```cpp
/**
 * Copyright (c) 2015, Yaacov Zamir <kobi.zamir@gmail.com>
 * Copyright (c) 2017, Andrew Voznytsa <andrew.voznytsa@gmail.com>,
 *   FC_WRITE_REGISTER and FC_WRITE_MULTIPLE_COILS support
 * Copyright (c) 2019, Soroush Falahati <soroush@falahai.net>,
 *   total communication rewrite, setUnitAddress(),
 *   FC_READ_EXCEPTION_STATUS support, general refactoring
 *
 * Permission to use, copy, modify, and/or distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF  THIS SOFTWARE.
 */

#include <string.h>
#include "ModbusSlave.h"

/**
 * ------------------------------------------------------
 *                CONSTANTS AND MACROS
 * ------------------------------------------------------
 */

#define MODBUS_FRAME_SIZE 4
#define MODBUS_CRC_LENGTH 2

#define MODBUS_ADDRESS_INDEX 0
#define MODBUS_FUNCTION_CODE_INDEX 1
#define MODBUS_DATA_INDEX 2

#define MODBUS_BROADCAST_ADDRESS 0
#define MODBUS_ADDRESS_MIN 1
#define MODBUS_ADDRESS_MAX 247

#define MODBUS_HALF_SILENCE_MULTIPLIER 3
#define MODBUS_FULL_SILENCE_MULTIPLIER 7

#define readUInt16(arr, index) word(arr[index], arr[index + 1])
#define readCRC(arr, length) word( \
  arr[(length - MODBUS_CRC_LENGTH) + 1], \
  arr[length - MODBUS_CRC_LENGTH])

/**
 * ------------------------------------------------------
 *                  PUBLIC METHODS
```

```
 53   * ------------------------------------------------
 54   */
 55
 56  /**
 57   * Initialize the modbus object.
 58   *
 59   * @param unitAddress the modbus slave unit address.
 60   * @param transmissionControlPin the digital out pin to be used for RS485
 61   * transmission control.
 62   */
 63  Modbus::Modbus(uint8_t unitAddress, int transmissionControlPin)
 64      : Modbus(Serial, unitAddress, transmissionControlPin)
 65  {
 66  }
 67
 68  /**
 69   * Initialize the modbus object.
 70   *
 71   * @param serialStream the serial stream used for the modbus communication.
 72   * @param unitAddress the modbus slave unit address.
 73   * @param transmissionControlPin the digital out pin to be used for RS485
 74   * transmission control.
 75   */
 76  Modbus::Modbus(
 77    Stream &serialStream,
 78    uint8_t unitAddress,
 79    int transmissionControlPin)
 80      : _serialStream(serialStream)
 81  {
 82      // set modbus slave unit id
 83      Modbus::setUnitAddress(unitAddress);
 84
 85      // set transmission control pin for RS485 communications.
 86      _transmissionControlPin = transmissionControlPin;
 87  }
 88
 89  /**
 90   * Sets the modbus slave unit address.
 91   *
 92   * @param unitAddress the modbus slave unit address.
 93   */
 94  void Modbus::setUnitAddress(uint8_t unitAddress)
 95  {
 96      if (unitAddress < MODBUS_ADDRESS_MIN || unitAddress > MODBUS_ADDRESS_MAX)
 97      {
 98          return;
 99      }
100      _unitAddress = unitAddress;
101  }
102
103  /**
104   * Gets the total number of bytes sent.
```

```
105   *
106   * @return the number of bytes.
107   */
108  uint64_t Modbus::getTotalBytesSent() {
109      return _totalBytesSent;
110  }
111
112  /**
113   * Gets the total number of bytes received.
114   *
115   * @return the number of bytes.
116   */
117  uint64_t Modbus::getTotalBytesReceived() {
118      return _totalBytesReceived;
119  }
120
121  /**
122   * Begins initializing the serial stream and preparing to read request messages
123   *
124   * @param boudrate the serial port boudrate.
