A User Guide to the *DAMSON* Language for BIMPA

D J Allerton

Department of Automatic Control
and Systems Engineering

University of Sheffield

Version 3.3
17 May 2010

Table of Contents

1		Introduction	3
2		The *DAMSON* Language	3
3		Program Structure	4
	3.1	Data Declarations	5
	3.2	Expressions	6
4		Directives	6
5		Built-in Procedures	7
6		Built-in Functions	8
7		Packets	9
8		Event-driven *DAMSON*	10
9		Interrupts	11
10		Threads and Semaphores	12
11		Examples	13
	11.1	A Small Program (with integers)	13
	11.2	Another Small Program (with floats)	15
	11.3	Clock Interrupts	16
	11.4	Packet Transfers	17
	11.5	Plotting	20
	11.6	An Integrate-and-fire Neuron Model	23
	11.7	Threads and Semaphores	27
12		Running a *DAMSON* Program	29
13		Limitations	29
14		Run-time Errors	29
Appendix 1		Fixed-point Arithmetic	31
Appendix 2		Intermediate Code	34

1 Introduction

DAMSON[1] is a compiler and simulator for the Spinnaker architecture. It is a subset of the C programming language, allowing the software for Spinnaker cores to be defined and simulated. Floating-point operations are implemented using fixed-point arithmetic, appropriate to the ARM processor used on Spinnaker. D*AMSON* enables users to define a set of network nodes and monitor their behaviour, simulating the multicast packet transfers provided by Spinnaker. The syntax of *DAMSON* supports the event-driven behaviour found in Spinnaker.

The main aim of *DAMSON* is to provide a tool to simulate software running on a number of Spinnaker cores sending packets between cores. Such a tool enables users (possibly without Spinnaker hardware) to develop and test their software on a PC. Although the code will run relatively slowly on a PC, it should be equivalent, in terms of its functional behaviour, on both Spinnaker hardware and off-line emulation. To provide a high degree of portability and a well-defined interface, *DAMSON* has the following characteristics:

- It is compatible with the majority of the syntax of ANSI-C;
- It will simulate 1000+ nodes running concurrently;
- It compiles and runs on gcc/MinGW for WindowsXP and gcc for Linux;
- It supports the type *float* using 32-bit fixed-point arithmetic where a *float* has a 16-bit integer part and a 16-bit fraction part (one possible implementation of scaled fixed-point arithmetic for the ARM), as outlined in Appendix 1;
- It supports multicast packet transmission – based on the underlying router architecture used in Spinnaker;
- The compiler produces code for a stack machine (similar to BCPL O-code), defined in Appendix 2, so that production of a code generator for the ARM processor should be straightforward or alternatively, a standard C compiler could be adapted for fixed-point multiplication and used with a library for interrupt handling.

2 The *DAMSON* Language

DAMSON is based on the C programming language. The only data types are *int* (32-bit integers) and *float* (32-bit floating-point) variables. *DAMSON* provides a full set of arithmetic and logical operations on variables, allows users to define procedures and functions and provides a set of built-in procedures and functions to transfer packets, access local clocks and send output to the terminal.

The following features in C are <u>not</u> currently supported in *DAMSON*:

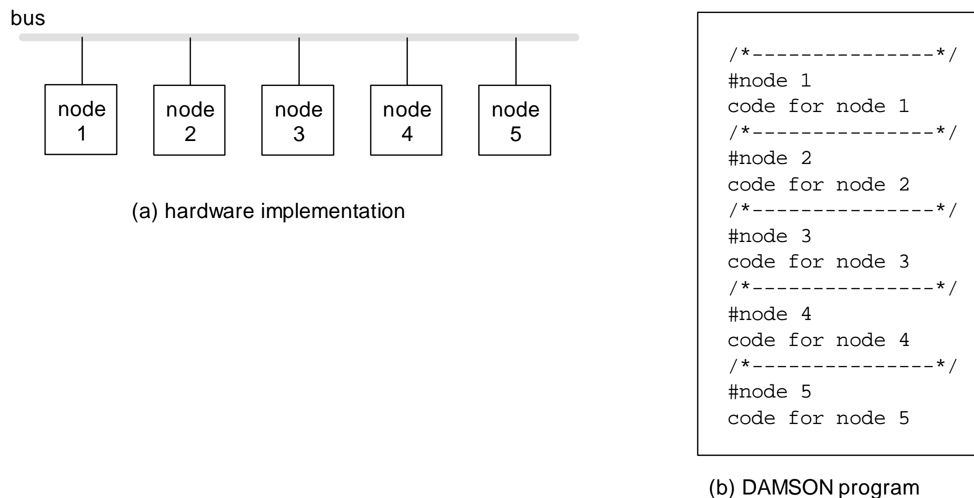
- unsigned int, char, unsigned char, double and long
- structures (the . and -> operators)
- the * (indirection) operator
- the ++ and -- operators

[1] *DAMSON* stands for Dave Allerton's Multi-core Simulation of Networks (until someone can think of a better acronym).

The argument for these restrictions in the language is that BIMPA applications are likely to be organised as multiple cores running relatively simple code, communicating via packets, where the omission of such features is unlikely to be significant.

3 Program Structure

DAMSON provides a software emulation of a network as shown in Figure 1. In these notes, the term *node* is used to define a processor or core running on a Spinnaker chip. Nodes can be declared in any order as sections of code to run on a specific processor. The typical organisation of a *DAMSON* program is shown in Figure 1.



(a) hardware implementation

(b) DAMSON program

Figure 1 Equivalence of a *DAMSON* program and its hardware implementation

The set of five nodes shown in Figure 1(a) is defined as a *DAMSON* program in Figure 1(b). Although the software for the five nodes is written as a sequential notation, it is simulated as concurrent processes. The actual node numbering is arbitrary; this example simply shows nodes 1 to 5 for clarity.

Figure 2 outlines the overall structure of a *DAMSON* program. The program defines all the nodes to be implemented. The variables declared at the start of a node are effectively static global variables (within that node). Variables passed in procedures or defined within a procedure are local variables, with the same scoping restrictions as C. User procedures must be declared as prototypes. The integer function *main* must be declared in the code section of each node, in order to provide a start address to execute code in a node.

