IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D44 (WP8): Costing-by-construction Exemplar

Due date of deliverable: February 2009
Actual submission date: March 2009

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: LASMEA Revision: 1.13

Purpose: The purpose of this deliverable is to evaluate the Hume costing-by-construction methodology by applying it to a substantial exemplar.

Results: The main result of this deliverable is the development of a digital filter application for the Renesas M32C microcontroller guided by the methodology illustrating its effectiveness for designing, implementing and validating embedded systems.

Conclusion: We conclude that our methodology provides significant support for the development of embedded systems for which resource constraints are a significant factor.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⁎
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Contents

Costing-by-construction Exemplar

Norman Scaife <Norman.Scaife@lasmea.univ-bpclermont.fr>,
Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>,
Greg Michaelson <G.Michaelson@hw.ac.uk>,
Jocelyn Sérot <Jocelyn.Serot@lasmea.univ-bpclermont.fr>

Abstract

In this document we present an evaluation of the Hume programming language methodology for embedded and real-time systems. This is achieved by the development of a relatively simple but realistic application program using the methodology.

1 Introduction

We have developed the Hume programming language methodology as part of the EmBounded project, and presented it in Deliverable D27 WP8b [SS08]. Furthermore, as part of the evaluation of the Hume language in Deliverable D33 WP8c [SLMS08] we presented partial analysis of some of the applications described therein using this methodology. The methodology itself is described in detail in the former deliverable and summarized again in the latter so we do not present the methodology again here other than to name and number the individual steps in order to index them in this deliverable.

Missing from the applications we have described previously is a complete worked example using the resource analysis and guided by the methodology. Thus we have developed a new application, the implementation of a biquadratic filter[1] for the Renesas M32C, for which we have the required cost tables but which is somewhat restricted in both processing and memory resources. For these reasons this application is both highly relevant to Hume's resource analysis but also limits the complexity of the application we can tackle. The filter application is a compromise between these two conflicting motivations.

2 Methodology

As we have stated in previous documents, the key novel aspect of our methodology is the integration of resource analysis into the entire development process. Our principal advantage over traditional methods is that we allow design decisions concerning resource usage (memory, processing) to be made at an early point in the design process. The other major advantage is that we can also provide formal guarantees about such resources giving confidence in existing code in the event of changing circumstances, for example in the case of reuse of guaranteed components in later design modifications. By attaching cost information to types in the program, our method is modular and we can provide guarantees for individual functions and boxes allowing constructive use of these Hume elements as components.

In summary, our methodology is formally-grounded, refinement-based, cost-driven and relates to the language levels provided by Hume. Thus we have three phases: an initial *system-level* phase, where overall system

[1] also known as a *biquad* filter.

characteristics are defined; a *box-level* phase, where the box structure is elaborated; and a final *validation* phase, in which we ensure compliance with the cost analysis. In the system-level phase, we determine properties of the system as a whole, designing a network of interconnected boxes, but do not define any box functionality. In the box-level phase, we then proceed to define the functionality of each box before obtaining cost analysis information which is used to drive subsequent refinement of and improvements to the design.

In this description we use the term "property" in its most abstract sense: our primary focus here is on resource bounds constraints encoded as extended types. Here, we summarize the basic steps in the methodology, see EmBounded deliverable D27 WP8b [SS08] for a full description.

2.1 Phase 1: System-level Phase.

We first determine the overall characteristics of the system.

(**1**) Ascertain the overall resource requirements for the whole program.

(**2**) Consider external characteristics of the inputs/outputs.

At this point we arrive at a *system-level* characterisation of the system. We can now proceed to consider the program structure in terms of boxes and types.

(**3**) Define the box structure for the inputs and outputs.

(**4**) Based on (**2**), add timing constraints for each input and output.

(**5**) Outline the overall box structure and I/O box links.

(**6**) Refine box structure, if necessary.

(**7**) Define any required types and datatypes.

At this point, we arrive at a *black-box* characterisation of the system. Most importantly, types and resource requirements are attached to the wires. The content of the box is left unspecified at this stage.

2.2 Phase 2: Box-level Phase.

In this phase, we specify the contents of each box.

(**8**) Specify the functionality of each box.

(**9**) Revisit the design, if necessary, looping back to Step (**5**).

(**10**) Implement the top-level functionality for each box.

(**11**) Implement functions that successively refine the functionality of each box.

(**12**) Apply costing.

At this point, we arrive at a *white-box* characterisation of the system in which we specify the type and resource information as well as the functional behaviour of each box.

2.3 Phase 3: Validation Phase.

The final validation steps are driven by the results of the cost analyses.

(**13**) Refine the functionality to respect costs. It may involve:

- returning to Step (**11**) and adapting functions so that they are more time/space efficient;
- returning to Step (**8**) and revising the box functionality with a similar objective; or
- refining the top-level box structure, Step (**6**), or modifying the overall box structure Step (**5**), perhaps by replacing complex boxes with smaller, easier to cost, boxes.

(**14**) If the constraints still can't be met:

- Relax margins on constraints, if this is acceptable.
- Revise the application specification.
- Update the hardware requirements.

(**15**) If it is still not possible to meet the costing requirements, then the application design criteria must be revised and the design process restarted, taking on board any relevant lessons from the original, failed design.

3 The Biquadratic filter application

3.1 The biquad filter

The biquadratic filter is a second-order recursive linear filter and is so named because the transfer function is a ratio of two quadratic polynomials. It is commonly used in audio work and being quite general, depending on the filter coefficients, can be used to implement low-pass, high-pass, band-pass and notch filters. Well-known recipes exist for computing these coefficients from the desired gain, centre frequency and sample rate of the application [KLT06].

The digital implementation can be simply expressed as (direct form 1 difference equation):

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} - a_1 y_{n-1} - a_2 y_{n-2}$$

For example, if we express this in the following form:

$$y_n = \frac{b_0}{a_0} x_n + \frac{b_1}{a_0} x_{n-1} + \frac{b_2}{a_0} x_{n-2} - \frac{a_1}{a_0} y_{n-1} - \frac{a_2}{a_0} y_{n-2} \quad (1)$$

we can derive the coefficients for a low-pass filter (LPF) as follows:

$$\begin{aligned} w_0 &= 2\pi f_0 / F_s \\ \alpha &= sin(w_0)/(2Q) \\ b0 &= (1 - cos(w_0))/2 \\ b1 &= 1 - cos(w_0) \\ b2 &= (1 - cos(w_0))/2 \\ a0 &= 1 + \alpha \\ a1 &= -2cos(w_0) \\ a2 &= 1 - \alpha \end{aligned}$$

where Q is the Q factor, f_0 is the centre frequency and F_s is the sample frequency.

Simple sets of equations exist for other filter types.

3.2 The Hume M32C biquad filter implementation

Our application is an implementation of such a filter on the M32C using the on-board resources. The idea is to read the input from the analogue to digital converter (A2D), possibly perform some pre-processing such as scaling and limiting, apply the filter and then output the filtered signal to the digital to analogue outputs (D2A), again with possible post-processing on the signal.

It should be noted here that the A2D/D2A on the M32C is not of a sufficiently high quality for audio work, either in sample rate, resolution or signal-to-noise ratio so our filter could only be used for low-rate data streams requiring no more than about 8 to 10 bits of accuracy.

Figure 1 is a block diagram of our intended application. The circles represent M32C microcontroller resources, solid boxes are functional modules which may be represented as one or more Hume boxes in the final application and dashed boxes are additional modules included for debug purposes during development. The thick arrows represent the critical path through the system.