125   */
126  void Modbus::begin(uint64_t boudrate)
127  {
128      // initialize transmission control pin state
129      if (_transmissionControlPin > MODBUS_CONTROL_PIN_NONE)
130      {
131          pinMode(_transmissionControlPin, OUTPUT);
132          digitalWrite(_transmissionControlPin, LOW);
133      }
134
135      // disable serial stream timeout and cleans the buffer
136      _serialStream.setTimeout(0);
137      _serialStream.flush();
138      _serialTransmissionBufferLength = _serialStream.availableForWrite();
139
140      // calculate half char time time from the serial's baudrate
141      if (boudrate > 19200)
142      {
143          _halfCharTimeInMicroSecond = 250; // 0.5T
144      }
145      else
146      {
147          _halfCharTimeInMicroSecond = 5000000 / boudrate; // 0.5T
148      }
149
150      // set the last received time to 3.5T on the future to ignore
151      // request currently in the middle of transmission
152      _lastCommunicationTime = micros() +
153        (_halfCharTimeInMicroSecond * MODBUS_FULL_SILENCE_MULTIPLIER);
154
155      // sets the request buffer length to zero
156      _requestBufferLength = 0;
```

```
157 }
158
159 /**
160  * Checks if we have a complete request, parses the request, executes the
161  * corresponding registered callback and writes the response.
162  *
163  * @return the number of bytes written as response
164  */
165 uint8_t Modbus::poll()
166 {
167     // if we are in the writing, let it end first
168     if (_isResponseBufferWriting)
169     {
170         return Modbus::writeResponse();
171     }
172
173     // wait for one complete request packet
174     if (!Modbus::readRequest())
175     {
176         return 0;
177     }
178
179     // prepare output buffer
180     memset(_responseBuffer, 0, MODBUS_MAX_BUFFER);
181     _responseBuffer[MODBUS_ADDRESS_INDEX] =
182       _requestBuffer[MODBUS_ADDRESS_INDEX];
183     _responseBuffer[MODBUS_FUNCTION_CODE_INDEX] =
184       _requestBuffer[MODBUS_FUNCTION_CODE_INDEX];
185     _responseBufferLength = MODBUS_FRAME_SIZE;
186
187     // validate request
188     if (!Modbus::validateRequest())
189     {
190         return 0;
191     }
192
193     // execute request and fill the response
194     uint8_t status = Modbus::createResponse();
195
196     // check if the callback execution failed
197     if (status != STATUS_OK)
198     {
199         return Modbus::reportException(status);
200     }
201
202     // writes the response being created
203     return Modbus::writeResponse();
204 }
205
206 /**
207  * Reads and returns the current request message's function code
208  *
```

```
209  * @return a byte containing the current request message function code
210  */
211 uint8_t Modbus::readFunctionCode()
212 {
213     if (_requestBufferLength >= MODBUS_FRAME_SIZE && !_isRequestBufferReading)
214     {
215         return _requestBuffer[MODBUS_FUNCTION_CODE_INDEX];
216     }
217     return FC_INVALID;
218 }
219 
220 /**
221  * Reads and returns the current request message's target unit address
222  *
223  * @return a byte containing the current request message unit address
224  */
225 uint8_t Modbus::readUnitAddress()
226 {
227     if ((_requestBufferLength >= MODBUS_FRAME_SIZE) &&
228       !_isRequestBufferReading)
229     {
230         return _requestBuffer[MODBUS_ADDRESS_INDEX];
231     }
232     return MODBUS_INVALID_UNIT_ADDRESS;
233 }
234 
235 /**
236  * Returns a boolean value indicating if the request currently being processed
237  * is a broadcast message and therefore does not need a response.
238  *
239  * @return true if the current request message is a broadcase message;
240  * otherwise false
241  */
242 bool Modbus::isBroadcast()
243 {
244     return Modbus::readUnitAddress() == MODBUS_BROADCAST_ADDRESS;
245 }
246 
247 /**
248  * Read coil state from input buffer.
249  *
250  * @param offset the coil offset from first coil in this buffer.
