DEBUGGING STATE MACHINES

Debugging

Any non-trivial software must be tested. In testing, errors are being detected and corrected – this process is known in software development as debugging. Program may have two kinds of error: coding and application failures. Obvious coding errors manifest themselves by crashing the program or other programs and in the worst case – the operating system. Several not so obvious coding errors result in misbehaviour of the application "simulating" in a way logical errors. A notorious example of such a failure is the unfortunate comparison operator (= =) used in C /C++ which often stays undetected by a compiler if confused with the assignment operator (=). I would like to have money corresponding to the time invested for searching this kind of simple error. Application errors mean that the program does not realize the requirements. Hence, the reasons for application errors are very often not clear – they can be logical errors or coding failures.

Classical debugging is done in a program which means that we are trying to find the erroneous code and correct it. Such debugging is not easy as the two aspects: code and application errors are overlaid.

In StateWORKS debugging is clearer – in principle we are looking for application errors. The probability of a coding error is extremely low as the application is built using a very robust, standard code (RTDB) which has been used and tested over many years in a variety of applications. Of course, nobody can guarantee 100% reliable code but if the failure is really in the RTDB it must be corrected by the manufacturer of StateWORKS.

Thus, for the user, debugging of the RTDB based application means testing whether the system of state machines works properly. Testing of a state machine may be tricky. To facilitate testing a few things must be available such as: a trace facility, a debugging mode, an automation of test sequences and a service mode. Good, up-to-date documentation has to round up the development environment.

System Consistency

Debugging of the RTDB application begins with a start-up during which a SULOG.TXT file is produced. This file is a log file with a list of objects that could not be built due to some inconsistencies in the system specification file SWD. The inconsistencies are allowed on purpose in StateWORKS to test not-quite-ready applications. Of course, the ultimate application should produce an empty SULOG file.

```
VFSM System Startup Log
-----------------------

Config File:  G:\StateWORKS\Projects\Test\SurfaceView3\SWSystem\Spec\SWSystem.swd
Startup Time: 14-Jan-04 09:29:18

W1   4   Not found enumeration in/or IOD-File: joystickdigital       X:JoystickDigital:Cmd
W4   27  String Resource not found:  IDS_AL_START_ERROR              Axes:Al:Motor_Start_Problem
W135 8   Not found User Defined OutFunction OfuCheckDeviceNumber     Allsens:OFun:CheckDeviceNum

There are:
   3 Warning(s)
```

Text 1 Example of a SULOG.TXT file

An example of a SULOG.TX file is shown in Text 1. In a rather cryptic form it contains 3 warnings signalling missing:

- Cmd names list
- Missing string for an alarm text
- Missing output function.

The number in the second column (4, 27, 8) is an error / warning type – in all there are 30 types of them.

Trace

Trace means the availability to record all or selected steps performed by a program. All RTDB objects have the trace facility which may be activated or disabled. Normally, the trace facility is disabled. If activated the trace facility causes all changes of object states (and for some objects also their data values) be logged into a TRACE.TXT file. Trace can be activated from Monitors by setting the Trace flag of an object to *true* (*1*). Any number of objects may be traced simultaneously. In addition, the TRACE.TXT file can be closed and opened again at any time from Monitors. After re-opening the TRACE file is always empty. An example of a trace is shown in Text 2. A line in a trace file shows:

- the time of the event
- the object type
- the object name
- the numerical value of the object state
- the name of the object state (if available)
- the data value (if appropriate).

For instance, at 21:57:29 the object NO with the name Pressure1:No:SetPressure changed its state to SET (5) because its value had been set to 900. A zero value for a CMD object means that no command is active: other values will usually have names assigned.

```
VFSM System Trace File
----------------------
(Date: 18-Oct-04 21:57:22)
21:57:29 CMD      Main:Cmd                                0
21:57:29 CMD      Pressure1:MyCmd                         1     Cmd_Start
21:57:29 VFSM     Pressure:01                             3     Starting
21:57:29 CMD      Pressure1:MyCmd                         0
21:57:29 TI       Pressure1:Ti:Timer                      3     RUN
21:57:29 NO       Pressure1:No:SetPressure                3     CHANGED   90(
21:57:29 NO       Pressure1:No:SetPressure                5     SET       90(
21:57:29 OFUN     Pressure1:Ofun:ActualPressure_CalcLimit  1
21:57:35 SWIP     Pressure1:Swip:ActualPressure_Supervison 3     IN
21:57:35 TI       Pressure1:Ti:Timer                      2     STOP
21:57:35 VFSM     Pressure:01                             4     Regulating
```

The trace illustrates well the events in the state machine Pressure1: it received the Cmd_Start and went into the state Starting (3) where the following entry actions were performed: the Cmd was cancelled, the Timer started, the Numerical Output set to 900 (two entries because NO data was changed and set as output) and the Output Function was called to calculate the switchpoint limits – the operation was successful which was signalled by the return value 1. 6 seconds later SWIP object had signalled with its IN state that the input pressure was within

limits which forced the state machine to go to the state Regulating. Before it left the state Starting it had stopped the Timer.

