Type Encoding of Logix Structures

in CIP Data Table R/W

(3-Nov-06, Rev. 1.2)

Type Encoding of Logix Structures in CIP Data Table R/W

Logix products support the following communication services to access named data (Tags): **CIP Data Table Read** and **CIP Data Table Write** (CIP DT R/W)**.** These services contain an abbreviated type code as part of the service data. This document is an aid to users who need to compute the abbreviated type code of a Logix data structure for these services.

"CIP" refers to the application-level **"Common Industrial Protocol"** shared between ControlNet, DeviceNet, EtherNet/IP, and Componet. The CIP Standard is available from ODVA, the Open Device Vendor Association (www.odva.org). The calculation of the structure abbreviated type code for the vendor-specific CIP DT R/W services uses the standard CRC algorithm (CRC-16 polynomial) described in the "CIP Common" specification (Appendix C: Data Management, section C-7), but the calculation is performed on a vendor-specific "**Type Encoding String**" rather than a CIP standard type encoding string.

This abbreviated type code can also be obtained without calculation from:

1. the reply packet of a CIP Data Table Read of a structure Tag, or
2. the Logix structure template (refer to "LDA manual" below).

Note: This document was originally intended as an appendix to the **"Logix Data Access Reference Manual"**, publication 1756-RM005A-EN-E. Please refer to that document for further information on the CIP services and data types supported by the Rockwell Automation Logix family of products. This "LDA manual" is available for download from the www.ab.com website.

As shown in the "CIP Commands" section of the "LDA manual", the CIP DT R/W service data includes the named data (Tag) along with the **Abbreviated Data Type**. Abbreviated Data Types are described in the "Data Type Reporting" section of the LDA manual. This type code is used to check that the data type of the Tag matches at the client and server. Note that the structure type code, as a CRC, is not totally unique to that structure. The type code would likely change if the structure data type is modified, allowing the client or server to detect a mismatch, however, the type code does not completely guarantee uniqueness.

Logix data structures are mixed collections of

- atomic types; e.g. SINT, INT, DINT, REAL, etc.
- arrays of any type, atomic or structure.
- other structures

Logix data structures are categorized as:

- Predefined Data Types (PDT); e.g. Counter, Timer, PID, etc. (including String).
- Module-Defined Data Types (MDT); created by the module's configuration profile.
- User-Defined Data Types (UDT); a data structure created by the user.

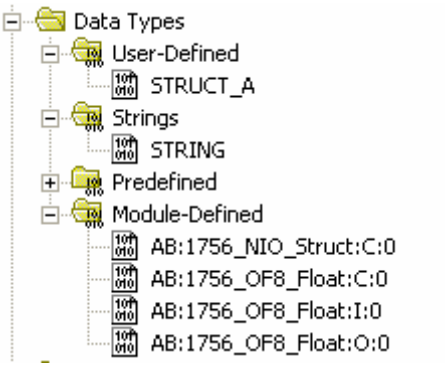
***** NOTE *****

This document only applies to UDT structures, and only those UDTs without hidden members (except for BOOL – see below) and without nested PDTs or MDTs.

PDT and MDT structures can contain hidden members, and can change over time, and are therefore not addressed by this document. It's also possible for a user to insert a hidden member

into a UDT by importing a modified L5K export file, but this document does not address UDTs that have been modified in that manner. BOOL are implemented either as a hidden SINT or a hidden 64-bit array. This is further explained in the examples to follow.

Logix structure data types are shown below as seen in the organizer view of a RSLogix5000 project. This example is from a project where the user has created a UDT (STRUCT_A), and a 1756-OF8 module has been added to the system, automatically creating the OF8 Input, Output, and Config MDTs. The PDT (Predefined) list is not expanded below, but includes over 80 structures. A STRING is a form of UDT.



In Logix, the structure data type is described in an ASCII **Type Encoding String** containing:

- Name of structure
- Name of member structures
- Name of member data types (e.g. SINT, DINT, TIMER, etc)
- Comma delimiters between each element of the string

The **order of the elements** in the Type Encoding String is based on the order of the data in the structure template, which is also the order of data in Logix memory and in the CIP DT R/W packet. Typically, for UDTs, this is the same as the order of the structure members as they appear in the RSLogix5000 Data Monitor view, but that may not always be the case. The order of data can be confirmed by reading the **structure template** (see LDA manual), or viewing the L5K export file, and checked whenever needed. A change in order of members may not always result in a different abbreviated type code.

