0SRC1 = 0x00000002;

MLI_Remote_Ready=0;

MLIO_config_local_n_standby();

while(MLI_Remote_Ready==0);

target_write = MLI0_SWIN3+0xd00;
*target_write = 0x0000000;
wait_df_neg;
wait_df;

target_write = MLI0_SWIN1;
for(i=0;i<=5;i++)
{
*target_write=0xaaaa0000+i;
```

```
wait_df_neg;
wait_df;
target_write=target_write+1;
}

target_read = MLI0_SWIN0;
target_read_store = 0xd0006000;
for(i=0;i<=5;i++)
{
xread = *target_read;
wait_rf_neg;
wait_rf;
target_read= target_read+1;
target_read_store = target_read_store+1;

 }

while(1);

// USER CODE END
```

4.3 Setting up MC2

All the operations below shall be performed on the PC connected to MC2.

4.3.1 Configuration of the remote controller

The remote controller MC2 can be configured using DAvE (v2.1). Most of the actions below are similar to the ones described in section 4.2.

Open a new project for TC1766.

Project settings:

1. General Settings
- Rename the Main Source File into Main_Remote.c.
- Rename the Main Header File into Main_Remote.h.
- Select Tasking 2.0.

2. System Clock
- Change external clock frequency to 15 MHz .
- Change input divider PDIV to 2.
- Change VCOSEL to 400MHz – 500 MHz.
- Change feedback divider NDIV to 64.
- Change output divider KDIV to 6.

3. Interrupt system
- Enable the Interrupt System globally.

MLI0:

4. Module clock
- Change Divider Mode Control to “Select normal divider mode”.
- Change “Required Module Clock” to 20MHz.

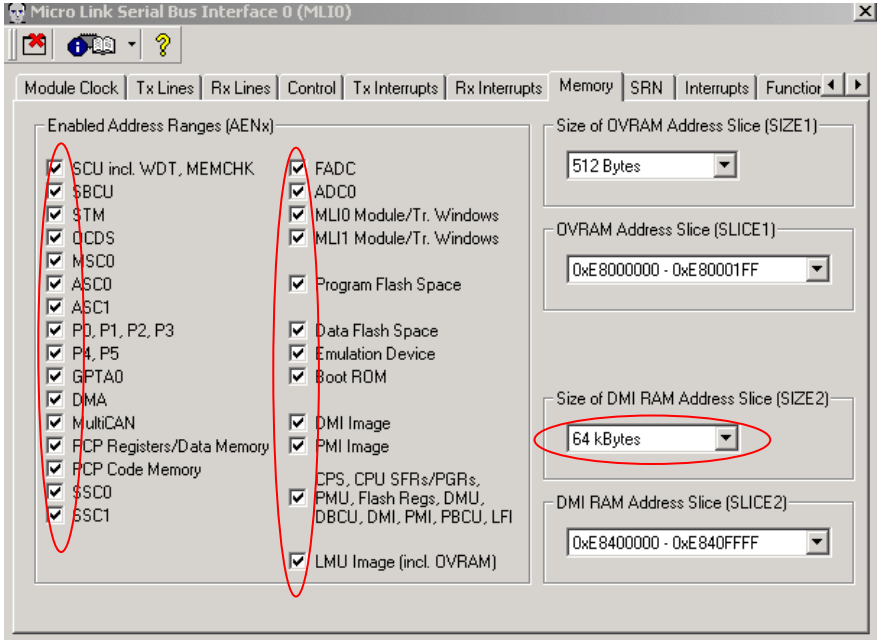
5. Tx Lines
- Configure Alternate Port Function to TCLKA=P2.0, TREADYA=P2.1, TVALIDA=P2.2, TDATA=P2.3. (Select medium driver)
- Change Transmitter Ready Selector to TREADYA.
- Enable output signal TCLK, input signal TREADY, output signal TVALIDA.

6. Rx Lines
- Configure Alternate Port Function to RCLKA=2.4, RREADYA=P2.5, RVALIDA=P2.6, RDATA=P2.7.
- Change Receiver Data Selector to RDATAA.
- Change Receiver Clock Selector to RCLKA.
- Change Receiver Ready Selector to RREADYA.
- Change Receiver Valid Selector to RVALIDA.
- Enable input signal RCLK and RVALID.

7. Control
- Select MLI Transmitter ON
- Disable optimized frames

8. Rx Interrupt
