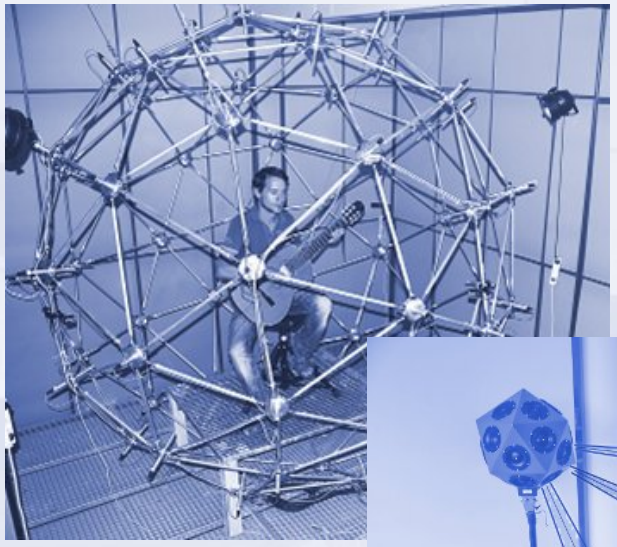
Implementierung von Algorithmen
Pure Data: Messages



IOhannes m zmölnig

Pd Objekte

- Funktionaler Ansatz
  - Objekte manipulieren Daten: „Funktionen“
  - Eingangsdaten → Ausgangsdaten

  - Dataflow
    - Objekte(Funktionen) werden nicht nacheinander abgerufen

      sondern
    - Daten „fließen“ durch Programm und werden modifiziert

Pd Messages

- Daten
- **flüchtig**: existieren nur zum Zeit*PUNKT* des Auftretens
- asynchron/on demand
  - user/patch bestimmt Zeitpunkt des Auftretens
- MessageBox
  - „eingefrorene Message“
  - wird „aufgetaut“ durch
    - Klicken
    - Message, die an MessageBox geschickt wird

Pd Objekte

- Objektorientierter Ansatz
  - Objekte haben **inneren Zustand**
    - von Außen nicht sichtbar
  - Methoden
    - interagieren mit innerem Zustand
- „Klasse“: Abstrakte Idee
  - „Zahlenspeicher“
- „Objekt“: Instanz einer Klasse

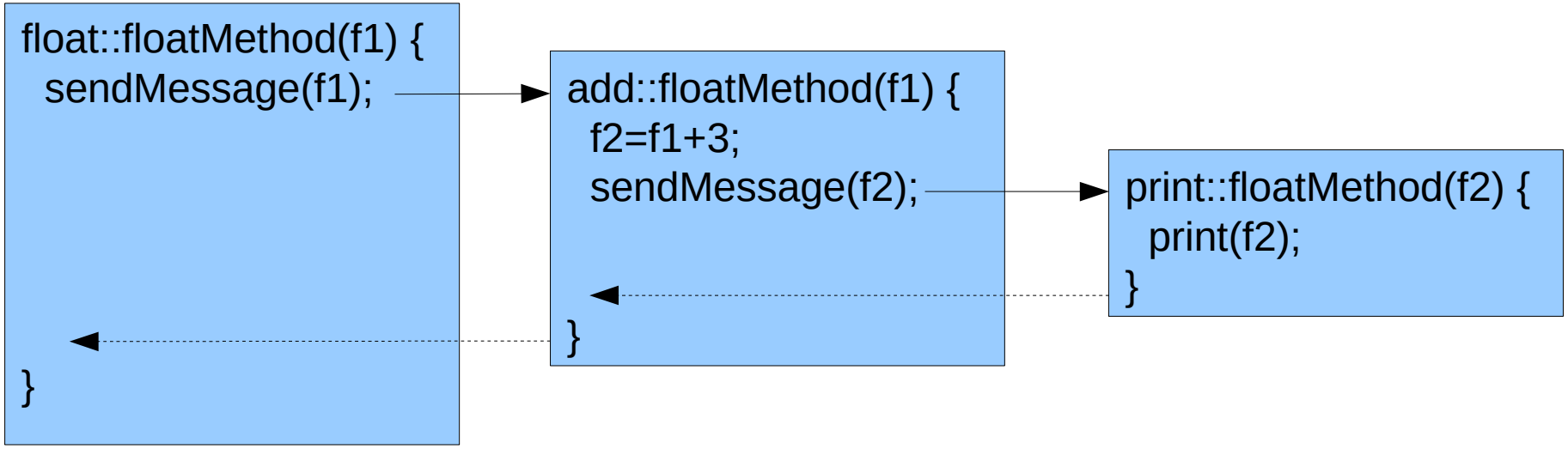
Pd Objekte: Methoden

- Messages rufen Methoden auf
- Objekt bestimmt, was mit Message (Daten) passiert
  - z.B. ob Ausgangsdaten produziert werden

  - MyObject::fooMethod(input) {
    output=function(input);
    sendMessage(output);
    }

Pd Objekte: Methoden

- rekursives Abarbeiten



- Pd-Dispatcher „ ⟶ “
  - nimmt Message auf Kanal entgegen
  - schickt Message an alle Objekte, die an diesem Kanal „hören“
- Messages bestimmen, welche Methode aufgerufen wird

Messages: Aufbau

- <selector> {<atom>}

- <atom>
  - Zahl (floating point)
  - Symbol (String in Hashtable)
  - Zeiger
- <selector>
  - Symbol

  - **jede** Message hat einen Selector
    - falls nicht explizit angegeben, wird automatisch „list“ (bzw. „float“) hinzugefügt

Atome

- float
  - alle Zahlen in Pd:
    - Single Precision (IEEE 32bit float)
    - Integers nur bis ±16777216
- symbol
  - Hashtable
    - jeder „string“ wird in einer Tabelle gespeichert, und fortan nur noch indiziert
    - Tabelle wächst!

