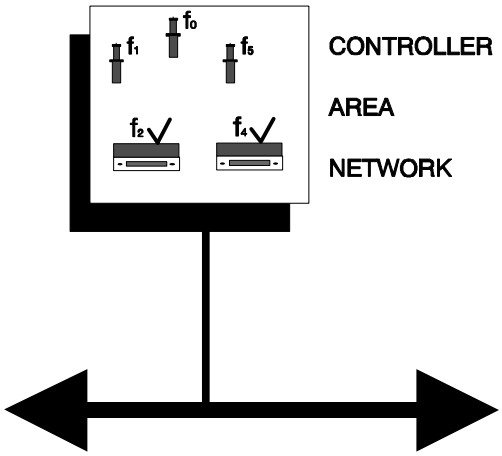
**CAN in Automation (CiA)**
**International Users and Manufacturers Group e.V.**




# CAN Application Layer for Industrial Applications
# CiA/DS202-3
# February 1996

## CMS Data Types and Encoding Rules

# 1. Scope

This document contains the encoding rules that are used to transfer CMS data values across the CAN Network and definitions application specifc extended data types. This document is part of a set of documents that standardize the CAN Application Layer for Industrial Applications.

# 2. References

/1/ CiA/DS202-1, CMS Service Specification

/2/ CiA/DS207, Application Layer Naming Conventions

/3/ ANSI/IEEE Standard 754-1985 for Binary Floating-PointArithmetic.
Reprinted in: ACM SIGPLAN Notices 22(2), 9-25 (1987).

# 3. General Description

To be able to exchange meaningful data across the CAN network, the format of this data and its meaning have to be known by the sender and receiver(s). CMS models this by the concept of data types.

The CMS encoding rules define the representation of values of data types and the CAN network transfer syntax for the repesentations. Values are represented as bit sequences. Bit sequences are transferred in sequences of octetts (bytes). For numerical data types the CMS encoding is little endian style.

Applications require data types beyond the basic data types. Using the compound data type mechanism the list of available data types can be extended. Some general extended data types are defined.

# 4. Data Type Definitions

A data type determines a relation between values and encodings for data of that type. Data types are assigned names in their type definitions. The syntax of data and data type definitions is as follows.

```
<data definition>          ::=<type name> <data name>
<type definition>          ::=<constructor> <type name>
<constructor>              ::=<compound constructor> |
                               <basic constructor>
<compound constructor>     ::=<array constructor>|
                               <structure contructor>
<array constructor>        ::=ARRAY [<array length>] OF <type name>
<structure constructor>    ::=STRUCT OF <component list>
<component list>           ::=<component> |
                               <component>, <component list>
<component>                ::=<type name> <component name>
<basic constructor>        ::=BOOLEAN |
                               VOID<bit size> |
                               INTEGER<bit size> |
                               UNSIGNED<bit size> |
                               REAL32 |
                               NIL
<array length>             ::=positive integer
<bit size>                 ::=1|2|...|64
<type name>                ::=symbolic name (see /2/)
<component name>           ::=symbolic name (see /2/)
<data name>                ::=symbolic name (see /2/)
```

Recursive definitions are not allowed.

The data type defined by <type definition> is called basic (resp.~compound) when the constructor is <basic constructor>(resp. <compound constructor>).

# 5. Bit Sequences

## 5.1 Definitions

A bit can take the values 0 or 1. Let $b_0,..., b_{n-1}$ be bits, *n* a non-negative integer. Then

$$b = b_0\ b_1\ ...\ b_{n-1}$$

is called a bit sequence of length |b| = n. The empty bit sequence of length 0 is denoted ε.

*Examples*: 10110100, 1, 101, etc. are bit sequences.

The inversion operator (¬) on bit sequences assigns to a bit sequence

$$b = b_0\ b_1\ ...\ b_{n-1}$$

the bit sequence

$$\neg b = \neg b_0\ \neg b_1 ...\ \neg b_{n-1}$$

Here ¬0 =1 and ¬1 = 0 on bits.

The basic operation on bit sequences is concatenation.

Let $a = a_0...a_{m-1}$ and $b = b_0\ ...\ b_{n-1}$ be bit sequences. Then the concatenation of *a* and *b*, denoted *ab*, is

$$ab = a_0\ ...\ a_{m-1}\ b_0\ ...\ b_{n-1}$$

*Example:* (10)(111) = 10111 is the concatenation of 10 and 111.

