A New, Lightweight Dataflow System for SDR and Control Systems

Dr. János Selmeczi
HA5FT
ha5ft@freemail.hu

Hello everybody. My name is Janos Selmeczi and my ham radio call sign is HA5FT. As you see from my callsign I am from Hungary. In this session I will present You a new, lightweight data flow framework which You could use to build SDR application and control and communication systems on various platforms. I could be reached at the e-mail address ha5ft@freemail.hu or in the high frequency amateur bands.

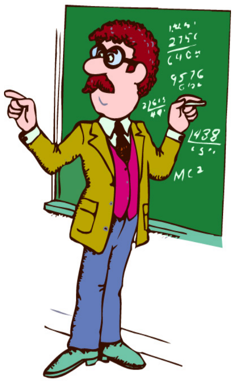
Introduction

- Electronic engineer for 40 years
- Equipments for space probes
- Industrial control systems
- Country wide financial systems

- HA5FT
- Operator since 1968
- Callsign since 1982
- AMSAT related works at HG5BME/HA5MRC

First of all let me introduce myself. I am an electric engineer. In my professional life I worked on many fields of my profession from designing and building equipment for space probes to creating large, country wide financial systems like an interbank clearing system. I have been ham radio operator since 1968 and I have got my license and my call sign in 1982. I was involved in some AMSAT related work at the radio club of the Technical University of Budapest. I am having been a pensioner since the beginning of this year and hopefully I will have more time for my hobby.

What is it all about?



- I have a dream
- Back to the school
- Do you speak SDF?
- Implementation

The data flow framework what I will talk about is not ready yet. It is in alpha stage, but I feel important to present it to You because it is different from the other data flow systems available today and because I like to have Your feedback on my ideas. My system is different because it uses a different data flow model, it is written in C, it could be run without an operating system and it has designed to support distributed systems. You could use it to develop embedded applications for small processors like the ARM Cortex-M4.

I have a dream

- Component based architecture
- Model driven development
- Distributed system support
- Multiple platforms
- A variety of processors

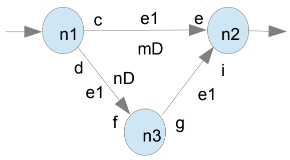
I have dreamed of this system for many years. In my dream there was a development framework which let you concentrate on writing algorithms, which frees you from the boring job of writing glue code and wich and which allows you to make distributed applications. After some research I realized that such a system should be model driven and component based.

Dreaming of componnents

- Large components
- Primitives and composites
- Primitives written in C or Verilog
- Special language for composition
- Static or shared libraries
- Embedded or dynamically loaded

I have decided to use coarse-grained components. Due to this the efficiency of the glue code is not so important. This simplifies the code generation and allows the use of a virtual machine for running the glue code. There are two kind of components in the framework. We have primitives which are written in C language and we have composites which are constructed from the primitives and other composites. So the component system is hierarchical. The composite components are defined using a special language, the SDF language which is part of the framework.

Dreaming of models



- Synchronous dataflow
- Static schedule
- Textual model description
- Extensions
  - hierarchical description
  - explicit control data, parameters
  - C-like switch
  - iterator

My framework is based on the synchronous data flow model and not on the dynamic model used in gnu radio and in Photos SDR. The synchronous model has some advantages. You could precompute the execution schedule. This simplifies the runtime system. The synchronous data flow model enables you the explicit use of feedback loops which is currently not possible in gnu radio and Photos SDR. I have extended the basic synchronous model. The extensions increase the usability of the model. The most important of the extensions are the hierarchical description and hierarchical scheduling, one to many connections, the explicit use of control parameters, a C like switch construct and an iterator.

```
composite M
   context
      input     float[5] i1[]
      output    float[5] o1[]
      parameter int      p1
   end
   signals
      stream    float[5] s1[]
      const     int      c1
273
   end
   actors
      primitive  P1  a1
      composite  C1  a2
      primitive  A7  a7
   end
   topology
      a1.i1  <<  i1
      a1.o1  >>  s1
      a1.p1  <<  p1
      a2.i1  <2< s1
      a2.p1  <<  c1
      a2.o1  >>  o1
   end
   schedule
      auto  a1
   end
end
```

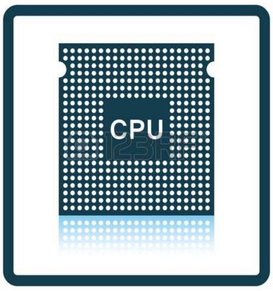
Dreaming of compilers

- Compiles the SDF language
- It is a declarative language
- Describes composit components
- Compiler generates
  - binary virtual machine code
  - C code
  - verilog code

I desided to use a declarative language for describing the composites. I have not found any existing language for this job, so I created a new language and the necessary compiler infrastructure. To have a feeling of the SDF language I show you a short composite declaration on this slide. The language is text based. The framework uses a special compiler to translate the model description into a runnable code. Today the compiler generate code which should be run by on a virtual machine. In the future C code generation will be possible, but if you really use coarse-grained components the speed advantage of a C language glue code is not substantial.