Messages

- flüchtig!
  - Messages werden vom System on-demand erzeugt und wieder zerstört
  - malloc()/free()

Spezielle Message-Selektoren

- **bang**
  - Trigger!
  - immer nur <selector>, keine daten
- **float**
  - immer genau ein <atom> vom typ **number**
- **symbol**
  - immer genau ein <atom> vom typ **symbol**
- **list**
  - allgemeine Daten

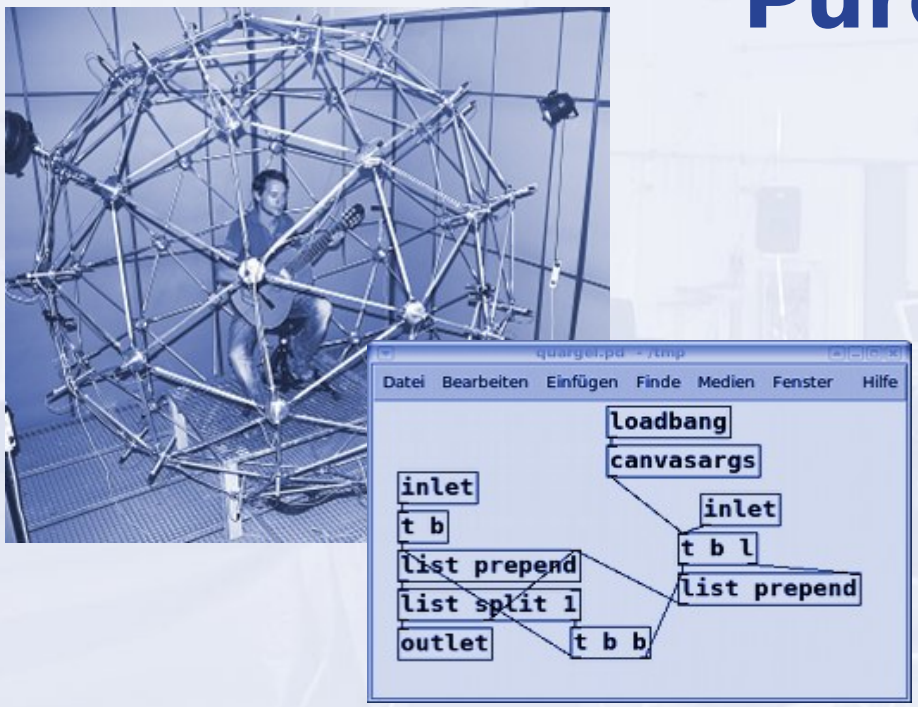
Dispatcher

- `<selector>` einer Message wählt Methode eines Objektes aus
  - Objekte melden Methoden für bestimmte `<selector>`en an
  - Evtl. catch-all Methode
- Message an Objekt
  - gibt's Methode für `<selector>` → aufrufen
  - sonst „catch-all“
- Implizite Konvertierung
  - „float <f>“ ↔ „list <f>“
  - „bang“ ↔ „list“
  - „symbol <s>“ ↔ „list <s>“

Pd-Objectmaker

- auch „Objekte“ werden als Message erzeugt:
  - <selector>: Objektname
  - {<atom>}: Übergabeargument
- der *objectmaker* hat eine Methode für jede bekannte Klasse
- catch-all (unbekannte Klassen)
  - versucht von Festplatte Klassen nachzuladen

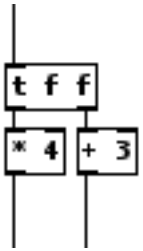
Implementierung von Algorithmen
Pure Data: Scheduler



IOhannes m zmölnig

Logische Zeit

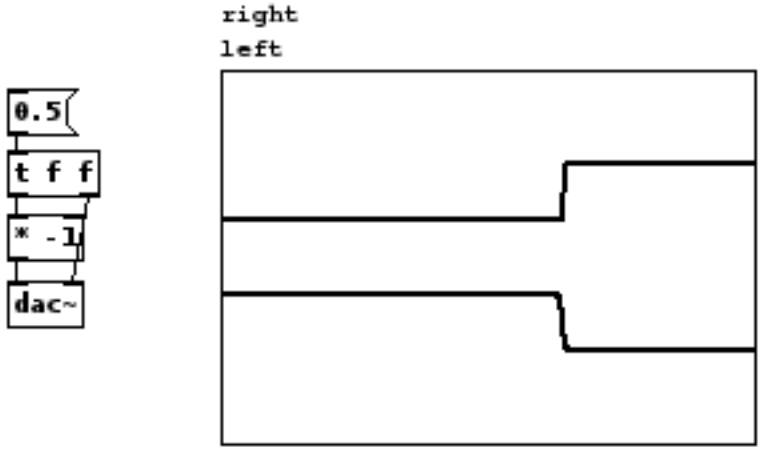
- innerhalb von Pd
  - alle Events finden zur „richtigen“ logischen Zeit statt (timestamp)
  - Folge-Message
    - finden zum gleichen logischen Zeitpunkt statt!
    - Messages passieren in „Null-Zeit“
  - Gleichzeitigkeit
    - *semantisch*: Messages mit gleichem Timestamp
    - *logisch*: Events werden immer sequentiell abgearbeitet!
      - deterministisch!



