Embedding it better...

µTasker Document

MODBUS Module

Table of Contents

1. Introduction 3
2. What exactly is MODBUS and where can detailed information about its specification be found? 3
3. µTasker MODBUS ports 4
3.1. Optional Dedicated TCP/Serial Gateway Functionality 5
4. How to add the µTasker MODBUS Module to a Project 6
5. Project Defines 7
6. Configuring a MODBUS Serial Slave Device 9
6.1. Serial Interface Settings 14
6.2. Multiple Serial Slaves on a Single Interface 15
7. Example of Configuration and Control of Coils 17
7.1. Example of the use of fnPreFunction() 19
7.2. Example of the use of fnPostFunction () 20
7.3. fnPreFunction() and fnPostFunction () Message Routing 22
8. Overlaying Coils and Discretes with Registers 23
9. User Defined Function Codes and Handling of Public Function Codes by the Application 25
9.1. User Function Routing 26
10. MODBUS RTU Mode 27
11. RS485 Operation 28
12. Configuring a MODBUS TCP Slave Device 30
12.1. Multiple TCP Slaves on a Single TCP Socket 32
13. Configuring MODBUS TCP to UART gateways 34
14. Configuring Address-based Gateways 36
14.1. Serial to Serial MODBUS Bridge 36
14.2. TCP to Serial MODBUS Bridge 41
14.3. Mapping Slave Addresses in a Gateway Transfer 44
15. Configuring a MODBUS master 45
15.1. MODBUS Master Transmission 46
15.2. MODBUS Master Call-Back Events 47
16. MODBUS Queues 49
17. Delayed Responses 52
18. USB Slave Support 54
19. TCP Connection Details – Masters and Gateways 55
20. FLASH and SRAM Consumption by the MODBUS Module 60
21. Testing the MODBUS Reference Project in the µTasker Simulator 61
22. Conclusion 63
Appendix A – Public Functions Supported by the µTasker MODBUS Module 65
Appendix B – Hardware Dependencies 69
a) Hardware Timers used by the MODBUS Module 69
b) Processor-specific Hardware Defines 70
c) RTS Delay Time Measurements 77

1. Introduction

The µTasker project V1.4 delivers the foundation required for building modular and reliable user defined applications: its operating system is tightly integrated with a TCP/IP stack and device drivers for various single-chip processors; the project includes a powerful demonstration with many features required in real-world projects and ready-to-build projects for various compiler environments; the µTasker simulator enables testing and further developing within a comfortable and efficient environment in order to reduce project development times.

The µTasker MODBUS software module is a 'drop-in' protocol module which enables devices to be integrated into a MODBUS environment in a simple but very flexible manor. It allows operation as a ***serial slave*** device (RTU or ASCII), a ***TCP slave*** device, a ***TCP gateway*** to serial (RTU or ASCII), a ***TCP master*** or ***serial master*** (RTU or ASCII), and a mixture of these types to ***multiple TCP*** ports and/or ***multiple serial interfaces*** as desired.

The software is a single module with all features configurable by define and/or at run time making it a truly universal module, suitable for just about any use.

The µTasker MODBUS software is easy to integrate into existing projects or new projects and can be fully simulated to ensure efficient development of user applications and validation.

2. What exactly is MODBUS and where can detailed information about its specification be found?

MODBUS is quite an old, but still very popular, industrial protocol. It has its roots in the serial communication between two or more devices on a bus, using either RS232 or RS485. There is one master on the bus which initiates all requests or commands and one or more slaves. The serial communication can be ASCII or else binary RTU (Remote Terminal Unit) based. The full specification of the serial mode can be found at http://www.modbus.org/ - the document used for the implementation and validation of the µTasker MODBUS software is "***MODBUS over Serial Line – Specification and Implementation Guide V1.02***"

The application protocol is generally not serial communication specific and is detailed in the document "***MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b***"

An extension to MODBUS to work over Ethernet (MODBUS TCP) is detailed in the document "***MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b***".

It is assumed that the reader understands the basics of MODBUS operation during the rest of the document. The abovementioned documents should be adequate as an introduction to all concepts and as a reference to the technical details.

For details concerning the UART interface in the µTasker project see the "**UART User's Guide**" at http://www.uTasker.com/docs/uTasker/uTaskerUART.PDF

3. µTasker MODBUS ports

MODBUS Serial Ports

Each ***MODBUS serial port*** type can be assigned to a ***HW UART*** of its choice, which then has its own defined configuration (Baud rate, parity, etc. and mode - ASCII or RTU).

MODBUS TCP slaves (listeners)

Each ***MODBUS TCP slave (listener)*** has a TCP port number and a definable number of sessions. The number of sessions determines how many MODBUS TCP masters can be connected to each TCP port at the same time.

MODBUS TCP masters (clients)

Each ***MODBUS TCP master (client)*** has an IP address and port number for contacting and communicating with its destination TCP slave (listener).

All references to MODBUS ports are performed via their MODBUS port numbers. This means that a command to send a MODBUS master request can be easily sent to either a MODBUS serial master port or a MODBUS TCP master port as shown in the following example:

```
#define MODBUS_TCP_MASTER_PORT    4
#define MODBUS_SERIAL_MASTER_PORT 1

static const MODBUS_READ_QUANTITY read_coils = {
         57,                                        // starting at coil address 57
         11};                                       // read 11 coils

fnMODBUS_Master_send(MODBUS_TCP_MASTER_PORT, SLAVE_ADDRESS, MODBUS_READ_COILS,
                                                    (void *)&read_coils);

fnMODBUS_Master_send(MODBUS_SERIAL_MASTER_PORT, SLAVE_ADDRESS, MODBUS_READ_COILS,
                                                    (void *)&read_coils);
```

The first transmission is from the TCP master and the second from the serial master.

All communication is thus MODBUS port number specific and generic, irrespective of whether the ports are physically association with serial HW or Ethernet TCP.

The following shows a typical configuration, illustrating some possible configuration relationships.

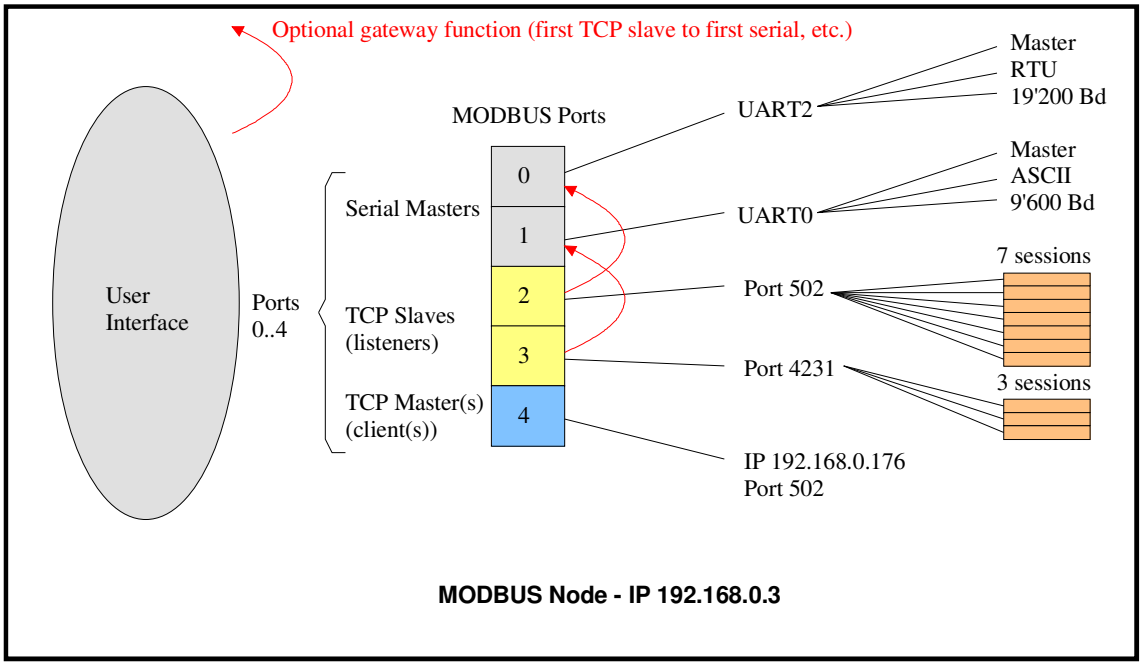

Figure 3-1: A typical configuration with two UART masters, two TCP slaves and a TCP master

3.1. Optional Dedicated TCP/Serial Gateway Functionality

The MODBUS TCP slave ports can optionally act as dedicated gateways to MODBUS serial ports. Whether this function is active or not is defined by a MODBUS configuration parameter.

In the above diagram, both MODBUS TCP slave ports are defined as gateways. Per default, there is one MODBUS TCP slave for each MODBUS serial slave port and so the coupling between these is also defined by the ordering in the port number sequencing. The first MODBUS TCP slave port is always coupled with the first MODBUS serial port. The second MODBUS TCP slave port is always coupled with the second MODBUS serial port. Since the coupling is optional, only the first or second pair could be defined. *Additional methods of configuring gateway routing based on slave addresses is described later in the document.*

The MODBUS TCP slave ports are configured in the MODBUS_PARS table as detailed later in the document. The following shows the configuration flags belonging to TCP port 2 for the gateway functionality, plus its initialisation call:

```
...
(MODBUS_TCP_SERIAL_GATEWAY | MODBUS_TCP_SLAVE_PORT), // MODBUS mode -
                                                        TCP slave port 2
...
fnInitialiseMODBUS_port(2, &modbus_configuration, &modbus_slave_callbacks, 0);
```

The dedicated gateway functionality (TCP slave to UART master) is selected when the flag

MODBUS_TCP_SERIAL_GATEWAY is set and the final initialisation parameter (a master call back routine) is zero.

Note that the number of sessions belonging to each MODBUS TCP slave port is defined in config.h by defines MODBUS_SOCKETS_0 and MODBUS_SOCKETS_1 , etc.

MODBUS TCP slave port 2 acts as a gateway to MODBUS serial port 0 due to its gateway parameter flag MODBUS_TCP_SERIAL_GATEWAY. Received MODBUS queries are therefore passed on to the UART 2. MOSBUS responses from a remote slave are returned via the TCP slave to the requesting remote MODBUS TCP master.

4. How to add the µTasker MODBUS Module to a Project

The MODBUS protocol software is delivered as three files which can be added to any project:

- MODBUS.c [contains all MODBUS support and the MODBUS task]
- modbus.h [contains defines required by the MODBUS module and its user application]
- modbus_app.c [contains an example application configuration]

The software is written in a processor independent manor as far as possible (for example it is little- and big-endian independent) but includes small amounts of hardware timer code when serial RTU mode support is required. *See Appendix B to find out whether the RTU mode is ready for your processor*.

The protocol requires one MODBUS task to be operating, which is added to a project using the following method.

In TaskConfig.h the task name is defined, eg.

```
#define TASK_MODBUS      'O'           // MODBUS task
```

The MODBUS task prototype is added:

```
extern void fnMODBUS(TTASKTABLE *);
```

The task is added to the task list:

```
const UTASK_TASK ctNodes[] = {
  DEFAULT_NODE_NUMBER,    // configuration our single node
...
  TASK_MODBUS,            // MODBUS task
...
};
```

And finally to the task table:

```
const UTASKTABLEINIT ctTaskTable[] = {
...
{ "O-MOD", fnMODBUS, MEDIUM_QUE,(DELAY_LIMIT)(NO_DELAY_RESERVE_MONO), 0,
                                              UTASKER_STOP}, // MODBUS task
...
};
```

This doesn't start the MODBUS task but enters it into the scheduling table and gives it resources ready for it to be used.

The next step is to configure the MODBUS interface and actively support it where appropriate. The next sections contain various typical configurations and give specific examples of the use in these modes. This practical approach should help understanding of the way that the module is configured and how it runs, what work it does autonomously and where the application is required to interface to it to achieve its project-specific goals.

5. Project Defines

In order to activate general MODBUS support in the project the define `USE_MODBUS` must be active in `config.h`.

Further defines in this file control in more detail exactly which aspects of MODBUS are included in the code and the demo software. Features that are not required can be easily removed to save space and improve efficiency.

```
#define USE_MODBUS_SLAVE            // support slave mode of operation
#define USE_MODBUS_MASTER           // support master mode of operation
```

MODBUS serial port type:

```
#define MODBUS_RTU                  // support binary RTU mode
#define MODBUS_ASCII                // support ASCII mode
#define STRICT_MODBUS_SERIAL_MODE   // automatically adjust the UART character
                                       length according to mode (ASCII 7 bit
                                       characters and RTU 8 bit) and force 2 stop
                                       bits if no parity is selected
#define MODBUS_RS485_SUPPORT        // support RTS control for RS485 mode
#define FAST_MODBUS_RTU             // speeds of greater than 19'200 use calculated
                                       RTU times rather than recommended fixed
                                       values
#define MODBUS_SUPPORT_SERIAL_LINE_FUNCTIONS // support the serial line function at
                                                the slave
#define MODBUS_SUPPORT_SERIAL_LINE_DIAGNOSTICS // support serial line diagnostics
#define MODBUS_CRC_FROM_LOOKUP_TABLE // use a look up table for RTU CRC calculation
                                       (requires more FLASH memory but faster)
```

```
#define MODBUS_SERIAL_INTERFACES    2    // the number of MODBUS serial ports
#define MODBUS_USB_SLAVE                 // MODBUS serial slave realised as USB
```

MODBUS TCP port type:

```
#define MODBUS_TCP                       // support MODBUS TCP
#define MODBUS_TCP_GATEWAY               // TCP gateway support enabled
#define MODBUS_TCP_SERVERS         2     // the number of MODBUS TCP slaves
#define MODBUS_TCP_MASTERS         1     // the number of MODBUS TCP masters
#define MODBUS_SOCKETS_x           5     // the number of sessions on each
                                            MODBUS TCP slave (where x is the server
                                            number 0, 1, 2, etc.)
```

MODBUS slave options:

```
#define MODBUS_SHARED_SERIAL_INTERFACES 3
                                    // number of slave interfaces sharing UARTs
#define MODBUS_SHARED_TCP_INTERFACES 2
                                  // number of slave interfaces sharing TCP sockets
#define NOTIFY_ONLY_COIL_CHANGES    // notify user of individual coil changes only
#define MODBUS_DELAYED_RESPONSE     // allow slave parameter interface to delay
                                       request responses - for example to prepare
                                       latest data from external location
```

`MAX_QUEUED_REQUEST_LENGTH` sets the maximum length of queued requests, which can be increased or reduced depending on knowledge of request utilisation in the system. This is discussed further in the section about delayed responses.

Gateway options:

```
#define MODBUS_GATE_WAY_ROUTING    // configurable routing from slave gateways
                                      (used together with masters)
```

Queue options:

```
#define MODBUS_GATE_WAY_QUEUE      // support queuing of MODBUS transmissions -
                                      advisable for gateways and useful for masters
```

In addition to the project defines in `config.h` the hardware specific part of the MODBUS interface can be controlled in `app_hw_xxxx.h` (where `xxxx` is the processor type used for the project). The µTasker demo project is delivered with an example setup of both `config.h` and `app_hw_xxxx.h` which contains MODBUS defines.

Examples from `app_hw_xxxx.h` are:

```
#define MODBUS_UART_0              2       // MODBUS serial port 0 uses UART 2
#define MODBUS_UART_1              0       // MODBUS serial port 1 uses UART 0
```

See appendix B for further hardware defines which may be required for your particular target processor.

6. Configuring a MODBUS Serial Slave Device

The serial slave device sits on a MODBUS serial bus. It never initiates any commands itself but responds to the MODBUS master. The MODBUS master will generally be polling certain values supplied by the slave and can set certain values at the slave.

The MODBUS data model is based on 4 access types (although this can be extended by adding vendor specific functions):

- Discrete inputs – binary inputs ('0' or '1') which can be read but not modified (eg. a port input state)
- Coils – binary values ('0' or '1') which can be read and modified (eg. a port output state)
- Input registers – 16 bit registers which can only be read (eg. A/D value)
- Holding registers – 16 bit registers that can be read and modified (eg. EEPROM value or a system variable)

The grouping of the access types is quite flexible and it is also possible to overlap them so that, for example, the holding registers (R/W) could also be accessed (read) via the input registers (R).

The following diagram shows a MODBUS serial slave device with various inputs and outputs which can be grouped into the MODBUS access types.

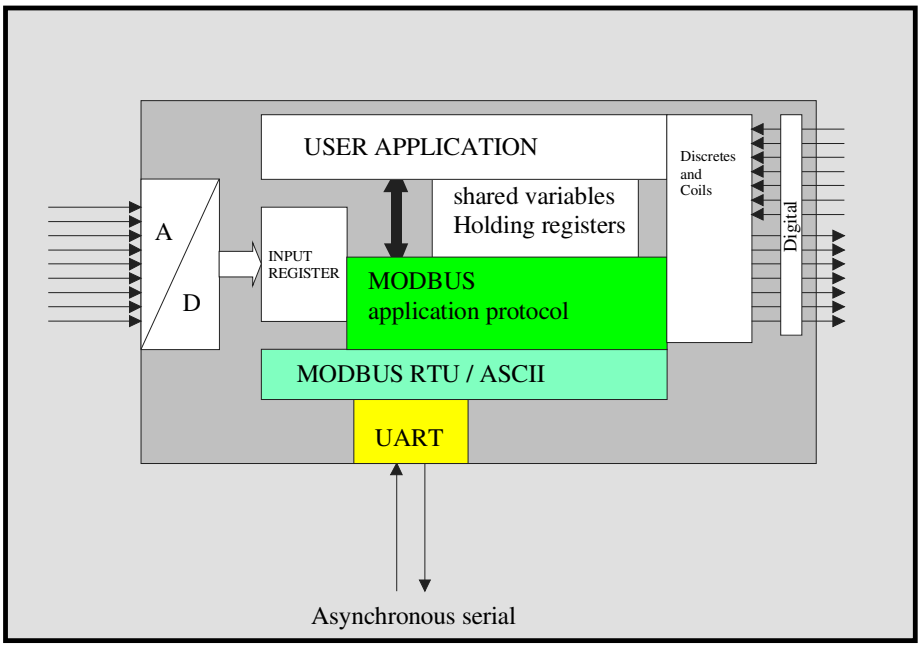

Figure 6-1: A typical MODBUS slave showing various inputs and outputs which can be accessed and controlled via the serial MODBUS connection

The master can use various public function codes to request values (single discrete inputs or register values or range of values) or modify states (single or multiple coils or single or multiple registers). Examples of the public function codes are 02 to read Discrete Inputs and 16 to write multiple holding registers – the document "***MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b***" contains the complete set and examples of their use.

The following is the user code to initialise the MODBUS serial interface:

```
...
 (MODBUS_MODE_RTU | MODBUS_SERIAL_SLAVE | MODBUS_RS485_NEGATIVE), // serial port 0 mode
                                                                    setting in
                                                                    MODBUS_PARS block
...


fnInitialiseMODBUS_port(0,                                        // MODBUS port 0
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                        0);                 // initialise MODBUS serial interface - port 0
```

```
This uses the standard MOBUS port initialisation routine, which is defined as:

extern int fnInitialiseMODBUS_port(unsigned char ucMODBUSport,
                                   const MODBUS_CONFIG *ptrMODBUS_config,
                                   const MODBUS_CALLBACKS *ptrModbus_callbacks,
                                   int ( *master_callback )( int iType, MODBUS_RX_FUNCTION *));
```

The value 0 is passed to inform the MODBUS module that this is the first MODBUS serial interface (MODBUS serial port 0). It is possible to call it again with the values 1, 2 etc. to assign further serial interfaces which could either share data or else have their own autonomous sets of data. Here a second UART interface is configured (*in this example the serial MODBUS port is a master and so also enters a master call-back function*):

```
...
 (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_POSITIVE), // serial port 1 mode
                                                                    setting in
                                                                    MODBUS_PARS block
...
fnInitialiseMODBUS_port(1,                                        // MODBUS port 1
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                        fnMODBUSmaster);      // initialise MODBUS serial interface - port 1
```

It is thus possible to have one device which is attached to two or three (limited by available serial interfaces) different serial MODBUS buses where the same device's resources are seen on each bus or each bus sees different resources. *Multiple serial connections are however usually more useful for interfacing a single TCP gateway to multiple MODBUS buses as will be seen later.*

The MODBUS module requires a global parameter pointer `ptrMODBUS_pars` which contains the configuration details of all serial interfaces. This can be a `const struct` if the configuration details are fixed but the demonstration example creates the `struct` on heap so that it can be modified if required and thus saved in the parameter system.

```
ptrMODBUS_pars = uMalloc(sizeof(MODBUS_PARS));     // get memory for the MODBUS parameters
fnGetMODBUS_parameters();       // fill the working parameters from configuration settings
```

The definition of the `MODBUS_PARS` struct can be found in `modbus.h` and an example of project-specific configuration for 2 serial channels may look like the following (from `modbus_app.c`):

```
static const MODBUS_PARS cMODBUS_default = {
    MODBUS_PAR_BLOCK_VERSION,
    {
        0x23,                           // slave address - serial port 0
        0,                              // (master address not relevant) - serial port 1
    },
    {
        (CHAR_8 + RS232_EVEN_PARITY + ONE_STOP + CHAR_MODE),
                  // serial interface settings (ASCII mode uses 7 bits) - serial port 0
        (CHAR_8 + RS232_EVEN_PARITY + ONE_STOP + CHAR_MODE),
                  // serial interface settings (ASCII mode uses 7 bits) - serial port 1
    },
    {
        SERIAL_BAUD_19200,              // baud rate of serial interface - serial port 0
        SERIAL_BAUD_9600,               // baud rate of serial interface - serial port 1
    },
    {
        (MODBUS_MODE_RTU | MODBUS_SERIAL_SLAVE | MODBUS_RS485_NEGATIVE),
                                        // default to RTU mode as slave - serial 0
        (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_POSITIVE),
                                        // default to RTU mode as master - serial 1
    },
    {                                   // only when ASCII mode enabled
        (DELAY_LIMIT)(1*SEC),           // max. inter-character delay (ASCII) - serial port 0
        (DELAY_LIMIT)(2*SEC),           // max. inter-character delay (ASCII) - serial port 1
    },
    {
        0x0a,                           // ASCII mode line feed character - serial port 0
        0x0a,                           // ASCII mode line feed character - serial port 1
    },
    {                                   // only when master operation enabled
        (DELAY_LIMIT)(2*SEC),  // MODBUS master maximum wait for a response from a slave - 0
        (DELAY_LIMIT)(3*SEC),  // MODBUS master maximum wait for a response from a slave - 1
    },
    {
        (DELAY_LIMIT)(0.2*SEC),         // MODBUS master broadcast timeout - serial port 0
        (DELAY_LIMIT)(0.2*SEC),         // MODBUS master broadcast timeout - serial port 1
    },
    {                                   // only when supporting serial line functions
        'u', 'T', 'a', 's', 'k', 'e', 'r', '-', 'M', 'O', 'D', 'B', 'U', 'S', '-',
                                                          's', 'l','a','v','e' },

    ...                 // others - see MODBUS TCP below

};
```

Inter-character delays and line feed characters are only required if ASCII mode is to be supported.