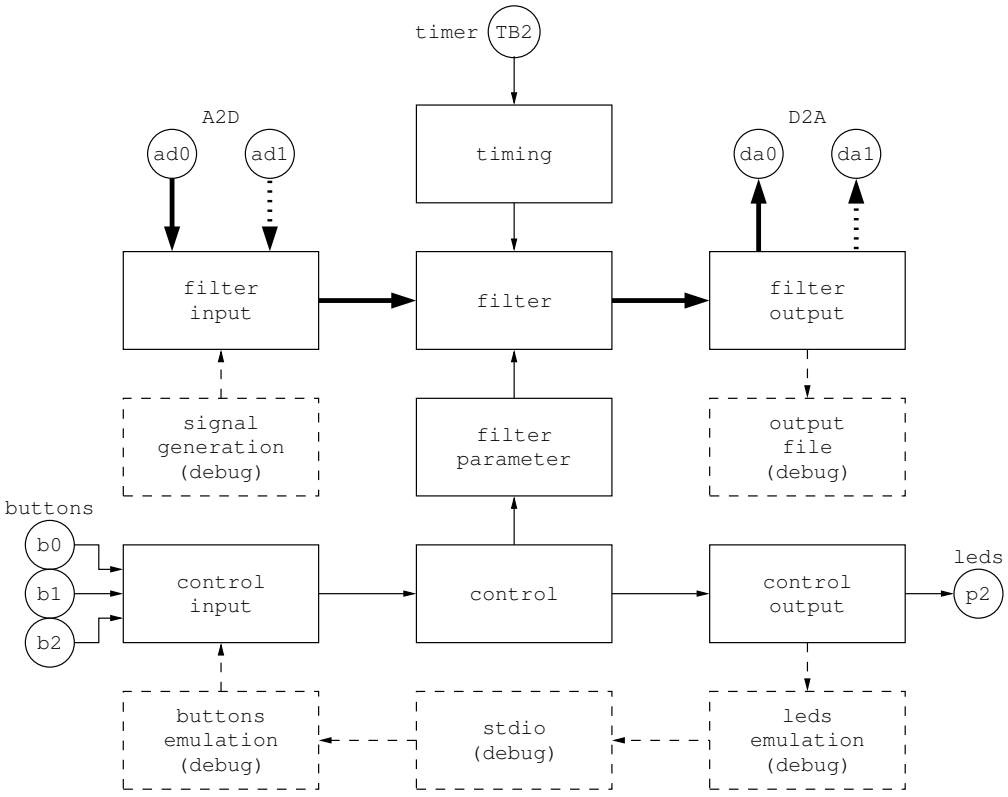

Figure 1: Biquad filter M32 application block diagram

The M32C resources required are listed in Table1. We are limited by the fact that the only input/output available outside of debugger support are the three buttons linked to the INT0 to INT2 interrupt pins and the eight LEDs connected to the 8-bit P2 output port. However, this gives us enough scope for providing the user of the filter with some configuration options and some status information. During debugging we replace these resources with emulated features from the standard IO on the host system. This allows a much faster edit/compile/debug cycle.

The main filter operation itself is between the AD0 analogue to digital input and the DA0 digital to analogue output. However, the M32C actually has two D2A outputs and two A2D converters which can be multiplexed

between up to 16 inputs where the sample rate decreases as more channels are multiplexed. Thus we have the option of implementing a second filter on the same board. However, we found that in practice the code size became too large when this second channel was implemented.

Finally, for filter accuracy we clock the filter using the on-board timer TB2. We actually use the combination of TB0/TB1 for timing measurements during our experiments but this is invisible to the application program.

AD0	Digital input to filter
DA0	Digital output from filter
AD1	Optional digital input for second channel
DA1	Optional digital output for second channel
B0,B1,B2	Three buttons linked to INT0 to INT2 interrupts
P2	8-bit parallel port connected to 8 LEDs
TB2	On-board timer configured to generate periodic interrupts

Table 1: M32C resources for biquad filter application

3.2.1 Functionality

The block diagram in Figure 1 constitutes a loose functional specification for our project. Here, we describe the basic functionality required by each module. In the next section we will proceed to apply our methodology to this specification.

Note that we use the term "functionality" in its loosest sense, here. We specify not just the functionality of the intended running components in the final implementation but also aspects of the source code required to arrive at that implementation, including configuration, initialisation and other support code such as types and constants.

3.2.1.1 Configuration module Even in a small project such as the biquad filter implementation we may require different versions of the code either for slightly different application environments, to allow room for expansion/contraction of the design during development, allow experimentation with features which may or may not be useful in the final version or simply to provide debug support during development. Hume does not have any inbuilt features for version control such as this and, in fact, hinders and discourages the development of multiple versions of the same application in unified source code due to the relatively inflexible wiring syntax and the current experimental status of the graphical tools. However, the current implementation of Hume does allow the use of the C pre-processor during parsing of Hume programs which allows version control using the familiar `#define` directives. This is useful but does tend to obfuscate the code somewhat. Future versions of Hume will have better library support and graphical tools to aid this aspect of system development. We create this *configuration module* to house such code.

3.2.1.2 Streams module This is an important aspect of Hume programs since all input/output is via streams. Even interrupt sources and external memory ports are treated as streams. It is a matter of user preference as to whether the stream definitions in a Hume program are placed next to the wiring statements to which they belong or in a single block elsewhere in the code. Our methodology does not preclude either of these styles but in keeping with our philosophy of having a "module" for all aspects of the code, we implement a streams module which will be modified as each computational module is developed.

3.2.1.3 Types module Our methodology explicitly recommends that types for a given component are devised at an early point during development to allow the partial specification of boxes during the refinement process.

Again, these could be attached to individual modules but we contain them in a single *types module* for this project.

Note that we exclude from this module the types which are created to define loopback variables for boxes. These are rather personal to the individual module and are rarely used elsewhere so we attach them to the module to which they pertain.

3.2.1.4 Constants module This is not a functional module in the final implementation but at some stage we may need to specify constants for the program. This may be overall parameters for the application or merely mathematical constants to be used in computations. Sometimes, however, in developing Hume programs we find we need to implement a box or function to compute these values if they are complex (for example, the initial model for the CyCab application program described in D27 WP8b requires a couple of hundred lines of code to compute despite the fact that it is only used as an `initially` value for one of the boxes.) This code also has to be treated as part of the methodology so we create this "dummy" module as a catch-all for such code.

3.2.1.5 Signal generation (debug) module This is the first of the actual code modules. Here we generate a synthetic input signal intended for supporting debugging of the final implementation. The signal can be generated internally, for example a sine wave, or can be read from a file. As with all the debug modules, this can be elided from the sources using a suitable pre-processor definition. In Hume terms, we envisage a single box to either generate the signal data or read data from the file and convert into a suitable signal format. The type of input signal is also controlled by a C pre-processor directive.

3.2.1.6 Timing module This module is only required to provide a dummy value on a single output wire at a rate which specifies the sample rate for the filter. If the filter were run unconstrained it will have a natural frequency which would be difficult to predict and may fluctuate depending on other circumstances in the running system (for example if the input dropped to zeros the resulting arithmetical operations on input values would speed up). For the coefficients we have computed to remain valid we require a predictable and stable sample rate.

For the M32C there are a number of timers which can be configured in different modes (for example interval timers or periodic timers). Some of these are linked internally to other on-board functionality on the M32C but we don't actually require any of these. However, we wish to determine the accuracy of our predictions independently of the timer module in the application program so we reserve timers TB0 and TB1 for timing measurements. These can be linked together to give a 64-bit decrementing counter and thus be used as a wall-clock albeit with a small amount of drift due to reading the counter which has to be stopped between readings. We then setup TB2 as a periodic timer at the sample rate we require.

When we run the filter code on the host system under debug, we actually implement an auxiliary box (`box timer`) which reads the wall clock and generates the timer signal based on elapsed time. We could, in theory, generate a timer signal from timers implemented by the operating system but this has not been implemented as part of the current Hume compiler implementation. In fact, on the M32C we can elide this box altogether and just connect the timer interrupt stream directly to the main filtering box.

Thus, for the M32C we only require that the timer source is setup at the required frequency (this is done in the startup code external to the Hume program) and for the debug version we envisage a single Hume box with the Hume clock stream as input and an emulated interrupt as output.

3.2.1.7 Filter input module The function of this module is to condition the input signal suitable for passing through the filter. Since we do not know the characteristics of the input signal in advance we attempt to implement a simple gain and offset adjust function in this module. Normally, these would be set to unitary gain and zero offset but, in practice, we can improve the numerical performance of the filter if we provide a virtual signal

from $[0.0, 1.0)$ derived from a possibly small range of bit values in the $[0, 255]$ 8-bit input value from the A2D. In practice, this type of gain adjust would be done using high-quality op-amps on the input itself.

Apart from the simple numerical scaling and type-conversion functions, this module is required to input control signals from the control module, independently from the scaling function.

3.2.1.8 Filter output module Strictly speaking we do not need to implement scaling on output since it should follow the input scaling. In practice, the parameters computed for the filters are not necessarily neutral gain. Adding a scaling function to the output is also trivial and can be viewed as a primitive "gain control" for the overall filter implementation.

More seriously, it can be difficult to display the output sensibly during debug depending on how narrow the range of output is so we implement a simple auto-gain facility which ensures peak output in $[0, 255]$. This can be viewed as a debug facility.

3.2.1.9 Filter module Despite being the central object of the application, the implementation of the filter itself is trivial. It is just a direct implementation of Equation 1, the entire module can be expressed in a small Hume box:

```
| ((),sample,*,(a0,a1,a2,a3,a4,x1,x2,y1,y2)) ->
    let result = (a0 * sample) + (a1 * x1) + (a2 * x2) - (a3 * y1) - (a4 * y2) in
      (result,(a0,a1,a2,a3,a4,sample,x1,result,y1));
```

Controlling this box is accomplished by completely resetting the whole internal state from external control sources (this may result in spikes and unwanted harmonics when the filter is switched!). Note, however, that we have templated this box in order to allow future implementations of other filter streams in the same application (for example, the `AD1` to `DA1` filter channel possible on the M32C).

3.2.1.10 Filter parameter module Although mathematically complex, this module is a straightforward implementation of the recipes for computing the filter parameters from the desired gain and timing properties. We may wish to split up the functionality, however, since the computation of the basic parameters should be shared between the different sets of filter parameter equations. We would prefer to be able to adjust the external filter parameters from the control module but three potentially widely-ranging floating-point values cannot easily be managed by a three-button input and 8-led output display interface so we content ourselves with simply switching the filter type between a set of pre-defined filter parameters.

