# پروژه Slave Modbus

MODBUS © Protocol یک ساختار پیام رسان است ، که به طور گسترده برای ایجاد ارتباطات master-slave بین دستگاه های هوشمند استفاده می شود. یک پیام MODBUS که از یک مستر به یک اسلیو ارسال می شود ، حاوی آدرس اسلیو ، "دستور" (به عنوان مثال "خواندن" یا "نوشتن") ، داده ها و یک جمع (LRC یا CRC) است.

از آنجا که پروتکل Modbus فقط یک ساختار پیام رسانی است ، مستقل از لایه فیزیکی زیرین است. به طور سنتی با استفاده از RS232 ، RS422 یا RS485 اجرا می شود.

|  |  |  |  |
| --- | --- | --- | --- |
| Object type | Access | Size | Address Space |
| Coil | Read-write | 1 bit | 00001 - 09999 |
| Discrete input | Read-only | 1 bit | 10001 - 19999 |
| Input register | Read-only | 16 bits | 30001 - 39999 |
| Holding register | Read-write | 16 bits | 40001 - 49999 |

## درخواست

کد عملکرد در درخواست به دستگاه اسلیو آدرس داده شده می گوید که چه نوع عملی را انجام دهد. بایت داده ها حاوی هرگونه اطلاعات اضافی است که اسلیو برای انجام عملکرد نیاز دارد. به عنوان مثال ، کد عملکرد 03 از اسلیو درخواست می کند تا رجیسترهای نگهدارنده را بخواند و با محتوای آنها پاسخ دهد. قسمت داده باید حاوی اطلاعاتی باشد که به اسلیو می گوید از کدام رجیستر شروع کرده و چه تعداد رجیستر را باید بخواند. قسمت بررسی خطا روشی را برای تأیید صحت محتوای پیام برای اسلیو فراهم می کند.

## پاسخ

اگر اسلیو پاسخ عادی بدهد ، کد عملکرد در پاسخ ، پژواک کد عملکرد در درخواست است. بایت های داده حاوی داده های جمع آوری شده توسط اسلیو مانند مقادیر یا وضعیت رجیستر هستند. در صورت بروز خطا ، کد عملکرد تغییر داده می شود تا نشان دهد پاسخ یک پاسخ خطاست و بایت داده ها حاوی کدی است که خطا را توصیف می کند. قسمت بررسی خطا به مستر اجازه می دهد اعتبار محتوای پیام را تأیید کند.

کنترل کننده ها می توانند برای برقراری ارتباط در شبکه های استاندارد Modbus با استفاده از هر دو حالت انتقالASCII یا RTU تنظیم شوند.

## حالت ASCII

وقتی کنترل کننده ها برای برقراری ارتباط در شبکه Modbus با استفاده از حالت ASCII (کد استاندارد آمریکایی برای تبادل اطلاعات) تنظیم می شوند ، هر بایت هشت بیتی در یک پیام به عنوان دو کاراکتر ASCII ارسال می شود. مزیت اصلی این حالت این است که اجازه می دهد فواصل زمانی حداکثر تا یک ثانیه بین کاراکترها بدون ایجاد خطا رخ دهد.

## سیستم کدگذاری

نویسه های قابل چاپ هگزادسیمال ASCII 0 ... 9 ، A ... F

بیت در هر بایت

1 بیت شروع

7 بیت داده ، حداقل بیت مهم ابتدا ارسال می شود

1 بیت برای برابری زوج / فرد - بدون بیت برای هیچ برابری

در صورت استفاده از برابری 1 بیت-در صورت عدم برابری 2 بیت

بررسی خطا(LRC)

## کدهای عملکردی

**Function 01 (01hex) Read Coils**

Reads the ON/OFF status of discrete coils in the slave.

**Request**  
The request message specifies the starting coil and quantity of coils to be read.Example of a request to read 10...22 (Coil 11 to 23) from slave device address 4:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 04 | 0 4 |
| Function | 01 | 0 1 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 0A | 0 A |
| Quantity of Coils Hi | 00 | 0 0 |
| Quantity of Coils Lo | 0D | 0 D |
| Error Check Lo | DD | LRC (E 4) |
| Error Check Hi | 98 |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The coil status response message is packed as one coil per bit of the data field. Status is indicated as: 1 is the value ON, and 0 is the value OFF. The LSB of the first data byte contains the coil addressed in the request. The other coils follow toward the high-order end of this byte and from low order to high order in subsequent bytes. If the returned coil quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeroes (toward the high-order end of the byte). The byte count field specifies the quantity of complete bytes of data.