Echt(e) Zeit

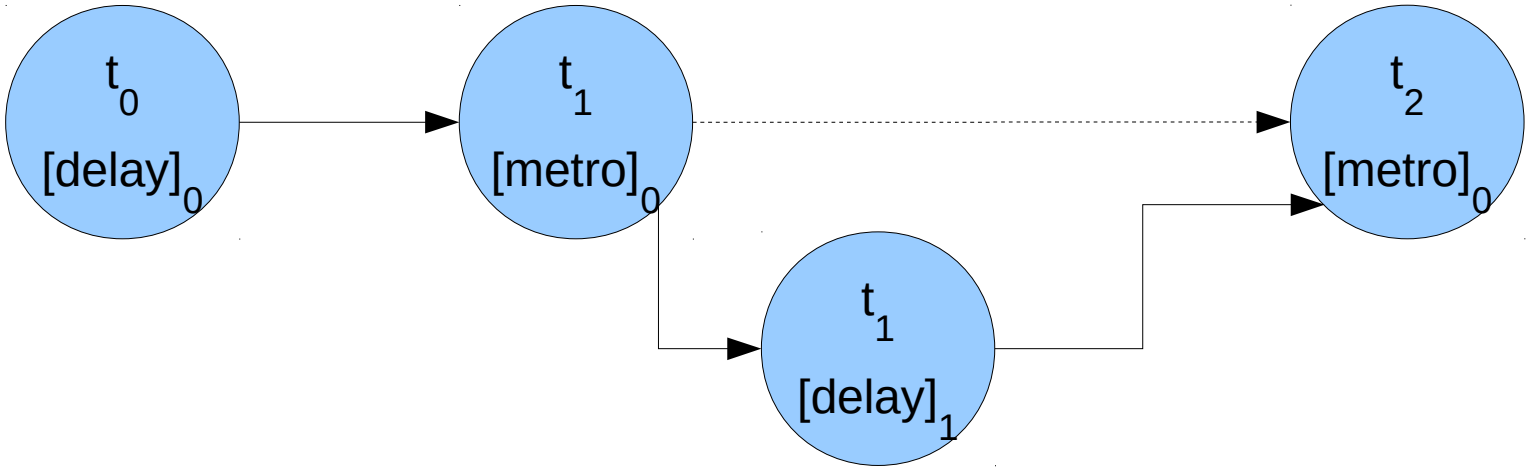
- Message-Verarbeitung braucht Zeit (CPU-Zyklen)!
- Messages werden in Bursts abgearbeitet
  - am Beginn des Tick-Zyklus
- Tick-Zyklen „schwimmen“ innerhalb eines Buffers

- Objekte die mit der Real World synchronisiert sind, können Timestamps verwenden um die logische Zeit in echte Zeit zu übersetzen



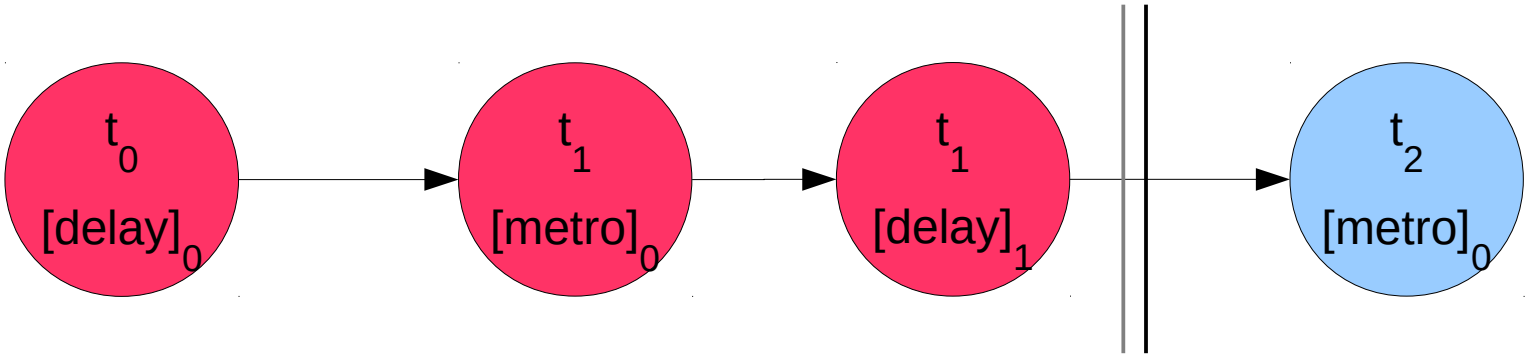
Message Scheduler

- geordnete Liste von Events (Timestamp+Objekt)
  - Events werden *am Ende* des jeweiligen Timestamps hinzugefügt



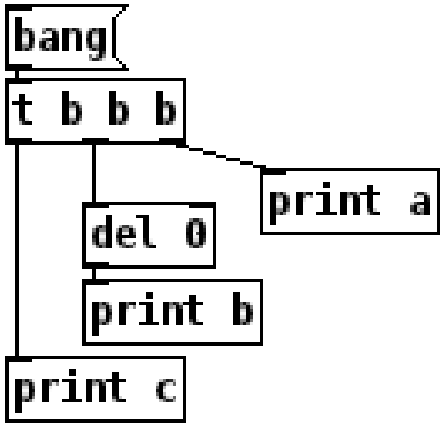
Message Scheduler

- Tick @ **τ**
  - für alle Events mit timestamp **t** < **τ**:
    - *setze logische Zeit auf* **t**
    - *rufe Objekt::tickMethod() auf*
    - *entferne Event*



Message Scheduler

- τ: „bang“ event from scheduler
  - [print a]
  - schedule event at τ+0=τ
  - [print c]
- τ+0: new „bang“ event from scheduler
  - [print b]