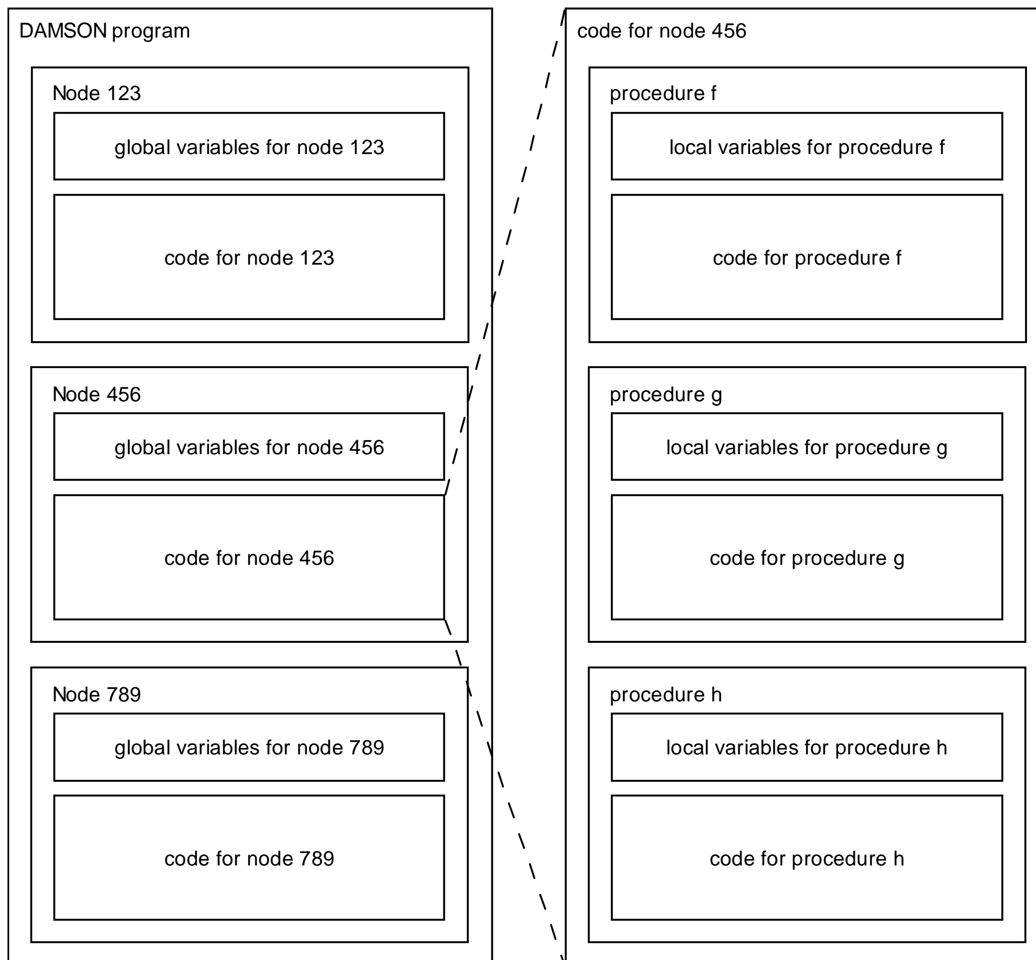


Figure 2 Structure of Nodes and Procedures in a *DAMSON* Program

In this example of three nodes, the nodes 123, 456 and 789 run concurrently. Note that the order of definition of the nodes is not important. The code generation for a node starts from the point of declaration of the node (#node n) and continues until a new node is defined. A node cannot access variables defined within another node and therefore, it is valid to declare variables with the same name in different nodes of a *DAMSON* program.

3.1 Data Declarations

DAMSON supports 32-bit integers in the range -2147483648 to +2147483647 and 32-bit floating point numbers represented by 16 bits for the integer part and 16 bits for the fractional parts in the range -32768.0 to +32767.0, as outlined in Appendix 1. This format for floating-point variables is only a convention, which could be changed or extended, if necessary. The scope of a global variable is limited to the node where the variable is declared whereas the scope of a local variable is the procedure where the variable is declared. Variables declared in one node can only access variables in other nodes by passing variables in a packet. The standard naming conventions of C apply, particularly the distinction between upper case and lower (e.g. xyz and XYZ are different variables).

3.2 Expressions

An expression describes arithmetic and logical operations. Local and built-in procedures can be called and functions can return a result. Normal operator precedence is assumed, e.g. a+b*c is the same as a+(b*c). A considerable depth of bracketing is provided for expressions and to override operator precedence. The full range of arithmetic and logical operators used in C is supported. The compiler supports the same rules as C for arithmetic with differing types and also provides explicit coercion of expressions, e.g. (int) and (float).

4 Directives

Directives are used in *DAMSON* to inform the compiler of some condition or setting and do not generate any code. They are identified by a hash symbol (#) at the start of a line and are summarised in Table 1.

Directive	Function	Arguments	Default
#timestamp	The true system-wide time is displayed when any output is generated (not the local node time)	'on' or 'off'	on
#node	Defines the start of a node	The node number	n/a
#monitor	Packets transfers are monitored and the details of each transfer are displayed	'on' or 'off'	on
#define	The same as the #define directive in C e.g. #define maxtablesize 5000		n/a
#record	Defines the period and sampling for data recording (x-axis in plotting)	t1, t2, Δt	1 ms
#plot	Defines the channel allocated to a plotted variable, the range of the channel and the axis label (y-axis in plotting)	n, var, r1, r2, "label"	n/a
#interrupt	Defines the list of nodes which transmit packets to the node and the code entry point to respond to each interrupt	A list of nodes	n/a
#include	The same as a #include directive in C e.g. #include "myheaderfile.h"	File name	n/a

Table 1 Directives

Note:

- The time displayed by the #timestamp directive is the true system time - the local clock of a node can be set to drift from the true system time (by default, local clocks are initially aligned with system-wide time with zero drift);

- The #monitor directive enables every packet transfer to be logged, displaying the source node, the destination node, the port number, the packet content and the time the packet is received.
- The #define directive is similar to C.
- The #interrupt directive specifies a list of source nodes from which a node can expect a packet – this is used by the compiler to construct a routing table, so that multicast packets are only transmitted to a specified list of nodes. For example, #interrupt 3, 12, 34..36, 50..59, 100 implies that a node expects to receive packets from nodes 3, 12, 34, 35, 36, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59 and 100. This directive also defines the entry point for specific interrupts (or group of interrupts) which must be provided as a void procedure, as outlined in section 9. In *DAMSON*, packet interrupts are numbered from 1 and the clock interrupt is assigned 0.

5 Built-in Procedures

DAMSON provides eight built-in procedures, which are used to sequence nodes and to transfer packets between nodes, as defined in Table 2.

Procedure	Description	Arguments
sendpkt(p, d)	Broadcasts the variable *d* from the current node, using port *p*	*d* - packet value *p* – port number (0-4095)
delay(n)	Delay the current node by *n* clock ticks	*n* - number of clock ticks
printf("message", args)	Same as the printf procedure used in C	Any number of arguments matching the message string
setclk(n)	Set the clock drift rate for the local clock of a node - initially zero	*n* - clock drift rate in ticks per unit time (only used to simulate clock drift)
exit(n)	Stop executing the code in a node	*n* - the exit value printed
signal(&s)	signal the semaphore *s*	*s* – a semaphore (int)
wait(&s)	wait for the semaphore *s*	*s* – a semaphore (int)
tickrate(t)	Set the clock to interrupt every *t* ticks – by default, t=1000	*t* – clock interrupt interval

Table 2 Built-in Procedures

Note:

- sendpkt broadcasts a packet to the other nodes;
- delay suspends the current process for the specified number of clock ticks (not time) where the clock tick rate is defined by the procedure tickrate;
- printf is identical to printf used in C;
- setclk resets the drift rate of the local clock in ticks per unit time;
- exit terminates a node (for the remainder of the program) – the only way to stop a *DAMSON* program is for all nodes to execute an exit (or issue a keyboard control-C);
- signal and wait are conventional semaphore operations;
- tickrate enables the user to simulate a real-time interrupt rate.