Master maximum wait times and broadcast timeouts are only required when master mode is to be supported.

The string name is only required if serial line functions are to be supported.

The characteristics of each serial interface are defined so that the corresponding UART can be configured ready for use when `fnInitialiseMODBUS_port()` is called.

The physical UART used by each serial connection should be defined in `app_hw_xxxx.h` (where `xxxx` is the processor type):

```
#define MODBUS_UART_0             2       // MODBUS serial port 0 uses UART 2
#define MODBUS_UART_1             0       // MODBUS serial port 1 uses UART 0
```

The data model for the MODBUS slave connection is passed in the MODBUS configuration

```
static const MODBUS_CONFIG modbus_configuration = {
    &test_discretes,                      // read-only discrete input configuration
    &test_coils,                          // read/write coil configuration
    &test_input_regs,                     // read-only input registers
    &test_holding_regs                    // read/write input registers
};
```

The four access types can be defined here, where a 0 means that they don't exist. The discretes and coils have a table of type `MODBUS_BITS`:

```
#define DISCRETES_START       15          // start address
#define DISCRETES_END         43          // end address
#define DISCRETES_QUANTITY    ((DISCRETES_END - DISCRETES_START) + 1) // quantity
static MODBUS_BITS_ELEMENT discretes[_MODBUS_BITS_ELEMENT_SIZE(DISCRETES_QUANTITY)] = {0};
static MODBUS_BITS test_discretes = { discretes, {DISCRETES_START, DISCRETES_END}};

#define COILS_START           10          // start address
#define COILS_END             50          // end address
#define COILS_QUANTITY        ((COILS_END - COILS_START) + 1)         // quantity
static MODBUS_BITS_ELEMENT coils[_MODBUS_BITS_ELEMENT_SIZE(COILS_QUANTITY)] = {0};
static MODBUS_BITS test_coils = { coils, {COILS_START, COILS_END}};
```

and the input and holding registers a table of type `MODBUS_REGISTERS`:

```
#define INPUT_REGS_START      24          // start address
#define INPUT_REGS_END        29          // end address
#define INPUT_REGS_QUANTITY   ((INPUT_REGS_END - INPUT_REGS_START) + 1) // quantity
static unsigned short input_regs[INPUT_REGS_QUANTITY] = {0};
static MODBUS_REGISTERS test_input_regs = { input_regs, {INPUT_REGS_START, INPUT_REGS_END}};

#define HOLDING_REGS_START    2           // start address
#define HOLDING_REGS_END      6           // end address
#define HOLDING_REGS_QUANTITY ((HOLDING_REGS_END - HOLDING_REGS_START) + 1) // quantity
static unsigned short holding_regs[HOLDING_REGS_QUANTITY] = {0};
static MODBUS_REGISTERS test_holding_regs = { holding_regs, {HOLDING_REGS_START,
HOLDING_REGS_END}};
```

In both cases the start address and end address are specified (for valid address range checking) and then an optional pointer is set to where the entries are situated in memory. For example:

```
static MODBUS_BITS_ELEMENT discretes[_MODBUS_BITS_ELEMENT_SIZE(DISCRETES_QUANTITY)] = {0};
```

and

```
static unsigned short input_regs[INPUT_REGS_QUANTITY] = {0};
```

Here the entries are arrays of local variables (elements `MODBUS_BITS_ELEMENT` are used to hold bits states or coils and discrete inputs and each register has 16 bits, as defined by MODBUS).

Based on this information, the MODBUS module can autonomously handle reading and reporting of these entry contents and can also write to coil and holding registers with no user intervention. This may be adequate for some uses, where the user can simply update values when it feels fit and read latest holding register/coil values when it needs to do so. In this case that would be all that is required to get the MODBUS slave operating!

However in most real projects this may be a little simplistic and so extra control is available in the form of call-backs and access type definitions. The call-backs are passed during configuration with the `modbus_callbacks` parameter:

```
static const MODBUS_CALLBACKS modbus_callbacks = {
    fnMODBUSPreFunction,
    fnMODBUSPostFunction,
    fnMODBUSuserFunction,
};
```

The three call-backs are used in different circumstances, which can be summarised as follows:

- `fnMODBUSPreFunction()` is called before the MODBUS slave responds to a query. This allows the user to update any entries in the table if it requires *before* the response is sent. An example may be to update a discrete entry with the latest value read from a hardware processor port. The call-back saves the user from having to regularly update entries even when they are not being read and ensures that the returned value is always up-to-date.
  In case the application needs to first collect the data from a source requiring a request (eg. from an external device via a different interface) it is possible to delay returning the value by using the "Delayed Responses" technique as detailed in chapter 17.

- `fnMODBUSPostFunction()` is called after contents have been updated, this allowing the user to react to specific changes if needed.

- `fnMODBUSuserFunction()` is called when the MODBUS module cannot read or update entries by itself. For example the table may not exist physically in memory and so the user needs to handle the function itself. This can also be used to handle function codes which do not belong to the public range, but are to be handled as vendor specific types.

If no call-backs are required, the call-back table can be entered with a zero pointer. Likewise, individual call-backs which are not required may be entered as zero pointers. Furthermore,

call-backs can be entered but no parameter table used by entering the table as a zero pointer.

Each call-back type has the same parameters and have the same return values, as described later when discussing their control capabilities.

```
int ( * )( int iType, MODBUS_RX_FUNCTION * ); // general call-back prototype
```

6.1. Serial Interface Settings

The serial interface settings are configured in the MODBUS_PARS table, as illustrated in the previous section. In accordance with the MODBUS requirements the speed can be set from 1200 Baud to 115'200 Baud. When used in RTU mode the character length should be 11 bits (*start bit, 8 data bits, parity bit and stop bit*). In ASCII mode the character length should be 10 bits (*start bit, 7 data bits, parity bit, stop bit*).

In both cases, when no parity is used, two stop bits should be set to compensation the total character length.

When the project define STRICT_MODBUS_SERIAL_MODE is set, the length of the character is automatically adjusted according to the mode setting. This means that the data bit length will be forced to 8 when RTU mode is specified and two stop bits configured when parity is disabled. Similarly in ASCII mode 7 data bits are forced as well as assuring that the parity and stop bit length is correct. User setting combinations which do not correspond to the MODBUS specification are thus automatically 'corrected' based on the selected operating mode (MODBUS_MODE_RTU or MODBUS_MODE_ASCII).

In systems where both ends of the UART connection are under proprietary control and the MODBUS specification must not be adhered to, removing the define STRICT_MODBUS_SERIAL_MODE allows any combination of setting to be used, even if they do not adhere to the formal specification.

In all case it is however important that the user selects the same setting at all UART nodes on a MODBUS bus.

6.2. Multiple Serial Slaves on a Single Interface

A MODBUS serial slave has a unique slave address on the bus and responds to requests and commands destined to this address, or to broadcast commands. As described in this section the slave can possess a set of parameters and call-backs, which are acted on or called when MODBUS functions are handled.

Activating the option MODBUS_SHARED_SERIAL_INTERFACES enables the use of multiple MODBUS serial slaves on a single serial (UART) interface. Each of these slaves can be configured with its own unique parameters and call-backs, or share parts or all with other slave interfaces. This allows a single physical device on the MODBUS to behave as several independent slave devices, where each has its own unique slave address and so responds only to functions addressed to it. Received broadcast commands are handled by all slaves in the order that the slaves are defined.

The following code shows first a MODBUS slave port being initialized, followed by two additional MODBUS slaves added to it. All slaves share the physical serial interface resources and settings as defined by the main port initialization.

```
fnInitialiseMODBUS_port(0, &modbus_configuration, &modbus_slave_callbacks, 0);
                                          // initialise MODBUS serial interface - port 0
#if defined MODBUS_SHARED_SERIAL_INTERFACES > 0
    fnShareMODBUS_port(0, &modbus_configuration_1, &modbus_slave_callbacks_1, 2);
                      // additional MODBUS serial slave sharing port 0 UART - bus address 2
    fnShareMODBUS_port(0, &modbus_configuration_2, 0, 3);
                      // additional MODBUS serial slave sharing port 0 UART - bus address 3
#endif
```

The define MODBUS_SHARED_SERIAL_INTERFACES is set to the total amount of shared slaves (the value 2 is adequate for the example) in the system. The shared slaves can be added to any existing serial MODBUS slave available and any number can be added as long as the total does not exceed the MODBUS_SHARED_SERIAL_INTERFACES limit. As the example shows, each shared slave can have its own set of parameters and call-backs.

The address given to each shared slave is passed to the fnShareMODBUS_port() and not taken from the main MODBUS configuration. These values can however also be added to the main uParameterSystem parameter block to make them user configurable. The example assumes that the serial MODBUS slave main configuration uses the slave address 1 and the shared slaves are then defined consecutive addresses 2 and 3. *No two slaves on the MODBUS serial port should have the same address*.

- When serial line functions with diagnostics counters are used, each shared slave has its own individual set of counters.
- When broadcast functions are handled, each of the shared slaves process the commands, whereby the processing order begins with the main serial MODBUS slave (slave address 1 in the example) followed by the shared slaves in the order that they were defined (2 and then 3 in the example).

Figure 6-2 illustrates the case according to the example configuration. Although there is only one physical serial interface the MODBUS master can communicate with multiple MODBUS slaves. Each slave has its own unique parameters. Slave addresses 1 and 2 have their own call-backs allowing handling MODBUS requests and commands to each slave by its own application. Slave address 3 has been configured to have no call-backs (but could have its own, or share with other slaves).

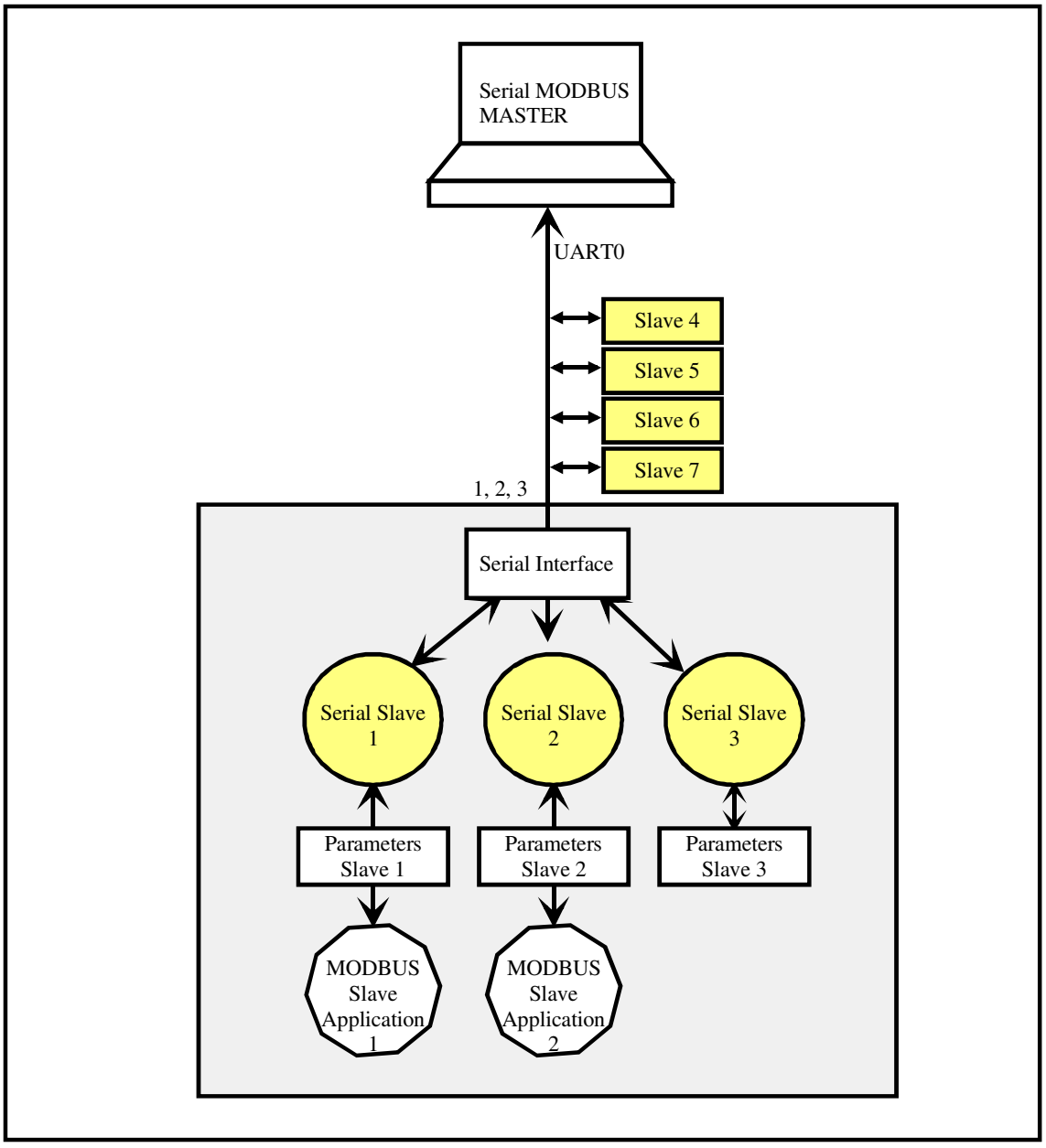

Figure 6-2: Multiple, shared MODBUS slaves on a single port

7. Example of Configuration and Control of Coils

Coils are just one example of MODBUS functions that can be controlled and will be used here to illustrate in more detail how the user specifies the control block content and how the MODBUS slave interacts with this block.

According to the MODBUS Application Protocol Specification V1.1b coils represent binary values in a system, which can be both read and written (modified). There is the capability to address up to 65'536 different coils (addresses from 0x0000 to 0xffff, which represent coils 1..65'536).

It is not necessary to support all 65'536 coils and most systems will also not have this many individual bits to control. In this case the coils are grouped together into a block with a particular address range. The following example shows the user configuration of just a small block of just 30 coils set to a block starting at the coil address 10 and ending at coil address 39.

```
#define COILS_START           10                                        // start address
#define COILS_END             39                                        // end address
#define COILS_QUANTITY        ((COILS_END - COILS_START) + 1)           // quantity
static MODBUS_BITS_ELEMENT coils[_MODBUS_BITS_ELEMENT_SIZE(COILS_QUANTITY)] = {0};
static MODBUS_BITS test_coils = { coils, {COILS_START, COILS_END}};
```

The coils block is just one of the MODBUS control block elements which are included in the user's configuration:

```
static const MODBUS_CONFIG modbus_configuration = {
    &test_discretes,                // read-only discrete input configuration
    &test_coils,                    // read/write coil configuration
    &test_input_regs,               // read-only input registers
    &test_holding_regs              // read/write input registers
};
```

Moreover, the configuration block can belong to either just one MODBUS slave interface or be shared between several MODBUS slave interfaces.

```
fnInitialiseMODBUS_port(0,
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                         0);         // initialise MODBUS serial interface - port 0
```

The MODBUS master, sending function codes to read or modify coils must be aware of the coil address locations. If it tries to address coils which are not defined, MODBUS *exceptions* are returned. The same is true of other invalid function requests, such as trying to read more coils than are available.

The *exception* handling is defined by the MODBUS specification as follows (see also the read coils state diagram, figure 11, in the specification).

- Function code not supported - *Exception Code 1*
- Quantity of requested output not between 1 and 2000 - *Exception Code 3*
- Requested address not in the available block, or the quantity exceeds the available address range - *Exception Code 2*
- Read could not be performed successfully - *Exception Code 4*

The following example MODBUS read coil illustrates a valid request (RTU format is used)

0x7f	0x01	0x00 – 0x0a	0x00 – 0x0b	0x49 – 0x59
Slave address	Function Code – Read Coils	Start add 10	Quantity 11	CRC

Since the start address is within the coil block range and requested quantity is also completely within the range this is a valid request and the MODBUS slave will answer with the requested data.

The next example MODBUS read coil illustrates an invalid request (RTU format is used)

0x7f	0x01	0x00 – 0x21	0x00 – 0x14	0x78 – 0x99
Slave address	Function Code – Read Coils	Start add 33	Quantity 20	CRC

Although the start address is within the coil block range, the requested quantity requires the end address to enclose the coil 52. This is outside of the defined range and so results in the exception 2 being returned.

The MODBUS slave has direct access to the coil values as held by the array `coils[]`. The size of this array is known and so the slave will never try to access outside of it.

The coils may not be const data blocks since they can also be modified by the MODBUS slave when it receives either Write Single Coil or Write Multiple Coil functions.

Reads can be handles without any user intervention if the coil values in the array are kept synchronised to system values. In some solutions the coil values in the coil array will be the real coil value in the system, but in other solutions the coil values in the array will be only representing other system values (eg. a power output state mapped to a certain coil bit).

In order to allow the user to synchronise coil values only when requested by a MODBUS master the `fnPreFunction()` call-back is used. This passes information including the function code, the address and the length as specified by the master and can be acted on as required; doing nothing, updating all content, or just updating the requested part of the content as required.

In the case of a write to coils, example with the function code 5 (Write Single Coil), the slave can also operate autonomously. That means that the MODBUS slave accesses and modifies the coil array, after which it allows the user to react to the changes by calling `fnPostFunction()` with information containing the function code, the address and the length as specified by the master. The user can again chose to either ignore the event, react to all states in the coil array, or else just to the ones which may have changed. In some cases the new coil states will be the system coil states directly or else the user will transform a state in the array to another element, such as a port output to control an external switch.

The following MODBUS single coil example illustrates a valid request (RTU format is used)

0x7f	0x05	0x00 – 0x0c	0xff – 0x00	0x49 – 0x59
Slave address	Function Code – Write single coil	Coil add 12	Value '1'	CRC

This will result in the third bit in the first byte of the coil array being set to '1'. This is because the address corresponds to the third coil from the start address of 10.

7.1. Example of the use of fnPreFunction()

```
// This routine is called by the MODBUS interface prior to a read
// It can be used to update the MODBUS tables if required
//
static int fnMODBUSPreFunction(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    switch (iType) {
    case PREPARE_COILS:           // coils are being read - update table values
        if ((modbus_rx_function->usElementAddress == COILS_START) &&
            (modbus_rx_function-> usElementLength == 16)) { // only allow reading of 16
                                                            // coils from start address
            coils[0] = 0xf0;      // update the values so latest states can be returned
            coils[1] = 0x0a;
        }
        else {
            return MODBUS_EXCEPTION_SLAVE_DEV_FAILURE;  // otherwise cause exception 4
        }
        break;
    default:
        break;
    }
    return 0;
}
```

This example shows special handling of the type `PREPARE_COILS`. This type is requested when the MODBUS master has requested that coils be read and the parameters `modbus_rx_function->usElementAddress` and `modbus_rx_function->usElementLength` refer to the starting address and the quantity of coils being read. The user call-back allows only reads of 16 coils from the start address of the coil table, otherwise it reports that the coils could not be read, causing the MODBUS slave driver to return exception 4. Normally, such restrictions would not be imposed by the user but this serves to illustrate both the basic operation and the control that the user has.

When a read of 16 coils, starting at the first coil address, is detected, the user code fills out the first two coil table entries with a fixed value, which will then be reported by the MODBUS slave driver to the requesting MODBUS master. This example assumes that the coils are saved as bytes in the local table, which however doesn't have to be the case as illustrated in the following chapter. Often it will not be necessary to update the coil table values in the `fnPreFunction()` since the values can be changed at any time previously to remain synchronised. Sometimes it may be more practical to synchronise them only on demand – eg. when they have to be read from a hardware port, which needs only to be updated when there is a need to inform of the instantaneous values.

7.2. Example of the use of fnPostFunction ()

The `fnPostFunction()` is called after the MODBUS slave driver has updated contents. The following shows the user interface reacting to particular coil changes:

```
static int fnMODBUSPostFunction(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    unsigned short usAddress = modbus_rx_function->usElementAddress;
    unsigned short usLength = modbus_rx_function->usElementLength;
    switch (iType) {
    case UPDATE_COILS:           // coils have been altered - react to changes if necessary
        {
            MODBUS_BITS_ELEMENT coil_bit;
            MODBUS_BITS_ELEMENT coil_element;
            unsigned short usTableAddress = (usAddress - COILS_START);
                                                              // referenced to the coil table
            coil_bit = (0x01 << usTableAddress%MODBUS_BITS_ELEMENT_WIDTH); // first coil bit
            coil_element = usTableAddress/MODBUS_BITS_ELEMENT_WIDTH;
                                                            // the first coil element location
            while (usLength != 0) {
                fnSetCoil(usAddress, (unsigned char)((coils[coil_element] & coil_bit) != 0));
                if (coil_bit & (0x1 << (MODBUS_BITS_ELEMENT_WIDTH - 1))) {
                    coil_bit = 0x01;
                    coil_element++;
                }
                else {
                    coil_bit <<= 1;
                }
                usAddress++;
                usLength--;
            }
        }
        break;
    }
    return 0;
}
```

The type `UPDATE_COILS` is called to indicate that coil values have been written to. `modbus_rx_function->usElementAddress` specifies the starting coil address and `modbus_rx_function-> usElementLength` the number of coils actually written to. The code calculates which coil storage elements and individual bits are involved and the sub-routine `fnSetCoil()` is then used to specifically react to certain coil changes. An example would be:

```
// Handle special coil controls
//
static void fnSetCoil(unsigned short usAddress, unsigned char ucState)
{
    switch (usAddress) {
    case 14:                                          // map this coil to a relay output
        if (ucState != 0) {
            RELAY1ON();
        }
        else {
            RELAY1OFF;
        }
        break;
    }
}
```

This example shows coil address 14 being dedicated to the control of a relay. Other coils are not supported but the routine could be extended to specifically handle every possible coil in a dedicated manner.

