UNSW SYDNEY

Binary Data

A/Prof Shiyang Tang

MTRN3500

Computing Applications in Mechatronics Systems

Why Binary Data?

Galil data

"MG @AN[2]\r"

ASCII encoding

```
Encoding::ASCII->GetBytes(MG @AN[2]\r);
```

0x4D 0x47 0x20 0x40 0x41 0x4E 0x5B 0x32 0x5D 0x0D

LiDAR scan data

```
sRA LMDscandata 0 1 9BF210 0 0 25E6 25EA 5B855C7F 5B85621E 0 0 7 0 0 1388
168 0 1 DIST1 3F800000 00000000 0 1388 169 152A 14A0 146E 1529 1573 15FE
17C8 17DF 1773 174B 174A 1746 1764 16BA A79 A7F A6C 933 822 833 8DB
92C 88B 855 8A9 907 8C3 88F 849 81B 7E5 740 714 70A 710 742 7BE 7BF 7CD
7FA 80A 810 831 879 8D7 8A0 837 85B 97D BD2 BAC B5D B32 B08 9AF 6A6 65D
616 5EE 5D0 5C7 5D6 5EA 5F1 5F5 5F9 604 60A 623 629 625 63D 635 63A 65D
71A 777 7E4 8A5 C5C D08 D26 D16 CD4 B26 A89 A42 A2A A1D A10 A02 9E4 9EF
9DD 9DB 9DC 9D0 9D7 9F2 A00 A11 A26 A41 A8D A4D A06 A0E C0B 106D 10A1 1176
1153 11F0 1222 1269 1280 12A4 131D 138F 1380 133C 133F 1318 EB2 D1A EAF
ED9 F3B 10C5 143A 1495 1760 178F 115A 1098 107B 109F 10CF 12CC 13F5 136D
1333 1362 13DC 1630 1686 1699 174F 1824 1815 1816 180D 1800 17E1 17EE 37E
375 363 362 328 321 312 31F 314 30B 304 309 309 30A 30B 328 31E 303 307
305 304 309 306 304 313 313 312 327 330 311 317 318 314 31B 312 312
31E 311 31D 31F 33E 369 3AD 3DE 43C 652 650 65F 669 65B 65E 661 66F 674
67A 678 66D 67A 683 68A 68A 68A 68A 68A 68A 68A 68A 68A 68A 68A 68A
5DE 5DE 5D9 5C8 5DF 5E5 602 607 624 64C 66B 6B8 52A 485 432 405 3ED 3F7
3E8 3D7 3D0 3C9 3D0 3CE 3B8 3BA 3A8 3AB 3A7 3A4 3A2 3A4 391 3AD 36A 196
0 11D 10A 111 10A 108 104 F0 ED DD D9 C9 D6 D2 CD CC C5 AC AD BC B9 B6 AC
B5 AD A6 AC A7 A6 B7 B0 AC AB AD B4 AF B2 B4 AB AF B5 A2 A0 A9 A3 A2 A4 94
A0 A9 A5 9B A7 94 A9 AD A7 9B 9F A7 B0 A2 AF B9 B5 A8 B1 B2 C5 C6 C6 CB CC
D0 C9 C1 C6 C2 B7 B4 9C 9A A1 AC 93 93 A3 A5 B3 AA 0 0 0 0 0 0
```

GNSS data

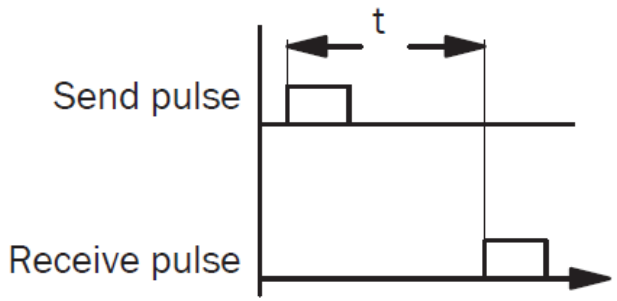
```
0xaa, 0x44, 0x12, 0x1c, 0xd6, 0x02, 0x02, 0x50, 0x00, 0x64, 0xb4, 0x94, 0x05,
0xf6, 0xc4, 0x39, 0x0b, 0x00, 0x00, 0x8c, 0xef, 0x81, 0x08, 0x00, 0x00, 0x00,
0x32, 0x00, 0x00, 0x38, 0x00, 0xcd, 0x5a, 0x13, 0x41, 0x00, 0x07, 0xb1, 0x8a,
0xbd, 0x57, 0x41, 0xfd, 0xbb, 0x41, 0x3d, 0x00, 0x00, 0x07, 0x0b, 0x32, 0x3f,
0xe8, 0x18, 0x5b, 0x40, 0x81, 0x7c, 0xa5, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xf4, 0x54, 0x03, 0x3d, 0x48, 0x41, 0x41, 0x00, 0x07, 0x0b, 0x32, 0x3f, 0x00,
0xaa, 0x44, 0x1c, 0xd6, 0x02, 0x20, 0x50, 0x00, 0x64, 0xb4, 0x94, 0x05, 0x00,
0xf6, 0xc4, 0x39, 0x0b, 0x00, 0x00, 0x8c, 0xef, 0x81, 0x08, 0x00, 0x00, 0x00,
0x32, 0x00, 0x00, 0x38, 0x00, 0xcd, 0x5a, 0x13, 0x41, 0x00, 0x07, 0xb1, 0x8a,
0xbd, 0x57, 0x44, 0xc3, 0x35, 0x3d, 0x00, 0x00, 0x07, 0x0b, 0x32, 0x3f, 0x00,
0x55, 0x47, 0xac, 0xe5, 0x3d, 0x41, 0x00, 0x00, 0x07, 0x0b, 0x32, 0x3f, 0x00,
0x5a, 0x44, 0x39, 0x0b, 0x00, 0x00, 0x8c, 0xef, 0x81, 0x08, 0x00, 0x00, 0x00,
0x32, 0x00, 0x00, 0x38, 0x00, 0xcd, 0x5a, 0x13, 0x41, 0x00, 0x07, 0xb1, 0x8a,
0xbd, 0x57, 0x40, 0x81, 0x7c, 0xa5, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xed, 0x54, 0x03, 0x3d, 0x48, 0x41, 0x41, 0x00, 0x07, 0x0b, 0x32, 0x3f, 0x00,
0xaa, 0x44, 0x12, 0x1c, 0xd6, 0x02, 0x02, 0x50, 0x00, 0x64, 0xb4, 0x94, 0x05,
0xf6, 0xc4, 0x39, 0x0b, 0x00, 0x00, 0x8c, 0xef, 0x81, 0x08, 0x00, 0x00, 0x00,
0x32, 0x00, 0x00, 0x38, 0x00, 0xcd, 0x5a, 0x13, 0x41, 0x00, 0x07, 0xb1, 0x8a,
0xbd, 0x57, 0x40, 0x81, 0x7c, 0xa5, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xed, 0x54, 0x03, 0x3d, 0x48, 0x41, 0x41, 0x00, 0x07, 0x0b, 0x32, 0x3f, 0x00,
```