The trace “description” corresponds of course exactly to the state machine specification.

Debugging mode (VFSM)

Probably the most difficult requirement is to test state machines “slowly”, step by step. A state machine or even worse a system of state machines may have loops. A loop in a state machine means that a state machine performs several state changes and / or input actions in response to a single stimulus (condition change). The changes are too fast to be noticed by a human being. For a coded implementation we could debug the code using a debugger step mode. A similar possibility is available in the StateWORKS development environment.

A step in the StateWORKS execution environment means to perform one state transition (with appropriate actions) and / or those input actions specified in a present state. The use of the step mode is especially easy in SWMon where a state machine has 3 radio buttons to control the step mode: Run, Hold and Step. Of course, we can control the step mode from other monitors by setting the Rmo (Run Mode) and NSt (Next Step) attributes.

If the Run button is chosen (default) as in Figure 1 the state machine runs and the Next box to the right to the radio buttons displays “-/none”: no input actions and no transition.

Figure 1 SWMon: VFSM Pressure1 is in the Run mode

Marking the Hold button as in Figure 2 stops the execution of the VFSM Executor but the next state is displayed in the frame right to the buttons. In the example below the next state is *Starting*.

Figure 2 SWMon: VFSM Pressure is in the Hold mode

Clicking on the Step button forces the VFSM executor to perform one step – in the example the transition to the state *Starting* (Figure 3 shows the SWMon display after the step). The system returns to the Hold mode and waits for the next step. In the example the next step means a transition to the state *Regulating*. (Note that the actual state and some previous states are shown by a right-wards scrolling of the right-most field.)

Figure 3 SWMon: VFSM Pressure1 has executed one step

If we wait until the timer expires, the Next box changes and signals now that there are input actions and a transition to the state Idle to be done (Figure 4). We see that all changes of the inputs are observed in the Hold mode – the next actions or transition corresponds always to the actual input situation.

Figure 5 SWMon: VFSM Pressure1 has executed one step and the timer expired

After the second step the Alarm is generated (it was the announced input action) and the Pressure1 state machine goes to the state Idle. There are no actions or transition due in this moment.

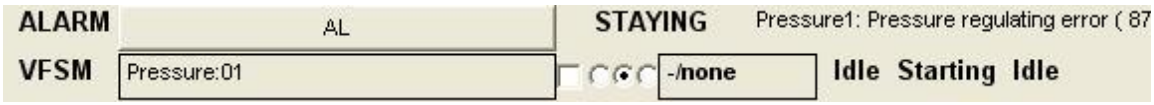

Figure 4 SWMon: VFSM Pressure1 has executed the second step

If there are no input actions or transition due and we click on the Step button the button gets marked and the VFSM Executor will perform the actions or transition if the corresponding condition becomes true.

Command files

Testing of the application is normally a long process. Using StateWORKS Studio we are testing from very early in the project. Testing means a repetition of different scenarios. This procedure is tiresome and requires automation, by means of command files, as shown in Figure 6.

```
sw c
sw n IL.
sw o CMD
sw s Main:Cmd.PeV 1
sw g Main.StN
sw g Pressure1:Ni:ActualPressure
sw s Main:Cmd.PeV 2
sw g Main.StN
sw d
```

Figure 6 SWTerm command file

The meanings of the commands – used here in their shortened forms - are:

- sw c — Connect to RTDB
- sw n IL. — Get the path of the (application) SWD file
- sw o CMD — Display names of CMD objects
- sw s Main:Cmd.PeV 1 — Set the 1 value (Cmd_Start) to Main:Cmd.PeV
- sw g Main.StN — Get the state name of the Main state machine
- sw g Pressure1:Ni:ActualPressure — Get the value of Pressure1:Ni:ActualPressure input
- sw s Main:Cmd.PeV 2 — Set the 2 value (Cmd_Stop) to Main:Cmd.PeV
- sw d — Disconnect

The SWTerm monitor can be used for generating automated test sequences. While running a series of commands SWTerm generates a log file SWTerm.log which stores all typed sequences. An example is shown in Figure 7. Renaming the log file to SWTerm.cmd (or some other Name.cmd of course) changes it to a command file. Lines not starting with “sw “ are ignored by the command interpreter. Of course, the command file can be also prepared and

edited in any text editor. Starting the SWTerm monitor with the -cSWTerm.cmd argument instructs the program to open the command file and issue the commands in that file line by line. Each command line has to be validated by the user, by means of the “enter” key.

