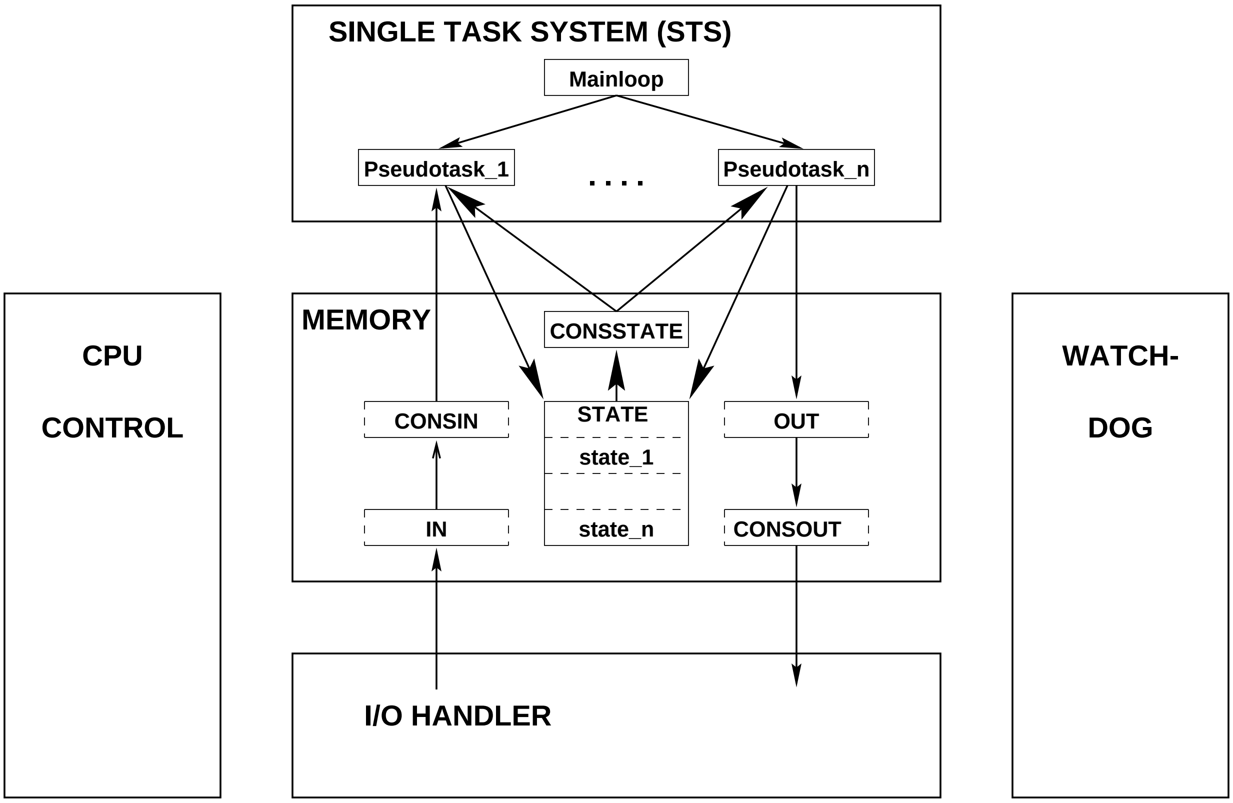
STS – Single Task System[1]

1 Architektur

STS ist ein sehr einfaches System zur Realisierung von pseudo-paralleler Verarbeitung mehrerer Tasks. Die Grundidee ist, daß die Tasks als Funktionen (z. B. C-functions) implementiert werden, welche selber die Verarbeitung nach einer festen Zeitspanne einstellen, und der nächsten Task den Zugang ermöglichen.[2] Eine Task besteht dabei aus einer Einsprungfunktion, die wiederum weitere "Unter-"Funktionen aufrufen kann.

Die Architektur des STS ist wie folgt:



Die Kommunikation basiert darauf, daß für `IN` und `OUT` jeweils ein unstörbarer Kopierbereich `CONSIN` bzw. `CONSOUT` zur Verfügung steht, der jeweils dann aktualisiert wird, wenn der Zugriff durch I/O-Handler bzw. Pseudo-Tasks darauf ausgeschlossen ist.

