CANed Objects

by
Alan M Robertson BSc
RAM Technology Systems Ltd – Poole Dorset UK

***Abstract**: The CAN's compact, eight byte, data frame makes a simple object message transport media for intelligent distributed transducers.*

Introduction

At the start of the development for a distributed transducer system it was decided to do the implementation using the IEEE 1451 specification. Part of this is the **T**ransducer **E**lectronic **D**ata **S**heets that describe the transducer and its capabilities. It soon became apparent that these TEDS were useable but did not do a great deal for the 'intelligence' of the transducer. The TEDS are a fixed format and any modification during commissioning requires the entire TEDS to be downloaded, modified and then uploaded. The position and size of each record is needed by the commissioning tool to enable this to be done and there is little room for future produce enhancement. Part I of the IEEE 1451 specification describes an object orientated interface that tries to simplify the intricacies of the TEDS and the system in general.

It was obvious that if the transducer had an object interface of its own it may be possible to interrogate the transducer and find out all that was needed to know with little prior knowledge. The following is the simple scheme devised to do this.

Objects

Objects are an encapsulated set of data and methods that may only be accessed with a message. The message is unique for the object and the data being interrogated or updated e.g.

Read: Value. Temperature

The message result is obvious if you understand the English language. We are asking for a temperature reading by applying the method **Read:** to the data **Value.** in the object **Temperature**. Following the convention used by Win32Forth[(1)] Classes[(3)], all the methods end in a colon (**:**) and all the data selectors end in a dot (**.**). Object names are just their ASCII equivalent.

This gives us the reading but we do not know if the temperature is in Kelvin, Celsius or Farenheight. To obtain this we might find the object would respond to:

Read: Units. Temperature

The data selectors are objects and the methods available to each are held in an ASCII comma delimited string whose data selector is **IV-List.** Sending the following message;

Read: IV-List. Temperature

may result in the following reply.

ObjectType.,Read:,Size:,Style:,Default.,Read:,Write:,Size:,Style:,Maximum.,Read:,Write:,Size:,Style:,Minimum.,Read:,Write:,Size:,Style:,Value.,Read:,Write:,Size:,Style:,Uncertainty.,Read:,Size:,Style:,Description.,Read:,Write:,Size:,Style:,Unit.,Read:,Write:,Size:,Style:Init.,Do:,Style:,IV-List.,Read:,Size:,Style:,

This comma delimited string gives you all you ever needed to know about the temperature object in this transducer. You will notice that the **IV-List.** selector is part of the string as this is necessary, as we will see later, when we need to hash the message.

All transducers have a primary or meta object called **Device**. This object has an **IV-List.** but also has the **ObjectList.**, which as its name implies is a comma delimited list of all the objects in the transducer. To reduce the prior knowledge required to interrogate the transducers a special form of message is used initially to retrieve the **Device ObjectList.** and **IV-List.**

Classes

Objects are created from a Class which is a template for the object. In our simple implementation the class structures do not allow reference to other classes, except the special classes created to implement the data selectors. The data may be in RAM, Flash ROM or EEPROM. For each type of memory there is an data class called **IN-RAM**, **IN-ROM**, **IN-FLASH** and **IN-EEPROM**. There is also another data type called **METHOD** which allows an address to be placed in EEPROM that may be executed by the **DO:** method. This gives us the flexibility to design the Object Interface at the beginning of the application and then code the functions performed by the objects later. The data classes already have their own methods for the manipulation of the data, **Read:, Write:** etc.

CANed Messages

The CAN data frame consists of a 10 or 29 bit ID field to distinguish the node, up to eight bytes of data and a checksum. To send a **Read:Value.Temperature** message as ASCII would take three CAN frames to transmit. However, if we hash the ASCII text of the method, selector and the object using an 8 bit CRC generator, $x^8+x^5+x^4+1$, we only need three bytes. The following code is used to generate the CRC;

```
: CRC+ ( c - )  8 0 DO DUP 2/ SWAP 1 AND
                       CHECKSUM C@ SWAP
                       IF  1 XOR
                       THEN
                           DUP  1 AND
                           IF   $18 XOR
                                2/ $80 OR
                           ELSE 2/ $7F AND
                           THEN CHECKSUM C!
                   LOOP DROP ;

: ?HASH ( addr count - c )
```

```
0 CHECKSUM C!
BOUNDS
DO
   I C@ CRC+
LOOP
CHECKSUM C@ ;
```

We allow 32 bit, 4 bytes, of data to be sent or returned at a time. This is fine for most data values but strings need much more. To address this we use the byte that is left from the 8 bytes available, 4 bytes of data plus 3 bytes for message keys, to specify the element of the data. Normally this is zero but a 16 byte string would allow elements 0, 1, 2 and 3. This means that the string may only be a maximum of 255*4 or 1020 bytes.

Also if we make a zero byte hash key a special case this always returns the **ObjectList.** from **Device**. After we have the objects we may then hash for an object and use zeros for the method and selector to get the **IV-List.** from each object. We now have enough to access the whole transducer.