Data is located in Logix memory according to certain **alignment rules**:

- Structures, Arrays, DINT, and REAL begin and end on 32-bit word boundaries
- INTs are on 16-bit word boundaries.
- SINTs are on 8 bit word boundaries
- BOOLs of 8 or less are packed into a SINT.
- BOOLs of more than 8 are packed into a 64-bit array, BOOL[64], with 32 bit alignment.

The data in the CIP DT R/W packet is the same as in Logix memory, including **pad bytes** between data members which result from this alignment.

The structure **Type Encoding String** is constructed as follows

1. The ASCII string starts with the name of the structure.
2. Concatenate the names of the data types of the members of the structure in their order in the structure, each separated by a comma.
3. BOOL are mapped to a hidden SINT or 64-bit array. Adjacent BOOL of 8 or less are replaced with "SINT" type in the string. Adjacent BOOL of more than 8 are replaced with a 64-bit array and "BOOL[64]" is included in the string.
4. Each nested structure is indicated by the structure name followed by the names of the data type of its members.
5. Each nested array is indicated by the name of the data types of its members, followed by the array size in brackets. Only single dimension arrays can be nested in Logix structures. For arrays of structures, the structure members are inserted between the structure name and the bracket, without a comma ahead of the bracket.

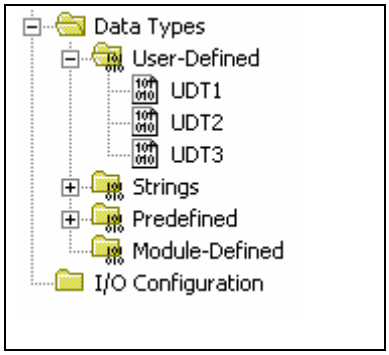
Note: Because the name of the structure data type and its nested structures are included in the Type Encoding String, these names have to be known in the both client and server.

The "**abbreviated type code**" is the CRC calculated over the Type Encoding String (including comma delimeters, but not including quotes).

Example Type Encoding Strings

Type Encoding Strings can get complicated, especially with nested UDTs and Arrays. Below is an example to illustrate the construction of the String for a UDT with nested UDTs.

Three UDT's are created in an RSLogix5000 project, as defined below. UDT2 includes UDT3 as a member, and UDT1 includes both UDT2 and UDT3 as members.



UDT1 contains a UDT2 and an array of UDT3

	Name	Data Type	Style	Description
	U1A	SINT	Decimal	
	U1B	SINT[2]	Decimal	
⊞	U1C	UDT2		
⊞	U1D	UDT3[4]		
✳				

UDT2 contains UDT3 and an array of UDT3

Name	Data Type	Style	Description
U2A	DINT	Decimal	
U2B	SINT[3]	Decimal	
⊞ U2C	UDT3		
⊞ U2D	UDT3[2]		
✳			

UDT3 does not contain any other UDTs

Name	Data Type	Style	Description
U3A	SINT	Decimal	
U3B	SINT[4]	Decimal	
✳			

This is a screenshot view of UDT1 data type in RSLogix5000 Data Monitor, expanded to show the members and sub-members

Members: Data Type Size: 72 byte(s)

Name	Data Type	Style	Description
U1A	SINT	Decimal	
U1B	SINT[2]	Decimal	
⊟ U1C	UDT2		
—U2A	DINT	Decimal	
—U2B	SINT[3]	Decimal	
⊟ U2C	UDT3		
—U3A	SINT	Decimal	
—U3B	SINT[4]	Decimal	
⊟ U2D	UDT3[2]		
—U3A	SINT	Decimal	
—U3B	SINT[4]	Decimal	
⊟ U1D	UDT3[4]		
—U3A	SINT	Decimal	
—U3B	SINT[4]	Decimal	
✳			

When the project is saved as an L5K file, these data types are described as follows:

```
DATATYPE UDT1 (FamilyType := NoFamily)
      SINT U1A;
      SINT U1B[2];
      UDT2 U1C (Radix := Decimal);
      UDT3 U1D[4] (Radix := Decimal);
END_DATATYPE
```



```
DATATYPE UDT2 (FamilyType := NoFamily)
      DINT U2A;
      SINT U2B[3];
      UDT3 U2C (Radix := Decimal);
      UDT3 U2D[2] (Radix := Decimal);
END_DATATYPE

DATATYPE UDT3 (FamilyType := NoFamily)
      SINT U3A;
      SINT U3B[4];
END_DATATYPE
```

The UDT3 data type does not contain any other UDTs, so its Type Encoding String is created from the above data types just by concatenating the UDT name with the data types of the individual members:

"**UDT3,SINT,SINT[4]**"

The UDT2 data type contains UDT3. Its Type Encoding String is created in a similar manner, concatenating the member types, but where UDT3 appears, the whole String for UDT3 is inserted. For the array of UDT2, the array subscript is appended to the UDT3 String **without a comma separator**.

