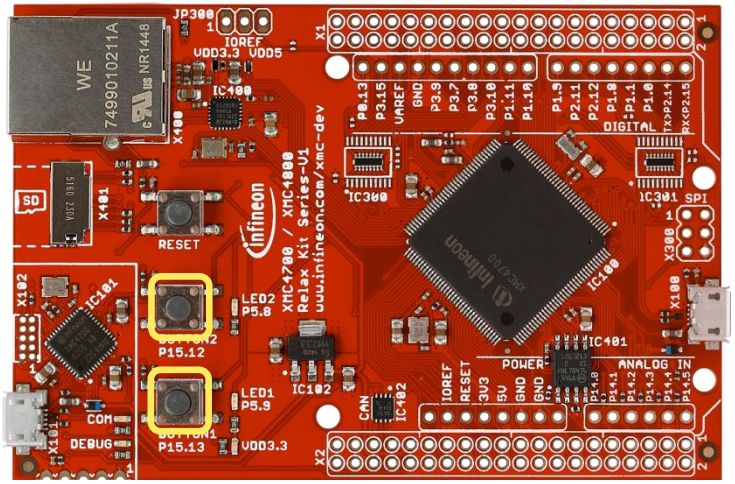
XMC4700 / 4800 MODBUS® TCP Client Example

Getting Started V1.0

1. Overview and Requirements
2. Setup
3. Implementing the application
4. How to test

1. Overview and Requirements
2. Setup
3. Implementing the application
4. How to test

Overview



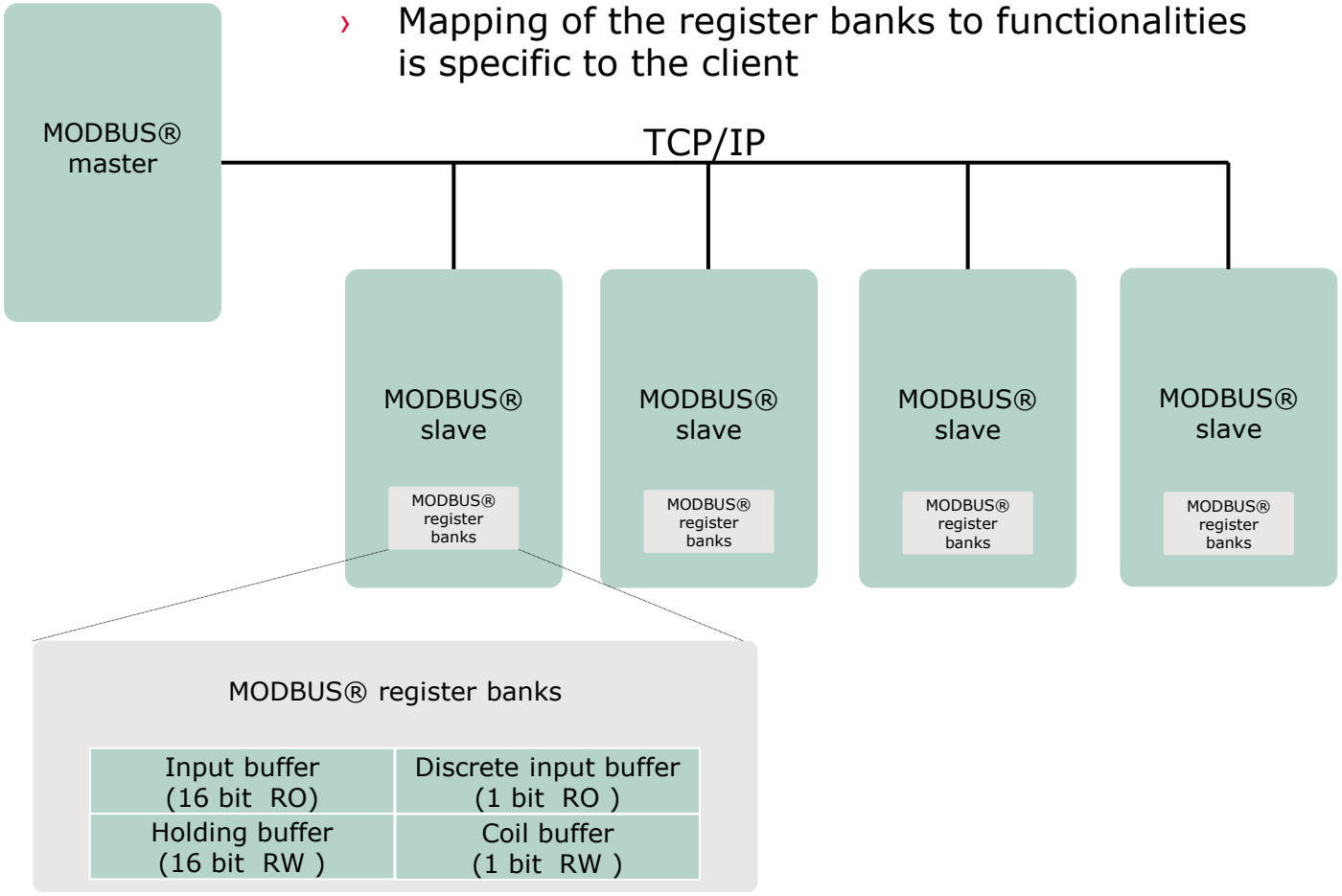
This example demonstrates the implementation of a TCP/IP Modbus® client. It is based on the FreeMODBUS implementation which is ported to XMC4000 family and provided within this example.

Within this documentation you will be guided through all the building blocks of a typical Modbus® application on the XMC4000 family. You will see the input buttons of the board mapped to binary Modbus® discrete input registers and learn to poll their status on your laptop which is used as a Modbus® host.

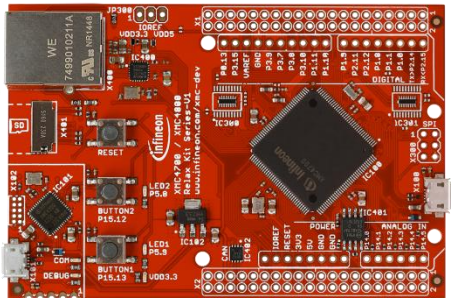
As a result you will be enabled to implement your own Modbus® client on the XMC4000 family.

Overview – ModBUS®

- Master can address each client individually
- Master has access to the register bank inside each client
- MODBUS® protocol defines 4 type of registers
- Mapping of the register banks to functionalities is specific to the client



Requirements – hardware



XMC 4700 Relax Kit

or optional

XMC4800 Relax EtherCAT Kit



RJ45 Ethernet Cable



Windows Laptop installed

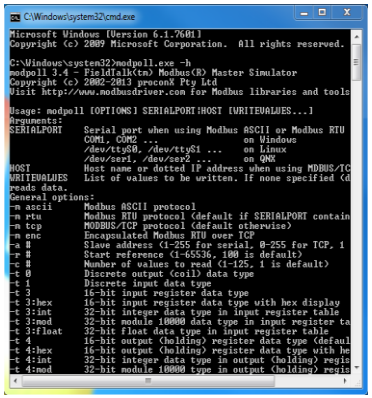
- DAVE v4 (Version4.1.4 or higher)
- Modpoll command line tool



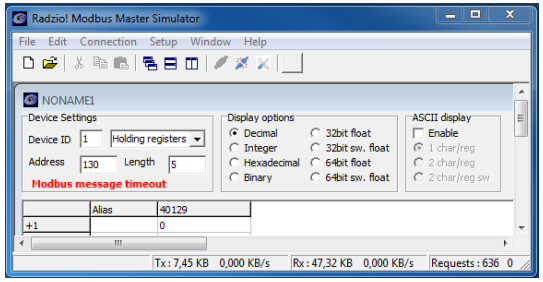
Micro USB Cable (Debugger and connector)

Requirements - free software downloads

DAVE™ (v4.1.4 or higher)
Link: Download DAVE (Version 4)