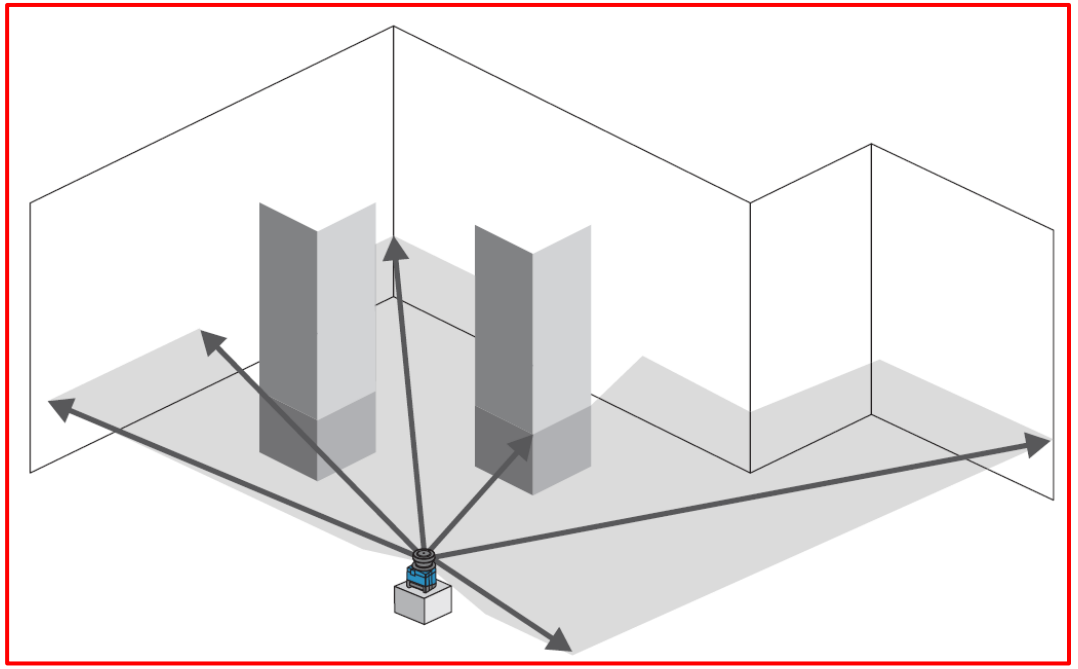
ASCII Data

- **ASCII:** American Standard Code for Information Interchange
- ASCII is a standard way for computers to represent text (characters) as numbers (binary data).
- Every letter, digit, symbol, or control key (like Enter or Tab) is assigned a unique number between 0 and 127, and this number is stored in binary (8 bits).

Character	ASCII decimal	Binary (8-bit)	Hex
A	65	0100 0001	0x41
B	66	0100 0010	0x42
a	97	0110 0001	0x61
0	48	0011 0000	0x30
9	57	0011 1001	0x39
Space	32	0010 0000	0x20
@	64	0100 0000	0x40
\r	13	0000 1101	0x0D
Newline (LF)	10	0000 1010	0x0A