3.2.1.11 Control module In our much-reduced functionality the control module is only required to manage the switching of filters and the adjustment of input/output gains. This module is therefore just a coordination module between the user interface and the filter functionality. We simply define a protocol between the user interface and the filter modules whereby three commands are defined which are then routed to the correct module (filter input, filter output or filter implementation):

```
data control_command =
   NextFilter
 | AdjustOutput (smp_type)
 | AdjustInput (smp_type);
```

note that `smp_type` is the type of the data stream associated with the inputs and outputs (actually `float 32`).

3.2.1.12 Control input module This module takes abstract input (ie. either from the button interrupts themselves or from buttons emulated by standard input) and translates this into commands for the control module. Here we maintain the internal state of the application (normal run mode, or various input modes allowing setting of the few parameters which need to be altered). The state is encapsulated in the following datatype:

```
data mode = RunMode | SetFilterMode | SetInputMode | SetOutputMode;
```

The button functionality is then as shown in Table 2. In fact, we can turn this table into a set of box patterns to implement this module.

Button	Mode	Command	Switch to
B1	RunMode		SetFilterMode
B1	SetFilterMode		SetInputMode
B1	SetInputMode		SetOutputMode
B1	SetOutputMode		RunMode
B2	SetFilterMode	NextFilter	SetFilterMode
B2	SetOutputMode	(AdjustOutput 1.1)	SetOutputMode
B3	SetOutputMode	(AdjustOutput 0.9)	SetOutputMode
B2	SetInputMode	(AdjustInput 1.1)	SetInputMode
B3	SetInputMode	(AdjustInput 0.9)	SetInputMode

Table 2: Button functionality

3.2.1.13 Control output module This module gathers status information from all other parts of the application and manages the status display in the form of the 8 LEDs on port `P2`. While computationally very simple (mostly just consisting of bitwise operators generating the output pattern which is written on the output port) the wiring associated with this module is relatively complex. This is complicated by the various `#define` directives which can elide various sources of status information. The allocation of LED segments to status information is therefore greatly complicated by these directives and, in addition, this module is highly dependent on other parts of the code, so changing almost any other part of the program will require changes here. Figure 2 shows the allocation of LEDs to status information for the case where only four filters are implemented.

3.2.1.14 Stdio emulation (debug) module This debug module has two functions, it takes the characters '1', '2' and '3' on the input stream and generates the same signals as for the three buttons when they are implemented by interrupts on the board and it also turns the output byte written to the LEDs port into a text string for debug on the host system.

4 Applying the Methodology

In this section, we apply the Hume programming methodology to the rough specification we have presented. The target platform is the Renesas M32C processor in the form of a development board which has the input/output resources listed in Table 1. In addition, we have 384k of flash ROM and 32k of static RAM in which we have to install our entire program: code, data and stack. Note that using our abstract machine HAM for execution means that the Hume heap and stack are embedded in the program's data section but that we still need a small C stack for function calls.

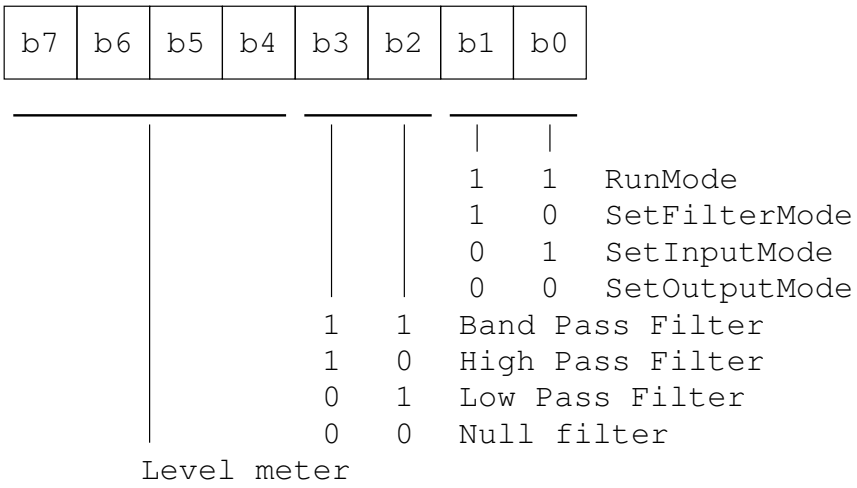

Figure 2: Biquad filter M32 application LEDs allocation

4.1 Step (1): Overall resource requirements

For this application, the 384k of ROM is more than sufficient to store the biquadratic filter code. However, we have a practical limit on the size of the code engendered by the Renesas compiler suite. The C code generated by `humec` is in the form of a single function containing a large number of labels and `goto` statements. Some compilers have problems with this type of code, most probably a complexity problem with resolving branches. The Renesas compiler can only compile about 6 to 7 thousand lines of code (loc) of this type before compilation times become unusably long. By design rather than accident, the biquad filter generates about 6k loc and is thus suitable for this target platform and target compiler.

More seriously, we are required to place the entire memory requirement of the application into the 32k of static RAM. This includes the program data segment plus a small amount of C stack used by the host C code. The Hume resource analysis is able to provide us with upper bounds on heap and stack usage within the Hume executable. This accounts for most of the memory required by the application but we also need a small margin for:

- The startup code requires a small amount of stack to call functions during initialisation and to call the main entry point to the Hume code.
- Once running, the Hume code is structured in a very flat manner but needs some C stack to call I/O functions, generate exceptions, create heap data and other "housekeeping" functions as well as any low-level externally implemented C functions such as copying memory.
- If interrupts are used, the M32C uses a separate interrupt stack which needs to be placed externally to the Hume program's memory.

In fact, since we do not call user-defined external functions, the sum total of these incidental memory requirements is not more than 1k of RAM.

If the filter we are designing were required for audio data streams we would have to run the filter at a frequency of about 48kHz. Unfortunately, this gives us about 20μs to implement the main loop which is about 666 clock cycles at the speed of the M32C. The quality of the analogue interfaces is also neither fast or accurate enough for such work so we are designing a low-grade data filter, not a high-quality audio interface. We thus do not place timing constraints on the overall loop but there are other timing constraints within the program which have to be met.

4.2 Step (2): External characteristics of inputs/outputs

In order to design the external interfaces to the application, we need to consider the characteristics of the inputs and outputs. These are:

- Analogue input. A/D conversion time is about 2μs for 10-bit conversion so we just read the input port without testing for completed conversion. This value will always be available on the Hume wire to which it is attached. Physically, the value is derived from a 16-bit memory-mapped register and the only processing required in the Hume runtime is to copy this value into a heap cell.
- Analogue output. Similarly for D/A the settling time is about 3μs so again we can just output without polling for ready status. The analogue inputs/outputs will easily run at the rate of the filter box and the output routine is just the reverse of the read routine, a heap cell is copied to an absolute memory address.
- Button input. Here we use interrupts linked to each of the three buttons on the board. This means we need to factor in the interrupt stack in our overall memory requirement but the interrupt service routines needed to service these interrupts are very small and use minimal memory. The interrupt service routine sets a flag which is checked by an input routine which in turn places a unit value on the heap. Since the input to the Hume box receiving the interrupt signal is just a unit value, we do not use significant memory in processing these inputs aside from the resources monitored by the analysis. Note also that these inputs are asynchronous and so there are no hard real-time deadlines associated with them.
- Parallel port output. This looks like an 8-bit value in absolute memory and output is asynchronous.
- Timer input. This is an asynchronous interrupt input which consumes similar resources to those described above. There is a hard real-time deadline associated with the sample rate we desire for the filter but this is guaranteed externally by the timer interrupt. We then just have to validate the fact that one superstep does not exceed the sample rate and this can be provided by our analysis.

All of these inputs and outputs are low-level values with no real-time constraints apart from the timer input. We can thus create our *system-level characterisation* for the application. To create a modular design we will assume that each input and output is placed on an individual box, apart from the buttons which we will keep together. Given the nature of our inputs and outputs the system-level characterisation is relatively simple and depicted in Figure 3.

4.3 Step (3): Define the box structure for inputs/outputs

As stated we have decided upon a single box for each input. This has several advantages from the analysis point of view since the costs associated with each input box are kept separate allowing problems with individual inputs to be more easily isolated.