The following holds for arbitrary bit sequences *a* and *b*:

$$|ab| = |a| + |b|$$

and

$$\varepsilon a = a\varepsilon = a$$

## 5.2 Transfer Syntax

For transmission across a CAN network a bit sequence is reordered into a sequence of octetts. Here and in the following hexadecimal notation is used for octetts. Let $b=b_0...b_{n-1}$ be a bit sequence with $n \leq 64$. Denote *l* a non-negative integer such that $8(l-1)<n\leq 8l$. Then *b* is transferred in *l* octetts assembled as shown in Figure 1. The bits $b_i$, $i \geq n$ of the highest numbered octett are don't care bits.

Octett 1 is transmitted first and octett *l* is transmitted last. Hence the bit sequence is transferred as follows across the CAN network:

$$b_7, b_6,...,b_0,b_{15},...,b_8,...$$

| | 7 | 6 | ... | 0 |
|---|---|---|---|---|
| 1 | $b_7$ | $b_6$ | ... | $b_0$ |
| 2 | $b_{15}$ | $b_{14}$ | ... | $b_8$ |
| *l* | $b_{8l-1}$ | $b_{8l-2}$ | ... | $b_{8l-8}$ |

Figure 1: Transfer Syntax for Bit Sequences

*Example:* The bit sequence 0011 1000 01 is transferred in two octetts: First $1c_h$ and then $02_h$.

# 6. Basic Data Types

For basic data types <type name> equals the literal string of the associated constructor, e.g.,

BOOLEAN BOOLEAN

is the type definition for the Boolean data type.

## 6.1 NIL

Data of basic data type NIL is represented by ε.

## 6.2 Boolean

Data of basic data type BOOLEAN attains the values TRUE or FALSE. The values are represented as bit sequences of length 1. The value TRUE (resp. FALSE) is represented by the bit sequence 1 (resp.0).

## 6.3 Void

Data of basic data type VOIDn is represented as bit sequences of length *n*. The value of data of type VOIDn is undefined. The bits in the a sequence of data of type VOIDn must either be specified explicitly or else marked "don't care".

Data of type VOIDn is useful for reserved fields and for aligning components of compound values on octett boundaries.

## 6.4 Unsigned Integer

Data of basic data type UNSIGNEDn has values in the non-negative integers. The value range is 0, ..., $2^n$-1. The data is represented as bit sequences of length *n*. The bit sequence

$b = b_0 \ldots b_{n-1}$

is assigned the value

$\text{UNSIGNEDn}(b) = b_{n-1}(2^n-1) + \ldots + b_1 2^1 + b_0 2^0$

Note that the bit sequence starts on the left with the least significant bit.

*Example:* The value 266 with data type UNSIGNED16 is transferred in two octetts across the bus, first $0a_h$ and then $01_h$.

## 6.5 Signed Integer

Data of basic data type INTEGERn has values in the integers. The value range is $-2^{n-1}$, ..., $2^{n-1}-1$. The data is represented as bit sequences of length $n$. The bit sequence $b = b_0 .. b_{n-1}$ is assigned the value

$$\text{INTEGERn}(b) = b_{n-2}\, 2^{n-2} + ... + b_1\, 2^1 + b_0\, 2^0 \quad \text{if } b_{n-1} = 0$$

and, performing two's complement arithmetic,

$$\text{INTEGERn}(b) = \text{INTEGERn}(\hat{}b) - 1 \quad \text{if } b_{n-1} = 1$$

Note that the bit sequence starts on the left with the least significant bit.

*Example:* The value -266 with data type INTEGER16 is transferred in two octetts across the bus, first $f6_h$ and then $fe_h$.

## 6.6 Floating Point Number

Data of basic data type REAL32 has values in the real numbers.
The data is represented as bit sequences of length 32. The encoding of values follows the IEEE 754-1985 Standard for floating point numbers, see the reprint /3/.

A bit sequence of length 32 either has a value (finite non-zero real number, $\pm 0$, $\pm$ _) or is NaN (not-a-number). The bit sequence $b = b_0 \ldots b_{31}$ is assigned the value (finite non-zero number)

$$\text{REAL32}(b) = (-1)^S\, 2^{E-127}\, (1 + F)$$

Here $S=b_{31}$ is the sign. $E = b_{30}\, 2^7 + \ldots + b_{23}\, 2^0$, $0 < E < 255$, is the un-biased exponent. $F = 2^{-23}\ (b_{22}\, 2^{22} + \ldots + b_1\, 2^1 + b_0\, 2^0)$ is the fractional part of the number. $E = 0$ is used to represent $\pm$ 0. $E = 255$ is used to represent infinities and NaN's. Note that the bit sequence starts on the left with the least significant bit.