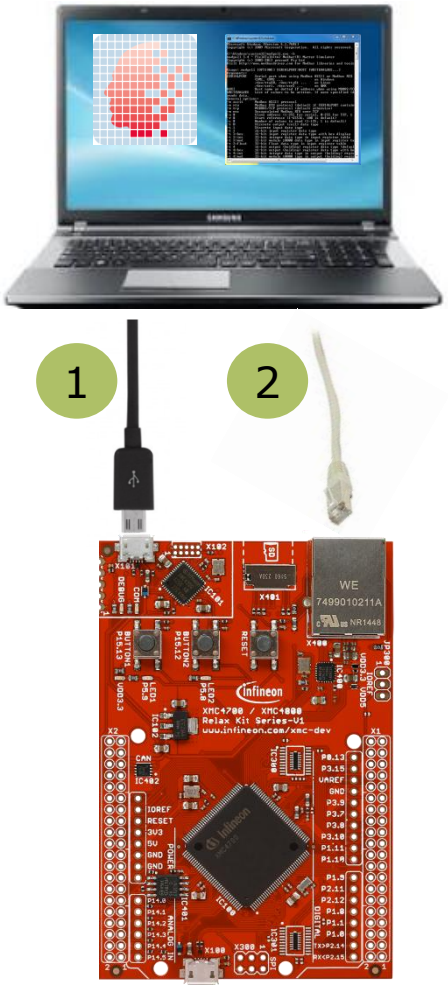
Modpoll Modbus Master Simulator
Link: modpoll.exe command line tool



Optional:
Radzio! Modbus Master Simulator
Link: Download Radzio!

1. Overview and Requirements
2. Setup
3. Implementing the application
4. How to test

Setup – Hardware



1 Micro USB cable connected to X101 debug connector

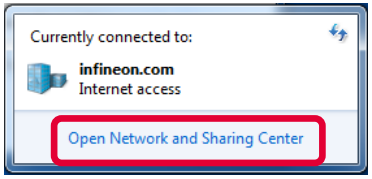
2 RJ45 Ethernet cable to connect XMC4700 Relax Kit with laptop

Setup – LAN configuration of laptop (1/2)

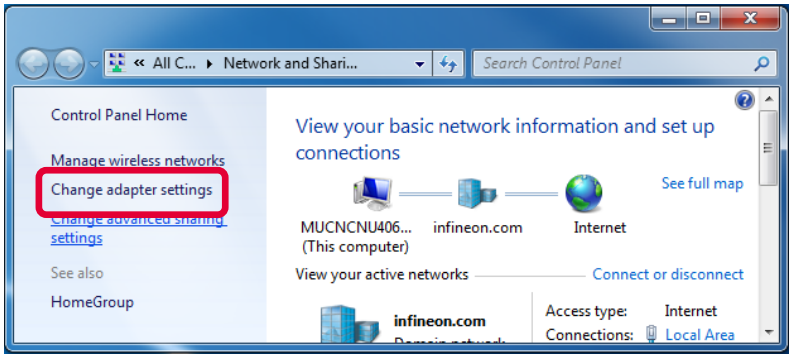
1



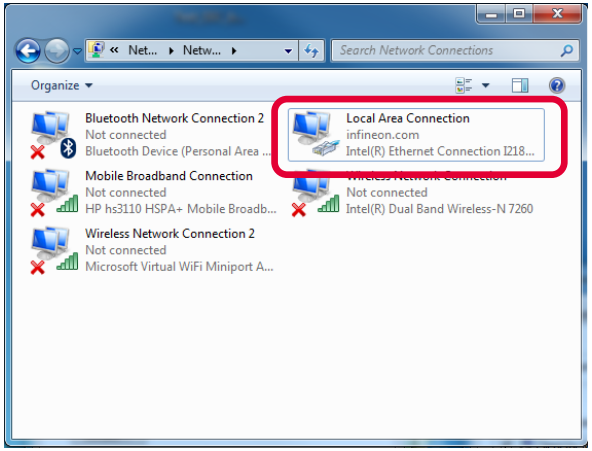
2



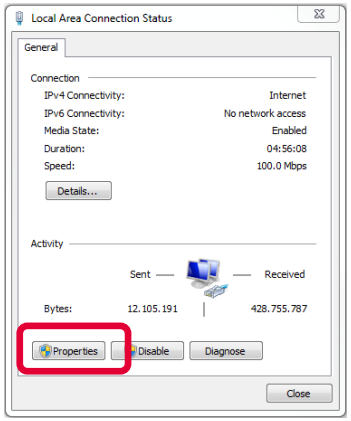
3



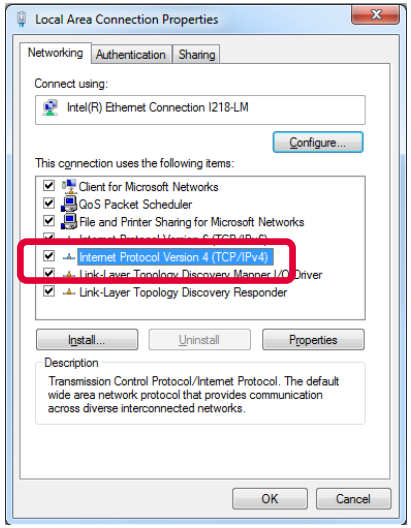
4



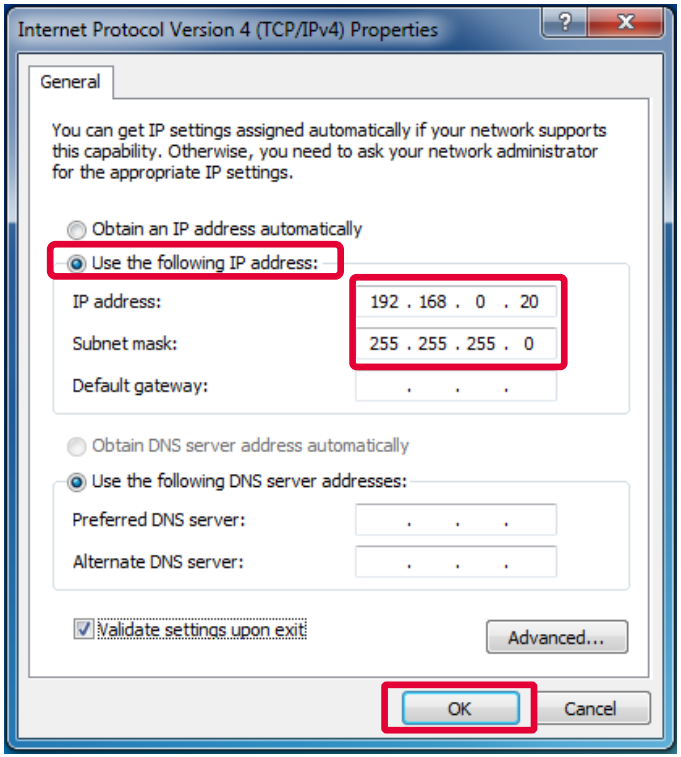
5



6



Setup – LAN configuration of laptop (2/2)



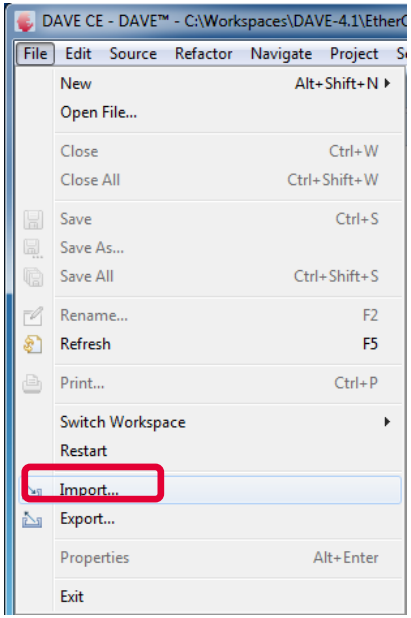
7 Select option „Use the following IP address“