4.3.1 Filter input module

We can thus write the box headers for input and output boxes, for example the input box attached to `AD0` will be:

```
{- Filter input module -}
template scale_in
  in (b::byte, ctrl::scale_control, ss::scale_control)
 out (sample::smp_type, ss'::scale_control)
```

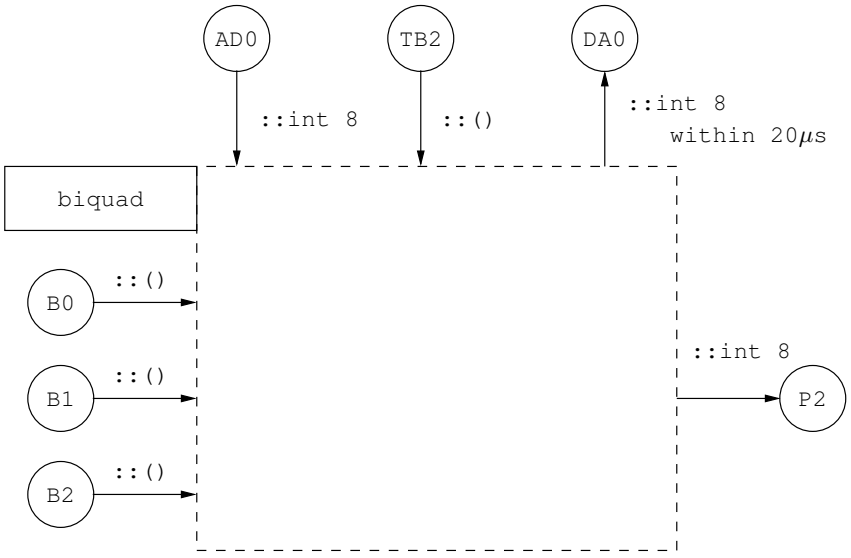

Figure 3: Biquad filter system-level characterisation

We take the input value from the A/D port and provide a value suitable for the control algorithm, which we keep abstract at this stage (`smp_type`). We assume that this box has some kind of state which stores parameterisation and configuration data for the input. This data will need to be controlled from somewhere else in the program so we add an input to allow this data to be written externally. Note that the input type is declared as `byte` despite the fact that the output port is 16-bits wide. In fact, we are only writing the top 8 bits of the port.

4.3.2 Filter output module

The output to `DA1` is almost the reverse of the input:

```
{- Filter output module -}
template scale_out
  in (result::smp_type, ctrl::scale_control, ss::scale_control)
 out (b::byte, level::byte, ss'::scale_control)
```

The difference here is that we also have to provide an output for the status display which is just a copy of the value sent to the D/A port.

4.3.3 Filter module

Using similar reasoning, we arrive at the following header for the timer input which we know will be the filter box:

```
{- Filter module -}
template biquad
   in (clk::(), sample::smp_type, ctrl::biquad_t, bq::biquad_t)
  out (result::smp_type, bq'::biquad_t)
```

Again, we have the internal state plus an input which allows it to be adjusted. We also include the processed input from the A/D module and the output to the processing box for the D/A module. If we were worried about the speed of the sampling for the filter, we could incorporate the A/D and D/A directly into the filter box. However, there is a complex relationship between the speed the box executes and the number of input patterns. It is not necessarily true that it would execute quicker if we concentrate all our inputs and outputs in one box.

4.3.4 Control input module

The buttons input box has nearly all of the above complications. We have the internal state but we actually store the master state for the whole program in this box since it is affected most strongly by the button inputs. Thus we have no other controlling inputs to this box. We do, however, have to send the current mode to almost all other parts of the program so there are additional outputs for this as well as the commands generated by the inputs:

```
{- Control input module -}
box buttons
   in (b1str::(), b2str::(), b3str::(), mode::mode)
  out (cc::control_command, output_mode::mode, mode'::mode)
```

4.3.5 Control output module

The box which generates the pattern for the LEDs gathers information from throughout the program and generates the status information from it. We thus have multiple inputs and only one output apart from the state loopback:

```
{- Control output module -}
box output
   in (mode::mode, state::control_state, outlevel::byte, leds::byte)
  out (p2::byte, leds'::byte)
```

4.3.6 IO box design

We thus have the I/O box headers. Note that we have templated the filter, filter input and filter output boxes. The reason for this is that we could expand the design at a later stage by adding another channel from AD1 to DA1 with an independent filter box. The overall design at this stage is depicted in Figure 4.

4.4 Step (4): Timing constraints for inputs/outputs

As discussed above, the only input with any hard real-time deadlines is the timer input. We know in advance that we can't possibly meet the stringent timing requirements for audio work but we can verify this with our analysis and even guarantee the sample rate up to an upper bound provided by our analysis. For this reason, we attach a timing constraint to the output of the filter box of 20μs.

We would like to apply the 20μs deadline to the *sum* of the filter input, filter and filter output modules (shown as thick arrows in the block diagram in Figure 1). Hume does not at present have the ability to express timing constraints which span different boxes and, in any case, we should really take the overall superstep time as our unit of time. Assuming our current execution semantics and placing the filter loop across three boxes means we could, at worst, have three supersteps in the filter loop. If, however, we assume that the filter's operation is not

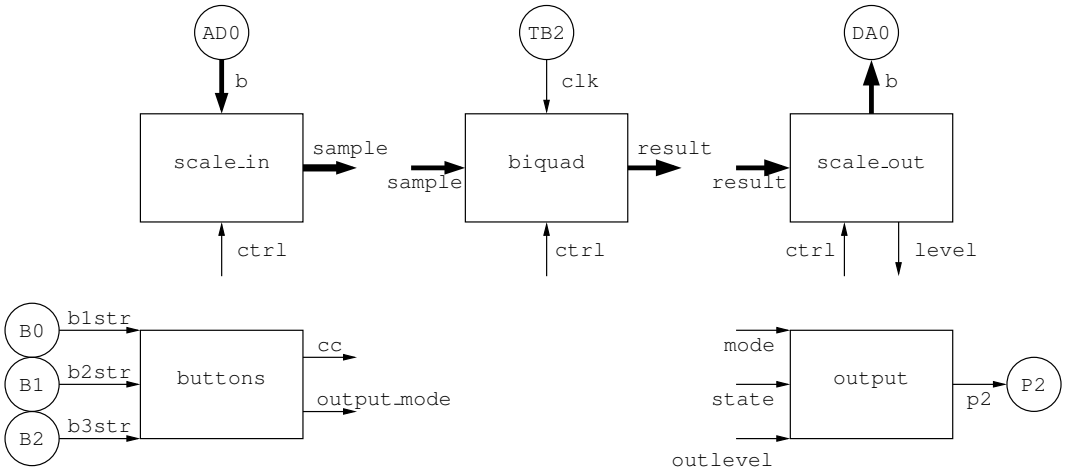

Figure 4: Biquad filter I/O box design

affected by control inputs[2], we can use the sum of the execution times for these three core boxes as a reasonable bound on the sample rate.

4.5 Step (5): Overall box structure outline and I/O box links

We are now required to link all the I/O boxes together and define the remaining internal boxes. As a first guess, we would expect the `scale_in`, `biquad` and `scale_out` boxes to be closely coupled in a direct pipeline. We would also expect there to be a central routing box connecting the button input, LEDs output and filter boxes with some kind of control logic. Figure 4 illustrates these points. The top three boxes are obviously going to be closely coupled but the `buttons` and `output` boxes have a lot of "loose" wires which need to be routed carefully through the design so we would expect a *routing* box of some sort to be involved.

There is one outstanding problem, however. We need to place the code to compute the filter parameters from the desired physical characteristics of the filter, as explained in Section 3.1. Here we have a choice as to whether to implement this as a function or functions applied to the outputs from the control modules or to implement a separate box to encapsulate this functionality. Both methods have their merits, here, but we would probably want a separate estimate of the computation time for the filter parameters and that is more easily separated from surrounding factors if we have a box for this purpose. Initially, therefore, we assume a box between the control module and the filter module which translates commands such as filter switching into filter parameters. This box is likely to be one of the most complex in the design and will probably need further refining.

We define two new boxes called `control` and `compute_filter` as discussed above and wire together the IO boxes using them. This design is presented in Figure 5, the `control` box has the following header:

```
{- Control module -}
box control
   in (command::control_command, state::control_state)
  out (scale_in_ctrl::scale_control, bq_ctrl::filter_def,
       scale_out_ctrl::scale_control, output_state::control_state,
       state'::control_state)
```

Again, we have the usual loopback wire and we read commands in from the control module. For output

[2]In fact, when the filter changes we will get a variety of digital noise out of the filter so we don't mind if the filter loop overruns during this operation.

we send the three control signals to the `scale_in`, `biquad` and `scale_out` boxes. Despite it's wiring complexity, this box will be computationally very simple, acting mostly as a router for command and control data to and from the other modules. Note, however, that we take much of the state from the buttons box into this new box. The buttons box still holds the overall mode of operation for the filter but we now store the control parameters for the filter (filter type, filter definition and input/output scaling parameters) here.