Note that type checking is relaxed for these system calls - there is no check on the type or number of arguments for built-in procedures and functions, e.g. exit(), exit(1), exit(2.3) and exit (3, 4, 5) are all valid, although only the first argument is used.

6 Built-in Functions

DAMSON provides five built-in functions as defined in Table 3.

Function	Description	Arguments
x=getclk()	Returns the current local clock value in ticks	n/a
x=abs(y)	Absolute value of the integer *y* is returned	An integer
x =fabs(y)	Absolute value of the float *y* is returned	A float
h=createthread(f, n)	Create a new thread *f*, with workspace *n*, returning a handle *h*	A procedure *f* and an integer *n*
Deletethread(h)	Delete the thread defined by handle *h*	An integer handle *h*

Table 3 Built-in Functions

Note:

- getclk reads the value of the local clock (which may drift) - the unit of ticks is not currently defined - it could for example be 1 μs;
- abs and fabs are the standard C absolute functions for ints and floats, respectively.
- createthread creates a new thread which invokes the procedure *f* with a workspace stack of size *n*. If *h*=0 the thread could not be created, otherwise *h* can be used to delete the thread. The thread is deleted when the procedure *f* returns.
- deletethread(h) deletes the thread associated with the handle *h*.

Note that type checking is also relaxed for these system calls.

7 Packets

At present, *DAMSON* supports multicast packet transfers. The other three forms of packet transfer (point-to-point, nearest-neighbour and fixed-route) will be provided as system procedures. The procedure sendpkt is used to transmit a packet to other nodes. At the receiving node, the #interrupt directive indicates where the node will be interrupted when a packet arrives from the transmitting node and the user provides a procedure to respond to this event.

At the transmitting node, *sendpkt* has two arguments: a port number and an optional data (or payload) value. The physical node address and the port number (0-4095) are combined as an identifier for the routing table hardware. The data can be a 32-bit integer or a 32-bit floating point value or can be omitted because the type checking and argument list checking rules are relaxed for system calls. For example, the following calls are valid for sendpkt:

```
#node 5

int a;
float x;

sendpkt(123, a);
sendpkt(123, x);
sendpkt(123);
```

At the receiving node, the packet interrupt is routed to a user-defined procedure. The node address of the transmitting node, the port number, the optional data content and the time of arrival of the packet are available as arguments to the user-defined procedure. For example, assume a multicast transmission by node 5 includes node 6 and that node 5 transmits to node 6 on three ports (123, 234 and 345). The following procedure myproc responds to the packet and can also respond to packets sent on different ports.

```
#node 6

float q;

#interrupt 5
void myproc(int n, int p, float d, int t);
{
    switch (p)
    {
        case 123:
            q = d;
            break;
        case 234:
            /* do something else */
            break;
        case 345:
            /* do something else */
            break;
    }
}
```

Note that the entry point could be defined for nodes 5 to 10, say, In this case, the node number *n* allows further classification/filtering of incoming packets.

For neural applications, the concept of a port corresponds to a neuron number. A core may simulate the behaviour of many neurons which transmit spikes to other neurons. At a receiving neuron, the port number identifies the neuron spike. In Figure 3, although all packets are transmitted via the bus and routing tables, the use of the port number provides discrete channels between node A and node B.

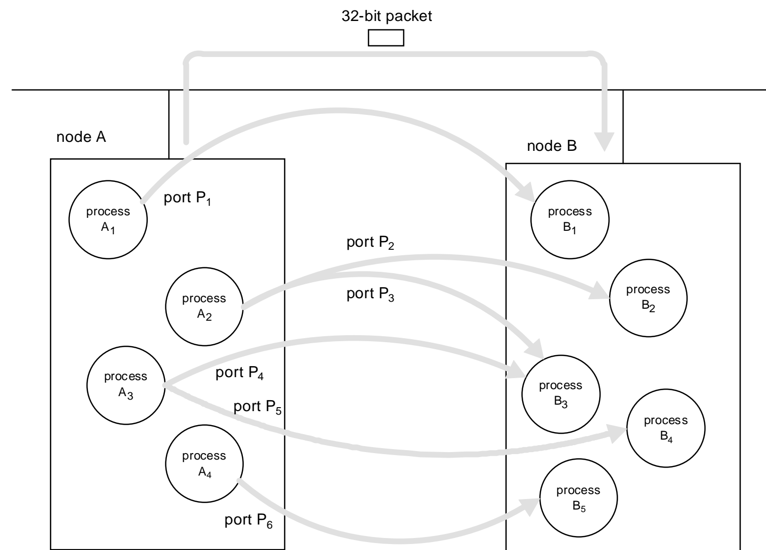


Figure 3 Port Numbers in packet transmissions

8 -Event-driven *DAMSON*

In conventional sequential C, a program waits for external events. For example, to read a packet from another computer, a typical programming construct could be:

```
s = getpacket(&v, mode)
```

where *s* is the status (or result) of the transfer, *v* is the memory address of the data structure set aside for the transfer and *mode* defines the form of transfer requested. In terms of the language syntax, transfers typically take the form of a system call. Other system calls are also provided to establish a link between a program and hardware ports (e.g. a socket for packet transfers), to terminate a link or to specify details of the transfer method.

The main problem with this approach is that the program waits at this point if no data is available (e.g. blocking I/O) or returns a status to indicate that the transfer failed to complete, which can include a timeout condition. However, it is difficult to avoid polling for such events and unnecessary delay can occur while the processor is waiting for an external event. In Spinnaker, polling needs to be avoided and a core should be set to standby mode when it has nothing to do, typically if it waiting for an input from another core or a clock tick or a DMA transfer to complete.

An alternative representation is to associate code directly with external events. The code is only invoked when a specific event occurs. If a core reaches a state when it has nothing to compute, it is set into standby (power save) mode and will only be reactivated when a new external event occurs or a clock interrupt occurs. Of course, the implication is that any number of concurrent processes (including zero) may be active at any time on a core, requiring an underlying scheduler for concurrent processes. One further implication is that if

several processes share a resource, they will need some form of synchronisation to ensure consistent access, e.g. a semaphore mechanism.

Event-driven coding is supported in *DAMSON* by directing interrupts (typically from incoming packets and the real-time clock) to specific code segments. This notation is illustrated in the following example:

```
#node 1

/* prototypes */
void f1(int, int, int, int);
void f2(int, int, int, int);
void f3(int, int, int, int);
void clockint(int, int, int, int);

int main()
{
    /* code for the main program */
   return 0;
}

# interrupt e1
void f1(int node, int port, int d1, int t1)
{
    /* code to respond to event 1 goes here */
}

# interrupt e2
void f2(int node, int port, int d2, int t2)
{
    /* code to respond to event 2 goes here */
}

# interrupt e3
void f3(int node, int port, int d3, int t3)
{
    /* code to respond to event 3 goes here */
}

# interrupt 0
void clockint(int node, int port, int data, int time)
{
    /* code to respond to a clock interrupt goes here */
}
```

The run-time system (or operating system) is responsible for management of the processes. When an event (given by e1, e2, e3 or a clock interrupt) occurs, the current process is suspended and the process associated with the new event is activated. On completion of a process, it is deleted (in terms of its work space) and another active process is resumed.