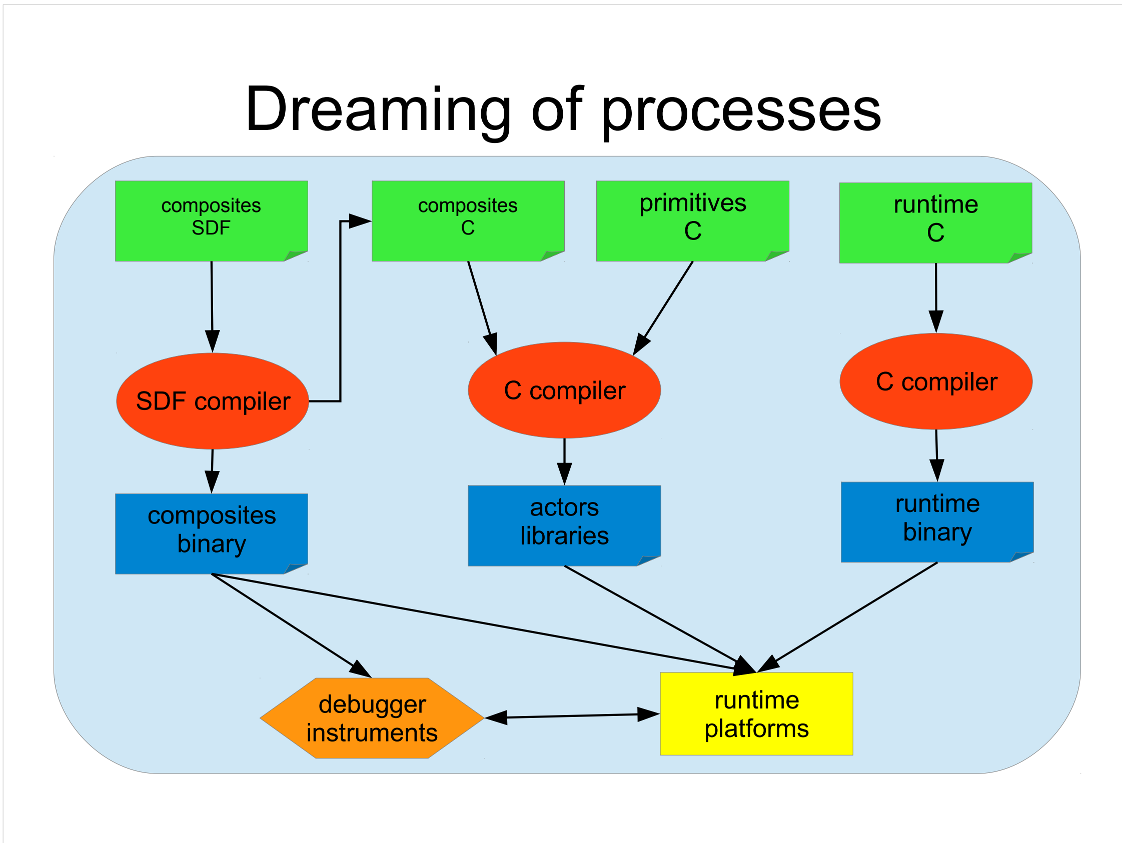
Dreaming of platforms

- Linux
- FreeRTOS
- no OS, bare metal



- Intel x64
- ARM Cortex-A9, Cortex-M4
- PIC32

I plan to support a number platforms and processors. Today the compiler runs on linux and the runtime system could run on linux or in an ARM Cortex-M4 processor without an operating system. Linux is supported on Intel and ARM processors. I have plan for supporting FPGAs and GPUs.



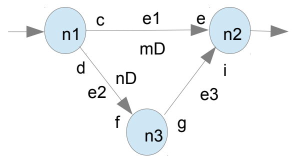
The development process has three threads. Most of You should be involved in the primitive and composite development. You must use the runtime development thread only if you like to have the components embedded into the runtime system.

Back to the school

- Synchronous actors
- Signals
- Synchronous data flow graph
- Topology matrix
- Balance equation
- Solving the equation
- Example ballance equation
- Scheduling

Now we will go back to the school to learn some of the theory of the data flow systems.

The data flow paradigm

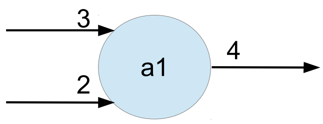


- A program is divided into algorithms and data which the algorithms are working on.
- Algorithms are executed whenever input data are available.
- A data flow system is described as a directed graph
- Nodes representing the algorithms
- Edges representing the data
- Nodes are usually called actors
- Edges are sometimes called signals

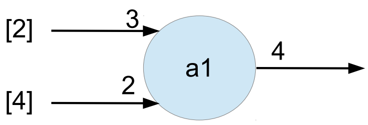
In the data flow paradigm we split our program into algorithms which do the data processing and data management components which manage the data the algorithms are working on. There are no other code components used. The algorithms will execute whenever they have enough input data. This kind of execution of the algorithms will provide the system functionality. To define a system we describe how the algorithms are connected by the data management components and what are the data need of the algorithms for the execution. For this we use a directed graph.

Actors

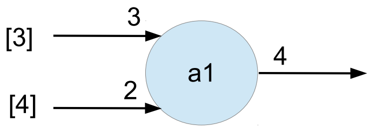
- Nodes of a data flow graph
- Atomic execution of their algorithms



- Cosumes 3 data elements on one input and 2 data elements on the other input
- Produces 4 data elements on the output