Example of a response to the request:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 04 | 0 4 |
| Function | 01 | 0 1 |
| Byte Count | 02 | 0 2 |
| Data (Coils 7...10) | 0A | 0 A |
| Data (Coils 27...20) | 11 | 1 1 |
| Error Check Lo | B3 | LRC (D E) |
| Error Check Hi | 50 | None |
| Trailer | None | CR LF |
| Total Bytes | 7 | 15 |

**Function 02(02hex) Read Discrete Inputs**

Reads the ON/OFF status of discrete inputs in the slave.

**Request**The request message specifies the starting input and quantity of inputs to be read.

Example of a request to read 10...22 (input 10011 to 10023) from slave device address 4:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 04 | 0 4 |
| Function | 02 | 0 2 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 0A | 0 A |
| Quantity of inputs Hi | 00 | 0 0 |
| Quantity of inputs Lo | 0D | 0 D |
| Error Check Lo | 99 | LRC (E 3) |
| Error Check Hi | 98 |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The input status response message is packed as one input per bit of the data field. Status is indicated as: 1 is the value ON, and 0 is the value OFF. The LSB of the first data byte contains the input addressed in the request. The other inputs follow toward the high-order end of this byte and from low order to high order in subsequent bytes. If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeroes (toward the high-order end of the byte). The byte count field specifies the quantity of complete bytes of data.

Example of a response to the request:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 04 | 0 4 |
| Function | 02 | 0 2 |
| Byte Count | 02 | 0 2 |
| Data (Inputs 17...10) | 0A | 0 A |
| Data (Inputs 27...20) | 11 | 1 1 |
| Error Check Lo | B3 | LRC (D D) |
| Error Check Hi | 14 | None |
| Trailer | None | CR LF |
| Total Bytes | 7 | 15 |

**Function 03 (03hex) Read Holding Registers**

Read the binary contents of holding registers in the slave.

**Request**The request message specifies the starting register and quantity of registers to be read.

Example of a request to read 0...1 (register 40001 to 40002) from slave device 1:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 01 | 0 1 |
| Function | 03 | 0 3 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 00 | 0 0 |
| Quantity of Registers Hi | 00 | 0 0 |
| Quantity of Registers Lo | 02 | 0 2 |
| Error Check Lo | C4 | LRC (F A) |
| Error Check Hi | 0B |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register the first byte contains the high-order bits, and the second contains the low-order bits.

Example of a response to the request:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 01 | 0 1 |
| Function | 03 | 0 3 |
| Byte Count | 04 | 0 4 |
| Data Hi | 00 | 0 0 |
| Data Lo | 06 | 0 6 |
| Data Hi | 00 | 0 0 |
| Data Lo | 05 | 0 5 |
| Error Check Lo | DA | LRC (E D) |
| Error Check Hi | 31 | None |
| Trailer | None | CR LF |
| Total Bytes | 8 | 19 |

**Function 04 (04hex) Read Input Registers**

Read the binary contents of input registers in the slave.

**Request**The request message specifies the starting register and quantity of registers to be read.

Example of a request to read 0...1 (register 30001 to 30002) from slave device 1:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 01 | 0 1 |
| Function | 04 | 0 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 00 | 0 0 |
| Quantity of Registers Hi | 00 | 0 0 |
| Quantity of Registers Lo | 02 | 0 2 |
| Error Check Lo | 71 | LRC (F 9) |
| Error Check Hi | CB |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register the first byte contains the high-order bits, and the second contains the low-order bits.

Example of a response to the request:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 01 | 0 1 |
| Function | 04 | 0 4 |
| Byte Count | 04 | 0 4 |
| Data Hi | 00 | 0 0 |
| Data Lo | 06 | 0 6 |
| Data Hi | 00 | 0 0 |
| Data Lo | 05 | 0 5 |
| Error Check Lo | DB | LRC (E C) |
| Error Check Hi | 86 | None |
| Trailer | None | CR LF |
| Total Bytes | 9 | 19 |

**Function 05 (05hex) Write Single Coil**

Writes a single coil to either ON or OFF.

