Autor: Diego Magno Montans Zucheram 12544329
Laboratório de Veículos Não Tripulados
Escola Politécnica - Universidade de São Paulo

Documentação Driver MTI 630

1 FUNDAMENTAÇÃO TEÓRICA

Figura 1 - Descrição dos pinos MTI - 630

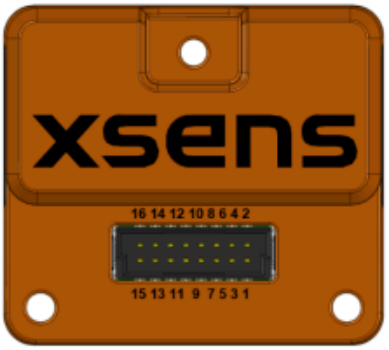


Figure 2: Pin configuration of the MTi 600-series module (bottom view)

Table 12: Pin descriptions of the MTi-600

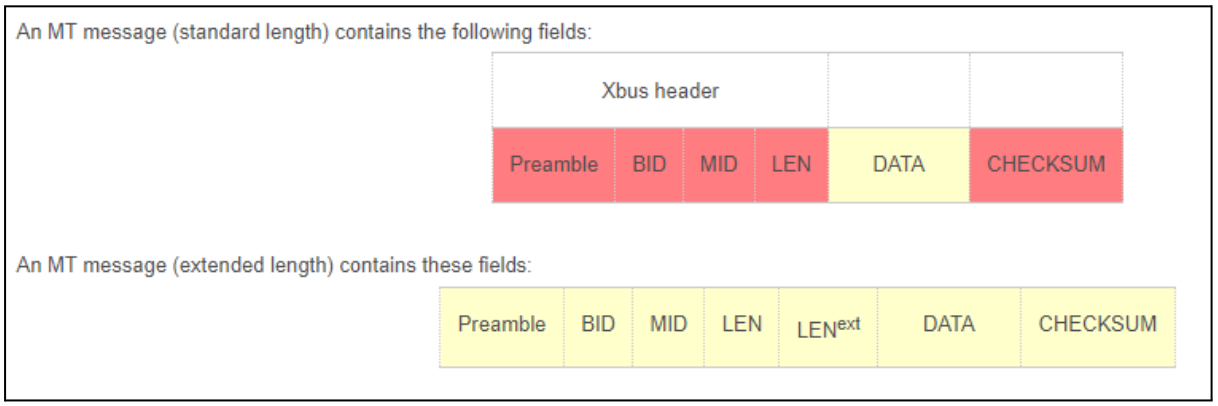
Pin	Name	I/O type	Description
1	VIN	PWR	Power input
2	GND	PWR	Ground
3	CAN_H	I/O	CAN bus differential high side
4	CAN_L	I/O	CAN bus differential low side
5	RS232_TxD	O	RS232 transmitter output to host
6	RS232_RTS	O	RS232 Ready To Send output to host
7	RS232_RxD	I	RS232 receiver input from host
8	RS232_CTS	I	RS232 Clear To Send input from host
9	SYNC_IN1	I	Multifunctional synchronization input
10	SYNC_IN2	I	Multifunctional synchronization input
11	GNSS_TxD[9]	O	RS232 transmitter output to GNSS module
12	GNSS_RxD[9]	I	RS232 receiver input from GNSS module
13	SYNC_OUT	O	Configurable synchronization output
14	GND	PWR	Ground
15	UART_TxD	O	UART transmitter output
16	UART_RxD	I	UART receiver input

Fonte: MTI - 600 Datasheet. Available at: https://www.xsens.com/hubfs/Downloads/Leaflets/MTi%20600-series%20Datasheet.pdf. Accessed on 07/10/24.

Como pode ser observado na Figura 1, o MTI 630 possui diversos tipos de comunicação entre o sensor e o usuário.

A fim de utilizar este sensor, precisamos entender como os dados do sensor são enviados para o usuário. Primeiramente, nota-se que uma mensagem na comunicação entre usuário e sensor é composta como mostram as Figuras 2 e 3. Ou seja, a mensagem é enviada em hexadecimal, onde cada par corresponde a um byte.

