Dr. Hartmut Schorrig, www.vishia.org

EventQueue Lösungen für Embedded Control

Eine Erfahrungssammlung, Schlussfolgerungen und darauf aufbauender Vorschlag zur Gestaltung von Event Queues für Statemaschinen

Inhaltsverzeichnis

1. Die Erfahrungen

A) Rhapsody-Einsatz in einem großen Team vor ca, 15 Jahren, Version bis 7.3:

* Es wurde im gesamten Team viel mit Statecharts und Codegenerierung gearbeitet.
* Basierend auf einem firmeninternen, aber umfangreichen Realtime-OS
* eigene Threads für die Statemachines
* Events mit new angelegt, „*so wie es Rhapsody macht*“.

Es gab mal einen Hänger, weil die Quelle fortwährend Events generiert hat, der Thread für die Entnahme aus der Queue aber aus irgendeinem Grund hakte. Speicher voll, kein debug mehr möglich.

Insgesamt hat das alles funktioniert.

B) Kenntnis von Statecharts in Rhapsody, des Prinzips seit über 20 Jahren (1998), Danach Statemaschinen in C programmiert aber nach Harel-Prinzip und festem Pattern, kommerzieller Einsatz in Grafischer Funktionsblockprogrammierung.

C) Event-Queue im Java gebaut für Inter-Thread-Kommunikation, mit einfacher eigener Statemachine-Lösung, im Grafik-GUI-Bereich

D) Software für schnelle Embedded Control, (Zyklus ca. 80 µs) im professionellem Einsatz, mit Statemachinen in C direkt programmiert aber streng nach einem festen Style. Es wurde allerdings im Team gegen Events, nur conditional, entschieden. Man hätte gar nicht mit Events ankommen dürfen. Die Statemaschinen für Beeinflussung der Regelung (Sollwert-Auswahl etc.) laufen im Hardware-Interrupt, so wie die gesamte Regelung. Verwendung untersetzter Abtastzeiten für einzelne langsamer notwendige Regelungsteile.

E) Es ist eine ganzheitliche Betrachtung notwendig. Dazu gehört auch die Frage, wo eine Statemachine abgearbeitet wird oder eine Eventbehandlung notwendig ist. Das kann auch in einem schnellen Regelungsinterrupt, vielleicht mit einer Zykluszeit von 10 µs notwendig sein. Somit sind Systeme, die einen eigenen Thread im Multithreading voraussetzen, nicht hilfreich. Daher auch die Überlegungen zu Abtastzeiten und Exceptionhandling in diesem Artikel.

2. Schnelle Abtastzeiten

Wie schnell müssen Abtastzeiten sein? Man versteht darunter teils 1 ms als „schnell“, im Embedded Linux Bereich scheint man nicht zu verstehen dass das nicht ausreicht.

Beispiel aus der Vergangenheit, 90-ger Jahre: Hauptabtastzeit 1.5 ms, Abtastzeit für Netzregelung 833 µs (1/24 der Netzperiode), Zwischenkreisspannungsüberwachungsinterrupt 250 µs mit einem 16-bit-Prozessor in C realisiert, Spitzenstromeingriff 16 µs im FPGA realisiert.

Heute möglich: 10 µs-Hardwareinterruptzyklus im DSP, umfangreiche Realisierungen mit Codegenerierung im FPGA möglich, dort < 1 µs.

Braucht man das?

Beispiel Stromregelungen: Die Abtastzeit muss < 1/10 der kürzesten physikalischen Zeitkonstante sein! Typische phys. Zeitkonstanten liegen bei 1 ms (Ausgangsdrosseln von Stromrichtern, Streuinduktivtät von Netztrafos). Ggf. kürzer, weil: Je kleiner, desto kleiner die Teile (in m und t Cu, Fe gemessen). Kostenfaktor der Leistungshardware. Die Größe der Induktivitäten richten sich nach den Schaltfrequenzen der Leistungshalbleiter, diese sind heute höher.

Die Totzeit der Regelung beträgt theoretisch mdst. 2 x Abtastzeit, praktisch 3..4 mal Abtastzeit, wegen Messertaufbereitung, Kommunikation und dergleichen. Totzeit bedeutet Phasenverschiebung des Signals. Phasenverschiebung bedeutet Schwingneigung: Niquist Kriterium (https://de.wikipedia.org/wiki/Stabilitäskriterium_von_Nyquist) Phasendrehung 180°, Kreisverstärkung muss dabei < 1.0 sein!

Ein integratives Glied dreht grundsätzlich 90°. Addiert sich zu dieser physikalischen Phasendrehung noch die Drehung wegen der Totzeit, kann das Nyquist-Kriterium eher erreicht werden, es schwingt. Bereits ein Integrator und zwei etwa zeitgleiche T1-Glieder erreichen die 180° im Bereich der Grenzfrequenz bei einer Dämpfung von 0.5 der T1-Glieder.

Eine Ausgangsdrossel eines Umrichters kann im wesentlichen als Integrator angesehen werden, denn die Wirkung des Leitungswiderstandes setzt erst bei solch hohen Strömen ein, bei denen der Umrichter bereits kaputt ist. Damit hat man eine Phasendrehung zwischen gestellter Ausgangsspannung und zu regelndem Strom von knapp 90°.

Bei einer Regelungszykluszeit von 100 µs kommt man auf eine Totzeit von 0.3 bis 0.4 ms. Das führt für bei einer Frequenz von 350 Hz (Periode 2.8 ms, gerechnet mit (360/45) * 0.35 ms Totzeit) bereits zu einer zusätzlicher Phasendrehung von 45°. Die Regelverstärkung muss durch natürliche T1-Glieder oder in die Regelung eingebrachte Dämpfungen (T1-Zeitkonstante von 0.45 ms =^350 Hz) bereits unter 1.0 der Kreisverstärkung liegen, die Dämpfungen bringen weitere 45° an der Knickfrequenz (Dämpfungsfaktor 0.707) mit sich. Damit sind Störungen mit Frequenzen über 350 Hz nicht mehr ordentlich beherrschbar. Diese treten aber auf bei Schalthandlungen (Leistungsschalter) im Zusammenhang mit Schwingneigungen von Kabeln und Leitungen. Man braucht folglich für solche Dinge eine Zykluszeit deutlich unter 100 µs. Die durch die Leistungshalbleiter erzeugten Schalthandlungen spielen dafür keine Rolle, denn diese sind gleichmäßig wiederholend.

Mechanische Bewegungen: 1 m/s ist noch langsam für eine normale Bewegung (Roboter), bedeutet 1 mm pro ms. Um genau zu treffen, richtig abzubremsen, sind auch dort Abtastzeiten < 1 ms notwendig.

Da die Prozessoren diese Abtastzeit können, werden sie auch genutzt.

2.1 Untersetzte Zykluszeiten anstatt Multithreading

Es gibt eine einfache Möglichkeit, verschiedene Zykluszeiten in einem Hardwareinterrupt zu realisieren:

* Der Hardwareinterrupt selbst ist die schnellste Zykluszeit.
* Im Hardwareinterrupt werden bestimmte Abarbeitung nur jedes n-te mal aufgerufen. Damit mit größerer Zykluszeit, aber niemals unterbrochen von einem anderen Zyklus.
* Es gibt eine Verteilung verschiedener Abarbeitungen auf die untersetzten Zyklen und weitere Untersetzung. Insgesamt sollte für eine gleichmäßige Auslastung der Abarbeitungszeit im Interrupt gesorgt sein.

Der Nachteil dieser Herangehensweise ist:

* Man muss abschätzen, wieviel und welche Abarbeitungen in jeder untersetzten Zeitscheibe untergebracht werden. Es gibt nicht den bestimmten lansameren Zyklus, sondern der langsame Zyklus muss zusätzlich geeignet in unabhängige Portionen aufgeteilt werden. Das kann bei manueller Programmierung als starker Zusatzaufwand empfunden werden.

Der Vorteil ist:

* Keine gegenseitigen Unterbrechungen. Damit entfallen alle Aufwände für Mutex. Das kann erheblich sein. Daher ist diese Variante für schnelle Abarbeitungen insbesondere geeignet.

Simulink bietet diese Variante der Untersetzung an. Selbst die Aufteilung auf die einzelnen Zeitschalen wird mit beispielsweise `ssSetOutputPortOffsetTime(…)` in S-Functions unterstützt. Wie damit konkret im generierten Code umgenangen wird, kann aber immer noch gut manuell bestimmt werden. Wichtig ist, dass die Simulink-Simulations-Engine intern adäquat arbeitet.