8 Set IP address to 192.168.0.20 and the subnet mask to 255.255.255.0

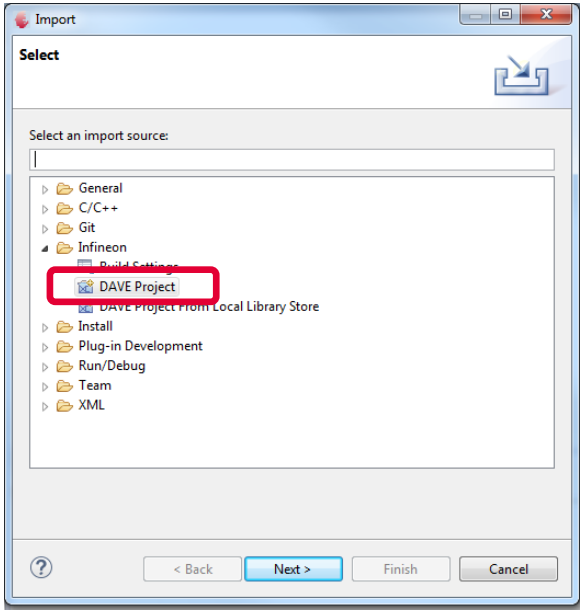
9 Confirm with OK

Setup – Import example project into DAVE™

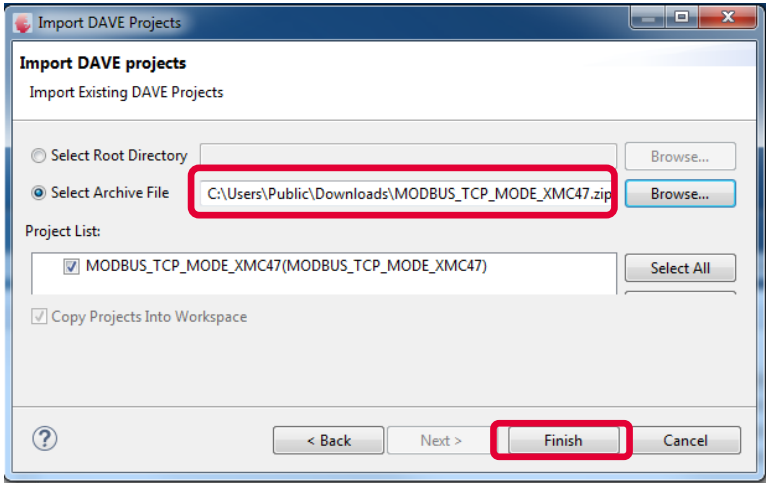
1



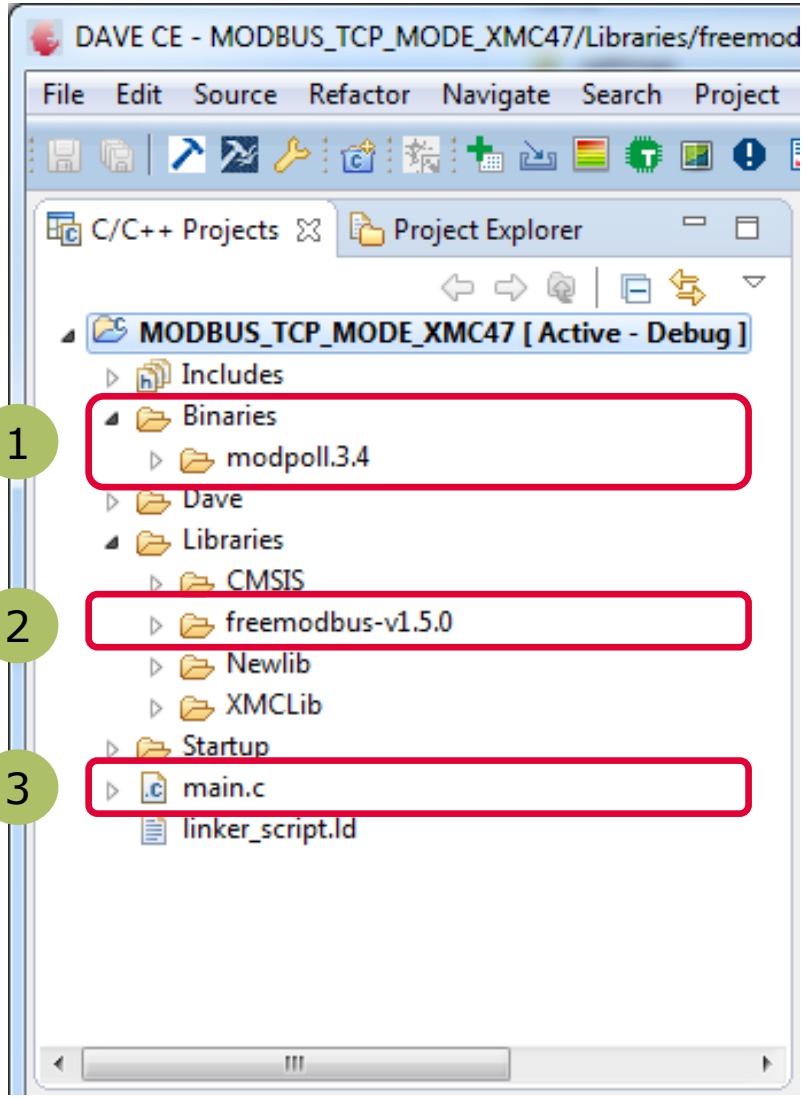
2



3



Setup – Import example project into DAVE™



Check the folder structure of the imported project.
Beside standard DAVE project items you will find these Modbus® related items:

1. Modpoll Modbus Master Simulator. A command line tool to test this example using your laptop as a master
2. The FreeMODBUS library including the port folder for XMC4000 family
3. The main file implementing the example application