9 Interrupts

Interrupts form the basis of event-driven *DAMSON*. In *DAMSON*, interrupts occur as a result of an incoming packet or a clock tick. The user can define the source nodes of expected packets (see the #interrupt directive in section 4) and also the entry point for interrupts. *DAMSON* allows a user to write an interrupt handler to respond to one specific interrupt, a range of interrupts or a group of interrupts.

In addition, *DAMSON* supports packets with no data payload and enables a user to identify the source address of an incoming packet, the port number and its time of arrival (in ticks). By accessing the source address of a packet, one common interrupt response can be used for a group of interrupts from different node, while allowing a response to be customised for specific interrupts. The time of arrival of arrival of a packet can enable delays or propagation to be modelled.

Consider a simple example:

```
#interrupt 3, 5..7, 10, 15..18
#interrupt 98

myprocedure(int n, int p, int x, int t)
{
    /* code to respond to the interrupt goes here */
}
```

Packets from nodes 3, 5, 6, 7, 10, 15, 16, 17, 18 and 98 will be responded to by the code in *myprocedure*. All these interrupts will activate a new process, calling *myprocedure* with four arguments: *n* – the source node of the packet, *p* the port number, *x* the packet content and *t* the local time the packet arrived. This procedure can call other local procedures as well as system procedures, such as printf. On completion of *myprocedure*, the process is terminated.

Note that an interrupt handler can be interrupted, for example, where two interrupts occur within a very short time of each other. At present, there is no prioritisation of interrupts. The parameter *x* should have the same type as the parameter used in *sendpkt* at the source node. The incoming packet is simply copied to *x* – there is no conversion.

10 Threads and Semaphores

Within a core, several processes may run concurrently. In particular, each interrupt generates a new process, which is created dynamically by the run-time system. When the primary procedure of a process returns, it is deleted as a process. Strictly, these processes are organised as threads. The threads share a common memory space (particularly global variables) and they can share code, but each thread has its own local work space, which is used for local variables and parameter passing.

A thread is either created and deleted implicitly by the run-time system or is created and deleted explicitly by the parent thread. In the latter case, two procedures are provided:

```
h =  createthread(f, n)
```

and

```
deletethread(h)
```

where *f* is a procedure defined in the local node and *n* is the workspace requested. If the thread is created, the handle *h* is non-zero. If an error occurs during creation of the thread, *h* = 0. The handle *h* is also used to delete the thread, unless a return from *f* is executed. As soon as the thread is created, it is active and can start executing code.

Each thread is in one of three states:

- Running – it is able to execute code, if scheduled
- Delaying – suspended until the period of delay is completed
- Waiting – waiting on a semaphore (and unable to run)

As a thread can be interrupted (or pre-empted), two semaphore operations are provided in *DAMSON* to enable threads to synchronise or to share common resources (mutual access):

```
signal(&s)
wait(&s)
```

where *s* is an integer variable, typically a global variable shared by several threads of a node. The signal and wait operations are atomic and are equivalent to the pseudo-code below:

```
signal(&s)             increment(s)

wait(&s)               repeat
                       {
                           if (s > 0)
                           {
                               decrement(s)
                               return
                           }
                       }
```

In practice, the implied delay in waiting for the semaphore *s* to become greater than zero, results in the thread being suspended until the semaphore condition is satisfied, rather than continuous polling of the semaphore.

11 Examples

The following examples are intended to illustrate the capability of *DAMSON*. The programs were run on a Viglen 2.00 GHz PC with 2 Gbyte of RAM under Debian Linux.

Example 11.1 A Small Program (with integers)

The program contains a single node and computes factorial numbers from 1! to 10!. Note that the function *main* terminates with an exit.

```
#node 1

#timestamp on

int f(int);

int main()
{
    int i;

    for (i=1; i<=10; i+=1)
    {
        printf("f(%d)=%d\n", i, f(i));
    }
    exit(99);
}

int f(int n)
{
    return (n <= 1) ? 1 : n * f(n-1);
}
```

Running the program produces the following output:

```
DAMSON Version 3.3
Copyright (C) Dave Allerton 2010
Fri May 14 11:53:58 2010

Nodes: 1
Workspace: 0.006004 MB
        21: 1   f(1)=1
        56: 1   f(2)=2
       102: 1   f(3)=6
       159: 1   f(4)=24
       227: 1   f(5)=120
       306: 1   f(6)=720
       396: 1   f(7)=5040
       497: 1   f(8)=40320
       609: 1   f(9)=362880
       732: 1   f(10)=3628800
Node (1) Exit 99
Execution time: 0.000149 secs
Computing ticks: 745
Standby ticks: 0 (   0.00%)
```

The preamble defines the compiler version number, the date and time of the compilation, the number of nodes in the program, the workspace requirement (Mbytes). The first column is the timestamp of the local clock of node 1. The 1 on each line denotes that the output is produced by node 1.

The information printed after the program runs, indicates that it terminated with code 99, the program ran for 0.000149s (physical compute time), it executed for 745 ticks and that the processor was in standby mode for 0 ticks (it was never waiting). In practice, the PC clock is too coarse to indicate very small computation times whereas Linux provides a more accurate measurement of elapsed time. The standby ticks value corresponds to the amount of time the core would have been switched to standby mode.

The resultant intermediate code is as follows:

```
DAMSON Version 3.2
Copyright (C) Dave Allerton 2010
Wed Mar 24 08:29:09 2010

Nodes: 1
Workspace: 0.006004 Mbytes
Routing Table
Node: 1
1: ENTRY main
2: PUSHC 1
3: POPL i
4: PUSHL i
5: PUSHC 10
6: COMP_<=
7: JF L21
8: JUMP L14
9: PUSHL i
10: PUSHC 1
11: ADD
12: POPL i
13: JUMP L4
14: PUSHSTR f(%d)=%d\n
15: PUSHL i
```

```
16: PUSHL i
17: CALL 25
18: PUSHC 3
19: SYS_CALL printf
20: JUMP L9
21: PUSHC 99
22: PUSHC 1
23: SYS_CALL exit
24: RETURN main
25: ENTRY f
26: PUSHL n
27: PUSHC 1
28: COMP_<=
29: JF L32
30: PUSHC 1
31: JUMP L38
32: PUSHL n
33: PUSHL n
34: PUSHC 1
35: SUB
36: CALL 25
37: MULT
38: RETURN f
```

The intermediate code for the function `main` is given on lines 1-24 and the intermediate code for the function `f` is on lines 25-38.