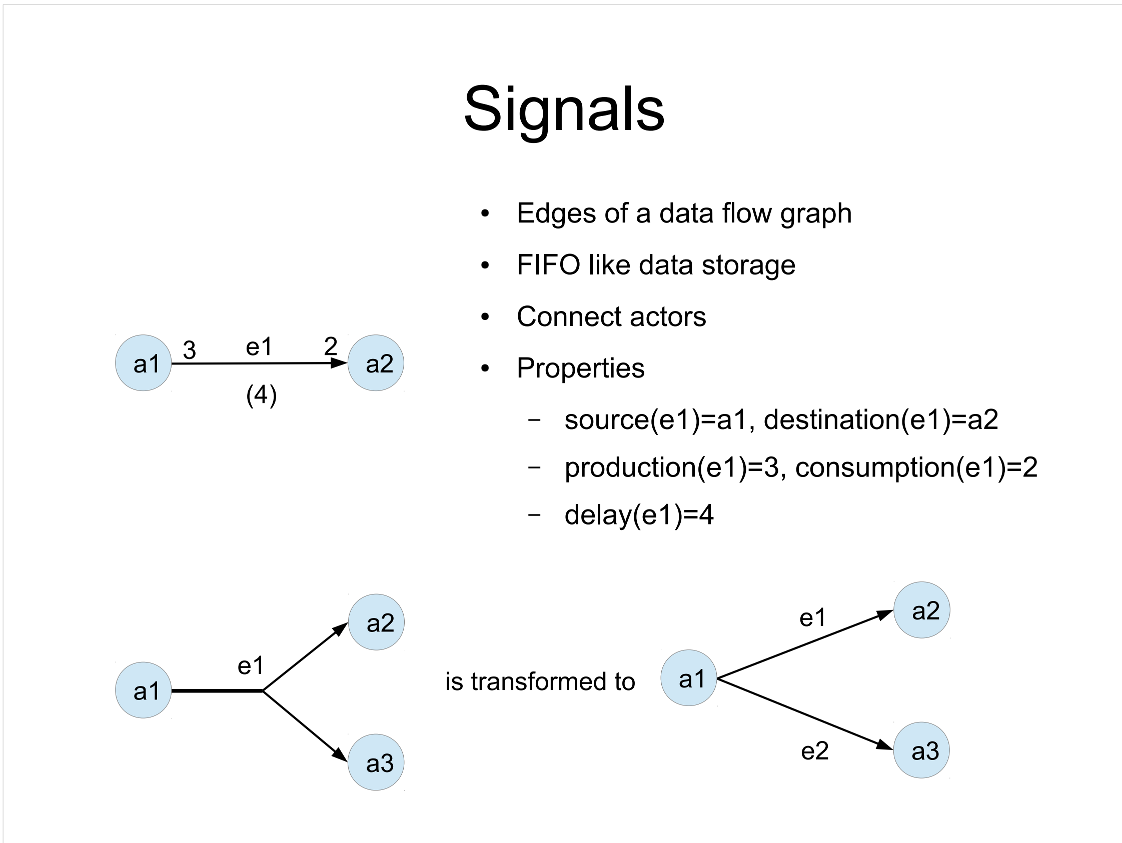
No execution



Execution

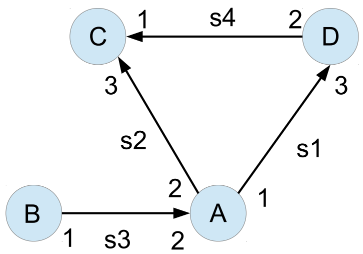
- Synchronous actor: fixed consumption and production

The instances of the algorithms usually called actors. They are the nodes of the directed graph. They do atomic execution of their algorithms. This execution is called fireing. They behavior is specified by how many data they consume and produce during a fireing. They always fire if they have enough data to work on. If the production and consumption behavior of an actor is fixed the actor is called synchronous.



The instances of the data management components usually called signals. They are the edges of the graph. They connect the actors. They behave like FIFO buffer with unlimited storage capacity. In the original model they have a single source and a single destination actor. So the multiple destinations connections used on block diagrams should be translated to multiple single destination connections. However I have extended the basic data flow model to allow multiple destinations connections. The behavior of a signal is determined by what is the source and what are the destination actors, by the data production of the source and by the data consumption of the destination actors and finally by the data delay through the signal. The delay is the data elements initially placed into the signal buffers.

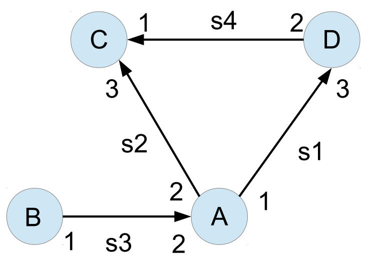
Synchronous data flow graph



- Directed multigraph
- Nodes are synchronous actors
- Edges are signals
- Multi destination signals are transformed multiple single destination signals
- Signals may have delays
- Example
  - actors: A, B, C, D
  - signals: s1, s2, s3, s4
  - no delays

The graph should not be fully connected, it could be a multi graph. If all the actor in the graph are synchronous the graph is called synchronous data flow graph. Synchronous graphs have special properties.

Topology matrix



$$
\Gamma = \begin{bmatrix} 1 & 0 & 0 & -3 \\ 2 & 0 & -3 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 \end{bmatrix}
$$

- Shows production and consumption behavior of the data flow graph
- columns correspond to actors
- rows correspond to signals

$$
\Gamma(s,a) = \begin{cases} prd(s), & \text{if } a = src(s) \\ -cns(s), & \text{if } a = snk(s) \\ 0, & \text{otherwise} \end{cases}
$$

- The evolution of the graph

  q(a)=inv(a), the number of invocation of actor **a**

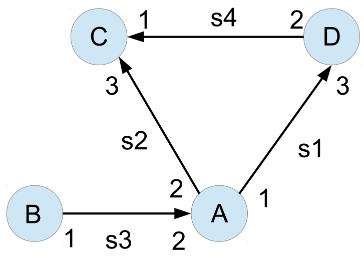
  b(s) the number of data element in signal **s**

  b0(s) the number of data elements in signal **s** before the execution

  b = Γq + b0

Now I will discuss how the amount of data stored in the signals change during the execution of the graph. This could be described by using the topology matrix. In this matrix the columns correspond to the actors and the rows correspond to the signals. A matrix element describe how many data an actor is producing to or consuming from a signal. Positive number means production, negative number means consumption. You could compute the data changes using the equation on the slide. The vectors b, b0 and q has integer elements. The element of q specify how many times the actors are executed. The elements of vector b show the number of data stored in the signals after the execution and the elements of b0 show the number of data in the signals before the execution.

Balance equation



$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & -3 \\ 2 & 0 & -3 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} 3 \\ 6 \\ 2 \\ 1 \end{bmatrix}$$

$$\mathbf{0} = \Gamma\mathbf{q}$$

- The solution shows the number of invocations of the actors after which the number of data elements stored in the signals will be unchanged.
- Has solution if the rank of the matrix is one less than the number of columns.

Now lets have a q vector with arbitrary integer elements. If we are lucky the number of data stored in the signal after the execution will be the same as it was before the execution. In this case we say that the q vector specifies a periodic execution of the system. If a system has periodic execution it could be executed forever with limited signal storage capacity. We could find a periodic execution by solving the balance equation of the system.

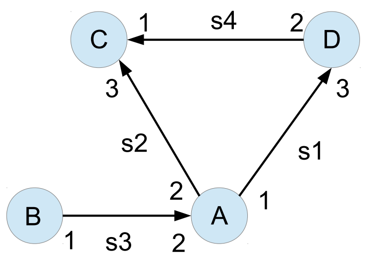
Solving the equation

```
procedure ComputeRepetition(G)
        for each A in actors(G) do reps[A] = 0;
        select A' from actors(G);
        SetReps(A',1);
        m = lcm({denom(reps(X)) | X in actors(G)});
        for each A in actors(G) do reps(A) = m * reps(A);
                for each e in edges(G) do
                        if (reps(src(e)) * prd(e)) <> (reps(snk(e)*cns(e)) then
                                error: inconsistent graph
                                exit
                        endif
                endfor
        endfor
        for each A in actors(G) do q[A] =
numer(ReducedFraction(reps[A]));
endproc

procedure SetReps(A,n)
        reps(A) = n;
        for each e in output(A) do
                if reps(snk(e)) = 0 then

SetReps(snk(e),ReducedFraction((n*prd(e))/cns(e)));
                endif
        endfor
        for each e in input(A) do
                if reps(src(e)) = 0 then

SetReps(src(e),ReducedFraction((n*cns(e))/prd(e)));
                endif
        endfor
endproc
```

- Recursive algorithm
- Uses fractional number arithmetic
- The algorithm
  - Choose an actor
  - Execute it once
  - If a connected actor has not been executed yet then execute it so that the edge connecting the two actors will be ballanced
  - Do this recursively for all actors
  - Convert the fractional number of executions to integer ones

If the rank of the topology matrix is one less than the number of actors the balance equation has a solution. The rank of the matrix means the number of independent row vectors of the matrix. There are many algorithms for solving this kind of equation. On the slide you see a particular algorithm which uses rational number arithmetic. It is a recursive algorithm. This is a very simple algorithm. You choose an arbitrary actor. You execute it ones. After the execution you visite all connected actors. If a connected actor have not been executed yet, you will executed it in such a way, that the signal connecting the two actors will be balanced. If the connected actor has already been executed you will do nothing with this actor. You will do this recursively for all the actors. Finally you convert the fractional execution number to integers by multiplying them with the least common multiple of their denominator.