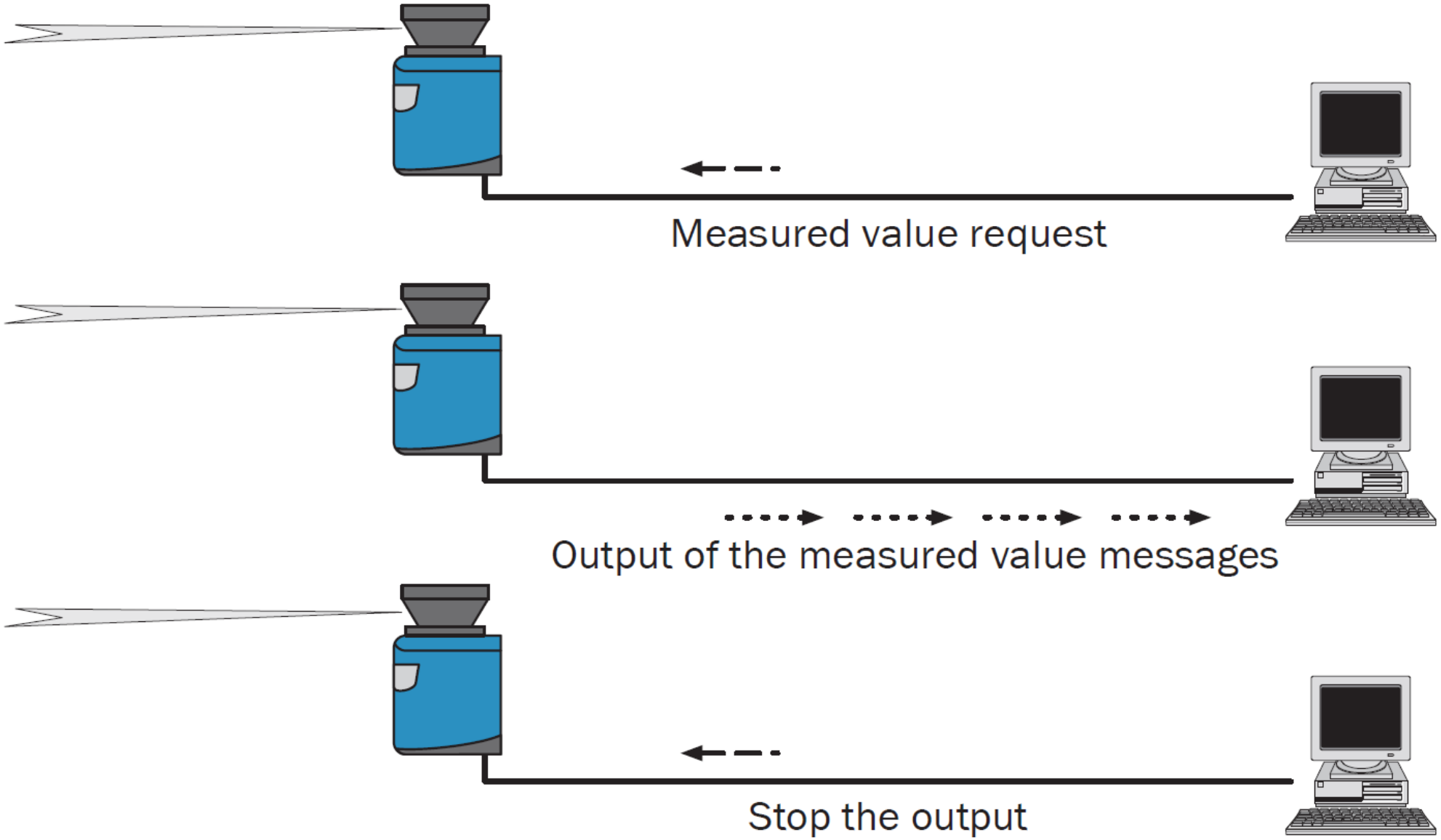
SICK LMS151 LiDAR



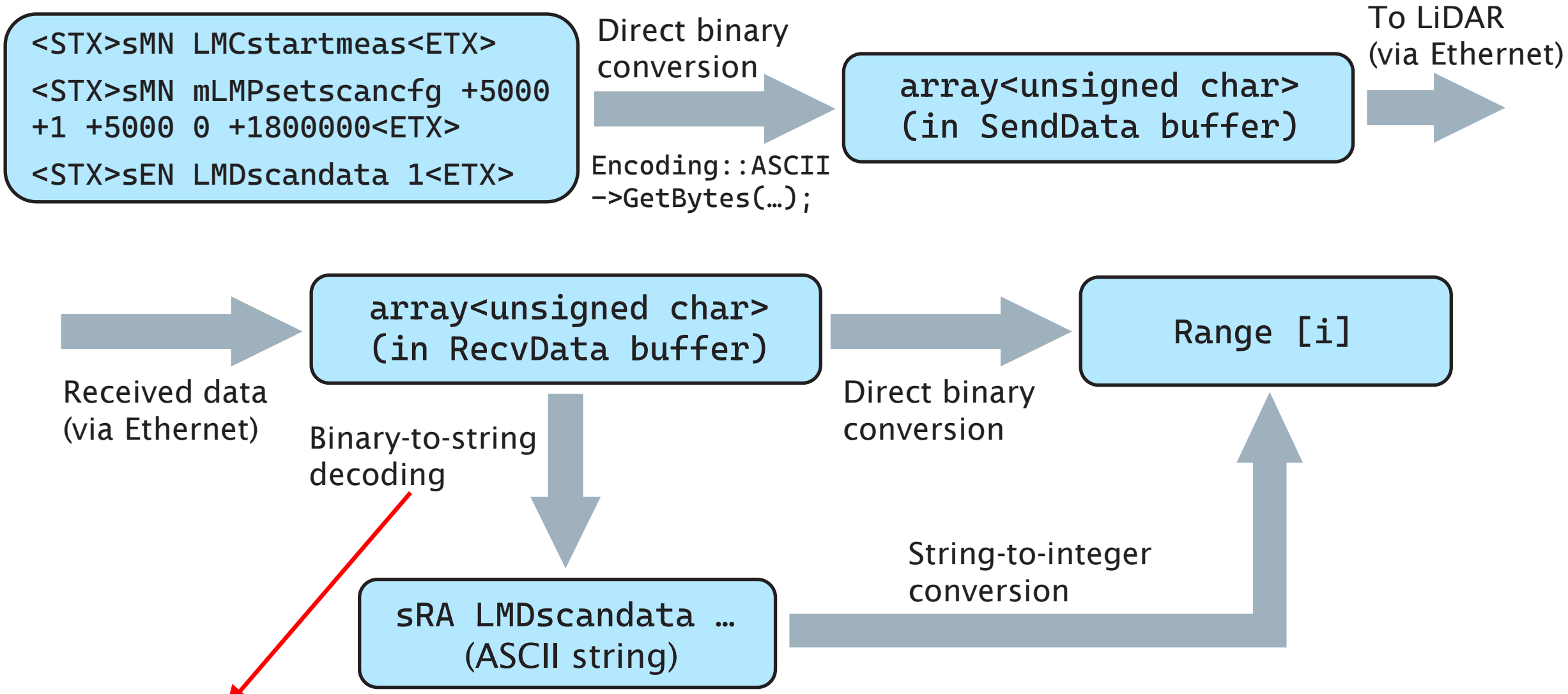


- field of view maximum 270˚
- resolution of the angular step width: 0.25/0.50˚
- rotation frequency 25/50 Hz
- configuration/measured value request using messages (command strings)
- scanning range up to 50 m with > 75% object remission (18 m with 10% object remission)

Acquiring LiDAR Data

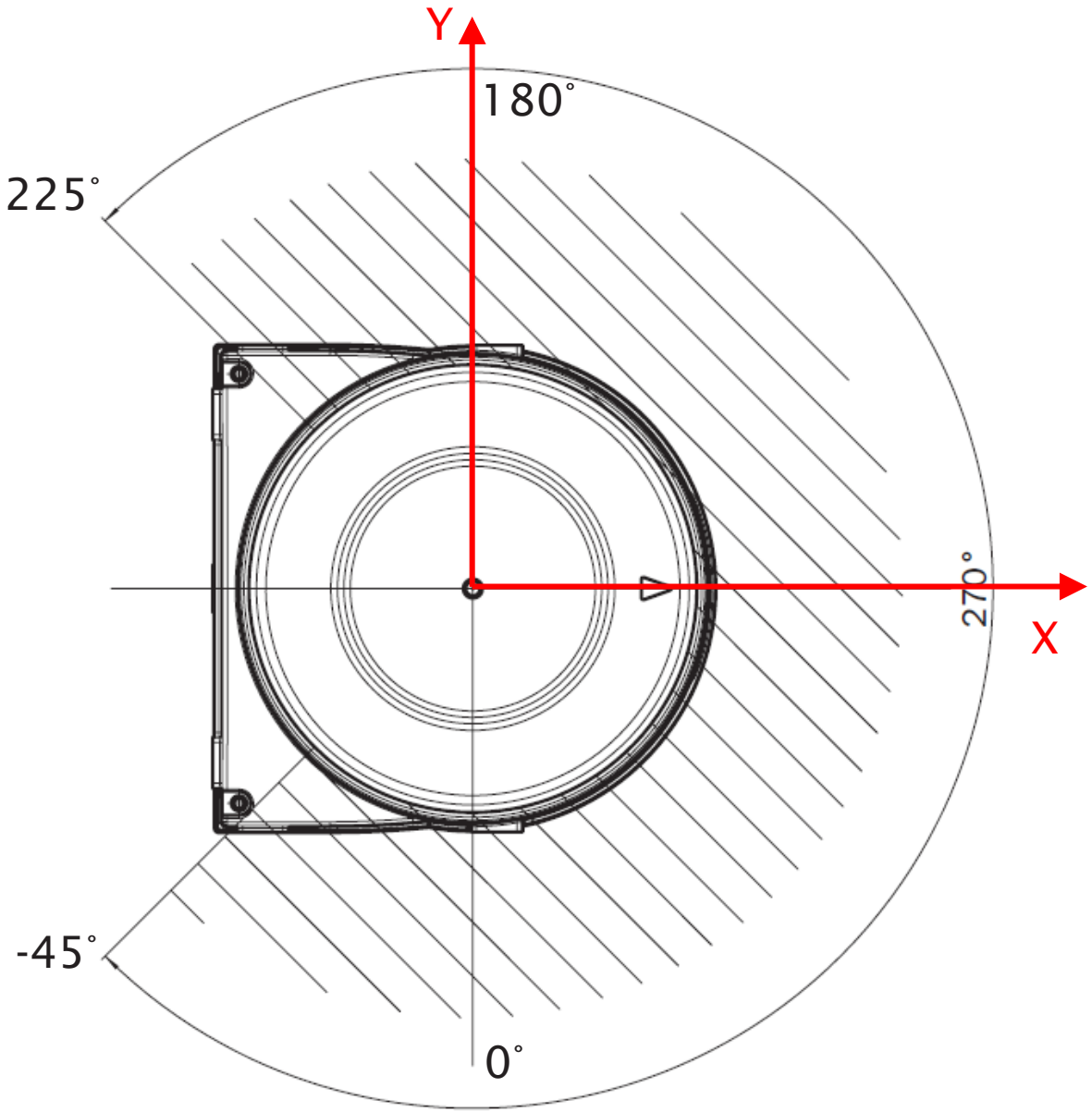


LiDAR Data Communication via TCP



`Encoding::ASCII->GetString(RecvData)`
It takes a **byte array** and converts it into a **managed .NET string**.