The example shows that the updated coil address block is signalled and the user can process the entire block. In some cases individual coil states may not have changed but this is not indicated directly. If the user prefers to only receive notification of coil changes, the project define `#define NOTIFY_ONLY_COIL_CHANGES` will result in the `fnPostFunction()` only being called on coil content change due to writes. In this case it will be called each time individual coils change state, with the `modbus_rx_function->usElementAddress` parameter set to the coil and the `modbus_rx_function->usElementLength` contains either 0 or 1 depending on the new coil state. This means that the `fnPostFuntion()` may be called multiple times for a single "Write Multiple Coils" function.

A corresponding `fnPostFunction()` for this case is illustrated below:

```
static int fnMODBUSPostFunction(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    switch (iType) {
    case UPDATE_COILS:           // coils have been altered - react to changes if necessary
        fnSetCoil(modbus_rx_function->usElementAddress,
                  (unsigned char) modbus_rx_function-> usElementLength);
                               // call specific coil handler
        break;
    }
    return 0;
}
```

There are three main differences.

1) The function is called *per changed coil* and not *per coil block*.
2) If there are no changes resulting from a coil write the user function will *not* be called at all.
3) The user doesn't need to map the address to a particular coil bit since the new coil state is passed directly.

7.3. fnPreFunction() and fnPostFunction () Message Routing

The Pre- and Post-Functions have the special ability to route any particular received Modbus message to any other MODBUS master port. This can be after already processing its content (for example after updating local values based on the function content) or instead of processing it locally. This allows routing based on particular function codes, particular element types, certain valid address ranges or even on particular data content.

To route the received message to another MODBUS port the user handler needs only to return the value `(MODBUS_APP_GATEWAY_FUNCTION – MODBUS port number to be routed to)`. For example

```
return (MODBUS_APP_GATEWAY_FUNCTION - 2);      // route to MODBUS port 2
```

For further details about gateway routing possibilities see the chapter "*Configuring Address-based Gateways*".

Note also that the user call back function (see the chapter "*User Defined Function Codes and Handling of Public Function Codes by the Application*") also has the ability to route messages received by a local MODBUS slave to other MODBUS ports based on further function codes and out-of-bounds addresses.

8. Overlaying Coils and Discretes with Registers

It is often practical to be able to access coils and discrete by not only their normal public functions, such as `Read Discrete Inputs (0x02)` or `Read Coils (0x01)`, but also using Register functions, such as `Read Input Register (0x04)` and `Read Holding register (0x03)`.

The µTasker MODBUS solution allows this to be achieved in an architectural independent manor as described in this section.

Figure 8-1 shows an array of coils which is configured to be constructed from an array of bytes. The coils are counted from right to left (LSB to MSB) according to the MODBUS convention. In addition, several standard registers (16 bit wide) are shown. These areas are totally independent.

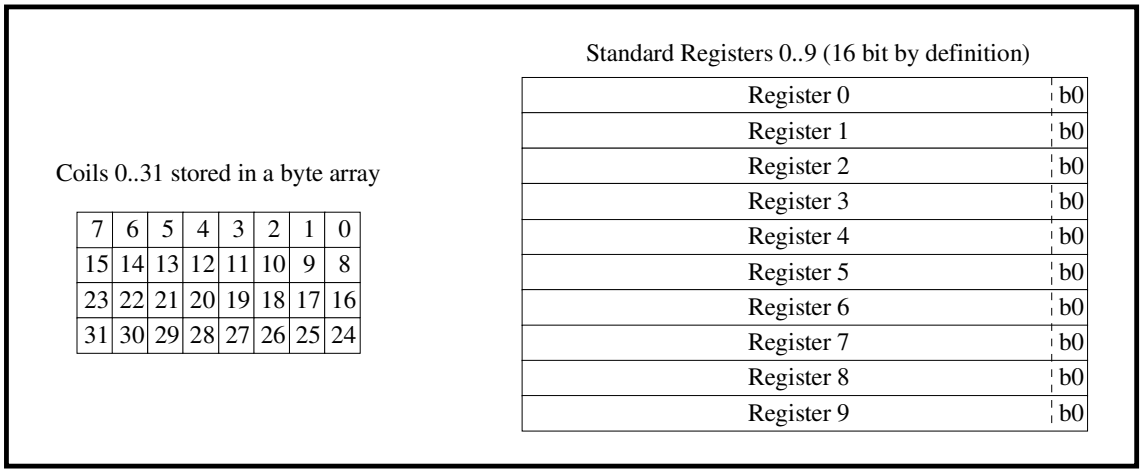

Figure 8-1 Independent coil and register tables using byte and short word storage elements respectively

Figure 8-2 shows the same standard registers sharing locations with the coils. This means that they are overlaid, occupying the same physical memory space.

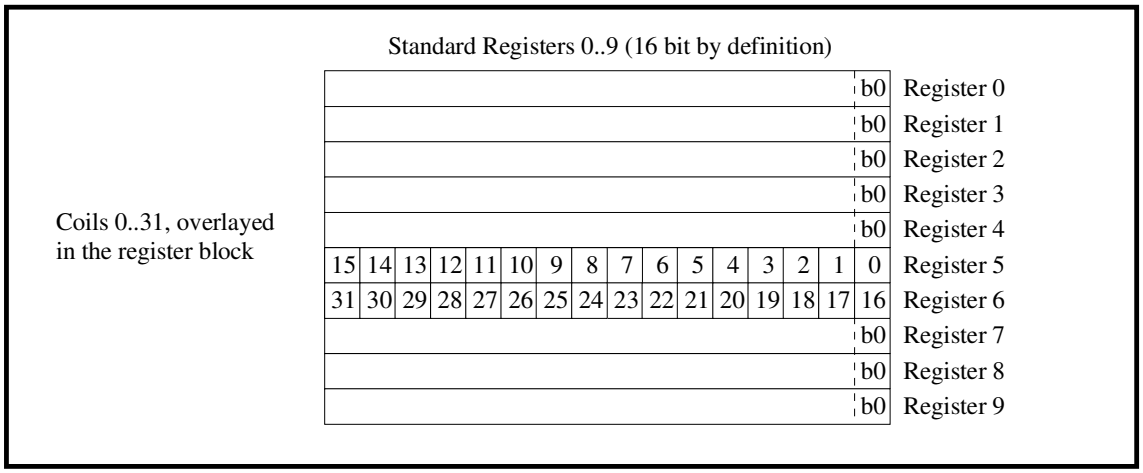

Figure 8-2 Overlaid coils and registers using short word storage elements

It is to be noted that this requires that the coils are allocated short word aligned locations so that they can overlay the short word length standard registers. In addition it is seen that the ordering of the coils in the array is still from right to left.

What is however not visible in this diagram is the fact that the actual byte storage of the coil bits is architectural dependent: big-endian will store them with coils 0..7 at an odd byte address and coils 8..15 at an even byte address; little-endian will store them with coils 0..7 at an even byte address and coils 8..15 at an odd byte address.

The user doesn't have to know the details since the accesses to the array may take place as short words, as defined, and the MODBUS slave interface automatically handles the details when accessing the register locations as coil types, irrespective of the architecture used.

As an example consider the coils 0..15 having the values
`0,0,0,1,0,0,1,0,0,1,0,0,1,0,0,0`.

MODBUS convention means that they will be stored in a short word as 0x1248, which means that a register read will return this same value. When the first 16 coils are read, the coil values returned will be 0x48 [coils 7..0], 0x12 [coils 15..9]. Of course it is also possible to read coils from 1..16, 2..17 etc. which means that the coil bits are simply shifted accordingly.

The user may chose the storage types of coils and discrete according to requirements. If no overlaying is required they can be stored as `unsigned char`. This is defined in `types.h` by setting `#define MODBUS_BITS_ELEMENT_WIDTH 8`. This simplifies the driver code slightly but doesn't guaranty that overlaying will operate on all architectures (*in fact it will operate correctly on a little-endian architecture as long as the coil block starts aligned to a short word address in the register block*).

To ensure that overlaying coils and registers will always operate correctly, the setting `#define MODBUS_BITS_ELEMENT_WIDTH 16` can be used. This ensures that the coil and discrete arrays operate with short word storage elements and that there is no architectural dependency in the results and should be enforced in a big-endian environment.

Figure 8-3 shows the setting `#define MODBUS_BITS_ELEMENT_WIDTH 32`, which can be used if coils are overlaid with extended register types (eg. Enron functions). In this case coil and discrete storage will automatically be in long word arrays and, again, all coil operations will automatically ensure that there are no architectural dependencies. In a little-endian environment the byte element storage is still possible as long as the coil block starts on a long word boundary.

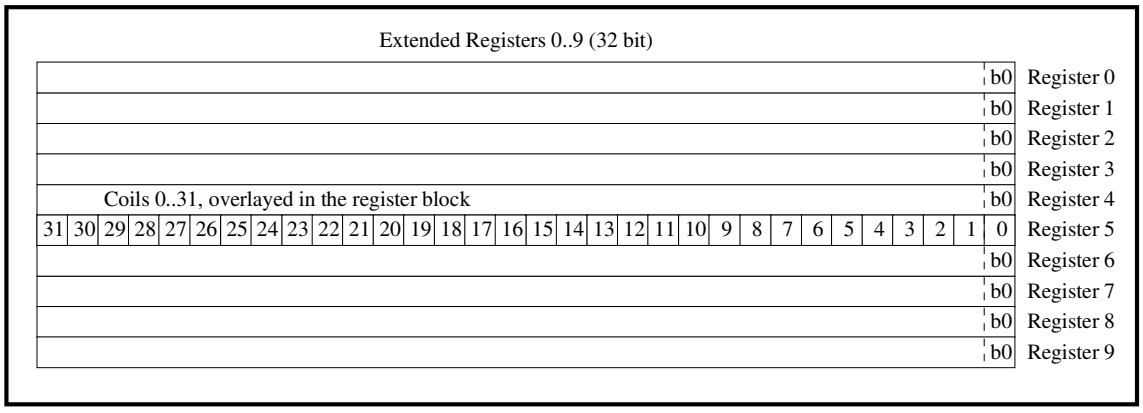

Figure 8-3 Overlaid coils and registers using long word storage elements

9. User Defined Function Codes and Handling of Public Function Codes by the Application

The MODBUS Application Protocol defines a number of functions such as Read Coil and Write Single register. These are known as public function codes since they are well defined, standardised and guaranteed to be unique. They occupy the range 1..65, 73..99 and 111..127.

Between the public function codes the ranges 65..72 and 100..110 are user defined function codes which may be used for user specific purposes. The use of these function codes is not guaranteed to be unique but are otherwise free in a proprietary system.

Function codes above 127 should generally not be used since they are reserved for use by legacy products.

A MODBUS slave can be defined an optional call-back function when the MOBUDS port is configured. This user call back can be used to handle all function codes which are not supported by the MODBUS package, whereby the supported functions is also configurable by removing all public functions which are not required to be handled automatically by the slave (the function codes to be left out are defined in `config.h`). The user call-back has the same prototype as all other call-backs:

```
int fnMODBUSuserFunction(int iType, MODBUS_RX_FUNCTION *modbus_rx_function);
```

The `MODBUS_RX_FUNCTION` pointer is to a struct containing all information about the received frame and a pointer to its data content, meaning that the user code can fully handle the frame as desired. In the case of function codes not supported in the MODBUS slave module the `iHandlingType` is `USER_RECEIVING_MODBUS_UNDEFINED_FUNCTION`.

- Should the user code not wish to handle the function code in question it can return `MODBUS_EXCEPTION_ILLEGAL_FUNCTION`, resulting in the MODBUS slave responding with this exception.
- If the user code handles the function code it should return 0 or, in cases of error, one of the return values
  `MODBUS_EXCEPTION_ILLEGAL_FUNCTION`,
  `MODBUS_EXCEPTION_ILLEGAL_DATA_ADD` ,
  `MODBUS_EXCEPTION_ILLEGAL_DATA_VALUE` or
  `MODBUS_EXCEPTION_SLAVE_DEV_FAILURE` as appropriate.
- Should the user not have entered a user call-back handler, the exception `MODBUS_EXCEPTION_ILLEGAL_FUNCTION` will always be returned automatically by the slave without the user being involved.

In addition to handling non-implemented function codes, the user call back can be used to handle codes acting on certain elements such as coils, registers etc. When the particular element is not declared in the user parameter table (it is configured with a zero pointer rather the address of the elements) the user call back will also be executed so that it can specifically handle this type. The value of `iHandlingType` in this case is `USER_RECEIVING_ALL_MODBUS_TYPE`. The user code must also fully handle the function code in this case and return 0, or an exception code as appropriate.

In some cases the user may wish to allow a specific address range of a certain function code to be handled automatically by the MODBUS slave but other ranges to be handled by the user call-back. In this case the user call back will be executed when the function code's address range doesn't correspond to the standard address range as defined by the element's parameters. In this situation the value of `iHandlingType` is `USER_RECEIVING_MISSED_RANGE`. Again the user handler should fully handle thus function code and return either 0, or an appropriate exception code.

Finally note that if the MODBUS port is declared without any parameters (zero pointer) all function codes for this MODBUS slave port will result in the user call-back handler being called, as if no function codes were supported in the slave module.

9.1. User Function Routing

In all cases above when the user call back is executed the user code can decide to route the received command/request to another MODBUS port. Depending on requirements, this can be after already processing its content (for example after updating local values based on the function content) or instead of processing it locally. This allows routing based on particular function codes, particular element types or on certain function code address ranges.

To route the received message to another MODBUS port the user handler needs only to return the value `(MODBUS_APP_GATEWAY_FUNCTION – MODBUS port number to be routed to)`. For example

```
return (MODBUS_APP_GATEWAY_FUNCTION - 2);      // route to MODBUS port 2
```

For further details about gateway routing possibilities see the chapter "*Configuring Address-based Gateways*".

Note also that the Pre- and Post-Functions (see the chapter "*Example of Configuration and Control of Coils*") also have the same ability to route messages received by a local MODBUS slave to other MODBUS ports.

10. MODBUS RTU Mode

MODBUS RTU serial mode of operation allows binary content to be sent and received. It is thus more efficient than ASCII mode since it doesn't have to send two 7-bit characters to represent one 8-bit value. Its framing operation is not based on control characters but instead on transmission timing. A frame starts as soon as a first character is transmitted and its end is recognised by the receiver when no more characters are detected during a period of 3.5 character periods. One character consists of always 11 bits (start-bit, 8 data-bits, a parity bit [usually even parity], plus one stop bit).

Using a baud-rate of 19'200 as reference, the transmission period of a single character is 573us. The termination of frame data on the bus is thus recognised when there is an idle period of longer than 2ms. During individual characters within a frame, idle periods of greater than 1.5 character periods are not allowed otherwise the frame is considered to be corrupted by the receiver and will be rejected.

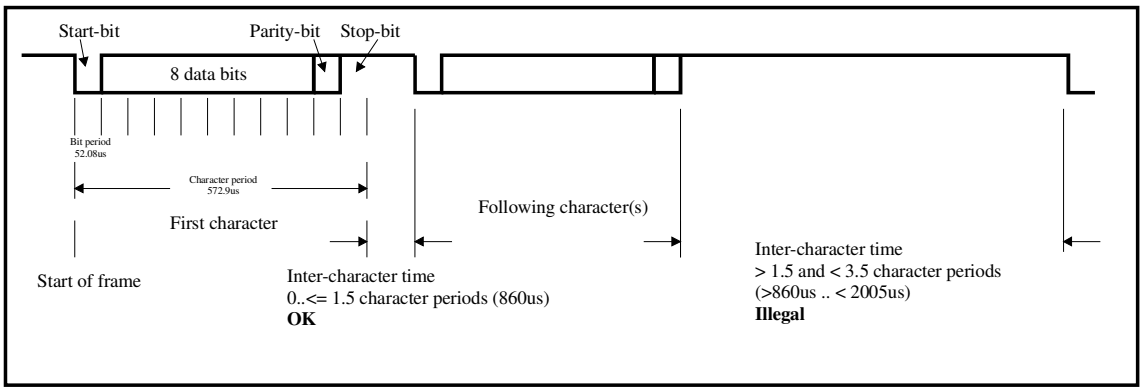

Figure 10-1 Character and illegal inter-character space timing at 19'200Baud reference

Figure 10-1 illustrates the timing of the 11 bit RTU mode character at 19'200 Baud. As long as the inter-character spacing of following characters remains less than or equal to 860us they are valid and so belong to the present frame being received. If a following character is detected after an inter-character period of greater than 860us but less that 2005us it is considered to be illegal and the frame rejected.

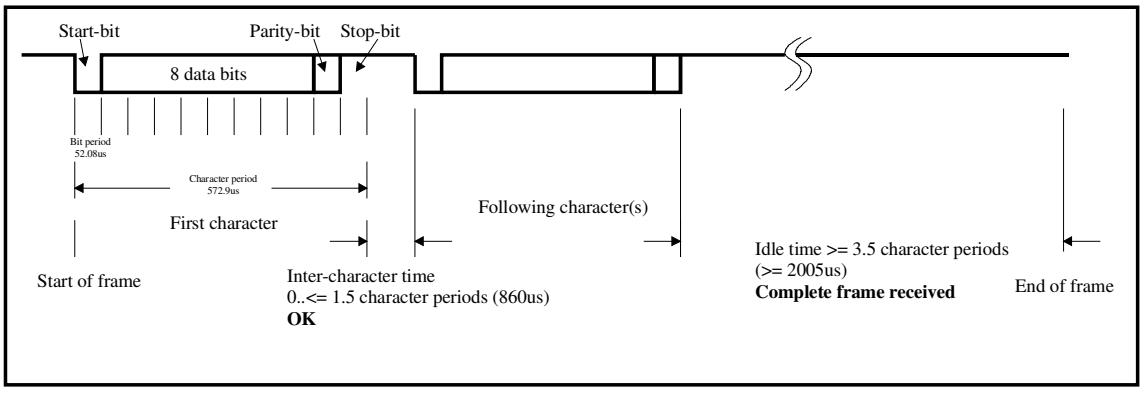

Figure 10-2 Complete RTU frame where end of frame is recognised by a 3.5 character period idle time

Figure 10-2 illustrates the timing of the 11 bit RTU mode character at 19'200 Baud in the case of a complete valid frame. When the receiver recognises the 3.5 character period idle time it treats the content of the received, valid frame.

For baud rates of greater than 19'200 the MODBUS specification recommends that fixed intervals of 750µs for the 1.5 character and 1'750µs for the 3.5 character periods are used by the receiver.

By activating the define `FAST_MODBUS_RTU` shorter times than the recommended fixed values are used for speeds greater than 19'200 Baud. In this case the inter-character space is calculated for the Baud rate used. This allows faster reaction times to RTU receptions – for example: 335µs at 115'200 Baud rather than 1'750µs. Its use may however need to be verified in the operating environment since it is also correspondingly more sensitive to small character spaces during frame reception.

The RTU mode of operation is specified in the serial MODBUS port configuration with the flag `MODBUS_MODE_RTU`. The `MODBUS_MODE_ASCII` would specify the alternative ASCII mode of operation.

11. RS485 Operation

It is usual to operate a multi-slave MODBUS on an RS485 bus. This can consist of a semi-duplex two wire electrical connection or a four wire full duplex connection. The MODBUS protocol doesn't however use the full duplex capability of such a bus.

When a device transmits on the RS485 bus it may need to drive its output transceiver. For this reason in the RS485 mode the RTS line is activated when transmitting and deactivated once the transmission has completed.

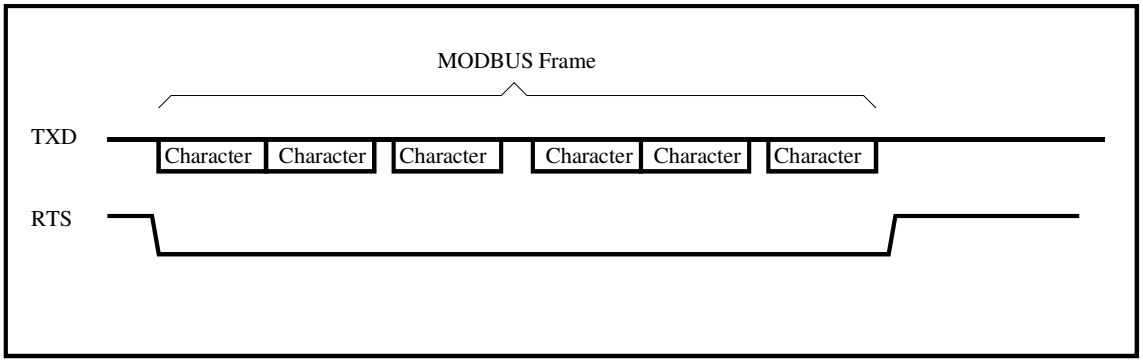

Figure 11-1 RTS control during a single MODBUS frame (showing negative logic RTS, which is hardware dependent and configurable)

As illustrated in figure 11-1 the RTS signal is asserted at the start of the first character of a MODBUS frame. The MODBUS frame consists of a number or characters which are each transmitted asynchronously. There can be a short gap between each character, although not longer than 1.5 character periods in the RTU mode. The RTS remains active until the end of the final stop bit of the last character in the frame, after which it is de-asserted as quickly as possible to avoid the bus being driven when the slave already starts sending back an answer. (A de-assertion of the RTS during the transmission of the final stop bit is generally acceptable since the idle state of the bus corresponds to the same state.)

- In the case of RTU operation the slave will process a received frame after a period of no less that 3.5 character intervals. This means that a slave will never answer within this time and the RTS control accuracy does not need to be very high.
- In the case of ASCII operation the slave detects the end of the MODBUS frame from its end characters. In this situation it can already handle the received frame as soon

as the last character in the frame has been received. It can thus also return an answer to a request much faster than in RTU mode. The requirement for RTS accuracy is thus higher in ASCII mode.

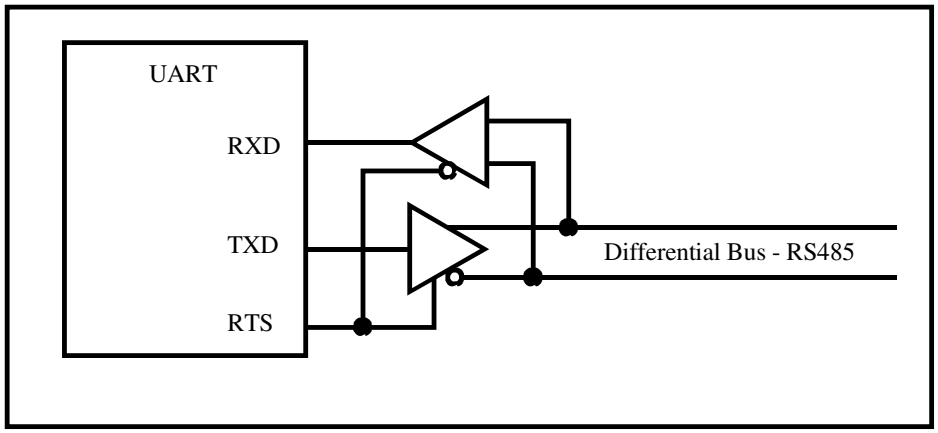

Figure 11-2 Standard RS485 half-duplex bus connection (RTS polarity may be configurable)

Figure 11-2 shows a typical circuit for driving a half-duplex RS485 transceiver. This assumes TX driving when RTS is high. The signal polarity is configured for each serial interface using either `MODBUS_RS485_NEGATIVE` or `MODBUS_RS485_POSITIVE` in the `MODBUS_PARS` configuration block as seen in the serial port configuration examples, whereby the polarity refers to the signal at the processor interface. This allows the correct signal to be generated for direct connection to the RS485 transceiver hardware.