Die eigenen Erfahrungen in einem Projekt der schnellen elektrischen Regelungen waren sehr gut. Dort wurde die Untersetzung zunächst eingesetzt, weil der Gesamtrechenzeitbedarf zu groß wurde und es vollkommen unsinnig ist, eine Grundschwingungsregelung mit gemittelten Größen im gleichen schnellen Zyklus zu rechnen wie eine Momentan-Stromregelung zum Abfangen von Spitzen.

Beispiel für eine mögliche Aufteilung: Der Abtastinterrupt für eine Netzregelung wird mit 47.61 µs festgelegt. Das sind 1/420 der Netzperiode von 20 ms (50 Hz). In dieser Teilung ist die Zahl 7 mit enthalten (420=2*2*3*5*7). Damit wird die durchaus auftretende 7. Harmonische noch mit wenig Aliasing erfasst, auch bei nicht netzsynchronisiertem Abtasttakt. Ein erster Untersetzungszyklus 4:1 realisiert 190 µs. Einer der 4 Slots dient als weitere Untersetzung 7:1, sprich 28:1 =^ 1.33 ms, mit 5 Slots in denen langsamere Aufgaben verteilt werden müssen. Zwei dieser Slots sind jeweils 1:3 und 1:5 untersetzt, ergibt 4 ms und 6.66 ms. Diese Zykluszeiten sind allgemein gut brauchbar für verschiedene Aufgaben an einem elektrischen 50 Hz-Netz. Wie genau aufgeteilt wird, kann anwendungsspezifisch sein. Eine Rechenzeitmessung (Differenz über schnellen Hardwarezähler) gibt genauen Aufschluss darüber, wieviel Rechenzeit aktuell und als max und min für welche Aufteilung benötigt wird, getrennt nach schnellen Hauptinterrupt und den jeweiligen Schalen:

```
********$$$$,,,,,
```

Einfach symbolisch dargestellt: `********` ist der Rechenaufwand des schnellen Zyklus, leicht pulsierend. In der Zeit `$$$` wird jeweils genau eine der Aufgaben der untersetzten Zyklen gerechnet, niemals in einem schnellen Zyklus verschiedene. Sind alle Aufgaben etwa gleich lang, gibt es wenig Pulsierung. Ist eine Aufgabe regelrecht leer (nicht besetzt), dann gibt es mehr Hintergrundzeit. `,,,,,` ist dann die Restzeit des Interrruptzyklus, in dem sich die CPU in der main-Schleife befindet.

Es sollte vermieden werden, dass es weitere Arbeits-Interrupts gibt, die den Haupt-Recheninterrupt unterbrechen oder dessen Ausführung auch nur verzögern. Nur so weit es hardwaremäßig notwendig ist, und dann nur kurz, muss es Interrupts nur zum Abholen und Ablegen von Daten geben.

2.2 Was ist dabei in der Hintergrundschleife abarbeitbar?

Der Prozessor fällt jeweils nach Interruptende in die Hintergrundschleife. In der Hintergrundschleife kann ein Scheduler ablaufen. Man kann aber auch für spezielle Embedded-Belange lediglich ein Polling betreiben und mögliche Informationen aus Socket-Stacks oder im FPGA abfragen und abholen. Insbesondere wurde in den Hintergrundprozess der Diagnosezugriff gelegt. Je stärker der Interrupt ausgelastet ist, desto weniger Zeit bleibt für den Hintergrundprozess, was aber ebenfalls gut diagnostizierbar ist.

Im Hintergrund können verschiedene Aufgaben per Polling nacheinander abgearbeitet werden, Wenn jede Aufgabe entsprechend kurz ist, und es ggf. eine Priorisierung von mehreren Aufgaben gibt, ist der Hintergrundprozess geeignet, auch komplexe Dinge zu tun. Die Notwendigkeit eines Multithread-Betriebssystems herkömmlicher Art (verdrängende Threads) wird damit mindestens für typische Embedded-Prozessoranwendungen in Frage gestellt.

Unabhängig davon kann man für bestimmte Aufgaben dennoch ein Mulitthread-Betriebssystem gut gebrauchen, dort wo es etabliert ist.

2.3 Darf der Interrrupt zeitlich überlaufen

Er darf, wenn dies typisch nur einmalig auftritt und die Zyklen direkt danach nicht wieder überlaufen. Es ist dabei wichtig darauf zu achten, dass die Ausgaben im Interrupt, die vor Beginn des nächsten Zyklus in die Hardware geschrieben werden, rechtzeitig ausgegeben werden. Daher kommt die *******-Reihe im Bild oben auch zuerst dran. Der Überlauf darf also in den untersetzten Abtastzeiten auftreten, dort ist er auch eher zu erwarten. Kommt es beispielsweise immer genau in einer untersetzten Abarbeitung zu einem Überlauf, weil die Aufteilung noch nicht ganz gut ist, dann ist das weniger tragisch weil im folgende Interrupt die Sache wieder eingefangen wird.

Werte aus dem Prozess müssen immer mit einer Hardwareflanke (im FPGA) gespeichert werden, die am Anfang des Zyklus liegt. Gleiches gilt für die Gültigkeitsschaltung der Ausgabeinformation. Dann ist das System robust gegen Rechenzeitschwankungen, wenn die notwendigen Werte jedenfalls im Zyklus eingelesen und ausgegeben werden.

2.4 Sind in der schnellen Zykluszeit Statemachinen notwendig?

Sollwertauswahl

Sonderfallbehandlung (z.B. Kurzschluss in elektrischen Netzen)

Umschalten von Bewegungen wenn Ziel in der Nähe

Untersetzte Abtastzeiten

Also eigentlich schon.

3. Event Queue in der schnellen Zykluszeit

Wie einleitend erwähnt, wurde dies in einem Entwicklerteam gar nicht als sinnvoll erachtet. Es wurde statt dessen auf conditional Statemachines gesetzt.

Das ist aber ein Schnell-Schluss.

* Ein Event kann mit wenig Aufwand in einen Queue eingeschrieben werden. Zeiger weitersetzen, eintragen.
* Der Eintrag eines Events in die Queue muss normalerweise unter Lock erfolgen. Es gibt aber einerseits das cmp_and-set-Prinzip (atomic access lockfree programming). Andererseits: Wenn das System so gebaut ist, dass die betreffenden Threads sich nicht unterbrechen können braucht es kein Lock bei überschaubaren Lösungen. Das ist bei den untersetzten Abtastzeiten der Fall.
* Man muss das Event nicht im Heap anlegen denken zu müssen (siehe Folgeabschnitt). Es kommt dabei auch darauf an, wieviel Daten mitgeführt werden.
* Die Abfrage ob ein Event vorliegt (meist nicht) geht viel schneller als das prinzipielle Eintreten in eine Statemachine und dort das Abfragen der teils komplexeren Bedingungen.

Die Event-Lösung kann also sogar rechenzeitoptimaler sein.

4. Müssen Event-Daten im Heap angelegt werden?

Events wurden im Rhapsody-Framework in Version 7.3 und davor im Heap angelegt (C++-Codegenerierung). Das war offensichtich nicht konfigurierbar und wurde auch nicht hinterfragt.

Andererseits ist es allerdings angeraten, in langlaufenden embedded Systemen keine dynamische Speicherallokierung zu verwenden. Das ist langjährig Stand der Technik und auch heute noch gültig:

* Beachtet man diese Regel nicht, passiert erstmal nichts. In der Regel werden die dynamischen Speicher alle immer wieder frei gegeben, so dass sich nicht die gefürchtete Fragementierung einstellt, bei der es nirgends mehr einen genügend großen Speicherplatz gibt.
* Das Fragmentierungsproblem stellt sich dann ein, wenn große und kleine Blöcke unterschiedlich lange im Heap angelegt werden und der Heap-Platz nicht allzugroß ist.
* Es ist ggf. schlecht vorhersagbar wann es ein Fragmentierungsproblem geben könnte.

4.1 Events auf festen Speicherplätzen

Events werden häufig überhaupt nicht mehrfach benötigt. Wenn es von einer Entry-Action eine Benachrichtigungsnotwendigkeit zu einem anderen Statechart gibt, so ist damit zu rechnen dass der andere Statechart in einem ähnlichen Zeitzyklus läuft (häufig sogar im selben Zyklus). Damit wird das queued Event im selben Zyklus oder zeitnahe wieder aus der Queue entnommen und folglich entweder appliziert oder verworfen. Dereferred Events stören dieses Prinzip, aber diese sind ggf. anderweitig auch störend, ich würde sie ausschließen.

Wenn ein Event in dieser Konstellation nicht aus der Queue geholt wird, dann klemmt wohl dieser Thread, es ist etwas sowieso nicht in Ordnung. Dann ist es auch nicht zweckdienlich, das gleiche Event mit einer zweiten Instanz nochmal in die Queue zu packen.

