Beate Koye
Engineering pro Time GmbH

"Objectmodul" - an universal application interface for distributed systems

"Objectmodul" is not just another layer 7 for CAN networks. It is a universal SW Interface best fitting in distributed real-time systems and decouples the application from the different ways of transporting data. So the application can be developed independent of the layers (CAL, Devicenet, SDS, CAN Kingdom, ...) of the network (CAN, RS232, Dual port Memory, ...). The existing "Objectmodul" is implemented as a standalone SW module, but it is also possible to integrate it as an extension of an operating system.

By using the "Objectmodul", the applications on various CAN nodes do not need any knowledge about the topology. This means, no knowledge is needed about the I/O is local or distributed by the network. The application just reads or writes objects and the "Objectmodul" controls the handling of the data. So the application program is completely independent of the way the data is transferred, the type of communication partner or data layout.

Embedded Systems often are busy working on data exchange of various kinds: There are serial interfaces to be read, the state of parallel I/O to be set or got, displays to be controlled and much more besides. The modules to be controlled can both be positioned local or wide spread in networks of any type.

The handling of all the communication mechanisms often leads to a high effort concerning the software development:

» The normally extensive communication part is not strictly separated from other SW modules so there is the danger of losing the track of the programming structure.

» Similar but not identical communication procedures within one project are often implemented several times. Besides the higher development effort this leads to susceptibility to errors and quite a long testing phase.

» Owing to lack of time during a project the programmer usually develops his libraries exactly suitable to his problem. It’s obvious that in later projects he has to rewrite it or even has to make it new.

» OS (if any used) and compilers differ from project to project, so system depending parts has to be adapted laboriously.

» Even if parts of the code are reusable - they have to be extracted, tended and documented. This takes up a great deal of time.

On the other hand a module providing the communication can make one's work easier:

» ... a tool which is separated from other modules, so the software is better structured and therefore easier to be developed, serviced, tested, varied, expanded, ...

» ... a tool which has the same uniform interface to all modules but offers a lot of different attributes and for that reason is universal and easy to use.

» ... a tool which can be used in most diversity projects with differing controllers, compilers and operating systems.

» ... a tool which is slim enough to fit in almost every system and fast enough to be a real alternative to conventional solutions.

And precisely here the "Objectmodul" comes into operation. As an intelligent data manager, it offers a uniform interface, no matter what should be controlled.

+++ a Definition +++

In principle the "Objectmodul" is an instance for saving and restoring data sets. The data itself and its attributes are called "objects". Objects can be: The on/off state of a motor, the text of a display, the value of an AD converter, a file to be downloaded, data to be exchanged between software modules, ...

The attributes of an object influence its behaviour at runtime. They are set during installation of an object (this is when an object comes into live) or via function parameters when the object is accessed to. The lifetime of an object begins with its installation and ends with its destallation. That is to say: all the memory space which an object needs is just reserved during the object's presence.

+++ Storing Data +++

For buffering its data, an object needs a connected memory area which is used as a FIFO buffer. When installing an object, the size of an element and the capacity of the FIFO buffer is fixed and the resulting memory area is allocated. In other words, the capacity of the buffer does not change during the lifetime of an object. When an object is no longer in use, its memory can be deallocated by destalling the object.

By the way, some may wonder why the object's capacity is not adapted to its actual contents (dynamically space allocation during object lifetime). This is because the programmer can keep the overview of the memory resources. Thus it never can come to a lack of memory when receiving data can't be processed in time. What is more, it's a preventive of fragmented memory which often causes some memory managers to work incorrectly.

+++ Basic Functionality +++

In simplest case the "Objectmodul" can be used as a normal FIFO buffer. You create an object with the specification of information length (element size) and the number of elements in the buffer. Now you have created the buffer where you can read from and write into. Buffer overflow and underflow will result in an error return.

It may be useful to deactivate the overflow or underflow control. It's possible to configure. This is usually needed for objects that only consists of one element, for example a buffer containing the current date and time. Thereby the buffer can be always updated and read-out when needed. The exception is the control of the first read access: There is no sense in reading-out a buffer with undefined contents.

+++ Relief of Application +++

The purpose of the "Objectmodul" is to take the load off other modules. For example, the application should only be concerned as less as possible with the data transfer to or from the HW drivers.

To divide the communication mechanisms from the application, the "Objectmodul" offers an interface to data collection. This means, the information to be transfered can be split up into pieces. You can access to parts of one object element ("blocks"). For instance this is required to feed the drivers with slices it can work with.

If you need to transfer data via Can which consists of, say, fourteen Bytes you write the information into the buffer at once (this is what the application does). Your Can driver reads out the memory in blocks of up to eight bytes. When coming to the end of the element, the reading instance will be informed about the last valid bytes. Of course likewise it's possible to write the data in blocks which is needed every time you receive data from drivers.

As a result of this, the "Objectmodul" can take over parts of higher protocol layers such as CAL.

+++ Signals +++

The best feature of all is the automatically signalising when one element is complete and this additionally in a way you decide. This is only possible because in this occurrence your own defined functions will be called.

First, you can decide *whether* and *how many* actions have to be done. Just install your defined functions consisting of a pointer to your function and one parameter of the function. This can be done at any time after you have installed the concerning object. Of course, you can destall single functions whenever you want.