251  * @return the coil state from buffer (true / false)
252  */
253 bool Modbus::readCoilFromBuffer(int offset)
254 {
255     if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] == FC_WRITE_COIL)
256     {
257         if (offset == 0)
258         {
259             // (2 x coilAddress, 1 x value)
260             return readUInt16(
```

```
261                 _requestBuffer,
262                 MODBUS_DATA_INDEX + 2) == COIL_ON;
263         }
264         return false;
265     }
266     else if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] ==
267       FC_WRITE_MULTIPLE_COILS)
268     {
269         // (2 x firstCoilAddress, 2 x coilsCount, 1 x valueBytes, n x values)
270         uint16_t index = MODBUS_DATA_INDEX + 5 + (offset / 8);
271         uint8_t bitIndex = offset % 8;
272 
273         // check offset
274         if (index < _requestBufferLength - MODBUS_CRC_LENGTH)
275         {
276             return bitRead(_requestBuffer[index], bitIndex);
277         }
278     }
279     return false;
280 }
281 
282 /**
283  * Read register value from input buffer.
284  *
285  * @param offset the register offset from first register in this buffer.
286  * @return the register value from buffer.
287  */
288 uint16_t Modbus::readRegisterFromBuffer(int offset)
289 {
290     if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] == FC_WRITE_REGISTER)
291     {
292         if (offset == 0)
293         {
294             // (2 x coilAddress, 2 x value)
295             return readUInt16(_requestBuffer, MODBUS_DATA_INDEX + 2);
296         }
297     }
298     else if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] ==
299       FC_WRITE_MULTIPLE_REGISTERS)
300     {
301         // (2 x firstRegisterAddress, 2 x registersCount, 1 x valueBytes,
302         //  n x values)
303         uint16_t index = MODBUS_DATA_INDEX + 5 + (offset * 2);
304 
305         // check offset
306         if (index < _requestBufferLength - MODBUS_CRC_LENGTH)
307         {
308             return readUInt16(_requestBuffer, index);
309         }
310     }
311     return 0;
312 }
```

```
313
314 /**
315  * Writes exception status to buffer
316  *
317  * @param offset the exception status flag.
318  * @param the exception status flag (true / false)
319  */
320 uint8_t Modbus::writeExceptionStatusToBuffer(int offset, bool status)
321 {
322     // check function code
323     if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] !=
324       FC_READ_EXCEPTION_STATUS) {
325         return STATUS_ILLEGAL_DATA_ADDRESS;
326     }
327
328     // (1 x values)
329     uint16_t index = MODBUS_DATA_INDEX;
330     uint8_t bitIndex = offset % 8;
331
332     // check offset
333     if (index >= _responseBufferLength - MODBUS_CRC_LENGTH)
334     {
335         return STATUS_ILLEGAL_DATA_ADDRESS;
336     }
337
338     if (status)
339     {
340         bitSet(_responseBuffer[index], bitIndex);
341     }
342     else
343     {
344         bitClear(_responseBuffer[index], bitIndex);
345     }
346
347     return STATUS_OK;
348 }
349
350 /**
351  * Writes coil state to output buffer.
352  *
353  * @param offset the coil offset from first coil in this buffer.
354  * @param state the coil state to write into buffer (true / false)
355  */
356 uint8_t Modbus::writeCoilToBuffer(int offset, bool state)
357 {
358     // check function code
359     if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] != FC_READ_DISCRETE_INPUT &&
360         _requestBuffer[MODBUS_FUNCTION_CODE_INDEX] != FC_READ_COILS) {
361         return STATUS_ILLEGAL_DATA_ADDRESS;
362     }
363
364     // (1 x valueBytes, n x values)
```

```
365     uint16_t index = MODBUS_DATA_INDEX + 1 + (offset / 8);
366     uint8_t bitIndex = offset % 8;
367
368     // check offset
369     if (index >= _responseBufferLength - MODBUS_CRC_LENGTH)
370     {
371         return STATUS_ILLEGAL_DATA_ADDRESS;
372     }
373
374     if (state)
375     {
376         bitSet(_responseBuffer[index], bitIndex);
377     }
378     else
379     {
380         bitClear(_responseBuffer[index], bitIndex);
381     }
382
383     return STATUS_OK;
384 }
385
386 /**
387  * Writes digital input to output buffer.
