Diese Anleitung gibt es auch online unter http://emaps.de.vu/lua/taskmemorizer. Die vorliegende PDF-Version hat gegenüber der Online-Version zwei Nachteile:

Die Code-Beispiele lassen sich aus der PDF nicht vernünftig rauskopieren (es geht schon, aber die Einrückungen verschwinden dabei; außerdem kann beim Kopieren über mehrere Seiten hinweg der Text der Fußzeile mitkopiert werden). In der Online-Version lassen sich alle Code-Beispiele komplett und problemlos kopieren.

Weiterhin gibt es in der Online-Version zu allen (oder zumindest den meisten) englischen Begriffen eine deutsche Übersetzung im Tooltip – dies ist in der PDF technisch nicht möglich.

TaskMemorizer_BH2

Der „TaskMemorizer“ ist ein Lua-Modul, das einfach in eigene Anlagen eingebunden und konfiguriert werden kann. Entgegen der ersten Vermutung kümmert sich der TaskMemorizer nicht um das eigentliche Merken von Aufgaben; dafür sind Zustandssignale erforderlich. Der TaskMemorizer übernimmt aber die regelmäßige Abfrage dieser Zustandssignale und führt bei Bedarf eine entsprechende Funktion aus.

Die Konfiguration ist für den Standardfall sehr einfach gehalten, wer es etwas individueller mag, kann auch durch das Hinzufügen von ein paar weiteren Parametern alles selbst konfigurieren.

Schnellstartanleitung

1. Installation

Nach dem Download die zip-Datei entpacken und die `Installation.eep` aufrufen, oder die `TaskMemorizer_BH2.lua` von Hand ins EEP-Verzeichnis in den Unterordner `LUA` kopieren.

2. Einbinden

Achtung! Vor diesem Schritt **unbedingt die aktuelle Anlage speichern**. Wenn das angegebene Skriptfile nicht gefunden wird (weil es nicht existiert, oder man sich verschrieben hat), kann es zum **Absturz von EEP** kommen!

Füge diese Zeile an den Anfang des Anlagen-Skripts ein:

```
1  TaskMemorizer = require("TaskMemorizer_BH2")
```

Hat das Einbinden geklappt, erscheint nach dem Neuladen des Skripts im EEP-Ereignis-Fenster der Hinweis `TaskMemorizer_BH2 v0.2 wurde eingebunden`.

3. Initialisieren

Alle Aufgaben (im Folgenden werde ich der Einfachkeit halber das englische Wort „Task“ verwenden) werden in einer einzigen Variablen gespeichert. Wie du sie nennst, bleibt dir überlassen. Der Übersichtlichkeit halber empfehle ich jedoch den Namen `Tasks`.

Folgender Code kommt (natürlich in angepasster Form, siehe unten unter „Konfiguration“) irgendwo in den Rumpf des Anlagen-Skripts (nicht in die `EEPMain`-Funktion):

```
1  Tasks = TaskMemorizer:new{
2    {
3      name="Test",
4      interval=30,
5      signalID=60,
```

```
    func=function() print("Getestet") return true end,
  },
}
```

Je nach Anzahl der Aufgaben kann dieser Code-Teil natürlich auch viel größer sein.

4. Verwenden

Jetzt muss nur noch dafür gesorgt werden, dass die Aufgaben auch regelmäßig abgearbeitet werden. Dazu muss nur noch eine Code-Zeile in die Funktion `EEPMain()` eingetragen werden:

```
Tasks:exec()
```

Die gesamte `EEPMain`-Funktion sollte dann also in etwa so aussehen:

```
function EEPMain()
--  anderer Code
    Tasks:exec()
    return 1
end
```

Weiterhin gibt es noch die Funktion `Tasks:print()`, die den Status aller Aufgaben im Ereignis-Fenster ausgibt. Dies kann vor allem zur Kontrolle der Konfiguration sinnvoll sein.

Die Ausgabe von `Tasks:print()` kann beispielsweise so aussehen:

```
8 registrierte Aufgaben: (aktuelle Sekundenzahl: 11958)
1 | Test (ID: 91, Pos: 1, Abfrage alle 10 Sekunden, nächste in 7 Sekunden)
  | Test2 (ID: 92, Pos: 1, Abfrage alle 60 Sekunden, nächste in 42 Sekunden)
  | SbfEinfahrtLinks (ID: 143, Pos: 2, Abfrage alle 10 Sekunden, nächste in 2 Sekunden)
X | SbfEinfahrtRechts (ID: 144, Pos: 2, Abfrage alle 10 Sekunden, nächste in 2 Sekunden)
...
```

Die erste Zeile zeigt die Gesamtzahl der registrierten Aufgaben und die aktuelle Sekundenzahl der EEP-Zeit - daran orientieren sich die Intervalle und Verschiebungen.