Message vs DSP

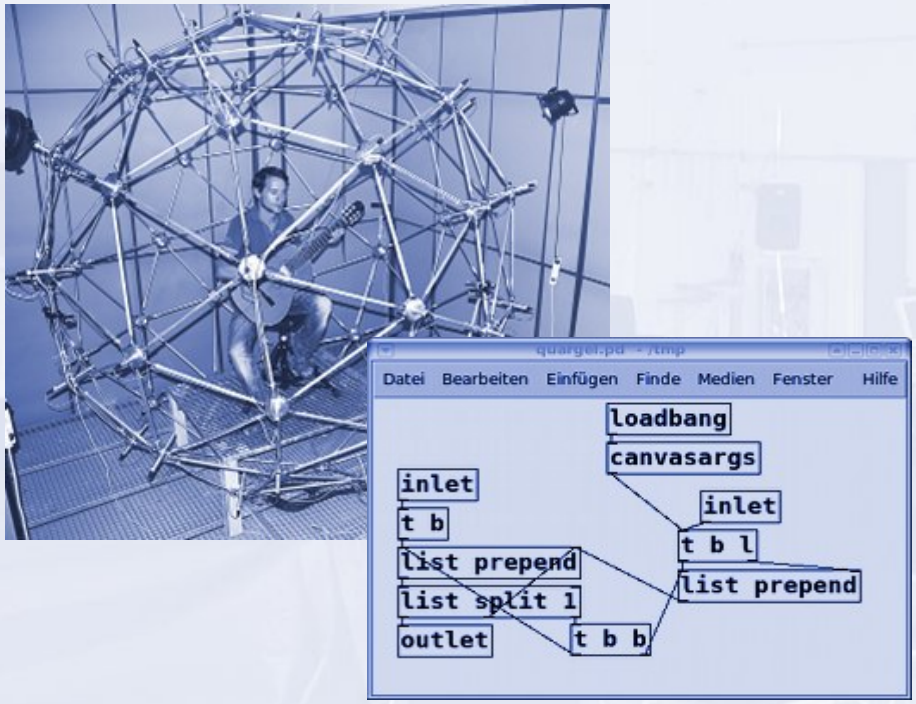
- Messages
  - asynchron
  - logische NULL-Dauer
  - variable Evaluierungszeit
- DSP
  - synchron zu Real World
  - fixe Dauer (Blocksize)
  - ~const. Evaluierungszeit

- Messages werden **vor** DSP-Berechnungen abgearbeitet
- Messages werden **immer alle** abgearbeitet
- DSP-Berechnungen können **ausfallen**!

Structured Programming

- Abstractions
- Lokale Variablen

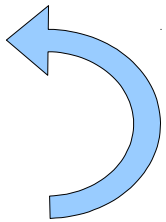
Implementierung von Algorithmen
Pure Data: Signal Processing