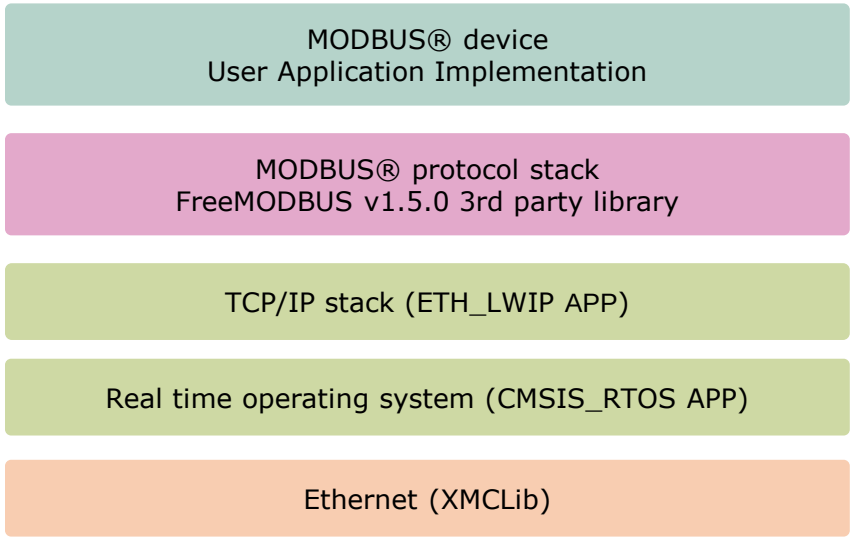
1 Overview and Requirements

2 Setup

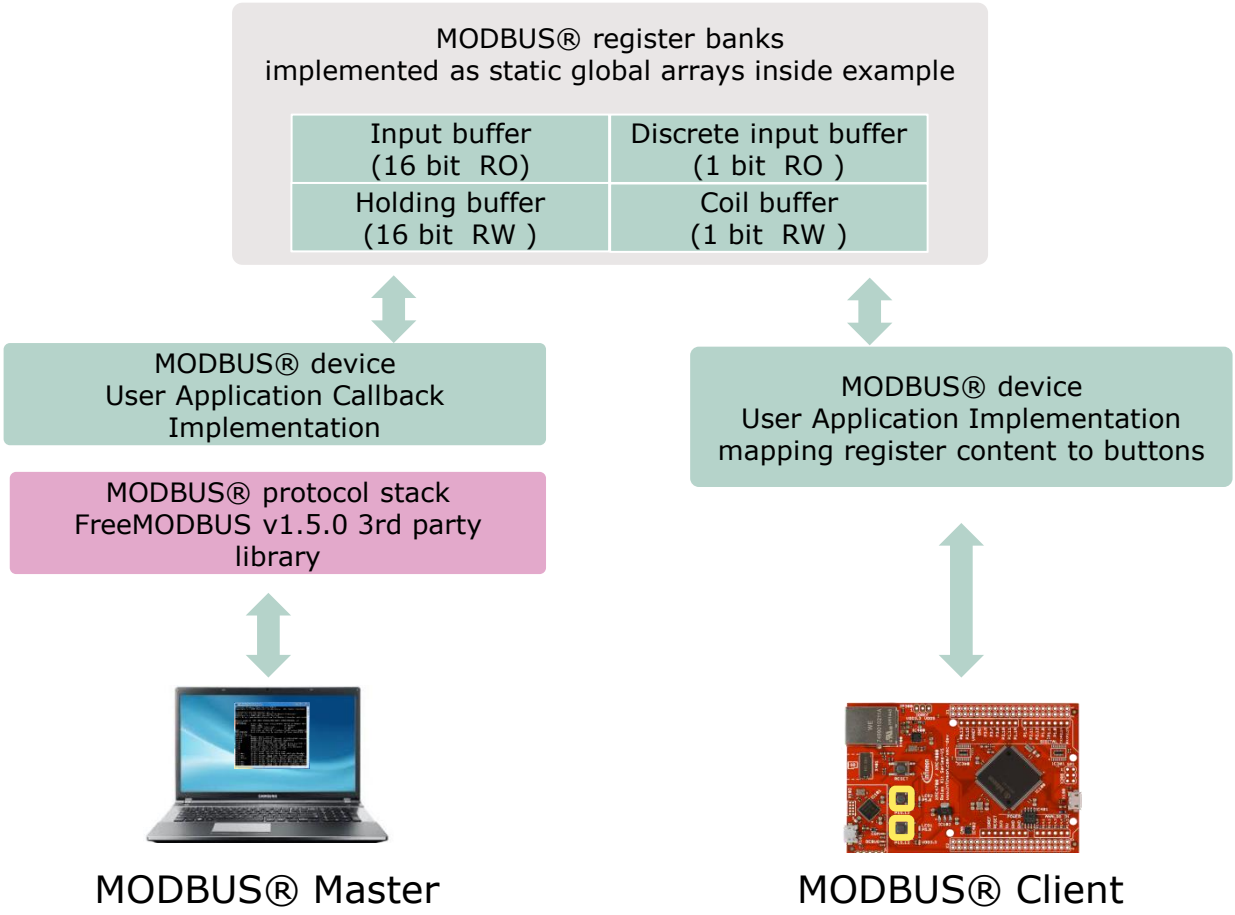
3 Implementing the application

4 How to test

Application – Architecture



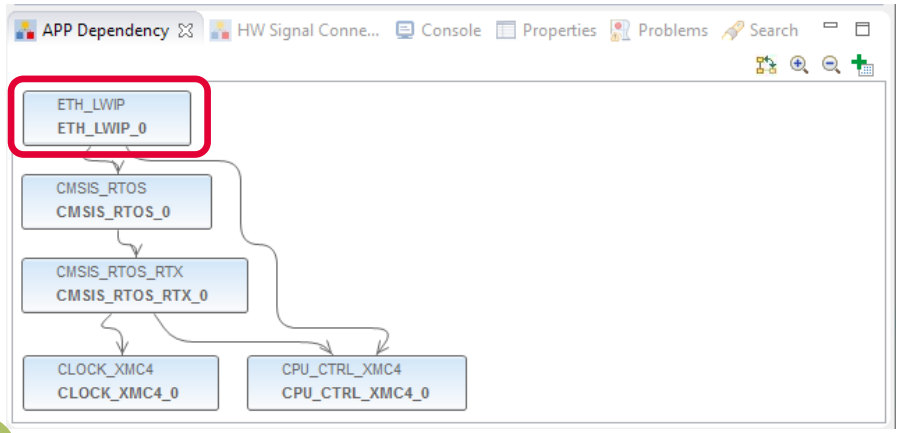
Application – Data flow



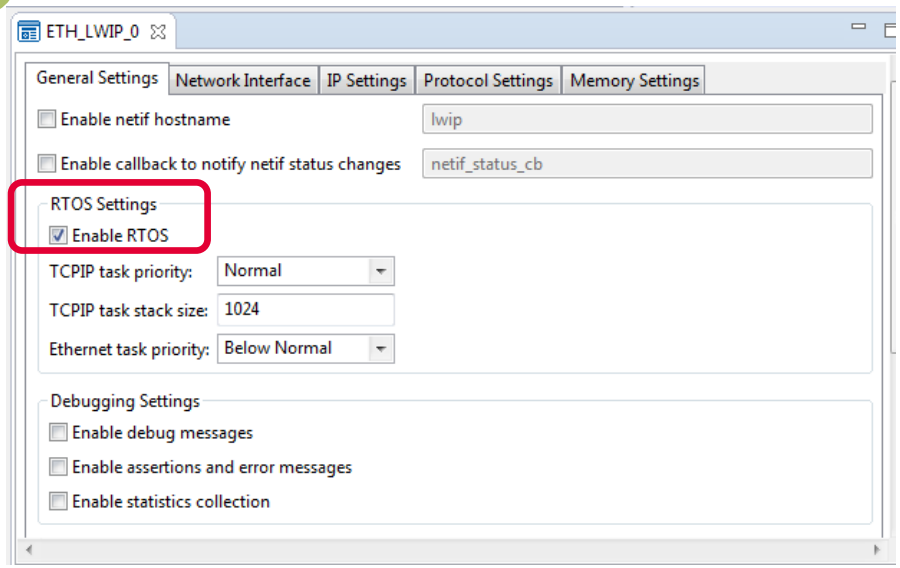
All read/write accesses of the MODBUS® Master to the register banks are handled inside callbacks from the protocol stack. The implementation of these callbacks is specific to the application.
Some of these registers are mapped to input button for demonstration.

APP structure – Ethernet configuration

1



2

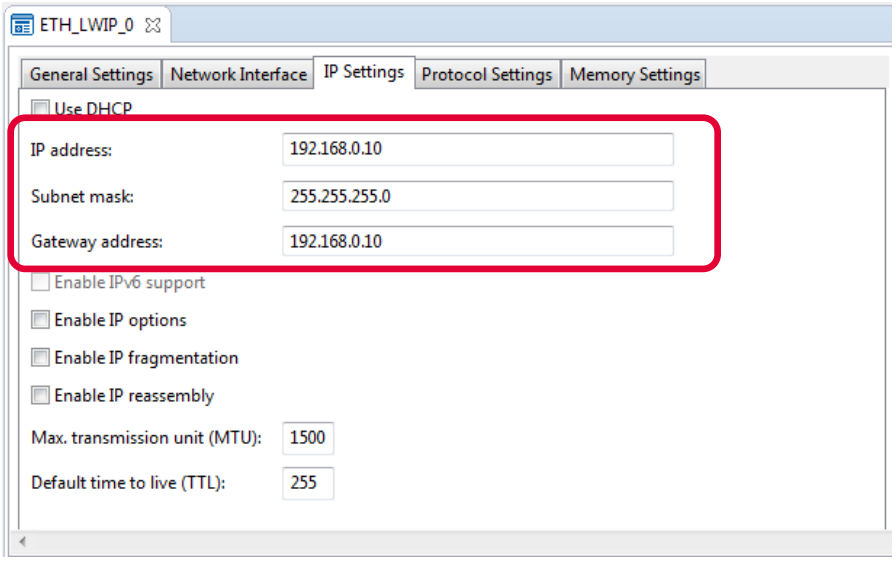


1 ETH_LWIP APP is used to implement the ethernet low level drivers and the lwIP TCP/IP protocol stack. It is the only APP actively added to the project. All other APPs are automatically included by DAVE™ when the dependencies are resolved