Example 11.2 Another Small Program (with floats)

The following *DAMSON* example shows a floating point function to compute the sine of an angle. The series expansion is truncated after three terms.

```
#node 1

float sin(float x);

int main()
{
    int a;
    float x;
    float s;

    for (a=0; a<=90; a=a+10)
    {
        x = a / 57.29577951;
        s = sin(x);
        printf("sin(%d) = %f\n", a, s);
    }
    exit(23);
}

float sin(float x)
{
    return x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0;
}
```

The output from this program is shown below.

```
DAMSON Version 3.3
Copyright (C) Dave Allerton 2010
Fri May 14 12:09:55 2010

Nodes: 1
Workspace: 0.006004 MB
1   sin(0) = 0.000000
1   sin(10) = 0.173645
1   sin(20) = 0.342010
1   sin(30) = 0.500000
1   sin(40) = 0.642776
1   sin(50) = 0.766037
1   sin(60) = 0.866028
1   sin(70) = 0.939697
1   sin(80) = 0.984772
1   sin(90) = 0.999863
Node (1) Exit 23
Execution time: 0.000137 secs
Computing ticks: 650
Standby ticks: 0 (  0.00%)
```

The emulation of this small program took 650 ticks and the program executed in 137 μs.

Example 11.3 Clock Interrupts

The following example shows the response to clock interrupts.

```
#node 1

void clockint(int, int, int, int);

int ticks = 0;

int main()
{
    int a, b;

    printf("starting\n");

    for (a=1; a<=1000; a+=1)
    {
        b = a * a;
    }

    printf("ticks=%d\n", ticks);
    exit(123);
}

#interrupt 0

void clockint(int n, int p, int d, int t)
{
    ticks += 1;
}
```

The code in *main* simply computes the square of an integer value 1000 times and stops. While *main* is executing, the clock interrupts from the local clock of node 1 invoke a process defined by the procedure *clockint*, which increments the clock tick count. By default, the clock tick rate is set to 1000 ticks/s (1000 Hz). Note that the variable *ticks* is a global variable, which is initialised by *main* and is incremented by *clockint*. The output is shown below:

```
DAMSON Version 3.3
Copyright (C) Dave Allerton 2010
Fri May 14 12:18:16 2010

Nodes: 1
Workspace: 0.006004 MB
1   starting
1   ticks=15
Node (1) Exit 123
Execution time: 0.002267 secs
Computing ticks: 15107
Standby ticks: 0 (  0.00%)
```

The program ran for 15107 ticks, accumulating 15 clock ticks in the interrupt handler in 2 ms.

Example 11.4 Packet Transfers

In the following example, there are three nodes, 100, 200 and 300. Node 100 broadcasts a packet to nodes 200 and 300. When they receive their packet, node 200 broadcasts a packet to nodes 100 and 300 and node 300 broadcasts a packet to nodes 100 and 200. The sequence repeats for 5 clock interrupts (5000 ticks).

```
/* pkt testing #1 */

#monitor on
#timestamp on

/* -------------------------------------------- */

#node 100

void clockint(int, int, int, int);
void pktint200(int, int, float, int);
void pktint300(int, int, float, int);

int ticks;

int main()
{
    tickrate(1000);
    ticks = 0;
    return 1;
}

#interrupt 200
void pktint200(int n, int p, float d, int t)
{
    printf("Node 100: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 300
void pktint300(int n, int p, float d, int t)
{
    printf("Node 100: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 0
void clockint(int n, int p, int d, int t)
{
    sendpkt(123, 1.0);
    ticks += 1;
    if (ticks >= 5)
        exit(1);
```

```
}

/* -------------------------------------------- */

#node 200

void clockint(int, int, int, int);
void pktint100(int, int, float, int);
void pktint300(int, int, float, int);

int ticks;

int main()
{
    tickrate(1000);
    ticks = 0;
    return 2;
}

#interrupt 100
void pktint100(int n, int p, float d, int t)
{
    printf("Node 200: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 300
void pktint300(int n, int p, float d, int t)
{
    printf("Node 200: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 0
void clockint(int n, int p, int d, int t)
{
    sendpkt(234, 2.0);
    ticks += 1;
    if (ticks >= 5)
        exit(2);
}

/* -------------------------------------------- */

#node 300

void clockint(int, int, int, int);
void pktint100(int, int, float, int);
void pktint200(int, int, float, int);

int ticks;

int main()
{
    tickrate(1000);
    ticks = 0;
    return 3;
}

#interrupt 100
void pktint100(int n, int p, float d, int t)
{
    printf("Node 300: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 200
void pktint200(int n, int p, float d, int t)
{
    printf("Node 300: port=%d pkt%d=%f t=%d\n", p, n, d, t);
}

#interrupt 0
```

```
void clockint(int n, int p, int d, int t)
{
    sendpkt(345, 3.0);
    ticks += 1;
    if (ticks >= 5)
        exit(3);
}
```

The output generated by the program is shown below:

```
DAMSON Version 3.3
Copyright (C) Dave Allerton 2010
Fri May 14 15:43:21 2010

Nodes: 3
Workspace: 0.018012 MB
      1005:  100->200 port 123 [65536] Rx:1005
      1005:  100->300 port 123 [65536] Rx:1005
      1012: 200   Node 200: port=123 pkt100=1.000000 t=1004
      1012: 300   Node 300: port=123 pkt100=1.000000 t=1004
      1014:  200->100 port 234 [131072] Rx:1014
      1014:  200->300 port 234 [131072] Rx:1014
      1021: 300   Node 300: port=234 pkt200=2.000000 t=1013
      1022: 100   Node 100: port=234 pkt200=2.000000 t=1014
      1023:  300->100 port 345 [196608] Rx:1023
      1023:  300->200 port 345 [196608] Rx:1023
      1031: 100   Node 100: port=345 pkt300=3.000000 t=1023
      1031: 200   Node 200: port=345 pkt300=3.000000 t=1023
      2005:  100->200 port 123 [65536] Rx:2005
      2005:  100->300 port 123 [65536] Rx:2005
      2012: 200   Node 200: port=123 pkt100=1.000000 t=2004
      2012: 300   Node 300: port=123 pkt100=1.000000 t=2004
      2014:  200->100 port 234 [131072] Rx:2014
      2014:  200->300 port 234 [131072] Rx:2014
      2021: 300   Node 300: port=234 pkt200=2.000000 t=2013
      2022: 100   Node 100: port=234 pkt200=2.000000 t=2014
      2023:  300->100 port 345 [196608] Rx:2023
      2023:  300->200 port 345 [196608] Rx:2023
      2031: 100   Node 100: port=345 pkt300=3.000000 t=2023
      2031: 200   Node 200: port=345 pkt300=3.000000 t=2023
      3005:  100->200 port 123 [65536] Rx:3005
      3005:  100->300 port 123 [65536] Rx:3005
      3012: 200   Node 200: port=123 pkt100=1.000000 t=3004
      3012: 300   Node 300: port=123 pkt100=1.000000 t=3004
      3014:  200->100 port 234 [131072] Rx:3014
      3014:  200->300 port 234 [131072] Rx:3014
      3021: 300   Node 300: port=234 pkt200=2.000000 t=3013
      3022: 100   Node 100: port=234 pkt200=2.000000 t=3014
      3023:  300->100 port 345 [196608] Rx:3023
      3023:  300->200 port 345 [196608] Rx:3023
      3031: 100   Node 100: port=345 pkt300=3.000000 t=3023
      3031: 200   Node 200: port=345 pkt300=3.000000 t=3023
      4005:  100->200 port 123 [65536] Rx:4005
      4005:  100->300 port 123 [65536] Rx:4005
      4012: 200   Node 200: port=123 pkt100=1.000000 t=4004
      4012: 300   Node 300: port=123 pkt100=1.000000 t=4004
      4014:  200->100 port 234 [131072] Rx:4014
      4014:  200->300 port 234 [131072] Rx:4014
      4021: 300   Node 300: port=234 pkt200=2.000000 t=4013
      4022: 100   Node 100: port=234 pkt200=2.000000 t=4014
      4023:  300->100 port 345 [196608] Rx:4023
      4023:  300->200 port 345 [196608] Rx:4023
      4031: 100   Node 100: port=345 pkt300=3.000000 t=4023
      4031: 200   Node 200: port=345 pkt300=3.000000 t=4023
      5005:  100->200 port 123 [65536] Rx:5005
      5005:  100->300 port 123 [65536] Rx:5005
      5012: 200   Node 200: port=123 pkt100=1.000000 t=5004
      5012: 300   Node 300: port=123 pkt100=1.000000 t=5004
```

```
      5014:  200->100 port 234 [131072] Rx:5014
      5014:  200->300 port 234 [131072] Rx:5014
      5021: 300   Node 300: port=234 pkt200=2.000000 t=5013
      5022: 100   Node 100: port=234 pkt200=2.000000 t=5014
      5023:  300->100 port 345 [196608] Rx:5023
      5023:  300->200 port 345 [196608] Rx:5023
      5031: 100   Node 100: port=345 pkt300=3.000000 t=5023
      5031: 200   Node 200: port=345 pkt300=3.000000 t=5023
Node (100) Exit 1
Node (200) Exit 2
Node (300) Exit 3
Execution time: 0.000358 secs
Computing ticks: 15102
Standby ticks: 14592 ( 96.62%)
Node=100 TxPkts=5 RxPkts=10
Node=200 TxPkts=5 RxPkts=10
Node=300 TxPkts=5 RxPkts=10
```