Figura 2 - Mensagem MT



Fonte: MTI - 600 Documentation-messages. Available at: https://mtidocs.movella.com/messages . Accessed on 07/10/24.

Figura 3 - Mensagem MT em bytes

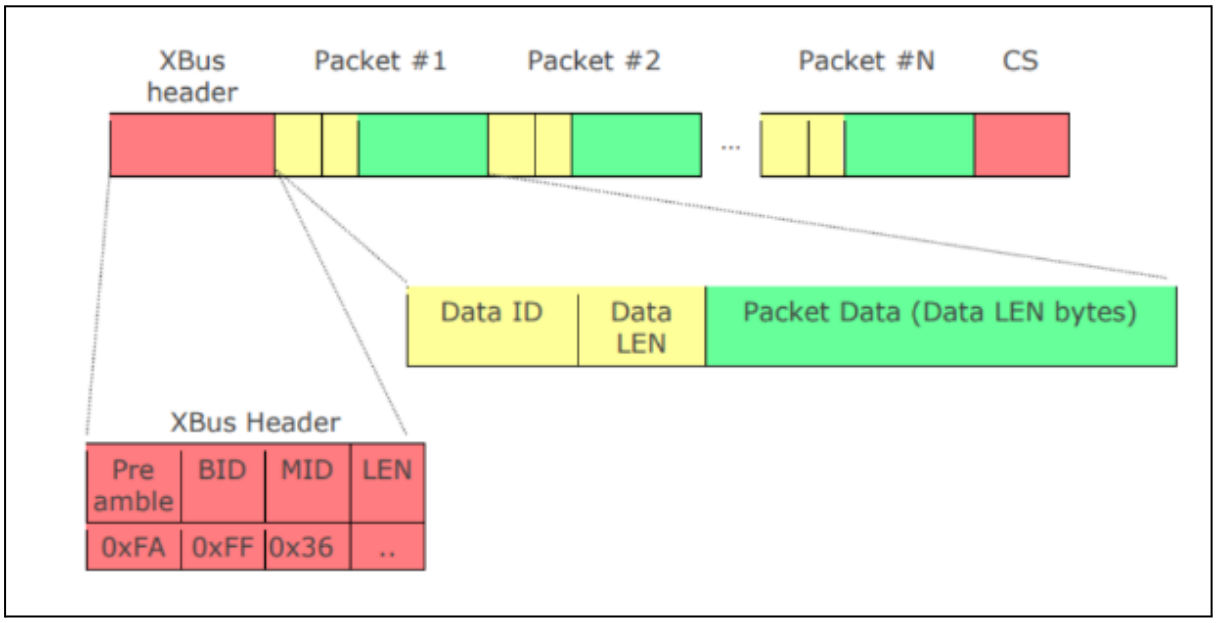
Construction of an Xbus message

Field	Field width	Description
Preamble	1 byte	Indicator of start of packet ⇒ 250 (0xFA)
BID	1 byte	Bus identifier or Address ⇒ 255 (0xFF)
MID	1 byte	Message identifier
LEN	1 byte	For standard length message: Value equals number of bytes in DATA field. Maximum value is 254 (0xFE) For extended length message: Field value is always 255 (0xFF)
EXT LEN	2 bytes	A 16 bit value representing the number of data bytes for extended length messages. Maximum value is 2048 (0x0800)
IND ID	1 byte	The type of indication received. Not used by MTi series.
DATA (standard length)	0 - 254 bytes	Data bytes (optional)
DATA (extended length)	255 - 2048 bytes	Data bytes
Checksum	1 byte	Checksum of the message

Fonte: MTI - 600 Documentation-messages. Available at: https://mtidocs.movella.com/messages . Accessed on 07/10/24.

No que condiz ao envio dos dados medidos pelo sensor ao usuário, por padrão, é uma mensagem do tipo MTDATA2, que pode ser observada na Figura 4. Nesse tipo de mensagem, caso não haja ext len, os 4 primeiros bytes correspondem ao Xbus Header, o último ao checksum, e os dados são divididos em pacotes que contém o ID do dado, seu tamanho e o dado em si.

Figura 3 - MTDATA2



Fonte: MTI - 600 Documentation-messages. Available at: https://mtidocs.movella.com/messages . Accessed on 07/10/24.

