New version of CMD object (Commands)

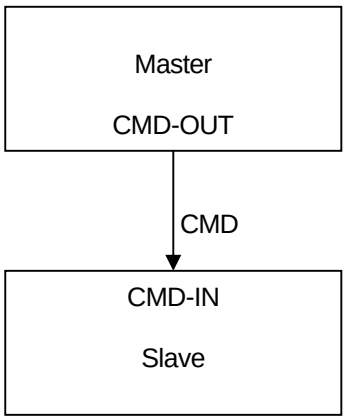
State machine Interface

In [1] we discussed the interface between state machines. The StateWORKS design philosophy recommends the Master – Slave principle as the basic relationship between state machines, where Masters sends commands to Slaves and use the states of Slaves as inputs.

A command is coded as a number but is given a name to make it more comprehensible. For instance: 1 means Cmd_Stop, 2 – Cmd_Start, 3 – Cmd Break, 4 – CmdContinue, etc.

CMD object

To serve as an interface between state machines a command must have two aspects: it is an input for Slave and an output for Master. The StateWORKS data base contains object CMD with this feature. It is not the only object with that feature, for instance a timer state is used as an input (mainly OVER) but on the other hand it is an output (start, stop, etc.) for the state machine which uses it. Similar properties are found in counters or swip. The main difference is that the CMD object is to be used as an interface between two state machines and the timer should normally be used by a single state machine for timeout, watchdog or similar time functions. To express these aspects of the CMD object and to make it more comfortable to work with, it appears as a CMD-IN object in a Slave state machine (its owner) and as a CMD-OUT object in a Master state machine (its user).

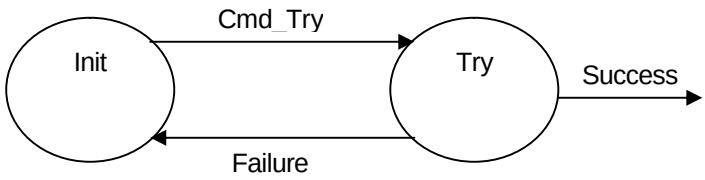


In principle, the CMD object (CMD-IN) in its owner (Slave state machine) should be used to define input names used for behavior specification, i.e. it should be visible in the Input Name Dictionary. On the other hand, the CMD object (CMD-OUT) in its user (Master state machine) should be used to define output names for action specification, i.e. it should be visible in the Output Name Dictionary. The StateWORKS implementation also allows the usage of the CMD-IN in the Output Name Dictionary; the reason for this is explained below.

Command life time

A state machine is triggered by events which cause some actions and / or state transitions. The event is a change of an input signal. The actions and transitions are specified by conditions which use input signal values. The input signal values have different life characteristics. Some signals are more like an event that is they appear and are “consumed” after usage. Other signals have a static nature, they cannot disappear, and they exist always. For instance, a digital input if *true* will not disappear after the value has been used, although it will possibly change to *false* at some later time.

A command raises difficulties when considering its life time. There are no definite rules which say when the command is "consumed". In many cases, we may assume that a command is valid until it is replaced by another one. Unfortunately, this rule is not always true. Imagine the situation as in the diagram below. A system is stuck in the state *Init* and can repeat some activity going to the state *Try*. It goes to the state *Try* on receiving the command *Cmd_Try*. Later, it leaves the state *Try* returning to the state *Init* if a signal *Failure* is true or continuing to some other state if a signal *Success* is true. In the former case, if the previous command *Cmd_Try* is still valid the state machine goes immediately from the state *Init* to the state *Try*. It is a typical example where the command must be in some way consumed if it is to control each transition from the state *Init* to the state *Try*.



The question is - when and how to consume the command.

There is no obvious solution for this dilemma. Let's discuss a few alternatives:

- To treat the command as a true event, i.e. to consume it after usage. This solution is unacceptable because in many cases we need to maintain a command during several state changes. In other words, the command very often requires different treatment in the same state machine: during some transitions it must be kept until it is replaced by another value; in other situations it must be consumed immediately after usage. The problem of "consuming" or "deleting" a command is also not obvious: a command is a number that always has a value. "Consuming" means that the command gets a value, conventionally zero, which is not used for definition of input names.

- To replace the command with another command if the present value is considered as consumed. This solution means that in a design we would always have to implement a true hand-shaking: the Master sends a command - the command causes something in the Slave - the Slave signals the effect to the Master – the Master sends another command. This principle sounds nice but is often too heavy: it leads to excessively complex solutions to simple problems.

- To delete the input name corresponding to the command value in the virtual environment. In fact, this solution has been applied in StateWORKS for many years by using the C(lear) field as a kind of entry action. Of course, it is not an entry action (therefore it uses a separate field) but just an indication for the Executor (in the run-time system) to remove the name from the virtual input. Of course, we can use the C(lear) field for removing any names from the virtual input but effectively, it has been used only for commands. In fact, it would not be correct to use it for most other signals. Imagine removing a name corresponding to a digital input: it would be just a fraud because a digital input cannot be consumed – it has always a value. We have used this solution but we were not happy with it, as after removing the name from the virtual name the real signal (command) is not correctly represented in the virtual input. Another consequence of this solution is the problem of command repetitions. The StateWORKS data base is a real time data base which generates events if an object changes its value. Removing a command name from the virtual input does not change the real command. Therefore, the command object has received a special treatment – it was the only object that generated events while maintaining the same value.