388  *
389  * @param offset the input offset from first input in this buffer.
390  * @param state the digital input state to write into buffer (true / false)
391  */
392 uint8_t Modbus::writeDiscreteInputToBuffer(int offset, bool state)
393 {
394     return Modbus::writeCoilToBuffer(offset, state);
395 }
396
397 /**
398  * Writes register value to output buffer.
399  *
400  * @param offset the register offset from first register in this buffer.
401  * @param value the register value to write into buffer.
402  */
403 uint8_t Modbus::writeRegisterToBuffer(int offset, uint16_t value)
404 {
405     // check function code
406     if (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX] !=
407       FC_READ_HOLDING_REGISTERS &&
408         _requestBuffer[MODBUS_FUNCTION_CODE_INDEX] !=
409       FC_READ_INPUT_REGISTERS) {
410         return STATUS_ILLEGAL_DATA_ADDRESS;
411     }
412
413     // (1 x valueBytes, n x values)
414     uint16_t index = MODBUS_DATA_INDEX + 1 + (offset * 2);
415
416     // check offset
```

```
417     if ((index + 2) > (_responseBufferLength - MODBUS_CRC_LENGTH))
418     {
419         return STATUS_ILLEGAL_DATA_ADDRESS;
420     }
421 
422     _responseBuffer[index] = value >> 8;
423     _responseBuffer[index + 1] = value & 0xff;
424 
425     return STATUS_OK;
426 }
427 
428 /**
429  * Writes arbitrary string of uint8_t to output buffer.
430  *
431  * @param offset the register offset from first register in this buffer.
432  * @param str the string to write into buffer.
433  * @param length the string length.
434  * @return STATUS_OK in case of success, STATUS_ILLEGAL_DATA_ADDRESS
435  *      if data doesn't fit in buffer
436  */
437 uint8_t Modbus::writeStringToBuffer(int offset, uint8_t *str, uint8_t length)
438 {
439     // (1 x valueBytes, n x values)
440     uint8_t index = MODBUS_DATA_INDEX + 1 + (offset * 2);
441 
442     // check string length.
443     if ((index + length) > _responseBufferLength - MODBUS_CRC_LENGTH)
444     {
445         return STATUS_ILLEGAL_DATA_ADDRESS;
446     }
447 
448     memcpy(_responseBuffer + index, str, length);
449 
450     return STATUS_OK;
451 }
452 
453 /**
454  * -------------------------------------------------
455  *                  PRIVATE METHODS
456  * -------------------------------------------------
457  */
458 
459 /**
460  * Reads a new request from the serial stream and fills the request buffer
461  *
462  * @return true if the buffer is filled with a request and is ready to
463  * be processed; otherwise false.