Balance equation example



$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & -3 \\ 2 & 0 & -3 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 \end{bmatrix} \quad \mathbf{q} = \begin{bmatrix} 3 \\ 6 \\ 2 \\ 1 \end{bmatrix}$$

Choose actor A

reps[A] = 1/1

for edge s1: reps[D]=(1/1)*(1/3) = 1/3

for edge s4: reps[C]=(1/3)*(2/1)=2/3

for edge s2: do nothing because reps[A]<>0

for edge s2: do nothing because reps[C]<>0

for edge s3: reps[B]=(1/1)*(2/1)=2/1

now reps={1/1, 2/1, 2/3, 1/3}

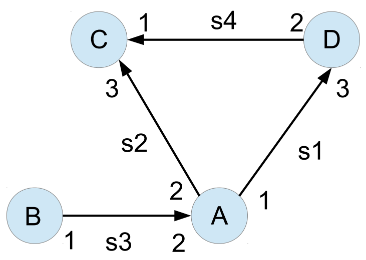
lcm(1,1,3,3)=3

reps=reps*3={3/1, 6/1, 6/3, 3/3}

q={3,6,2,1}

On the slide there is an example for solving the equation. Everybody interested in this could follow the detailed explanation on the slide and the description of the algorithm on the previous slide.

Scheduling



$$\Gamma = \begin{bmatrix} 1 & 0 & 0 & -3 \\ 2 & 0 & -3 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2 \end{bmatrix} \quad \mathbf{q} = \begin{bmatrix} 3 \\ 6 \\ 2 \\ 1 \end{bmatrix}$$

- Schedule is a sequence of actor executions
- Solution of the balance equation defines the number of actor executions for a periodic schedule
- Scheduling algorithms usually use simulation.
- Example: BBA BBA BBA D C
- Loop schedule: (3((2B)A))DC

A schedule is a sequence of actor execution. If in the schedule the execution number for each actors correspond to the solution of the balance equation then the schedule is called periodic schedule. A schedule is called admissible if whenever an actor is executed in the schedule it has enough input data to execute. A graph could have periodic schedule but not periodic admissible one. We could find periodic admissible schedule by simulation. There are several algorithms to do this. It is important to note that if we have an admissible schedule we could blindly execute the actors according to the schedule and do not gave to check if the actors have enough input data or not. For sure they have. It is very important that a graph could have periodic admissible schedule even if it has loops. If it do not have such a schedule you could put some delays on the feedback path. It could be proved that if we use enough delays on the feedback path and if the graph has an admissible schedule if you open up the feedback loop then the graph with feedback loop has an admissible schedule. This means, that using my model you could build systems with explicit feedback. Furthermore if we have admissible periodic schedule we could precompute how large data storage each signal must have.

Do you speak SDF?

- Composite actor declaration
- Signals and ports declaration
- Topology declaration
- Actors declaration
- Schedule declaration
- Example

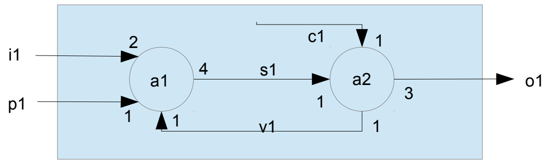
After learning some theory we will see how we could describe a system by the SDF language. The language is declarative, there is no program flow defined by the language. The language is line oriented. Each line is a sentence. Sentences are built from words. Words are separated by white spaces. The indentation on the examples are for clarity purposes only. Comments starts with a semi-colon and ends at the end of the line. Today the support for distributed system has not been finished yet, so I will not talk about it. The implementation is in progress and is based on proxy and wrapper functionality of composites.

Composite actor declaration

```
composite_declaration ::=
    "use" SP component_name NL
    {"use" SP component_name NL}
    "composite" SP composite_name NL
        context_section NL
        signal_section NL
        actor_section NL
        topology_section NL
        schedul_section NL
    "end" NL
interface_declaration ::=   ("primitive" | "composite") SP actor_name
                                context_section NL
                            "end" NL
SP ::= (" " | "\t") {(" " | "\t")}
NL ::= "\n" | "\r\n"
```

The purpose of the language is to describe composite components. The declaration of a composite component has two parts. The first part lists the primitives and composites used in the composite. We use the use sentence for this purposes. If the compiler finds a use sentence it will read the component's interface specification from the interface declaration file. This is similar to the include mechanism in the C language. The second part declares the composite. It has five sections. The context section declares the interface of the component. The signal section declars the signals used by the component. The actor section declares the actors of the components. The topology section declares how the signals connect the actors. Finally the schedul section declares what kind of scheduling should be used.

Composite actor example




```
;in fájl P.ctx
primitive P
   context
      input int[5] i1[2]
      input int i2[1]
      output float[3] o1[4]
      parameter long p1[1]
   end
end

;in fájl C.ctx
composite C
   context
      input float[3] i1[1]
      output int o1[3]
      output int o2[1]
      parameter float p1[1]
   end
end
```

```
;composite D
use P
use C
composite D
   context
      input int[5] i1[]
      output int o1[]
      parameter long p1[]
   end
   signals
      stream float[3] s1[]
      var    int v1
      const  float c1 3.14
   end
   actors
      primitive P a1
      composite C a2
   end
```

```
      topology
         a1.i1 << i1
         a1.p1 << p1
         a1.i2 << v1
         a1.o1 >> s1
         a2.i1 <1< s1
         a2.p1 << c1
         a2.o1 >> o1
         a2.o2 >> v1
      end
      schedule
   ;     manual a1
   ;        a1
   ;        do 4
   ;           a2
   ;        loop
   ;     end
         auto a1
      end
   end
```

Here is a simple composite declaration. The composite has only two actors and three signals. On the left-hand column you could see the interface declarations of the components. These declarations are in separate files. On the middle column you could see the use sentences and the interface, signal and actor declaration sections of the composite. Finally on the right hand column there are the topology and schedule declaration. Here the lines beginning with semicolon are comment line.