Folglich kann beim Speichern in die Queue die einzige Event-Instanz (für diese Verbindung) getestet werden, dass sie denn wie erwartet frei ist. Ein Event selben Typs, das in nicht parallelen Zweigen in verschiedenen Actions (Transition, Entry, Exit) gesendet an die selbe Zielinstanz, die aber so gewollt nicht sofort hintereinander betreten werden (weil es nicht gewollt ist dass das Event mehrfach gesendet wird), können ebenfalls von der selben Instanz des Events leben. Im Zweifelsfall können aber eigene Instanzen angelegt werden, wenn es keine Speicherknappheit gibt.

4.2 Realisierung Events mit festen Instanzen

Damit wird statisch für jedes Event-Sende-Vorkommen eine Event-Instanz des erforderlichen Typs entweder statisch angelegt oder einmalig beim Hochlauf im Heap allokiert. Einsparpotential bezüglich gemeinsamen Instanzen für garantiert nicht sofort hintereinanderliegende Event-Sende-Vorkommen sind möglich.

Man kann diese Instanzen manuell anlegen gemeinsam mit der Planung der Statemachines und muss in der Statemachine dann auf die entsprechende Event-Instanz verweisen:

```
send(evXy); //als Action im Statechart
```

gibt dann mit `evXy` die Instanz an.

Man kann auch aus einer Statemachine-Beschreibung (XMI, IEC-61499) bei dem der Event-Typ im Statechart angegeben ist aufgrund der Detektion der Transition die Instanz automatisch auswählen

(und vorher die notwendigen Instanzen automatisch anlegen). Das dürfte eher praktikabel sein, erfordert eine entsprechende Code-Generierung.

Auch bei Events, die aufgrund Interprozess- oder Kommunikation zwischen Geräten empfangen werden (Telegramme), kann der Telegramm-empfangende Prozess, der die Events generiert, dafür feste Instanzen verwenden. Es muss dafür bekannt sein, welche Events überhaupt empfangen werden können. Es gilt die gleiche Überlegung: Wird ein Event wiederholt empfangen, dass aber noch nicht verarbeitet ist, dann klemmt es wohl an der Verarbeitungsstelle.

4.3 Wie ist zu verfahren, wenn die Event-Instanz besetzt ist?

Es kann sein, dass in bestimmten Zuständen eher vom Zufall bestimmt das Ausräumen der Queue etwas länger dauert, oder ein Event-Emitter schneller als geplant auf das selbe Event kommt. Es ist dann denkbar, dass die Event-Daten erneuert werden, wobei dabei auf Mutex zu achten ist. Mutex entfällt aber, wenn dies in untersetzten Abtastzeiten stattfindet. Damit ist die Aktion nicht komplex.

Lösungen, das Event aus der Queue zu entnehmen und erneut einzureihen sind wahrscheinlich weniger zielführend da dann das Event wieder entsprechend später dran kommt.

Das Erzeugen des Events zu unterlassen weil das Event noch in Arbeit ist, ist dann hilfreich wenn der zweite Event-Emittierwunsch eigentlich nur eine Wiederholung des ersten ist. „*Es ist schon gesagt, nöl doch nicht*“. Erst wenn ein wirkliches Timeout abläuft und keine Antwort kommt, ist Aktion angesagt. Das ist aber anderweitig in der Statemachine sicher schon vorgesehen.

Wenn die Abarbeitung insgesamt klemmt, beispielsweise bei sich drehendem Run-to-completion, oder weil fälschlicherweise die selbe Instanz für Transitions geplant wurde, die schnell hintereinander kommen, kann diese Situation typisch auftreten. Das sind alles Entwurfsfehler, keine Fehler die normalerweise auftreten können.

Das typische Auftreten dieser Situation kann ein ***throw*** wert sein. Damit wird die Situation gemeldet, es wird abgebrochen, es gibt ein Fallback, wichtig wenn man sich in einer realen physikalischen Umgebung befindet. Damit wird zwar dieser Programmteil, die Statemachine, nicht mehr abgearbeitet, aber der Rest funktioniert noch einschließlich des möglichen Debug-Monitor-Zugriffs.

Will man im Embedded-Bereich ohne try-catch arbeiten, dann kann man jedenfalls einen Fehlerhinweise setzen, der mit Debug-onlinezugriff auch im laufenden Betrieb auslesbar ist, oder eine Meldung erzeugen, und jedenfalls das Senden des Event einfach ignorieren. Tritt dann dieser Fall im Betrieb doch unerwartet auf, dann ist die Ursache zu ergründen, oder es dabei zu belassen. Man kann jedenfalls den Algorithmus weiter laufen lassen, nur das Event nicht senden. Das ist kein Widerspruch für sich allein, da das Event sich in Sendung befindet, und das Problem sich also eher beim Abholen des Events befindet.

Es ist bei Grafikprogrammierung der Fall aufgetreten, dass der eventauslesende Prozess tatsächlich nicht mehr lief, weil das entsprechende Fenster zugeklappt war und der Prozess also beendet, das Event war aber noch in dieser Queue stand und damit blockiert war für weitere Nutzung. Eigentlich hätte der zugeklappte Prozess vorher seine Queue leeren müssen. Jedoch gibt es den Unterschied zwischen perfekter Software und noch nicht ganz fertiger Software, die wegen solcher Dinge dann klemmt.

In diesem Fall ist es aber nicht zielführend, beim Einschreiben irgend etwas mit einem blockierten Event zu tun. Vielmehr muss es eine zentrale Freigabe geben, die beispielsweise mit einem „clean“ des Systems zusammenhängen kann: Wenn in einer Grafikanwendung ein „clean“ in einem sonst

ruhigem Zustand gerufen wird, werden alle Event-Instanzen kontrolliert und freigegeben. Dann kann ein doch vorkommender Klemmer beseitigt werden.

Das clean muss sich ggf. auf die Queue, nicht das Einzelevent beziehen. Sind dann immer noch Events blockiert, besteht die Frage: „in welcher Queue?“.

Bei new-allokierten Events ist dieser Zustand auch möglich, nur das das Klemmen dann nicht bemerkt wird, erst wenn „Speicher aus“ ist.

4.4 Sind feste Event-Instanzen die universale Lösung?

Es spricht vieles bis alles dafür:

* Ein Event ist nie für längere Zeit in einer Queue, maximal die Zeit bis der verarbeitende Prozess seinen Zyklus hat.
* Es ist nicht zweckführend, dem verarbeitenden Prozess mehrere Events an die gleiche Stelle zu schicken in einem schnelleren Zyklus als er selbst abarbeitet. Bei Sonderfällen können an diesen Stellen mehrere Instanzen vorgesehen werden, beispielsweise wenn mit zwei gleichen Events hintereinander zwei Transitionen nacheinander geschaltet werden sollen. In solchen Fällen könnte es aber für den Entwurf der Statemaschine besser sein, zwei verschiedene Events zu verwenden. - könnte übersichtlicher sein.

5. EventQueue-Fblock für Simulink

Den Events, die in IEC-61499 die Abarbeitung auslösen, entsprechen in Simulink die Function-Call-Trigger. Diese gelten ähnlich wie in IEC-61499 für allgemeine Fblocks für den Aufruf von Function-Call-Fblocks in Simulink. In beiden Fällen werden sie auch für Events in Statemaschinen verwendet.

In Simulink gibt es außer den Function-call-like so genannten „Events“ auch die „Messages“ mit einer eigenen Queue im Statechart-Fblock pro Message-Input. Das erscheint als eine „andere“ Lösung , die hier nicht passt. Das wäre im einzelnen aber noch nachzuweisen.

Damit stellt sich die Aufgabe zunächst wie folgt:

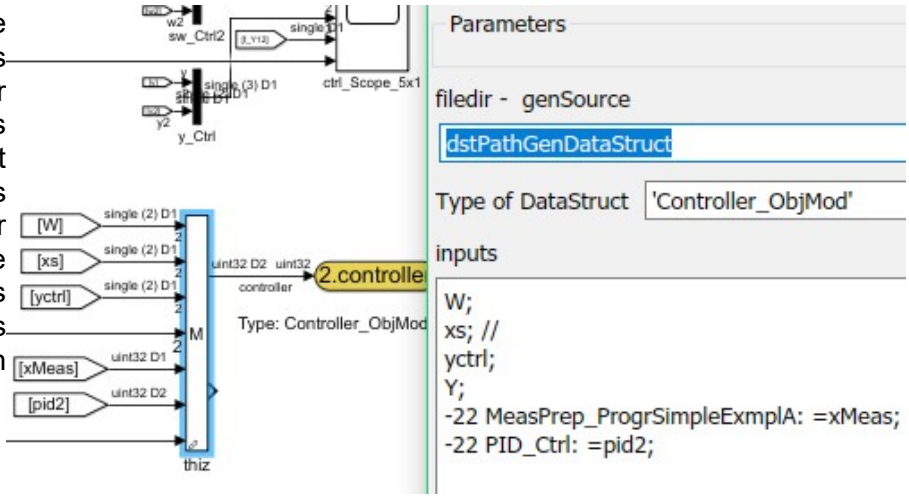
* Der Auftrag zum Erzeugen eines Function calls, in einer S-Function in C mit `ssCallSystemWithTid(...)` zu realisieren, ist als Event zu speichern.