LiDAR Coordinates




```
<STX>sMN mLMPsetscancfg +5000 +1 +5000
0 +1800000<ETX>
```

Scanning frequency: 50 Hz
Segment: 1 (no multiple segments for LMS151)
Angle resolution: 0.5°
Start angle: 0°
Stop angle: +180°

LiDAR Data Content

```
sRA/sSN LMDscandata VersionNumber DeviceNumber SerialNumber DeviceStatus
MessageCounter ScanCounter PowerUpDuration TransmissionDuration InputStatus
OutputStatus ReservedByteA ScanningFrequency MeasurementFrequency
NumberEncoders[EncoderPosition EncoderSpeed]

NumberChannels16Bit[MeasuredDataContent ScalingFactor ScalingOffset StartingAngle
AngularStepWidth NumberData [Data_1 Data_n]]

NumberChannels8Bit[MeasuredDataContent ScalingFactor ScalingOffset StartingAngle
AngularStepWidth NumberData [Data_1 Data_n]]

Position[XPosition Yposition ZPosition XRotation YRotation ZRotation RotationType]

Name[DeviceName]

Comment[CommentContent]

TimeInfo[Year Month Day Hour Minute Second Microseconds]

EventInfo[EventType EncoderPosition EventTime AngularPosition]
```

LiDAR Measurement Data Content

```
NumberChannels16Bit[MeasuredDataContent ScalingFactor ScalingOffset StartingAngle
AngularStepWidth NumberData [Data_1 Data_n]]
```

NumberChannels16Bit		Defines the number of 16-bit output channels on which measured data are output. If "0 output channels" are selected, no data are output.	uint_16	2	**0 ... 4** 0 to 4 output channels
Output channel 1 ... 4 (16 bit)	MeasuredDataContent	The message part defines the contents of the output channel.	string	5	**DIST1** Radial distance for the first reflected pulse **RSSI1** Remission values for the first reflected pulse **DIST2** Radial distance for the second reflected pulse **RSSI2** Remission values for the second reflected pulse
	ScalingFactor	Multiplier for the values in the message parts Data_1 to Data_n	Real	4	**00000000h ... FFFFFFFFh**
	ScalingOffset	For the LMS always 0	Real	4	**00000000h ... FFFFFFFFh**
	Starting angle	Information 1/10,000 degree	int_32	4	**-550,000 ... +1,250,000**
	Angular step width	Information 1/10,000 degree	uint_16	2	**1,000 ... 10,000**
	NumberData	Defines the number of items of measured data output	uint_16	2	**0 ... 1,082**
	Data_1 Data_n	Output of the measured values 1 to n. The contents and the unit depend on the message part "MeasuredDataContent". **DIST** in mm, **RSSI** in digits	uint_16 uint_16	2 2	**0000h ... FFFFh** **0000h ... FFFFh**

LiDAR Measurement Data String

Starting angle

Angular step width
(0x1388 = 5000_{10})

NumberData
(0x169 = 361_{10})

```
sRA LMDscandata 0 1 9BF210 0 0 25E6 25EA 5B855C7F 5B85621E 0 0 7 0 0 1388 168 0 1 DIST1
3F800000 00000000 0 1388 169 152A 14A0 146E 1529 1573 15FE 17C8 17DF 1773 174B 174A 1746 1764
16BA A79 A7F A6C 933 822 833 8DB 92C 88B 855 8A9 907 8C3 88F 849 81B 7E5 740 714 70A 710 742
7BE 7BF 7CD 7FA 80A 810 831 879 8D7 8A0 837 85B 97D BD2 BAC B5D B32 B08 9AF 6A6 65D 616 5EE
5D0 5C7 5D6 5EA 5F1 5F5 5F9 604 60A 623 629 625 63D 635 63A 65D 71A 777 7E4 8A5 C5C D08 D26
D16 CD4 B26 A89 A42 A2A A1D A10 A02 9E4 9EF 9DD 9DB 9DC 9D0 9D7 9F2 A00 A11 A26 A41 A8D A4D
A06 A0E C0B 106D 10A1 1176 1153 11F0 1222 1269 1280 12A4 131D 138F 1380 133C 133F 1318 EB2 D1A
EAF ED9 F3B 10C5 143A 1495 1760 178F 115A 1098 107B 109F 10CF 12CC 13F5 136D 1333 1362 13DC
1630 1686 1699 174F 1824 1815 1816 180D 1800 17E1 17EE 37E 375 363 362 328 321 312 31F 314 30B
304 309 309 30A 30B 328 31E 303 307 305 304 309 306 304 313 313 312 327 330 311 317 318 314
31B 312 312 31E 311 31D 31F 33E 369 3AD 3DE 43C 652 650 65F 669 65B 65E 661 66F 674 67A 678
66D 67A 683 68A 68A 68A 68A 68A 68A 68A 68A 68A 68A 68A 5DE 5DE 5D9 5C8 5DF 5E5 602 607
624 64C 66B 6B8 52A 485 432 405 3ED 3F7 3E8 3D7 3D0 3C9 3D0 3CE 3B8 3BA 3A8 3AB 3A7 3A4 3A2
3A4 391 3AD 36A 196 0 11D 10A 111 10A 108 104 F0 ED DD D9 C9 D6 D2 CD CC C5 AC AD BC B9 B6 AC
B5 AD A6 AC A7 A6 B7 B0 AC AB AD B4 AF B2 B4 AB AF B5 A2 A0 A9 A3 A2 A4 94 A0 A9 A5 9B A7 94
A9 AD A7 9B 9F A7 B0 A2 AF B9 B5 A8 B1 B2 C5 C6 C6 CB CC D0 C9 C1 C6 C2 B7 B4 9C 9A A1 AC 93
93 A3 A5 B3 AA 0 0 0 0 0 0
```

361 scan range data (in mm)

String to Data

Step 1:

Using the following code, we split **LMDscandata** into individual substrings separated by spaces.

```
array<wchar_t>^ Space = { ' ' };
array<String^>^ StringArray = LMDscandata->Split(Space);
```

This will make an **array of handles** to strings.