Note that monitoring is turned on so that each transfer is logged to display the source node, destination node, packet contents and time of transmission. Note also that the arguments of the system call *sendpkt* and the procedure *interrupt* are of type float but equally, could have been specified as type int.

Note that 15 packets were broadcast and 30 packets were received. The program completed in 15093 ticks and 14628 ticks occurred while the nodes were waiting for an interrupt (cores in standby mode).

<u>Example 11.5 Plotting</u>

Although *DAMSON* output can be routed directly to a file and post-processed, *DAMSON* also supports graphical output compatible with GNUplot. The start and stop times of the recording and the sampling interval are defined by the directive #record. Similarly, a variable in a node can be recorded on any one of up to 10 channels, where the numeric range and axis label is defined by the #plot directive.

Consider a simple example of two nodes running at a frame rate of 1 ms. One process generates a pulse train output at 5 Hz which is sent to the other node every 100 ms, which uses this pulse train to generate a 1.25 Hz pulse train. Both pulse trains are recorded and plotted. The following *DAMSON* program plots the variable *output* of both nodes 100 and 200.

```
/* pulse train example
   DJA 2 March 2010 */

#monitor on
#timestamp on
#record 0.0 5.0 0.001

/* --------------------------------------------- */
#node 100

void clockint(int, int, int, int);

float output = 1.0;
int ticks = 0;
#plot 1 output -2 2 node 100

int main()
{
    tickrate(1000);
    return 1;
}

#interrupt 0
void clockint(int n, int p, int x, int t)
{
    ticks += 1;
    if (ticks % 100 == 0)
    {
        output = -output;
        sendpkt(1234, output);
        if (ticks >= 5000)
        {
            exit(100);
        }
    }
}

/* --------------------------------------------- */
#node 200

void pktint100(int, int, float, int);
void clockint(int, int, int, int);

int ticks;
int pulses;
float output;

#plot 2 output -2 2 node 200

int main()
{
    ticks = 0;
    pulses = 0;
    output = 1.0;
    return 2;
}

#interrupt 100
void pktint100(int n, int p, float d, int t)
{
    pulses += 1;
    if (pulses % 4 == 0)
    {
        output = -output;
    }
}

#interrupt 0
void clockint(int n, int p, int x, int t)
```

```
{
    ticks += 1;
    if (ticks >= 5000)
    {
        exit(200);
    }
}
```

A fragment of the data file produced by the program is shown below:

```
0.093000, 0.000000, 1.000000
0.094000, 0.000000, 1.000000
0.095000, 0.000000, 1.000000
0.096000, 0.000000, 1.000000
0.097000, 0.000000, 1.000000
0.098000, 0.000000, 1.000000
0.099000, 0.000000, 1.000000
0.100000, -1.000000, 1.000000
0.101000, -1.000000, 1.000000
0.102000, -1.000000, 1.000000
0.103000, -1.000000, 1.000000
0.104000, -1.000000, 1.000000
0.105000, -1.000000, 1.000000
```

The *DAMSON* program also generates a plot script which is compatible with GNUplot. The script file is generated in the same directory as the *DAMSON* program. A *DAMSON* program xyz.d will generate the files xyz.dat and xyz.plt, if any data is explicitly recorded and is generated for plotting. A typical GNUplot script (automatically generated by *DAMSON*) is shown below:

```
set terminal png truecolor font arial 8 size 600,800
set output "test252.png"
set size 1,1
set origin 0,0
set lmargin 10
set multiplot
set grid
set format y "%5g"
set size 1.0, 0.500000
set origin 0, 0.500000
set ylabel "node 100"
set xr[0.000000:5.000000]
set yr[-2.000000:2.000000]
plot 'test252.dat' using 1:2 notitle with lines
set origin 0, 0.000000
set ylabel "node 200"
set xr[0.000000:5.000000]
set yr[-2.000000:2.000000]
plot 'test252.dat' using 1:3 notitle with lines
unset multiplot
reset
set output
```

The resultant .png file produced by running GNUplot can be imported directly into standard packages such as MS Word. In this case, a *DAMSON* program xyz.d will generate a file xyz.png, as shown in Figure 4.