2 Komponenten

2.1 Pseudotasks

Die Pseudotasks können in C-artiger Notation wie folgt definiert werden:

[1] zur Vorlesung vom 16.6.99

[2] Dies wird als *kooperatives Multi-Tasking* bezeichnet.

```
void ptask_1 ()
  {
   switch (getState(selector1))
   {
     case 0: ptask_10(); break;
     ...
     case m: ptask_1m(); break;
     default: ...
   }
  }
```

Dabei sehen die einzelnen Anwendungen der Pseudotasks folgendermaßen aus:

```
ptask1l ()
  {
   in1 = getInput(selector1l);                     (1)
   ...
   processInputs_1l(in1, ..., ink);
   putState (selector1l1, value1)
   putOutputs(selector1l2, value2);
  }
```

Anmerkungen:

- Diese Form der Abarbeitung der Pseudotasks entspricht genau dem Standardschema für die Implementierung endlicher Automaten. Dieses Schema ist auch bei reaktiven Systemen wiederzufinden, d. h. solchen Systemen, die kontinuierlich mit der Systemumgebung kommunizieren (z. B. Benutzungsoberflächen).
- `State_ptask_1` liegt im globalen State-Bereich für `ptask_1`
- Wie man in Zeile (1) der `ptask1l` sieht, ermittelt die Task selbst ob neue Inputs für sie vorliegen.[3] Damit wird die explizite Einführung einer Message-Queue überflüssig.

2.2 Mainloop

Die Mainloop spielt im STS die Rolle eines Betriebssystem-Kernels. Ein wesentlicher Punkt hierbei ist die Blockierung des I/O-Handlers für die Dauer des Datentransfers von `IN` nach `CONSIN`. Damit ist gewährleistet, daß `IN` nicht während des Transfers verändert werden kann. Dies erfordert jedoch eine Interruptsperre des I/O-Handlers.

Der Pseudocode für `mainloop` sieht wie folgt aus:

```
void mainloop ()
 {
  while (1)
  {
    blockInputs();
    copyIN2CONSIN();
    unblockInputs();
```

[3] Dies wird als *Polling* der Inputs bezeichnet.

```
    triggerwatchdog();
    ptask1();
    triggerwatchdog();
    ...
    ptaskn();
    triggerwatchdog();

    copyOut2CONSOUT();
    triggerOutput();
    copySTATE2CONSSTATE();
  }
 }
```

Zu Beginn eines jeden Schleifendurchlaufs werden die mit Inputs assoziierten Interrupts gesperrt und der Inhalt von `IN` nach `CONSIN` kopiert. Da während der Kopierphase keine Veränderungen auftreten können, erhält man in `CONSIN` ein konsistentes Bild der Inputs, welche im vorangegangenen Mainloop-Zyklus empfangen wurden. Nach dem Kopiervorgang wird die Interruptsperre wieder aufgehoben.

Dann werden die Pseudo-Tasks nacheinander gestartet. Alle lesen *denselben* Zustand `CONSSTATE` und *dieselben* Inputs `CONSIN`. Zustandsänderungen werden (am besten über eine ADT-Library, welche `STATE` und `CONSSTATE` versteckt) auf `STATE` durchgeführt, so daß sie noch nicht für andere Pseudo-Tasks sichtbar werden. Outputs werden nach `OUT` geschrieben und sind für den I/O-Handler noch nicht sichtbar.

Als nächstes wird `OUT` nach `CONSOUT` kopiert. Unter der Voraussetzung, daß die Outputs des vorherigen Zyklus im jetzigen Zyklus vor Aufruf von `copyOUT2CONSOUT()` abgewickelt werden konnten, ergibt sich damit in `CONSOUT` ein konsistentes Bild der Ausgaben, die nach dem folgenden `triggerOutput()`-Komando innerhalb des nächsten Mainloop-Zyklus zu erledigen sind.