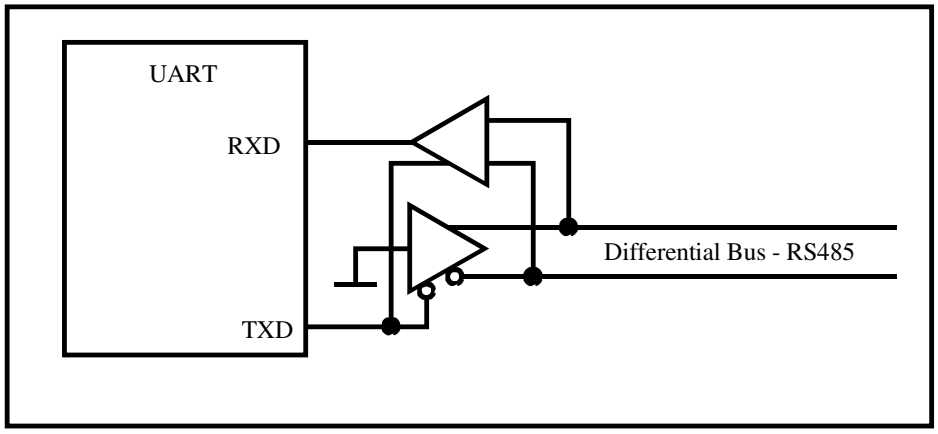

Figure 10-3 Alternate method for connecting RS485 half-duplex transceiver without RTS control

Figure 10-3 shows an alternative circuit for avoiding RTS control. The TX input is connected to 0V and the transceiver is controlled exclusively by the TXD line. When the TXD output is idle (‘1’) the receiver is enabled and the TX output is not driven. When the TXD output a ‘0’ (eg. the start bit) the receiver is disabled and the transmitter drives a ‘0’ state to the differential bus. Since the ‘1’ state is never driven to the bus it is assumed that the bus defaults to this state.

The RS485 mode of operation is specified in the serial MODBUS port configuration with the flag `MODBUS_RS485`. The serial driver must be configured to support the RTS control by activating `SUPPORT_HW_FLOW` in `app_hw_xxxx.h`. Processors which do not have RTS signals integrated into their UARTs configure general purpose outputs to perform the

function. The pin control is in this case also defined in `app_hw_xxxx.h`. See appendix B for configuration examples of all supported processors

.

12. Configuring a MODBUS TCP Slave Device

The application layer MODBUS protocol is the same whether a slave is connected via a serial connection or Ethernet. This means that the details about the interaction between the user interface and the MODBUS data is identical to the serial case.

Configuring the user interface for MODBUS TCP slave operation is very similar to configuring the serial MODBUS user interface:

```
fnInitialiseMODBUS_port(2,
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                        0);
```

The MODBUS TCP slave ports follow the MODBUS serial ports (when used). Assuming two MODBUS serial as in the previous examples the first MODBUS TCP slave has the MODBUS port number 2.

```
fnInitialiseMODBUS_port(3,
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                        0);
```

A second MODBUS TCP slave port, in this case sharing the parameters and call-backs, foolws as MODBUS TCP slave port 3.

These examples shows two TCP ports being defined whereby their mode and TCP port numbers are defined in the parameter configuration table

```
...
{
    MODBUS_TCP_SLAVE_PORT,                      // MODBUS listener mode - TCP slave port 2
    MODBUS_TCP_SLAVE_PORT,                      // MODBUS listener mode - TCP slave port 3
},
{
    MODBUS_TCP_PORT,       // MODBUS listener port number - standard - TCP slave port 2
    4321,                  // MODBUS listener port number - user defined - TCP slave port 3
},
{
   (2*60),     // MODBUS listener port number - 2 minute TCP idle timeout - TCP slave port 2
   INFINITE_TIMEOUT,// MODBUS listener port number - no connection timeout - TCP slave port 3
},
...
```

The first TCP port is defined to listen on the standard MODBUS port 502. The second is a user defined port 4321. The actual number of TCP ports which are foreseen is defined by

MODBUS_TCP_SERVERS in config.h. For each TCP slave there should be a corresponding entry in the parameter configuration table.

The example is using the same set of MODBUS data tables but more likely two different sets would be used for the two port access points.

The number of socket sessions for each port is defined in config.h. For example:

```
#define MODBUS_SOCKETS_0       5    // number of sessions for first
#define MODBUS_SOCKETS_1       3    // number of sessions for second
```

The MODBUS TCP slave port 2 (the first TCP slave) can thus have simultaneous connections from 5 different TCP masters on its TCP port. The MODBUS TCP slave port 3 (the second TCP slave) supports up to 3.

The following diagram shows a MODBUS TCP slave with two ports addresses, each with its own set of independent data.

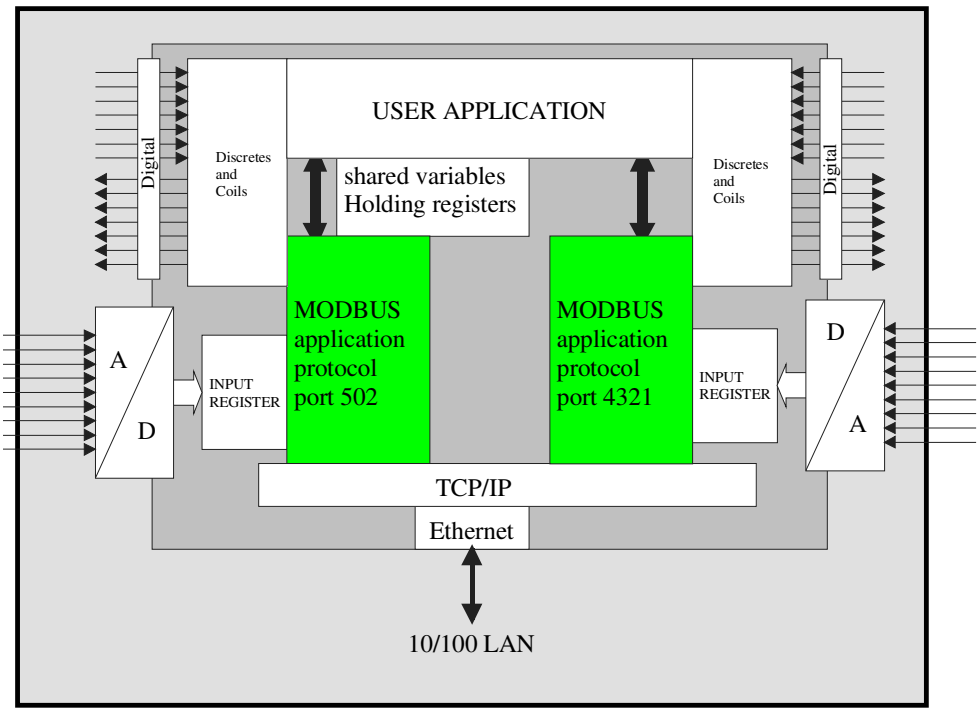

Figure 12-1: A typical MODBUS TCP slave showing various inputs and outputs which can be accessed and controlled via the Ethernet MODBUS connections

12.1. Multiple TCP Slaves on a Single TCP Socket

A MODBUS TCP slave is generally addressed by the MODBUS_TCP_NON_SIGNIFICANT_UNIT_IDENTIFIER (0xff or 255). If performing a serial gateway function, as discussed in the following chapter, all other addresses will be routed to its corresponding serial MODBUS port (UART).

A MODBUS TCP slave port can also be configured to perform multiple slave functions on a number of additional slave addresses. This operation is equivalent to multiple serial slaves on a single interface as described earlier in this document, as is its configuration. When the option MODBUS_SHARED_TCP_INTERFACES is set to a non-zero value it allows this many shared TCP slaves to be configured. Each slave will then respond on its own unique address and can have its own set of parameters and application call-backs.

A matching shared MOBUS slave address has priority over a possible gateway routing function on this address. Broadcast functions will be handled by the main slave on the non-significant unit identifier address and then by each shared slave in the order that these were configured. The broadcast will also be routed to any MODBUS serial or MODBUS TCP interfaces in accordance with the routing configuration. All slaves share the main TCP socket configuration.

```
fnInitialiseMODBUS_port(2, &modbus_configuration_1, &modbus_slave_callbacks_1, 0);
                                // initialise MODBUS TCP interface on first TCP port
#if defined MODBUS_SHARED_TCP_INTERFACES > 0
    fnShareMODBUS_port(2, &modbus_configuration_2, &modbus_slave_callbacks_2, 254);
                                // additional MODBUS slave sharing port 2 TCP - address 254
    fnShareMODBUS_port(2, 0, &modbus_slave_callbacks_3, 253);
                                // additional MODBUS slave sharing port 2 TCP - address 253
#endif
```

The define MODBUS_SHARED_TCP_INTERFACES is set to the total amount of shared slaves (the value 2 is adequate for the example) in the system. The shared slaves can be added to any existing MODBUS TCP slave available and any number can be added as long as the total does not exceed the MODBUS_SHARED_TCP_INTERFACES limit. As the example shows, each shared TCP slave can have its own set of parameters and call-backs.

The address given to each shared slave is passed to the `fnShareMODBUS_port()`. These values could also be added to the main uParameterSystem parameter block to make them user configurable. The MODBUS TCP slave main port uses always the slave address 255. In the example, the shared slaves are then defined addresses 254 and 253. *No two slaves on the MODBUS TCP port should have the same address.*

Figure 11-2 illustrates the case according to the example configuration. Although there is only one TCP socket the MODBUS master can communicate with multiple MODBUS slaves. Each slave has its own unique call-backs allowing handling MODBUS requests and commands to each slave by its own application. Slave addresses 255 and 254 have their own parameter set but the slave address 253 has been configured to have none (but could have its own, or share with other slaves).

Since the TCP socket supports multiple sessions, several TCP MODBUS masters can communicate with the TCP slaves. Although not shown in figure 12-2 the MODBUS TCP slave could also have a gateway function to route to either a dedicated UART or to multiple UARTs and multiple TCP bridges as described later in the chapter about address-based gateways.

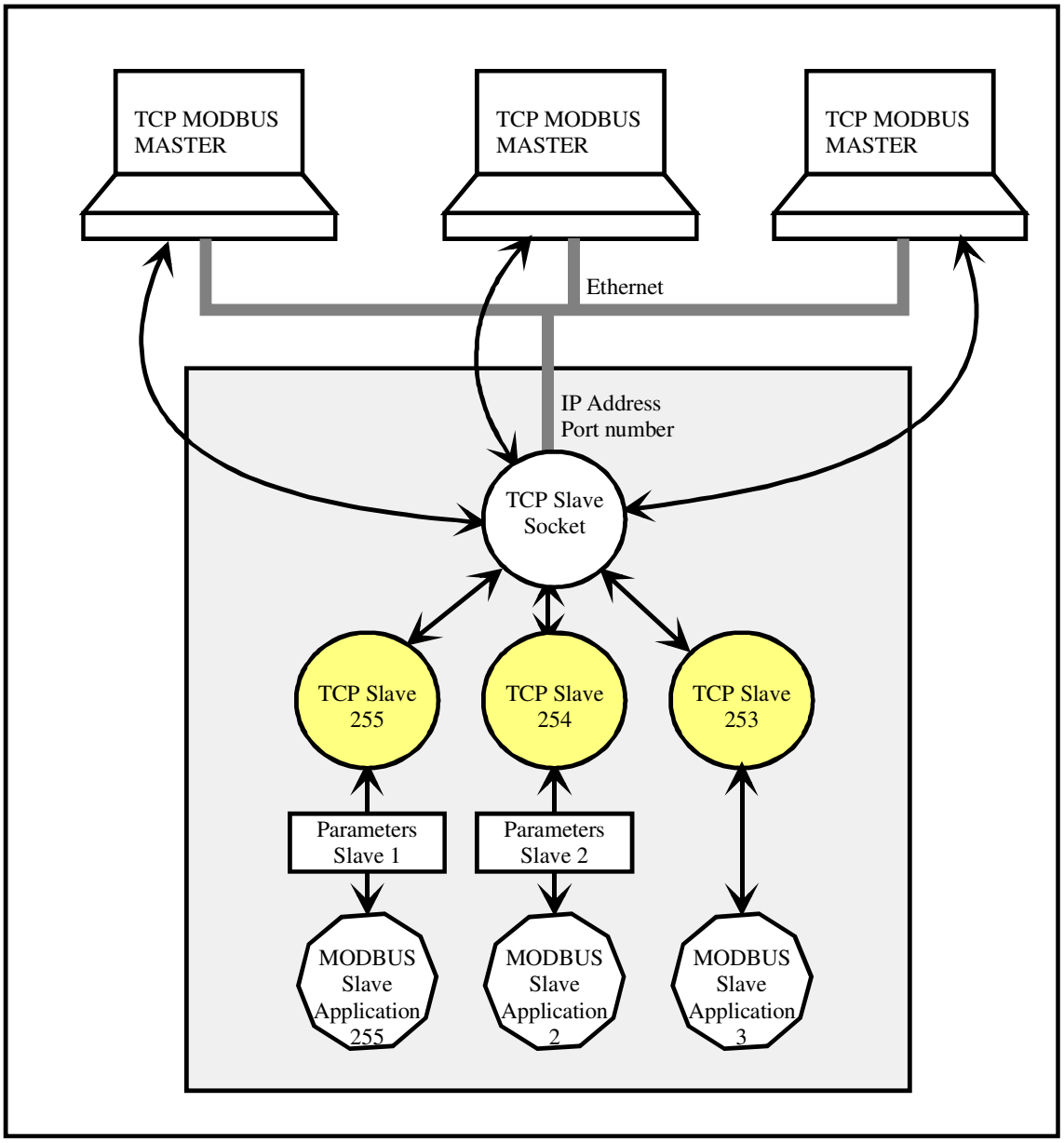

Figure 12-2: Multiple, shared MODBUS TCP slaves on a single TCP socket

13. Configuring MODBUS TCP to UART gateways

Whether a listening MODBUS TCP interface is a slave or a gateway, or both, depends on how it is configured. In the previous section we saw a TCP slave being configured. The following extends the functionality of the first TCP slave to perform gateway operation on received MODBUS receptions which do not address the slave itself.

```
...
 (MODBUS_TCP_SLAVE_PORT | MODBUS_TCP_SERIAL_GATEWAY),   // listener mode - TCP slave port 2
...
fnInitialiseMODBUS_port(2,
                        &modbus_configuration,
                        &modbus_slave_callbacks,
                        0);
```

Note that the gateway'ed MODBUS serial ports MUST be configured as `MODBUS_MASTER` in this case! MODBUS master configuration is discussed later in the document.

The following removes the slave functionality of the second TCP port, whereby the configuration and call-backs are also removed. It is however given a gateway function.

```
...
 (MODBUS_TCP_SERIAL_GATEWAY),                          // listener mode - TCP slave port 3
...
fnInitialiseMODBUS_port(3,
                        0,
                        0,
                        0);
```

All MODBUS TCP frames which are received this MODBUS TCP slave will be passed on to their corresponding MODBUS serial port.

The first MODBUS TCP slave has a set of slave configuration parameters and so can also be assessed as a slave device, rather than purely performing the gateway function. The local MODBUS TCP slave is accessed when the slave address is 0xff (the so called 'non-significant unit identifier'). All other MODBUS slave addresses cause the message to be routed to the defined MODBUS serial interface to address the corresponding slave address on the serial MODBUS bus. *As described in chapter 3, the MODBUS TCP slave port 2 will pass all gateway'ed frames to the MODBUS serial port 0*.

The following diagram shows a MODBUS slave with two TCP ports acting as gateways to two MODBUS serial connections. The main TCP port (502) also has local MODBUS TCP slave functionality. *As described in chapter 3, the MODBUS TCP slave port 3 will pass all gateway'ed frames to the MODBUS serial port 1.*

Figure 13.1 illustrates the complete functionality.

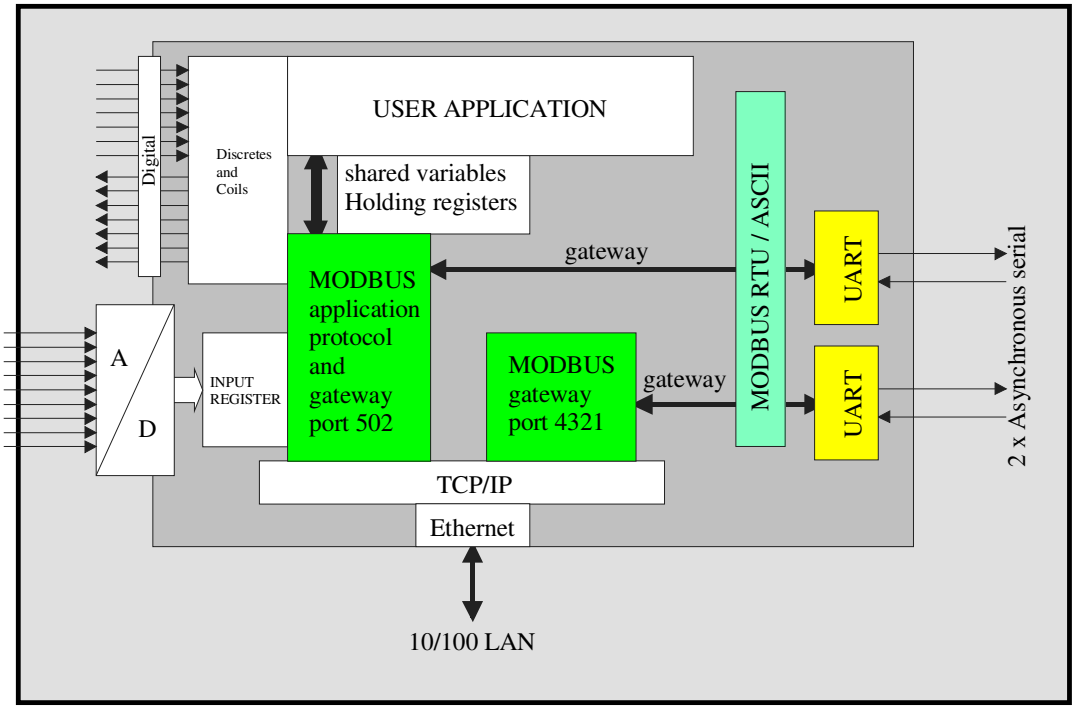

Figure 13-1: A MODBUS TCP slave showing dual gateway functionality, with additional local slave access on the 502 port. The MODBUS serial ports are both in master mode.

14. Configuring Address-based Gateways

Although the TCP/UART gateway may be the most common gateway function, the µTasker MODBUS module supports further highly flexible routing capabilities. A few such configurations are described here and can be used as base for various other combinations as required.

Section 14.3 shows in addition how it is possible to modify the slave address during the gateway process.

14.1. Serial to Serial MODBUS Bridge

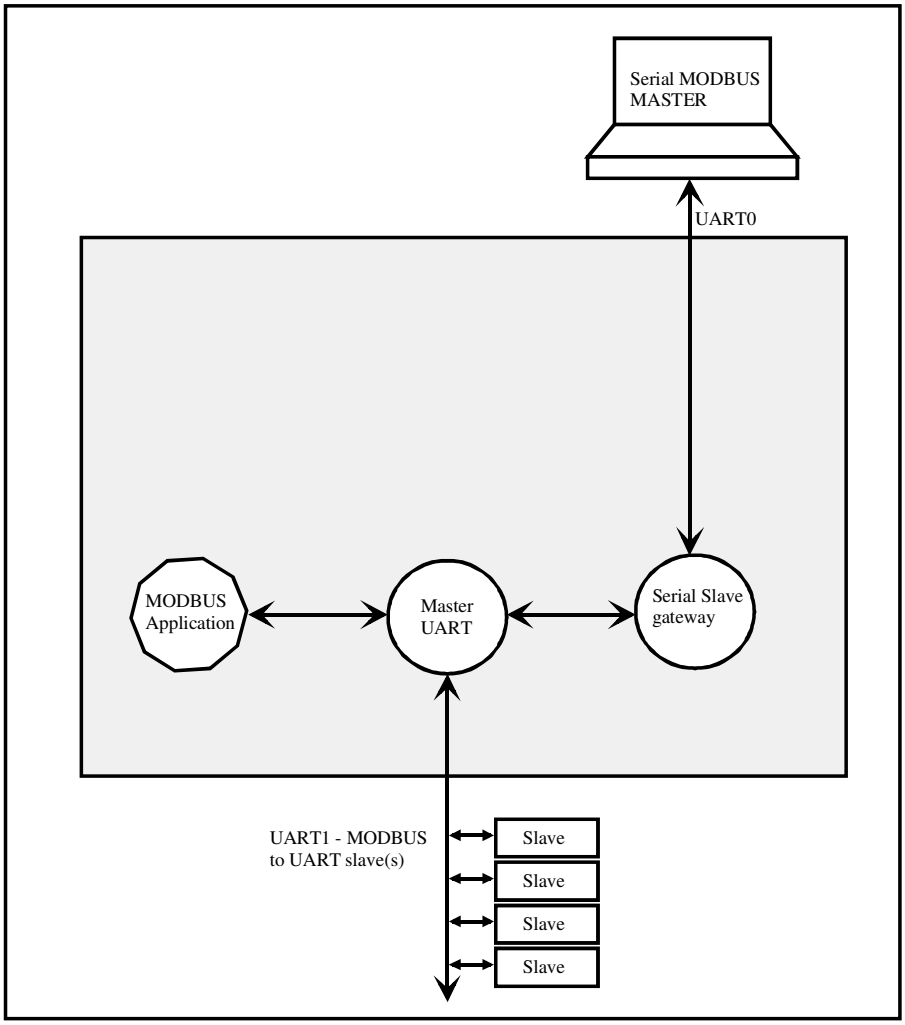

Figure 14-1: A MODBUS Serial Bridge

Figure 14-1 shows an application with two serial interfaces. The MODBUS application has defined a MODBUS serial master on UART1 which it can use to poll slave devices on the bus connected to that interface. This represents a normal MODBUS master function.