2 Select „Enable RTOS“ inside ETH_LWIP sttings. This allows concurrent execution of Modbus® protocol stack and your application in a seperate thread beside the lwIP TCP/IP stack processing

APP structure – Ethernet configuration

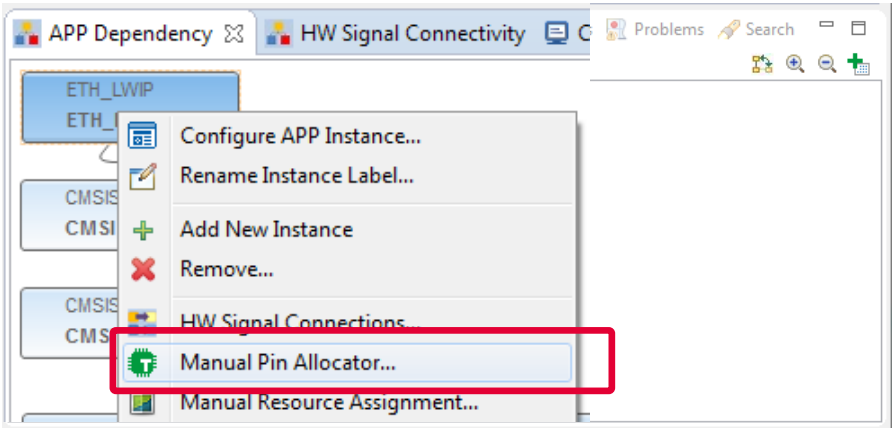
3



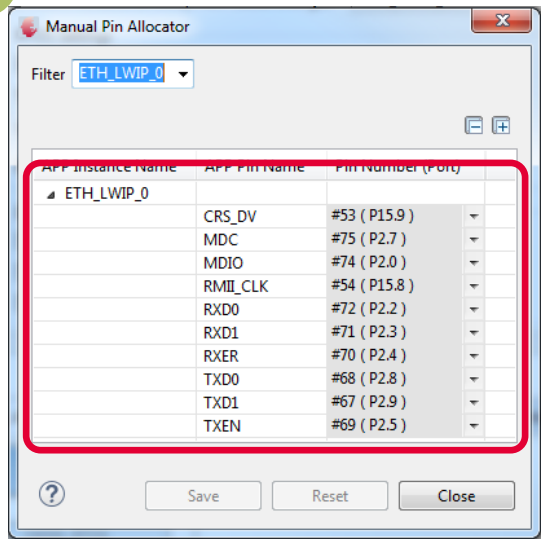
3 Inside tab „IP Settings“ you can customize your IP address. For this example the default settings are used

Pinning – Ethernet ports and physical connection

1



2



1 Right click on the ETH_LWIP APP. From the context menu select „Manual Pin Allocator“ to open the pin allocation for the ethernet module

2 Inside „Manual Pin Allocator“ you can configure the ethernet ports for your application. For the example provided, the configuration fits to the XMC4700 Relax Kit and XMC4800 Relax EtherCAT Kit

Application – Initialization inside main.c

main.c

```
/**
 * @brief
 *
 * Initializes the consumed APPs and creates threads for
 * modbus TCP, modbus polling and starts the RTOS kernel
 *
 *
 * */
int32_t main(void)
{
/*Initialize DAVE */
if(DAVE_Init() == DAVE_STATUS_FAILURE)
{
  /* Placeholder for error handler code.*/
  XMC_DEBUG(("DAVE APPs initialization failed\n"));
  while(1U)
  {
      /* do nothing */
  }
}

/* INITIALIZE BUTTON1 ON PORT 5.13 FOR INPUT */
/* Set mode to input tristate */
XMC_GPIO_SetMode(P15_13, XMC_GPIO_MODE_INPUT_TRISTATE);
/* Enable digital input. Only needed because P15.13 is an analog port */
XMC_GPIO_EnableDigitalInput(P15_13);
/* INITIALIZE BUTTON2 ON PORT 5.12 FOR INPUT */
/* Set mode to input tristate */
XMC_GPIO_SetMode(P15_12, XMC_GPIO_MODE_INPUT_TRISTATE);
/* Enable digital input. Only needed because P15.12 is an analog port */
XMC_GPIO_EnableDigitalInput(P15_12);

/*Thread for Modbus Polling*/
mbtcp_thread_id = osThreadCreate(osThread(mbtcp_thread), NULL);
/* Starting the Kernel for Scheduling*/
if(osKernelStart() == osOK)
{
  /* Placeholder for error handler code.*/
  XMC_DEBUG(("OS initialization failed\n"));
  while(1U)
  {
      /* do nothing */
  }
}

while(1)
{
  /* do nothing */
}
}
```

1. Initialize DAVE™ system including all APPs
2. Initialize the Relax Kit input ports of button1 and button2
3. Create and start thread to poll Modbus® protocol stack and input buttons
4. main idle loop

Application – Modbus® application thread

main.c

```
/**
 * @mbtcp_thread
 * Thread to setup tcp stack and process of modbus protocol stack
 * */
void mbtcp_thread(void const *argument)
{
  /* Init modbus stack.*/
  (void)eMBTCPInit(MB_TCP_DEFAULT_PORT);
  /* Configure the slave id of the device.*/
  (void)eMBSetSlaveID( 0x11U, true, NULL, 0U );
  /*Enable the Modbus Protocol Stack.*/
  (void)eMBEnable();

  /* Initialise the discrete input registers with zero's and one's for
   * demonstration purpose within this example. */
  xMBUtilSetBits( reg_discrete_input_buffer, 2U, 2U, 3U );    /* Set bit 2:3 to 11b. */
  xMBUtilSetBits( reg_discrete_input_buffer, 8U, 1U, 1U );    /* Set bit 8 to 1b. */

  /* Modbus polling loop waiting for an event*/
  for( ;; )
  {
    /* Process modbus protocol stack */
    (void)eMBPoll();

    /* Increment input register on address 1 to simulate changing input value.*/
    reg_input_buffer[0]++;

    /* Read input of button 1 and store inside bit 0 of discrete input registers */
    if (XMC_GPIO_GetInput(P15_13))
    {
      xMBUtilSetBits( reg_discrete_input_buffer, 0U, 1U, 1U );    /* Set bit 0 to 1b. */
    }
    else
    {
      xMBUtilSetBits( reg_discrete_input_buffer, 0U, 1U, 0U );    /* Set bit 0 to 0b. */
    }

    /* Read input of button 2 and store inside bit 1 of discrete input registers */
    if (XMC_GPIO_GetInput(P15_12))
    {
      xMBUtilSetBits( reg_discrete_input_buffer, 1U, 1U, 1U );    /* Set bit 1 to 1b. */
    }
    else
    {
      xMBUtilSetBits( reg_discrete_input_buffer, 1U, 1U, 0U );    /* Set bit 1 to 0b. */
    }
  }
}
```

1. Initialize, configure IDs and enable Modbus® stack
2. Initialize some of the local discrete input buffers. Only for demonstration purpose inside this example
3. Execute Modbus® protocol stack
4. Increment with every cycle the first entry of the local 16 bit read-only input buffer. Only for demonstration purpose inside this example
5. Read button 1 and button 2 state and write button state to bit 0 and bit 1 of discrete input buffer. Only for demonstration purpose inside this example