Every object can contain several functions. And every function can be configured individual. The interesting point is that diverse objects can use the same functions, but the attributes of equal functions can vary.

Second, it's your decision *in which case* the functions have to be called. Suppose you have a device which sends its status periodically. One Function watches whether the status arrives in time. The other one only becomes active when the status changes i.e. an error is reported. Each function can be told whether it should be called either information has been *received* or *changed*.

Third, you can decide *when* a function shall be called. They can be called either *automatically* when the element has become complete, or *manual* by function call. The last case means that some functions can be prevented from executing. Becomes a data array ready, the concerning functions will just be marked. Next time when you tell the object a manual function call, this marked functions will be executed one by another.

+++ Varying Data Length +++

At the beginning we said that an object, when installed, contains of a constant number of elements with a constant element size. But sometimes you are concerned with data which lengths vary.

A case in point: one module in the network has to control a display contending 25 columns. The text which has to be displayed is provided by the net. Because it is really not useful to transfer 25 bytes when the real information is less, you have the opportunity of manually finishing object writing when you detect the end of the transferred data. Then the behaviour of the "Objectmodul" then is like the element had been completed. To use the received data correctly, both the receiving and the reading instance must know the end syntax of your data. In this case this might be a terminating null character.

+++ Aborting +++

Reverse, you can also cancel the write process. Imagine, you get data in slices and suddenly notice the data is not valid. Then you can throw away the entries of the actual element you have done so far.

+++ System Independence +++

After having had a view on the functionality we now should examine how the "Objectmodul" can be used in different systems.

The code is written in ANSI-C. However, data types which are system depending ("int") were not used. Instead of this the types are defined separately. By this, the code is reusable with only adapting one header file. Furthermore the code was developed so robust that compiler which do not work 100 per cent ANSI-C compatible should not have any problems with this code.

Apart from this, the interface to the OS may be adapted. More exactly, you need one semaphore when a multitasking system is used, otherwise the concerning functions remain empty. If needed, memory access can also be modified.

To sum up, adaptation is very simple and almost without any effort.

+++ Performance +++

In spite of the wide variety of functionality, the "Objectmodul" hardly needs additional space. In opposite to the separate development of all communication mechanisms it even needs less space.

The second important point is the operation speed. When you make an access to an object the module knows its buffer position without any search operation. So you have a quick data access, no matter how many objects are in use.

+++ Summary +++

» Handles communications of every kind smart and quick

» Access data amounts in a uniform manner

» Controls buffer access

» Generates and disposes buffers dynamically

» Accesses to data synchronous or asynchronous

» Enables data splitting

» Signalises complete data sets through user function calls in a user-defined manner

» Enables system adapting easily

» Written in ANSI-C

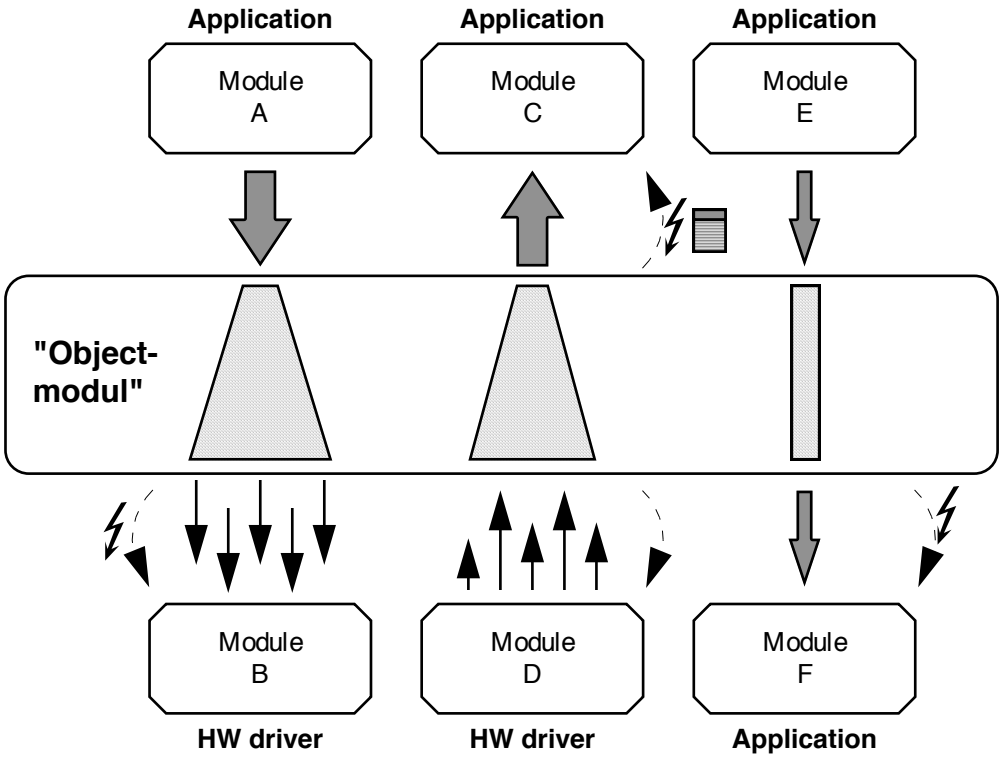

Fig. 1: Data flow -- Examples for Communication processes