It has however additionally defined a MODBUS serial slave gateway on UART0. The purpose of this gateway is to allow an external MODBUS master to also access the MODBUS bus on UART1 for its own use. This results in there being essentially two masters controlling the bus on UART1. The two UARTs can however have different configurations (speed, mode, etc.). In this example the external MODBUS master is connected in a point-to-point fashion and could use RS232, whereas RS485 will usually be used on the MODBUS bus on UART1.

The serial slave gateway can either perform the gateway to all slave addresses or only to defined addresses. In addition it can have its own slave address for local access to its own parameters.

Before showing the way that this is configured, the following figure illustrates how the bridge can also operate as an address aperture to a MODBUS bus controlled by the external MODBUS master. *Note that the local MODBUS application doesn't have access to the MODBUS bus of the external MODBUS master*.

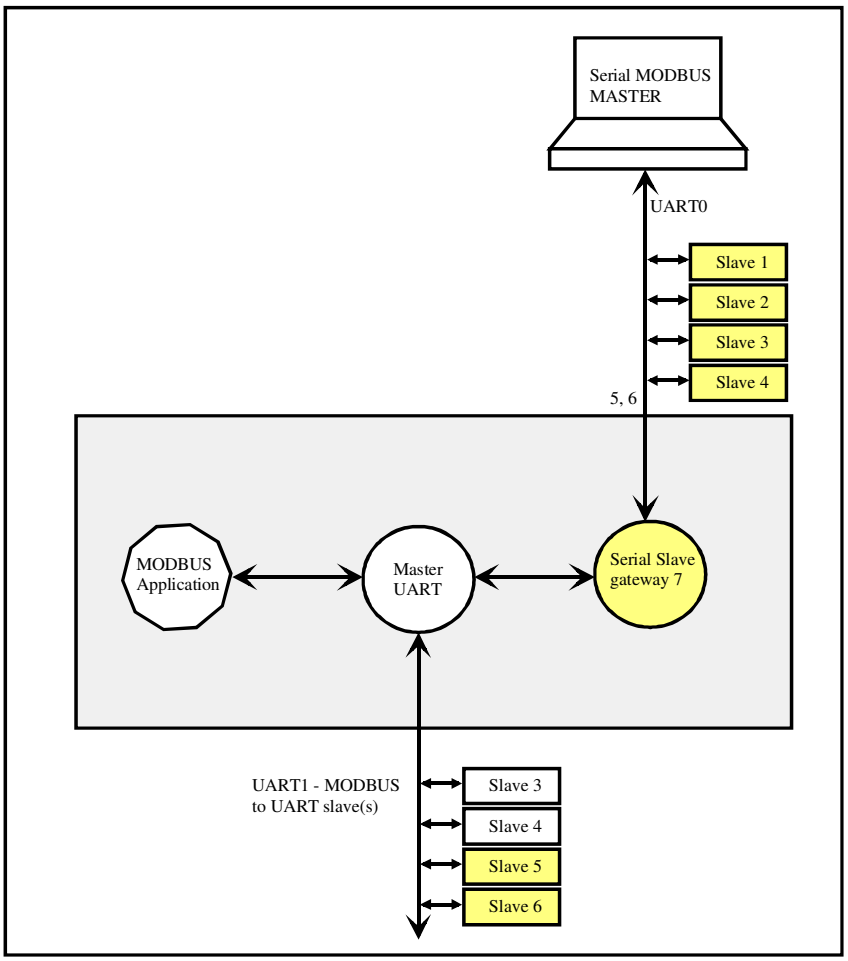

Figure 14-2: A MODBUS Serial Bridge acting as an address aperture

In figure 14-2 the serial slave gateway has been configured with its own slave address (7). It ignores slave addresses below 5 so that the external MODBUS master can access the devices on its local bus. Any higher addresses will be gateway'ed to UART1, meaning that the external MODBUS master also has access to remote slaves on the second bus through the defined address aperture.

The access addresses can be summarised as:

- 1..4 Local bus
- 5..6 Remote bus on UART1
- 7 Serial gateway slave address

The address range not covered in the example above can be configuration to allow further slave addresses to be routed to different UARTs or to MODBUS TCP connections, making it extremely flexible in operation. Since these are not required in the example, further slave addresses can also be configured to be ignored.

The following show the complete configuration for the case as illustrated in figure 14-2:

```
static const MODBUS_PARS cMODBUS_default = {
    MODBUS_PAR_BLOCK_VERSION,
    {
        7,                                                  // slave address - serial port 0
        0,                                                  // master - serial port 1
    },
    {
        (CHAR_8 + RS232_EVEN_PARITY + ONE_STOP + CHAR_MODE),  // configuration - serial port 0
        (CHAR_8 + RS232_EVEN_PARITY + ONE_STOP + CHAR_MODE),  // configuration - serial port 1
    },
    {
        SERIAL_BAUD_19200,                                  // baud rate - serial port 0
        SERIAL_BAUD_19200,                                  // baud rate - serial port 1
    },
    {
    (MODBUS_MODE_ASCII | MODBUS_SERIAL_SLAVE | MODBUS_SERIAL_GATEWAY | MODBUS_RS485_NEGATIVE),
                                                            // mode - serial port 0
    (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_POSITIVE), // mode - serial port 1
    },
    {
        (DELAY_LIMIT)(1*SEC),   // max. inter-character delay in ASCII mode - serial port 0
        (DELAY_LIMIT)(*SEC),    // max. inter-character delay in ASCII mode - serial port 1
    },
    {
        0x0a,                               // ASCII mode line feed character - serial port 0
        0x0a,                               // ASCII mode line feed character - serial port 1
    },
    {
        (DELAY_LIMIT)(0*SEC),               // MODBUS master timeout - serial port 0
        (DELAY_LIMIT)(1.5*SEC),             // MODBUS master timeout - serial port 1
    },

    { 'u', 'T', 'a', 's', 'k', 'e', 'r', '-', 'M', 'O', 'D', 'B', 'U', 'S', '-',
                                                              's', 'l','a','v','e' },
};
```

When the MODBUS ports are initialised the slave gateway port is passed parameters so that it can perform its local slave function (on address 7). It is also passed a master call back function which will be responsible for the gateway operation. This allows the gateway to first look at the frame content if it desires and change routing decisions on a request/command basis if necessary.

The master port initialisation is passed its own master call back function which is used for handling responses from its own requests and command to external MODBUS slaves.

```
    fnInitialiseMODBUS_port(0,
                            &modbus_configuration,
                            &modbus_slave_callbacks,
                            fnMODBUSgateway);   // initialise MODBUS serial interface - port 0
    fnInitialiseMODBUS_port(1,
                            0,
                            0,
                            fnMODBUSmaster);    // initialise MODBUS serial interface - port 1
```

The gateway call back is responsible for passing on requests and commands received from the external master to the defined destination. It can thus also count messages or even analyse their content and chance routing decisions as desired. Generally it will however simply pass the messages on to the internal µTasker MODBUS router.

```
static int fnMODBUSgateway (int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    if (iType == SERIAL_ROUTE_FROM_SLAVE) {
        return fnMODBUS_route(iType, modbus_rx_function, (MODBUS_ROUTE *)routing_table);
    }
    return 0;
}
```

Routing is based on slave addresses and controlled by a routing table which the gateway call- back passes. The routing strategy can thus also be changed during operation if the table is modified, or a different one is passed.

```
static const MODBUS_ROUTE routing_table[] = {
    {0xff,       0x04},                         // ignore addresses up to 4
    {1,          0x06},                         // route addresses 5 to 6 to MODBUS port 1
    {0xff,       0xff}                          // ignore all other addresses
};// port route, last address in block
```

The routing table is very simple. It states to which MODBUS port the slave addresses starting from the end of the last block up to the end of the entry block are to be routed to. A value of 0xff as port means that it is to be ignored. The block address range starts at 0 for the first entry and ends when 0xff is reached.

The gateway call-back is only used to handle the routing *to* the MODBUS master. The response from the remote MODBUS slave will automatically be routed back to the orignal source.

If the serial gateway slave were to have no slave functionality of its own (*address 7*) its configuration simply changes to

```
...
 (MODBUS_MODE_ASCII /*| MODBUS_SERIAL_SLAVE*/ | MODBUS_SERIAL_GATEWAY | MODBUS_RS485_NEGATIVE)
...
```

If it should route all messages it receives from the external MODBUS master, its routing table becomes simply

```
static const MODBUS_ROUTE routing_table[] = {
    {1,          0xff},                          // route all addresses to MODBUS port 1
};// port route, last address in block
```

Note that the user can also cause particular messages to be routed to other MODBUS master interfaces by returning `(MODBUS_APP_GATEWAY_FUNCTION – MODBUS port number to be routed to)`. For example

```
return (MODBUS_APP_GATEWAY_FUNCTION - 2);      // route to MODBUS port 2
```

This allows additional control in special cases which may not be handled fully by using the routing table alone as show in the following example. The standard routing table is used for all none-matching criteria but some special cases are determined in the call-back:

```
static int fnMODBUSgateway (int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    if (iType == SERIAL_ROUTE_FROM_SLAVE) {
        switch (modbus_rx_function->ucFunctionCode) {
        case MODBUS_WRITE_SINGLE_COIL:
            return (MODBUS_APP_GATEWAY_FUNCTION - 0); // route this function code to 0
        case MODBUS_WRITE_MULTIPLE_COILS:
            return(MODBUS_APP_GATEWAY_FUNCTION - 1);  // route this function code to 1
        }
        return fnMODBUS_route(iType, modbus_rx_function, (MODBUS_ROUTE *)routing_table);
    }
    return 0;
}
```

14.2. TCP to Serial MODBUS Bridge

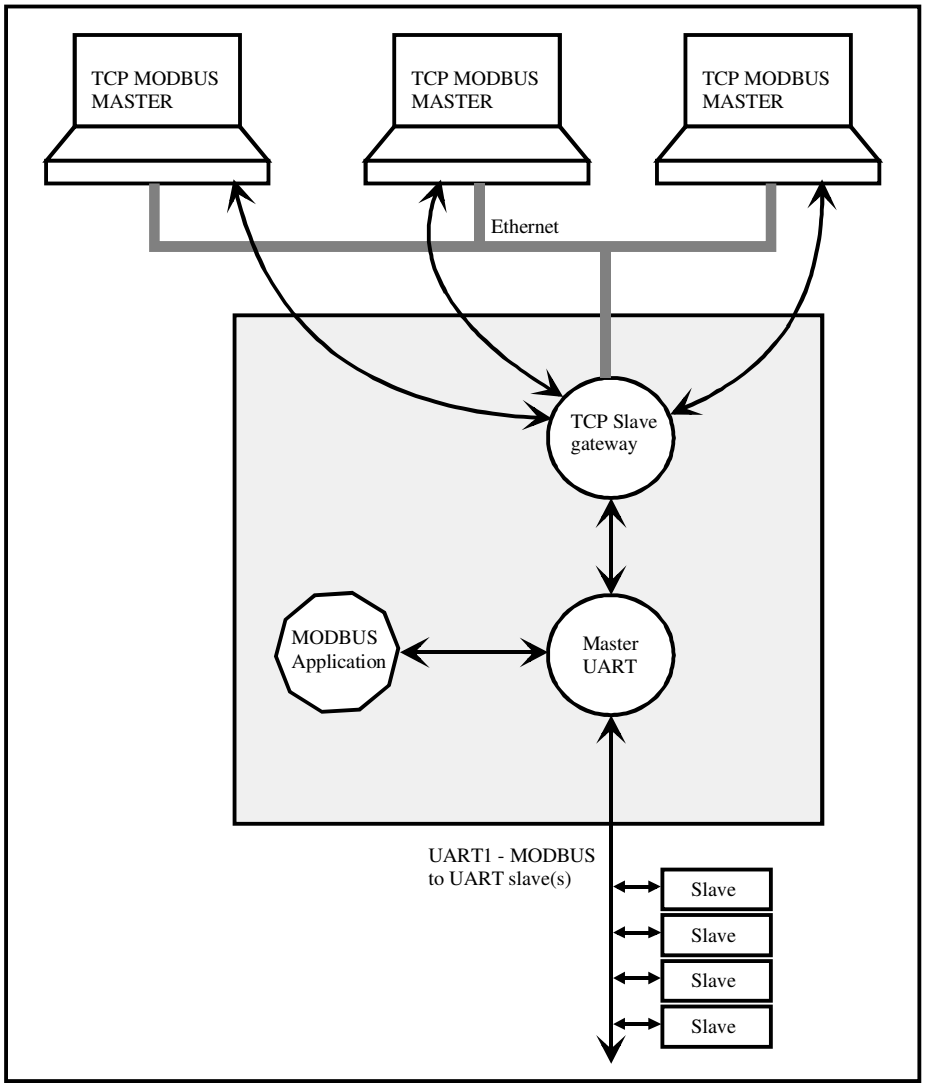

Figure 14-3: A MODBUS TCP/Serial Bridge

Figure 14-3 illustrates a MODBUS TCP to MODBUS serial bridge. The TCP slave gateway is a listener on either the MODBUS port 502 or any other user-defined port. Remote MODBUS TCP masters can establish TCP connections with the TCP slave gateway in order to then communicate with remote slaves on the UART1 connection. The number of remote TCP masters supported at the same time depends on the number of sessions configured for the TCP slave gateway.

As discussed in section 3.1 it is also possible to configure this type of gateway routing to a single fixed serial MODBUS port without using any slave address based routing.

Figure 14-4 thus shows a more advanced configuration where two serial MODBUS ports are available and the remote MODBUS TCP masters can access both serial connections based on slave addressing.

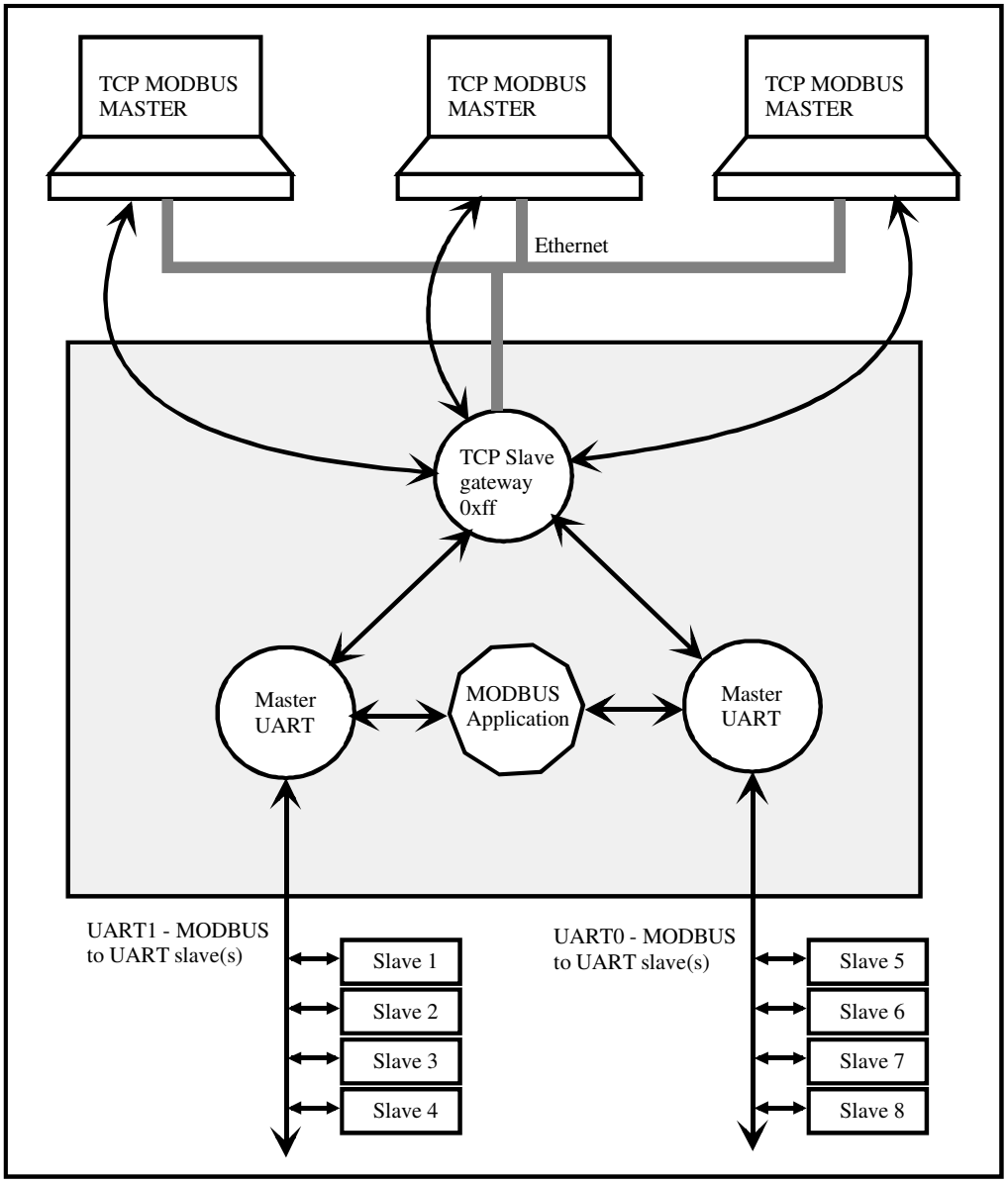

Figure 14-4: A MODBUS TCP/Serial Bridge with two serial MODBUS buses

It is now possible for a remote MODBUS TCP master to access both remote MODBUS serial buses on UARTs 0 and 1. As in the case of the serial to serial gateway case all routing is based on the slave address. The TCP MODBUS master doesn't need to know on which UART the destination device is connected and can, in this example, send requests or commands to any slave address from 1 to 8. Messages for slaves 1 to 4 are routed to UART1 and messages to slaves 5 to 8 are routed to UART0.

The TCP slave gateway can also have its own slave address, which is always 0xff (the non-significant unit identifier address). A TCP slave gateway with the flags `MODBUS_TCP_SERIAL_GATEWAY` and `MODBUS_TCP_SLAVE_PORT` will both route messages and respond to the non-significant unit identifier address, whereas a TCP slave gateway with only the flag `MODBUS_TCP_SERIAL_GATEWAY` will route all message and ignore the 0xff address.

Broadcasts received by the MODBUS TCP slave will be passed on to both MODBUS serial ports.

The configuration for the MODBUS TCP/Serial bridge as illustrated in figure 14-4 is shown below:

```
static const MODBUS_PARS cMODBUS_default = {
    MODBUS_PAR_BLOCK_VERSION,
    {
        0,                                                  // slave address - serial port 0
        1,                                                  // slave address - serial port 1
    },
    {
        (CHAR_8 + NO_PARITY + TWO_STOP + CHAR_MODE),        // serial port 0
        (CHAR_7 + EVEN_PARITY + ONE_STOP + CHAR_MODE),      // serial port 1
    },
    {
        SERIAL_BAUD_115200,                                 // baud rate of serial port 0
        SERIAL_BAUD_19200,                                  // baud rate of serial port 1
    },
    {
        (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_NEGATIVE),// RTU mode serial 0
        (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_POSITIVE), // ASCII serial 1
    },
    {
        (DELAY_LIMIT)(1*SEC),        // inter-character ASCII mode - serial port 0
        (DELAY_LIMIT)(0.2*SEC),      // inter-character ASCII mode - serial port 1
    },
    {
        0x0a,                        // ASCII mode line feed character - serial port 0
        0x0a,                        // ASCII mode line feed character - serial port 1
    },
    {
        (DELAY_LIMIT)(0.2*SEC),      // MODBUS master timeout - serial port 0
        (DELAY_LIMIT)(0.5*SEC),      // MODBUS master timeout - serial port 1
    },

    { 'u', 'T', 'a', 's', 'k', 'e', 'r', '-', 'M', 'O', 'D', 'B', 'U', 'S', '-',
                                                              's', 'l','a','v','e' },
    {
        (MODBUS_TCP_SERIAL_GATEWAY | MODBUS_TCP_SLAVE_PORT),  // MODBUS TCP slave port 2
    },
    {
        MODBUS_TCP_PORT,             // MODBUS listener port number- TCP slave port 2
    },
    {
        (5*60),                      // TCP idle time out
    },
    #endif
};
```

The MODBUS ports are initialised with:

```
    fnInitialiseMODBUS_port(0,
                            0,
                            0,
                            fnMODBUSmaster);   // initialise MODBUS serial interface - port 0
    fnInitialiseMODBUS_port(1,
                            0,
                            0,
                            fnMODBUSmaster);   // initialise MODBUS serial interface - port 1
    fnInitialiseMODBUS_port(2,
                            &modbus_configuration,
                            &modbus_slave_callbacks,
                            fnMODBUSgateway);  // initialise MODBUS TCP slave gateway
```

The gateway call back associated with the MODBUS TCP slave gateway is defined as follows:

```
static int fnMODBUSgateway(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    switch (iType) {
    case TCP_ROUTE_FROM_SLAVE:   // TCP frame from MODBUS TCP slave gateway has been received
        return fnMODBUS_route(iType, modbus_rx_function, (MODBUS_ROUTE *)routing_table_2);
                                                    // route the frame to a local MODBUS master
    }
}
```

The routing table is defined to route the address ranges to the appropriate MODBUS serial ports:

```
static const MODBUS_ROUTE routing_table_2[] = {
    {1, 0x04},     // all slave addresses up to 0x4 are routed to MODBUS port 1 (UART1)
    {0, 0x08},     // following slave addresses up to 0x8 are routed to MODBUS port 0 (UART0)
    {0xff, 0xff}   // any further addresses are ignored
};
```

14.3. Mapping Slave Addresses in a Gateway Transfer

During the gateway routing process two slave addresses is used. The second address is set by the MODBUS module per default to be the same as the slave address in the received message. If nothing else if performed it means that message received for slave address 9, for example, will be passed on to the gateway for the remote slave address 9; the reverse routed response will also be from the slave address 9.

In some situations is may be necessary to be change the slave address to which the message is sent to on the gateway'd MODBUS port. For example, a message received for slave address 9 is passed on to another MODBUS port but sent to the remote slave address 23. The response returned from the remote slave with address 23 is routed back to the requesting master with the slave address information 9.