**Request**The request message specifies the coil reference to be written. Coils are addressed starting at zero-coil 1 is addressed as 0.

The requested ON / OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the coil to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the coil.

Here is an example of a request to write coil 173 ON in slave device 17:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 05 | 0 5 |
| Coil Address Hi | 00 | 0 0 |
| Coil Address Lo | AC | A C |
| Write Data Hi | FF | 0 0 |
| Write Data Lo | 00 | F F |
| Error Check Lo | 4E | LRC (3 F) |
| Error Check Hi | 8B |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The normal response is an echo of the request, returned after the coil state has been written.  
Example of a response to the request:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 05 | 0 5 |
| Coil Address Hi | 00 | 0 0 |
| Coil Address Lo | AC | A C |
| Write Data Hi | FF | 0 0 |
| Write Data Lo | 00 | F F |
| Error Check Lo | 4E | LRC (3 F) |
| Error Check Hi | 8B |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Function 06 (06hex) Write Single Register**

Writes a value into a single holding register.

**Request**The request message specifies the register reference to be Written. Registers are addressed starting at zero-register 1 is addressed as 0.

The requested Write value is specified in the request data field. Here is an example of a request to Write register 40002 to 00 03 hex in slave device 17.

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 06 | 0 6 |
| Register Address Hi | 00 | 0 0 |
| Register Address Lo | 01 | 0 1 |
| Write Data Hi | 00 | 0 0 |
| Write Data Lo | 03 | 0 3 |
| Error Check Lo | 9A | LRC (E 5) |
| Error Check Hi | 9B |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Response**The normal response is an echo of the request, returned after the register contents have been written.

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 06 | 0 6 |
| Coil Address Hi | 00 | 0 0 |
| Coil Address Lo | 01 | 0 1 |
| Write Data Hi | 00 | 0 0 |
| Write Data Lo | 03 | 0 3 |
| Error Check Lo | 9A | LRC (E 5) |
| Error Check Hi | 9B |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Function 15 (0Fhex) Write Multiple Coils**

Writes each coil in a sequence of coils to either ON or OFF.

**Request**The request message specifies the coil references to be written. Coils are addressed starting at zero-coil 1 is addressed as 0.

The requested ON / OFF states are specified by contents of the request data field. A logical 1 in a bit position of the field requests the corresponding coils to be ON. A logical 0 requests it to be OFF.

Below is an example of a request to write a series of ten coils starting at coil 20 (addressed as 19, or 13 hex) in slave device 17.

The request data contents are two bytes: CD 01 hex (1100 1101 0000 0001 binary). The binary bits correspond to the coils in the following way:

**Bit:** 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1

**Coil:** 27 26 25 24 23 22 21 20 - - - - - - 29 28

The first byte transmitted (CD hex) addresses coils 27 ... 20, with the least significant bit addressing the lowest coil (20) in this set.

The next byte transmitted (01 hex) addresses coils 29 and 28, with the least significant bit addressing the lowest coil (28) in this set. Unused bits in the last data byte should be zero-filled.

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 0F | 0 F |
| Coil Address Hi | 00 | 0 0 |
| Coil Address Lo | 13 | 1 3 |
| Quantity of Coils Hi | 00 | 0 0 |
| Quantity of Coils Lo | 0A | 0 A |
| Byte Count | 02 | 0 2 |
| Write Data Hi | CD | C D |
| Write Data Lo | 01 | 0 1 |
| Error Check Lo | BF | LRC (F 3) |
| Error Check Hi | 0B |  |
| Trailer | None | CR LF |
| Total Bytes | 11 | 23 |

**Response**The normal response returns the slave address, function code, starting address, and number of coils written. Here is an example of a response to the request shown above

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 0F | 0 F |
| Coil Address Hi | 00 | 0 0 |
| Coil Address Lo | 13 | 1 3 |
| Quantity of Coils Hi | 00 | 0 0 |
| Quantity of Coils Lo | 0A | 0 A |
| Error Check Lo | 26 | LRC (C 3) |
| Error Check Hi | 99 |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**Function 16 (10hex) Write Multiple Registers**

Writes values into a sequence of holding registers

**Request**The request message specifies the register references to be written. Registers are addressed starting at zero-register 1 is addressed as 0.

The requested write values are specified in the request data field. Data is packed as two bytes per register.