- Command Frame Received Interrupt Enable: enable interrupts for pipe 3.
- Command Frame Received Interrupt Pointer: select Service Request Node 1.

9. Memory
- Enable all address ranges.
- Size of DMI RAM Address Slice: 64 kBytes.



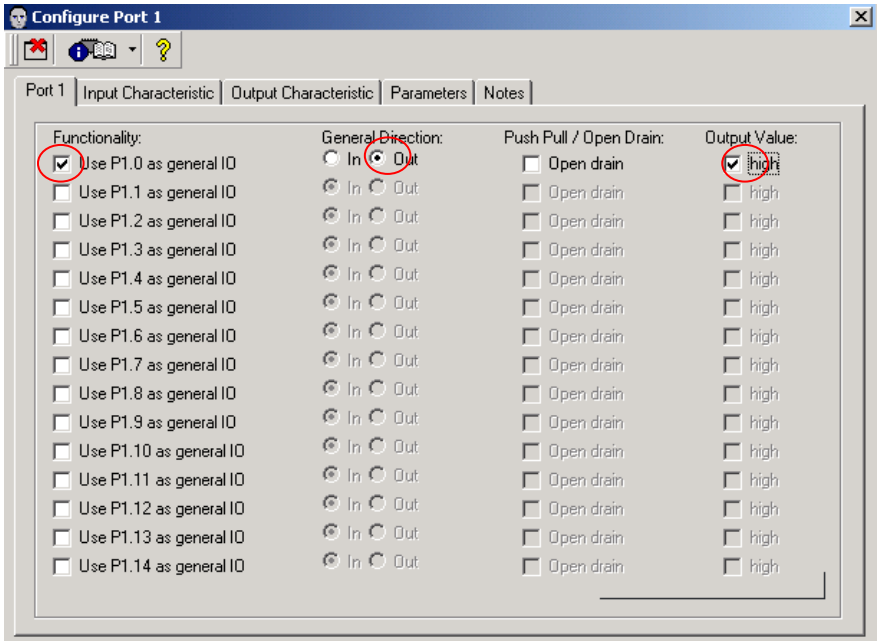
10. SRN
- Enable Service Request 1

11. Interrupts
- Drag and drop SRN1 to CPU – Level2

12. Functions
- Select all

PORT:

13. Ports -> Configure Port 1
- Use P1.0 as general IO
- General Description: Out
- Output value : high



14. Functions
- Select IO_vInit.

15. Saving and generating code

Now, the configuration of the Remote controller is finished. Create a new folder on your hard drive (for example C:\MLI_TC1766_QuickStart\MLI_Remote\) and save there DAvE project (for example MLI_Remote.dav).

Code can now be generated with DAvE. The following files will be created:
- MAIN_REMOTE.H
- MAIN_REMOTE.C

- MLI0.H
- MLI0.C
- IO.C
- IO.H

4.3.2 Setting up Tasking environment

16. Start Tasking v2.2.r1

17. Create a new project space
- 'File' -> 'New Project Space'
- Enter a path (for example C:\MLI_TC1766_QuickStart\) and a name (for example TC1766).

18. Create a new project
- Right - click once on the project space TC1766 (window on the left).
- Select 'Add New project'.
- Enter a path (for example C:\MLI_TC1766_QuickStart\MLI_Remote) and a name (for example MLI_Remote).
- Click OK

19. Add DAvE generated files to the project
- Right - click once on the project MLI_Remote (window on the left).
- Select 'Add existing files'.
- Add TC1766REGS.H, MAIN_Remote.H, MAIN_Remote.C, MLI0.H, MLI0.C, IO.C, IO.H.
- Click OK

20. Add two new files to the projects
- Right - click once on the project MLI_Remote (window on the left).
- Select 'Add new files'.
- Add "MLI0_Config_Remote.c"
- Click OK
- Repeat the previous steps and add "MLI0_Config_Remote.h"

Setting up the project options.

21. Repeat the same steps as for the Local controller

4.3.3 Programming of the Remote controller

In addition to the code automatically generated by DAvE, the following code shall be added. It is recommended to add this code in the dedicated area between the two comments:

```
// USER CODE BEGIN
//add code
// USER CODE BEGIN
```

All the files containing this code are attached and can be used directly. Comments are also included there.

4.3.3.1 MLI0.C

void MLI0_vInit(void)

- End of the routine:

```
// USER CODE BEGIN (Init,3)
MLI0_RCR = 0x00010000;
// USER CODE END
```

void INTERRUPT (MLI0_INT1) MLI0_viSRN1(void)

```
// USER CODE BEGIN (SRN1,17)
MLI0_config_remote_n_ready();
while(MLI0_RISR & MLI0_RISR_CFRI3 != 0);
// USER CODE END
```

4.3.3.2 MLI0_Config_Remote.H

```
// Start of file
#define remote_pipe0_base     0xd0008000

#define wait_cf         while(MLI0_ubTxCmdReady()==0)
#define wait_cf_neg     while(MLI0_ubTxCmdReady()!=0)
#define NOP             __asm("nop \n")

#define wait_states     10000

void MLI0_startup_procedure(void);
void MLI0_config_remote_n_ready(void);
```

```
void wait(int);
// End of file
```

4.3.3.3 MLI0_Config_Remote.C

```
// Start of file
#include "MAIN_Remote.h"
uword volatile pe1_flag, pe2_flag;



void MLI0_config_remote_n_ready(void)
{

MLI0_startup_procedure();
MLI0_vResetErrors();

MLI0_RIER = MLI0_RIER | 0x03ff0000;
while(MLI0_RISR!= 0);

MLI0_SCR = MLI0_SCR | MLI0_SCR_SMOD;

MLI0_vSendCmdUser(0);
wait_cf_neg;
wait_cf;
MLI0_TIER = MLI1_TIER | 0x03FF0000;

while(MLI0_TISR!= 0);
}



void MLI0_startup_procedure(void)
{
int k=0;
unsigned int line_delay;
line_delay = 0;
pe1_flag =0;
pe2_flag =0x00000001;

MLI0_TCR |= MLI0_TCR_RTY ;
MLI0_vResetCommunication();
MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
```

```
MLI0_vResetCommunication();
line_delay = MLI0_ubGetDelay()+1;
if (line_delay < 0xC)
        MLI0_vSetDelay(line_delay);
else
        NOP;
MLI0_vSendCmdDelay(line_delay+3);
MLI0_SCR = MLI0_SCR_CCV1;
wait(wait_states);
MLI0_SCR = MLI0_SCR_CTPE | MLI0_SCR_CNAE;
while((MLI0_TSTATR & MLI0_TSTATR_NAE) & (MLI0_TSTATR & MLI0_TSTATR_PE)
!=0);

MLI0_vParityErrorMode();
MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
pe1_flag= (MLI0_TSTATR & MLI0_TSTATR_PE);
if (pe1_flag==0)
   NOP;

MLI0_vParityNormalMode();
MLI0_SCR = MLI0_SCR_CTPE | MLI0_SCR_CNAE;
while((MLI0_TSTATR & MLI0_TSTATR_NAE) & (MLI0_TSTATR & MLI0_TSTATR_PE)
!=0);
MLI0_vSendCmdInt(0);
MLI0_SCR = MLI0_SCR_CCV0;
wait(wait_states);
pe2_flag= (MLI0_TSTATR & MLI0_TSTATR_PE) | (MLI0_TSTATR &
MLI0_TSTATR_NAE);
if (pe2_flag!=0)
   NOP;
}



void wait(int i)
{
int j;
for (j=0; j<=i;j++)    // wait until MDCstops
        {
        NOP;
        }
}//End of file
```

4.3.3.4 Main_Remote.H

At the end of the file

```
// USER CODE BEGIN (MAIN_Header,10)
#include "MLI0_Config_Remote.h"
// USER CODE END
```

4.3.3.5 Main_Remote.C

Main function

```
// USER CODE BEGIN (Main,9)
int i;
uword volatile *p;
p= remote_pipe0_base;

for(i=0;i<=5;i++)
{
*p=0xffff0000 +i;
p=p + 1;
}
while(1);
// USER CODE END
```

4.4 Running the applications

Once both programs below have been compiled successfully, 2 distinct debugger sessions (one for MC1, the other for MC2) can be started. The two programs can now be downloaded to MC1 and MC2.

The application in MC2 should be started first, then the application of MC1.

The following should be observed:

- The LED of the TriBoard of MC2 switches on.
- The values 0xaaaa0000, 0xaaaa0001,…, 0xaaaa0005 (which are defined in the main function of the application of MC1) can be read on MC2 at the following memory locations: 0xd000a000, 0xd000a004,…, 0xd000a014.
- The values 0xffff0000, 0xffff0001,…, 0xffff0005 (which are defined in the main function of the application of MC2) can be read on MC1 at the following memory locations: d0006000, 0xd0006004,…, 0xd0006014.

5 Ready-to-use files

The files attached with this application note can directly be used to run the application described in this document.

The attached .zip file contains especially the following files:

Folder MLI_local

- Local.dav (DAvE file)
- options.opt (project option file for Tasking v2.2)
- Main_Local.h
- Main_Local.c
- MLI0.h
- MLI0.c
- MLI0_Config_Local.h
- MLI0_Config_Local.c
- TC1766Regs.h

Folder MLI_Remote

- Remote.dav (DAvE file)
- options.opt (project option file for Tasking v2.2)
- Main_Remote.h
- Main_Remote.c
- MLI0.h
- MLI0.c
- MLI0_Config_Remote.h
- MLI0_Config_Remote.c
- IO.c
- IO.h
- TC1766Regs.h

http://www.infineon.com

Published by Infineon Technologies AG