Application – Defining local register banks

1

main.c

```
/*****************************************************************
 * MACROS AND DEFINES
 *****************************************************************/
/* Input Register Definition (16 bit; Read-Only) */
/* Input Register Start Address */
#define REG_INPUT_START_ADDR 1U
/* No of Input Registers*/
#define REG_INPUT_COUNT 4U

/* Holding Register Definition (16 bit; Read-Write) */
/* Holding Register Start Address */
#define REG_HOLDING_START_ADDR 10U
/* No of Holding Registers */
#define REG_HOLDING_COUNT 130U

/* Coil Register Definition (1 bit; Read-Write) */
/* Coil Register Start Address */
#define REG_COILS_START_ADDR 1000U
/* No of Coil Registers*/
#define REG_COILS_COUNT 16U

/* Discrete Inputs Definition (1 bit; Read-Only) */
/* Discrete Inputs Start Address */
#define REG_DISC_START_ADDR 2000U
/* No of Discrete Inputs */
#define REG_DISC_COUNT 16U
```

2

main.c

```
/*****************************************************************
 * LOCAL DATA
 *****************************************************************/
static uint16_t reg_input_buffer[REG_INPUT_COUNT];
/* Allocate buffer for Holding Register (16bit; Read-Write) */
static uint16_t reg_holding_buffer[REG_HOLDING_COUNT];
/* Allocate buffer for discrete input register (1bit; Read-Only) */
static uint8_t reg_discrete_input_buffer[ (((uint16_t)REG_DISC_COUNT-(uint16_t)1) / 8U) + (uint16_t)1U];
/* Allocate buffer for coil register (1bit; Read-Write) */
static uint8_t reg_coils_buffers[ (((uint16_t)REG_COILS_COUNT-(uint16_t)1) / 8U) + 1U];
```

1 Definition of size and start addresses of local register banks:
input registers (16 bit;RO)
holding registers (16 bit;RW)
coil registers (1 bit;RW)
discrete input registers (1 bit;RO)

2 Allocation of memory for Modbus® register banks

Application – Discrete input registers callback

main.c

```
eMBErrorCode eMBRegDiscreteCB( uint8_t *buffer, uint16_t address, uint16_t count )
{
  eMBErrorCode status = MB_ENOERR;
  int16_t signed_count = (int16_t)count;
  uint16_t bit_offset;

  /* Check if we have registers mapped at this block. */
  if( ( address >= REG_DISC_START_ADDR ) &&
    ( (uint16_t)(address + count) <= (uint16_t)(REG_DISC_START_ADDR + REG_DISC_COUNT)) )
  {
    bit_offset = (uint16_t)( address - REG_DISC_START_ADDR );
    while( signed_count > 0 )
    {
      if (signed_count > 8)
      {
         *buffer = xMBUtilGetBits( reg_discrete_input_buffer, bit_offset, (uint8_t)8 );
      }
      else
      {
         *buffer = xMBUtilGetBits( reg_discrete_input_buffer, bit_offset, (uint8_t)signed_count);
      }
      buffer++;
      signed_count -= 8;
      bit_offset += (uint8_t)8;
    }
  }
  else
  {
    status = MB_ENOREG;
  }
  return status;
}
```

Callback which is called from Modbus® protocol stack to read local register banks of 1 bit discrete input registers

Application – Coil registers callback

main.c

```
eMBErrorCode eMBRegCoilsCB( uint8_t *buffer, uint16_t address, uint16_t count, eMBRegisterMode mode )
{
  eMBErrorCode status = MB_ENOERR;
  int16_t signed_count = (int16_t)count;
  uint16_t bit_offset;

  /* Check if we have registers mapped at this block. */
  if( ( address >= REG_COILS_START_ADDR ) &&
      ( (uint16_t)(address + count) <= (uint16_t)(REG_COILS_START_ADDR + REG_COILS_COUNT)) )
  {
    bit_offset = ( uint16_t )( address - REG_COILS_START_ADDR );
    switch ( mode )
    {
      /* Read current values and pass to protocol stack. */
      case MB_REG_READ:
        while( signed_count > 0 )
        {
          if (signed_count > 8)
          {
            *buffer = xMBUtilGetBits( reg_coils_buffers, bit_offset, 8U);
          }
          else
          {
            *buffer = xMBUtilGetBits( reg_coils_buffers, bit_offset, (uint8_t)signed_count);
          }
          buffer++;
          signed_count -= 8;
          bit_offset += 8U;
        }
        break;
      /* Update current register values. */
      case MB_REG_WRITE:
        while( signed_count > 0 )
        {
          if (signed_count > 8)
          {
            xMBUtilSetBits( reg_coils_buffers, bit_offset, 8U, *buffer );
          }
          else
          {
            xMBUtilSetBits( reg_coils_buffers, bit_offset, (uint8_t)signed_count, *buffer );
          }
          buffer++;
          signed_count -= 8;
          bit_offset += (uint8_t)8;
        }
        break;
      default:
        status = MB_ENOREG;
        break;
    }
  }
  else
  {
    status = MB_ENOREG;
  }
  return status;
}
```

Callback which is called from Modbus® protocol stack to read and write local register banks of 1 bit coil input/output registers

Application – Holding registers callback

main.c

```
eMBErrorCode eMBRegHoldingCB( uint8_t *buffer, uint16_t address, uint16_t count, eMBRegisterMode mode )
{
  eMBErrorCode status = MB_ENOERR;
  uint16_t register_index;

  if ( ( address >= REG_HOLDING_START_ADDR ) &&
     ( (uint16_t)(address + count) <= (uint16_t)(REG_HOLDING_START_ADDR + REG_HOLDING_COUNT)) )
  {
    register_index = ( uint16_t )( address - REG_HOLDING_START_ADDR );
    switch ( mode )
    {
      /* Pass current register values to the protocol stack. */
      case MB_REG_READ:
        while( count > 0U )
        {
          *buffer = (uint8_t)( reg_holding_buffer[register_index] >> 8 );
          buffer++;
          *buffer = (uint8_t)( reg_holding_buffer[register_index] & 0xFFU );
          buffer++;
          register_index++;
          count--;
        }
        break;
      /* Update current register values with new values from the
       * protocol stack. */
      case MB_REG_WRITE:
        while( count > 0U )
        {
          reg_holding_buffer[register_index] = (uint16_t)((uint16_t)*buffer << 8);
          buffer++;
          reg_holding_buffer[register_index] |= *buffer;
          buffer++;
          register_index++;
          count--;
        }
        break;
      default:
        status = MB_ENOREG;
        break;
    }
  }
  else
  {
      status = MB_ENOREG;
  }
  return status;
}
```

Callback which is called from Modbus® protocol stack to read and write local register banks of 16 bit holding input/output registers