464  */
465 bool Modbus::readRequest()
466 {
467     /**
468      * Read one data packet and report when received completely
```

```
469       */
470      uint16_t lenght = _serialStream.available();
471      if (lenght > 0)
472      {
473          // if not yet started reading
474          if (!_isRequestBufferReading)
475          {
476              // And it already took 1.5T from the last message
477              if ((micros() - _lastCommunicationTime) >
478                (_halfCharTimeInMicroSecond * MODBUS_HALF_SILENCE_MULTIPLIER))
479              {
480                  // start reading and clear buffer
481                  _requestBufferLength = 0;
482                  _isRequestBufferReading = true;
483              }
484              else
485              {
486                  // discard data
487                  _serialStream.read();
488              }
489          }
490
491          // if already in reading
492          if (_isRequestBufferReading)
493          {
494              if (_requestBufferLength == MODBUS_MAX_BUFFER)
495              {
496                  // buffer is already full; stop reading
497                  _isRequestBufferReading = false;
498              }
499
500              // add new bytes to buffer
501              lenght = min(lenght, MODBUS_MAX_BUFFER - _requestBufferLength);
502              lenght = _serialStream.readBytes(
503                _requestBuffer + _requestBufferLength,
504                MODBUS_MAX_BUFFER - _requestBufferLength);
505
506              // if this is the first read,
507              // check the address to reject irrelevant requests
508              if (_requestBufferLength == 0 &&
509                  lenght > MODBUS_ADDRESS_INDEX &&
510                  (_requestBuffer[MODBUS_ADDRESS_INDEX] != _unitAddress &&
511                    _requestBuffer[MODBUS_ADDRESS_INDEX] !=
512                    MODBUS_BROADCAST_ADDRESS))
513              {
514                  // bad address, stop reading
515                  _isRequestBufferReading = false;
516              }
517
518              // move byte pointer forward
519              _requestBufferLength += lenght;
520              _totalBytesReceived += lenght;
```

```
521         }
522
523         // save the time of last received byte(s)
524         _lastCommunicationTime = micros();
525
526         // wait for more data
527         return false;
528     }
529     else
530     {
531         // if we are in reading but no data is available for 1.5T;
532         // this request is completed
533         if (_isRequestBufferReading && ((micros() - _lastCommunicationTime) >
534           (_halfCharTimeInMicroSecond * MODBUS_HALF_SILENCE_MULTIPLIER)))
535         {
536             // allow for new requests to be processed
537             _isRequestBufferReading = false;
538         }
539         else
540         {
541             // otherwise, wait
542             return false;
543         }
544     }
545
546     return !_isRequestBufferReading &&
547       (_requestBufferLength >= MODBUS_FRAME_SIZE);
548 }
549
550 /**
551  * Validates the request message currently in the input buffer.
552  *
553  * @return true if the request is valid; otherwise false
554  */
555 bool Modbus::validateRequest()
556 {
557     // minimum buffer size (1 x Address, 1 x Function, n x Data, 2 x CRC)
558     uint16_t expected_requestBufferSize = MODBUS_FRAME_SIZE;
559     // check data validity based on the function code
560     switch (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX])
561     {
562     case FC_READ_EXCEPTION_STATUS:
563         // broadcast is not supported
564         if (_requestBuffer[MODBUS_ADDRESS_INDEX] == MODBUS_BROADCAST_ADDRESS)
565         {
566             // ignore
567             return false;
568         }
569         break;
570     case FC_READ_COILS:             // read coils (digital read)
571     case FC_READ_DISCRETE_INPUT:    // read input state (digital read)
572     case FC_READ_HOLDING_REGISTERS: // read holding registers (analog read)
```

```
573     case FC_READ_INPUT_REGISTERS:   // read input registers (analog read)
574         // broadcast is not supported
575         if (_requestBuffer[MODBUS_ADDRESS_INDEX] == MODBUS_BROADCAST_ADDRESS)
576         {
577             // ignore
578             return false;
579         }
580         // (2 x Index, 2 x Count)
581         expected_requestBufferSize += 4;
582         break;
583     case FC_WRITE_COIL:     // write coils (digital write)
584     case FC_WRITE_REGISTER: // write regosters (digital write)
585         // (2 x Index, 2 x Count)
586         expected_requestBufferSize += 4;
587         break;
588     case FC_WRITE_MULTIPLE_COILS:
589     case FC_WRITE_MULTIPLE_REGISTERS:
590         // (2 x Index, 2 x Count, 1 x Bytes)
591         expected_requestBufferSize += 5;
592         if (_requestBufferLength >= expected_requestBufferSize)
593         {
594             // (n x Bytes)
595             expected_requestBufferSize += _requestBuffer[6];
596         }
597         break;
598     default:
599         // unknown command
600         Modbus::reportException(STATUS_ILLEGAL_FUNCTION);
601         return false;
602     }
603
604     if (_requestBufferLength < expected_requestBufferSize)
605     {
606         // data is smaller than expected, ignore
607         return false;
608     }
609
610     // set correct data size
611     _requestBufferLength = expected_requestBufferSize;
612
613     // check crc
614     uint16_t crc = readCRC(_requestBuffer, _requestBufferLength);
615     if (Modbus::calculateCRC(
616       _requestBuffer,
617       _requestBufferLength - MODBUS_CRC_LENGTH) != crc)
618     {
619         // ignore
620         return false;
621     }
622
623     return true;
624 }
```

```
625
626 /**
627  * Fills the output buffer with the response to the request already in the
628  * input buffer.