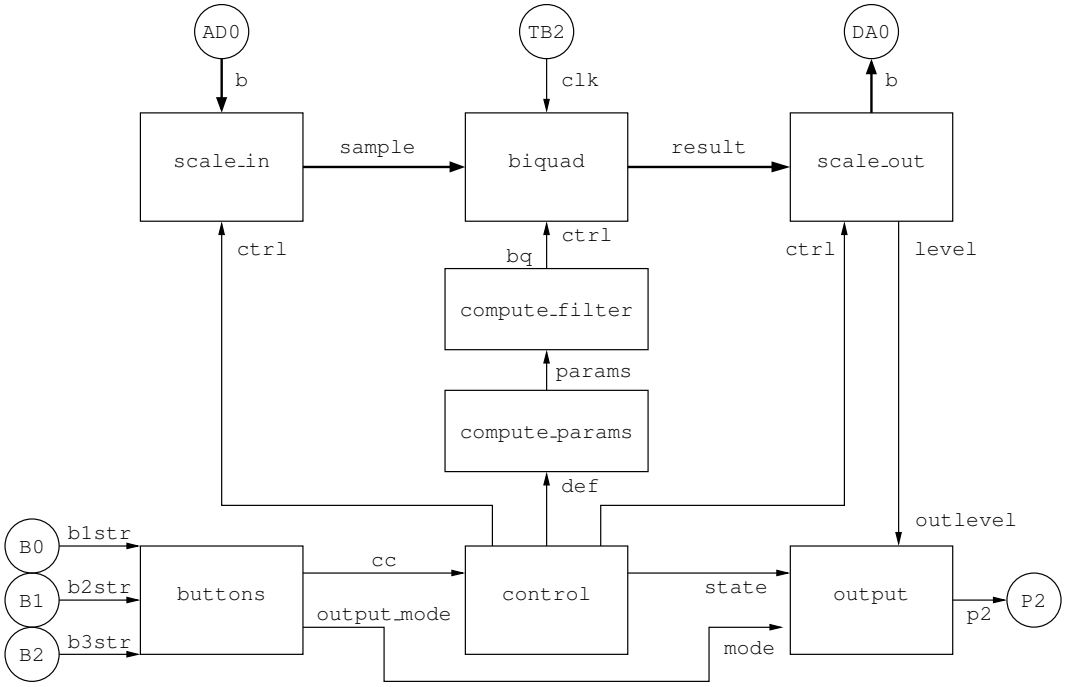

Figure 5: Biquad filter overall box design

4.6 Step (6): Refine box structure

The biquad filter application is quite simple and our initial attempt at a design has been arrived at by sound reasoning. However, during implementation of the `compute_filter` box it was found that all of the filter parameter computations used the same set of common intermediate values. To allow sharing of these values between the different filter types we could again either implement this using a common function or we could use a box to append these values to the filter definition sent by the `control` box. Again, we elect to add a box for this purpose so we can assess the resources used by this common code section in isolation. We thus split the `compute_filter` box into two boxes, `compute_params` and `compute_filter`, with the following box headers:

```
{- Filter parameter module -}

box compute_params
   in (def::filter_def)
  out (params::filter_params)

box compute_filter
   in (params::filter_params)
  out (bq::biquad_t)
```

4.7 Step (7): Define types and datatypes

At this point we have the overall structure of the program. We now begin to fill in some of the details and start by defining the abstract types which we have used to write the box headers. In fact, we do not need many types for this application since they are uniform throughout the code and we omit the types for the loopback wire holding the boxes' states.

Firstly, we concretize the basic types, we need an 8-bit unsigned integer type for the A/D and D/A raw data. We also decide upon the type of the filter computation which will be floating point. Fixed point arithmetic would give a much faster computation here since we can control the numerical range of the inputs and outputs but we do not as yet have this datatype in the Hume compiler implementation.

Next we need a datatype to act as the filter state. This houses the filter parameters plus the historical data needed by the filter equation. Normally, this would be internal to the filter box but we implement control of the filter by writing directly to the state. Consequently, this type becomes the control parameter for the filter box and is needed by other modules.

We also need to define a filter definition in terms of the physical parameters of the filter for the parameter computation and control modules to work on. This comprises the four parameters of; gain, centre frequency, sampling rate and bandwidth plus an indication of the filter type for which we define an auxiliary datatype.

Finally, we need a definition for the input/output scaling which is just a gain/offset combination and a set of control commands to allow the input module to communicate with the filter module.

The complete types module is presented in Figure 6.

4.8 Step (8): Specify box functionality

We now have our black-box characterisation of the system. We have the IO characteristics, the box skeleton and the types of the wires connecting the boxes. We proceed to specify the functionality of each box.

4.8.1 Box `scale_in`

- Initialise with default scaling parameters.
- Read the A/D input from AD0, convert into floating point value and apply linear scaling and offset, pass result on to `biquad` box input.
- Read in control value (scale and offset parameters) and update internal state with new parameters.

4.8.2 Box `scale_out`

- Initialise with default scaling parameters.
- Read result from `biquad` box output, apply scaling and offset. If the resulting value is greater than 255, adjust scaling, if less than zero, adjust offset, retain new scaling and offset. Convert to 8-bit integer and write to DA0 port.
- Read in control value (scale and offset parameters) and update internal state with new parameters.
- Send current output level to the `output` box.

4.8.3 Box `biquad`

- Initialise with default filter parameters.
- Read sample from `scale_in` box, apply Equation 1, write result to `scale_out` box.
- Read in new filter parameters and store in internal state.

```
{- Types module -}

{- Fundamental types -}
type byte = word 8;
type smp_type = float 64;

{- This holds the data required to update samples through a filter -}
--                a0,       a1,       a2,       a3,       a4;
type biquad_t = (smp_type, smp_type, smp_type, smp_type, smp_type,
--                x1,       x2,       y1,       y2;
                  smp_type, smp_type, smp_type, smp_type);

{- filter types -}
data filter_type =
    NULLF {- null filter -}
  | LPF   {- low pass filter -}
  | HPF   {- High pass filter -}
  | BPF;  {- band pass filter -}

{- Filter definition -}
type filter_def = (filter_type, {- filter type -}
                   smp_type,    {- gain of filter or _A=10**(dbGain/40.0) -}
                   smp_type,    {- center frequency -}
                   smp_type,    {- sampling rate -}
                   smp_type);   {- bandwidth in octaves -}

macro init_filter = (NULLF, 1.05925, 278.4375, 10000.0, 1.0); -- _A=10**(dbGain/40.0)

{- Input/output scaling parameters -}

--                     gain,     offset
type scale_control = (smp_type, smp_type);

-- Initial gain, actually convert [0x00,0xff] to [0.0,1.0)
macro initial_gain = ((1.0/256.0),0.0);

{- Control commands -}

data control_command =
   NextFilter              {- Switch to next filter -}
 | AdjustOutput (smp_type) {- Increase input/output scaling -}
 | AdjustInput (smp_type); {- Decrease input/output scaling -}
```

Figure 6: Biquad filter types module

4.8.4 Box `buttons`

- Receive interrupts from buttons B1, B2 and B3 and convert into control commands for the `control` box. Implement the mode logic defined in Table 2. Since parameter adjustment is relative we only need to store the current mode as the state of this box.

4.8.5 Box `control`

- Initialise with default input and output scaling parameter. Initialise with default filter parameters. Note that these initial parameters should match those used to initialise the filter chain so we compute the default

values at a common point in the program.

- Read in filter and input/output scaling commands from the `buttons` box. Convert into commands to be sent to the `scale_in`, `biquad` and `scale_out` boxes.
- Send current parameters to the `output` box.

4.8.6 Box `output`

- Create the default output pattern from knowledge of the default parameters for the system.
- Read in current output level from the `scale_out` box and convert into bitwise pattern for the relevant bits of the LED port.
- Read in the current filter parameters from the `control` box and set a pattern representing the filter type.
- Read in the current overall mode from the `buttons` box and set an output pattern to represent the mode.

4.8.7 Box `compute_params`

- Read filter definition from `control` box and append the common parameters for filter parameter computation, write to `compute_filter` box.

4.8.8 Box `compute_filter`

- Read filter definition and common parameters from `compute_params` box, calculate filter parameters according to filter type and physical parameters and write new filter parameters to the `biquad` box.

4.9 Step (9): Design refinement

At this point we have the specification of the individual boxes on top of the box framework and the wiring with the types on all wires concrete. Our methodology suggests a refinement step here to allow this phase of the design to be adjusted to meet the functional specification. This step, however, is more useful when we are looping back from the validation phase after the application of costing and we are looking to improve our design to meet both functional requirements and resource limits at the same time. We skip this step on the first loop round on a simple application such as a digital filter.

4.10 Step (10): Implement top-level functionality for boxes

Given our functional specification and the necessary wiring which defines the types it is now relatively straightforward to implement the interiors of the boxes, ignoring any external function calls which can be written later. For the most part, the box input patterns follow the items listed above in the functional specification section.

We also present the wiring statements here because the `initially` statements are effectively part of the box initialisation code.