The following code, added to any MODBUS application call-back, shows how this is achieved:

```
    if (modbus_rx_function->ucSourceAddress == 9) {       // if message to slave address 9
        modbus_rx_function->ucMappedAddress = 23;         // map the slave address to 23
        return (MODBUS_APP_GATEWAY_FUNCTION - 0);         // gateway this to MODBUS port 0
    }
```

By manipulating the mapped address (`modbus_rx_function->ucMappedAddress`) the slave address to which the received massage is sent to on the gateway'd MODBUS port is modified. When the response is received from the MODBUS slave 23 on that MODBUS port it will automatically routed back to the originating MODBUS port with its original slave address.

15. Configuring a MODBUS master

Whether a MODBUS port has master functionality or slave functionality is defined when it is initialised.

The following shows a MODBUS serial port being configured and initialised for master mode:

```
....
    (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_POSITIVE),
                                                    // RTU mode as master - serial port 0
....
    (DELAY_LIMIT)(2*SEC),   // MODBUS master timeout for a response from slave - serial port 0
....
    (DELAY_LIMIT)(0.2*SEC), // MODBUS master broadcast timeout - serial port 0
....

fnInitialiseMODBUS_port(0,
                        0, 0,              // no slave parameters
                        fnMODBUSmaster);   // initialise MODBUS serial interface - port 0
```

This MODBUS serial port has been initialised as a MODBUS master and passed a master call back routine as final parameter. The slave configuration parameters have been set to zero since they are not used.

The maximum time waited for a response from a slave is entered as well as a timeout for transmission of broadcast messages.

- If an addressed slave doesn't respond within the timeout period it is considered that it will no longer respond and the master can continue with subsequent transmissions if it wants. The master call-back will receive the event `MODBUS_NO_SLAVE_RESPONSE` and can decide whether to attempt a repeat of a message or to declare the slave as no longer active.
- A broadcast message will never receive a response but the broadcast timeout allows slaves time to process the broadcast before the master continues with further transmissions.

The following shows a MODBUS TCP port being configured and initialised for master mode:

```
....
        {192, 168, 0, 3},               // IP address of TCP slave - TCP master port 4
....
         MODBUS_TCP_PORT,               // TCP port number of TCP master port 4
....
        (2*60),             // MODBUS master - 2 minute TCP idle timeout - TCP master port 4
....
        (DELAY_LIMIT)(2*SEC),  // maximum wait for a response from a slave - TCP master port 4
....
        (DELAY_LIMIT)(0.2*SEC), // MODBUS master broadcast timeout - TCP master port 4
....

fnInitialiseMODBUS_port(4,
                        0, 0,              // no slave parameters
                        fnMODBUSmaster);   // initialise MODBUS TCP interface - port 4
```

See the example MODBUS application `app_modbus.c` for complete examples of slave and master configurations.

15.1. MODBUS Master Transmission

A MODBUS master must transmit commands and requests to MODBUS slaves. These can be destined to slaves connected to a MODBUS serial port or a remove MODBUS TCP slave, or slaves connected to a remove MODBUS gateway.

When a MODBUS master needs to send a command or request it can do so using the function:

```
extern int fnMODBUS_Master_send(
    unsigned char ucInterface,                  // MODBUS port number
    unsigned char ucSlave,                      // slave address
    unsigned short usFunction,                  // public function
    void *details);                             // data content
```

When sending public functions the application code doesn't need to be fully aware of the data format but instead needs only to pass details about the access range as illustrated in the following example.

Read a quantity of coils

```
static MODBUS_READ_QUANTITY read_coils = { 57, 11};
                                       // starting at coil address 57 read 11 coils
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_COILS,
                     (void *)&read_coils);
```

The MODBUS master will transmit a `MODBUS_READ_COILS` request whereby the information passed in the structure `read_coils` [the starting coil address and the quantity of coils] is the only information that the application has to specify for the corresponding message to be constructed.

The master interface can be a MODBUS serial port or a MODBUS TCP port. The transmission over the corresponding physical interface, in the correct data format (eg. ASCII or RTU when transmitting over a serial MODBUS port) is automatically performed by the MODBUS master port.

See appendix A for a complete list of supported public MODBUS functions.

In order for the MODBUS master to send content not conforming to the supported public functions it can pass the data to be sent in transparent form using a structure of type MODBUS_DATA_CONTENT. The following example shows a user defined function code 66 being used, whereby the data content is transparent to the MODBUS master transmission and is sent as raw data content.

```
unsigned char ucData[] = {1,2,3,4,5,6,7,8,9,10};                // RAW data content
MODBUS_DATA_CONTENT frame_content = { ucData, sizeof(ucData)};   // define data content
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, 66, &frame_content); // send to MODBUS port 0
```

When a MODBUS master sends a command or request the routine fnMODBUS_Master_send() returns either 0 when the transmission was successful or else one the following none-zero values:

- MODBUS_TX_MESSAGE_QUEUED – The master is presently waiting for the response from a slave but the new message was successfully queued and will be sent as soon as possible.
- MODBUS_TX_MESSAGE_LOST_NO_QUEUE_SPACE – The master is presently waiting for the response from a slave. The new message couldn't be queued since there is no more queue space.
- MODBUS_TX_MESSAGE_LOST_QUEUE_REQUIRED – The master is presently waiting for the response from a slave. Since no queues are configured for use in this case the message was lost. The master should usually never attempts to send in this case since queue operation is not foreseen. *See chapter 16 for details about MODBUS queues and when they are useful*.

15.2. MODBUS Master Call-Back Events

When a response has been received from a slave the master is informed by the event MODBUS_SLAVE_RESPONSE. In this case the master can check whether it is an exception by looking at the most significant bit of the function code.

The following example shows the identification of exceptions and several function code responses.

```
static int fnMODBUSmaster(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    switch (iType) {
    case MODBUS_SLAVE_RESPONSE:
        if (modbus_rx_function->ucFunctionCode & 0x80) {
            fnDebugMsg("Exception: ");
        }
        switch (modbus_rx_function->ucFunctionCode & 0x7f) {
        case MODBUS_READ_COILS:                                    // function codes
            fnDebugMsg("Coils\r\n");
            break;
        case MODBUS_READ_DISCRETE_INPUTS:
            fnDebugMsg("Inputs\r\n");
            break;
        case MODBUS_READ_HOLDING_REGISTERS:
            fnDebugMsg("Hol Regs\r\n");
            break;
        case MODBUS_READ_INPUT_REGISTERS:
            fnDebugMsg("Input Regs\r\n");
            break;
        }
        break;
    }
}
```

When a slave doesn't respond to a request the event MODBUS_NO_SLAVE_RESPONSE will be received after the timeout has expired. The master can decide whether to try again or declare the slave as no longer present.

After a broadcast has been sent (slave address 0 – or BROADCAST_MODBUS_ADDRESS) the event MODBUS_BROADCAST_TIMEOUT will be received after the broadcast timeout has expired. The master application can use this as an event to continue with slave polling.

A MODBUS TCP master will receive further events which can be used to monitor the TCP connection or check its reliability.

- MODBUS_CONNECTION_ESTABLISHED - A TCP connection has been established with the remote slave.
- MODBUS_CONNECTION_ABORTED – A TCP connection attempt has been rejected.
- MODBUS_CONNECTION_CLOSED – A TCP connection has been closed.
- MODBUS_TCP_REPETITION – A retransmission was performed due to the slave not acknowledging message reception. Note that this is failure of acknowledgement of the TCP frame reception and not a MODBUS slave timeout which can still occur when the TCP frame was correctly received at the slave TCP port.
  The application doesn't need to repeat transmission due to this event since the MODBUS module automatically performs retransmission of lost message content.

16. MODBUS Queues

Message queues are optional but may be useful at MODBUS master interfaces and are recommended when TCP or gateways are in use.

The simplest example of a queue is to consider a simple MODBUS serial master controlled by a local user interface. In normal situations the user interface will dictate that the master interface sends a command or request to a slave on the MODBUS interface. When this command or request is sent, a monitoring timer is started by the MODBUS master (the timeout value is defined by the configuration parameter `serial_master_timeout` or `tcp_master_timeout`). The transaction is complete, and further can be started, when either the slave responds (this could also be an exception response if there is an error in the requested function or its values) or when the master monitor times out. In most simple situations there are no requirements for a queue at the master interface since the user interface will wait for the first transaction to terminate before following ones are attempted.

If however the master interface is being shared with a gateway (either from a TCP gateway or a serial gateway) the user interface isn't aware of whether the master is presently occupied with a transaction originating from the gateway source. Alternatively, a transaction from a gateway source can arrive at any time, also when a local master transaction is in progress. In these circumstances a queue is necessary in order to avoid a second transaction attempt from being rejected.

The amount of data that can be queued is configurable per interface. The number of individual messages which can be queued depends in turn on the total amount of queue size and the length of individual messages.

It is also possible to define a queue for use by a simple master. In this case the user interface can send multiple commands and requests to the master without waiting for each individual transaction to complete. The master will then process each of the waiting messages in the same order that they were queued in (FIFO). The master will also receive responses from slaves in that same order.

All queued messages, whether arriving from a gateway or the local master, are processed in the same order as they were received and the slave response is then returned to the corresponding transmission source.

Queues are enabled by using the define `MODBUS_GATE_WAY_QUEUE`. When queues are in operation, the size of each serial interface queue is defined in the configuration block using additional parameters as shown below:

```
    {
        (MODBUS_MODE_RTU | MODBUS_SERIAL_MASTER | MODBUS_RS485_NEGATIVE),
                                            // default to RTU mode as master - serial port 0
        (MODBUS_MODE_RTU | MODBUS_SERIAL_SLAVE | MODBUS_RS485_POSITIVE),
                                            // default to RTU mode as slave - serial port 1
    },
    #ifdef MODBUS_GATE_WAY_QUEUE
    {
        1024,             // buffer size for queuing waiting messages on MODBUS serial port 0
        0,                // no buffer on MODBUS serial port 1 since it is a slave
    },
    #endif
```

Similarly, there is also an additional queue size for each MODBUS TCP port:

```
#ifdef MODBUS_TCP
    #ifdef MODBUS_TCP_SERVERS
    {
       (MODBUS_TCP_SERIAL_GATEWAY | MODBUS_TCP_SLAVE_PORT),    // TCP slave port 2
        MODBUS_TCP_SLAVE_PORT,                                 // TCP slave port 3
    },
    {
        MODBUS_TCP_PORT,                                    // port number - TCP slave port 2
        4321,                                               // port number - TCP slave port 3
    },
    {
        (2*60),  // MODBUS listener port number - 2 minute TCP idle timeout - TCP slave port 2
        INFINITE_TIMEOUT,                       // no TCP connection timeout - TCP slave port 3
    },
    #endif
    #if MODBUS_TCP_MASTERS > 0
    {
        {192, 168, 0, 10},                      // IP address of TCP slave - TCP master port 4
        {192, 168, 0, 12},                      // IP address of TCP slave - TCP master port 5
    },
    {
         MODBUS_TCP_PORT,                                   // port number - TCP master port 4
         4321,                                              // port number - TCP master port 5
    },
....
    {
        (DELAY_LIMIT)(0.2*SEC),         // MODBUS master broadcast timeout - TCP master port 4
        (DELAY_LIMIT)(0.2*SEC),         // MODBUS master broadcast timeout - TCP master port 5
    },
    #endif
    #ifdef MODBUS_GATE_WAY_QUEUE
    {
        512,                  // buffer size for queuing waiting messages on MODBUS tcp port 2
        256,                  // buffer size for queuing waiting messages on MODBUS tcp port 3
        128,                  // buffer size for queuing waiting messages on MODBUS tcp port 4
        64,                   // buffer size for queuing waiting messages on MODBUS tcp port 5
    },
    #endif
#endif
```

The MODBUS message queues are realised as µTasker queues (as used by inter-task communication). In order for the queues to allow writing, the define `SUPPORT_INPUT_QUEUE_WRITE` must be active. Also a queue must be reserved in the queue pool for each interface. This is performed as in the example configuration in `config.h`

```
    #define NUMBER_MODBUS_QUEUES (MODBUS_SERIAL_INTERFACES + MODBUS_TCP_SERVERS + MODBUS_TCP_MASTERS)
```

The queue operation is usually transparent to the user. The user must however ensure that the queue sizes are adequate for the intended environment. For each queued message there is a small overhead as follows:

- Messages originating from an internal master or from a serial gateway: +3 bytes header.
- Messages originating from a TCP gateways or queued for transmission over a TCP socket: + 9 bytes plus the size of a pointer to the TCP socket session (generally 4 bytes), giving +13 bytes header.

If there is no space available to save a new message to a queue it will be discarded, generally resulting in the master having to retry later. A MODBUS master transmission attempt by a local application will receive the return value `MODBUS_TX_MESSAGE_LOST_NO_QUEUE_SPACE` if a message was discarded, otherwise it will receive either `MODBUS_TX_MESSAGE_QUEUED` or 0, when it was immediately sent.

Should a message be received from a TCP gateway which couldn't be queued it will be discarded locally, but the TCP socket will also not acknowledge its TCP frame reception. This will generally result in a number of repetitions at the TCP level, after which it may be possible to accept it. In some MODBUS TCP based systems this enables the TCP connection itself to operate as a form of queue and may even be adequate to treat infrequent gateway collisions if a minimum of memory consumption is of highest priority.

17. Delayed Responses

The `fnPreFunction()` was discusses in section 7.1. This is called during the processing of a MODBUS request by a slave interface. It is called before the slave returns a response containing the requested information and allows the user interface to update such data content as it sees fit.

In some situations the user interface may not be capable of updating the information without first having to request it from a remote device, such as a second processor connected via a slow serial interface or even a further isolated MODBUS master interface. The user interface is given the possibility to delay the response by returning the appropriate code. In this case the application can start the request of the remote information and remember the `modbus_rx_function->usReference` value associated with the MODBUS request before return the value `MODBUS_APP_DELAYED_RESPONSE`.

The slave will in turn save the request until one of the following events occurs:

- A following request is received from the MODBUS master on the MODBUS slave port. This signifies that the master timed out as the response was not received within its timeout period. This causes the original delayed response to be deleted.
- The user interface calls
  `fnMODBUS_delayed_response(usDelayedReference);`
  where the reference number corresponds to the one remembered when initiating the update of local data. This means that the user interface managed to obtain and update the local data before the master timed out (*this is a general requirement for any system based on this technique*) and it causes the original response to be continued at the place that it was interrupted. The reference number is no longer valid once the response processing has terminated.

When the MODBUS slave interface saves the MODBUS request, waiting for the user interface to prepare data, it saves just the relevant information required to complete the processing. Requests tend to be constructed from short MODBUS frames and so usually a small size can be defined for the data (the length doesn't need to include the slave address nor the function code). In `config.h` the define `MAX_QUEUED_REQUEST_LENGTH` sets the maximum request data length, which can be increased or reduced depending on knowledge of request utilisation in the system.

The following example illustrates a coil response in a certain range requiring the application to first request remote information (requiring a delay before becoming available).

```
static int fnMODBUSPreFunction(int iType, MODBUS_RX_FUNCTION *modbus_rx_function)
{
    unsigned short usAddress = modbus_rx_function->usElementAddress;
    switch (iType) {
    case PREPARE_COILS:           // coils are being read - update table values if necessary
        usAddress -= COILS_START; // reference to start of coil region
        if ((usAddress + modbus_rx_function ->usLength) > 10) &&
            (usAddress < 20)) {                          // special coil region
            fnRequestRemote(modbus_rx_function->usReference);
                                                         // start request and save reference
            return MODBUS_APP_DELAYED_RESPONSE;          // delay response
        }
        break;
    }
    return 0;
}
```

First a remote request is started with a reference to the request number. This function could for example save the request number and start a serial request to a remote device.
When the data has been successfully received it can be updated in the local coil table and the response completed as shown in the following example:

```
static void fnRemoteDataReceived(unsigned char *ucData, unsigned char ucLength,
                                                           unsigned short usDelayedReference)
{
    uMemcpy(&coils[10], ucData, ucLength);                 // update the special coils values
    fnMODBUS_delayed_response(usDelayedReference);         // complete MODBUS response
}
```

This function is called when the new data has been received so that the coil values can be updated and then the MODBUS response can be completed.

In many polling based MODBUS systems the delayed response technique will not be required since a small delay in response content value can be tolerated. If the remote device updating the local data is being polled at the same rate as the MODBUS polling, the largest latency will be the delay between two polling requests. In systems where such information is only requested when needed (neither the remote device is being polled by the application nor the MODBUS slave being polled on the MODBUS bus) the delayed response technique enables optimum efficiency to be achieved.

18. USB Slave Support

In some cases it may be useful to be able to connect an external MODBUS serial master to a slave directly via a USB connection, rather than RS232 or RS485. This may be the case when there is only a single slave or when the slave is performing a gateway function to further MODBUS busses.

The MODBUS master SW is generally running on a PC and so can operate using a virtual COM port over US rather than a standard COM port. As long as the target hardware is capable of supporting the USB device CDC class it can be connected directly to the USB host connection.

The µTasker project includes a USB device CDC example which can be simply configured for use together with a MODBUS slave application. In order to use this, the USB interface must be enabled (`USB_INTERFACE` in `config.h`) and the USB MODBUS slave support activated (`MODBUS_USB_SLAVE` in `config.h`). These defines also configure the µTasker demo project to operate in this configuration by making the following small modifications:

- The MODBUS initialisation `fnInitModbus()` is not performed by `application.c` but by the USB task once the USB interface has been configured for CDC (Communication Device Class, which is equivalent to a UART over USB) use.
- When the USB interface is opened, its owner task is set to the MODBUS task and the USB task doesn't handle any reception data.
- `USBPortID_comms`, the USB port handle, is exported so that it can be used by the MODBUS module.

The use of a USB interface instead of a physical UART is defined in `app_hw_xxxx.h` by defining the particular MODBUS serial port to a value higher than the available processor UARTs:

```
#ifdef MODBUS_USB_SLAVE
    #define MODBUS_UART_1  3     // set higher than UART so that USB is used instead
#else
    #define MODBUS_UART_1  1
#endif
```

This shows that MODBUS port 1 will use UART 1 when its define is a valid UART number (eg. 0,1 or 2 for a processor with 3 UARTs). A value of 3 (or higher) causes the USB interface to be used instead.

The MODBUS module handles all further details of USB operation and the user MODBUS interface remains fully compatible to a UART serial based one.

USB slave mode is however only possible when the MODBUS serial mode is set to ASCII mode (not RTU). This is because the RTU timing rules are not relevant to the packet oriented USB operation and so inter-space character timing cannot be used to recognise MODBUS framing. The use of ASCII has no disadvantages when operating over USB since the higher USB speed compensates for the fact that the data is transmitted in ASCII format rather than directly in binary; the fact that a weaker check sum is used for frame integrity checking is also compensated for by USB's in-build redundancy checks.

19. TCP Connection Details – Masters and Gateways

MODBUS TCP masters are configured with the IP and address of the MODBUS TCP slave that they are to communicate with. The TCP connection to this slave does not exist when the module is started and the connection attempt is not made until it is needed; it is automatically established the first time that the master sends a request or command to the slave.

A MODBUS TCP master being used to gateway to is another type of MODBUS TCP master and obeys the same operation rules in that it automatically attempts to establish a TCP connection to its TCP slave when data is to be sent – in this case as the result of it being transferred to it from a local MODBUS slave acting as gateway.

A TCP connection may break down if the connection path is interrupted or the slave is no longer active (eg. powered down). If the TCP socket has an idle timeout, an idle connection will also be automatically closed after this interval with no activity.

After closing a TCP connection due to an idle timeout or due to an error the MODBUS TCP master will automatically attempt to re-establish a connection as soon as further data is to be sent.

This connection establishment is illustrated in figure 19-1 where a MODBUS TCP gateway is shown, which is performing a gateway function between two TCP connections – the remote MODBUS TCP master is sending a query once every second.

After a short time the remote MODBUS TCP slave is stopped and the TCP connection breaks down; this is shown in figure 19-2. Later still the remote MODBUS TCP slave is started again so that the TCP connection is re-established and the normal functional takes place again; this is shown in figure 19.3.

After each figure a more detailed explanation of what is taking place is given.

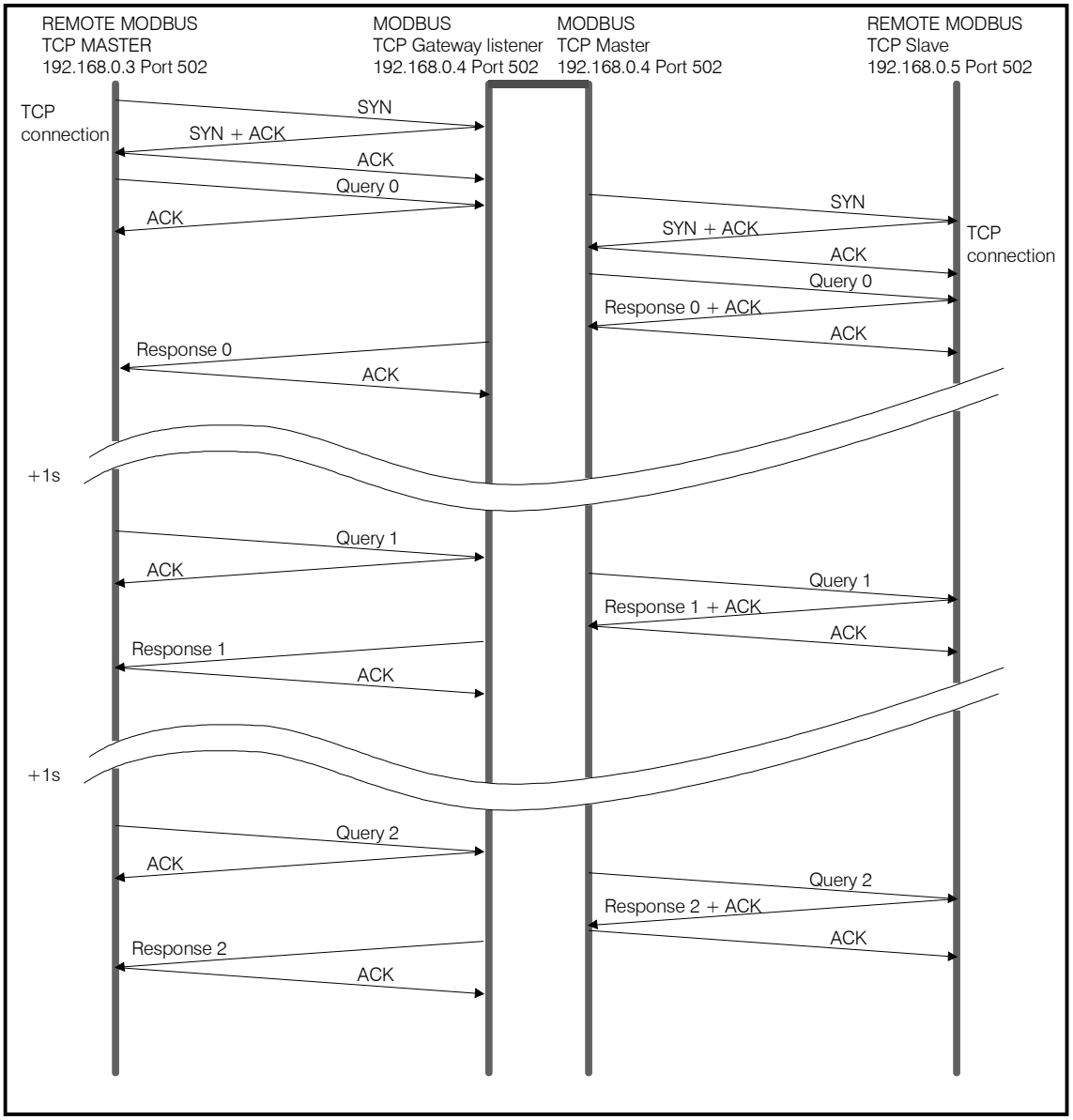

Figure 19-1: A MODBUS TCP master connecting to a remote slave via a gateway and polling once a second

Figure 19-1 shows that the remote master polls the remote slave once a second. The first query causes the establishment of a TCP connection with the gateway listener at IP address 192.168.0.4, which subsequently automatically connects with the remote slave at IP address 192.168.0.5.