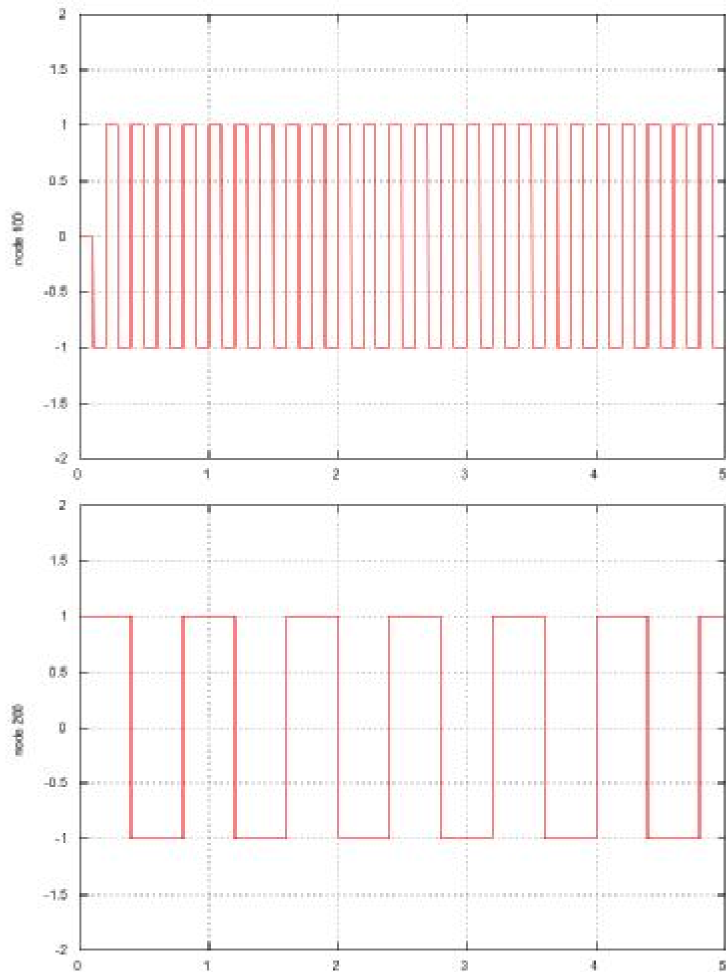


Figure 4 A GNUplot Example

Example 11.6 An Integrate-and-fire Neuron Model

The following *DAMSON* program of three nodes implements a simple integrate-and-fire neuron model with spiking inputs produced by node 1, where node 2 exhibits the characteristics of a Tsodyks and Markram facilitating synapse and node 3 represents a Tsodyks and Markram depressing synapse.

```
/* IF_cond_exp /w Tsodyks & Markram synapses example
   Leaky integrate and fire model with fixed threshold
   and exponentially-decaying post-synaptic conductance.

   IF neurons fed by spike source array with
   depressing and facilitating Tsodyks & Markram synapses.

   Node 2 models a Tsodyks and Markram facilitating synapse
   Node 3 models a Tsodyks and Markram depressing synapse */

#timestamp on
#record 0.0 2.0 0.02   /* 50 Hz  sampling */

/* ---------------------------------- */
/* spike_source_array */

#node 1

void clockint(int, int, int, int);
```

```
int ticks;

int main()
{
    tickrate(20000);  /* 50 Hz frame rate */
    ticks = 0;
    return 1;
}

#interrupt 0
void clockint(int n, int p, int x, int t)
{
    float source;
    #plot 1 source 0.0 1.0 source

    ticks += 1;
    if ((ticks < 100) && (ticks % 10) == 0)
    {
        source = 1.0;
        sendpkt(0, source);  /* generate a spike every 200 ms, during the 1st sec) */
    }else{
        source = 0.0;
    }
    if (ticks >= 200)
    {
        exit(1);
    }
}

/* ---------------------------------- */
/* IF_cond_exp neuron */

#node 2

void clockint(int, int, int, int);
void pktint(int, int, float, int);

float G, ge, gi, i_offset, i_inj, e_rev_E, e_rev_I, c_m;
float v, v_reset, v_rest, tau_m;
float tau_syn_E, tau_syn_I, dt;
float R, U, tau_rec, tau_fac, Use, Gtm;
int ticks, pkt1;

int main()
{
    tickrate(20000);

    ge=0.0;
    gi=0.0;
    i_offset=0.1;
    i_inj=0.0;
    e_rev_E=0.0;
    e_rev_I=-75.0;
    c_m=1.0;
    v_reset=-70.0;
    v_rest=-65.0;
    tau_m=200.0;
    tau_syn_E=5.0;
    tau_syn_I=5.0;
    dt=1.0;
    ticks=0;

    v = v_rest;

    /* Facilitating TsodyksMarkram mechanism  */
    tau_rec=100.0;
    tau_fac=20000.0;
    Use=0.04;
    R=1.0;
```

```
    U=0.0;
    pkt1 = 0;
    return 2;
}

#interrupt 1    /* Synapse input */
void pktint(int n, int p, float x, int t)
{
    pkt1 = 1;
}

#interrupt 0
void clockint(int n, int p, int t, int clk)
{
    float output;
    #plot 2 output 0.0 0.002 facilitating synapse (nS)

    ticks += 1;
    if (pkt1 != 0)
    {
        gi+=0.01;
        // TsodyksMarkram mechanism
        R = R + dt*((1.0-R)/tau_rec - U*R);
        U = U + dt*((-U/tau_fac) + Use*(1.0-U));
        pkt1 = 0;
    }
    else
    {
        R = R + dt*((1.0-R)/tau_rec);
        U = U + dt*((-U/tau_fac));
    }

    /* Synapse - exponential-decaying post-synaptic conductance  */
    Gtm = R*U;
    G = (ge*(e_rev_E-v) + gi*Gtm*(e_rev_I-v) + i_offset + i_inj)/c_m;
    ge = ge + dt*(-ge/tau_syn_E);
    gi = gi + dt*(-gi/tau_syn_I);

    /* Membrane potential */
    v = v + dt*(G + (v_rest-v)/tau_m);
    output = gi*Gtm;

    if (ticks >= 200)
    {
        exit(2);
    }
}
/* ---------------------------------- */
/* IF_cond_exp neuron */

#node 3

void clockint(int, int, int, int);
void pktint(int, int, float, int);

float G, ge, gi, i_offset, i_inj, e_rev_E, e_rev_I, c_m;
float v, v_reset, v_rest, tau_m;
float tau_syn_E, tau_syn_I, dt;
float R, U, tau_rec, tau_fac, Use, Gtm;
int ticks, pkt1;

int main()
{
    tickrate(20000);

    ge=0.0;
    gi=0.0;
    i_offset=0.1;
    i_inj=0.0;
    e_rev_E=0.0;
```

```
    e_rev_I=-75.0;
    c_m=1.0;
    v_reset=-70.0;
    v_rest=-65.0;
    tau_m=200.0;
    tau_syn_E=5.0;
    tau_syn_I=5.0;
    dt=1.0;
    ticks=0;

    v = v_rest;

    /* Depressing TsodyksMarkram mechanism  */
    tau_rec=800.0;
    tau_fac=0.0;
    Use=0.5;
    R=1.0;
    U=Use;
    pkt1 = 0;
    return 3;
}

#interrupt 1    /* Synapse input */
void pktint(int n, int p, float x, int t)
{
    pkt1 = 1;
}

#interrupt 0
void clockint(int n, int p, int t, int clk)
{
    float output;
    #plot 3 output 0.0 0.002 depressing synapse (nS)

    ticks += 1;
    if (pkt1 == 1)
    {
        gi = gi + 0.01;
        // TsodyksMarkram mechanism
        R = R + dt*((1.0-R)/tau_rec - U*R);
        pkt1 = 0;
    }
    else
    {
        R = R + dt*((1.0-R)/tau_rec);
    }

    /* Synapse - exponential-decaying post-synaptic conductance  */
    Gtm = R*U;
    G = (ge*(e_rev_E-v) + gi*Gtm*(e_rev_I-v) + i_offset + i_inj)/c_m;
    ge = ge + dt*(-ge/tau_syn_E);
    gi = gi + dt*(-gi/tau_syn_I);

    /* Membrane potential */
    v = v + dt*(G + (v_rest-v)/tau_m);
    output = gi*Gtm;

    if (ticks >= 200)
    {
        exit(3);
    }
}
```

The GNUplot output from the *DAMSON* program is shown in Figure 5.