4.10.1 Box `scale_in`

Here we have two simple patterns, one to read in configuration data which we just write directly into the box state and the other to read the AD0 input and do the input scaling.

```
template scale_in
  in (b::byte, ctrl::scale_control, ss::scale_control)
 out (sample::smp_type, ss'::scale_control)
match
   (*,ctrl,_) -> (*,ctrl)
 | (b,*,(gain,offset)) -> (((((b::byte) as smp_type) * gain) + offset),(gain,offset));

instantiate scale_in as scale_in_ad0;

wire scale_in_ad0 (ad00, control.scale_in_ctrl, scale_in_ad0.ss' initially initial_gain)
                  (biquad0.sample, scale_in_ad0.ss);
```

4.10.2 Box `scale_out`

Again we just have the two patterns. The complication here is the automatic adjustment of the output scaling which is small enough to be encapsulated in the box expression. If it became larger or more complex we could put it out into a separate function.

```
template scale_out
  in (result::smp_type, ctrl::scale_control, ss::scale_control)
 out (b::byte, level::byte, ss'::scale_control)
match
   (*,ctrl,_) -> (*,*,ctrl)
 | (result,*,(gain,offset)) ->
     let b0 = ((result::smp_type) - offset) / gain in
     let gain' = if b0 > 255.0 then (result - offset) / 256.0 else gain in
     let offset' = if b0 < 0.0 then result else offset in
     let b = ((((result::smp_type) - offset') / gain') as byte) in
(b,b,(gain',offset'));

instantiate scale_out as scale_out_da0;

wire scale_out_da0 (biquad0.result, control.scale_out_ctrl, scale_out_da0.ss' initially initial_gain)
                   (da0, output.outlevel, scale_out_da0.ss);
```

4.10.3 Box `biquad`

As mentioned in Section 3.2.1.9, this module is a direct implementation of the filter difference equation. We merely have to add the control pattern and define the wiring for this box:

```
template biquad
   in (clk::(), sample::smp_type, ctrl::biquad_t, bq::biquad_t)
  out (result::smp_type, bq'::biquad_t)
match
   (*,*,ctrl,_*) -> (*,ctrl)
 | ((),sample,*,(a0,a1,a2,a3,a4,x1,x2,y1,y2)) ->
     let result = (a0 * sample) + (a1 * x1) + (a2 * x2) - (a3 * y1) - (a4 * y2) in
       (result,(a0,a1,a2,a3,a4,sample,x1,result,y1));

instantiate biquad as biquad0;

wire biquad0 (tb2, scale_in_ad0.sample, compute_filter.bq, biquad0.bq')
             (scale_out_da0.result, biquad0.bq);
```

4.10.4 Box `buttons`

Again, the specification in Table 2 can be implemented directly in Hume. We only need to add catchall patterns for unrecognized combinations of input patterns. We do, however, include the implementation of the loopback type here because it is the global operation mode mentioned earlier:

```
data mode = RunMode | SetFilterMode | SetInputMode | SetOutputMode;

init_mode = RunMode;

box buttons
   in (b1str::(), b2str::(), b3str::(), mode::mode)
  out (cc::control_command, output_mode::mode, mode'::mode)
match
   ((),*,*,RunMode)       -> (*,                  SetFilterMode, SetFilterMode)
 | ((),*,*,SetFilterMode) -> (*,                  SetInputMode,  SetInputMode)
 | ((),*,*,SetInputMode)  -> (*,                  SetOutputMode, SetOutputMode)
 | ((),*,*,SetOutputMode) -> (*,                  RunMode,       RunMode)
 | (*,(),*,SetFilterMode) -> (NextFilter,         *,             SetFilterMode)
 | (*,(),*,SetOutputMode) -> ((AdjustOutput 1.1), *,             SetOutputMode)
 | (*,*,(),SetOutputMode) -> ((AdjustOutput 0.9), *,             SetOutputMode)
 | (*,(),*,SetInputMode)  -> ((AdjustInput 1.1),  *,             SetInputMode)
 | (*,*,(),SetInputMode)  -> ((AdjustInput 0.9),  *,             SetInputMode)
 | (*,_,*,mode) -> (*,*,mode)
 | (*,*,_,mode) -> (*,*,mode);

wire buttons (b1, b2, b3, buttons.mode' initially init_mode)
             (control.command, output.mode, buttons.mode);
```

4.10.5 Box `control`

In this box we translate commands from the control input module into filter and input/output scaling parameters. The conversion is complex enough and there is enough commonality between input scaling and output scaling that we might want to use auxiliary functions to encapsulate this functionality. In fact, this makes the box itself quite simple:

```
box control
   in (command::control_command, state::control_state)
  out (scale_in_ctrl::scale_control, bq_ctrl::filter_def, scale_out_ctrl::scale_control,
       output_state::control_state, state'::control_state)
match
   (NextFilter,((ft,_A,freq,srate,_bandwidth),ipsc,opsc)) ->
     let filt = ((next_filter ft),_A,freq,srate,_bandwidth) in
       (*,filt,*,(filt,ipsc,opsc),(filt,ipsc,opsc))
 | (AdjustOutput amt,(filt,ipsc,opsc)) ->
     let sc = adjust amt opsc in
       (*,*,sc,(filt,ipsc,sc),(filt,ipsc,sc))
 | (AdjustInput amt,(filt,ipsc,opsc)) ->
     let sc = adjust amt ipsc in
       (sc,*,*,(filt,sc,opsc),(filt,sc,opsc))
 | (_,state) ->
     (*,*,*,*,state);

wire control (buttons.cc, control.state' initially init_control_state)
             (scale_in_ad0.ctrl, compute_params.def, scale_out_da0.ctrl, output.state, control.state);
```

4.10.6 Box `output`

The `output` box is similar to the `control` box in that it is a box which converts multiple data streams. Again, we keep the conversion code separate from the routing function and the box becomes a simple router which calls the appropriate functions on its data streams:

```
box output
   in (mode::mode, state::control_state, outlevel::byte, leds::byte)
  out (p2::byte, leds'::byte)
match
   (mode,*,*,leds) ->
     let leds' = (leds ^& 0xfc) ^| (mode2leds mode) in
       (leds',leds')
 | (*,state,*,leds) ->
     let leds' = (leds ^& 0xf3) ^| (state2leds state) in
       (leds',leds')
 | (*,*,outlevel,leds) ->
     let leds' = (leds ^& 0x0f) ^| (outlevel ^& 0xf0) in
       (leds',leds');

wire output (buttons.output_mode initially init_mode,
             control.output_state initially init_control_state,
             scale_out_da0.level initially 0x00,
             output.leds' initially 0x00)
            (p2, output.leds);
```

4.10.7 Box `compute_params`

This is a simple one-input, one-output box. Where there is no complexity in the input patterns the box becomes syntactically very similar to a function so, despite the complexity of the calculations here, we perform these computations in the box expression. One point to note, however, is that we actually initialise the filter from here. We place the default filter definition on the initial inputs to this box and it gets passed along to the filter box as a normal filter-change command. This allows us to specify the initial filter using unconverted (ie. more human-readable) filter definitions:

```
box compute_params
   in (def::filter_def)
  out (params::filter_params)
match
   (ftype,_A,freq,srate,_bandwidth) ->
    let omega = 2.0*(M_PI*(freq/srate)) in
    let sn = sin(omega) in
    let cs = cos(omega) in
    let alpha =
      if sn == 0.0
      then 0.0
      else sn*(sinh((M_LN2/2.0)*(_bandwidth*(omega/sn)))) in
      (ftype,_A,cs,alpha);

wire compute_params (control.bq_ctrl initially init_filter) (compute_filter.params);
```

4.10.8 Box `compute_filter`

Again this box has simple connectivity but complex internals and again, we incorporate the code into the box apart from a common section which was required to post-process the values from individual filter types (`post_compute`). The equations implemented here are simply the recipes for the different filters mentioned in Section 3.1:

```
box compute_filter
   in (params::filter_params)
  out (bq::biquad_t)
match
   (NULLF,_,_,_) ->
     post_compute (1.0,0.0,0.0,1.0,0.0,0.0)
 | (LPF,_A,cs,alpha) ->
     let _b0 = (1.0 - cs) / 2.0 in
     let _b1 = 1.0 - cs in
     let _b2 = (1.0 - cs) / 2.0 in
     let a0 = 1.0 + alpha in
     let a1 = (-2.0) * cs in
     let a2 = 1.0 - alpha in
       post_compute (_b0,_b1,_b2,a0,a1,a2)
 | (HPF,_A,cs,alpha) ->
     let _b0 = (1.0 + cs) / 2.0 in
     let _b1 = 0.0 - (1.0 + cs) in
     let _b2 = (1.0 + cs) / 2.0 in
     let a0 = 1.0 + alpha in
     let a1 = (-2.0) * cs in
     let a2 = 1.0 - alpha in
       post_compute (_b0,_b1,_b2,a0,a1,a2)
 | (BPF,_A,cs,alpha) ->
     let _b0 = (1.0 + cs) / 2.0 in
     let _b1 = 0.0 - (1.0 + cs) in
     let _b2 = (1.0 + cs) / 2.0 in
     let a0 = 1.0 + alpha in
     let a1 = (-2.0) * cs in
     let a2 = 1.0 - alpha in
       post_compute (_b0,_b1,_b2,a0,a1,a2);

wire compute_filter (compute_params.params) (biquad0.ctrl);
```

4.11 Step (11): Box refinement and function implementation

When implementing the functions required to support the box implementations we may decide to turn an existing box implementation into a function or, conversely, implement a function as a supplementary box. In fact, during the first pass round the development loop the `compute_params` box arose from functional support for the `compute_filter` box. There is little scope in a small application of this type for such refinement, however, and our design is actually consistent with the specification after a single pass.