"UDT2,DINT,SINT[3],**UDT3,SINT,SINT[4],UDT3,SINT,SINT[4]**[2]"

The UDT1 data type contains both UDT2 and UDT3 as members. Its String is composed in the same manner, inserting the whole UDT2 String above where it UDT2 appears.

"UDT1,SINT,SINT[2],**UDT2,DINT,SINT[3],UDT3,SINT,SINT[4],UDT3,SINT,SINT[4][2]**,UDT3,SINT,SINT[4][4]"

Note: If a Tag is created which is an array of UDT1, its data type is still the same as UDT1, and the Type Encoding String is the same as UDT1.

If a new data type, UDT0, is created which is an array of 10 UDT1, its Type Encoding String would differ from UDT1, as follows:

"**UDT0**,UDT1,SINT,SINT[2],UDT2,DINT,SINT[3],UDT3,SINT,SINT[4],UDT3,SINT,SINT[4][2],UDT3,SINT,SINT[4][4]**[10]**"

Note again how the array subscript is appended to the end, without a comma separator.

In this example, the byte order is shown below in the expanded column, with "PAD" bytes inserted since DINT must be aligned on a 32-bit word boundary, and the packet must contain multiples of 32-bit words.

Data Monitor	**Byte order (without pad bytes)** (low byte first – little endian)	**Padding of bytes in packet** (low byte left – high byte right)
SINT	SINT,	00 **00 00 00**
SINT[2]	SINT,SINT,	11 22 **00 00**
UDT2		
DINT	DINT,	44 33 00 00
SINT[3]	SINT,SINT,SINT,	55 66 77 **00**
UDT3		
SINT	SINT,	88 **00 00 00**
SINT[4]	SINT,SINT,SINT,SINT,	99 aa bb cc
UDT3[2]		
SINT	SINT,	dd **00 00 00**
SINT[4]	SINT,SINT,SINT,SINT,	ee ff 10 11
	SINT,	12 **00 00 00**
	SINT,SINT,SINT,SINT,	13 14 15 16
UDT3[4]		
SINT	SINT,	17 **00 00 00**
SINT[4]	SINT,SINT,SINT,SINT,	18 19 1a 1b
	SINT,	1c **00 00 00**
	SINT,SINT,SINT,SINT,	1d 1e 1f 20
	SINT,	21 **00 00 00**
	SINT,SINT,SINT,SINT,	22 23 24 25
	SINT,	26 **00 00 00**
	SINT,SINT,SINT,SINT	27 28 29 2a

The CRC calculation over the UDT1 Type Encoding String above produces the "abbreviated data type" value of **0x5f58** which is highlighted in the packet trace below (along with pad bytes) of the response packet to the CIP Read of UDT1 over EtherNet/IP.

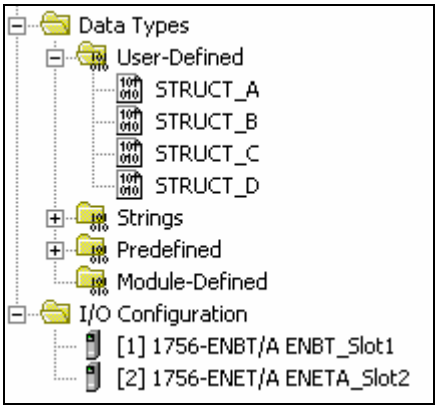
```
0000  00 00 bc 05 1d c4 00 00 bc 05 1d c7 08 00 45 00
0010  00 a6 6e 97 00 00 40 06 fb 4a 82 97 85 a1 82 97
0020  85 a0 af 12 04 04 c0 b7 04 ae c0 52 71 be 50 18
0030  10 00 16 c5 00 00 70 00 66 00 00 06 02 0f 00 00
0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0050  00 00 00 00 02 00 a1 00 04 00 01 11 9e 00 b1 00
0060  52 00 53 82 cc 00 00 00 a0 02 58 5f 00 00 00 00
0070  11 22 00 00 44 33 00 00 55 66 77 00 88 00 00 00
0080  99 aa bb cc dd 00 00 00 ee ff 10 11 12 00 00 00
0090  13 14 15 16 17 00 00 00 18 19 1a 1b 1c 00 00 00
00a0  1d 1e 1f 20 21 00 00 00 22 23 24 25 26 00 00 00
00b0  27 28 29 2a
```


The following screen shot is the RSLogix5000 Data Monitor view of a structure Tag named "complex" of type "UDT1" with the data values in the packet above. It shows the hierarchy of data in the structure and the corresponding order of bytes in the packet. Note the low-byte first (little endian) order of the bytes as shown by the DINT member.