Following the successful connections the query (transaction 0) is also passed on and the slave's response routed back to the source at IP address 192.168.0.3.

The process continues with the master polling the remote slave once a second. After the connections were established automatically on the first query the connections remain established and only data and TCP acks are transmitted thereafter.

Note that no ARP frames are shown in figure 19-1, whereby they may be present when resolving the IP/MAC addresses within the local network.

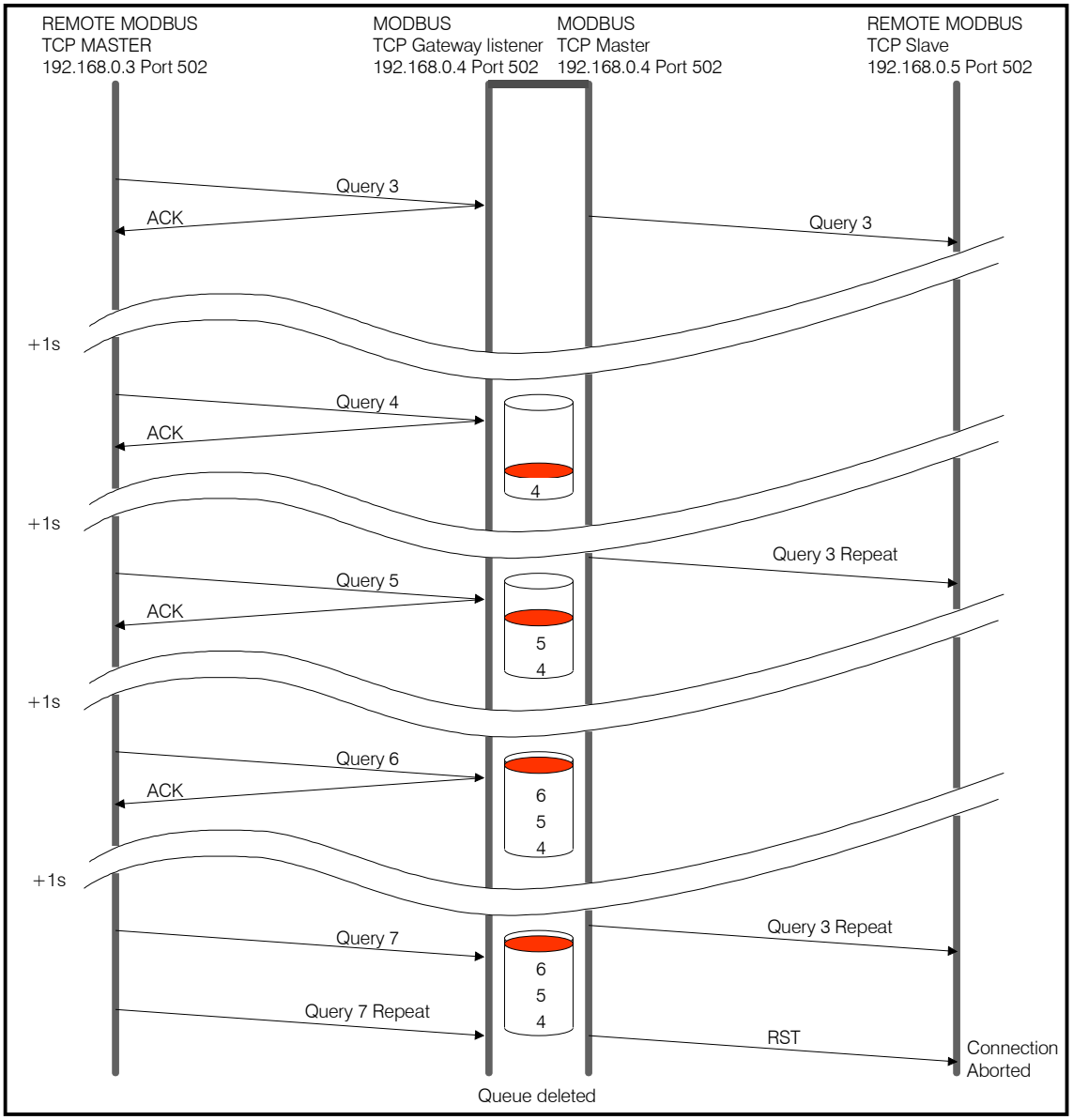

Figure 19-2: The remote slave connection breaking down due to no slave acknowledgements

Figure 19-2 shows what happens when the remote slave is powered down. The gateway listener initially still accepts queries from the remote MODBUS master but they can't be sent to the remote slave as it is presently not answering. Query 3 is repeated a few times until the connection fails and a TCP RST is transmitted as the connection is aborted. (The TCP repetition times and attempts are only illustrational, whereby they are generally 4s with 6 repeats before the abort finally takes place).

The MODBUS gateway initially accepts and acknowledges all queries from the remote master and puts them to the gateway master's queue. *If the slave were to acknowledge one of the outstanding queries the queue content could subsequently be transmitted (each entry as an individual MODBUS query expecting its own response) and thus short intervals of broken connection can be bridged.* The example shows the master's queue becoming full and after that no more queries being accepted and no TCP ACK being returned to the remote master. The remote master's TCP layer will in this case start performing TCP transmissions as the local TCP master is presently doing.

When the connection to the remote TCP finally aborts it is clear that the interruption is of a serious nature and so the internal queue is also flushed, meaning that all presently waiting queries will be lost and the remote MODBUS master will need to handle this situation at its MODBUS application layer. Note that it is in fact the polling master will probably already know that there is a problem with the remote slave since it generally expects an immediate response to each query sent. In the case of TCP gateways there could however be multiple remote MODBUS masters polling the same slave and so several queries still be queued.

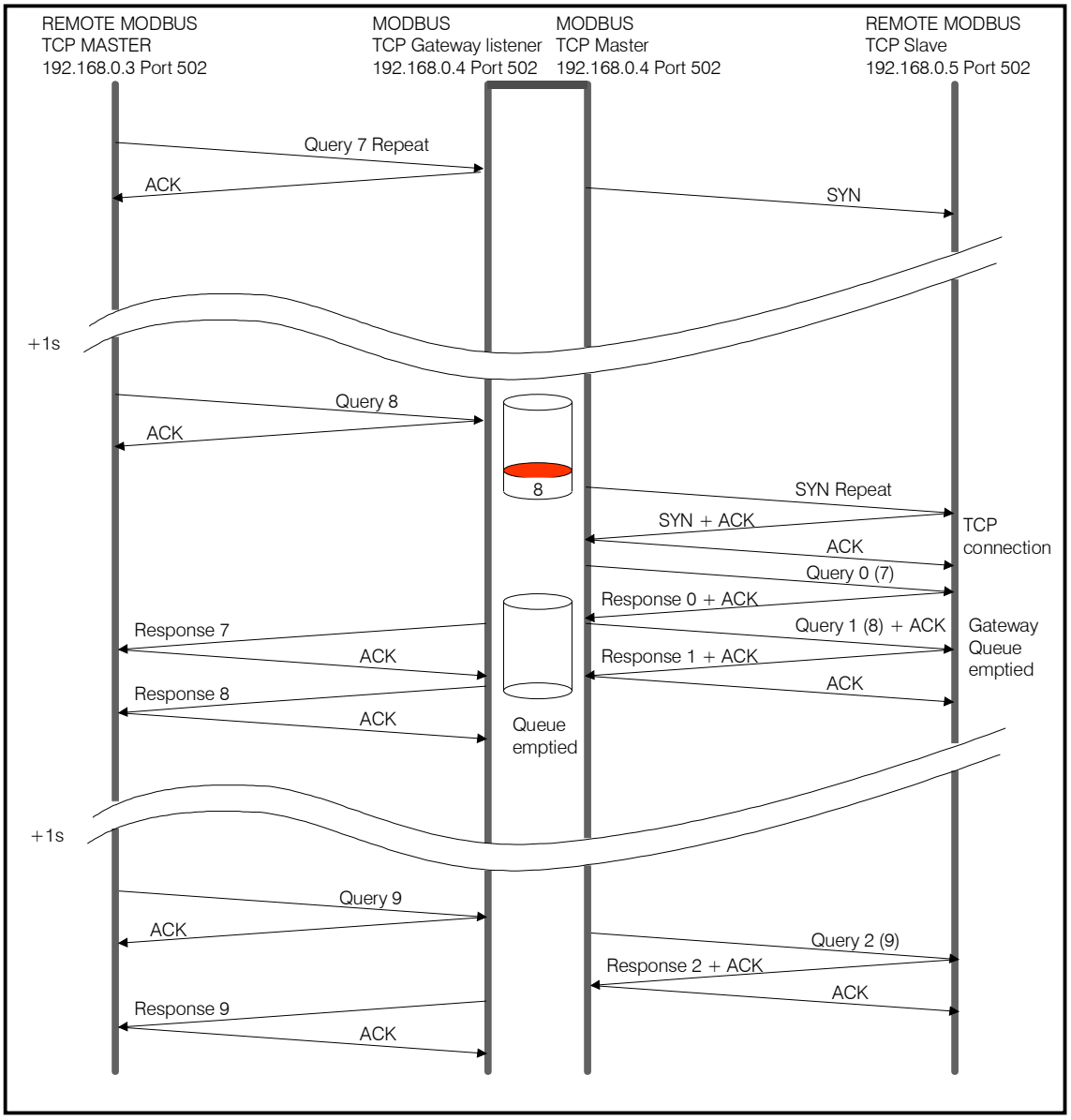

Figure 19-3: The remote slave powered up again

Figure 19-3 shows that once the gateway queue has been deleted due to the abort of the remote slave's TCP connection further queries are again acknowledged and subsequent can again be stored in the gateway master's queue. A further TCP connection attempt to the remote slave is also automatically initiated, although this connection attempt is initially unsuccessful (no response to the TCP SYN).

The first repetition of the TCP SYN does however invoke a response from the remote slave, which has just been powered up again in this example, and so a new TCP connection is then established. It is to be noted that MODBUS messages on a TCP connection have their own TCP transaction identifies and so the query with transaction identifier 7 on the connection from the remote master now has the transaction ID 0 on the connection to the remote slave. Its response is however routed back to the remote master with its original transaction ID of 7.

Once the query 7 has been answered, the queued query 8 is immediately sent, causing its response to also be routed back to the remote master, emptying the gateway queue.

The remote master now continues polling the remote slave at 1 second intervals as was originally the case.

The size of the gateway's queue can be individually configured as detailed in chapter 16 ("*MODBUS Queues*").

This example showed that the mechanisms enable short interruptions in the TCP connection to be bridged but this doesn't necessarily mean that the MODBUS slave timeouts are still respected. It also shows that broken connections will be automatically re-established as soon as possible; queued gateway messages will however be deleted when the connection aborts and so some data loss is to be expected. The MODBUS application at the remote MODBUS TCP master has always the final responsibility of monitoring successful data exchanges and repeating in case this is required when queries are either not responded to or only after an unacceptable delay.

20. FLASH and SRAM Consumption by the MODBUS Module

Since the µTasker MODBUS module is very flexible in the functions that it supports and also has highly flexible detailed configuration it is possible to activate just the support items necessary for a particular project. This results in an optimum use of system resources; FLASH and RAM.

The following table represents typical FLASH and RAM requirements for various typical configurations. The values are *in addition* to the resources used by the µTasker base project, including the µTasker operating system, serial drivers and TCP/IP stack.

	MODBUS module		**Example MODBUS application**	
Configuration	**FLASH**	**RAM**	**FLASH**	**SRAM**
MODBUS slave including standard function support (base for serial and/or TCP)	3k5	1k	500	150
Serial RTU and ASCII	1k5	-	-	-
Line function and diagnostics	3k	-	-	-
Master mode	1k5	-	1k	350
Gateway routing	1k	200	-	-
routing queues + RS485	1k5	+ queues on heap	-	-
MODBUS TCP/IP – master and slave	2k	+ TCP transmission buffers on heap (256 bytes / socket)	500	-
Total for complete functionality	14k	1k5 + Heap	2k	500

Note that the code sizes are also processor, compiler and optimisation dependent. The reference values were taken from a Coldfire project compiled with CodeWarrior V7.1 with optimisation for size.

21. Testing the MODBUS Reference Project in the µTasker Simulator

One of the most powerful features offered by the µTasker project is its capability to accurately simulate complete projects, including the UART and Ethernet operation of the target processor. The MODBUS demo `modbus_app.c` makes use of this fact by configuring a complete MODBUS system on a single (simulated) board for testing both UART and TCP interface as illustrated in figure 21.1.

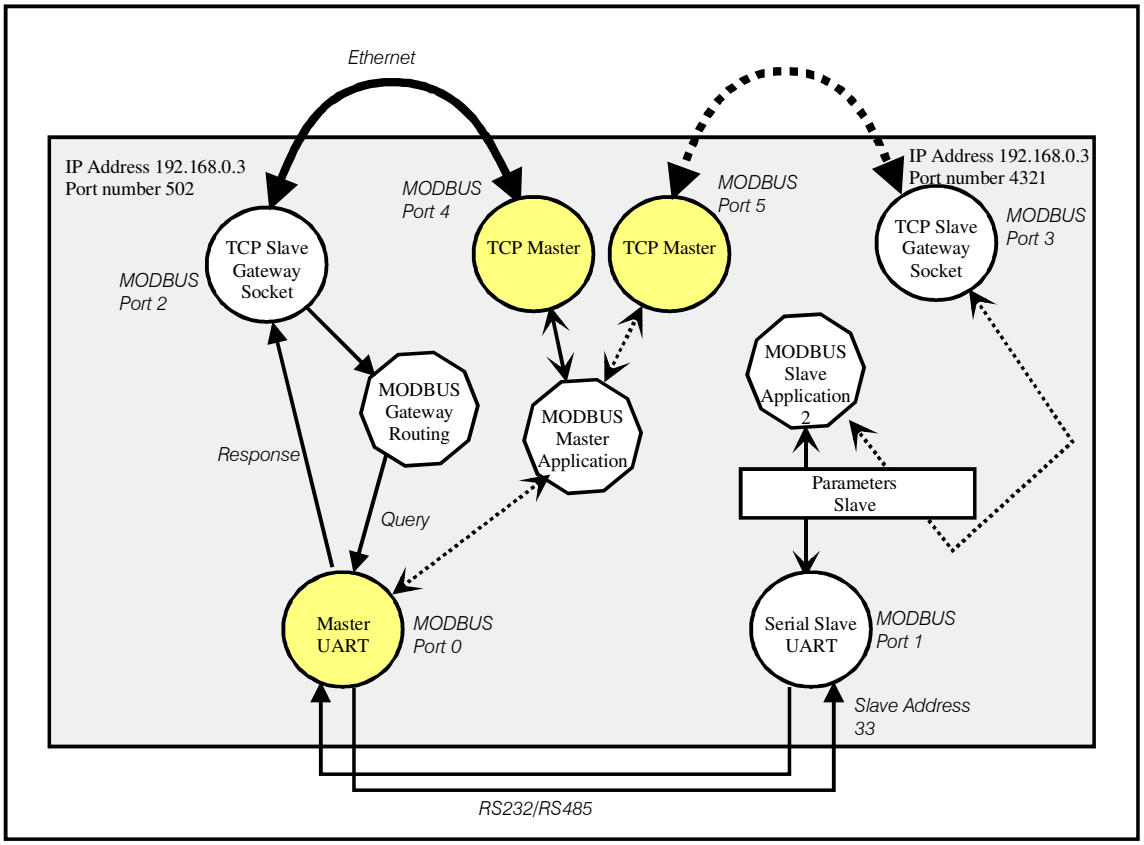

Figure 21-1: A MODBUS TCP/Serial Bridge

The default configuration requires a processor with Ethernet and two UARTs, whereby processors without Ethernet will automatically be configured to use only the UART part of the configuration with the Master Application using serial master MODBUS port 0 directly instead of the TCP master MODBUS port 4.

The Ethernet option `PSEUDO_LOOPBACK` (in `config.h`) allows a simulated device to receive IP frames on its own IP address. This is very practical for simulation use since the entire TCP/IP protocol stack at master and slave are paused when breakpoints are reached, which ensures that the protocol cannot break down due to debug intervention. This is not possible on a real target and tests with a real target require the master TCP part to be on a different node to the TCP slave part.

The MODBUS application is essentially the routine `fnModbusTest()` together with the master call-back `fnMODBUSmaster()`. `fnModbusTest()` is called after the configuration has been initialised and sends the first MODBUS query. On each slave response (or an exception or a timeout) the next test query is sent until all test requests/commands have

been completed and the test begins from the first query again. The master application doesn't use a timer of its own since it makes use of call-back events to kick off the following test, resulting in a test sequence which runs as fast as the slave can respond to each individual request/command.

In the default configuration the TCP slave (MODBUS port 2) which is connected to the TCP master (MODBUS port 4) receives all queries and routes them actively to the MODBUS serial master (MODBUS port 0) which relays them to the slave address 33.

By connecting the two UARTs using COM port mapping (either a cross-over cable between real COM ports or a virtual loop back using "com0com", or similar) the serial query is received by the serial slave (MODBUS port 1) where it is handled based on local parameter and the intervention of its local pre-/post- call-back routines.

The slave response is automatically routed back to the destination TCP master (MODBUS port 4) and received by its master call-back which then initiates the following test.

The master handler generates debug messages (either on the debug UART, TELNET or USB depending on the configuration and processor capabilities).

When MODBUS over TCP is enabled the test is started using the command

```
fnModbusTest(4);                          // start a MODBUS master test on TCP master
```

When MODBUS over TCP is disabled it is initiated by

```
fnModbusTest(0);                        // start a MODBUS master test on the serial port
```

In this case only the serial interfaces are involved.

In order to perform a similar test with a second TCP master/slave combination, but without serial routing, the test can be initiated using:

```
fnModbusTest(5);                          // start a MODBUS master test on TCP master
```

This involves a second TCP connection between MODBUS ports 5 and 3, whereby the TCP slave directly performs the slave function. The value of `TEST_SLAVE_ADDRESS` should be changed (in `modbus_app.c`) from 33 to `MODBUS_TCP_NON_SIGNIFICANT_UNIT_`IDENTIFIER in this case so that the TCP slave actually handles the reception.

Note that a further advantage of testing over a TCP MODBUS connection is the fact that Wireshark can be used to monitor the Ethernet traffic. Wireshark interprets MODBUS content, which allows for comfortable testing and analysis of the demo and real MODBUS projects.

22. Conclusion

The µTasker MODBUS extension module has been designed to be a flexible and efficient solution to most modern day MODBUS requirements. It has rich features allowing it to not just perform classic master and slave modes of operation but to fulfil demanding requirements with multiple serial and TCP interfaces as well as powerful gateway and routing functions.

The features, as described in detail within this document, can be summarised as follows:

MODBUS Serial RTU/ASCII

- Multiple serial master and slave interfaces (limited only by processor UART availability)
- Multiple slaves at each UART interface (each with its own slave address, resources and application interface)
- Routing of received messages to other MODBUS master interfaces based on function codes, access ranges or other decisions.
- RS485 RTS control
- Serial/serial bridge function
- ASCII slave support via USB

MODBUS TCP

- Multiple master and slave TCP ports
- Each slave TCP port supports multiple sessions (multiple master connections)
- Configurable idle connection timeout on each TCP port
- Multiple slaves on each TCP port (each with its own slave address, resources and application interface)
- Slave gateway operation to TCP or serial MODBUS masters
- Slave routing operation to multiple TCP or serial masters via a single TCP port

General

- Autonomous handling of commands and requests or with application intervention.
- Coils and discrete elements can be optimally overlaid with registers (architecturally independent)
- Simple generation of public MODBUS function data content, simplifying application interface.
- Supports delayed slave responses while external data is accessed.

The µTasker project includes an example MODBUS configuration which allows testing of the supported public MODBUS functions on various targets of within the µTasker simulator. This exercises TCP master/slave and also serial master/slave connections, including a gateway transfer between TCP and UART interfaces, allowing the operation to be easily analysed.

Real projects can utilise the demonstration project as a foundation for new configurations allowing demanding MODBUS functionality to be achieved in a very short time with a minimum learning curve involved.

The µTasker MODBUS project supports various target processors and is fully supported by e-mail and on the µTasker forum http://www.uTasker.com/forum/

Modifications:

- V1.00 15.07.2009 First release
- V1.01 28.07.2009 Add AVR32 details in appendix A
- V1.02 26.08.2009 Add USB slave support {V1.02} and additional routing control {V1.05}
- V1.03 25.09.2009 Add chapter explaining details of TCP connections and breakdowns {V1.07}
- V1.04 26.09.2009 Add explanation of mapping slave addresses in a gateway transfer
- V1.05 13.01.2010 Add not on RS485 RTS frame diagram – figure 11-1
- V1.06 20.05.2010 Correct Write single register (0x06) example in appendix A
- V1.07 29.07.2012 Add STM32 and Kinetis details in Appendix B

Appendix A – Public Functions Supported by the µTasker MODBUS Module

When a MODBUS master needs to send a command or request it can do so using the function:

```
extern int fnMODBUS_Master_send(
    unsigned char ucInterface,                  // MODBUS port number
    unsigned char ucSlave,                      // slave address
    unsigned short usFunction,                  // public function
    void *details);                             // data content
```

When sending public functions the application code doesn't need to be fully aware of the data format but instead needs only to pass details about the access range. The following transmission examples show the use of each supported public function, whereby the request is being sent to MODBUS port 0 and to a slave address defined by `TEST_SLAVE_ADDRESS`.