Assim, o objetivo deste driver é pegar o MTDATA2, que é uma mensagem “crua”, e guardar os dados de cada pacote recebido em um vetor de dados.

2 METODOLOGIA

2.1 DataDriver

- DataDriver é uma estrutura feita para armazenar os pacotes de dados enviados pelo sensor. O idName é uma implementação apenas para facilitar o entendimento dos dados por parte do usuário. A estrutura possui tanto armazenamento de uint quanto de floats devido às diferentes medições possíveis.

```
// Estrutura para armazenar informações de ID
typedef struct {
    uint16_t id;        // ID do dado (por exemplo, 0x1060)
    uint8_t length;     // Número de bytes de dados armazenados em
formato byte ou número de dados armazenados em formato float
    uint32_t data[256]; // Buffer para armazenar os dados em formato
bytes
```

```
    float dataf[64];     // Buffer para armazenar os dados em formato
float (máx 64 floats)
    char idName[50];    // Nome do ID (descrição)
} DataDriver;
```

2.2 knownIDs

- Todos os IDs conhecidos podem ser vistos na tabela “Available MTData2 packets” da Referência 1. Além disso, o tipo de dado emitido por cada sensor e suas explicações podem ser vistas na Referência 4, onde é definido MTDATA2 dentro da função ReqData.
- Observe que neste caso o y utilizado foi 0, ou seja, os dados estão na forma Float32 e ENU. Caso utilize um y diferente, é preciso implementar uma função para converter qual seja o formado para real, mudar o tipo de dado armazenado pelo DataDriver é preciso mudar o id, de acordo com o y do DataDriver dos seguintes dados:
    - Temperature (081y)
    - Quaternion (201y)
    - Rotation Matrix (202y)
    - EulerAngles (203y)
    - DeltaV (401y)
    - Acceleration (402y)
    - FreeAcceleration (403y)
    - AccelerationHR (404y)
    - RateOfTurn (802y)
    - DeltaQ (803y)
    - RateOfTurnHR (804y)
    - MagneticField. (C02y)

Figura 4 - Explicação do y

Where:

'x' = The hex value of the Type bits

'y' = The hex value of the Format bits (see table below). The value is formed by doing a bitwise OR of the available fields. For example: Quaternion orientation output (201y) expressed in the NED coordinate system with fixed point 16.32 numbers:

- Fp16.32 = 0x2
- NED = 0x4
- Fp16.32 and NED = 0x6

The resulting hex value for the identifier will be 0x2016

Settings for MTData2 Data Identifier

Field	Format	Description	Short name
Precision			
	0x0	Single precision IEEE 32-bit floating point number	Float32
	0x1	Fixed point 12.20 32-bit number	Fp1220
	0x2	Fixed point 16.32 48-bit number	Fp1632
	0x3	Double precision IEEE 64-bit floating point number	Float64
Coordinate system[11]			
	0x0	East-North-Up coordinate system	ENU
	0x4	North-East-Down coordinate system	NED
	0x8	North-West-Up	NWU