Tag Name	Value	Force Mas	Style	Type
-complex	{...}	{...}		UDT1
+complex.U1A	16#00		Hex	SINT
-complex.U1B	{...}	{...}	Hex	SINT[2]
+complex.U1B[0]	16#11		Hex	SINT
+complex.U1B[1]	16#22		Hex	SINT
-complex.U1C	{...}	{...}		UDT2
+complex.U1C.U2A	16#0000_3344		Hex	DINT
-complex.U1C.U2B	{...}	{...}	Hex	SINT[3]
+complex.U1C.U2B[0]	16#55		Hex	SINT
+complex.U1C.U2B[1]	16#66		Hex	SINT
+complex.U1C.U2B[2]	16#77		Hex	SINT
-complex.U1C.U2C	{...}	{...}		UDT3
+complex.U1C.U2C.U3A	16#88		Hex	SINT
-complex.U1C.U2C.U3B	{...}	{...}	Hex	SINT[4]
+complex.U1C.U2C.U3B[0]	16#99		Hex	SINT
+complex.U1C.U2C.U3B[1]	16#aa		Hex	SINT
+complex.U1C.U2C.U3B[2]	16#bb		Hex	SINT
+complex.U1C.U2C.U3B[3]	16#cc		Hex	SINT
-complex.U1C.U2D	{...}	{...}		UDT3[2]
-complex.U1C.U2D[0]	{...}	{...}		UDT3
+complex.U1C.U2D[0].U3A	16#dd		Hex	SINT
-complex.U1C.U2D[0].U3B	{...}	{...}	Hex	SINT[4]
+complex.U1C.U2D[0].U...	16#ee		Hex	SINT
+complex.U1C.U2D[0].U...	16#ff		Hex	SINT
+complex.U1C.U2D[0].U...	16#10		Hex	SINT
+complex.U1C.U2D[0].U...	16#00		Hex	SINT
+complex.U1C.U2D[1]	{...}	{...}		UDT3
+complex.U1D	{...}	{...}		UDT3[4]

Example Type Encoding Strings of the Addressing Examples in Chapter 2 of LDA Manual

Below are the view in the RSLogix5000 controller organizer, the view of each structure in RSLogix5000 Data Monitor, the data type in the L5K export file, and the corresponding Type Encoding String.



STRUCT_A

Members: Data Type Size: 16 byte(s)

	Name	Data Type	Style	Description
	limit4	BOOL	Decimal	
	limit7	BOOL	Decimal	
	travel	DINT	Decimal	
	errors	DINT	Decimal	
	wear	REAL	Float	
*				

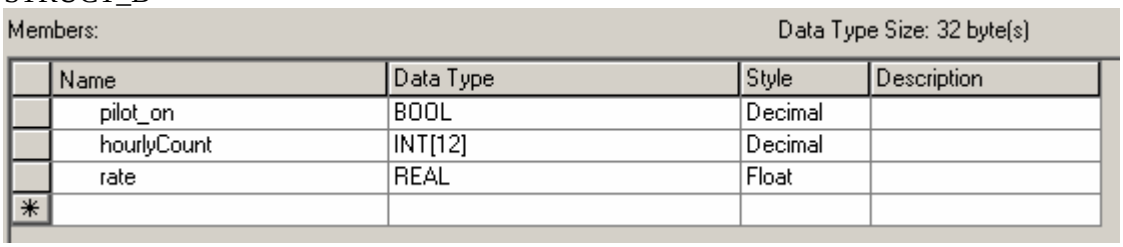
```
DATATYPE STRUCT_A (FamilyType := NoFamily)
                SINT ZZZZZZZZZZSTRUCT_A0 (Hidden := 1);
                BIT limit4 ZZZZZZZZZZSTRUCT_A0 : 0;
                BIT limit7 ZZZZZZZZZZSTRUCT_A0 : 1;
                DINT travel;
                DINT errors;
                REAL wear;
        END_DATATYPE
```

STRUCT_A Type Encoding String:

"**STRUCT_A,SINT,DINT,DINT,REAL**"

In this structure the BOOL are mapped to the hidden SINT and it is the SINT which is included in the Type Encoding String.