According to the data description in the LMS151 manual:

- `StartAngle` is `StringArray[23]`
- `Angular step width` (i.e. resolution) is `StringArray[24]`
- Number of range data points (`NumberData`) in `StringArray[25]` followed by
- That many Range data in mm

```
sRA LMDscandata 0 1 9BF210 0 0 25E6 25EA 5B855C7F 5B85621E 0 0 7 0 0 1388 168 0 1 DIST1
3F800000 00000000 0 1388 169 152A 14A0 146E 1529 1573 15FE 17C8 17DF 1773 174B 174A 1746 1764
```

String to Data

Step 2:

Getting StartAngle, Resolution, NumberData, and Range via string-integer conversion.

```
double StartAngle = System::Convert::ToInt32(StringArray[23], 16);
double Resolution = System::Convert::ToInt32(StringArray[24], 16) / 10000.0;
int NumberData = System::Convert::ToInt32(StringArray[25], 16);

array<double>^ Range = gcnew array<double>(NumberData);
array<double>^ RangeX = gcnew array<double>(NumberData);
array<double>^ RangeY = gcnew array<double>(NumberData);

for (int i = 0; i < NumberData; i++)
{
        Range[i] = System::Convert::ToInt32(StringArray[26 + i], 16);
        RangeX[i] = Range[i] * sin(i * Resolution);
        RangeY[i] = -Range[i] * cos(i * Resolution);
}
```

System::Convert::ToInt32("string", 16): converts a string into an Int32 (a 32-bit integer).

The second argument 16 indicates that the number should be interpreted as **hexadecimal**.

Data Memory Size

Category	Type	Typical Size (bytes)	Value Range
Character types	`char`	1	-128 to 127 *(signed)* or 0 to 255 *(unsigned)*
	`unsigned char`	1	0 to 255
	`signed char`	1	-128 to 127
Integer types	`short`	2	-32,768 to 32,767
	`unsigned short`	2	0 to 65,535
	`int`	4	-2,147,483,648 to 2,147,483,647
	`unsigned int`	4	0 to 4,294,967,295
	`long long`	8	$\pm 9.22 \times 10^{18}$
	`unsigned long long`	8	0 to 1.84×10^{19}
Floating-point types	`float`	4	$\sim \pm 3.4 \times 10^{38}$
	`double`	8	$\sim \pm 1.7 \times 10^{308}$
Boolean	`bool`	1	`true` or `false`
Wide characters	`wchar_t`	2 (Windows)	0 to 65,535 (UTF-16) or 0 to 4,294,967,295 (UTF-32)
Pointer (any type)	`T*`	4 (32-bit) / 8 (64-bit)	Memory address

IEEE-754 Double Precision

`double A = 12345.54321`

`double A` takes 8 bytes (64 bits) of memory space, and is stored internally as:

$$(-1)^{sign\ bit} \times (1.mantissa\ bits)_2 \times 2^{exponent\ bits - 1023}$$

Bit range	Name	Length	Purpose
63	**Sign bit**	1 bit	0 = positive, 1 = negative
62–52	**Exponent**	11 bits	Encodes the power of 2
51–0	**Mantissa (Fraction)**	52 bits	Encodes the significant digits (precision)

Conversely, you can take 8 bytes of data, combine them, and interpret the resulting 64-bit binary according to the above IEEE 754 standard, yielding a value represented as a managed double (important for reading GNSS data).

IEEE-754 Double Precision

Step 1: Sign bit

$$A > 0 \rightarrow \text{sign bit} = 0$$

Step 2: Normalise the number

Find e so that $A = (1.f) \times 2^e$ with $1 \leq 1.f < 2$.

- $2^{13} = 8192,\ 2^{14} = 16384 \rightarrow$ choose $e = 13$
- $A/2^{13} = 12345.54321/8192 = 1.507024317626953$

So

$$A = (1.507024317626953) \times 2^{13}$$

IEEE-754 double gives **16 digits of precision**. Keeping more digits will not affect the final 64-bit representation

Step 3: Exponent field (11 bits)

Bias = 1023

Unbiased exponent $e = 13 \rightarrow$ biased exponent $E = 13 + 1023 = 1036$.

1036 in binary (11 bits) = `100 0000 1100` = **exponent bits**

IEEE-754 Double Precision

Step 4: Fraction (mantissa) field (52 bits)

Fraction = $1.507024317626953 - 1 = 0.507024317626953$

Encode 52 fraction bits:

$$M = \text{round}\,(0.507024317626953 \times 2^{52}) = 2283434527932526 = \text{0x81CC587E7C06E}$$

Binary of 0x81CC587E7C06E (52 bits):

1000 0001 1100 1100 0101 1000 0111 1110 0111 1100 0000 0110 1110

Step 5: Assemble the 64 bits

```
[ sign ][     exponent     ][                    mantissa (52)                    ]
    0       10000001100      1000000111001100010110000111111001111100000001101110
```

As one 64-bit string:

0100 0000 1100 1000 0001 1100 1100 0101 1000 0111 1110 0111 1100 0000 0110 1110

Group into bytes (MSB→LSB):

01000000 11001000 00011100 11000101 10000111 11100111 11000000 01101110

0x40 0xC8 0x1C 0xC5 0x87 0xE7 0xC0 0x6E

Final bytes: 40 C8 1C C5 87 E7 C0 6E (big-endian) / 6E C0 E7 87 C5 1C C8 40 (little-endian).

GNSS Data Structure

1. Basic format of:

Header	3 Sync bytes plus 25 bytes of header information. The header length is variable as fields may be appended in the future. Always check the header length.
Data	variable
CRC	4 bytes

2. The 3 Sync bytes will always be:

Byte	Hex	Decimal
First	AA	170
Second	44	68
Third	12	18

3. The CRC is a 32-bit CRC (see *32-Bit CRC on Page 24* for the CRC algorithm) performed on all data including the header.

Cyclic Redundancy Check (CRC) is a 32-bit error-detection code appended to the end of every binary message sent by the GPS receiver. CRC allows the receiving system to verify data integrity to ensure that no bits were corrupted during transmission.

GNSS Header Data Structure

Field #	Field Name	Field Type	Description	Binary Bytes	Binary Offset	Ignored on Input
1	Sync	Char	Hexadecimal 0xAA.	1	0	N
2	Sync	Char	Hexadecimal 0x44.	1	1	N
3	Sync	Char	Hexadecimal 0x12.	1	2	N
4	Header Lgth	Uchar	Length of the header.	1	3	N
5	Message ID	Ushort	This is the Message ID number of the log (see the log descriptions in *Table 44, OEM4 Family Logs in Order of their Message IDs* on *Page 151* for the Message ID values of individual logs).	2	4	N
...						
14	Receiver Status	Ulong	32 bits representing the status of various hardware and software components of the receiver between successive logs with the same Message ID (see *Table 81, Receiver Status on Page 303*)	4	20	Y
15	Reserved	Ushort	Reserved for internal use.	2	24	Y
16	Receiver S/W Version	Ushort	This is a value (0 - 65535) that represents the receiver software build number.	2	26	Y