Fonte: Movella Documentation - Message Structure. Available at: https://mtidocs.movella.com/messages$Available%20MTData2%20packets . Accessed on 7/10/24.

```
// Inicializa o array com os IDs conhecidos
DataDriver knownIDs[MAX_IDS] = {
    {0x0810, 0, {0}, {0}, "Temperature"},       // index: 0     dado:
Real (R) -> Float32 se y = 0
    {0x1010, 0, {0}, {0}, "UtcTime"},           // index: 1     dado:
U4, U2, U1, U1, U1, U1, U1
    {0x1020, 0, {0}, {0}, "PacketCounter"},     // index: 2     dado:
U2
    {0x1060, 0, {0}, {0}, "SampleTimeFine"},    // index: 3     dado:
U4
    {0x1070, 0, {0}, {0}, "SampleTimeCoarse"},  // index: 4     dado:
U4
    {0x2010, 0, {0}, {0}, "Quaternion"},        // index: 5     dado: R
R R R
```

```
    {0x2020, 0, {0}, {0}, "RotationMatrix"},     // index: 6     dado: R
R R R R R R R R
    {0x2030, 0, {0}, {0}, "EulerAngles"},        // index: 7     dado: R
R R
    {0x3010, 0, {0}, {0}, "BaroPressure"},       // index: 8     dado:
U4
    {0x4010, 0, {0}, {0}, "Delta V"},            // index: 9     dado: R
R R
    {0x4020, 0, {0}, {0}, "Acceleration"},       // index: 10    dado: R
R R
    {0x4030, 0, {0}, {0}, "Free Acceleration"}, // index: 11    dado: R
R R
    {0x4040, 0, {0}, {0}, "AccelerationHR"},     // index: 12    dado: R
R R
    {0x8020, 0, {0}, {0}, "Rate of Turn"},       // index: 13    dado: R
R R
    {0x8030, 0, {0}, {0}, "Delta Q"},            // index: 14    dado: R
R R R
    {0x8040, 0, {0}, {0}, "RateOfTurnHR"},       // index: 15    dado: R
R R
    {0xE010, 0, {0}, {0}, "StatusByte"},         // index: 16    dado:
U1
    {0xE020, 0, {0}, {0}, "StatusWord"},         // index: 17    dado:
U4
    {0xC020, 0, {0}, {0}, "MagneticField"},      // index: 18    dado: R
R R
    // Adicione mais IDs conforme necessário
};
```

2.3 hexCharToByte

- Essa função tem como entrada um caractere char da mensagem MTData2, que por sua vez representa um hexadecimal. Sabe-se que os chars da mensagem são enviados de dois em dois, então o que se quer é transformar esses chars em um 1 par hexadecimal, ou seja um byte.
- Dentro da main, este byte será armazenado em um array chamado message, de forma que esse array será igual à mensagem MTData2 mas com 1 byte em cada índice.
- Observações:
    - byte <<=4 faz 4 bits de byte deslocaram para a esquerda. Ex: byte = 00001100 vira 11000000, neste caso isto é utilizado para guardar os dois hexadecimais
    - a operação |= é um or bit a bit. Neste caso, byte recebe o byte anterior em or bit a bit com o resultado da operação. Ex: 0010 + 1000 = 1010

- Exemplo: ‘C2’, C - ‘A’ em ASCII é 2, + 10 temos 12, que é C em decimal e assim byte = 00001100, já que 12 é 1100 em binário.

```
// Função para converter um par de caracteres hexadecimais em um byte
uint8_t hexCharToByte(const char *hex) {
    uint8_t byte = 0;
    for (int i = 0; i < 2; ++i) {
        byte <<= 4;
        if (hex[i] >= '0' && hex[i] <= '9') {
            byte |= hex[i] - '0';
        } else if (hex[i] >= 'A' && hex[i] <= 'F') {
            byte |= hex[i] - 'A' + 10;
        } else if (hex[i] >= 'a' && hex[i] <= 'f') {
            byte |= hex[i] - 'a' + 10;
        }
    }
    return byte;
}
```

2.4 findIDIndex

- Essa função é utilizada para comparar o id da message com os ids armazenados, para saber qual pacote de dado está sendo tratado naquele momento.
- Esta função é utilizada dentro da main para armazenar os pacotes de dados recebidos na mensagem em seus respectivos vetores DataDriver.

```
// Função para encontrar um ID na lista de IDs conhecidos
int findIDIndex(uint16_t id, DataDriver *entries, int numEntries) {
    for (int i = 0; i < numEntries; ++i) {
        if (entries[i].id == id) {
            return i;
        }
    }
    return -1;
}
```

2.5 uint32ToFloat

- Essa função pega 4 bytes consecutivos da mensagem e guarda como float. Ela é utilizada na main quando os index referenciam para sensores que tem como saída float.

```
// Função para converter 4 bytes para float
```

```
float uint32ToFloat(const uint32_t bytes) {
    float value;
     memcpy(&value, &bytes, sizeof(float));   // Copia 4 bytes para o
valor float
    return value;
}
```

2.6 GuardaMsg

- É a função principal do driver, que guarda a mensagem MTData2 de uma forma a conseguir utilizar os dados recebidos pelo sensor de uma forma fácil.
- Ela tem não tem saída (void), e recebe dois argumentos, sendo o primeiro um array do tipo DataDriver onde você deseja que os dados serem armazenados, e o segundo a mensagem MTData2 na forma de string.

```
// Função para guardar a mensagem que vem em string nos respectivos
vetores
void GuardaMsg(DataDriver *Dest, char hexMessage[]) {
```