For a filter implementation of this kind we would expect the Hume to be at a low level in the Hume level hierarchy. In fact, none of the code presented so far rises above FSM-Hume and we would like to retain this level since it is in keeping with the application and is likely to lead to more accurate and reliable costings.

So, when implementing support functions, we choose to eliminate features such as recursion since that would move the program up into PR-Hume. A good example of this is the bit manipulations in the `output` box. We could simplify the code and make it more generic, readable and flexible, if we recursed over the bit numbers to

generate the output byte pattern. However, we in fact use more verbose pattern-matches specific to each case thus keeping the FSM-Hume level. An example is the `byte2leds` function which we use in the debug version of the code to turn the LED pattern into a text string:

```
byte2leds :: byte -> vector 8 of int 8;
byte2leds b =
  << if (b ^& 0x80) == 0x80 then 1 else 0,
     if (b ^& 0x40) == 0x40 then 1 else 0,
     if (b ^& 0x20) == 0x20 then 1 else 0,
     if (b ^& 0x10) == 0x10 then 1 else 0,
     if (b ^& 0x08) == 0x08 then 1 else 0,
     if (b ^& 0x04) == 0x04 then 1 else 0,
     if (b ^& 0x02) == 0x02 then 1 else 0,
     if (b ^& 0x01) == 0x01 then 1 else 0 >>;
```

Here are the support functions at this stage of the development:

4.11.1 Box `control`

Here, we have the `next_filter` function which would have been unwieldy if incorporated in the `control` box and the `adjust` function which is common between input and output scaling:

```
next_filter :: filter_type -> filter_type;
next_filter ft =
  case ft of
    NULLF -> LPF
  | LPF -> HPF
  | HPF -> BPF
  | BPF -> NULLF;

adjust :: smp_type -> scale_control -> scale_control;
adjust amt (gain,offset) = (gain * amt,offset);
```

4.11.2 Box `output`

Again, `case` patterns in boxes can lead to sytactic confusion so we lift these conversions out into separate functions:

```
mode2leds :: mode -> byte;
mode2leds m =
  case m of
    RunMode       -> 0x03
  | SetFilterMode -> 0x02
  | SetInputMode  -> 0x01
  | SetOutputMode -> 0x00;

state2leds :: control_state -> byte;
state2leds state =
  case state of
    ((NULLF,  _,_,_,_),_,_) -> 0x00
  | ((LPF,    _,_,_,_),_,_) -> 0x04
  | ((HPF,    _,_,_,_),_,_) -> 0x08
  | ((BPF,    _,_,_,_),_,_) -> 0x0c;
```

4.11.3 Box `compute_filter`

Again we have common code between individual match patterns in a box so we lift it out into a function:

```
post_compute :: (smp_type,smp_type,smp_type,smp_type,smp_type,smp_type) -> biquad_t;
post_compute (_b0,_b1,_b2,a0,a1,a2) = (_b0/a0,_b1/a0,_b2/a0,a1/a0,a2/a0,0.0,0.0,0.0,0.0);
```

4.12 Step (12): Apply costing

Our analysis tools are now sufficiently advanced to allow relatively quick application of costing to Hume code, as integral part of our costing-by-construction methodology. Since our program is in FSM-Hume our analysis can easily infer linear bounds on the resource consumption.

4.12.1 Costing results

Applying the analysis to the code designed so far gives the costs presented in Table 3. For comparison, we give the actual measured values in Table 4 although the main objective here is not to verify the analysis results but to indicate the interpretation of the analysis results in the context of our methodology.

At this point we have our *white-box* characterisation of the system and we have also managed to derive bounds for memory and execution time for this design. We now proceed to the validation phase of our methodology.

4.13 Step (13): Refine the functionality to respect costs

4.13.1 Interpretation of the analysis results

The bounds produced by our analysis [JLH07a, JLH07b] are encoded as numerical annotations to the input and output types of boxes and functions. This has the concrete advantage that our bounds are not only size-

Box	Heap	Stack	Time
biquad0	33	32	10330
compute_filter	73	62	26392
compute_params	40	38	47307
scale_in_ad0	14	15	3919
scale_out_da0	33	33	16044

Table 3: Analysis costs for biquad filter application

Box	Heap	Stack	Time
biquad0	33	32	5848
compute_filter	73	59	13176
compute_params	40	34	16107
scale_in_ad0	10	15	1844
scale_out_da0	33	33	5920

Table 4: Measured costs for biquad filter application

dependent, as would be expected, but more generally data-dependent. As an example, the `compute_filter` box, as discussed in Section 4.10.8, takes a 4-tuple as input. The first element of the tuple is an enumeration type `filter_type`, with elements `NULLF`, `LPF`, `HPF`, `BPF`, specifying the kind of filter that shall be applied. Our analysis attributes specific costs to each of these four constructors, expressing the fact that some costs are only incurred when the corresponding constructor arrives as input. Compared to a model that encodes a cost bound as a closed formula over the input size, our approach results in more accurate upper bounds. The practical disadvantage of our approach is the difficulty for the inexperienced user in interpreting the cost bounds, which are presented as annotated types. To alleviate this problem, the latest version of the analysis provides an elaboration module (discussed in Deliverable D43 WP7 [JH09]), which provides a textual interpretation of the cost bounds for commonly occuring cases. For a worst-case execution time analysis of `computer_filter` the elaboration module produces the following explanation, with costs measured in clock cycles:

```
Worst-case Time-units required to compute box compute_filter once
                                                    in relation to its input:
  359 + 9374*X1 + 16659*X2 + 16123*X3 + 14570*X4
    where
        X1 = one if 1. wire is live, zero if the wire is void
        X2 = number of "BPF" nodes at 1. position
        X3 = number of "HPF" nodes at 1. position
        X4 = number of "LPF" nodes at 1. position
```

In particular, since `BPF`, `HPF` and `LPF` are elements of an enumeration type, selecting a band pass, high pass or low pass filter, respectively, we know that only one of the three costs attached to these constructors (16659, 16123 or 14570) will apply. Furthermore, in the case where a null filter is selected, by providing `NULLF` as input, none of these three costs applies and therefore the time bound for this case is 9733 clock cycles. Being data-dependent, this parameterised bound is more accurate than the worst-case bound specified in Table 3, where we take the worst-case over all constructors to derive the simplistic absolute value of 26392 clock cycles.

Here we analyse the biquad filter application assuming the M32C platform and the simple time constraints we have derived from the system specification presented in Section 3.2.1 above. We outline by example the techniques which should be employed in arriving at a verified design although, in our case we realize that the

design must fail to meet the stringent timing constraints we have imposed upon it. This allows us to illustrate more points during the validation process than for a design which succeeds at the first attempt.

4.13.1.1 Memory costs The current Hume semantics is such that the box memory is reset after each invocation so for our overall upper memory bound for box execution for the application we need the box with the greatest sum of heap plus stack memory which in this case is the `compute_filter` box.

Notice that none of the memory costs are anywhere large enough to cause us problems within the 32k of RAM on the board. Each heap cell is four bytes for this architecture and the maximum memory usage for any box is 73 $heap$ + 62 $stack$ cells for the `compute_filter` box which makes 540 bytes maximum.

We note that the previously developed, simple cost analysis infers 77 $heap$+61 $stack$ cells for this box. This is less accurate than the amortised-cost-based analysis, developed in this project. Thus, our new analysis opens up the possibility for allocating heap and stack space with tighter bounds, thus reducing memory consumption without losing resource-safety. Future versions of the Hume development tools will have better integration between the analysis and compiler but already now, we can use the analysis result values by forcing the heap and stack space to pregiven values using arguments to the `humec` compiler. Actual measurement on the board validates the numbers provided by the analysis.

Based on these bounds of dynamic space consumption we can calculate the total space consumption by adding the statically determined space. This includes the heap storage for wires which we can compute statically from the types of the data on the wires. For the biquad filter application, we compute a total of 436 $heap$ cells which is 1744 bytes of heap memory.

We also have to include the host C compiler's stack usage and we can get an upper bound for this by simply measuring the maximum stack usage at the most deeply nested function called[3]. This has already been validated at less than 1k bytes of RAM and will not vary much between applications. If external functions were called, this memory usage could be significant but we are presently unable to provide manual input of bounds for such code so we do not consider using such functions in our current analyses.

Finally, we have allowed $0x300 = 768$ bytes of data for the interrupt stack although this is a very generous allowance and only a tiny fraction of it is ever used.

Summing these four memory costs we get $540 + 1744 + 1024 + 768 = 4076$ bytes of RAM which is well within our design maximum of 32k.

4.13.1.2 Execution costs As mentioned previously, for this application it is not the overall maximum execution time which counts so much as the loop time round the filter chain. This partial analysis is not part of our current Hume toolset but with a small amount of manual computation we can arrive at the bounds we need.