Here is an example of a request to write two registers starting at 40002 to 00 0A and 01 02 hex, in slave device 17:

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 10 | 1 0 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 01 | 0 1 |
| Quantity of Registers Hi | 00 | 0 0 |
| Quantity of Registers Lo | 02 | 0 2 |
| Byte Count | 04 | 0 4 |
| Data Hi | 00 | 0 0 |
| Data Lo | 0A | 0 A |
| Data Hi | 01 | 0 1 |
| Data Lo | 02 | 0 2 |
| Error Check Lo | C6 | LRC (C B) |
| Error Check Hi | F0 |  |
| Trailer | None | CR LF |
| Total Bytes | 13 | 23 |

**Response**The normal response returns the slave address, function code, starting address, and quantity of registers written. Here is an example of a response to the request shown above.

|  |  |  |
| --- | --- | --- |
| Field Name | RTU (hex) | ASCII Characters |
| Header | None | : (Colon) |
| Slave Address | 11 | 1 1 |
| Function | 10 | 1 0 |
| Starting Address Hi | 00 | 0 0 |
| Starting Address Lo | 01 | 0 1 |
| Quantity of Registers Hi | 00 | 0 0 |
| Quantity of Registers Lo | 02 | 0 2 |
| Error Check Lo | 12 | LRC (D C) |
| Error Check Hi | 98 |  |
| Trailer | None | CR LF |
| Total Bytes | 8 | 17 |

**LRC Example Code**

This function is an example how to calculate a LRC BYTE using the C language.  
  
**BYTE LRC (BYTE \*nData, WORD wLength)  
{  
BYTE nLRC = 0 ; // LRC char initialized  
  
for (int i = 0; i < wLength; i++)  
nLRC += \*nData++;  
  
return (BYTE)(-nLRC);  
  
} // End: LRC**

# ساختار کد slave ASCII

برای دریافت داده ها به از طریق پورت سریال سه سناریو مورد بررسی قرار گرفت. سناریوی اول پولینگ کردن روی پورت سریال تا زمانی که درخواستی از طرف مستر به اسلیو ارسال شود داده ها دریافت شود و سپس پردازش گردد.

در این سناریو به علت اینکه بافر سخت افزاری روی پرت سریال وجود ندارد باید سرعت پولینگ کردن پرت سریال از زمان مورد نیاز برای دریافت بایت بعدی بیشتر باشد در غیر اینصورت احتمال از دست دادن داده ها در پورت سریال بسیار بالاست. همچنین به علت پولینگ کردن دائم امکان انجام پردازش های دیگر توسط CPU از دست خواهد رفت.

در سناریوی بعدی امکان دریافت داده ها از طریق فعال کردن اینتراپت روی پرت سریال مورد بررسی قرار گرفت. بدین ترتیب که هر داده ای توسط پرت سریال دریافت شد اینتراپت فعال شود و داده دریافتی قبل از دریافت بایت دیگر خوانده شود مزیت این روش این است که نیازی به اشغال کردن CPU به علت پولینگ وجود ندارد اما باید فرایندهایی برای تشخیص شروع و پایان ان پس از دریافت هر بایت انجام گیرد که همین باعث اشغال شدن CPU میشود.

در سناریوی سوم بدون استفاده از پردازشگر داده ها با استفاده از DMA به حافظه منتقل می شوند واین فرایند تا زمانی که پرت سریال به حالت بیکار میرسد ادامه می یابد مزیت استفاده از این روش ان است که تمام فرایند دریافت درخواست از طریق سخت افزار انجام میگرد و نرم افزار فقط وظیفه تفسیر داده دریافتی را داراست. در برنامه نوشته شده از روش سوم استفاده شده است.

HAL\_UARTEx\_ReceiveToIdle\_DMA(&huart1, mbBuffer, ASCII\_FRAME\_SIZE);

داده ی دریافتی سپس از طریق کال بک زیر

**void** **HAL\_UARTEx\_RxEventCallback**(UART\_HandleTypeDef \*huart, uint16\_t Size){

HAL\_UARTEx\_ReceiveToIdle\_DMA(&huart1, mbBuffer, ASCII\_FRAME\_SIZE);

execute\_modbus\_command(mbBuffer,Size);

}

بوسیله فانکشن execute\_modbus\_command پردازش شده و پاسخ متناسب به مستر از طریق پورت سریال ارسال میگردد.