- Primeiramente, inicia-se o vetor destino igual ao knownIDs para ter a mesma compatibilidade de IDs e nomes.

```
    // Copia os IDs conhecidos para Dest
    memcpy(Dest, knownIDs, sizeof(knownIDs));

    int hexMessageLength = strlen(hexMessage);
```

- Como citado em 2.3, a mensagem MTData2, tida como uma string formada por hexadecimais, é convertida em um array de bytes, ignorando espaços. Este código basicamente aplica a função hexCharToByte() enquanto tira os espaços em branco.

```
// Converta a string hexagonal para um array de bytes, ignorando
espaços
    uint8_t message[hexMessageLength / 2];
    int messageLength = 0;

    for (int i = 0; i < hexMessageLength; i += 2) {
        while (hexMessage[i] == ' ') i++; // Ignorar espaços em branco
         if (hexMessage[i] == '\0' || hexMessage[i + 1] == '\0') break;
// Verificar fim da string
        message[messageLength++] = hexCharToByte(&hexMessage[i]);
    }
```

● Aqui pega-se 2 bytes a cada iteração, encontra-se o índice do id, pega-se o tamanho do dado, e a partir dessas informações, atualiza a variável de destino dos dados criada.
● Observações:
    ○ pos = BYTES_IRRELEVANTES = 4, para ignorar os 4 primeiros bytes
    ○ pos < messageLength-1 para que ele ignore o CS (CountSum) e o CS não pareça como id desconhecido
    ○ message[pos] << 8, para que o byte que estava guardado no índice pos da message, seja guardado nos 8 bits mais significativos do id, já que id tem 16 bits. Ex: message[pos] = 0x10 = 00010000 e message[pos+1] = 0x60 = 01100000. Assim, id = 00010000 01100000 = 0x1060

```
    int numKnownIDs = MAX_IDS;  // Número de IDs conhecidos
    int pos = BYTES_IRRELEVANTES; // Começar após os primeiros bytes
irrelevantes

    while (pos < messageLength - 1) {
        // Pega o ID (2 bytes)
        uint16_t id = (message[pos] << 8) | message[pos + 1];
        pos += 2;

        // Encontra o índice do ID
        int index = findIDIndex(id, knownIDs, numKnownIDs);
        if (index == -1) {
            printf("ID desconhecido: 0x%04X\n", id);
            break; // Se encontrar um ID desconhecido, interrompe
        }

        // Pega o comprimento dos dados associado ao ID
        uint8_t length = message[pos];
        pos++;
```

- Armazenamento de dados conforme o index do sensor na lista do DataDriver. Isto para diferenciar a saída de cada um, já que alguns tem saída real, outros u1, u4, etc.

```
// Verifica o tipo de dado e processa de acordo
        switch (index) {
            case 0: case 5: case 6: case 7: case 9: case 10: case 11:
            case 12: case 13: case 14: case 15: case 18: {
                for (int i = 0; i < length / 4; ++i) {
                    Dest[index].dataf[i] = uint32ToFloat(((message[pos]
<< 24) | (message[pos + 1] << 16) | (message[pos+2] << 8) | message[pos
+ 3]));
```

```
                    pos += 4;
                }
                Dest[index].length = length / 4; // O comprimento é o
número de floats
                break;
            }

            case 3: case 4: case 8: case 17: {
                Dest[index].data[0] = ((message[pos] << 24) |
(message[pos + 1] << 16) | (message[pos+2] << 8) | message[pos + 3]);
                pos += 4;
                Dest[index].length = length / 4; // O comprimento é o
número de inteiros
                break;
            }

            case 2: {
                Dest[index].data[0] = ((message[pos] << 8) |
message[pos + 1]);
                pos += 2;
                Dest[index].length = length / 2; // O comprimento é o
número de inteiros
                break;
            }

            case 1: {
                Dest[index].data[0] = ((message[pos] << 24) |
(message[pos + 1] << 16) | (message[pos+2] << 8) | message[pos + 3]);
                pos += 4;
                Dest[index].data[1] = ((message[pos] << 8) |
message[pos + 1]);
                pos += 2;
                Dest[index].data[2] = message[pos++];
                Dest[index].data[3] = message[pos++];
                Dest[index].data[4] = message[pos++];
                Dest[index].data[5] = message[pos++];
                Dest[index].data[6] = message[pos++];
                Dest[index].length = 7; // Comprimento é 7 para esse
caso
                break;
            }

            case 16: {
```

```
                Dest[index].data[0] = message[pos];
                Dest[index].length = length;
                pos += length;
                break;
            }
        }
    }
}
```