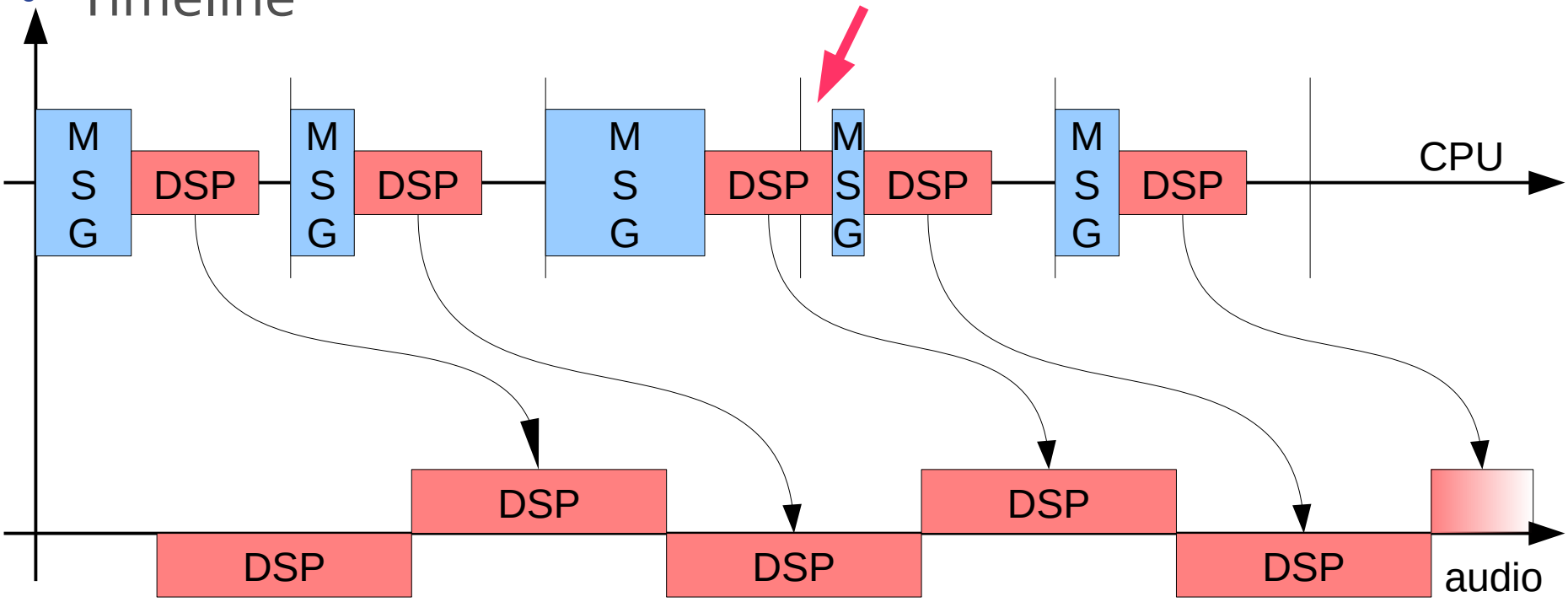
IOhannes m zmölnig

Scheduler

- Loop
  - messages (var.Dauer)
  - dsp (rel.const.Dauer)
  - *sleep*

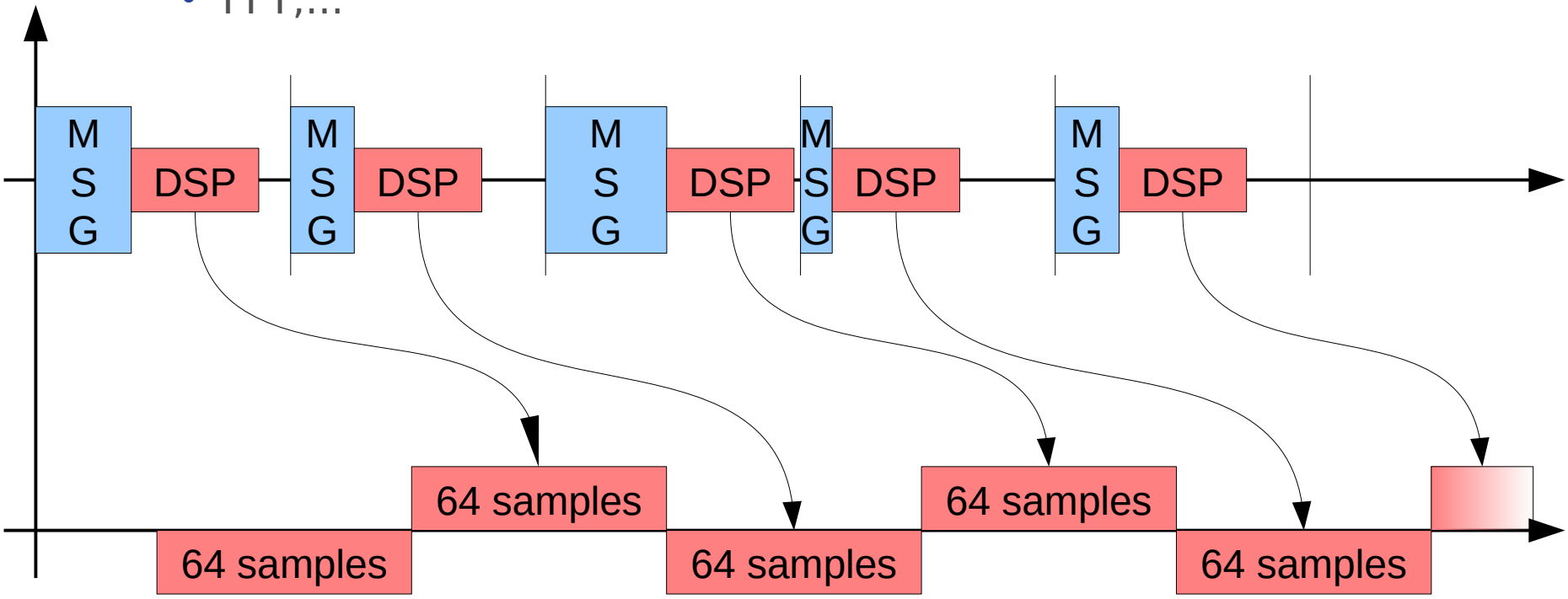


- Timeline



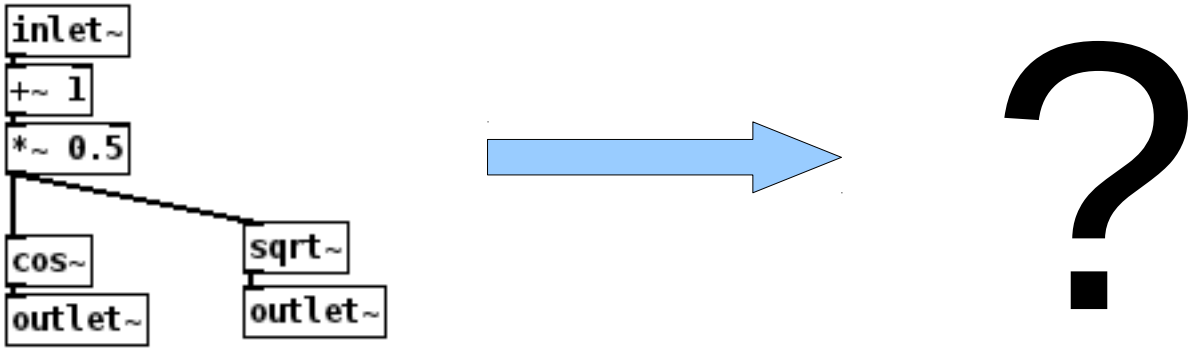
DSP

- Blockverarbeitung
  - Latenz ↔ Performance
  - Nachteil
    - rekursive Filter
  - Vorteil
    - FFT,...