629  *
630  * @return the status code representing the success of this operation
631  */
632 uint8_t Modbus::createResponse()
633 {
634     uint16_t firstAddress;
635     uint16_t addressesLength;
636     uint8_t callbackIndex;
637
638     /**
639      * Match the function code with a callback and execute it
640      * as well as preparing the response buffer
641      */
642     switch (_requestBuffer[MODBUS_FUNCTION_CODE_INDEX])
643     {
644     case FC_READ_EXCEPTION_STATUS:
645         // add response data length to output buffer length
646         _responseBufferLength += 1;
647
648         // execute callback and return the status code
649         return Modbus::executeCallback(CB_READ_EXCEPTION_STATUS, 0, 8);
650     case FC_READ_COILS:          // read coils (digital out state)
651     case FC_READ_DISCRETE_INPUT: // read input state (digital in)
652         // read the the first input address and the number of inputs
653         firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
654         addressesLength = readUInt16(_requestBuffer, MODBUS_DATA_INDEX + 2);
655
656         // calculate response data length and add to output buffer length
657         _responseBuffer[MODBUS_DATA_INDEX] =
658           (addressesLength / 8) + (addressesLength % 8 != 0);
659         _responseBufferLength += 1 + _responseBuffer[MODBUS_DATA_INDEX];
660
661         // execute callback and return the status code
662         callbackIndex = _requestBuffer[MODBUS_FUNCTION_CODE_INDEX] ==
663           FC_READ_COILS ? CB_READ_COILS : CB_READ_DISCRETE_INPUTS;
664         return Modbus::executeCallback(
665           callbackIndex, firstAddress, addressesLength);
666     case FC_READ_HOLDING_REGISTERS:// read holding registers (analog out state)
667     case FC_READ_INPUT_REGISTERS:  // read input registers (analog in)
668         // read the starting address and the number of inputs
669         firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
670         addressesLength = readUInt16(_requestBuffer, MODBUS_DATA_INDEX + 2);
671
672         // calculate response data length and add to output buffer length
673         _responseBuffer[MODBUS_DATA_INDEX] = 2 * addressesLength;
674         _responseBufferLength += 1 + _responseBuffer[MODBUS_DATA_INDEX];
675
676         // execute callback and return the status code
```