Die folgenden Zeilen stehen für jeweils eine Aufgabe. Ganz links steht eine "Kurzinfo" zum Status: Bei als Zähler definierten Aufgaben steht hier der aktuelle Zählerstand, ansonsten ein X für „Task muss noch erledigt werden“ oder nichts für „Task erledigt, nichts zu tun“. Dahinter sind noch jeweils der Name des Tasks, die ID des zugehörigen Signals inkl. der aktuellen Stellung sowie das Abfrage-Intervall und die verbleibende Zeit bis zur nächsten Abfrage aufgeführt.

Konzept

Der TaskMemorizer hat eine ähnliche Aufgabe, wie sie früher von Schaltkreisen übernommen wurde: Dauerndes Abfragen, ob irgendeine Aufgabe ansteht. Durch den Einsatz von Lua kann das natürlich gezielter geschehen.

Die „Task-Liste“ ist ein Array (also eine automatisch durchnummerierte Tabelle) von Einzel-Tasks, die wiederum durch eine Tabelle mit verschiedenen benannten Einträgen repräsentiert werden. Eine genaue Erklärung aller Einträge gibt es weiter unten unter „Konfigurationsmöglichkeiten“.

Durch den Aufruf der Funktion `TaskMemorizer:new()` werden fehlende Einträge mit Standardwerten initialisiert und ungültige Tasks (fehlende Signal-ID bzw. Funktion) komplett rausgeworfen (mit Hinweismeldung im Ereignis-Fenster). Außerdem wird die Task-Tabelle hier auch um die Funktionalität (also `exec()` und `print()`) ergänzt.

Die Funktion `EEPMain()` wird von EEP automatisch fünfmal pro Sekunde aufgerufen, und somit auch die (hoffentlich) darin notierte Funktion `Tasks:exec()`. Da sich dieses „fünfmal pro Sekunde“ aber auf die echte Zeit, und nicht auf die EEP-Zeit bezieht, wird erstmal geprüft, wieviele EEP-Sekunden seit dem letzten Aufruf der Funktion vergangen sind (im Zehnfach-Zeitraffer wären es durchschnittlich zwei), und für jede vergangene Sekunde wird die Task-Liste einmal durchgearbeitet.

Für jeden Task wird nun geprüft, ob die „Ausführungszeit“ gekommen ist (mathematisch ausgedrückt: (aktuelle Zeit) modulo (Prüfungs-Intervall) = (Prüfungs-Offset)), und ob das angegebene Signal noch nicht auf „Task erledigt“ steht. Außerdem wird geprüft, ob der Task nicht in der temporären Ausschlussliste (siehe unten) steht. Sind alle drei Bedingungen erfüllt, wird die angegebene Funktion aufgerufen. Als Parameter wird die Tabelle des aktuellen Tasks übergeben. Diese muss von der Funktion nicht verarbeitet werden, kann aber. Wenn man die Task-Tabelle um zusätzliche Einträge ergänzt, kann so z.B. eine Funktion von mehreren Tasks verwendet werden, aber ihr Verhalten an den aktuellen Task anpassen.

Wenn die aufgerufene Funktion nicht ausdrücklich `false` zurückgibt (was soviel bedeutet wie „Ich habe zwar versucht, den Task auszuführen, das hat aber nicht geklappt. Versuche es bitte später nochmal“), wird das zum Task gehörende Zustandssignal auf „Task erledigt“ gesetzt, oder, falls es sich um einen Zähler handelt (`isCounter=true`), wird dieser 1 runtergezählt.

Achtung: Weil der TaskMemorizer immer alle seit dem letzten Aufruf vergangenen Sekunden abarbeitet, kann es sein, dass das Skript nach dem Ändern der EEP-Zeit erstmal für einige Sekunden beschäftigt ist.