Signals and ports

signal_declaration ::= **signal_class SP signal_type**[**vector_size**] **SP signal_name**
[**vector_count**][**set_size**] **SP** {**initializator**}
signal_class ::= "stream" | "var" | "const"
signal_type ::= "char" | "short" | "int" | "long" | "float" | "double" |
"uchar" | "ushort" | "uint" | "ulong" | "string"
vector_size ::= "["**uint_literal**"]"
vector_count ::= "["**uint_literal**"]" | "[""]"
set_size ::= "{"**uint_literal**"}"
initializator ::= **long_literal** | **double_literal** | **character_literal** | **string_literal**
port_declaration ::= **port_class SP signal_type**[**vector_size**] **SP port_name**
[**vector_count**][**set_size**]
port_class ::= "input" | "output" | "parameter"

Examples:

```
stream   float[15]    s1[]{8}
input    double[1024] i1[3]
constant int          c1 3476
```

Now lets have a look on the signal and port declarations. This is the most complex part of the language. Signal and port declarations are similar. We have three signal classes: stream, variable and constant. Streams has the FIFO behavior discussed previously. Variables and constants are similar to the global variables in C and are omitted from the scheduling. They could be used for providing control parameters for the actors. The signals have type. We use the familiar C language types and the string type. The signals could be scalars or vectors. Scalars are one element long vectors. If the vector size is greater than 1 we have to specify it after the type declarator. We could specify the storage size by using the vector count after the signal identifier or we could left to the compiler to determine the storage size automatically. Finally we could declare a finite set of signals by the set size inside curly braces at the end of the sentence. We have three port classes: input, output and parameter. The type, the vector size and the set size should be matched by the corresponding properties of the signal connected to the port. The vector count here means the number of vectors the actor uses in a single execution. On the slide s1 is a signal set of 8 signals. The signals in the set are float vectors with vector length of 15. The storage size of the signals should be determined by the compiler. i1 is an input port which should be connected to a signal of float vectors with vector size of 1024. The actors consume 3 vectors during its fireing. c1 is a constant of int type with a value of 3476.

Signals and ports

context_section ::= "context" **NL**
port_declaration NL
{**port_declara**tion **NL**}
"end" **NL**
signal_section ::= "signals" **NL**
signal_declaration NL
{**signal_declaration NL**}
"end" **NL**

Example:

```
primitive P
    context
        input float[12] i1[3]
        output float[12] o1[3]{6}
        parameter int p1
    end
end
```

Signal declaration sentences must be used in the signal section. Port declaration sentences must be used in the context section in the composite or in the component interface declaration. On the slide you could see the interface declaration of the primitive component P.

Topology

connection_declaration ::=
(**port_name** (">>" | "<<") **composite_port_name**) |
(**port_name** SP (">>" | "<<") **SP signal_name**) |
(**input_port_name SP** ("<"**uint_literal**"<") **SP signal_name**) |
port_name ::= **actor_instance_name**"."**actor_port_name**
input_port_name ::= **actor_instance_name**"."**actor_input_port_name**
topology_section ::= "topology" **NL**
connection_declaration NL
{**connection_declaration NL**}
"end" NL

Example:

```
topology
    a1.i1 << i1
    a1.o1 >> s1
    a2.i1 <2< s1
    a2.o1 >> o1
end
```

In the topology section we declare the connections. We always declare to where an actor port is connected to. We could connect the actor port to a port of the composite or to a signal. The connection shows the direction of the signal flow and optionally the delay. For example port i1 of actor a1 is connected to the input port i1 of the composite. The port o1 of actor a1 is connected to the signal s1. The i1 port of actor a2 is connected to the signal of s1 through a delay of 2. Finally port o1 of actor a2 is connected to the o1 port of the composite.

Actors

primitive_declaration ::= "primitive" **SP primitive_name SP actor_instance_name**
composite_declaration ::= "composite" **SP composite_name SP actor_instance_name**
simple_actor_section ::= "actors" **NL**
(**primitive_declaration** | **composite_declaration**) **NL**
{(**primitive_declaration** | **composite_declaration**) **NL**}
"end" **NL**
switch_declaration ::= "switch" **SP switch_instance_name SP** "("**switch_variable_name**")" **NL**
context_section NL
simple_actor_section NL
[**signal_section**] **NL**
"topology" **NL**
{**case_section**} **NL**
default_section NL
"end" **NL**
"end" **NL**
case_section ::= "case" **SP** "("**integer_literal**")" **NL**
connection_declaration NL
{**connection_declaration NL**}
"end" **NL**
default_section ::= "default" **NL**
connection_declaration NL
{**connection_declaration NL**}
"end" **NL**

Actor declaration sentences should be used in the actors section. We have four actor classes: primitive, composite, switch and iterator. For primitive and composite actors we use a single sentence for declaration. In the sentence we declare the actor's class, the component's name and the actor's name. For switch and iterator classes the declaration uses multiple sentences. These multi-sentence declarations are in-line composite declarations with spetial extensions..

Actors

actor_section ::=
"actors" **NL**
(**primitive_declaration** | **composite_declaration** | **switch_declaration**) **NL**
{(**primitive_declaration** | **composite_declaration** | **switch_declaration**) **NL**}
"end" **NL**

```
signals
    ...
    variable int    swvar
end
actors
    primitive P1    a1
    switch sw1 (swvar)
        context
            input int[5] i1[2]
            output float[3] o1[4]
        end
        signals
            constant int c1 125
        end
        actors
            primitive P2 sa1
            composite C1 sa2
            composite C2 sa3
        end
```

```
        topology
            case (1)
                sa1.i1 << i1
                sa1.i2 << c1
                sa1.o1 >> o1
            end
            case (12)
                sa2.i1 << i1
                sa2.o1 >> o1
            end
            default
                sa3.i1 << i1
                sa3.o1 >> o1
            end
        end
    end
end
```

On this slide you should see a primitive actor and a switch declaration in the actor section. In the switch declaration we declare which variable controls the switch, the switch external interface, the optional signals used inside the switch, the actors used by the switch and finally how the actor are connected to the ports of the switch and optionally to the internal signals. Inside a switch you could use only primitive or composite actors, but not switches or iterators. The iterator declaration in concept similar to the switch declaration. The iterator uses set of signals in his inputs and / or outputs and in a single invocation it iterates through the signals of the sets. In each iteration step it could use the same or different actors.