Events im allgemeinen sind mit Daten verbunden. In IEC-61499 werden die Daten zum Event zugehörig aber an eigenen Pins verdrahtet. Sie werden also nicht etwa mit `ev.data` oder `ev.getData()` zugegriffen. Simulink kennt dies auch nicht. A priori haben Events in Simulink-Statechart-Fblocks keine Daten. Insoweit kann man bei Simulink wie bei IEC-61499 vorgehen. Es werden Daten zu Events zugehörig definiert, als Datenpins an den Statechart-Fblocks.

* Zu den Events werden zugehörige Daten definiert.

Die Definition der zugehörigen Daten kann in Simulink grundsätzlich mit einer Bus-Definition erfolgen, die nach der Codegenerierung dann eine `struct` darstellt. Da aber sowieso mit einem Satz spezieller S-Function gearbeitet wird, und die Busdefinition in Simulink bedientechnisch so schön nicht ist, wird eine Arbeit des Verfassers wiederverwendet: *DataStruct_Inspc*-Fblocks.

Wie im Bild erkennbar, die Datendefinition folgen aus einem Dialog. Bei einer automatischen Generierung des Simulink kann dieser Text passend generiert werden. Das Beispiel zeigt eine Datenstruktur eines Moduls, dessen Handle (uint32-Wert) an ein anderes Modul weitetgegeben wird, dass dann direkt auf die Daten zugreifen kann.



Zugehörig zu einem Event-Einschreibe-Fblock (Event-Erzeugung) kann dieser Daten-FBlock beliebig definiert werden, das sind die Event-Instanzdaten.

Zum Einschreiben eines Events wird ein *EventCreate*-Fblock definiert, der den Dateneingang als Handle besitzt und einen Trigger-Eingang benutzt. Ist dieser Boolean, dann kann die steigende Flanke als Einschreibeimpuls genutzt werden. Wird dieser Fblock selbst in einem Triggered Subsystem oder in einem Statechart-Fblock gerufen, dann schreibt jeder Aufruf das Event in die Queue. Das ist konfigurierbar. Der *EventCreate*-Fblock ist mit einer Event-Id parametriert.

Von den Daten werden in der Event-Queue nur das Handle gespeichert. Es muss im Daten-Fblock gesichert sein, dass die Daten nicht geändert werden bevor das Event verarbeitet wird. Die Daten stellen die Event-Instanz dar, insoweit ist das ok.

Es wird allerdings auch der einfache Fall vorgesehen: Daten direkt in die Queue mit einschreiben, für einfache Daten (int32-Info) statt dem Handle. Dann braucht es keinen extra Daten-Fblock, das Modell wird einfacher.

Es gibt einen zentralen EventQueue-Fblock pro Event-Queue (pro Zykluszeit ein Fbllock). Dieser hat einen Ausgang, ein Handle (uint32). Parameter für Speichertiefe. Weitere Debug-Ausgänge (Anzahl derzeit in Queue befindliche Einträge).

Zum Auslesen wird je ein *EventRead*-Fblock an diesen EventQueue-Fblock verbunden (Handle-Verbindung). Der FB ist parametriert mit einer Event-Id (die beim Einschreiben vergeben wird). Alle verbundenen *EventRead*-Fblock sind am *EventQueue*-Fblock registriert. Erkennt der *EventQueue*-Fblock ein ausgelesenes Event mit der passenden Id, schreibt er dann auf die Ausgänge des *EvenRead*-Fblock die entsprechenden Daten und löst die `ssCallSystemWithTid(...)` -Funktion aus. Diese ist mit einem Ausgang des *EventRead*-Fblock verbunden, der als Triggerausgang entweder an ein Function-Call-Subsystem verbunden ist oder eben an einen Statechart-FBlock.

Sollte funktionieren. Wird gebaut.

Die Kern-Funktionen der S-Functions in C werden bei Codegenerierung dann direkt gerufen (in Simulink über tlc-Files organisiert). Die Kern-Funktionen dieser Fblocks stellen dann gleichzeitig die Realsiierungen im Embedded-Bereich dar. Codegenerierung soll betrieben werden, zunächst als Test mit Visual Studio am PC.

6. EventQueue nutzbar in schnelle Interrupts (abgetastete Systeme)

Folgende Beschreibung geht von einer konkreten Realisierung für Simulink aus. Die Kern-C-Realisierung ist auch für abgetastete Systeme geeignet, die häufig in schnellen Interrupts bearbeitet werden, aber auch für klassische Multithread-OS-Umgebungen anwendbar. Diese soll auf den statischen Event-Instanzen aufbauen.

Die Software ist in die emC-Library eingegliedert und in einem eigenem und einem Statemachine-Test standardgemäß einem Test unterzogen.

https://github.com/JzHartmut/src_emC: Quellarchiv, dort `emC/StateM/*` als c und h-Files.

https://github.com/JzHartmut/Test_emC: Testumgebung, Test Eventqueue eingebettet in den Gesamttest in `src/test/cpp/org/vishia/emC/StateM/test_StateM/*`

Nachfolgend werden einige Details und Grundsatzüberlegungen dargestellt.

6.1 Lockfree Mutex mit AtomicAccess CmpAndSet für Einschreiben von Events

Das Einschreiben von Events in die Queue muss threadsafe sein. Es müssen mehrere Threads oder auch direkt Hardware-Interrupts einschreiben können, ohne dass dies zu Störungen führt.

Traditionell wird dafür Mutex-Lösungen mit Zugriffs-Guards (Monitor, Semaphore) verwendet. D.h. dem Multithread-OS-Scheduler wird mitgeteilt, dass für diese bestimmte Zeit kein Threadwechsel erfolgen soll für andere Threads, die die selbe Mutex-Controll-Instanz verwenden.

* Das erfordert einen gewissen Rechenzeitbedarf, da zunächst das Mutex-Control-Object selbst unter unteilbaren Zugriff entsprechend gesetzt werden muss. Dieser tritt auch dann auf, wenn es keinen konkurrierenden Zugriff gibt.
* Gibt es einen konkurrierenden Zugriff, dann hat der Scheduler bereits den Thread gewechselt. Es wird zurück gewechselt auf den Thread, der den Mutex-Control inne hat, Threadwechselzeiten.
* Ein Interruptzugriff zum Einschreiben in die Queue geht damit gar nicht. Der Interrupt muss erst noch einen entsprechenden Multithread-OS-kontrollierte Behandlung starten. Nur dort kann das Event eingeschrieben werden.

Für einfache Embedded Systeme kann ein Mutex-Zugriff auch einfach unter Interruptsperre für die Dauer des Einschreibens erfolgen. Allerdings wird diese Möglichkeit häufig zu Recht kritisch betrachtet, da zumeist alle Interrupts gesperrt werden.

Wegen diesem Zeitaufwand lohnt sich auch bei Vorliegen eines Mulitthread-OS die Nutzung des Lockfree-AtomicAccess, wenn dieser möglich ist. Er ist für die Eventqueue möglich.

Prinzip des Lockfree-Atomic-Access:

Die Grundlage wurde in Intel-CPUs schon ab 486 geschaffen: Es gibt Maschinenbefehle, im Beispiel mit C-Aufruf: (siehe `org/vishia/emC/sourceSpecials/hw_Intel_x86_Gcc/os_atomic.c`)

```
int32 compareAndSwap_AtomicInteger(int32 volatile* reference, int32 expect, int32 update)
{ __typeof (*reference) ret;
  __asm __volatile ( "lock cmpxchgl %2, %1"
```

```
        : "=a" (ret), "=m" (*reference)
        : "r" (update), "m" (*reference), "0" (expect));
  return ret;
}
```

Dieser Typ cmpxchg führt das Setzen eines Speicherwertes unteilbar in der CPU nur dann aus, wenn ein erwarteter Wert dort steht, nur dieser wird also überschrieben. Bei einem anderen Wert als erwartet wird der Speicher nicht geändert. Wesentlich dabei ist, dass dieser Befehl auch durch den Cache hindurch greift. Es wird also auf dem gemeinsamen RAM bei Mulitcore-Systemen getestet und geändert.

