Hierarchical systems of state machines

System of state machines

A single state machine can control a relatively simple application task, for instance a motor. The limit is the size (number of states) which a designer can effectively develop and understand. There is no fixed maximum number of states that a manageable state machine should not exceed. The maximum number of states depends on several factors, like for instance: the chosen state machine model or complexity of the application. In practice, the maximum number of states is between 30 and 100.

Normally, applications have several tasks to control. In practice, the tasks are not controlled independently – they form a system in which the parts interact. Thus, to master such an application we partition the control among several state machines. We need to know how to organize such a system. Should it be totally free system of loosely coupled state machines or should it have a strict organization? Though there is no ultimate answer to this question we know from experience that a complex system requires an organization. A lesson from software development is that trivial problems are not problems – any solution will do, even spaghetti code - but the difficulties encountered in development of complex systems cannot be solved by ad-hoc solutions. They require a well thought-out approach that allows the entire system be partitioned into several subsystems which communicate smoothly among themselves. This consideration is the basis of the StateWORKS approach to the problem, which recommends a hierarchical system of state machines.

The discussion of a (hierarchical) system of state machines should cover several topics: the theoretical model, communication among state machines, design procedure. In this note we will focus on the two last questions leaving the first one for more academic discussion.

Master/Slave interface

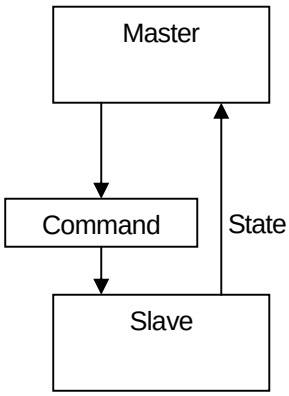
The basis of a system of state machines is the communication among them. In the literature, we find many software structure oriented discussions which concentrate mostly on event driven systems or message passing mechanisms. Using StateWORKS we do not need to care about this level of implementation; we may discuss the problem on a purely logical level – how to organize the interface among state machines.

The question is: what kind of information should state machines exchange? If state machines exchange data, than they would require a message as the information carrier. The data in the message body would contain also control information. The VFSM concept of StateWORKS is based on a total separation between data and control flow, which means that there is no use for data exchange among state machines. State machines exchange only control information.

We recommend a Command (CMD object) / State (VFSM object) interface between state machines. A relation between two state machines is a Master – Slave one: the Master sends a command to the Slave and uses the Slave's states as inputs (see the following diagram). The command is just a code number which defines a request for a state machine, such as: *Start*, *Stop*, *Go*, etc. (the command names correspond to command numbers).

The realization of the Master/Slave interface requires an additional object[1] which passes the command from Master to Slave. One state machine (the Master) sets the value of the command as an output action; the destination state machine (the Slave) uses this value to define input names used for control. There are very few objects that could be used as interface elements, the most obvious being CMD and XDA (a general-purpose read/write data object).

[1] The StateWORKS implementation of the VFSM concept defines a family of objects (organized in a real time data base) where control properties are used to specify the behavior. An object can be used as an interface element if it is both: an output and an input object.



It is reasonably clear why a state of a Slave state machine should be used as an input for the Master state machine: the state represents the full control information about the situation in the Slave. The question of whether to use a state of Master in a Slave must be answered negatively: we should not do it. First of all, it would not be a Master-Slave relation anymore. The second more important reason is that in using commands the Master is very flexible in definition of requests: it may use the same command in different states or it may generate several commands in the same state. The Master – Slave relation means that Master commands the Slave to do something and supervises the Slave by watching its state: if the Slave reaches a certain state the Master can assume that its command has been carried out. This definition is based on the principle that a state of the state machine represents the entire control situation as covered by this particular state machine.

What about other objects? Theoretically, we can use any object with input and output functions as an interface among state machines. For instance, we can start a timer in one state machine and use the timer status OVER in another state machine as an input signal. This kind of interface may sometimes be needed but it should not be overused. As an exception it can be tolerated but we should not build state machine systems with this kind of interface as a rule. Eventually, we want to understand the system behavior – the more standard is the interface the easier it is to understand. Any special tricks are liable to introduce chaos in software[2]. The only object that could be treated as an alternative to the CMD object is the XDA object. The value of this solution is discussed in the next section.

CMD or XDA

In StateWORKS the CMD object type should be used as a primary interface between two state machines: Master sends commands to Slave(s). An XDA object is similar to CMD: it is a number which can be set as an output action and used to define input signals (names), i.e. one state machine (Master) can set the number and the other state machine (Slave) can use it as an input signal. There are also differences between these two object types.