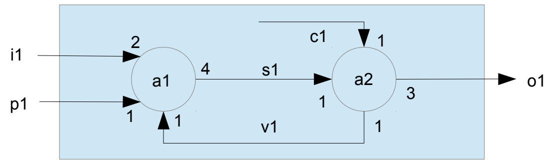
Schedule

```
shedule_element ::= actor_instance_name | loop_element
loop_element ::= "do" SP uint_literal NL
                         shedule_element NL
                         {shedule_element}
                 "loop"
manual_schedule ::= "manual" SP actor_instance_name NL
                         schedule_element NL
                         {scedule_element} NL
                    "end"
auto_schedule ::= "auto" SP actor_instance_name NL
schedule_section ::=  "scedule"
                         (manual_schedule | auto_schedule) NL
                         {(manual_schedule | auto_schedule) NL}
                      "end"
```

```
shedule
   auto a1
   manual a5
      a5
      do 3
         a6
         do 2
            a7
         loop
      loop
   end
end
```

We could have automatic or manual schedule. For each connected subgraph we should declare what kind of schedule we like to have. For manual schedule we should specify the execution sequence of the actors. The sequence specification could have loops.

Composite actor example




```
;in fájl P.ctx
primitive P
   context
      input int[5] i1[2]
      input int i2[1]
      output float[3] o1[4]
      parameter long p1[1]
   end
end

;in fájl C.ctx
composite C
   context
      input float[3] i1[1]
      output int o1[3]
      output int o2[1]
      parameter float p1[1]
   end
end
```

```
;composite D
use P
use C
composite D
   context
      input int[5] i1[]
      output int o1[]
      parameter long p1[]
   end
   signals
      stream float[3] s1[]
      variable int v1
      constant float c1 3.14
   end
   actors
      primitive P a1
      composite C a2
   end
```

```
   topology
      a1.i1 << i1
      a1.p1 << p1
      a1.i2 << v1
      a1.o1 >> s1
      a2.i1 <1< s1
      a2.p1 << c1
      a2.o1 >> o1
      a2.o2 >> v1
   end
   schedule
;     manual a1
;        a1
;        do 4
;           a2
;        loop
;     end
      auto a1
   end
end
```

Here is a declaration of a composite component. It uses two actors and three signals. It has three ports. You could see the the interface declarations of the components used. They are included in the composite declaration by the two use sentences. It is important to note, that in the interface declaration we must specify the vector counts, because the compiler must know the production and consumption behaviors of the actors. In the composite declaration on the other hand we leave the vector count blank to indicate, that the compiler should compute them according to the schedule. In the comment lines of the schedule section you see a manual schedule which is periodic admissible schedule.

Implementation

- Compiler
- Assembler
- Binary code structure
- Running the dataflow
- Primitive interface
- Virtual machine
- Composite interface

Today I have an implementation with a working compiler, assembler and runtime system. They run under 64 bits Linux operating system. I have an implementation of the runtime system, which runs on ARM Cortex-M4 in bare metal mode and I have tested the virtual machine on PIC32.

Compiler

- Line oriented, each line is a sentence
- Sentences are built from words
- Sentence processing:
  - scans for words
  - parses the sentence
  - checks the rules
  - adds items to the dataflow graph
- Consistency checking of the graph
- Finding the connected subgraphs
- Scheduling the subgraphs
  - solving the balance equation
  - computing the schedule by simulation
- Output assembler source code

The compiler is line oriented, each line is a sentence. The sentences are built from words. The working of the compiler is the following. The compiler scans the sentences for words. After that it parses the sentences, checks the rules and build the data flow graph sentence by sentence. After the graph has been built the compiler checks the consistency of the graph and searches for connected subgraphs. For each subgraph the compiler solves the balance equation and computes the schedule. Finally the compiler emits the assembly language source code and the interface declaration of the component.

Assembler

- Assembler source code platform independent
- Binary code platform dependent
- Line oriented syntax
- Uses the same parser the compiler uses
- Two passes
  - Scanning and parsing
  - Binary code generation
- Assembler code allows different data flow implementations

The assembly language code emitted by the compiler is platform independent. On the other hand the assembler generate platform specific code. The assembler is line oriented too. It uses the same parser the compiler uses. The assembler is a two pass assembler. It allows different data flow implementations.

Binary code

- 4 segments: meta, code, data, context
- context segment: defines pointer offsets
- meta segment:
  - symbolic information
  - initialized data values
  - code for loading component actors
  - code for actor and signal instance creation
  - code for deallocation resources
- code segment
  - code for initialization
  - code for scheduling
  - code for cleanup
- data segment
  - signals
  - actor instances
  - context structures

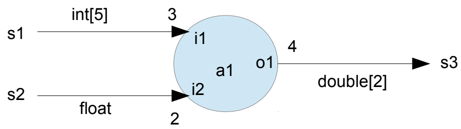
The binary code emitted by the assembler has four segments. The context segments specify the order of the interface pointers of the actor. The meta segment stores symbolic information and the code necessary to bootstrap the execution of the composite or the wrapup after the execution. The code segment contains the code necessary to run the schedule. Finally the data segments stores all the signals.

Running the dataflow

- Loading the top level composite's binary code
- Executing the component load code
  - Loading all component actors
  - Executing the component actor's load function
- Executing the instance creation (make) code
  - Creating actor and signal instances, context structures
  - Executing instance creation code of the component actors
- Executing the initialization code
  - Initialization for scheduling
  - Executing the initialization code of the components
- Executing the schedule
- Executing cleaup code
- Executing resource deallocation (delete) code

We run a composite in three stages. The first is the bootstrap stage. In this stage we load the components to be used if necessary, we create actors and signals and we initialize those signals and actors. The second stage is the running of the schedule. In this stage we execute the actors according to the schedule. The third stage is the wrap up stage. In this stage we cleans the actors, delete actors and signal and finally unload components if necessary.