Für einfache Embedded-Prozessoren, die diesen Befehlttyp nicht haben, genügt ein einfacher Ersatz, der unter Interruptsperre testet und setzt. Die Interruptsperre ist dabei nur für diese kurze Befehlsfolge nötig, also nicht auf der Ebene der Anwendungsprogrammierung in diesem Fall für für die Event-Queue.

Die moderne Unterstützung für Microsoft-PCs sieht dafür wie folgt aus:

```
int32 compareAndSwap_AtomicInteger(int32 volatile* reference, int32 expect, int32 update) {
  return InterlockedCompareExchange(reference, update, expect);
}
```

Der `InterlockedCompareExchange(…)` gehört zur Windows-API, intern wird intrinsic dieser Spezialbefehl der CPU erzeugt.

In beiden Fällen erfolgt der Aufruf aus Anwenderebene (hier dem Eventqueue-Algorithmus) mit `compareAndSet_AtomicInteger(…)`. Es gibt dabei noch ein Detail swap versus set. Das Original heißt swap, man braucht aber nur ein Set. Die folgende Befehlszeile zeigt die Einbettung in `inline bool add_EvQueue_StateM_vishiaOrg(…)`, siehe `org/vishia/emC/StateM/evQueue.h`.

```
  int pwrCurr;  //The position where the event should be written to the queue.
  do {
    int volatile* ptr = &thiz->pwr;
    pwrCurr = thiz->pwr;
    int pwrNext = pwrCurr + 1;
    if(pwrNext >= thiz->sizeQueue) { pwrNext =0; } //modulo
    if(pwrNext == thiz->prd) return false;    //RETURN: no more space in queue
    ok = compareAndSet_AtomicInteger(ptr, pwrCurr, pwrNext); //set for next access.
  } while(!ok && --abortCt >=0); //repeat if another thread has changed thiz->pwr.
```

Die Verwendung von do … while ist Strategie. Wichtig ist, dass mit einer atomic-Änderung der gesamte Zugriff ordnungsgemäß als Mutex erledigt ist. Müssen mehr als eine Datenzelle extern unter Mutex geändert werden, dann kann diese Strategie nicht verwendet werden.

Die Schleife wird nur dann wiederholt, wenn gerade mal zwischendurch ein anderer Thread auch auf die pwr-Position zugegriffen hat. Das passiert normalerweise selten. Das einmalige Wiederholen der aller Befehle der Schleife ist ein kürzerer Rechenzeitaufwand als das Setzen und Löschen der Mutex-Semphore. Sollte sich ein Fehler eingeschlichen haben oder ein anderer Thread sehr schnell dauerhaft an dem pwr spielen, dann wird die Schleife mit `abortCt` abgebrochen. Das ist aber eine Fehlersituation:

```
if(abortCt <0) {
    //it is a problem of thread workload, compareAndSet does not work.
    THROW_s0n(IllegalStateException, "thread compareAndSet problem", 0, 0);
    return false; //RETURN for non-exception handling, THROW writes a log entry only.
  }
```

Wenn Exceptionhandling verwendet wird, geht es beim CATCH in der Aufrufumgebung weiter. Ohne Exceptionhandling gibt es hier einen Logeintrag (falls entsprechend programmiert) und mit `return false` die Verhinderung, dass dieses Event abgelegt wird, es geht nicht. Zum Exceptionhandling siehe https://www.vishia.org/emc/html/Base/ThCxtExc_emC.html und https://www.vishia.org/emc/html/Base/ExceptionHandling_de.html

Im Java-Bereich ist beispielsweise die `java.util.concurrent.ConcurrentLinkedQueue` ebenso realisert, mit folgendem Code-Spnippet aus der zugehörigen originalen Source aus Java 1.8.0.211

```
    public E poll() {
        restartFromHead:
        for (;;) {
            for (Node<E> h = head, p = h, q;;) {
                E item = p.item;

                if (item != null && p.casItem(item, null)) {
                    // Successful CAS is the linearization point
                    // for item to be removed from this queue.
                    if (p != h) // hop two nodes at a time
                        updateHead(h, ((q = p.next) != null) ? q : p);
                    return item;
                }
                else if ((q = p.next) == null) {
                    updateHead(h, p);
                    return null;
                }
                else if (p == q)
                    continue restartFromHead;
                else
                    p = q;
            }
        }
    }
```

Interessant ist dabei schon, dass es in Java auch eine Art Goto gibt, zwar strukturiert verwendet. Der Aufruf `p.casItem(item, null)` ist in der gleichen source:

```
    boolean casItem(E cmp, E val) {
            return UNSAFE.compareAndSwapObject(this, itemOffset, cmp, val);
        }
```

Wobei `UNSAFE` eine entsprechende Instanz aus `sun.misc.Unsafe` ist. `Unsafe` sollte nicht irritieren, die Verwendung der `ConcurrentLinkedQueue` und weiterer damit gebauter Container ist sicher und verbreitet. In Java ist diese Art der Programmierung seit der Version 1.5 etwa ab 2004 eingeführt. Insbesondere für intensive Algorithmen serverseitig ist das Einsparen von Mutex-Controlled lock (ist in Java standardgemäß mit `synchonized(obj) { }` realisierbar) wesentlich.

Für das Schreiben in die EventQueue ist das atomic cmpAndSet nun verwendbar wie folgt:

* Für das Einschreiben wird der Write-Index (`pwr`) weiterbewegt, der neue Stand wird mit cmpAndSet eingetragen. Damit ist die Position zum Einschreiben eines Event-Entries prinzipiell gesichert.
* Wird während des Incrementierens des pwr dieser Prozess unterbrochen, dann schlägt cmpAndSet fehl und wird wiederholt. Der unterbrechende Thread hat sich dann nach vorn geschoben und zuerst komplett fertig geschrieben.

* Ist die Write-Position gesichert, dann kann ein danach unterbrechender Thread ein weiteres Entry fertig schreiben.
* Ein lesender Thread stellt auf der möglichen Leseposition (der pwr ist bereits incrementiert) fest, dass kein `evIdent` eingetragen ist (`==0`) und liest daher nicht auf dieser Position. Auch ein danach bereits fertig eingetragenes Event-Entry wird nicht gelesen.
* Das Entry wird belegt, als Abschluss wird evIdent eingetragen. Nunmehr kann ein sofort unterbrechender Leseprozess dort lesen und seine Leseposition `prd` erhöhen. Damit ist nur ein vollständig abgelegtes Event-Entry lesbar, damit Mutex-gesichert.
* Die Queue ist leer entweder wenn `prd == pwr` (Leseposition liegt auf der Schreibposition) oder wenn `entries[prd].ident ==0`. Der letzte Eintragsplatz wird nicht genutzt, dann wäre wieder `prd == pwr` und die volle Queue kann nicht mehr von der leeren Queue unterschieden werden.
* prd und pwr arbeiten beide umlaufend, nach dem Incrementieren wird auf Überlauf getestet, es wird nur ein gültiger Index abgespeichert.

Der gesamte Algorithmus sieht damit wie folgt ungekürzt aus, inline im Header `org/vishia/emC/StateM/evQueue.h` :

```
/**Adds an event with given evIdent and data to the queue with atomic lockfree mutex.
 * This operation can be invoked in any thread or interrupt.
 * @arg idEvent to add to queue in the next free Entry_EvQueue_StateM_vishiaOrg.
 * @arg hData will be stored in the entry.
 * return false if no more space in queue. true on success.
 */
inline bool add_EvQueue_StateM_vishiaOrg ( EvQueue_StateM_vishiaOrg_s* thiz
  , uint16 idEvent, uint32 hData) {
  #ifndef NO_PARSE_ZbnfCheader
  int abortCt = 100;
  bool ok;
  if(idEvent >= thiz->sizeInstances) {
    THROW_s0n(IllegalArgumentException,"faulty event Id =",idEvent, thiz->sizeInstances);
    return false;  //for systems without exception handling
  }
  int pwrCurr;  //The position where the event should be written to the queue.
  do {
    int volatile* ptr = &thiz->pwr;
    pwrCurr = thiz->pwr;
    TEST_INTR1_ADD_EVQUEUE_StateM_emC
    int pwrNext = pwrCurr + 1;
    if(pwrNext >= thiz->sizeQueue) { pwrNext =0; } //modulo
    if(pwrNext == thiz->prd) return false;    //RETURN: no more space in queue
    ok = compareAndSet_AtomicInteger(ptr, pwrCurr, pwrNext); //set for next access.
  } while(!ok && --abortCt >=0); //repeat if another thread has changed thiz->pwr.
  if(abortCt <0) {
    //it is a problem of thread workload, compareAndSet does not work.
    THROW_s0n(IllegalStateException, "thread compareAndSet problem", 0, 0);
    return false; //RETURN for non-exception handling, THROW writes a log entry only.
  }
  //the pwrCurr is proper for this entry, another thread may incr thiz->pwr meanwhile
  //but the other thread uses the position after.
  Entry_EvQueue_StateM_vishiaOrg_s* e = &thiz->queue.a[pwrCurr];
  e->hdata.h = hData;
  TEST_INTR2_ADD_EVQUEUE_StateM_emC
    e->evIdent = idEvent;
  return true;
  #endif//NO_PARSE_ZbnfCheader
}
```