If we sum the time bounds for the `scale_in`, `biquad0` and `scale_out` boxes we get $3919 + 10330 + 16044 = 30293$ clock cycles, or 947μs. Note that there is a small amount of computational overhead to be added to this partial analysis, principally the time required to fail all of the other boxes in the program. This overhead is minimal, however (just checking the status flags for each box), and could be taken into account in future by a partial analysis routine. This gives us a sample rate of about 1.056kHz, obviously well short of audio sampling rates but this is a concrete guarantee for the application given the above *proviso*. The measurements in Table 4 confirm these bounds, produced by the analysis. The reason for the fairly high differences between time measurements and analysis bounds are the operations on floating-point numbers, which dominate the execution time in these boxes. Our cost table uses WCET bounds on all these operations, as obtained from the aiT tool. Since the floating point operations are implemented in software on this platform, the bounds for the primitive operations are already fairly loose.

From the point of view of the application there could be a more serious limitation to this analysis in that implementing the filter chain as separate boxes can (but not necessarily does) introduce delays into the filter

[3]None of the runtime system's functions allocate more than a few tens of bytes of data on the stack so the maximum stack usage occurs approximately at the most deeply nested function.

computation. This could be a problem in some applications but for our simple filter application we can just ignore this feedback between the target language semantics and the theoretical design of the filter dynamics.

4.13.2 Refine program according to cost analysis

4.13.2.1 Memory costs For memory, we are obviously well within our design limits. However, if our analysis had indicated that our 32k RAM limit could be violated, there are several possibilities for reducing memory usage:

- We could rewrite functions to be more memory efficient but with the same functionality (Step (**11**)). For example, we could switch from lists to vectors as a basic datatype which are more compact but may require rewriting to achieve the same functionality. They may also alter the computational complexity of the algorithms used and thus affect the computation time.
- We could revisit the box functionality (Step (**8**)). For example, we could split a large, complex pattern into two smaller patterns which might save a small amount of memory. This requires rewiring the box and possibly altering the box expressions to match. We might also review our decision as to whether a box should be implemented as a function or not. There is little direct saving to be made converting a box into a function but by extracting functions from multiple boxes and refining the related computations there can be more significant savings to be made by spreading the costs more evenly over the boxes.
- We could revise the top-level box structure (Step (**6**)) or the overall box structure (Step (**5**)). For example, we could split a box with high memory usage into two smaller boxes such that the maximum usage for the two boxes is less than for the single box. The savings here will not be great but again, this step could be combined with a review of the computations involved leading to a greater overall saving. We illustrated this operation when we split the `compute_filter` box into two parts. We have to be careful here, because we actually increased the amount of wire memory used in this case by having to add the intermediate values for the computation to the wires.

4.13.2.2 Execution costs For execution time costs, we have to admit that we cannot match the imposed bound, given the current hardware and software configuration. We attempt to reduce the execution costs following similar reasoning as for memory:

- In fact, we can make an immediate saving without any rewriting or restructuring of the program if we consider the `scale_out_da0` box execution time to be a large overestimate because of the automatic scaling adjust. The scaling adjust is only triggered during the initial few box invocations after a filter changes and is then no longer called once the scaling parameters settle down for normal operation. We could account for this behaviour in our analysis by using an algebraic data-type for the incoming control parameters, separating the cases where scaling adjust has to be performed, from the case where it need not be performed. In the latter case, a lower analysis bound can be attached to the constructor representing this case.
- The functionality of these boxes is so simple that there are no savings to be made by either revising the computations (Step (**11**)) or changing the box functionality (Step (**8**)).
- However, going back to the top-level box structure (Step (**6**)) we can eliminate the `scale_in_ad0` and `scale_out_da0` boxes and incorporate the scaling/descaling functions into the `biquad0` box. However, this reduces the overall bound only slightly.

This is about the best we can achieve by simply rewriting the program within the given specification.

4.14 Step (14): Cost failure

At this point we have a failed design and furthermore have been unable to bring the implementation within the specified bounds by simple restructuring and algorithmic considerations. At this point:

- We cannot relax the margins on our constraints without seriously degrading the specification. We could use this design for another purpose, for example as a low-speed data filter but this is technically outside the scope of what we are trying to achieve with this design.
- We can, however, revise the application specification. Removing scaling functions from the filter chain altogether gives an upper bound of 10330 clock cycles, or 323μs, for the `biquad0` box, with a sample rate of about 3.1kHz. This represents the fastest loop time we can achieve with our target platform for this application. In fact, at this point we have a simple "bare-bones" implementation of the filter with as tight a read-compute-write loop as we can possibly have.
- Thus we conclude that we need to revise our hardware requirements, possibly moving to special-purpose hardware such as the Texas Instruments TMS320 range of DSP chips.

4.15 Step (15): Design failure

This step only applies if there is a serious flaw in the system specification such that no available hardware can possibly meet its requirements.

5 Summary

We have presented in detail the development of a simple small-scale embedded application on a microcontroller board using Hume and the Hume programming methodology. Our principal objective with this experiment was to illustrate how the cost analysis could be used to guide the development process.

With such a small application it was difficult to provide enough scope to investigate all the possible scenarios which could arise in the application of our methodology or to provide comprehensive guidelines for incorporating our cost analysis into the design and implementation process. We have, however, given a case study of what our tools can provide and we can summarize this experiment as follows:

- We showed how a loose specification can be written in such a way as to incorporate timing and execution bounds at an early stage of the specification process. Although we only showed an overall memory limit plus an (unreasonable) timing constraint, it is easy to see how more complex sets of resource constraints could be incorporated into the design. For this simple application there was little interaction between the separate constraints but for much larger systems, keeping track of large numbers of complex, interrelated constraints would be difficult and error-prone without our ability to formally represent them in the specification.
- We illustrated in detail how the specification could be turned into a working implementation in Hume. There is nothing particularly advanced about the way we structure and control the implementation process, it proceeds in logical stages familiar to any developer used to structured methodologies. However, we have included in the methodology various "hooks" in the implementation process where we can attach cost considerations and marked various points in the process to where we can return, should our cost constraints be exceeded. Again, our simple application only gives an idea of what could be achieved with this mechanism but we showed the reasoning processes which apply before triggering a loop back into an earlier phase or step of the design process.

- Once we achieved an implementation of the design we applied our cost analysis and proceeded to verify the resource bounds. At this point we do not even have to implement the program on the target platform. Our analysis tools have all the relevant characteristics of the target architecture built into them. Furthermore, our analysis runs in a few seconds giving the system implementer instant feedback on the state of the application without recourse to the time-consuming instrumentation, compilation, loading and running of the application simply to measure memory usage.

- Once we isolated the specific point in our design where the resource bounds were broken we then tackled a revision of the design in order to bring the implementation into alignment with the specification. Working backwards from the point of failure to progressively earlier points in the implementation process we modified the implementation to reduce the resource usage in the program. Essentially, we failed to do this for our filter implementation but at each stage of this process we knew from the analysis results exactly how much more rework was required and also the point at which we have a failed design.

- During this process we have to be careful that we do not break good resource bounds by fixing bad ones. Our simple filter application did not show this but, for example, it is very often possible to exchange computation for memory. An example of this would be using a lookup table with precomputed values rather than computing the values each time. One method uses more memory but executes faster, the other uses less memory but requires greater computation. When we loop back and review the design with a view to reducing resource usage we need to keep an eye on all related constraints at the same time. Since our analysis gives us a rapid update on resource constraints during the revision of the implementation it is much easier to make compromises of this type.

- Finally, once we arrive at a verified design we have formal assurances that our resource bounds are indeed upper bounds and cannot be broken by some future unforeseen circumstance in the application's operating environment. This combination of *rapid development plus concrete guarantees* on the verification process makes our technique very powerful for developing specific kinds of embedded applications. Those, for example, which have tight budgetary constraints where the ability to use smaller safety margins on memory could allow significant cost savings. Similarly, the development of highly safety-critical code may require formal assurances on execution time, for example in scheduling problems. Again, the speed with which we can give these formal bounds can play a significant role in reducing the overall development costs of such systems.

6 Conclusions

With some reservations, we have demonstrated many of the points in Section 5 in our small filter application. We showed how to incorporate resource bounds into a simple design and how to turn the specification into an implementation. Our cost analysis then gave us the ability to monitor the revision of the implementation and the design by providing instant information on the resource bounds. Finally, the cost analysis gave us incontestable proof of the failure of the design. Obviously, where we go from there is very much dependent upon the circumstances of the application and its intended users but we strongly feel that incorporating our resource analysis into all stages of the design process is a sound way to proceed in the development of systems which are sensitive to resource bounds issues.

References

[JH09] S. Jost and K. Hammond. Analysis Robustification. EmBounded Project Deliverable, March 2009. Deliverable D43.

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11.

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14.

[KLT06] Sen M. Kuo, Bob H. Lee, and Wenshun Tian. *Real-Time Digital Signal Processing: Implementations and Applications*. Wiley, 2nd edition, April 2006. ISBN: 978-0-470-01495-0.

[SLMS08] N. Scaife, H-W. Loidl, G. Michaelson, and J. Sérot. Evaluation of Hume and the Hume Methodology. EmBounded Project Deliverable, September 2008. Deliverable D33.

[SS08] J. Sérot and N. Scaife. Real-time Computer Vision Algorithms. EmBounded Project Deliverable, January 2008. Deliverable D27.