Schließlich wird `STATE` nach `CONSSTATE` kopiert. Da hierbei keine der Pseudo-Tasks läuft, erhält man auch hier in `CONSSTATE` ein konsistentes Bild des Nachzustandes, der zum Ende des Schleifendurchlaufs von den Pseudo-Tasks in `STATE` hinterlassen wurde.

Mit diesem Schema der quasi-parallelen Verarbeitung implementiert man die Semantik synchroner Spezifikationssprachen im Discrete-Time Model (z. B. State Charts): alle Tasks sehen zu Beginn eines Verarbeitungsschrittes denselben Systemvorzustand (Eingabe und interner State). Die Veränderungen, welche eine Task während des Verarbeitungsschrittes durchführt, werden für andere Tasks (und für die Außenwelt) erst im nächsten Schritt (d. h. im nächsten Mainloop-Zyklus) sichtbar.

Hieraus resultiert, dass bei gegebenem Vorzustand jede Task unabhängig von der Task-Aufrufreihenfolge immer zum selben Ergebnis kommt. Die Zustandsübergänge und Ausgaben des Gesamtsystems sind unabhängig von der Task-Reihenfolge, wenn die Zustandsvariablen und Ausgabe-Kanäle, welche von den Tasks beschrieben werden, disjunkt sind.
Anmerkung: Wenn zwei Tasks die selben Variablen oder Outputs schreiben, so daß eine Reihenfolgeabhängigkeit vorliegt, nennt man das eine *Racing Condition*

3 I/O Handler

Der I/O Handler reagiert interrupt-gesteuert auf Inputs und legt die empfangenen Daten im Speicher `IN` ab. Die Mainloop mit den Pseudo-Tasks wird hiervon unterbrochen. Da die Pseudo-Tasks Eingaben aus `CONSIN` lesen, hat die Veränderung von `IN` auf sie keine Auswirkung.

Die Verarbeitung neuer Outputs wird von der Mainloop getriggert. Daraufhin übernehmen die zuständigen Output-Driver des I/O Handlers die Daten aus `CONSOUT` und schreiben sie auf die Ausgabeschnittstelle. Die Ausgaben können "Rückmelde"-Interrupts der angesprochenen Peripherie nach sich ziehen, auf die Folgedaten oder Wiederholungsversendungen an die Ausgabeschnittstelle übergeben werden. Dies unterbricht wiederum die Pseudo-Tasks ohne ihren Zustandsraum zu beeinflussen.

Die Ausgabebehandlung verläuft korrekt, solange jedes Peripheriegerät in der Lage ist, die kompletten Outputs in kürzerer Zeit abzuschließen als der Mainloop-Zyklus benötigt.

4 Verwendung von STS

Voraussetzung für die Verwendung eines solchen STS ist, daß die `mainloop` so schnell abläuft, daß während eines Durchlaufs der `mainloop` nicht mehrere Signale von außen in `IN` abgelegt werden können. Die Rolle des `watchdog` ist dabei die Erkennung von Todschleifen innerhalb einzelner Tasks, die das System blockieren würden. Eine einfache Variante des watchdogs würde etwa auf dem Bus mithören. Probleme mit dem `watchdog` kann es bei kontinuierlich arbeitenden Systemen geben: hier muß die Wiederherstellung des "richtigen" Systemzustandes Bestandteil der recovery-Strategie sein.

- Nachteile:
  - Eine Erweiterung des Systems um weitere Tasks ist im Betrieb nicht möglich, und erfordert daher ein Neuladen des gesamten STS.
  - Durch den ungeteilten Adreßraum ist die Gefahr des Überschreibens sensibler Bereiche vorhanden.
  - Pseudotasks sind nicht unterbrechbar.
- Vorteile:
  - Die einfache Struktur erlaubt eine einfache Realisierung des STS. Der Pseudocode kann schnell in gängige Programmiersprachen übertragen werden.
  - Die Einhaltung von Timing Requirements kann auf einfache Art verifiziert werden, da die Ausführungszeiten für Tasks einheitlich und von vorne herein festgelegt sind.
  - Es wird keine Extrazeit für Kontextwechsel benötigt.