Dazu folgende Anmerkungen:

* Das `#ifndef NO_PARSE_ZbnfCheader` dient lediglich der Vereinfachung für das Parsen des Headers für Reflection und Simulink, Internas der inline-Funktionsdefinition sind unwichtig.
* `TEST_INTR1_ADD_EVQUEUE_StateM_emC` ist im Normalfall leer definiert, es ist für den Test gedacht um an dieser Stelle künstlich einen unterbrechenden Aufruf einzupflanzen.

6.2 Auslesen von Events

Das Auslesen von Events aus der Queue geschieht grundsätzlich nur in einem Thread, braucht also die cmpAndSet-Strategie nicht. Aber: Es muss verhindert werden, dass nach der Bereitstellung der Auslesepositon diese bereits wieder überschrieben werden. Daher ist das Auslesen in zwei Stufen realisiert:

```
/**Returns the pointer to the current entry or null if the queue is empty.
 * After evaluation of the referenced content the operation
 * [[quitNext_EvQueue_StateM_vishiaOrg(...)]] has to be invoked to increment the read
 * position respectively to free the entry for writing on end of the queue..
 */
inline Entry_EvQueue_StateM_vishiaOrg_s*
getNext_EvQueue_StateM_vishiaOrg ( EvQueue_StateM_vishiaOrg_s* thiz) {
  if(thiz->prd == thiz->pwr) return null;
  else {
    Entry_EvQueue_StateM_vishiaOrg_s* e = &thiz->queue.a[thiz->prd];
    if(e->evIdent == 0) return null; //the entry is not finished written on add
    else return e;
  }
}
```

Die rückgelieferte Referenz e zeigt also noch auf einen Bereich in der Queue, der noch nicht überschrieben werden kann. Es braucht kein unnötiges lokales kopieren. Die über den Returnwert referenzierten Daten sollten jetzt ausgewertet werden, indem die Event-Operation aufgerufen wird oder in einer Trigger-Instanz die Daten und der Trigger vermerkt werden.

Danach erfolgt der Aufruf von

```
inline void quitNext_EvQueue_StateM_vishiaOrg ( EvQueue_StateM_vishiaOrg_s* thiz
, ThCxt* _thCxt) {
  ASSERT_emC(thiz->prd != thiz->pwr, "faulty quit", thiz->prd, thiz->pwr);
  thiz->queue.a[thiz->prd].evIdent = 0;  //mark as unused first for next wr and rd.
  int32 prd = thiz->prd +1;
  if(prd >= thiz->sizeQueue){ prd -= thiz->sizeQueue; } //wrapping
  thiz->prd = prd;
  thiz->evCt +=1;  //only for debug
}
```

Das `ASSERT_emC(…)` Makro ist leer für das Zielsystem, eine Hilfestellung für Test und Entwicklung (PC-Plattform). Das `thiz->evCt +=1;` ist auch für den Zielsystemeinsatz zielführend. Die Beobachtung eines event-Counter mit Debugmitteln auf dem Zielsystem kann hilfreich sein. Es ist keine wesentliche Rechenzeitbelastung.

6.3 Die Eventqueue-Daten

Die Daten sind vordergründig für den C-Einsatz (kein C++) in einer `struct` definiert. Dabei wird auch auf dem Zielsystem die Benutzung eines C++-Compilers empfohlen (ist kompatibel), aber es ist C. Ein Wrapping für die syntaktisch schönere C++-Umgebung ist möglich.

Zunächst die Definition eines Entry pro Event:

```
typedef struct Entry_EvQueue_StateM_vishiaOrg_T {

  uint32 evIdent;

  /**It is the reference to the data, or maybe only a uint32-data item itself. */
  HandlePtr32Union_emC(ObjectJc) hdata;
} Entry_EvQueue_StateM_vishiaOrg_s;
```

Es wird die `evIdent` gespeichert, und eine zugehöriger Datenreferenz. Die Datenreferenz ist mit einem Spezial-Makro versehen, dass aus der Simulink-Sfunction-Situation heraus entstanden ist aber allgemein verwendbar ist: Anstatt einem 64-bit-Pointer wird ein Handle-Werte als uint32-Type gespeichert, der als Index für eine Adresstabelle verwendet wird. Für ein 32-bit-Zielsystem ist der Handle-Wert dann identisch mit der eigentlichen Speicheradresse, so dass der Zugriff dort optimal schnell ist. Es entstehen hier gleiche Speicherlayouts, was unter Umständen wichtig ist.

```
#ifdef DEF_HandlePtr64
  #define HandlePtr32Union_emC(TYPE) union {uint32 h; }
#else
  #define HandlePtr32Union_emC(TYPE) union {uint32 h; TYPE* p; }
#endif
```

In beiden Fällen werden 32 bit belegt. Für ein 64-bit-System geht es immer auch um ein 8-bit-Alignment, was in der `struct` damit ebenfalls eingehalten wird. Der Zugriff auf diese referenzierten Daten erfolgt dann mit einem Makro `PTR_Handle2Ptr(handle, TYPE)`, dass für die 64-Bit-Variante über eine Routine geht, die auf die Adresstabelle zugreift und den Typ testet und casted. Für ein 32- oder 16-bit-System ist das ein einfaches Makro

```
  #define PTR_Handle2Ptr(handle, TYPE) ((TYPE*) handle)
```

das also sowohl den Pointerwert p oder das Handle h als Input bekommt, beide sind gleich. In den generierten Codes wird `name.h` übergeben da die generierten Codes sich nicht unterscheiden zwischen 64-Bit und Zielsystem. Der `name.p`-Wert ist eher für das debuggen gedacht, um im Schritt-Test die Strukturdaten sofort aufzulösen.

Die Aufbau der EventQueue:

```
typedef struct EvQueue_StateM_vishiaOrg_T {

  union { ObjectJc obj; } base;

  uint16 sizeQueue;
  uint16 sizeInstances;

  int evCt;

  int prd;  //use signed because difference building.

  int pwr; //+rd and wr-pointer

  union{ Entry_EvQueue_StateM_vishiaOrg_s* a; Deb.._Enntry.. .. .._s* dbg; } queue;
```

```
  float Tstep;

  uint evIdForCreator;

  union{ struct EvInstance_StateM_vishiaOrg_T** a;
         DebugArray_EvInstance_StateM_vishiaOrg_s* dbg; } instances;

} EvQueue_StateM_vishiaOrg_s;
```

Die struct ist für die Übersicht ohne Comments dargestellt, siehe `org/vishia/emC/StateM/evQueue.h`

Die `ObjectJc` tritt hier als Basisstruktur auf. Sie enthält für Simulink notwendig die Referenz auf Typdaten und Typinformationen als sogenannte Reflection, auch im Zielsystem für allgemeinen Datenzugriff möglich. Im Zielsystem kann diese `ObjectJc` aber auch auf eine Speichereintrag (32 oder 16 bit) reduziert werden und, für Debuggen günstig, nur eine Ident-Nummer enthalten.

Insgesamt sind die Datenstrukturen so aufgebaut, dass sowohl in 32- als auch 64-Bit-Systemen es immer ein Alignment auf die Speichergrenzen gibt (4 Byte, 8 Byte), damit die struct ohne Alignment-Korrektur vom Compiler genau so abgebildet wird, ‚*packed*‘ vorausgesetzt. Der Speicheraufbau ist auch für 32- und 16-Bit-Systeme optimal. Es ist wichtig, dass bei Speicherabzügen für Debuggingzwecke nicht andere Speicheradressen als erwartet verwendet werden. Es wird `int`, `int16` und `int32` bewusst unterschieden verwendet. In 64-bit-Systemen ist `int` immer 32 bit breit (es ist kein anderes System bekannt, auch wenn ein Cxx- Standard dies nicht so klar definiert).