*Example:* $6.25 = 2^{E-127}\, (1 + F)$ with $E = 129 = 2^7 + 2^0$ and $F = 2^{-1} + 2^{-4} = 2^{-23}(2^{22}+2^{19})$. Hence the number is represented as:

6.25: 0000 0000 0000 0000 0001 0011 0000 0010

Figure 2 shows example encodings for REAL32 as sequences of four octetts for transfer across the CAN network.

# 7. Compound Data Types

Type definitions of compound data types expand to a unique list of type definitions involving only basic data types. Correspondingly, data of compound type ´*type_name*´ are ordered lists of component data named component_i of basic type ´*basic_type_i*´.
Compound data types constructors are ARRAY and STRUCT OF.

STRUCT OF
<basic_type_1> <component_1>,
<basic_type_2> <component_2>,
… …
<basic_type_N> <component_N>
<type_name>

ARRAY [<length>] OF <basic_type> <type_name>

The bit sequence representing data of compound type is obtained by concatenating the bit sequences representing the component data. Assume that the components ´*component_i*´ are represented by bit sequences $b_i$, for $i$ =1,…,N Then the compound data is represented by the concatenated sequence $b_1 \ldots b_N$.

*Example:* Consider the data type

STRUCT OF
INTEGER10 i,
UNSIGNED5 u
NewData

Assume $i$ = - 423 and $u$ =30. Let $b(i)$ and $b(u)$ denote the bit sequences representing the values of $i$ and $u$, respectively. Then:

$b(i)$ = 1001101001
$b(u)$ = 01111
$b(iu) = b(i)\ b(u)$ = 1001101001 01111

The value of the structure is transferred with two octetts, first $59_h$ and then $79_h$.

# 8. Extended Data Types

The extended data types consist of the basic data types and the compound data types defined in the following subsections.

## 8.1 Octett String

The data type *OctettString<length>* is defined below. *length* is the length of the octett string.

```
ARRAY [<length>]   OF UNSIGNED8        OctettString<length>
```

## 8.2 Visible String

The data type *VisibleString<length>* is defined below. The admissable values of data of type VisibleChar are $0_h$ and the range from $20_h$ to $7E_h$. The data are interpreted as ASCII characters. *length* is the length of the visible string.

```
UNSIGNED8                          VisibleChar
ARRAY[<length>]  OF VisibleChar    VisibleString<length>
```

## 8.3 Date

The data type *Date* is defined below. It follows from the definition and the encoding rules that data of type *Date* is represented as bit sequences of length 56.

```
STRUCT OF
        UNSIGNED16      ms,
        UNSIGNED6       min,
        VOID2           reserved_1,
        UNSIGNED5       hour,
        VOID2           reserved_2,
        BOOLEAN         su,
        UNSIGNED5       day_of_month,
        UNSIGNED3       day_of_week,
        UNSIGNED6       month,
        VOID2           reserved_3,
        UNSIGNED7       year,
        VOID1           reserved_4
Date
```

Figure 1 contains descriptions and value ranges for the components of data of type *Date*. The components reserved_*i*, *i* =1,...,4, are reserved with all bits equal 0.

| Component | Description | Value Range |
|---|---|---|
| ms | milliseconds | 0,...,59999 |
| min | minutes | 0,...,59 |
| hour | hour | 0,...,23 |
| su | standard or summer time | 0 = standard, 1 = summer |
| day_of_month | day of month | 1,...,31 |
| day_of_week | day of week | 1 = monday, 7 = sunday |
| month | month | 1,...,12 |
| year | year modulo centuries | 0,...,99 |

Figure 1:Descriptions for *Date*

## 8.4 Time of Day

The data type *TimeOfDay* represents absolute time. It follows from the definition and the encoding rules that *TimeOfDay* is represented as bit sequences of length 48.

Component ms is the time in milliseconds after midnight. Component days is the number of days since January 1, 1984.

```
STRUCT OF
        UNSIGNED28      ms,
        VOID4           reserved_1,
        UNSIGNED16      days
TimeOfDay
```

## 8.5 Time Difference

The data type *TimeDifference* represents a time difference. It follows from the definition and the encoding rules that *TimeDifference* is represented as bit sequences of length 48.

Time differences are sums of numbers of days and milliseconds. Component ms is the number milliseconds. Component days is the number of days.

```
STRUCT OF
        UNSIGNED28      ms,
        VOID4           reserved_1,
        UNSIGNED16      days
TimeDifference
```