STRUCT_B



Members: Data Type Size: 32 byte(s)

	Name	Data Type	Style	Description
	pilot_on	BOOL	Decimal	
	hourlyCount	INT[12]	Decimal	
	rate	REAL	Float	
*				

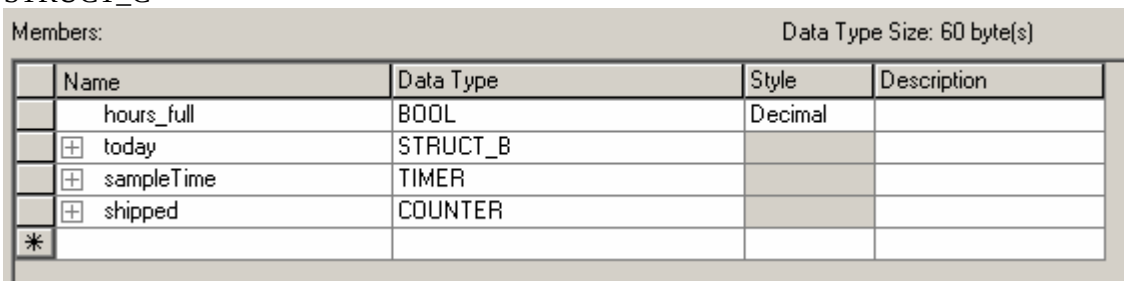
```
DATATYPE STRUCT_B (FamilyType := NoFamily)
        SINT ZZZZZZZZZZSTRUCT_B0 (Hidden := 1);
        BIT pilot_on ZZZZZZZZZZSTRUCT_B0 : 0;
        INT hourlyCount[12];
        REAL rate;
END_DATATYPE
```

STRUCT_B Type Encoding String:

"**STRUCT_B,SINT,INT[12],REAL**"

As with STRUCT_A, the BOOL is mapped to the hidden SINT and the SINT is included in the Type Encoding String. The array of INT is included with the dimension of the array in brackets.

STRUCT_C



Members: Data Type Size: 60 byte(s)

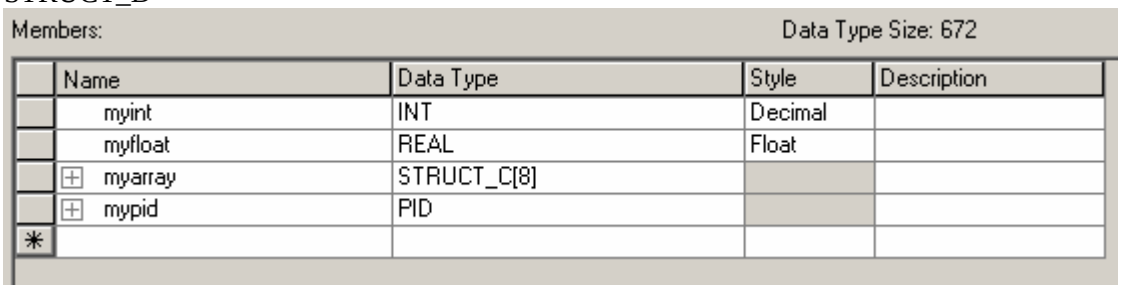
	Name	Data Type	Style	Description
	hours_full	BOOL	Decimal	
⊞	today	STRUCT_B		
⊞	sampleTime	TIMER		
⊞	shipped	COUNTER		
*				

```
DATATYPE STRUCT_C (FamilyType := NoFamily)
        SINT ZZZZZZZZZZSTRUCT_C0 (Hidden := 1);
        BIT hours_full ZZZZZZZZZZSTRUCT_C0 : 0;
        STRUCT_B today (Radix := Decimal);
        TIMER sampleTime (Radix := Decimal);
        COUNTER shipped (Radix := Decimal);
END_DATATYPE
```

STRUCT_C Type Encoding String:

***** NOT ADDRESSED BY THIS DOCUMENT SINCE IT CONTAINS A TIMER AND A COUNTER, BOTH OF WHICH ARE PREDEFINED (PDT) STRUCTURES ***.**

STRUCT_D


Members: Data Type Size: 672

Name	Data Type	Style	Description
myint	INT	Decimal	
myfloat	REAL	Float	
⊞ myarray	STRUCT_C[8]		
⊞ mypid	PID		
*			

```
DATATYPE STRUCT_D (FamilyType := NoFamily)
                    INT myint;
                    REAL myfloat;
                    STRUCT_C myarray[8] (Radix := Float);
                    PID mypid (Radix := Float);
                END_DATATYPE
```

STRUCT_D Type Encoding String:

***** NOT ADDRESSED BY THIS DOCUMENT SINCE IT CONTAINS PREDEFINED (PDT) STRUCTURES *****

1. **PID**
2. **TIMERS IN NESTED ARRAY OF STRUCT_C**
3. **COUNTERS IN NESTED ARRAY OF STRUCT_C**