Die union enthält nur für Debug einen Zeiger auf ein künstliches Array mit 10 Elementen, dass die Beobachtung der ersten Elemente ermöglicht.

```
typedef struct DebugArray_EvInstance_StateM_vishiaOrg_T {
  struct EvInstance_StateM_vishiaOrg_T* a[10];
} DebugArray_EvInstance_StateM_vishiaOrg_s;
```

Diese Struktur wird als Zeiger verwendet und dient lediglich dazu, im Debug-Schritt-Test der IDE mehrere Elemente zu sehen, wenn es ansonsten nur den einfachen Zeigertyp auf ein beliebig groß allokiertes Array gibt, eine bewährte Hilfe, nicht für den Codeablauf selbst verwendet.

6.4 Event-Instanz

Eine Event-Instanz ist mit der Realisierung im Simulink notwendig, da es für die Fblocks, die das Call für Triggered Subsystems bzw. für die Events auslösen, eine Instanz notwendig ist.

Das Prinzip ist dann verallgemeinert: Man kann anstatt direkt mit dem Eventqueue-Auslesen Statemachine-Instanzen zu rufen, die Event-Instanzen informieren, die dann im jeweils an der gewünschten Programmstelle die Statemaschinen aufrufen. Voraussetzung dafür, im Simulink notwendig, anderweitig nicht störend, ist, dass ein Event pro (schneller) Zykluszeit verarbeitet wird.

Die EventQueue schreibt die Instanzen nicht grundsätzlich vor, sie unterstützt sie aber.

Das Ausräumen der Event-Queue sieht mit den Instanzen wie folgt aus:

```
bool checkForListener_EvQueue_StateM_vishiaOrg ( EvQueue_StateM_vishiaOrg_s* thiz
                                                , ThCxt* _thCxt){
  Entry_EvQueue_StateM_vishiaOrg_s* ev;
  //Note: dequeue only one event.
  if( (ev = getNext_EvQueue_StateM_vishiaOrg(thiz)) !=null) {
    if(ev->evIdent <= thiz->sizeInstances) {
      EvInstance_StateM_vishiaOrg_s* evInstance = thiz->instances.a[ev->evIdent];
      if(evInstance !=null) {
        evInstance->hdata.h = ev->hdata.h; //The current data to the event, .
        evInstance->stateTrg = 1;          //notify event is given
        //The calling of the event is done on execution of the instance,
      }
    }
    quitNext_EvQueue_StateM_vishiaOrg(thiz, _thCxt);
    return true;
  }
  else {
    return false; //no event in queue.
  }
}
```

Hier sind die in 6.2 Auslesen von Events, Seite 17 vorgestellten Operations `getNext_EvQueue_StateM_vishiaOrg(…)` und `quitNext_EvQueue_StateM_vishiaOrg(…)` verwendet. Dazwischen wird lediglich ein Markierungsbit in der Instanz gesetzt.

Im Simulink wird in einer Sfunction folgendes ausgeführt (gekürzt):

```
static void mdlOutputs(SimStruct *simstruct, int_T tid) { …..
        uint32 _ret_ = hasEvent_EvInstance_StateM_vishiaOrg(thiz, hdata_y, ctEv_y);

        if(_ret_) {
          if(!ssCallSystemWithTid(simstruct, 0, tid)) {
            return; //error occured, will be reported by the simulink engine.
        } }
```

Es wird also die Event-Instanz geprüft, ob sie aktiv ist. Daraufhin wird im Simulink das *Triggered Subsystem* aktiviert was dem Aufruf der Statemachine entspricht. Wollte man adäquates in einem schnellen Interrupt in C programmieren, dann würde an dieser Stelle der Statemaschinenaufruf mit dem Event stehen, der ebenfalls zeitschnell durchlaufen werden kann. Die Statemaschinenaufrufe erfolgen also jeweils mit Test eines Events, nicht an der Queue. Das hat einen Vorteil: Die Queue ist zentral, möglicherweise nicht nur für den einen Interrrupt. Im Interrrupt wird möglicherweile sowieso nur ein bestimmtes Event getestet. Dann ist man hiermit fertig. Ansonsten braucht man für jeden

Thread, der Statemaschinen aufruft, einen eigene Queue. Dann muss aber der Sender der Events wissen, in welche Queue er schreibt. Gibt es eine zentrale Queue, dann muss es einen Verteilungsmechanismus geben, oder eben nur einen Thread, der für alle Statemachines zuständig ist. Das ist eine häufige Lösung für Multithreading-Systeme, aber nicht unbedingt für Interruptgesteuerte Systeme geeignet.

Die Routine die das Event testet sieht wie folgt aus:

```
/**Returns the hdata if an event was applied
 * after calling [[checkForListener_EvQueue_StateM_vishiaOrg(...)]].
 * A further call returns 0.
 * @return 0 if it has not an event.
 * @simulink Operation-FB, fnCallTrg.
 */
inline uint32 hasEvent_EvInstance_StateM_vishiaOrg (
  EvInstance_StateM_vishiaOrg_s* thiz, uint32* hdata_y, int32* ctEv_y) {
  if(thiz->stateTrg ==1) {
    thiz->stateTrg = 0;
    thiz->ctEvents +=1;
    if(hdata_y !=null) {
      *hdata_y = thiz->hdata.h;
    }
    if(ctEv_y !=null) {
      *ctEv_y = thiz->ctEvents;
    }
    return thiz->hdata.h;
  } else {
    return 0;
  }
}
```

Der Returnwert ist das Handle auf die Daten, eventuell `-1=0xffffffff` wenn es keine Daten gibt, also nicht 0, oder 0 wenn kein Event vorliegt. Das ist hier polling, für Simulink notwendig, aber auch für Abarbeitung im Interrupt notwendig. Denn: Das Interruptprogramm kann nur „nachschauen“. Kurze Anmerkung zu Polling: In manchen Ausführungen gelangt man zu dem Schluss, Polling sei ein ständiges Nachschauen in einer Schleife. Das ist natürlich Unsinn. Wann nachgeschaut wird, bestimmt der Nachschauende. Im Interrupt ist das einmal an einer Stelle. Sie schauen zuhause ja auch nicht ständig in den Briefkasten.

6.5 Exceptionhandling, zielsystemunabhängige Programmieurng, C++

Die gesamte hier verwendete Programmierung ist nicht auf ein bestimmtes Zielsystem abgestimmt sondern für alle Zielsysteme optimiert compilierbar. Dafür steht das Kürzel emC: embedded multiplatform C/++. Der Schlüssel hierzu ist:

* Es gibt wenige Headerfiles, die zielsystemspezifisch ausgelegt sind: `compl_adaption.h`. Über den Include-Path bei der Compilierung wird der richtige Header ausgewählt. Dort stehen wenige Definitionen, die mit der Plattform zu tun haben, nicht mit Applikationsspezifika. Damit erfolgt die Anpassung. Beispielsweise gibt es Prozessoren, DSPs von Analog Devices, die nur datenwortweise in 32-bit-Schritten die Adressen zählen, folglich eigentlich auch keine 16-bit-Speicherbreiten unterstützen. Die passende Umsetzung befindet sich in der `compl_adaption.h` auf den Zielprozessor abgestimmt.
* Es gibt einen Headerfile und daraus abgeleitet bestimmte dort includierte Headerfiles, die Applikationsspezifika definieren: `applstdef_emC.h`. Dort wird definiert, wie `ObjectJc` realisiert werden soll und dergleichen.
* In Anwenderfiles werden außer C-Standard-Header, die jedes System kompatibel anbieten muss, niemals betriebssystem- oder compilerspezifische Header eingebunden, also etwa `Winbase.h`. Es erfolgt daher auch keine Unterscheidung `#ifdef GNU, #ifdef Windows` oder dergleichen.
* Für Systemzugriffe sind Header im Bereich emC/OSAL einheitlich definiert, deren Implementierungen systemspezifisch dann auf die eigentlichen Betriebssystemfunktionen gehen. Damit sind Threads anlegbar etc. in einheitlicher Weise im Anwenderraum. Selbst bei Filezugriffen, die in C eigentlich als ‚standardisiert' erscheinen, ist der Weg über die OSAL-Adaption angemessen: Einerseits gibt es den Slash versus Backslash, andererseits gibt es Spezialfilesysteme im Embedded Bereich, dritterseits gibt es unterschiedliche Strategien der Betriebssysteme für File-Locking usw.

Eine interessante oft kontroverse oder dann ablehnend diskutiert ist das Exceptionhandling.