Konfigurationsmöglichkeiten

Hier erstmal eine Übersicht über alle möglichen Schlüssel in der Task-Tabelle, die in der Funktion `TaskMemorizer:new(...)` unterstützt, verarbeitet oder ergänzt werden:

```
Tasks = TaskMemorizer:new{
  {
    name="",
    signalID=1,
    func=IrgendeineFunktion,
    interval=60,
    offset=0,
    signalPosDone=2,
    isCounter=false,
    exclude={},
  },
  {
    signalID=2,
    func=IrgendeineAndereFunktion,
  },
}
```

Kleiner Hinweis am Rande: LUA ist es egal, ob man `TaskMemorizer:new({...})` schreibt, oder die runden Klammern einfach weglässt: `TaskMemorizer:new{...}`

Die gesamte `Tasks`-Tabelle besteht aus einem Array von einzelnen Task-Tabellen. Jeder dieser Task-Tabellen kann folgende Einträge haben:

- `name`: Name des Tasks. Wird zur Identifizierung auszuschlißender Tasks und bei der Ausgabe der Taskliste mittels `Tasks:print()` benötigt. Kann weggelassen werden, Standardwert ist `""`

(leerer String)

- `signalID`: Signal-ID des Zustandssignals, das speichert, ob der aktuelle Task aktiv ist und abgearbeitet werden muss. Pflichtangabe; fehlt diese, wird der komplette Task aus der Task-Liste entfernt.
- `func`: Funktion, die zur Abarbeitung des Tasks aufgerufen wird. Als Parameter bekommt sie die komplette Tabelle des aktuellen Tasks übergeben. Liefert diese Funktion `false` zurück, wird der Task (bzw. das zugehörige Signal) nicht zurückgesetzt, ansonsten schon. Hier kann entweder eine „anonyme Funktion“ (ohne eigenen Namen) direkt definiert werden, oder es wird der Name einer woanders definierten Funktion angegeben (ohne runde Klammern; hier ist die Funktion selbst gefragt, nicht das Ergebnis des Funktionsaufrufs). Beispiele dazu gibt es unten unter „Beispielkonfigurationen“. Pflichtangabe; ist keine gülte Funktion angegeben, wird der komplette Task aus der Task-Liste entfernt.
- `interval`: Abstand in Sekunden, wie häufig geprüft werden soll, ob dieser Task abgearbeitet werden muss. Kann weggelassen werden, Standard-Wert ist `60`.
- `offset`: Verschiebung der Abarbeitungs-Prüfung in Sekunden. Beispiel mit `interval=30`: Ist der `offset=0`, so wird immer zu den Sekunden 0 und 30 geprüft. Ist der `offset=10`, so wird immer zu den Sekunden 10 und 40 geprüft. Kann weggelassen werden, Standardwert ist `0`.
- `signalPosDone`: Signal-Position, die für „Task erledigt“ steht. Alle anderen Signal-Positionen gelten als „Task muss noch erledigt werden“. Nach erfolgreicher Abarbeitung des Tasks wird das Signal in diese Position gebracht (wenn es kein Zählersignal ist, dann wird einfach eins runtergezählt). Kann weggelassen werden, Standard-Wert ist `2` („Halt“) bzw. `1` bei Zählersignalen („00“).
- `isCounter`: Gibt an, ob das Signal als Zählersignal behandelt werden soll. Zu den Auswirkungen siehe eins höher (`signalPosDone`). Kann weggelassen werden, Standard-Wert ist `false` (normales Signal, kein Zählersignal).
- `exclude` (ab v0.2): Eine Liste von Tasknamen, die temporär ausgeschlossen werden sollen, sobald dieser Task erledigt wurde. Dieser temporäre Ausschluss gilt immer für einen kompletten Aufruf von `Tasks:exec` bzw. von `EEPMain`. In dieser Zeit werden von EEP keine Signalstellungen aktualisiert, eine erneute Abfrage eines zuvor (von einem anderen Task) gestellten Signals kann also eine Fehlinformation liefern. Um dies zu verhinden, kann diese erneute Abfrage ausgeschlossen werden, indem der Signal-abfragende Task in die Ausschluss-Liste des Signal-stellenden Tasks eingetragen wird. Ein Anwendungsbeispiel findet sich unten in der Beispielkonfiguration. Achtung: Ausgeschlossene Tasks werden nicht automatisch wiederholt, sondern erst im nächsten Taktzyklus (definiert durch `interval` und `offset`) erneut abgefragt. Kann weggelassen werden, Standard-Wert ist `{}` (leere Liste, nichts wird ausgeschlossen).

Wer Platz sparen will, kann die Einträge für einen Task natürlich auch in eine Zeile schreiben (siehe auch die Beispiele unten).