Application – Input registers callback

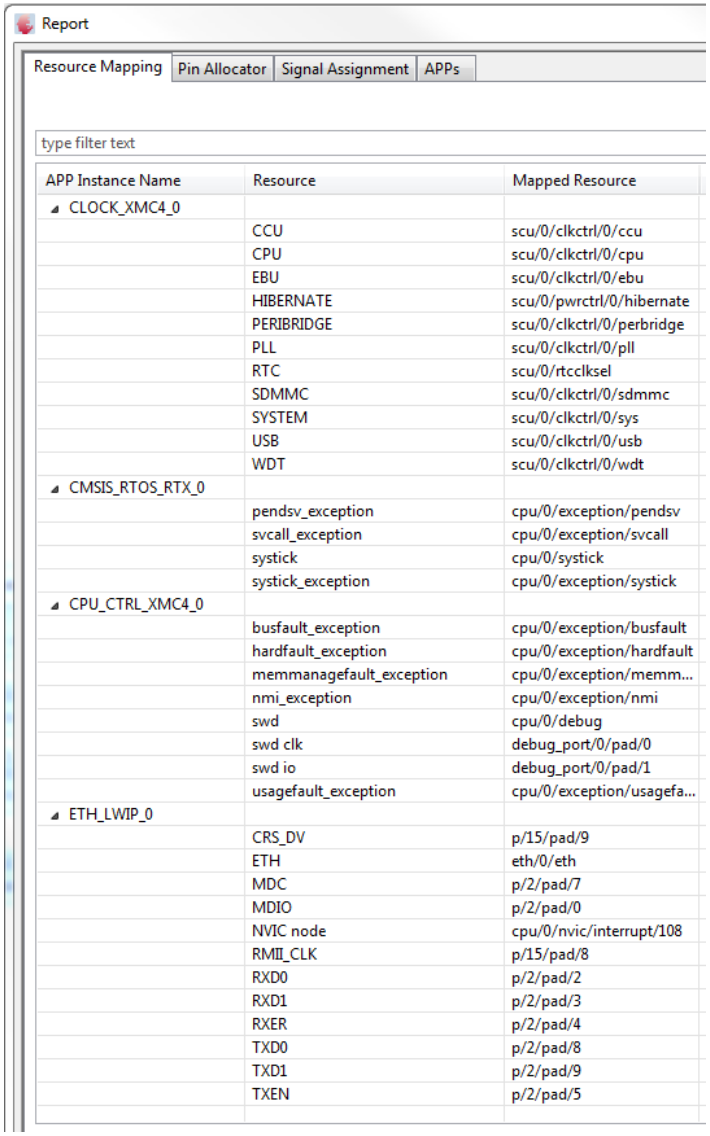
main.c

```
eMBErrorCode eMBRegInputCB( uint8_t *buffer, uint16_t address, uint16_t count )
{
  eMBErrorCode status = MB_ENOERR;
  uint16_t     register_index;

  if (( address >= (uint16_t)REG_INPUT_START_ADDR )
     && ( (uint16_t)(address + count) <= (uint16_t)(REG_INPUT_START_ADDR + REG_INPUT_COUNT) ))
  {
    register_index = ( uint16_t )( address - REG_INPUT_START_ADDR );
    while( count > 0U )
    {
      /* Pass current register values to the protocol stack. */
      *buffer = ( uint8_t )( reg_input_buffer[register_index] >> 8 );
      buffer++;
      *buffer = ( uint8_t )( reg_input_buffer[register_index] & 0xFFU );
      buffer++;
      register_index++;
      count--;
    }
  }
  else
  {
    status = MB_ENOREG;
  }
  return status;
}
```

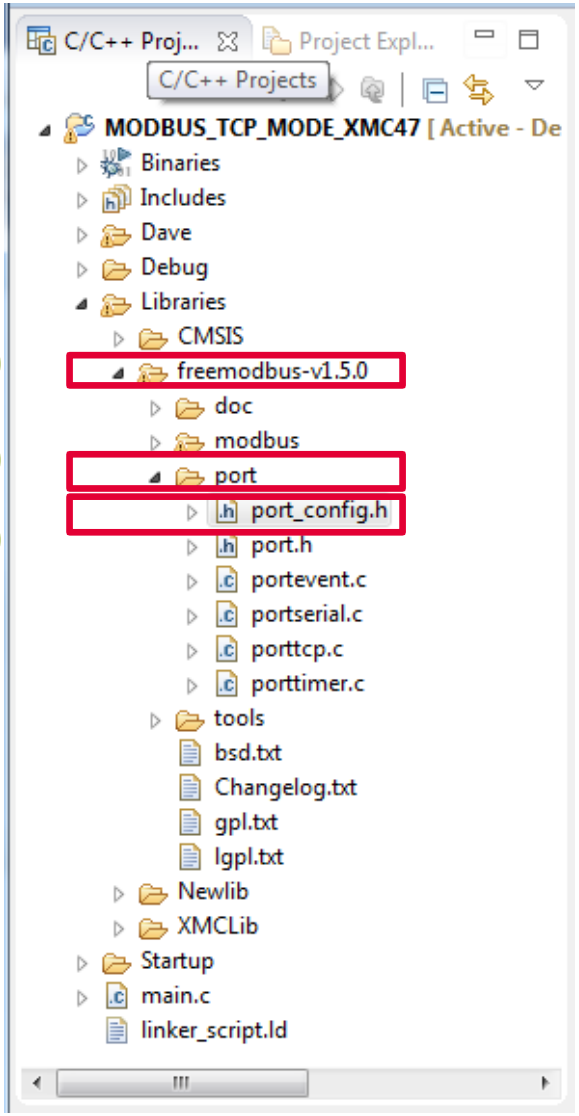
Callback which is called from Modbus® protocol stack to read local register banks of 16 bit input registers

Application – Overview on used HW resources



APP Instance Name	Resource	Mapped Resource
CLOCK_XMC4_0		
	CCU	scu/0/clkctrl/0/ccu
	CPU	scu/0/clkctrl/0/cpu
	EBU	scu/0/clkctrl/0/ebu
	HIBERNATE	scu/0/pwrctrl/0/hibernate
	PERIBRIDGE	scu/0/clkctrl/0/perbridge
	PLL	scu/0/clkctrl/0/pll
	RTC	scu/0/rtcclksel
	SDMMC	scu/0/clkctrl/0/sdmmc
	SYSTEM	scu/0/clkctrl/0/sys
	USB	scu/0/clkctrl/0/usb
	WDT	scu/0/clkctrl/0/wdt
CMSIS_RTOS_RTX_0		
	pendsv_exception	cpu/0/exception/pendsv
	svcall_exception	cpu/0/exception/svcall
	systick	cpu/0/systick
	systick_exception	cpu/0/exception/systick
CPU_CTRL_XMC4_0		
	busfault_exception	cpu/0/exception/busfault
	hardfault_exception	cpu/0/exception/hardfault
	memmanagefault_exception	cpu/0/exception/memm...
	nmi_exception	cpu/0/exception/nmi
	swd	cpu/0/debug
	swd clk	debug_port/0/pad/0
	swd io	debug_port/0/pad/1
	usagefault_exception	cpu/0/exception/usagefa...
ETH_LWIP_0		
	CRS_DV	p/15/pad/9
	ETH	eth/0/eth
	MDC	p/2/pad/7
	MDIO	p/2/pad/0
	NVIC node	cpu/0/nvic/interrupt/108
	RMII_CLK	p/15/pad/8
	RXD0	p/2/pad/2
	RXD1	p/2/pad/3
	RXER	p/2/pad/4
	TXD0	p/2/pad/8
	TXD1	p/2/pad/9
	TXEN	p/2/pad/5

Library – FreeMODBUS v1.5.0 protocol stack



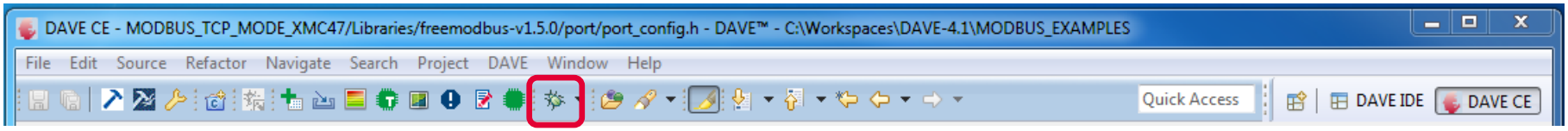
1. The FreeMODBUS protocol stack library is located inside the Libraries folder
2. Source code for porting the library to XMC4000 family is located inside port folder
3. You can modify port_config.h for customized configurations. For example, here you can define, which port (default: 502) is used by your client

1. Overview and Requirements
2. Setup
3. Implementing the application
4. How to test

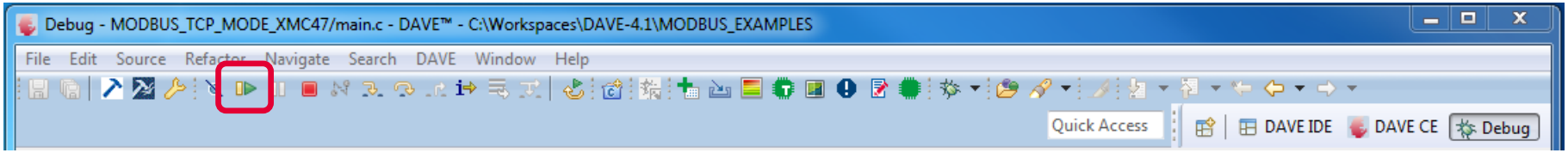
How to test – Build and start the Modbus® client to run

