Rom based filter graph

The ROM based filter graph is implemented by a number of ROM based lists which are NULL terminated. Each list is maintained in its separate section. These sections need to be alphabetically sorted and consecutively allocated.   
Unfortunately there is no portable way to achieve this. Usually variable attributes handle section assignment and a linker script takes care of correct placement.

# Sources of information / Outputs

An asterisk (\*) denotes global statements.   
Each source maintains its list of tasks to wake on activation.

define\_output(*c-type*, *outputname*); \*implement a source of information

define\_event(*outputname*); \*this source can only wake tasks (c-type is void)

define\_interval\_timer(*outputname,milliseconds*); \*continuous source of events

define\_timeout(*outputname,milliseconds*); \*one event at a specific time

## Importing

declare\_source(*c-type*, *outputname*); \*import a source of information from a different file

declare\_event(*outputname*); \*import an event from a different file

## Data passing

output\_prepare(*outputname*)= *value*; fill output with information (lvalue)

output\_available(*outputname*); filling completed, activate tasks waiting on this event

output\_get(*outputname*); read last value

## Timer control

stop\_timer(*outputname*); stop interval timer or timeout

restart\_timer(*outputname, milliseconds*); restart interval timer or timeout

## Predefined events

program\_start\_src event at program start (once)

idle\_source activated whenever there are no tasks ready to run   
(no work to do)

# Tasks

A task consists of a RAM based variable part (task\_ram\_t) and some ROM based fixed information (task\_t). A task is implemented by a function which receives the pointer to the ROM based information, the RAM information (which is extensible) is available as ->var.

define\_task(*taskname*, *functionname*); \*create task structure for a function

define\_task3(*taskname*, *functionname,variable*); \*create task structure for a function with a user provided variable implementation (see example2.c)

# Connections

connect(*outputname*, *taskname*); \*wake task when new output available

# System infrastructure

int sr=RBF\_enter\_critical\_section(); disable interrupts

RBF\_leave\_critical\_section(sr); restore interrupt state

# Extension 1: Event buffering

Each output buffer implements a cyclic array and a list of inputs,   
each input buffer implements its read pointer – combining to a fifo.  
The read pointer is incremented by the available function on overflows.

define\_output\_buffer(*c-type*, *outputname,size*); \*Event buffer of given size

define\_input\_buffer(*outputname,inputname*); \*Coupled read pointer into event buffer

## Importing

declare\_source\_buffer(*c-type*, *outputname*); \*import a buffer from a different file

## Data passing

output\_buffer\_prepare(*outputname*)= *value*; move read pointer on overflow, fill next element with information (lvalue)

output\_buffer\_available(*outputname*); fill complete, update write pointer, activate listening tasks

## Reading

rbf\_buffer\_index\_t *i*; loop variable  
LOOP\_OVER\_BUFFER(*i*, *inputname*) determine next element in the input buffer  
{  
 printf(“%d ”, buffer\_get(*outputname*,*i*)); access element data  
}

rbf\_buffer\_index\_t buffer\_eval(*inputname*); index of next element in buffer, …

RBF\_outbuf\_invalid this return value indicates empty buffer

buffer\_getlast(*outputname*); asynchronously read last value

# Extension 2: Sinks

Sinks rely on buffer pairs to implement multi source data processing.  
On a global perspective a source drives multiple readers, while a sink listens to multiple sources.

define\_sink(*c-type*, *sinkname*); \*List of input buffers forming a sink

declare\_sink(*c-type*, *sinkname*); \*Import a sink from a different file

connect\_sink(*outputname*, *sinkname*); \*Create input buffer, and connect to source and sink

void *function*(const task\_t\* t) Sink sample implementation  
{ const *c-type* \**i*;  
 LOOP\_OVER\_SINK(*i*, *sinkname*) determine next unread input element  
 { printf(“%d ”,\**i*);  
 }  
}

sink\_implementation(*sinkname, function*); \*Implement the sink task using this function

# Hints

* A source can provide data of a structured type.
* Most hardware devices form either sources and/or sinks
* Initializing hardware can happen in tasks waiting for program\_start\_src, there is no need to add initialization calls to main. Just link the driver into the executable.
* If you need a specific order of (hardware) initialization define events for later stages. Activate these either directly from preceding initialization tasks or at the end of the previous stage (e.g. by listening to idle\_source).

# Possible extensions (which are implemented in iv4)

* Task priorities: Add priority field to task\_t and do priority insert in wake\_process
* Runtime bookkeeping: Add accumulated time to task\_ram\_t, current\_task pointer in main, current task run time counting in tick\_1ms.
* List of tasks: Add task to a ROM list in define\_task
* Stopping a task: Add limit to task\_t, mark task as not ready, implement longjump from timer interrupt to while loop in main, requires runtime bookkeeping
* System timer: Increment a global variable in tick\_1ms