برای سادگی برنامه عملیات پردازش درون روتین وقفه نوشته شده اما در صورت نیاز میتوان ان را به بک گروند(تسک سیستم عامل) منتقل کرد. اما باید مراقب بود که زمان عملیات پردازش و ارسال پاسخ بیش از تایم اوت مستر نشود.

در صورت انجام عملیاتی که به زمان بندی دقیق و تسک ها اولویت بندی دقیق دارند، می توان تابع پردازش را به یک تسک منتقل کرد و آن را با داده دریافتی کال بک بالا از طریق یک فلگ یا سمافور سینکرونایز کرد.

فانکشن پردازش (execute\_modbus\_command)

|  |  |
| --- | --- |
| اگر طول داده دریافتی از 9 کمتر باشد به معنی این است که داده ناقص دریافت شده است بنابراین از ادامه عملیات پردازش صرفنظر میکنیم | **if** ( size < 9 )  {  clear\_frame(buffer); // bad frame discard , minimun 9 characters for valid  **return**;  } |
| مقدار LRC پکت دریافتی را محاسبه میکند و با مقدار خود پکت مقایسه میکند در صورتی که این مقادیر با هم برابر نباشند از ادامه عملیات پردازش صرفنظر میشود در غیر اینصورت به مرحله بعد میرویم. | lrc = AsciiToByte(buffer[size-2], buffer[size-1]);  size -= 2; //skip lrc  fun = lrc\_calc(buffer,size);  **if** ( fun != lrc )  {  clear\_frame(buffer); // bad LRC, frame discard  **return**;  } |
| در این مرحله ادرس پکت دریافتی محاسبه و با آدرس دیوایس مقایسه می شود در صورتی که آدرس صفر باشد به معنی این است که پیغامی برای تمامی دیوایس های موجود در شبکه ارسال شده و متغیر broadcast=1 می شود. در غیر اینصورت در مرحله بعدی ادرس پکت ارسالی با ادرس دیوایس چک شده و درصورت برابر بودن به مرحله بعد میرویم در غیر اینصورت از ادامه برنامه صرفنظر میکنیم | /\* frame ok \*/  is\_me = AsciiToByte(buffer[1], buffer[2]);  **if** ( is\_me == 0 )  {  broadcast = 1;  }  **else**  {  broadcast = 0;  **if** ( DIR != is\_me )  {  clear\_frame(buffer);  **return**;  }  } |
| در این قسمت مشخصات پکت ارسالی مانند ادرس شروع و پایان عملیات و نوع فانکشن مشخص میشود. | mb=getModbusInfo(buffer, size);  // response buffer start(1)+address(2)+protocol(2)  **for**( sendIndex=0;sendIndex<5;sendIndex++){  sendBuffer[sendIndex]=buffer[sendIndex];//copy request header to response header  }  fun = AsciiToByte(buffer[3], buffer[4]); |
| در این قسمت با توجه به نوع فانکشن پکت، فانکشن پردازشی متناسب با آن اجرا میشود. | **switch** ( fun )  {  **case** 1:  {  sendIndex=ResponseReadCoilsStatus\_01(buffer,size,mb);  **break**;  }  **case** 2:  {  sendIndex=ResponseReadInputStatus\_02(buffer,size,mb);  **break**;  }  **case** 3:  {  sendIndex=ResponseReadHoldingRegisters\_03(buffer,size,mb);  **break**;  }  **case** 4:  {  sendIndex=ResponseReadInputRegisters\_04(buffer,size,mb);  **break**;  }  **case** 5:  {  sendIndex=ResponseForceSingleCoil\_05(buffer,size);  **break**;  }  **case** 6:  {  sendIndex=ResponsePresetSingleRegister\_06(buffer,size);  **break**;  }  **case** 15:  {  ResponseForceMultipleCoils\_15(buffer,size,mb);  **break**;  }  **case** 16:  {  sendIndex=ResponsePresetMultipleRegisters\_16(buffer,size,mb);  **break**;  }  **default**:  {  **break**;  }  } |
| در این قسمت پس از تولید پاسخ متناسب با درخواست LRC تولید و بافر ارسال را با ان پر میکنیم سپس بافر ارسال به مستر ارسال می شود. | //generate lrc and fill lrcf at the end of send buffer  gen\_lrc(sendBuffer,sendIndex);  sendIndex+=4;//add 4 bytes to size of buffer  //transfer send buffer asynchronously by DMA  tx\_ascii\_frame(sendBuffer,sendIndex); |