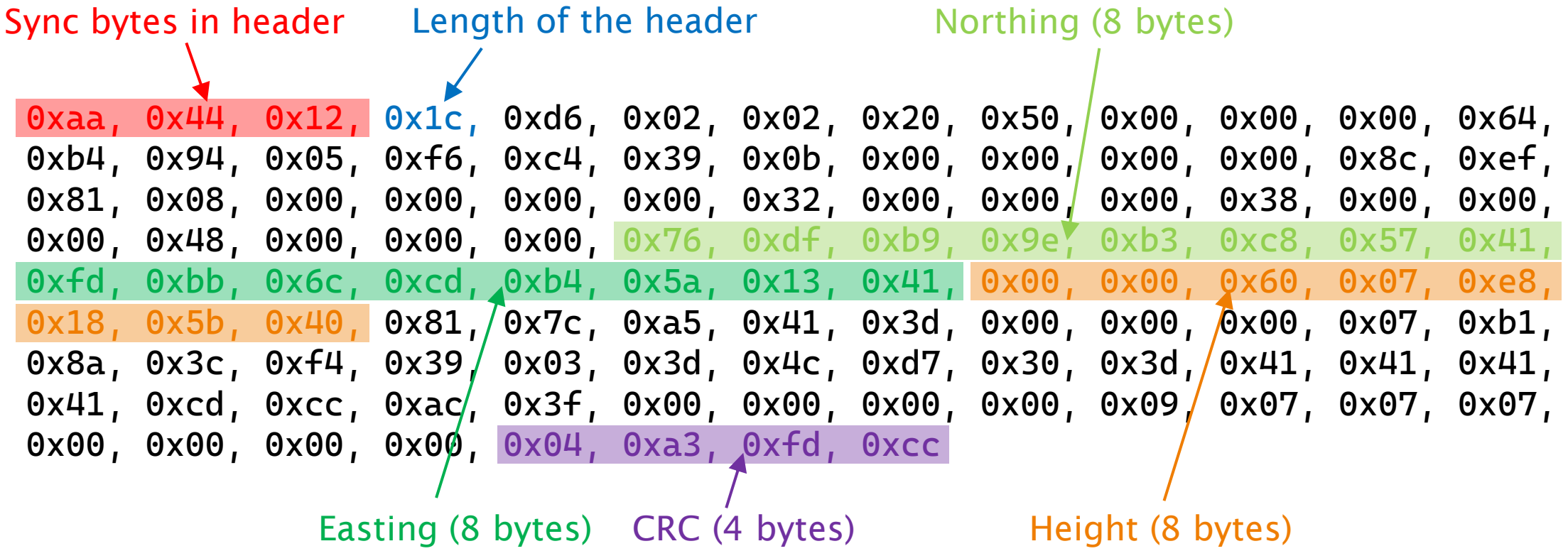
Header length = 26+2 = 28 (0x1C) bytes

GNSS UTM Data Structure

Field #	Field type	Data Description	Format	Binary Bytes	Binary Offset
1	header	Log header		H	0
2	sol status	Solution status, see *Table 48, Solution Status* on *Page 163*	Enum	4	H
3	pos type	Position type, see *Table 47, Position or Velocity Type* on *Page 162*	Enum	4	H+4
4	z#	Longitudinal zone number	Ulong	4	H+8
5	zletter	Latitudinal zone letter	Ulong	4	H+12
6	northing	Northing (m) where the origin is defined as the equator in the northern hemisphere and as a point 10000000 metres south of the equator in the southern hemisphere (that is, a 'false northing' of 10000000 m)	Double	8	H+16
7	easting	Easting (m) where the origin is 500000 m west of the central meridian of each longitudinal zone (that is, a 'false easting' of 500000 m)	Double	8	H+24
8	hgt	Height above mean sea level	Double	8	H+32
...					
25	xxxx	32-bit CRC (ASCII and Binary only)	Hex	4	H+80
26	[CR][LF]	Sentence terminator (ASCII only)	-	-	-

Total data length = 26+2+80+4 = 112 bytes

GNSS Data Stream

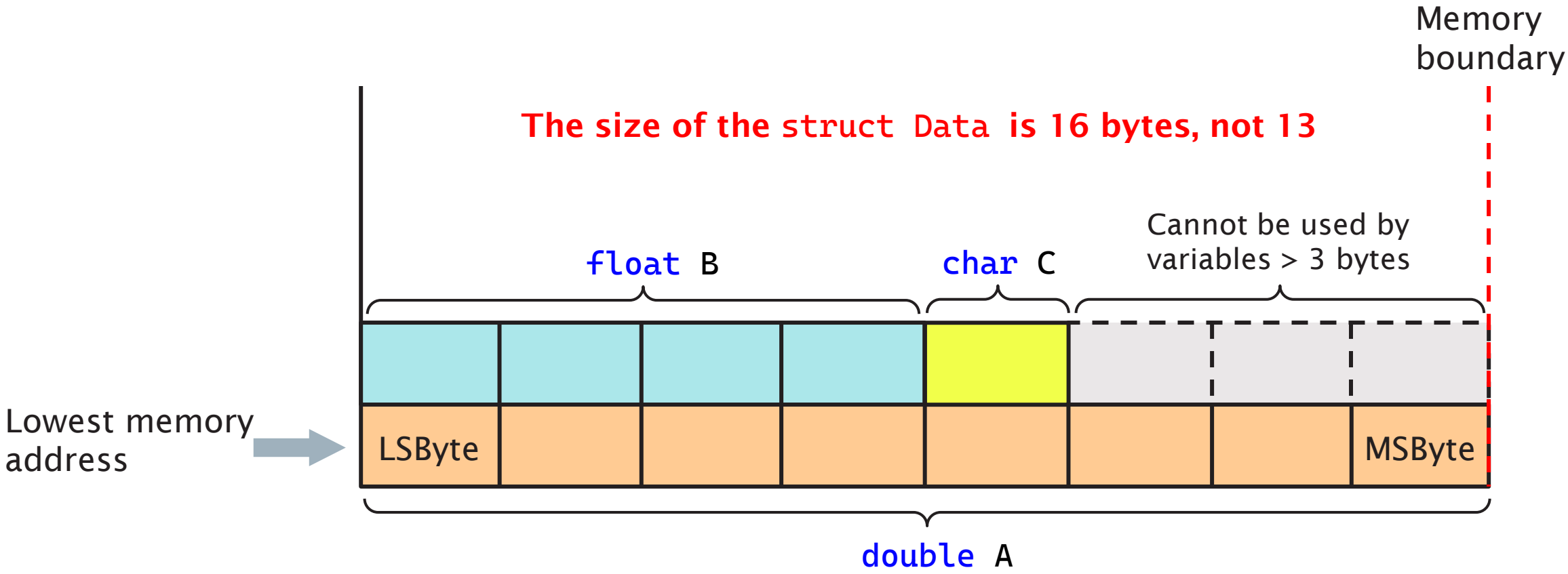


GNSS data structure

- **4 bytes of Header**
- 40 bytes to skip
- **8 bytes of Northing**
- **8 bytes of Easting**
- **8 bytes of Height**
- 40 bytes to skip
- **4 bytes of CRC**