Primitive interface




```
typedef struct _ctxA1
{
   int  *i1;
   float *i2;
   double *o1;
}ctxA1_t;

void fireA1(ctxA1_t *ctx);

int s1[30];
float s2[4];
double s3[16];

ctxA1_t ctxa1;
```

```
//initialization

ctxa1.i1 = &s1[0];
ctxa1.s2 = &s2[0];
ctxa1.o1 = &s3[0];

// fire a1

firea1(&ctxa1);

// increment context pointers

ctxa1.i1 += 3 * 5;
ctxa1.s2 = 2 * 1;
ctxa1.o1 = 4 * 2;
```

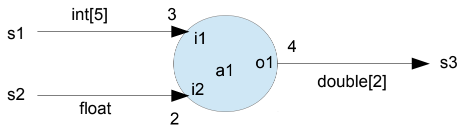
A primitive has six entry function. The load, make, clean and delete function are component level and the init and fire functions are instance level functions. In the load function the primitive could allocate component level resources. These resources shouild be deallocated in the delete function. The make function is used for instance creation and the clean function is used for deleting an instance.The init function do the initialization for fireing and the fire function execute the algorithm of the primitive. Each function get a single pointer. It points to a structure which contains the pointers to the signals connected to the actor ports. From the starting address specified by a signal pointer the primitive could reach the number of vectors declared in the interface declaration. The layout of the vectors are the same as the layout of a two dimensional array in C. The vectors are the rows of the array. At the beginning of an execution period the pointers are initialize to the beginning of the signal's storage buffer. After the execution the pointers are incremented according to the number of vectors used by the actor. Today we are not using circular buffers to reduce the storage size of the signals.

Virtual machine

```
vm_run(vm_t *vm)
{
   int *ip=vm->ip;
   static void *instructions[NR_OF_INSTRUCTIONS] =
   {
      [INST_FIRST] = &&inst_first,
      // ...
      [INST_n] = &&inst_n,
      //...
      [INST_LAST] = &&inst_last
   };

   goto instructions[*ip++];
   return;

inst_first:
   // the code for inst_first goes here
   goto instructions[*ip++];
// other instructions
inst_n:
   // the code for inst_n goes here
   goto instructions[*ip++];
// other instructions
inst_last:
   // the code for inst_last goes here
   goto instructions[*ip++];

   return;
}
```

The binary code emitted by the assembler should be executed by a virtual machine. The virtual machine is a threaded code virtual machine. It uses the labels as values feature of the gcc compiler. I choose this because it is more friendly to the processors branch prediction algorithms than the use of the switch statement of the C language. The labels stored in a static array defined inside of a C function. The array elements could be used for the target of a goto statement.

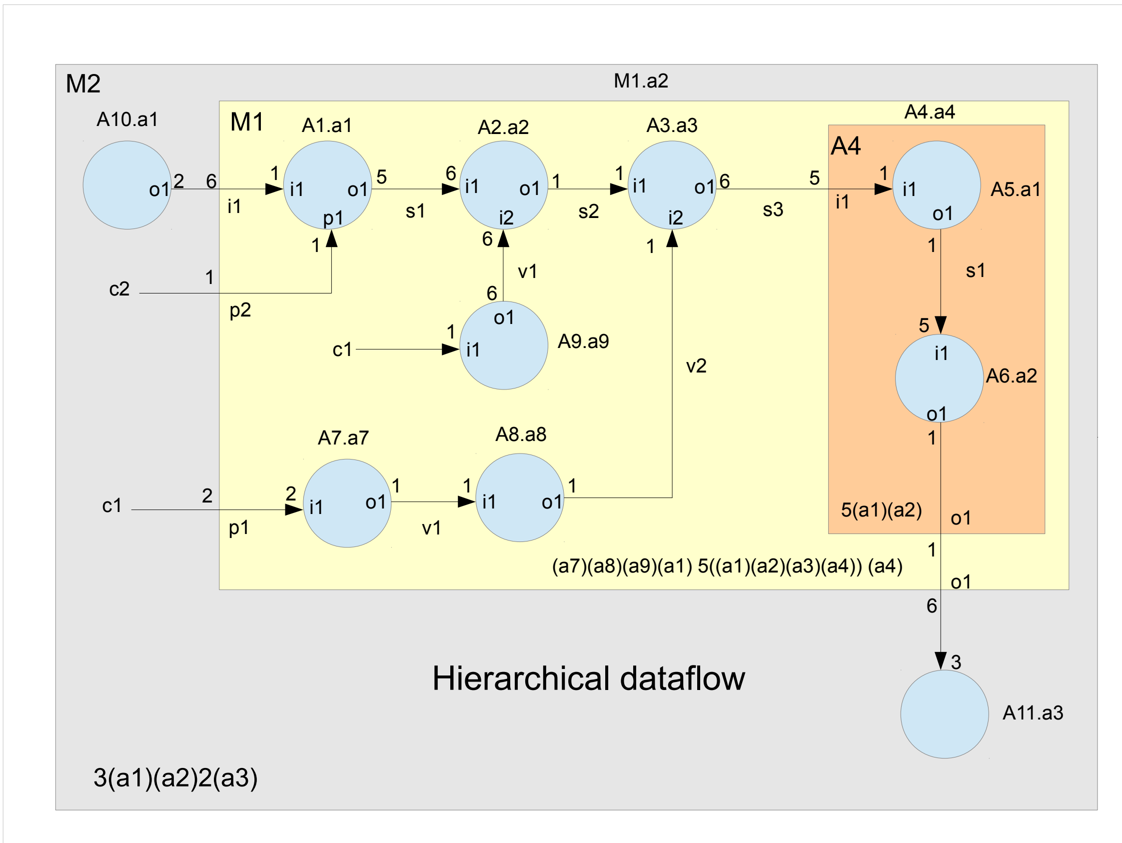
Composite interface




```
typedef struct _dC
{
   struct _ctxA1
   {
      int  *i1;
      float *i2;
      double *o1;
   } ctxA1;
   struct _inst_a1
   {
      void **dseg;
      void **ctx;
   } insta1;
   int s1[30];
   float s2[4];
   double s3[16];
} dC_t;
typedef struct _fire_composite
{
   int instruction_code;
   int instance_offset;
} fire_composite_t;
```

```
// from the vm_run function

   void **p, **data;
   void **context, **sp;
   int *ip;

fire_composit:
   p = data + *ip;
   *(--sp) = data;
   *(--sp) = context;
   (--sp) = (void**) ip;
   data = *p++;
   context = *p;
   ip = *((int **) data);
   goto instructions[*ip++];
ret:
   ip = (int *) *sp++;
   context = *sp++;
   data = *sp++;
   goto instructions[ip++];
```

The composite interface is similar to the primitive interface. It has the same functions that a primitive has.The composite gets a pointer called context which points to a pointer array of the interface signals. The invocation of the function is done by the virtual machine. The virtual machine saves the current context, data and instruction pointers into a stack and loads the new context, data and instruction pointers and continues the execution. The ret instruction restores the saved pointers.