# Memory Mapping

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| نوع متغیر | آدرس | متغیرهای 16 بیتی درون رم STM32  holdingRegister  inputRegister | نوع متغیر | ادرس |
| 32-bit  unsigned int-32  Int-32  float | U1 | 0 | 16-bit  unsigned int-16  int-16 | R1 |
| 1 | 16-bit  unsigned int-16  int-16 | R2 |
| 32-bit  unsigned int-32  Int-32  float | U2 | 2 | 16-bit  unsigned int-16  int-16 | R3 |
| 3 | 16-bit  unsigned int-16  int-16 | R4 |
|  | … |  |  | … |
| 32-bit  unsigned int-32  Int-32  float | Un | 2n-2 | 16-bit  unsigned int-16  int-16 | R2n-1 |
| 2n-1 | 16-bit  unsigned int-16  int-16 | R2n |

رجیسترهای تعریف شده پروتکل مدباس به صورت مجازی در میکروکنترلر با اختصاص فضای رم داخلی به وجود امده است. تعداد این رجیستر ها محدود است و در حالت پیش فرض تعداد Holding Register ها و Input Register ها عدد 50 است. این رجیسترها به صورت پیش فرض 16 بیتی هستند و می توان در انها متغیر int16\_t و uint16\_t ذخیره کرد. اما در صورت نیاز به متغیر های 32 بیتی مانند uint32\_t,int32\_t, float می توان از هر دو رجیستر یک متغیر 32 بیتی را ذخیره کرد در اینصورت ادرس دهی به صورت زیر تغییر میکند.

با استفاده از توابع زیر می توان با توجه به ادرس متغیرهای مد باس را با توجه به ادرس مقدار دهی کرد

/\* functions prototypes \*/

uint8\_t **GetCoilValue**(uint16\_t adr);

**void** **SetCoilValue**(uint16\_t adr, uint8\_t value);

//Get Holding Registers

uint16\_t **GetHoldingRegisterValue\_u16**(uint16\_t adr);

int16\_t **GetHoldingRegisterValue\_s16**(uint16\_t adr);

uint32\_t **GetHoldingRegisterValue\_u32**(uint16\_t adr);

int32\_t **GetHoldingRegisterValue\_s32**(uint16\_t adr);

**float** **GetHoldingRegisterValue\_f32**(uint16\_t adr);

//Set Holding Registers

**void** **SetHoldingRegisterValue\_u16**(uint16\_t adr, uint16\_t value);

**void** **SetHoldingRegisterValue\_s16**(uint16\_t adr, int16\_t value);

**void** **SetHoldingRegisterValue\_u32**(uint16\_t adr, uint32\_t value);

**void** **SetHoldingRegisterValue\_s32**(uint16\_t adr, int32\_t value);

**void** **SetHoldingRegisterValue\_f32**(uint16\_t adr, **float** value);

//Get Input Registers

uint16\_t **GetInputRegisterValue\_u16**(uint16\_t adr);

int16\_t **GetInputRegisterValue\_s16**(uint16\_t adr);

uint32\_t **GetInputRegisterValue\_u32**(uint16\_t adr);

int32\_t **GetInputRegisterValue\_s32**(uint16\_t adr);

**float** **GetInputRegisterValue\_f32**(uint16\_t adr);

//Set Input Registers

**void** **SetInputRegisterValue\_u16**(uint16\_t adr, uint16\_t value);

**void** **SetInputRegisterValue\_s16**(uint16\_t adr, int16\_t value);

**void** **SetInputRegisterValue\_u32**(uint16\_t adr, uint32\_t value);

**void** **SetInputRegisterValue\_s32**(uint16\_t adr, int32\_t value);

**void** **SetInputRegisterValue\_f32**(uint16\_t adr, **float** value);

هشدار

باید هنگام تعیین متغیرهای داخلی مد باس دقت شود که هیچ دو متغیر مستقل دارای ادرس یکسان درون برنامه نباشند در غیر اینصورت با تغییر یک متغیر، متغیر دیگر که از همان حافظه استفاده میکند نیز دچار تغییر میشود که باعث رخداد حالات ناخواسته میگردد. برنامه نویس مسئول تعیین متغیرهای درون برنامه است.