Executing the command file as in the example in Figure 6 will produce the command and answers as shown in Figure 7. Note that they look exactly as though the commands had been entered interactively from a keyboard, which would often be the case for a first attempt to create a command file interactively.

```
Taking commands from a command file SWTerm.cmd
to continue press Enter
q to quit program
h to display help
v to display the version


Enter command: sw c
Using default Host address(LOCALHOST) and Port number(9091)
CONNECTED

Enter command: sw n IL.
G:\StateWORKS\Projects\Examples\Pumps\Conf\Pumps.swd

Enter command: sw o CMD
CX1000:Cmd:01
Device:MyCmd
Main:Cmd
Pressure1:MyCmd
Pressure2:MyCmd


Enter command: sw s Main:Cmd.PeV 1
Value set

Enter command: sw g Main.StN
StN = On

Enter command: sw g Pressure1:Ni:ActualPressure
Dat = 88.2548

Enter command: sw s Main:Cmd.PeV 2
Value set

Enter command: sw g Main.StN
StN = Idle

Enter command: sw d
DISCONNECTED

Enter command:
```

Figure 7 The SWTerm monitor executing the command file from Figure 6

Service mode

While testing we like to have a possibility for simulating values which influence a control. This feature called Service or Force mode is especially important for external input signals. StateWORKS provides a Service mode for all external I/O objects: DI, DO, NI (represented by SWIP) and NO. Additionally the internal I/O objects CMD and VFSM can also run in service mode to allow us to debug the Master – Slave interface.

Figure 8 shows details of a service mode for DI – it allows us to test the system without its hardware digital inputs. The default mode is the Auto mode which means the DI values are coming from the I/O Unit, effectively from the hardware. The Peripheral Value (PeV) comes from the hardware via the IO-Unit and is passed in Auto mode (SvM = *false*) to Val stored in RTDB. In Service mode (SvM = *true*) Val gets the service value SvV set in Monitor.

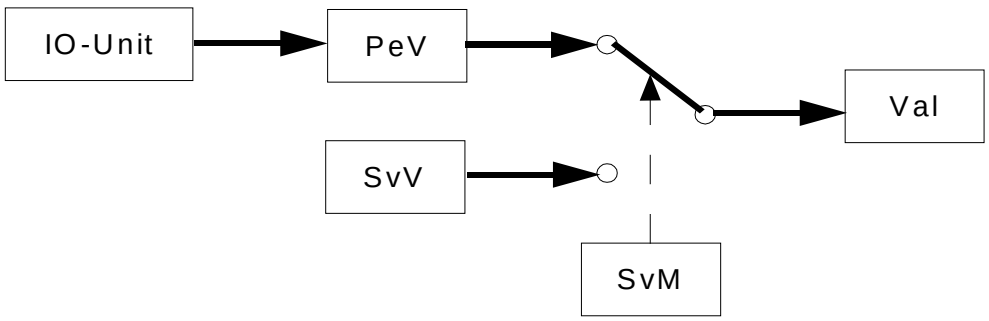

Figure 8 Service mode for DI object

The service mode for other objects functions similarly.

The role of documentation

The software documentation is the only information source for several groups within the company producing the product:

- to plan the testing requires a good overview about the software,
- the product test team must create the test plans,
- customer documentation must be written
- product support and maintenance needs detailed information and
- further product development cannot count on information stored as code.

Because of this a complete and up-to-date software documentation shall be a key requirement for all software manufacturers. The creation of this documentation has to start as soon as possible to ensure that all other tasks like test plans or customer documentation will be completed in good time. During debugging several aspects of a design will change: state machines, the object properties, the command files. These changes influence other software documents produced by cooperating teams.

The entire content of the StateWORKS application: the configuration of a system of state machines, state transition diagrams, state transition tables are available as JPG or WMF files. All information is also produced in XML documents.

As in StateWORKS any change in state machine behaviour can be done only “officially” in the StateWORKS Studio the documentation is always up to date. There is no way to corrupt the documentation as it is the inherent part of the development system, reflecting always the actual

state of the application being developed. To sum it up, in StateWORKS the distance to the updated documentation is very short - just one click.

Conclusions

The note shows all aspects of debugging as implemented in StateWORKS. These debugging features accelerate the development and allow us to call the StateWORKS specification an *executable specification* whose characteristics are taken over directly to the final implementation.