Beispielkonfigurationen

Nachfolgend ein paar Beispielkonfigurationen (die natürlich eine entsprechende Anlage mit diesen Signalen voraussetzen).

```
1  function Test(task)
2    local zs=EEPGetSignal(task.signalID)
3    print("Zählerstand: ",zs-1,"->",zs-2)
4  end
```

```
5
6  Tasks = TaskMemorizer:new{
7    {name="Test",interval=10,offset=15,signalID=91,func=Test,isCounter=true},
8    {name="Test2",interval=60,signalID=92,signalPosDone=1,func=function()
9       print("Ampel stand auf rot")
10      return true
11    end},
12   {name="TestOhneFunktion",interval=10,signalID=39,func=Test3,isCounter=true},
13   {name="TestOhneSignal",interval=10,func=Test},
14   {name="SbfEinfahrtLinks",interval=10,signalID=143,func=function()
15     if EEPGetSignal(89)==1 and sbf:einfahrt() then
16       EEPSetSignal(89,2)
17       EEPSetSignal(63,1)
18       return true
19     else return false end
20   end,exclude={"SbfEinfahrtRechts"}},
21   {name="SbfEinfahrtRechts",interval=10,signalID=144,func=function()
22     if EEPGetSignal(89)==1 and sbf:einfahrt() then
23       EEPSetSignal(89,2)
24       EEPSetSignal(64,1)
25       return true
26     else return false end
27   end,exclude={"SbfEinfahrtLinks"}},
28   {name="SbfAusfahrtLinks",interval=10,signalID=146,func=function()
29     if EEPGetSignal(90)==1 and sbf:ausfahrt() then
30       EEPSetSignal(90,2)
31       EEPSetSwitch(61,2)
32       return true
33     else return false end
34   end,exclude={"SbfAusfahrtRechts"}},
35   {name="SbfAusfahrtRechts",interval=10,signalID=145,func=function()
36     if EEPGetSignal(90)==1 and sbf:ausfahrt() then
37       EEPSetSignal(90,2)
38       EEPSetSwitch(61,1)
39       return true
40     else return false end
41   end,exclude={"SbfAusfahrtLinks"}},
42   {name="SbfNebenbahnEinfahrt",interval=10,signalID=149,func=function()
43     if sbfNebenbahn:einfahrt() then
44       EEPSetSignal(40,1)
45       return true
46     else return false end
47   end},
48   {name="SbfNebenbahnAusfahrt",interval=10,signalID=148,func=function()
49     return sbfNebenbahn:ausfahrt()
50   end},
51 }
```

Jeder Task-Eintrag ist ein Beispiel für sich und wird hier nochmal separat erklärt:

```
1  function Test(task)
2    local zs=EEPGetSignal(task.signalID)
3    print("Zählerstand: ",zs-1,"->",zs-2)
4  end
```

```
7  {name="Test",interval=10,offset=15,signalID=91,func=Test,isCounter=true},
```

Hier ist das Signal mit der ID 91 ein Zählersignal, dass alle 10 Sekunden abgefragt wird (zur Sekunde 5, 15, 25, 35, 45 und 55; `offset=15` ist gleichbedeutend mit `offset=5`). Wenn das Signal nicht auf 0 steht, wird die Funktion `Test` aufgerufen, die woanders definiert wurde (vorher, bei `func=Test` muss `Test` schon definiert sein!). Diese Funktion nutzt die Informationen des

übergebenen Tasks, und schreibt eine kurze Info ins Ereignis-Fenster, nämlich den alten (aktuellen) und neuen Zählerstand (nach der automatischen Dekrementierung).

```
 8  {name="Test2",interval=60,signalID=92,signalPosDone=1,func=function()
 9    print("Ampel stand auf rot")
10    return true
11  end},
```

Hier wird eine Ampel (Signal-ID 92) zu jeder vollen Minute überprüft. Steht sie auf Rot, wird einfach „Ampel stand auf rot“ im Ereignis-Fenster ausgegeben. Hier wurde die Funktion direkt an Ort und Stelle definiert.

```
12  {name="TestOhneFunktion",interval=10,signalID=39,func=Test3,isCounter=true},
13  {name="TestOhneSignal",interval=10,func=Test},
```

Hier sind zwei Beispiele für unvollständige Tasks, die automatisch aus der Taskliste entfernt werden. Der erste Task „TestOhneFunktion“ hat zwar einen Funktionsnamen angegeben. Da dieser aber nirgendwo definiert wurde, kann er auch nicht aufgerufen werden. Der zweite Task „TestOhneSignal“ hat zwar eine gültige Funktion angegeben (siehe oben), aber keine Signal-ID.

```
14  {name="SbfEinfahrtLinks",interval=10,signalID=143,func=function()
15    if EEPGetSignal(89)==1 and sbf:einfahrt() then
16      EEPSetSignal(89,2)
17      EEPSetSignal(63,1)
18      return true
19    else return false end
20  end,exclude={"SbfEinfahrtRechts"}},
```