```
677           callbackIndex = _requestBuffer[MODBUS_FUNCTION_CODE_INDEX] ==
678             FC_READ_HOLDING_REGISTERS ?
679             CB_READ_HOLDING_REGISTERS :
680             CB_READ_INPUT_REGISTERS;
681           return Modbus::executeCallback(
682             callbackIndex,
683             firstAddress,
684             addressesLength);
685       case FC_WRITE_COIL: // write one coil (digital out)
686           // read the address
687           firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
688
689           // add response data length to output buffer length
690           _responseBufferLength += 4;
691           // copy parts of the request data that need to be in the response data
692           memcpy(
693             _responseBuffer + MODBUS_DATA_INDEX,
694             _requestBuffer + MODBUS_DATA_INDEX,
695             _responseBufferLength - MODBUS_FRAME_SIZE);
696
697           // execute callback and return the status code
698           return Modbus::executeCallback(CB_WRITE_COILS, firstAddress, 1);
699       case FC_WRITE_REGISTER:
700           // read the address
701           firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
702
703           // add response data length to output buffer length
704           _responseBufferLength += 4;
705           // copy parts of the request data that need to be in the response data
706           memcpy(
707             _responseBuffer + MODBUS_DATA_INDEX,
708             _requestBuffer + MODBUS_DATA_INDEX,
709             _responseBufferLength - MODBUS_FRAME_SIZE);
710
711           // execute callback and return the status code
712           return Modbus::executeCallback(
713             CB_WRITE_HOLDING_REGISTERS,
714             firstAddress,
715             1);
716       case FC_WRITE_MULTIPLE_COILS: // write coils (digital out)
717           // read the starting address and the number of outputs
718           firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
719           addressesLength = readUInt16(_requestBuffer, MODBUS_DATA_INDEX + 2);
720
721           // add response data length to output buffer length
722           _responseBufferLength += 4;
723           // copy parts of the request data that need to be in the response data
724           memcpy(
725             _responseBuffer + MODBUS_DATA_INDEX,
726             _requestBuffer + MODBUS_DATA_INDEX,
727             _responseBufferLength - MODBUS_FRAME_SIZE);
728
```

```
729         // execute callback and return the status code
730         return Modbus::executeCallback(
731           CB_WRITE_COILS,
732           firstAddress,
733           addressesLength);
734     case FC_WRITE_MULTIPLE_REGISTERS: // write holding registers (analog out)
735         // read the starting address and the number of outputs
736         firstAddress = readUInt16(_requestBuffer, MODBUS_DATA_INDEX);
737         addressesLength = readUInt16(_requestBuffer, MODBUS_DATA_INDEX + 2);
738
739         // add response data length to output buffer length
740         _responseBufferLength += 4;
741         // copy parts of the request data that need to be in the response data
742         memcpy(
743           _responseBuffer + MODBUS_DATA_INDEX,
744           _requestBuffer + MODBUS_DATA_INDEX,
745           _responseBufferLength - MODBUS_FRAME_SIZE);
746
747         // execute callback and return the status code
748         return Modbus::executeCallback(
749           CB_WRITE_HOLDING_REGISTERS,
750           firstAddress,
751           addressesLength);
752     default:
753         return STATUS_ILLEGAL_FUNCTION;
754     }
755 }
756
757 /**
758  * Executes a callback
759  *
760  * @return the status code representing the success of this operation
761  */
762 uint8_t Modbus::executeCallback(
763   uint8_t callbackIndex,
764   uint16_t address,
765   uint16_t length)
766 {
767     if (cbVector[callbackIndex])
768     {
769         return cbVector[callbackIndex](
770           Modbus::readFunctionCode(), address, length);
771     }
772     else
773     {
774         return STATUS_ILLEGAL_FUNCTION;
775     }
776 }
777
778 /**
779  * Writes the output buffer to serial stream
780  *
```

```
781   * @return The number of bytes written
782   */
783  uint16_t Modbus::writeResponse()
784  {
785      /**
786       * Validate
787       */
788
789      // check if there is a response and this is supposed to be the first write
790      if (
791        _responseBufferWriteIndex == 0 &&
792        _responseBufferLength >= MODBUS_FRAME_SIZE) {
793          // set status as writing
794          _isResponseBufferWriting = true;
795      }
796
797      // check if we are not in writing or the address is broadcast
798      if (
799        !_isResponseBufferWriting ||
800        _responseBuffer[MODBUS_ADDRESS_INDEX] == MODBUS_BROADCAST_ADDRESS) {
801          // cleanup and ignore
802          _isResponseBufferWriting = false;