DSP-Graph

- „Kompilieren“ des Datenfluss-Diagramms in ausführbares „Programm“



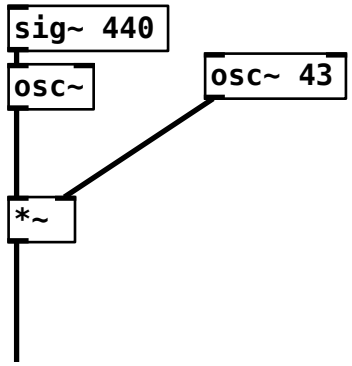
DSP-Perform Routine

- „Signal-Methode“ eines Objektes
- wird für jeden DSP-tick aufgerufen
- generiert aus einem Block Eingangssamples einen Block Ausgangssamples
- Input- und Outputbuffer können sich überlappen!

```
cos~::perform(size_t num_samples, sample_t input, sample_t output)
{
    for i in num_samples:
        output[i]=cos(input[i]);
}
```

DSP-Graph

- Auflösen von Abhängigkeiten
  - **Senke** kann erst ausgeführt werden, wenn *alle* Eingänge befüllt sind
  - → d.h. nachdem jede (verbundene) **Quelle** ausgeführt wurde

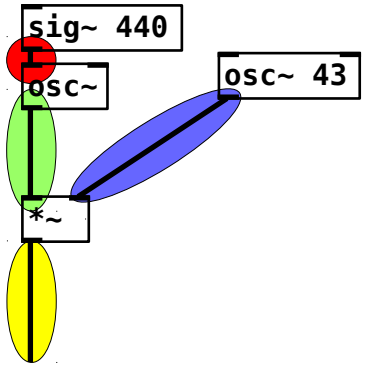


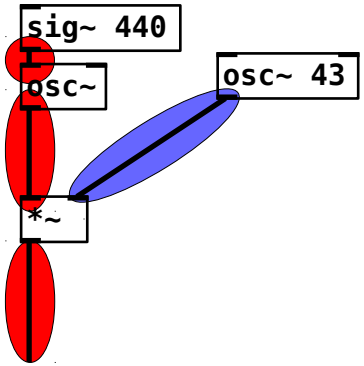
1. [sig~ 440]
2. [osc~]
3. [osc~ 43]
4. [*~]

1. [osc~ 43]
2. [sig~ 440]
3. [osc~]
4. [*~]

DSP-Graph

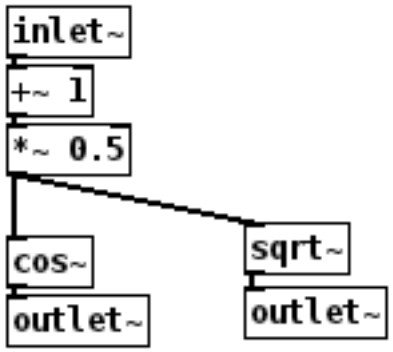
- Bufferverwaltung
  - Copy Optimierung
    - Quelle schreibt direkt in den Eingangsbuffer einer Senke
  - Cache Optimierung
    - Wiederverwendung von Buffern
      - Eingangsbuffer = Ausgangsbuffer
      - Mehrere Objekte verwenden den gleichen Buffer





DSP-Graph

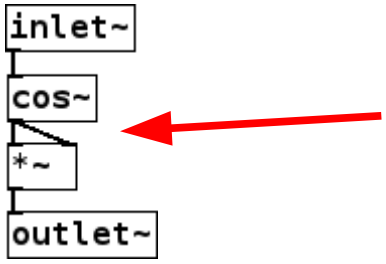
- „Kompilieren“ des Datenfluss-Diagramms in linearisierten DSP-Graphen
  - Auflösen von Abhängigkeiten
  - Cache-Optimierung
- ~~linked list~~ → array
  - „perform“ routinen + speicher für I/O



+~	vec1	vec1
*~	vec1	vec1
cos~	vec1	vec2
outlet~	vec2	–
sqrt~	vec1	vec3
outlet~	vec3	–

DSP: Order of Execution

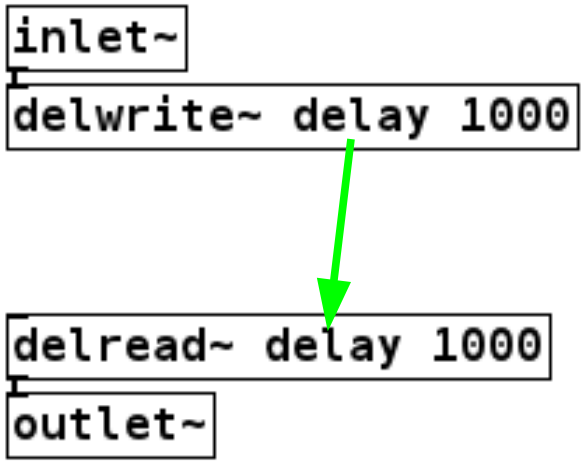
- Dataflow
  - Alle Eingänge müssen befüllt sein, bevor Ausgabe generiert werden kann
  - **synchron**
    - **bei jedem Zyklus müssen *immer alle* Eingänge befüllt werden**
    - **hot/cold wird nicht benötigt!**



`signalmul::perform(in1, in1, out1);`
wird erst aufgerufen, nachdem [cos~] den „in1“ Vector geschrieben hat

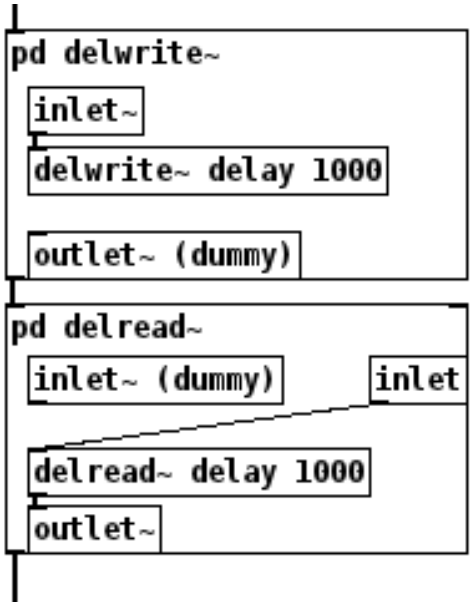
DSP: Implizite Verbindungen

- Unklar, welches Objekt zuerst Output generieren muss/soll!
  - wenn [delread~] vor [delwrite~] ausgeführt wird, gibt’s ein Delay von (mindestens) 1 Block
- „trigger für DSP“?