Dieser Task (vom Schattenbahnhof meiner Anlage „Bahnhof Falls“ übernommen) kümmert sich um Züge, die von der linken Seite in meinen Schattenbahnhof einfahren wollen. Das Zustands- bzw. Anforderungssignal hat hier die ID 143. Signal 89 ist ein weiteres Zustandssignal, das anzeigt, ob nicht gerade schon ein Zug in den Schattenbahnhof einfährt (z.B. von der rechten Strecke). Steht dieses in Stellung 1 („Fahrt“) und ist eine Einfahrt in den Schattenbahnhof möglich, dann wird die Einfahrt für folgende Züge gesperrt (`EEPSetSignal(89,2)`). Anschließend wird das Einfahrsignal (ID 63) auf Fahrt gestellt und `true` zurückgegeben, womit dieser Task als erledigt gilt. Gleichzeitig wird der Task „SbfEinfahrtRechts“ in die temporäre Ausschluss-Liste aufgenommen, um zu verhindern, dass dieser das noch nicht umgestellte Zustandssignal abfragt. Waren die Ausgangsbedingungen nicht erfüllt (Einfahrstrecke blockiert oder Einfahrt in Schattenbahnhof nicht möglich), so gibt die Funktion `false` zurück, wodurch der Task weiterhin als „noch zu erledigen“ gilt.

Die Tasks „SbfEinfahrtRechts“, „SbfAusfahrtLinks“ und „SbfAusfahrtRechts“ funktionieren im Prinzip genauso, hier werden lediglich andere Signale angesprochen. Daher wird hier nicht nochmal gesondert darauf eingegangen. Bei der Ausfahrt muss allerdings kein Einfahrsignal auf Fahrt gestellt werden, stattdessen wird eine Weiche gestellt, sodass die Züge in die richtige Richtung fahren.

```
42  {name="SbfNebenbahnEinfahrt",interval=10,signalID=149,func=function()
43    if sbfNebenbahn:einfahrt() then
44      EEPSetSignal(40,1)
45      return true
```

```
46    else return false end
47  end},
```

Die Einfahrt in den Schattenbahnhof der Nebenstrecke gestaltet sich deutlich einfacher: Hier müssen keine konkurrierenden Strecken geprüft werden, sondern nur auf die „Einfahrterlaubnis“ des Schattenbahnhofs gehört werden. Wird diese erteilt, wird das Einfahrsignal auf Fahrt gestellt und `true` zurückgegeben, sodass der Task als erledigt gilt.

```
48  {name="SbfNebenbahnAusfahrt",interval=10,signalID=148,func=function()
49    return sbfNebenbahn:ausfahrt()
50  end},
```

Noch einfacher ist die Ausfahrt aus dem Schattenbahnhof: Hier gibt es keine weiteren Aufgaben zu erledigen. Ergibt die Schattenbahnhofs-Ausfahrt-Funktion `true`, ist die Ausfahrt erledigt, ansonsten nicht. Daher könnte man diese Funktion auch direkt verwenden, folgende Zeile hätte also genau den selben Effekt (vorausgesetzt, die Funktion `sbfNebenbahn:ausfahrt` ist zu diesem Zeitpunkt schon definiert):

```
48  {name="SbfNebenbahnAusfahrt",interval=10,signalID=148,func=sbfNebenbahn:ausfahrt},
```

Changelog

v0.2 vom 14.12.2014:
- Temporäres Ausschließen von Tasks, die auf die gleichen Zustandssignale zugreifen

v0.1 vom 03.08.2014:
- Abarbeitung von wiederkehrenden Aufgaben mit Anforderung über Zustandssignale
- Einfache Konfiguration mit vielen Konfigurationsmöglichkeiten
- Es werden sowohl einfache Zustandssignale („ja“/„nein“) als auch Zählersignale unterstützt

So, ich hoffe, dass nun alles wichtige gesagt ist, und ich nichts vergessen habe. Natürlich freue ich mich über Lob, Kritik, Anregungen oder Skriptwünsche. Ihr könnt mir entweder eine E-Mail an benjamin.hogl@gmx.de schreiben, oder im MEF (http://www.eepforum.de) in meiner Konstrukteurssprechstunde posten.
Auch ein Blick auf meine Homepage (http://emaps.de.vu) dürfte sich hin und wieder lohnen...

Viel Spaß mit dem Skript wünscht

Benny (BH2)