On the slide there is a hierarchical composite. The top level has no input or output, so it could be run by the runtime system. The M1 composite has a nontrivial schedule. All the schedules have been computed by the compiler. The M1 has two subgraphs. The subgraph which contains the a1, a2, a3 and a4 actors is a connected subgraph in which the actors are connected by streams. The remaining three actors form a virtual subgraph in which the actors are connected through variables and / or constants or they are standalone actors. The purpose of such a subgraph is to implement some computation on control parameters. This subgraph is scheduled by a different algorithm and each actors are executed only once and executed before the other subgraphs. This subgraph could not has loops, we should be able to arange the actors in a topological order.

```
use A5
use A6

composite    A4
    context
        input    float[1] i1[]
        output   float[5] o1[]
    end
    signals
        stream   int[1]   s1[]
    end
    actors
        primitive    A5   a1
        primitive    A6   a2
    end
    topology
        a1.i1 << i1
        a1.o1 >> s1
        a2.i1 << s1
        a2.o1 >> o1
    end
    schedule
        auto a1
    end
end
```

```
use M1
use A10
use A11

composite    M2
    context
    end
    signals
        stream       float[5] s1
        stream       float[5] s2
        const        int[1]   c1[2] 1 2
        const        int[5]   c2[1] 1 2 3 4 5
    end
    actors
        primitive    A10 a1
        composite    M1  a2
        primitive    A11 a3
    end
    topology
        a1.o1    >>   s1
        a2.i1    <<   s1
        a2.o1    >>   s2
        a2.p1    <<   c1
        a2.p2    <<   c2
        a3.i1    <<   s2
    end
    schedule
        auto a1
    end
end
```

This slide shows the declaration of the A4 and M2 composites.

```
use A1
use A2
use A3
use A4
use A7
use A8
use A9
composite    M1
    context
        input        float[5] i1[]
        output       float[5] o1[]
        parameter    int[1]       p1[2]
        parameter    int[5]       p2[1]
    end
    signals
        stream   float[1] s1[]
        stream   float[1] s2[]
        stream   float[1] s3[]
        var      int[3]       v1[1]
        var      float[1] v2[1]
        var      int[1]       v3[6]
    end
    actors
        primitive    A1   a1
        primitive    A2   a2
        primitive    A3   a3
        composite    A4   a4
        primitive    A7   a7
        primitive    A8   a8
        primitive    A9   a9
    end
```

```
    topology
        a1.i1    <<   i1
        a1.o1    >>   s1
        a1.p1    <<   p2
        a2.i1    <<   s1
        a2.i2    <<   v3
        a2.o1    >>   s2
        a3.i1    <<   s2
        a3.i2    <<   v2
        a3.o1    >>   s3
        a4.i1    <<   s3
        a4.o1    >>   o1
        a7.i1    <<   p1
        a7.o1    >>   v1
        a8.i1    <<   v1
        a8.o1    >>   v2
        a9.i1    <<   v1
        a9.o1    >>   v3
    end
    schedule

        auto a7
        auto a1
    end
end
```

This slide shows the declaration of the M1 composite.

```
         .meta
.name    string   "A4"
.version uint 00000001
A5.n     string   "A5"
A6.n     string   "A6"
a1.n     string   "a1"
a2.n     string   "a2"
s1.n     string   "s1"
i1.n     string   "i1"
o1.n     string   "o1"
.load
         ld.prim  A5.n a1.n
         ld.prim  A6.n a2.n
         meta.exit
.make
         mk.prim.inst A5.n a1.n a1
         mk.prim.inst A6.n a2.n a2
         mk.buffer    int[5] s1.n s1.p
         meta.exit
.delete
         meta.exit
         .endseg

         .context
i1       ptr
o1       ptr
         .endseg

         .code
         exit
```

```
.init
     cp.ctx.ptr     a1.i1    i1
     cp.ptr         a1.o1    s1.p
     cp.ptr         a2.i1    s1.p
     cp.ctx.ptr     a2.o1    o1
     init.prim      a1
     init.prim      a2
     ret
     end.cycle
.fire
     cp.ctx.ptr     a1.i1    i1
     cp.ptr         a1.o1    s1.p
     cp.ptr         a2.i1    s1.p
     cp.ctx.ptr     a2.o1    o1
     do             5
.l1
     fire.prim      a1
     inc.ptr        a1.i1    4
     inc.ptr        a1.o1    4
     loop           .l1
     fire.prim      a2
     inc.ptr        a2.i1    20
     inc.ptr        a2.o1    20
     ret
     exit
.clean
     cp.ctx.ptr     a1.i1    i1
     cp.ptr         a1.o1    s1.p
     cp.ptr         a2.i1    s1.p
     cp.ctx.ptr     a2.o1    o1
     cleanup.prim   a1
     cleanup.prim   a2
     ret
     .endseg
```

This slide shows the meta and code segments in the generated assembly code for the A4 composite.

```
        .data
s1.pptr s1
s1      int[5]
a1      ptr
a1.i1   ptr
a1.o1   ptr
a2      ptr
a2.i1   ptr
a2.o1   ptr
        .endseg
```

```
    .meta
;catalog start
    char     's'      ;signature
    char     'd'
    char     'f'
    char     0        ;platform
    address  .name    ;name offset
    address  .version ;version offset
    address  .size    ;object code size
    address  .code.offset
    address  .data.offset
    int      0        ;reserved
    int      0        ;reserved
;catalog end
;meta header
    address  .load
    address  .make
    address  .delete
    .endseg
    .code        ;code header
    address  .fire
    address  .init
    address  .clean
    .endseg
    .data        ;data header
    ptr      0   ;pointer to .fire
    ptr      0   ;pointer to .init
    ptr      0   ;pointer to .clean
    ptr      0   ;pointer to meta seg
    int      0   ;make flag
    int      0   ;reserved
    .endseg
```

This slide shows the data segment in the generated code for the A4 composite and the headers inserted into the code by the assembler.

```
schedule
    a7
    a8
    a9
    a1
    do 5
    a1
    a2
    a3
    a4
    loop
    a4
end
```

```
schedule
    do 5
    a1
    loop
    a2
end
```

```
schedule
    do 3
    a1
    loop
    a2
    do 2
    a3
    loop
end
```

This slide shows the compiler computed schedules for the M1, M2 and A4 composites.