ACTIONS

1. Build and download the example application software to the XMC4700 / XMC4800 Relax Kit and start the debugger

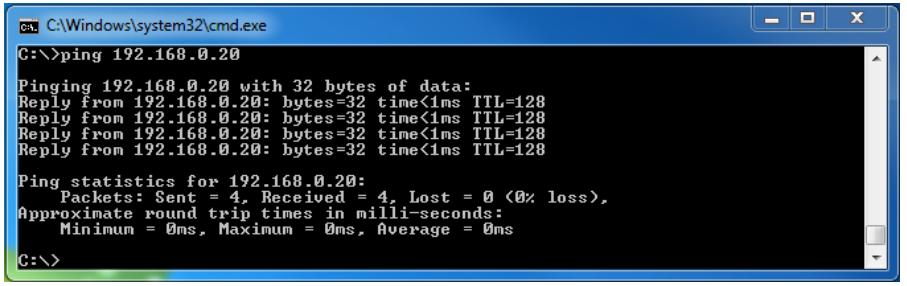


2. Start the software by the run button



OBSERVATIONS

Your Modbus® client replies to ping



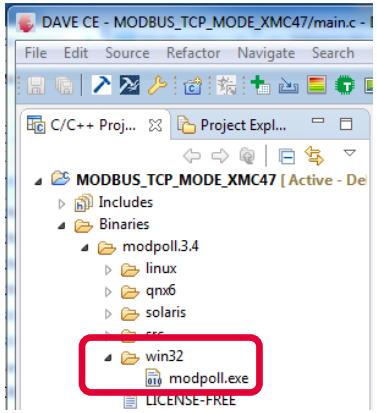
How to test – Read 16 bit input registers



ACTIONS

Use „Modpoll Modbus Master Simulator" which is provided inside example project to poll input registers 1 to 4:

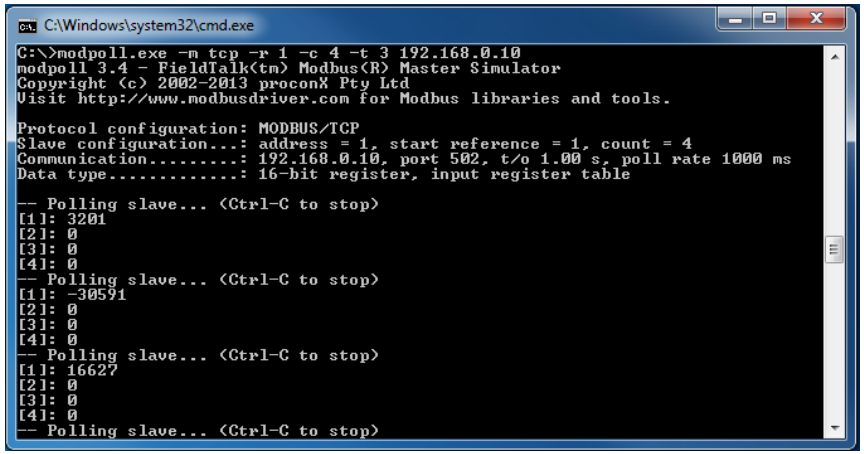
```
modpoll.exe -m tcp -r 1 -c 4 -t 3 192.168.0.10
```





OBSERVATIONS

Input registers 1 to 4 are polled every second by master.
The value of register 1 is non-static, because it is incremented inside the mbtcp_thread endless loop of the example application



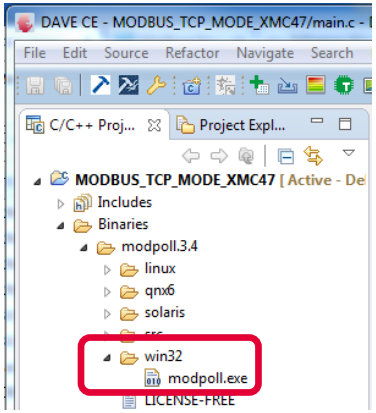
How to test – Read invalid 16 bit input registers



ACTIONS

Use „Modpoll Modbus Master Simulator" which is provided inside example project to poll input registers 2 to 5:

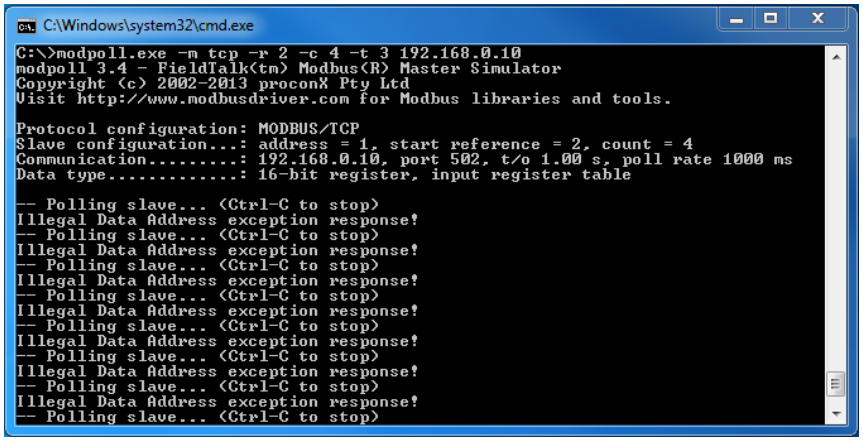
```
modpoll.exe -m tcp -r 2 -c 4 -t 3 192.168.0.10
```





OBSERVATIONS

Modbus® client returns error code because register address 5 is not defined inside example application



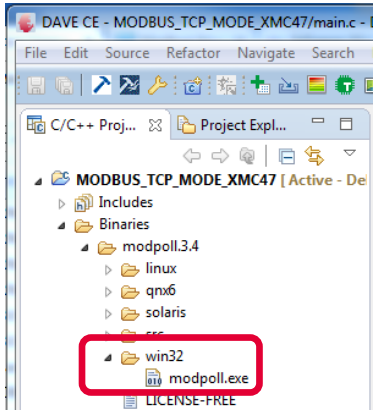
How to test – Read 1 bit discrete input registers



ACTIONS

1. Use „Modpoll Modbus Master Simulator" which is provided inside example project to poll discrete input bits on address 2000 to 2001:

```
modpoll.exe -m tcp -r 2000 -c 2 -t 1 192.168.0.10
```

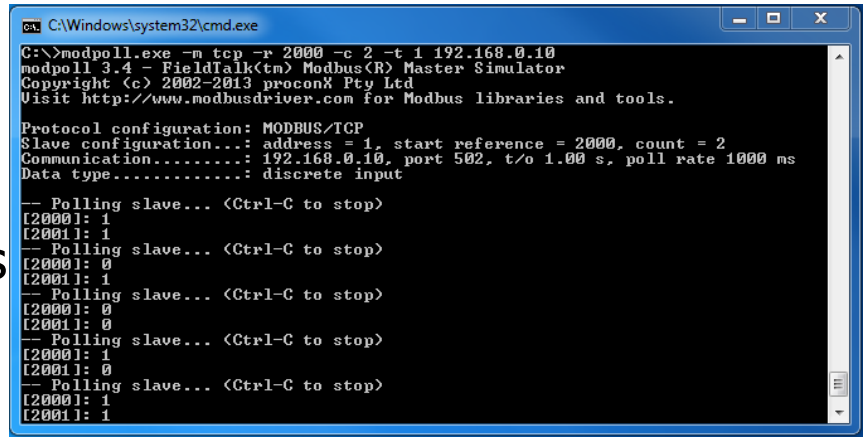


2. Push button1 and button2 while polling



OBSERVATIONS

Discrete input registers on address 2000 to 2001 are polled every second. Their value reflects the state of button1 and button2





Part of your life. Part of tomorrow.

Infineon