3 DRIVER COMPLETO

Deixo aqui o driver completo para facilitar a cópia para determinada aplicação. Link do repositório no GitHub: https://github.com/DmmZ13/Driver_MTI630.git . Note que para utilizar o driver disponibilizado, basta colocar os arquivos driver_mti630.c e driver_mti630.h no seu projeto, e implementar na main como pode ser visto no código abaixo, trocando hexMessage[] pela mensagem que você está recebendo do sensor via serial.

```
        // Mensagem de exemplo em formato de string com espaços
         char hexMessage[] = "FA FF 36 4E 10 60 04 02 53 35 CB 20 10
10 3F 7F FA 66 BB 40 34 B8 3C 49 96 2D 3B 5A 1F 82 80 20 0C 39 FF 91 01
BA EC DD 81 BA 86 53 80 40 20 0C BE 64 DB 99 BC E7 FF 83 41 1D C2 CF C0
20 0C BE 90 48 80 3F 67 32 04 BF 54 89 04 E0 20 04 00 00 00 03 C9";

        // Cria uma estrutura destino que será passada para a função
        DataDriver destino[MAX_IDS];

        // Chama a função para armazenar os dados da mensagem
        GuardaMsg(destino, hexMessage);
```

Figura 5 - Exibição dos dados armazenados utilizando o arquivo driver_last.c

```
ID 0x0810 Temperature:
ID 0x1010 UtcTime:
ID 0x1020 PacketCounter:
ID 0x1060 SampleTimeFine: 39007691
ID 0x1070 SampleTimeCoarse:
ID 0x2010 Quaternion: 0.999915 -0.002933 0.012304 0.003328
ID 0x2020 RotationMatrix:
ID 0x2030 EulerAngles:
ID 0x3010 BaroPressure:
ID 0x4010 Delta V:
ID 0x4020 Acceleration: -0.223494 -0.028320 9.860061
ID 0x4030 Free Acceleration:
ID 0x4040 AccelerationHR:
ID 0x8020 Rate of Turn: 0.000487 -0.001807 -0.001025
ID 0x8030 Delta Q:
ID 0x8040 RateOfTurnHR:
ID 0xE010 StatusByte:
ID 0xE020 StatusWord: 3
ID 0xC020 MagneticField: -0.281803 0.903107 -0.830216
```

4 IMPLEMENTAÇÃO

Para utilizar o sensor, primeiramente deve-se colocar o MTI630 no modo de configuração para setar a frequência de medição de seus sensores. Depois, voltar para o modo de medição e receber os dados. Um exemplo desse procedimento pode ser visto na Referência 3. Ainda, nota-se que essa configuração já vem padronizada e não é necessário realizá-la todas as vezes que for utilizar o sensor. Ao ligá-lo ele já estará no modo de recepção de dados.

REFERÊNCIAS

1. Movella Documentation - Message Structure. Available at: https://mtidocs.movella.com/messages$Available%20MTData2%20packets . Accessed on 7/10/24.
2. Movella Documentation - MT Low Level Communication Protocol Example - Article.Available at: https://base.movella.com/s/article/How-to-use-Device-Data-View-to-learn-MT-Low-Level-Communications?language=en_US . Accessed on 07/10/24.
3. Movella Documentation - MT Low Level Communication Protocol Example. Available at: https://mtidocs.movella.com/mt-low-level-communication-protocol-example. Accessed on 07/10/24.
4. Movella Documentation - Message type for each sensor. Available at: https://mtidocs.movella.com/messages . Accessed on 9/10/24.