CANed Object Message Format:

Send

Object#	Selector#	Method#	Element	32 bit Data

Receive

Error	Element	Size	Type	32 bit Data

The received message contains the 32 bit value associated with the Element of the data but also gives the Size of the data, its Type and if there was a problem returning the value an Error. The Size information is the number of bytes or elements in the data. The Type gives the information as to the way the data should be displayed. The following table defines the data Types supported:

$00 USHORT	- Unsigned byte integer
$10 UINTEGER	- Unsigned word (2 byte) integer
$20 UDOUBLE	- Unsigned double (4 byte) integer
$30 ADDR	- 2 byte address value
$40 SSHORT	- Signed byte integer
$50 SINTEGER	- Signed word (2 byte) integer
$60 SDOUBLE	- Signed double (4 byte) integer
$70 REAL	- Floating point 32 bit
$72 STAMP	- Date/Time stamp

$80 STRING - ASCII string
    $80 NORMAL string modifiers
    $82 COMMA
    $84 BOLD
    $86 ITALIC
$90 TEDS - IEEE 1451 compatibility
    $90 META-TED
    $92 META-ID-TED
    $94 CHAN-TED
    $96 CHAN-ID-TED
    $9C CAL-TED
$A0 Tables - Lookup Table types
    $A0 FLOW-TABLE
    $A2 LIN-TABLE
$F0 GENERIC-DATA - Undefined data type of any length
    $F2 DATA-PAGES - Stored data page

The signed and unsigned data types are further modified to give fixed point format if necessary. It is often better to supply data in a fixed point integer format rather than floating point to retain accuracy. A signed double integer of type code $62 and integer value of -1234 would be a displayed value of -12.34. Every read from an object returns the Type information so there is no need to request this separately, although this is possible with the **Style:** method which returns both the Size and Type of a selector.

The Compiler

The compiler was implemented with IRTC[2] running under Win32Forth[1]. It creates objects from the classes at compile time. Objects cannot be created at runtime as the class information is only an extension of the Forth compiler. This overcomes the necessity for garbage collection associated with other object implementations like Java. In an embedded sensor the overhead of this and the RAM necessary to implement runtime objects is often commercially not viable.

Below is an example of the Classes for a simple sensor;

```
SELECTOR: ObjectType.
SELECTOR: Address.
SELECTOR: Description.
SELECTOR: Notepad.
SELECTOR: ObjectList.
SELECTOR: IV-List.
SELECTOR: Init.
SELECTOR: Product.
```

```
SELECTOR: Release.
SELECTOR: Version.

CREATE SENSOR-TYPE     START" Sensor"     "END

CREATE SENSOR-IVS
             START" ObjectType.,Read:,Size:,Style:,"
               $,-T Address.,Read:,Write:,Size:,Style:,"
               $,-T Description.,Read:,Write:,Size:,Style:,"
               $,-T Notepad.,Read:,Write:,Size:,Style:,"
               $,-T Product.,Read:,Size:,Style:,"
               $,-T Release.,Read:,Size:,Style:,"
               $,-T Version.,Read:,Size:,Style:,"
               $,-T Init.,Do:,Style:,"
               $,-T IV-List.,Read:,Size:,Style:,"
               $,-T ObjectList.,Read:,Size:,Style:,"
             "END
CREATE OBJECT-LIST     START" 0x0131,"   \ X8+X5+X4+1
                         $,-T Device,"
                         $,-T Temperature,"
                         $,-T Setup,"
                       "END
:CLASS META-OBJECT
      STRING SENSOR-TYPE  ObjectType.  IN-ROM    \ SENSOR TYPE
                  SSHORT  Address.     IN-EEPROM \ NODE
 COMMA STRING OBJECT-LIST ObjectList.  IN-ROM    \ OBJECTS
 COMMA STRING  SENSOR-IVS IV-List.     IN-ROM    \ IVS
                    ADDR  Init.        METHOD    \ INIT
               32 STRING  Description. IN-FLASH  \ DESCRIPTION
              256 STRING  Notepad.     IN-FLASH  \ NOTEPAD
               32 STRING  Product.     IN-FLASH  \ PRODUCT
               16 STRING  Release.     IN-FLASH  \ RELEASE
               16 STRING  Version.     IN-FLASH  \ VERSION
                          SELERROR
;CLASS

\ Define the DEVICE object in the Target

META-OBJECT    DEVICE
```

This is the definition of the META-OBJECT Class and the creation of the DEVICE object in the Target. The other Classes are defined in a similar way to create the objects in the object list. Each has its own **IV-List.** defining the selectors and their methods in the object.

Conclusions

This simple object interface requires little prior knowledge to access and use the sensor data. The data, its value, nomenclature and units may be displayed with only the knowledge of the communication protocol, the Type information and the Units, if the IEEE 1451 scheme is used rather than an ASCII string. Future products may be as simple or as complicated as necessary but if an old sensor is replaced by a new one, with the same measurement quantity name, the system will continue to function correctly.

References

1. Win32Forth for the PC
   Tom Zimmer

2. Interactive Remote Compilation for Development and Machine Integration
   Alan M Robertson – EuroFORMAL ’89

3. CLASS.F in Win32Forth by Andrew McKewan

4. P1451.1 and 2 Draft Standard for a Smart Transducer Interface for Sensors and Actuators.

Information on IRTC is available from:

RAM Technology Systems Ltd
3 Kellaway Road
POOLE, Dorset, BH17 8PD, UK
Fax: +44 870 7065815
www.ram-tech.co.uk