- To delete the command, or more precisely to set it to a value (0) that is not used for input name definition. This solution is the best one. It gives the designer the possibility to decide about the command life period and it no longer requires any special treatment of the

command object. The correspondence between the virtual input and the real signals is guaranteed.

The last solution - deleting the command - has been implemented in the latest version of the StateWORKS development system and the run-time system. It explains why the CMD-IN now appears also in the Output Name dictionary: we can define there a (true) output action used for deleting the command value.

Several command objects for one state machine

Normally, a state machine needs only one command object (CMD-IN). This object is used to define all commands required for state machine specification. Sometimes, a state machine requires more command objects. The reason may be a requirement to have a group of commands for testing that we do not want to mix, for safety reasons, with the operational commands. Another reason could be the use of a second command as a parameter for the main command.

Though commands are numbers, they are "known" in the state machine design by names. The names are defined in the state machine IOD – file as strings used by the run-time system (they are also defined as enumerations in the state machine H – file for any programming purpose). If a state machine has more commands, for each additional command a pair of IOD - and H – files is generated with the command names. The name of the files is created by concatenation of the state machine name and the command name. The name of the IOD - file (without extension) should be used as a **Type** property in the **Cmd Properties** window.

Example

The state machine discussed above (with *Init* and *Try* states) will have the state transition tables shown below.

The state machine has a CMD object **MyCmd** and a XDA object **Result** to control the trials. The **MyCmd** object is used by its Master to trigger some activities in this state machine. The **Result** object is used for acknowledgement of some actions for instance by a client. Analyzing the requirement we find out that both signals must be consumed after they have caused the required transitions. If the state machine receives in the state *Init* the command *Cmd_Try* it goes to the state *Try* where it does some action, clears the **MyCmd** and waits for an acknowledgement. If the acknowledgement is *Result_Failure* the state machine returns to the state *Idle* and the trial will be repeated after receiving again the command *Cmd_Try*. Note that the **Result** signal must be cleared also on entering the state *Idle*. If we forget to clear either the **MyCmd** in the state *Try* or the **Result** object in the state *Idle* their value will stay and cause false transitions on entering the state next time. This is a typical example where signals must be consumed immediately after they are used to provoke a transition: they are losing their meaning at this instant.

If the acknowledgement is *Result_Success* the state machine goes to the state *Continue* and the trials are terminated.

Maybe, you could ask why we should not use values *Failure* or *Success* of the **MyCmd** as an acknowledgement; thus solving both "consuming" problems: producing *Cmd_Failure* or *Cmd_Success* will replace *Cmd_Try* in the *Try* state and *Cmd_Try* will replace *Cmd_Failure* in the *Init* state. This would be against Master – Slave principle: **MyCmd** belongs to this state machine and only one Master should have an access to this object. A Master should never acknowledge itself the result of the operation. The acknowledgment must come from an external device.

This example also shows the close similarities between CMD and XDA objects. The fine differences between them decide which should be used in a given situation.

Init	E:	Result_Clear
	X:	
Try	Cmd_Try	

Try	E:	SomeAction MyCmd_Clear
	X:	
Init	Result_Failure	
Continue	Result_Success	

What about XDA ?

The XDA object is similar to the CMD but it was never intended to be used for commands. The XDA object is a store for a number (as an integer). In fact, this feature is a side effect, as the primary function of the XDA object is to manage a memory space. This basic XDA feature is used in Output Functions.

Anyway, sometimes we have used the XDA feature to store a number, replicating the CMD functions. This could be justified before the CMD-IN got the **Clear** output value as the XDA object could be used as input and output in a state machine. Hence, we had no problem to determine the life time of the XDA value. After introducing the **Clear** output value for a CMD object there are no longer any valid reasons to use the XDA object as a CMD object. The XDA object has some disadvantages in comparison with the CMD object, especially that the XDA object values cannot be “known” in the RTDB system by names. Thus, we could operate only with numbers when accessing the object from outside.

In addition to the memory management function mentioned above, the XDA object can be used for handshaking between the RTDB and IO unit. We may discuss this topic in a separate note.

In the Cmd example the XDA object has been used to acknowledge the result of an action. It is a typical application for the XDA object: the Result object is not an interface between state machines but it is an acknowledgement from somewhere else.

Running the example

When you install the StateWORKS Studio you will find the entire project in the folder ..\Project\Examples-Web\CMD-Object. You may run the SWLab with Cmd_Example and monitor the system using SWMon. To try out the system is not very exciting: using SWMon you

just send a command Cmd_Try and acknowledge the result using the Result signal to see that it works. Studying the specification in SWEdit is probably as convincing as looking at this trivial exercise.

Summary

The CMD object is very important for the realization of the interface between state machines. The introduction of the **Clear** output is the ultimate solution for proper handling of the life time of a command value.

References

[1] Technical Note: Hierarchical system of state machines, May 2003,