Read a quantity of coils (0x01)

```
static MODBUS_READ_QUANTITY read_coils = { 57, 11};
                                        // starting at coil address 57 read 11 coils
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_COILS,
                     (void *)&read_coils);
```

Read discrete inputs (0x02)

```
static MODBUS_READ_QUANTITY read_discretes = { 15, 7};
                                         // starting at input address 15 read 7 inputs
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_DISCRETE_INPUTS,
                     (void *)&read_discretes);
```

Read holding registers (0x03)

```
const MODBUS_READ_QUANTITY read_holding_regs = { 2, 1};
                                         // starting at register address 2 read 1 inputs
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_HOLDING_REGISTERS,
                     (void *)&read_holding_regs);
```

Read input registers (0x04)

```
const MODBUS_READ_QUANTITY read_input_regs = { 24, 3};
                                       // starting at register address 24 read 3 registers
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_INPUT_REGISTERS,
                     (void *)&read_input_regs);
```

Write single coil (0x05)

```
static MODBUS_COIL_STATE write_coil = { 50, 1};           // set coil 50 to state 1
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_WRITE_SINGLE_COIL,
                     (void *)&write_coil);
```

Write single register (0x06)

```
const MODBUS_SINGLE_REGISTER write_register = {2, 0x1234}; // write register at address 2
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS, MODBUS_WRITE_SINGLE_REGISTER,
                     (void *)&write_register);
```

Read exception status (0x07) – (Serial Line Only)

```
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_READ_EXCEPTION_STATUS, 0);
```

This request has no data part.

Diagnostics (0x08) – (Serial Line Only)

```
const MODBUS_DIAGNOSTICS_REQUEST diag = {MODBUS_DIAG_SUB_RETURN_QUERY_DATA, 0x1234};
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS, MODBUS_DIAGNOSTICS,
                     (void *)&diag);
```

This request has several sub-function codes but always 2 bytes of data. The possible sub-functions are:

- MODBUS_DIAG_SUB_RETURN_QUERY_DATA
- MODBUS_DIAG_SUB_RESTART_COMS
- MODBUS_DIAG_SUB_RETURN_DIAG_REG
- MODBUS_DIAG_SUB_CHANGE_ASCII_DELIM
- MODBUS_DIAG_SUB_FORCE_LISTEN_ONLY
- MODBUS_DIAG_SUB_CLEAR_DISGNOSTICS
- MODBUS_DIAG_SUB_RTN_BUS_MSG_CNT
- MODBUS_DIAG_SUB_RTN_BUS_COM_ERR_CNT
- MODBUS_DIAG_SUB_RTN_BUS_EXC_ERR_CNT
- MODBUS_DIAG_SUB_RTN_SLAVE_MSG_CNT
- MODBUS_DIAG_SUB_RTN_SLAVE_NO_RSP_CNT
- MODBUS_DIAG_SUB_RTN_SLAVE_NAK_CNT
- MODBUS_DIAG_SUB_RTN_SLAVE_BSY_CNT
- MODBUS_DIAG_SUB_RTN_BUS_CHR_ORUN_CNT
- MODBUS_DIAG_SUB_CLEAR_ORUN_AND_FLG

Get Comm Event Counter (0x0b) – (Serial Line Only)

```
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_GET_COMM_EVENT_COUNTER, 0);
```

This request has no data part.

Get Comm Event Log (0x0c) – (Serial Line Only)

```
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_GET_COMM_EVENT_LOG, 0);
```

This request has no data part.

Write Multiple Coils (0x0f)

```
const static unsigned char coil_values[] = { 0x55, 0xaa, 0x07 };
static MODBUS_BITS multi_coils = { (MODBUS_BITS_ELEMENT *)&coil_values[0],
                                   {55, (55 + 8 -1)}}; // write 8 coils starting at coil 55
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS, MODBUS_WRITE_MULTIPLE_COILS
                     (void *)&multi_coils);
```

Write Multiple Registers (0x10)

```
static const unsigned short reg_values[] = { 0x55aa, 0xcc33, 9876, 0xabcd };
const MODBUS_REGISTERS multi_regs = { (unsigned short *)&reg_values[0], {2, 5}};
                     // write 4 registers starting at address 2 and finishing at address 5
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS,MODBUS_WRITE_MULTIPLE_REGISTERS,
                     (void *)&multi_regs);
```

Report Slave ID (0x11) – (Serial Line Only)

```
fnMODBUS_Master_send(0, TEST_SLAVE_ADDRESS, MODBUS_REPORT_SLAVE_ID, 0);
```

This request has no data part.

Mask Write Register (0x16)

```
const MODBUS_SINGLE_REGISTER_MASK reg_mod = {2, 0xff00, 0x0055};
                        // AND register at address 2 with 0xff00 and then OR with 0x0055
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS, MODBUS_MASK_WRITE_REGISTER,
                     (void *)&reg_mod);
```

Read/Write Multiple Registers (0x17)

```
static const unsigned short reg_values[] = {0x1234, 0x5678, 0x9abc, 0xdef0, 0x55aa};
const MODBUS_READ_WRITE_REGISTERS multi_regs = {{2, 5},
      {(unsigned short *)&reg_values[0], {2, 6}}}; // write 5 holding registers starting
                                                      at address 2 and finishing at
                                    address 6, then read 5 registers starting at address 2
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS,
                      MODBUS_READ_WRITE_MULTIPLE_REGISTER, (void *)&multi_regs);
```

Read FIFO Queue (0x18)

```
const FIFO_ADDRESS FIFO_Add = {0x0003};   // read FIFO queue from FIFO pointer register 3
fnMODBUS_Master_send(ucMasterTestPort, TEST_SLAVE_ADDRESS, MODBUS_READ_FIFO_QUEUE,
                     (void *)&FIFO_Add);
```

None-supported functions

- *Read File Record (0x14)*
- *Write File Record (0x15)*
- *Get Comm Event Log (0x0c)- no log retained*
- *Encapsulated Interface Transport (0x2b)*
  - *CANopen General Reference Request and Response (0x2b / 0x0d)*
  - *Read Device Identification (0x2b / 0x0e)*

Appendix B – Hardware Dependencies

a) Hardware Timers used by the MODBUS Module

Depending on the target processor and the operating modes supported by the MODBUS module some hardware timers may be required. The timers are used to perform the following functions on serial MODBUS ports:

- RTU inter-character space timing. One timer per UART receiver, unless the UART contains an idle line timer built in to its hardware.
- RTS line control in RS485 mode. One timer per UART transmitter when this mode is used, unless the UART supports RS485 operation in HW or has other timing suitable for control without extra delays.

Note that these two functions can be performed by a single hardware timer since their activities never overlap during MODBUS operation. This is true due to the fact that the MODBUS protocol is not full-duplex in nature and so only either the receiver or the transmitter is ever active at any specific point in time.

Furthermore it is possible to allocate the same hardware timer to control the operation of multiple UARTs as long as the UARTs are not used at the same time. *This restriction is however not practical in the majority of MODBUS systems*.

The following section details the timers used and the method of configuring the specific hardware.

b) Processor-specific Hardware Defines

Freescale Coldfire V2 MCU

When either RTU mode or RS485 mode is enabled the Coldfire requires a single DMA timer per enabled UART. The choice of DMA timers for each UART is configured in `app_hw_m5223x.h`:

```
#define _TIMER_INTERRUPT_SETUP    DMA_TIMER_SETUP  // the timer interface type for MODBUS
#define SUPPORT_DMA_TIMER         // always activate DMA timer support for use by MODBUS
#define MODBUS0_DMA_TIMER_CHANNEL              3
                                  // use DMA timer channel 3 for first MODBUS serial port
#define MODBUS0_DMA_TIMER_INTERRUPT_PRIORITY   DTIM3_INTERRUPT_PRIORITY
#define MODBUS1_DMA_TIMER_CHANNEL              1
                                  // use DMA timer channel 1 for second MODBUS serial port
#define MODBUS1_DMA_TIMER_INTERRUPT_PRIORITY   DMA_TIMER1_INTERRUPT_PRIORITY
#define MODBUS2_DMA_TIMER_CHANNEL              2
                                  // use DMA timer channel 2 for third MODBUS serial port
#define MODBUS2_DMA_TIMER_INTERRUPT_PRIORITY   DMA_TIMER2_INTERRUPT_PRIORITY
```

The mapping of MODBUS serial port to UART is configured by

```
    #define MODBUS_UART_0     1             // MODBUS UART port 0 uses physical UART 1
    #define MODBUS_UART_1     2             // MODBUS UART port 1 uses physical UART 2
    #define MODBUS_UART_2     0             // MODBUS UART port 2 uses physical UART 0
```

When RS485 mode is selected the RTS signal belonging to the UART is automatically configured and controlled.

ATMEL SAM7X

The SAM7X supports idle line timing on USARTs 0 and 1 but not on the debug unit UART 2 (DBGU). It also supports RS485 mode on USARTs 0 and 1 but not on the debug unit UART 2 (DBGU). If only USARTs 0 and 1 are used, no hardware timers are required for any serial MODBUS modes.

When a serial MODBUS port is configured on UART2 (DBGU) one hardware timer will be required (timers 0..2 are available). This is configured in `app_hw_sam7x.h` as in the following example:

```
#define SUPPORT_TIMER                       // MODBUS required HW timer in RTU/RS485 modes
#define SUPPORT_US_TIMER                    // allow us resolution
#define _TIMER_INTERRUPT_SETUP  TIMER_INTERRUPT_SETUP
                                            // the timer interface type for MODBUS
#define MODBUS2_TIMER_CHANNEL    0          // use timer 0 for third MODBUS serial port

#define PRIORITY_HW_TIMER        4          // interrupt priority of HW timers
```

The mapping of MODBUS serial port to UART is configured by

```
#define MODBUS_UART_0     1     // MODBUS UART port 0 uses physical UART 1
#define MODBUS_UART_1     0     // MODBUS UART port 1 uses physical UART 0
#define MODBUS_UART_2     1     // MODBUS UART port 2 uses physical UART 2 (DBGU)
#define DBGU_UART               // enable serial on DBGU
```

When RS485 mode is selected the RTS signal belonging to the USARTs are automatically configured and controlled. The RTS signal used by UART2 must be additionally defined as follows since a general purpose port is used:

```
#define RTS_2_PIN          PA29             // PA29 is used as RTS line
#define RTS_2_PORT         PIOA             // on port A
#define RTS_2_PORT_OUT     PIO_OER_A        // register to enable output
#define RTS_2_PORT_SET     PIO_SODR_A       // register to set output high
#define RTS_2_PORT_CLEAR   PIO_CODR_A       // register to set output low
```

Note that the polarity of the RTS signal from the USARTs (channel 0 or channel 1) is fixed as ‘1’ during transmission and cannot be configured as can other RTS outputs. This is due to the operation of the device in RS485 mode where it drives this signal directly. This polarity is however suitable for most RS485 transceiver circuits.

ATMEL AVR32

The AVR32 supports idle line timing on all 4 USARTs. It also supports RS485 mode on USARTs 1 only. If only USART 1 is used, no hardware timers are required for any serial MODBUS modes.

When a serial MODBUS port is configured on one of the other USARTs (0, 1 or 2) and RS485 mode is needed, one hardware timer will be required for each RTS output (timers 0..2 are available). This is configured in `app_hw_avr32.h` as in the following example:

```
#define SUPPORT_TIMER                       // MODBUS required HW timer in RS485 modes
#define SUPPORT_US_TIMER                    // allow us resolution

#define _TIMER_INTERRUPT_SETUP  TIMER_INTERRUPT_SETUP
                                            // the timer interface type for MODBUS

#define MODBUS0_TIMER_CHANNEL    0        // use timer 0 for first MODBUS serial port
#define MODBUS2_TIMER_CHANNEL    1        // use timer 1 for third MODBUS serial port
#define MODBUS3_TIMER_CHANNEL    2        // use timer 2 for fourth MODBUS serial port
```

The mapping of MODBUS serial port to UART is configured by

```
    #define MODBUS_UART_0     1     // MODBUS UART port 0 uses physical UART 1
    #define MODBUS_UART_1     0     // MODBUS UART port 1 uses physical UART 0
    #define MODBUS_UART_2     3     // MODBUS UART port 2 uses physical UART 3
    #define MODBUS_UART_3     2     // MODBUS UART port 3 uses physical UART 2
```

When RS485 mode is selected the RTS signal belonging to the USARTs are automatically configured and controlled.

Note that the polarity of the RTS signal from USART1 (channel 1) is fixed as ‘1’ during transmission and its polarity cannot be configured as can other RTS outputs. This is due to the operation of the device in RS485 mode where it drives this signal directly. This polarity is however suitable for most RS485 transceiver circuits.

Luminary Micro

When either RTU mode or RS485 mode is enabled the Luminary Micro targets requires a single timer per enabled UART. The choice of timers for each UART is configured in `app_hw_lm3sxxxx.h`:

```
#define SUPPORT_TIMER                           // enable timer support

#define _TIMER_INTERRUPT_SETUP  TIMER_INTERRUPT_SETUP

#define MODBUS0_TIMER_CHANNEL    0              // MODBUS serial port 0 used timer 0
#define MODBUS1_TIMER_CHANNEL    1              // MODBUS serial port 1 used timer 1
```

The mapping of MODBUS serial port to UART is configured by

```
#define MODBUS_UART_0     1             // MODBUS UART port 0 uses physical UART 1
#define MODBUS_UART_1     0             // MODBUS UART port 1 uses physical UART 0
```

When RS485 mode is selected the RTS signal belonging to the UARTs is configured as follows:

```
#define RTS_0_PORT          GPIODATA_D     // on port D
#define RTS_0_PIN           PORTD_BIT1     // Port D-1 is used as RTS0 line
#define RTS_0_PORT_POWER    CGC_GPIOD      // port's power bit
#define RTS_0_PORT_ENABLE   GPIODEN_D      // port's enable bit
#define RTS_0_PORT_DDR      GPIODIR_D      // ports data direction register

#define RTS_1_PORT          GPIODATA_D     // on port D
#define RTS_1_PIN           PORTD_BIT0     // Port D-0 is used as RTS1 line
#define RTS_1_PORT_POWER    CGC_GPIOD      // port's power bit
#define RTS_1_PORT_ENABLE   GPIODEN_D      // port's enable bit
#define RTS_1_PORT_DDR      GPIODIR_D      // ports data direction register
```

NXP LPC2XXX

When RTU mode is enabled the LPC2xxx targets require a single timer per enabled UART. The choice of timers for each UART is configured in `app_hw_lpc23xx.h`:

```
#define SUPPORT_TIMER                            // enable timer support

#define _TIMER_INTERRUPT_SETUP  TIMER_INTERRUPT_SETUP

#define MODBUS0_TIMER_CHANNEL    1             // MODBUS serial port 0 used timer 1
#define MODBUS1_TIMER_CHANNEL    2             // MODBUS serial port 1 used timer 2
#define MODBUS2_TIMER_CHANNEL    3             // MODBUS serial port 2 used timer 3
```

Note that the LPC2xxx uses one of its timers for the µTasker operating tick. This timer can therefore not be assigned for controlling a MODBUS UART. Which timer is used by the tick interrupt is also defined in `app_hw_lpc23xx.h`. The default timer is timer 0, and alternatives are set by `#define TICK_TIMER_X`, where X is the hardware timer number

The mapping of MODBUS serial port to UART is configured by

```
    #define MODBUS_UART_0     1              // MODBUS UART port 0 uses physical UART 1
    #define MODBUS_UART_1     0              // MODBUS UART port 1 uses physical UART 0
```

When RS485 mode is selected the RTS signal belonging to the UARTs is configured as follows:

```
    #define RTS_0_PORT_SET     FIO0SET       // register to set RTS0
    #define RTS_0_PORT_CLR     FIO0CLR       // register to clear RTS0
    #define RTS_0_PIN          PORT0_BIT6    // Port 0.6 is used as RTS0 line
    #define RTS_0_PORT_DDR     FIO0DIR       // ports data direction register
    #define RTS_0_PORT         PORT_0        // on port 0

    #define RTS_1_PORT_SET     FIO0SET       // register to set RTS1
    #define RTS_1_PORT_CLR     FIO0CLR       // register to clear RTS0
    #define RTS_1_PIN          PORT0_BIT7    // Port 0.7 is used as RTS1 line
    #define RTS_1_PORT_DDR     FIO0DIR       // ports data direction register
    #define RTS_1_PORT         PORT_0        // on port 0
```

ST-Micro STM32

When RTU mode is enabled the STM32 targets require a single timer per enabled UART. The choice of timers for each UART is configured in `app_hw_stm32.h`:

```
#define SUPPORT_TIMER                           // enable timer support

#define _TIMER_INTERRUPT_SETUP  TIMER_INTERRUPT_SETUP

#define MODBUS0_TIMER_CHANNEL    3             // MODBUS serial port 0 used timer 3
#define MODBUS1_TIMER_CHANNEL    4             // MODBUS serial port 1 used timer 4
```

Note that the STM32 uses timers 2, 3, 4 or 5 for this function due to the fact that these timers are available in all devices.

The mapping of MODBUS serial port to UART is configured by

```
        #define MODBUS_UART_0     1              // MODBUS UART port 0 uses physical UART 2
        #define MODBUS_UART_1     0              // MODBUS UART port 1 uses physical UART 1
        #define MODBUS_UART_1     3              // MODBUS UART port 3 uses physical UART 4

(Note that the STM32 USARTs count from 1)
```

When RS485 mode is selected the RTS signal belonging to the UART may have several remapping possibilities. The set used can be configured using defines such as USART2_REMAP – see the STM32 developer's document for more details. It is to be noted that the RTS signal is actually configured and controlled in GPIO mode rather than using automatic RTS control, which is not suitable for this operation; this detail is however not of significance to the user of the UART interface.

Freescale Kinetis

When RTU mode is enabled the Kinetis targets require a single PIT timer per enabled UART. The choice of PIT used for each UART is configured in `app_hw_kinetis.h`:

```
#define SUPPORT_PITS                                                 // support PITs

#define _TIMER_INTERRUPT_SETUP     PIT_SETUP

#define MODBUS0_PIT_TIMER_CHANNEL  0
#define MODBUS1_PIT_TIMER_CHANNEL  1
#define MODBUS2_PIT_TIMER_CHANNEL  2
#define MODBUS3_PIT_TIMER_CHANNEL  3

#define MODBUS0_PIT_INTERRUPT_PRIORITY  PIT0_INTERRUPT_PRIORITY
#define MODBUS1_PIT_INTERRUPT_PRIORITY  PIT1_INTERRUPT_PRIORITY
#define MODBUS2_PIT_INTERRUPT_PRIORITY  PIT2_INTERRUPT_PRIORITY
#define MODBUS3_PIT_INTERRUPT_PRIORITY  PIT3_INTERRUPT_PRIORITY
```

The mapping of MODBUS serial port to UART is configured by

```
    #define MODBUS_UART_0     1              // MODBUS UART port 0 uses physical UART 1
    #define MODBUS_UART_1     0              // MODBUS UART port 1 uses physical UART 0
    #define MODBUS_UART_1     3              // MODBUS UART port 3 uses physical UART 3
```

When RS485 mode is selected automatic RTS control mode in the device is used.

c) RTS Delay Time Measurements

The following results were obtained by measuring the final transmission interrupt of the UART interface, operating at 19'200Baud, sending an RTU frame. *Devices supporting DMA operation were also measured in DMA mode, where the final interrupt corresponds to the DMA completion interrupt.*

The values were used to configure a timer (where required) to generate a time delay suitable for deactivating the RTS signal as the final stop bit is sent to the MODBUS bus.

Freescale Coldfire V2 MCU

- M522xx Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 17us before first bit. Last character interrupt about 10.9 bit times before end (0.57ms).
- M522xx DMA 19'200 Baud (Bit Period 52us). RTS activated about 17us before first bit. Last character (DMA complete) interrupt about 22 bit times before end (1.144ms).

ATMEL SAM7X

- SAM7X Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character interrupt about 9.8 bit times before end (0.512ms).
  Note: UART2 (using ports as RTS) has activation stable at about 37us and identical off delay.
- SAM7X DMA 19'200 Baud (Bit Period 52us) [USART0 and USART1]. RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character (DMA complete) interrupt about 21 bit times before end (1.08ms).
  Note: UART2 (using ports as RTS) has activation stable at about 37us and identical off delay.
- SAM7X in RS485 mode 19'200 Baud (Bit Period 52us) [USART0 and USART1 only]. RTS activated about 20us..35us (varies due to synchronisation to the UART clock) before first bit. RTS is active High. RTS automatically negated (low) exactly at the end of the stop bit. (*By setting US_TTGR to 1..255 additional bit period delays can be added, but not needed for the MODBUS driver*).
- NOTE: DBGU of SAM7X [UART2] cannot operate in 7 bit mode. This means that it can only operate in RTU mode.

ATMEL AVR32

- AVR32 Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character interrupt about 9.8 bit times before end (0.512ms).
  Note: UART2 (using ports as RTS) has activation stable at about 37us and identical off delay.
- AVR32 DMA (*to be added*)
- AVR32 in RS485 mode 19'200 Baud (Bit Period 52us) [USART1 only]. RTS activated about 20us..35us (varies due to synchronisation to the UART clock) before first bit. RTS is active High. RTS automatically negated (low) exactly at the end of the stop bit. (*By setting US_TTGR to 1..255 additional bit period delays can be added, but not needed for the MODBUS driver*).

Luminary Micro

- LM3Sxxxx Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 13us before first bit. Last character interrupt about 11.9 bit times before end (0.62ms).
- LM3Sxxxx DMA 19'200 Baud (Bit Period 52us). RTS activated about 13us before first bit. DMA transmission complete interrupt about 11.9 bit times before end (0.62ms).

NXP LPC23XX

- LPC23XX Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 20us..70us (varies due to synchronisation to the UART clock) before first bit. Last character interrupt about 1 bit before end (43us).

ST-Micro STM32

- STM32 Interrupt 19'200 Baud (Bit Period 52us). RTS activated about 13us before first bit. Last character interrupt about 11.9 bit times before end (0.62ms).

Freescale Kinetis

- When RS485 mode is selected automatic RTS control mode in the device is used which automatically controls the activation and negation of the line.