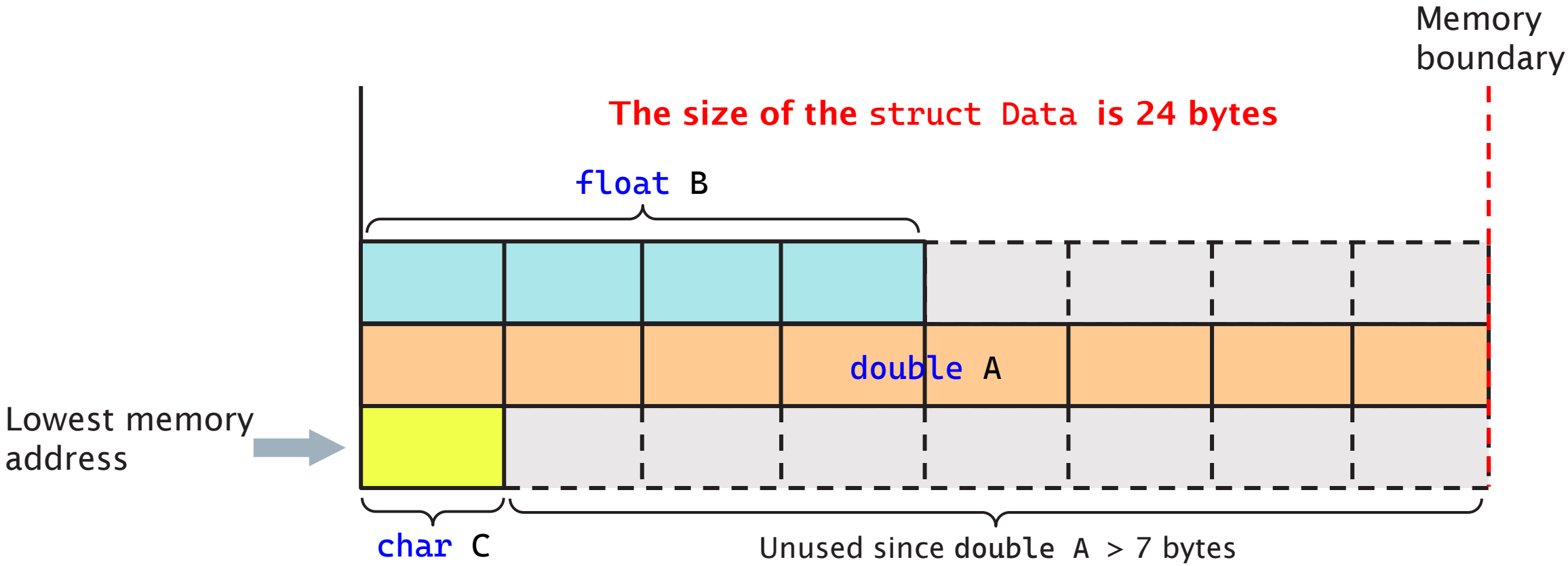
How Data Structures Store Data

```
struct Data
{
        double A; //8 bytes
        float B; //4 bytes
        char C; //1 byte
};
```



How Data Structures Store Data

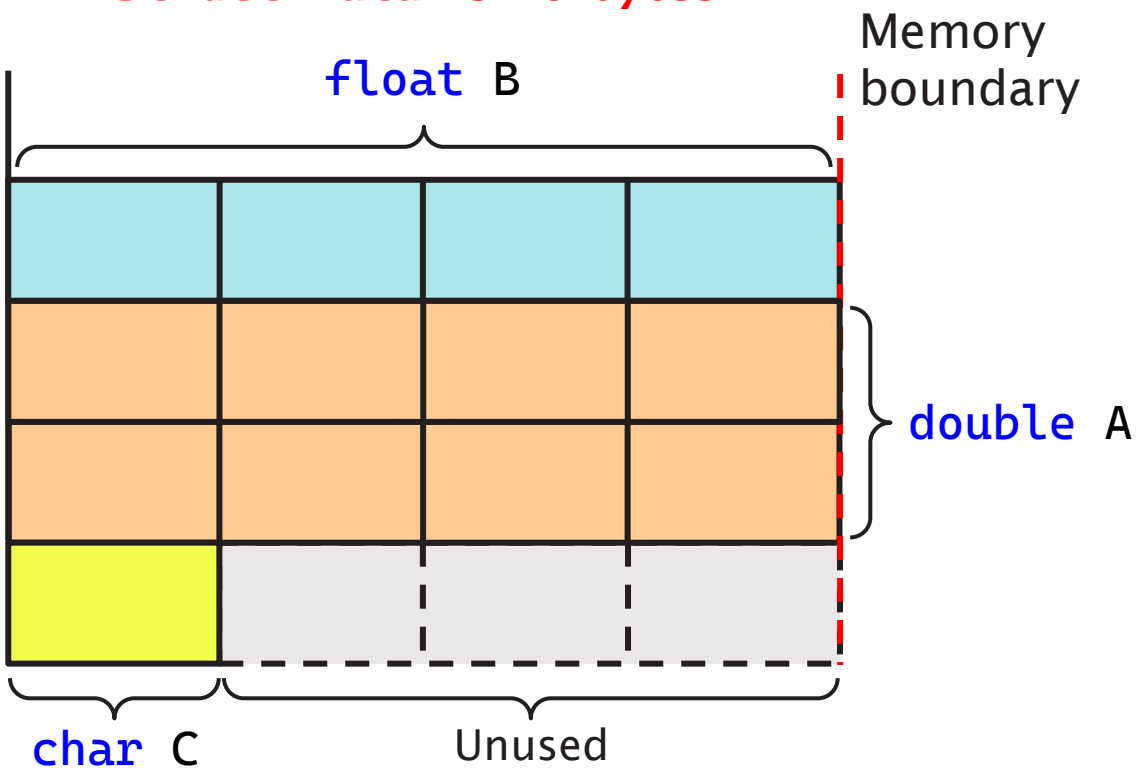
```
struct Data
{
        char C; //1 byte
        double A; //8 bytes
        float B; //4 bytes
};
```