803          _responseBufferWriteIndex = 0;
804          _responseBufferLength = 0;
805          return 0;
806      }
807
808      /**
809       * Preparing
810       */
811
812      // if this is supposed to be the first write
813      if (_responseBufferWriteIndex == 0) {
814          // if we still need to wait
815          if (
816            (micros() - _lastCommunicationTime) <=
817            (_halfCharTimeInMicroSecond * MODBUS_HALF_SILENCE_MULTIPLIER))
818          {
819              // ignore
820              return 0;
821          }
822
823          // calculate and fill crc
824          uint16_t crc = Modbus::calculateCRC(
825            _responseBuffer,
826            _responseBufferLength - MODBUS_CRC_LENGTH);
827          _responseBuffer[_responseBufferLength - MODBUS_CRC_LENGTH] =
828            crc & 0xff;
829          _responseBuffer[(_responseBufferLength - MODBUS_CRC_LENGTH) + 1] =
830            crc >> 8;
831
832          // enter transmission mode
```

```
833         if (_transmissionControlPin > MODBUS_CONTROL_PIN_NONE) {
834             digitalWrite(_transmissionControlPin, HIGH);
835         }
836     }
837
838     /**
839      * Transmit
840      */
841
842     // send buffer
843     uint16_t length = 0;
844     if (_serialTransmissionBufferLength > 0) {
845         uint16_t length = min(
846             _serialStream.availableForWrite(),
847             _responseBufferLength - _responseBufferWriteIndex
848         );
849
850         if (length > 0) {
851             length = _serialStream.write(
852                 _responseBuffer + _responseBufferWriteIndex,
853                 length
854             );
855             _responseBufferWriteIndex += length;
856             _totalBytesSent += length;
857         }
858
859         if (
860           _serialStream.availableForWrite() <
861           _serialTransmissionBufferLength)
862         {
863             // still waiting for write to complete
864             _lastCommunicationTime = micros();
865             return length;
866         }
867
868         // if buffer reports as empty; make sure it really is
869         // (`Serial` removes bytes from buffer before sending them)
870         _serialStream.flush();
871     } else {
872         // compatibility for badly written software serials; aka AltSoftSerial
873         length = _responseBufferLength - _responseBufferWriteIndex;
874
875         if (length > 0) {
876             length = _serialStream.write(_responseBuffer, length);
877             _serialStream.flush();
878         }
879
880         _responseBufferWriteIndex += length;
881         _totalBytesSent += length;
882     }
883
884     if (_responseBufferWriteIndex >= _responseBufferLength &&
```

```
885           (micros() - _lastCommunicationTime) >
886         (_halfCharTimeInMicroSecond * MODBUS_HALF_SILENCE_MULTIPLIER)) {
887
888           // end transmission
889           if (_transmissionControlPin > MODBUS_CONTROL_PIN_NONE) {
890               digitalWrite(_transmissionControlPin, LOW);
891           }
892
893           // cleanup
894           _isResponseBufferWriting = false;
895           _responseBufferWriteIndex = 0;
896           _responseBufferLength = 0;
897       }
898
899       return length;
900   }
901
902   /**
903    * Fills the output buffer with an exception in regard to the request already
904    * in the input buffer and writes the response. No need to do it later.
905    *
906    * @param exceptionCode the status code to report.
907    * @return the number of bytes written
908    */
909   uint16_t Modbus::reportException(uint8_t exceptionCode)
910   {
911       // we don't respond to broadcast messages
912       if (_requestBuffer[MODBUS_ADDRESS_INDEX] == MODBUS_BROADCAST_ADDRESS)
913       {
914           return 0;
915       }
916       _responseBufferLength = MODBUS_FRAME_SIZE + 1;
917       _responseBuffer[MODBUS_FUNCTION_CODE_INDEX] |= 0x80;
918       _responseBuffer[MODBUS_DATA_INDEX] = exceptionCode;
919
920       return Modbus::writeResponse();
921   }
922
923   /**
924    * Calculates the CRC of the passed byte array from zero up to the
925    * passed length.
926    *
927    * @param buffer the byte array containing the data.
928    * @param length the length of the byte array.
929    *
930    * @return the calculated CRC as an unsigned 16 bit integer.
931    */
932   uint16_t Modbus::calculateCRC(uint8_t *buffer, int length)
933   {
934       int i, j;
935       uint16_t crc = 0xFFFF;
936       uint16_t tmp;
```

```
937
938     // calculate crc
939     for (i = 0; i < length; i++)
940     {
941         crc = crc ^ buffer[i];
942
943         for (j = 0; j < 8; j++)
944         {
945             tmp = crc & 0x0001;
946             crc = crc >> 1;
947             if (tmp)
948             {
949                 crc = crc ^ 0xA001;
950             }
951         }
952     }
953
954     return crc;
955 }
```