DSP: Order Forcing

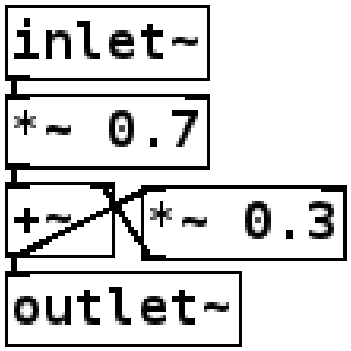
- Explizites Festlegen der Ausführungsreihenfolge
- Subpatches/Abstraktionen werden immer „gemeinsam“ ausgeführt
- Subpatches/Abstraktionen können mit iolet~s geordnet werden



„dummy“-outlet~→„dummy“ inlet~ garantiert, dass
[pd delwrite~] (und damit [delwrite~])
immer *vor*
[pd delread~] (und damit [delread~])
ausgeführt wird

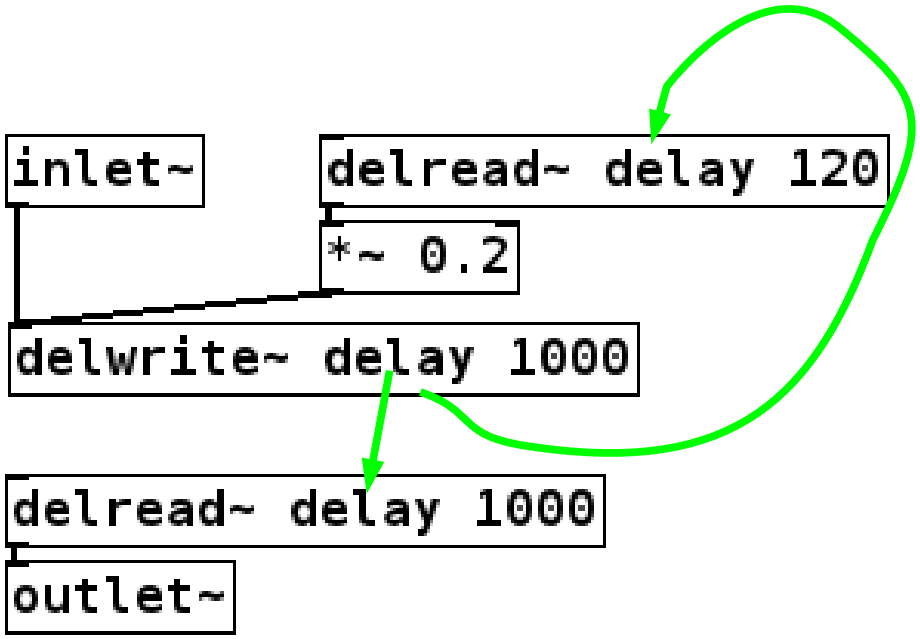
DSP: Rekursion

- Dataflow-Bedingung (alle Eingänge befüllt) kann **nicht** immer **eingehalten** werden