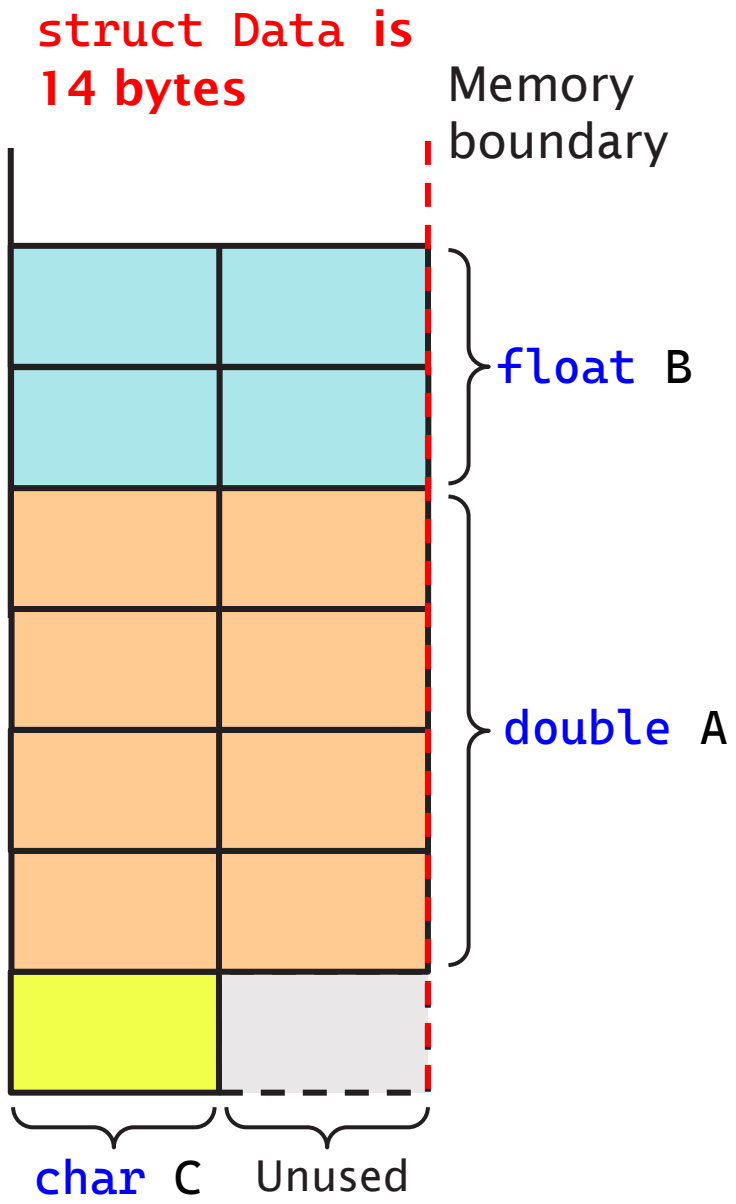
How Data Structures Store Data

```
#pragma pack(push, 4)
struct Data
{
        char C; //1 byte
        double A; //8 bytes
        float B; //4 bytes
};
```

struct Data is 16 bytes



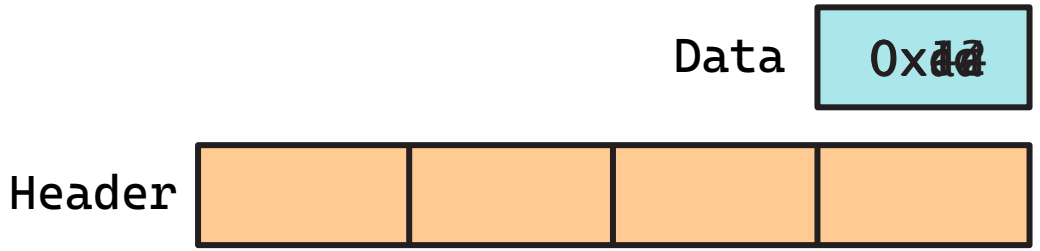
struct Data is 14 bytes



Recognise GNSS Data Header

```
0xaa, 0x44, 0x12, 0x1c, 0xd6, 0x02, 0x02, 0x20, 0x50, 0x00, 0x00, 0x00, 0x64, 0xb4, 0x94,
0x05, 0xf6, 0xc4, 0x39, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x8c, 0xef, 0x81, 0x08, 0x00, 0x00,
0x00, 0x00, 0x32, 0x00, 0x00, 0x00, 0x38, 0x00, 0x00, 0x00, 0x48, 0x00, 0x00, 0x00, 0x76,
0xdf, 0xb9, 0x9e, 0xb3, 0xc8, 0x57, 0x41, 0xfd, 0xbb, 0x6c, 0xcd, 0xb4, 0x5a, 0x13, 0x41,
0x00, 0x00, 0x60, 0x07, 0xe8, 0x18, 0x5b, 0x40, 0x81, 0x7c, 0xa5, 0x41, 0x3d, 0x00, 0x00,
0x00, 0x07, 0xb1, 0x8a, 0x3c, 0xf4, 0x39, 0x03, 0x3d, 0x4c, 0xd7, 0x30, 0x3d, 0x41, 0x41,
0x41, 0x41, 0xcd, 0xcc, 0xac, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x09, 0x07, 0x07, 0x07, 0x00,
0x00, 0x00, 0x00, 0x04, 0xa3, 0xfd, 0xcc
```

```
unsigned int Header = 0; //4 bytes: 00000000 00000000 00000000 00000000
do
{
        Data = GNSSStream->ReadByte(); //ReadByte() allows to read a single byte
                                           from a stream of (TCP) binary data
        Header = (Header << 8) | Data;
} while (Header != 0xaa44121c)
```



Appendix 1: Type Cast

```
double A = 12345.54321; // MSB 40 C8 1C C5 87 E7 C0 6E LSB
unsigned char* BytePtr = (unsigned char*)&A;
```

Expression	Meaning
`&A`	The address of `A` in memory (type: `double*`)
`(unsigned char*)&A`	**Type cast** that converts the pointer from `double*` to `unsigned char*`
`unsigned char* BytePtr`	Declares a pointer that can read/write individual bytes of `A`

Why cast to `(unsigned char*)`

- `&A` has type `double*`
- We cannot do byte-wise access (`*BytePtr++`) through a `double*` (each increment move 8 bytes)
- By casting to `unsigned char*`, you tell the compiler: "Treat this memory as a sequence of bytes, not as doubles."

If using `&A` directly, the compiler assumes each element access through that pointer is a double, so it will move **8 bytes** per step. Example:

```
double* p = &A;
p++;  //jumps 8 bytes ahead, not 1
```

Appendix 2: #pragma pack()

```
#pragma pack(push, 4)
struct GNSS
{
    unsigned int Header;
    unsigned char Discards1[40];
    double Northing;
    double Easting;
    double Height;
    unsigned char Discards2[40];
    unsigned int CRC;
};
#pragma pack(pop, 4)
```

`#pragma pack()` is a compiler directive that controls structure padding – how the compiler aligns data fields in memory.

Compilers often insert unused padding bytes between members so that each member begins at an address that matches its size requirement. This improves CPU access speed, but it makes the total struct size **larger**.

`#pragma pack(push, n)` saves the current packing alignment setting. The n specifies that members are aligned on n-byte boundaries instead of their natural boundaries.

`#pragma pack(pop, n)` restores the packing alignment to what it was before the `push`. This prevents the changed alignment from affecting other structs later in your code.

Appendix 3: BitConverter

```
Northing = BitConverter::ToDouble(ByteSet, 40);
Easting = BitConverter::ToDouble(ByteSet, 48);
Height = BitConverter::ToDouble(ByteSet, 56);
```

`BitConverter::ToDouble()` is a static method of the .NET class `System::BitConverter`, which converts **8 bytes** from a byte array into a double-precision floating-point number (IEEE-754 64-bit).

```
BitConverter::ToDouble(array<unsigned char>^ value, int startIndex);
```

Method	Bytes read	Returns
`ToChar()`	2	`char`
`ToInt16()`	2	`short`
`ToInt32()`	4	`int`
`ToUInt32()`	4	`unsigned int`
`ToInt64()`	8	`long long`
`ToSingle()`	4	`float`
`ToDouble()`	8	`double`
`ToString()`	variable	`String^`

Data to convert (byte array)

The starting position (zero-based) within the array from which to begin reading **8 bytes**.