The CMD object has been intended as an interface object and has several features which make its usage comfortable:

- it can be given different type names in Slave (CMD-IN) and Master (CMD-OUT),

[2] It is a general software problem: most of us agree that in programming we should follow some rules, use methods, specify (or at least understand) the problem before we start programming, and so on. The theory is full of such golden rules and recommendations. Unfortunately, reality is not so tidy, and very often catastrophic. The main reason is that when programming very often we want to solve a small difficulty in a code by by-passing the rules, using some dirty trick. In this very moment we think that this one deviation from some rules, standards etc cannot do any harm. The consequence is well known: excessive use of such tricks leads to poor software quality, software which is difficult to understand and manage and sometimes even unacceptable or unusable.

- the Slave CMD input names are accessible (displayed) while designing the Master state machine,

- the Slave CMD input names are accessible in the execution environment and can be displayed, for instance on the operator panel.

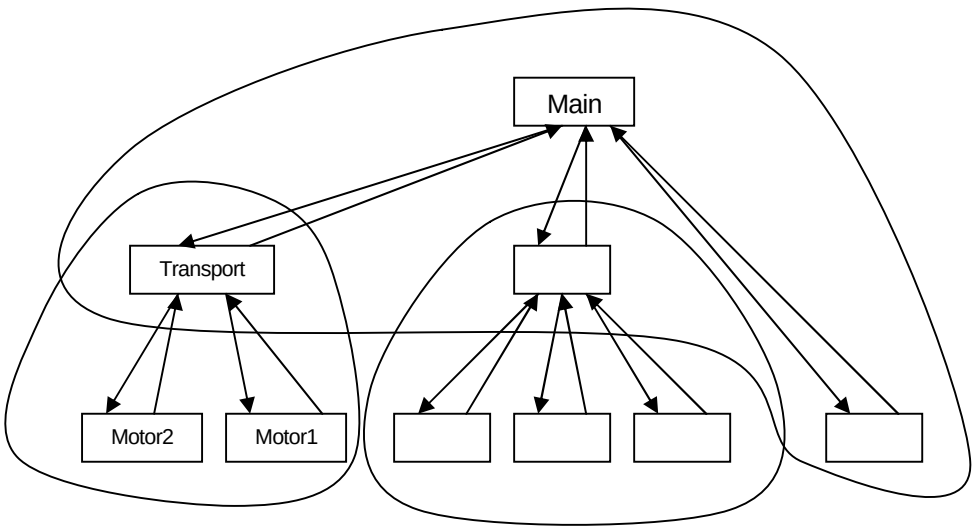
The XDA object is very simple and was essentially provided for handshaking between state machines and I/O units (actions that cause some changes in the outside world via the I/O unit must be acknowledged in the I/O unit). The XDA object has one interesting feature, namely it can be set in any state machine; it is “symmetrical” and not “uni-directional” as is a CMD object. In some situations this feature could be interesting. The consequences of the XDA usage and a more detailed comparison with a CMD object will be discussed in a future note.

Design procedure

Of course, the Master/Slave interface between state machines does not dictate a hierarchical structure; it can be used in any system of state machines for exchange of control signals.

The recommendation to use a hierarchy of state machines is based on experience and some suggestions of a theoretical nature. When designing state machines we can make logical errors which result for instance in infinite loops or deadlocks. The more state machines are in the system, the higher is the probability of such errors and finding of those errors is more difficult. Imagine a system of 100 state machines where each state machine may send a command to any other state machine and uses states of any state machine for definition of its behavior. It is very difficult to understand how such a system truly behaves and a structure of such a system corresponds to non-structured software.

In contrast to a non-structured system a hierarchical system is easier to design and understand as in its design or debugging we usually solve several local problems: Master and its Slaves (see the following diagram).



The diagram shows a hierarchical system with 3 levels: the upper level containing a single Master state machine (*Main*), the second level containing two state machines and the third, lowest level with 6 state machines. A design of any state machine requires always an analysis of a limited part of the system. A design of the *Motor1* state machine covers the outside signals and commands from the *Transport* state machine. While designing the *Transport* state machine we are interested only in the states of state machines *Motor1* and *Motor2* and commands from the *Main* state machine. While designing the *Main* state machine we take into account only states of its Slaves (*Transport*, etc.). In a correctly designed hierarchical system any state machine “sees” only the states of its slaves which represent an abstracted but all the same

complete set of the control signals which are relevant for the state machine. The only state machines that “communicate” with the outside signals are the *Main* state machine and the state machines on the lowest hierarchy level.