Grundsätzlich sollte man sich darüber verständigen, dass Exceptions nicht für den Normalfall sein sollten, auch in Java, C#, C++ sondern nur für wirkliche Ausnahmen. `FileNotFoundException` dürfte es nicht geben, denn dass ein File nicht vorhanden ist, kann normal sein. `ArrayIndexOutOfBoundsException` oder `NullPointerException` ist dagegen eine wirkliche Ausnahme, die eigentlich nur bei noch vorhandenen Programmierfehlern auftreten dürfte.

Erwartbare Fehlerfälle (File not found) müssen korrekt in der Applikation behandelt werden.

Für komplexe Software in C++, C#, Java und dergleichen gilt dann mit Exception die Regel: Eine nicht erwartete Exception beendet eine aufgerufene Funktion, diese geht nicht im konkreten Anwendungsfall, kann mit anderen Daten ggf. erneut gestartet werden, aber die Applikation läuft weiter.

Für komplexe Software im Embedded-Bereich gilt häufig: Es dürfen keine Fehler auftreten, wenn die Software so wie getestet verwendet wird. Wenn Fehler auftreten, dann kann es einen automatischen Neustart beispielsweise eines Steuergerätes geben, so dass der gesamte Ablauf zwar mit Verlusten, aber weiterhin funktioniert. Die Fehler werden erfasst und mit folgenden Softwareversionen oder Gerätegenerationen zur Beseitigung anempfohlen oder beauftragt, wenn dies finanziell zweckmäßig erscheint.

Daher wird in Embedded Systemen, insbesondere im C-Bereich, häufig auf die Verwendung des Exceptionhandlings verzichtet.

Müssen Softwarelösungen mit und ohne Exceptionhandling nun grundsätzlich unterschiedlich programmiert werden? Antwort NEIN.

Die Lösung ist folgende:

* Exceptionhandling wird in Makros gekleidet, das origniale C++ `try, catch, throw` wird nicht direkt verwendet. Die Makros heißen `TRY, END_TRY, CATCH, THROW_s0` und dergleichen, siehe `org/vishia/emC/Base/ExcThCxtBase_emC.h` und über verschiedene Ausprägungen der `applstdef_emC.h` gerufene spezielle Definitionen.
* Je nach Ausprägung wird das `THROW` auf ein C++-`throw`, auf einen `longjmp` der im C-Bereich gute Dienste leistet, oder auf eine möglicherweise sehr einfache Log-Ausgabe definiert.
* Für die letzte Variante, kein Exceptionhandling mit throw oder longjmp, muss der Algorithmus sicher weiterlaufen, möglicherweise ein zweckdienliches return ausführen oder dergleichen. Diese Variante ist immer zu programmieren, wenn die Software in solchen Umgebungen eingesetzt werden soll. Das bedeutet immer für allgemeine Routinen.
* Beispielsweise beim Einschreiben von Events: Im Fehlerfall werden diese halt nicht eingeschrieben, schlimmer ist es nicht.
* Für den Nicht-Exceptionjhandling-Fall ist das Auftreten des Fehlers protokolliert mit der über das THROW ausgelösten Logmeldung. Es reicht dazu, etwas in einen Speicher zu schreiben, der irgendwann ausgelesen wird. D.h. es ist neben der Tatsache, dass da irgend etwas nicht funktioniert, das Auftreten des konkreten THROW auslesbar.
* Man kann dann die Software in einer PC-Umgebung mit Excpetionhandling testen, bei der Entwicklung diverse Fehler feststellen, ausbauen so gut wie gefunden, und dann ohne Codeänderungen mit der THROW-Log-Methode ausliefern. Da bei der Entwicklung das leistungsfähige Exceptionhandling genutzt werden konnte, dürfte die damit erzielbare Fehlerrate niedrig sein.

Zur Unterstützung des Exceptionhandling gibt es die STACKTRACE-Makros, die ebenfalls verminderbar sein, aber in einer leistungsfähigen Umgebung gute Dienste zur Eingrenzung des Fehlers leisten. Dazu ist ein sogenannter ThreadContext notwendig, der selbst bei einfachen Prozessoren bereitgestellt werden kann, mit Umschaltung eines zentralen Zeigers zwischen Hintergrundschleife und Interrupts. Der ThreadContext ist ein Speicherbereich, der threadspezifisch ist. Für Multithread-OS wird dieser mit der Thread-ID und einer Tabelle unterstützt, für einfache Systeme wie eben dargestellt mit einer Zeigerumschaltung am Anfang und Restaurierung am Ende einer Interruptbehandlung.

Folgende Strategie benutze ich für die Verwendung von C++:

* Alle Quellen in .c sollen zwar mit einem Standard-C99-Compiler für C übersetzbar sein, auf dem PC verwende ich aber immer einen C++-Compiler, GNU (mit QT) oder Visual Studio ab 2015.
* D.h. in C-Language wird nur die Schnittmenge zwischen C und C++ verwendet. Es gibt wenige Dinge, die in C gehen und in C++ nicht, die braucht man in der Regel nicht. Wichtig ist dies zu detektieren.
* Datenstrukturen werden jedenfalls als C-struct mit typedef definiert und sind damit im C-Einsatz vom Memory-Layout wohl definiert.
* C++ hat dann Voteile, weil die Schreibweise für Methodenaufrufe etwas kürzer ist, weil es überladene Operatoren gibt usw.

* Es ist dafür möglich, eine C-Struktur ein einer class zu vererben:

```
class EvQueue_StateM_vishiaOrg : public EvQueue_StateM_vishiaOrg_s {
  public bool add(uint16 idEvent, uint32 hData) {
    return add_EvQueue_StateM_vishiaOrg(this, idEvent, hData);
  }
  …...
```

(...für die EvQueue derzeit noch nicht erledigt)

Die Methoden sind gecovert, rufen also intern direkt die C-Operationen auf. Die Ableitung `this` auf den struct-Typ macht der Compiler automatisch.

* Damit kann eine Applikation, die C++ verwenden möchte, die schönere Syntax nutzen.
* Bezüglich Interfaces, Functionscall gibt es in emC eine Lösung im reinen C, die pro dynamischer Operation etwas länger braucht weil Zusatztests ausgeführt werden. Solche dynamischen Operationen sind aber nicht typisch für fast realtime. Bei der Nutzung der virtual Operations, Mehrfachvererbung etc. bin ich selbst etwas vorsichtig. Ein Softwarefehler, der class-Instanzdaten zerstört, kann in C++ zum Absturz führen. In C gibt es nur Datenfehler, wenn ein Function-Pointer nicht in den Daten direkt steht und der Aufruf über Function-Pointer eben gesichert wird. Das ist ein Beitrag für SIL-Programmierung.
* Standard-Template-Library usw. ist für Embedded eher nichts, es wird zuviel new aufgerufen.

Namespaces, lange include-paths

In C sind die Namespaces aus C++ nicht verwendbar (?neuere C-Standards),

Für Includes habe ich mir angewöhnt, immer lange Pfade zu schreiben, also eigentlich

```
#include <org/vishia/emC/StateM/evQueue.h>
```

wobei ich im Moment noch beim Umbauen bin, das `org/vishia/` davor zu setzen. Bisher nur ab `emC/`. Das ist die gleiche Strategie wie in **Java** mit den Package-Paths, die sich über Jahrzehnte bewährt hat, unverändert seit Beginn. Es hat sich etabliert, dass ein package-Path immer mit der umgekehrten URL des Herstellers beginnt, damit ist dies weltweit eindeutig. Bei Firmenübernahmen oder Abteilungsumbenennungen werden die Package-Paths nicht geändert. Daher gibt es noch `com.sun…`

Das gilt außer den oben erwähnten `applstdef_emC.h` und `compl_adaption.h`, die eben im Includepath spezifisch gefunden werden müssen. Mit dieser Lösung ist das includen clashfrei.

Für die Benennung global sichtbarer Identifier benutze ich die gleiche Strategie als Suffix, von früher her oft nur _Jc als Endung (aus einem *CRuntimeJavalike*-projekt aus 200x heraus), dann `_emC` und konsequenterweise hier `_Package_vishiaOrg`. Damit sind clashs bei Identifier ebenfalls vollständig vermieden.

Die C-Operationen enden immer mit dem Suffix der zugehörigen struct-Ident, sind damit eindeutig, auch Objektorientiert weil den Daten zugeordnet.

Für alle struct soll ein suffix `_s` verwendet werden, `_T` für den struct-tag-Namen (nicht überall konsequent realisiert bisher), so dass die zugehörige class den einfachen Stammnamen haben kann, siehe obiges class-Beispiel. Die Reflectiongenerierung und Simulink-Sfunction-Generierung ist darauf abgestimmt.

*****