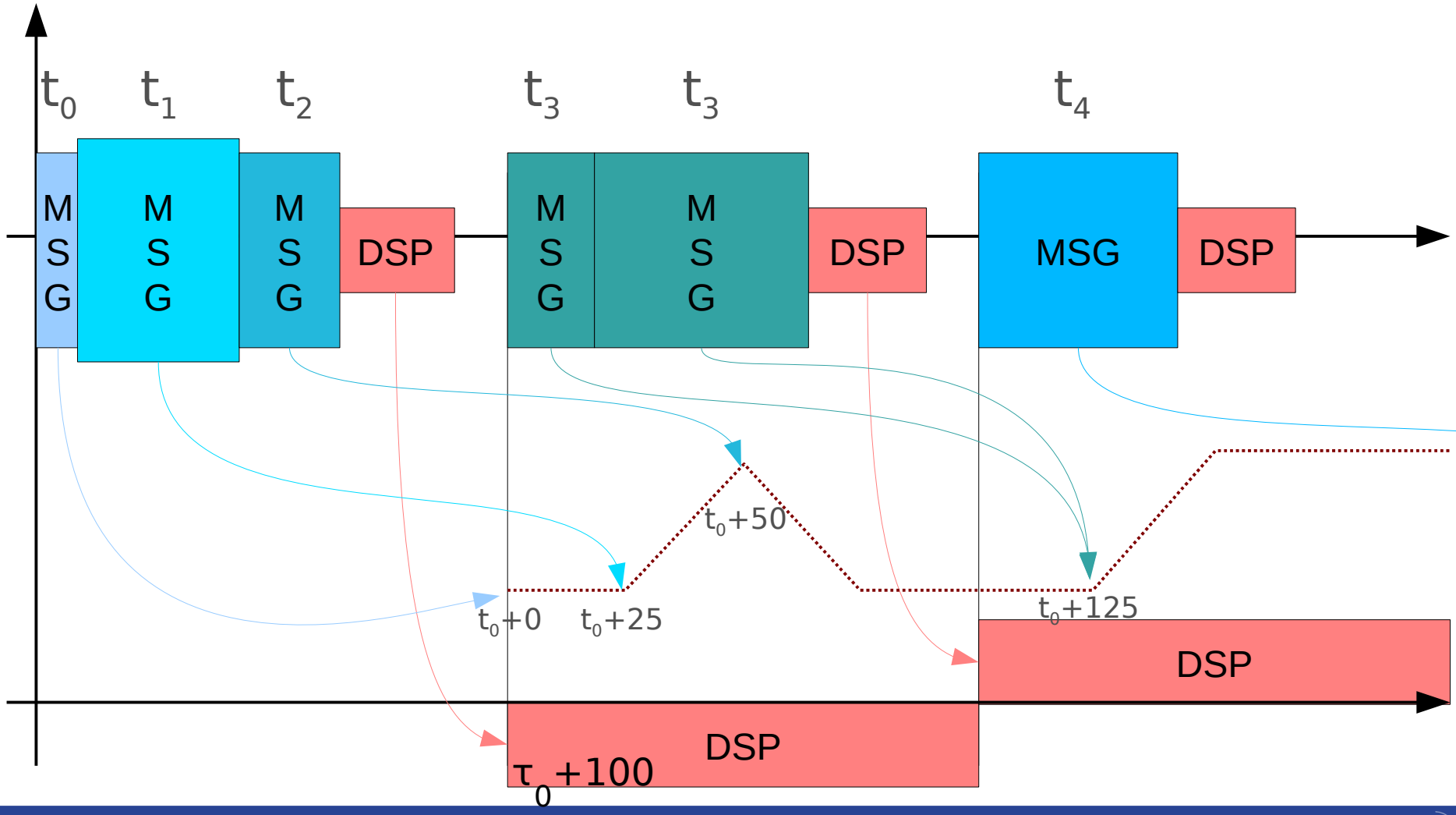
DSP: Rekursion

- → Verzögerung
  - Um (mindestens) einen ganzen Block(!)



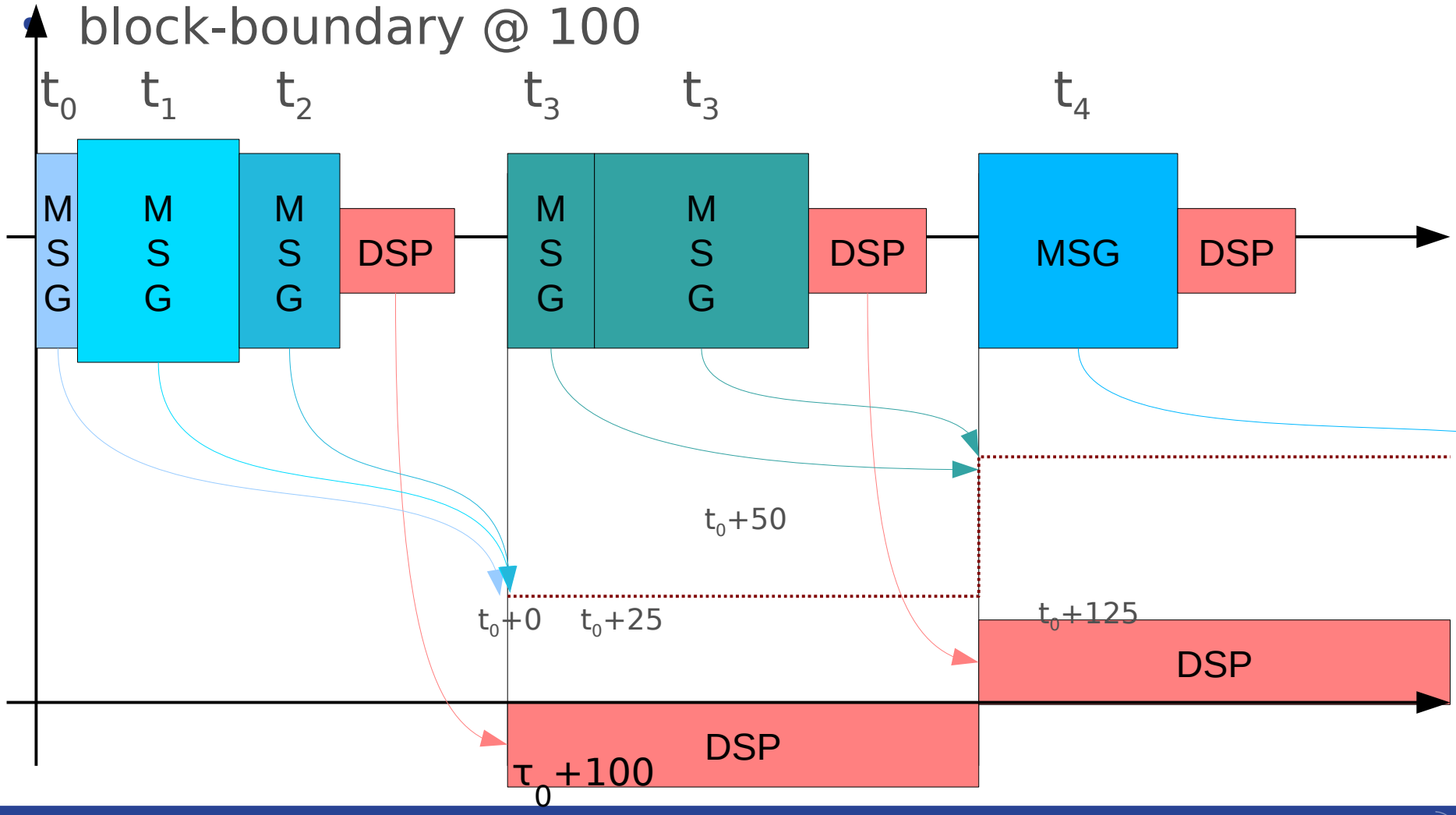
logical time and DSP time (ideal)

- $t_0 = t_1\text{-}25 = t_2\text{-}50 = t_3\text{-}125 = t_4\text{-}250$



logical time and DSP (block boundaries)

- $t_0 = t_1\text{-}25 = t_2\text{-}50 = t_3\text{-}125 = t_4\text{-}250$
- block-boundary @ 100



(sub)-sample accurate timing

- $t_0 = t_1\text{-}25 = t_2\text{-}50 = t_3\text{-}125 = t_4\text{-}250$