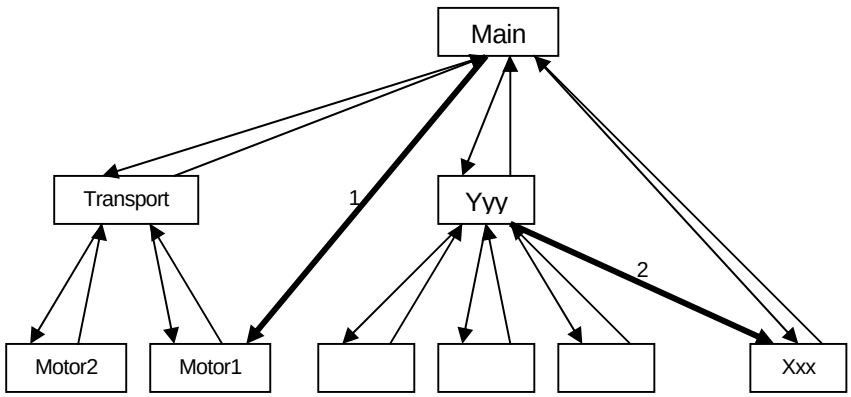
A hierarchical system allows us to think locally in designing state machines, seeing only the few involved state machines (Master and its Slaves) in an abstract way. For instance, the *Transport* state machine has no idea about the details of the Motor control (inputs, outputs, delays, timeouts, parameters, etc.), it sees only the (abstract) situation in motor control, represented by the *Motor* state.

The other problem is how we should proceed in the design of a state machine system. Also in this case there is no strict rule but it is obvious that we have to start where the information is more or less complete. As a rule we have relatively good information about the behavior required of the state machines on the lowest level: how to control peripheral devices. We probably also know about commands for the Main state machine but unfortunately we do not know much about its Slaves. Therefore, in most cases the only reasonable way is to start with the lowest level of state machines. Having this layer ready we get some idea how to organize the Master layer above it. So, we continue until we reach the Main state machine.

The next question is the design of the hierarchy: how many layers and which state machines. There are some fixed points, for instance we need a Main state machine at the top and we know very soon which state machines are required at the lowest level - the lowest level is well defined by the controlled devices. The definition of the rest is a process that is strongly bound to progress in designing the lowest level of state machines. Building a hierarchy is an evolutionary process with several trials and it does not have an ultimate solution. The solution reflects the designer's ability and preferences: some people like to use a few, rather complex state machines, other people prefer many simpler state machines in an elaborate hierarchy. There is no definite answer as to which approach is a better one, as long as the designer can be quite certain that he fully comprehends the way in which each state machine will function, in both normal and abnormal situations.

Sins

StateWORKS does not impose any barriers considering the structure of the system. So, we may design any system, for instance having in principle a hierarchical system with additional links between some state machines. How much deviation from a hierarchical system could be tolerated? The following diagram shows two examples of sins committed by a system designer.



The wrong links 1 and 2 are in bold. In case 1 a true Master (*Transport*) of *Motor1* state machine is by-passed, receiving commands from the *Main* state machine. It is very difficult to understand the behavior of such a system. The probability of malfunctioning is very high as the true Master of the state machine does not “know" about the additional command. The additional commands are in fact a kind of unexpected input that must be “corrected” by the Master. On the other hand, if we consider the influence of these additional commands in design of a Master

state machine we find no reason for them: those commands could be passed directly to the true Master. It is difficult to design a state machine like *Xxx*, which gets commands from *Main* and also from *Yyy*. What kind of design philosophy to follow?

There are many possibilities to corrupt the hierarchical system, similar to any software corruption practice. We should avoid them as in general there is no real need for them. It is better to put in some additional design effort and make a correct design, avoiding any non-hierarchical deviations in the system, instead of succumbing to the illusion that doing it quickly but wrongly will save time.

Example

We have put on our Web site a case study “A hierarchical system of state machines”. To make it simple, as required for easy comprehension, we limited the system to a Master state machine and four Slaves. It illustrates the ideas formulated in this note and demonstrates some details of a StateWORKS implementation.

Summary

Using StateWORKS we recommend, for building a system of state machines:

- to use Master/Slave (command/state) interfaces for communication between state machines

- to organize the system as a hierarchy of state machines based on a Master – Slave principle

- to use bottom – up design

- to avoid corruption of the hierarchy by “wild” links among state machines.

Those of you with over 20 years experience of software design will see a very clear parallel between these rules and those of "structured programming".

Using these rules we are able to build reliable and maintainable control systems with a high degree of reusability. Well-designed state machines can be stored in a library and used in future projects. The reuse of state machines applies especially for state machines on the lowest level which control peripheral devices.