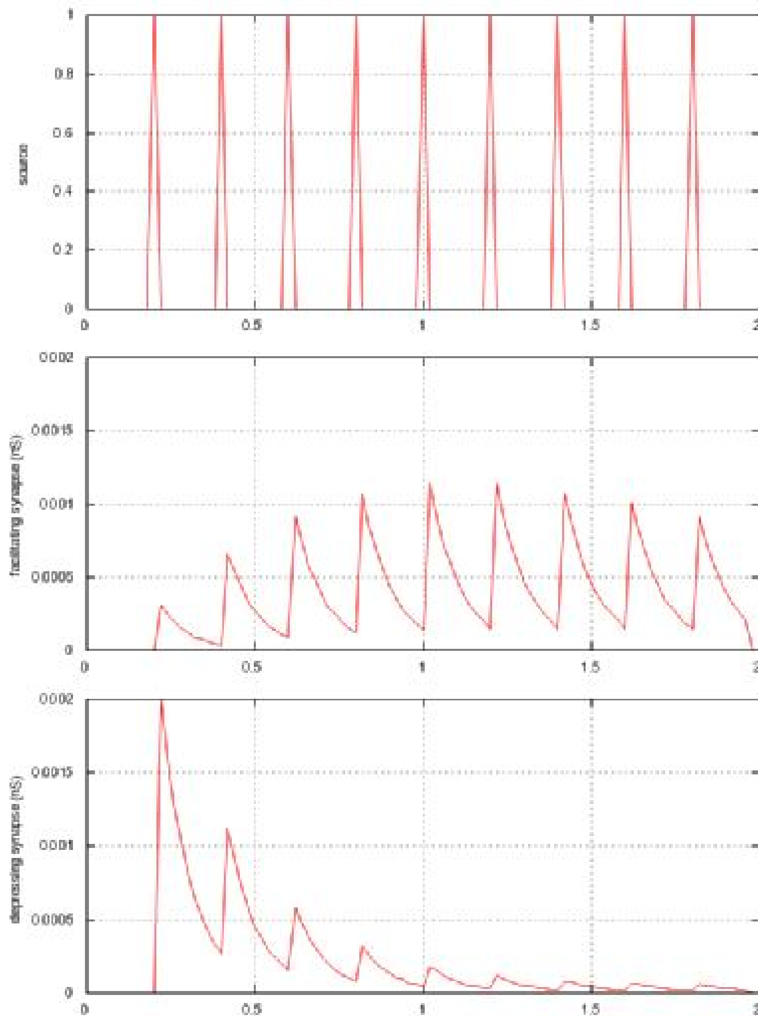


Figure 5 Graphical Output from GNUplot – Integrate-and Fire Neuron Model

Example 11.7 Threads and Semaphores

The 'dining philosophers' problem illustrates an application of multiple threads and semaphores. Assume there are five philosophers; they either think or eat at a table set for five places, as shown in Figure 6, where both actions take a random time.

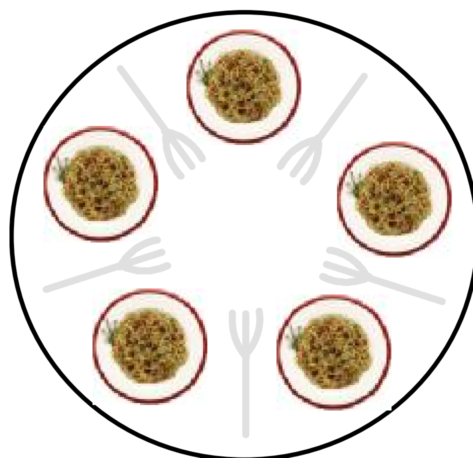


Figure 6 The Dining Philosophers Problem

However, in order to dine, a philosopher needs both a left hand fork and a right hand fork. If the adjacent left hand fork or right hand fork is in use, the philosopher must wait to eat. The objective is to implement an algorithm using semaphores to ensure that no philosopher starves (the issues of hygiene of sharing forks and spaghetti slipping through the forks are not addressed here – these are strictly hardware problems).

The code for a single philosopher is shown below:

```
void p2()
{
    int eating;
    int thinking;

    for (;;)
    {
        thinking = randomnumber(100);
        printf("p2 thinking %d\n", thinking);
        delay(thinking);

        wait(&c);
        wait(&f2);
        wait(&f3);
        signal(&c);
        eating = randomnumber(20);
        printf("p2 eating %d\n", eating);
        delay(eating);
        signal(&f2);
        signal(&f3);
    }
}
```

The semaphore *c* is used as a guard to access the set of forks (initially c=4 for 5 philosophers), while the nth philosopher needs to access the nth fork and the $(n+1)^{th}$ fork, with their respective semaphores. The procedure *p2* is created as a separate thread, with five threads for the five philosophers. Note that the actions of thinking and eating are modelled as random delays.

The code to create the threads is shown below:

```
int main()
{
    int h;

    Seed = 123456789;

    h = createthread(p1, 500);
    h = createthread(p2, 500);
    h = createthread(p3, 500);
    h = createthread(p4, 500);
    h = createthread(p5, 500);
    delay(50000);
    exit(12345);
}
```

The procedure *main* is created as a thread, which in turn creates the five threads *p1* to *p5*, representing the five philosophers. The procedure *main* is itself delayed for 50,000 ticks until it terminates with an *exit*, stopping the node (and all its threads).

12 Running a *DAMSON* Program

From an MS Windows MinGW terminal or a Linux shell, type:

```
./damson filename.d
```

where *filename* is the name of the *DAMSON* source file. Note that *DAMSON* expects source files to have an extension .d. The *DAMSON* program is compiled and error messages are logged if any compilation errors are detected.

If no errors are detected during compilation, the *DAMSON* program is executed and the output from the program is written to the terminal. To divert output from a *DAMSON* program to a file (e.g. results.txt) use:

```
./damson filename.d > results.txt
```

To view the intermediate code produced by the *DAMSON* compiler, add the switch *-dis* to the command line, for example:

```
./damson test23.d –dis
```

To generate a routing table, add the switch –rt. A file name the same name as the source file and the extension .tab will be generated.

```
./damson prog.d –rt
```

13 Limitations

The *DAMSON* language is designed to provide a simple programming language to develop BIMPA applications. However, the compiler is small and has a number of limitations:

- Compilation errors only indicate the line where the error was detected, not the position within the line;
- There are no checks on the number of parameters or types of parameters passed in built-in (system) procedures or functions – this is similar to printf in C;
- There is a predefined limit to the program code size of a compiled *DAMSON* program, the number of variables declared for a node and workspace size for the stack of each node. These are arbitrary constants in the compiler and can easily be modified.

14 Run-time Errors

With event-driven applications, there are relatively few run-time errors. For example, packets are not queued, rather they generate a new process in another node and similarly, it may be appropriate for a packet never to be generated, making a read timeout condition redundant.

The following list is a summary of the major run-time errors that can be encountered in *DAMSON* programs:

Error	Description
Too Many Processes	Each interrupt activates a process, which requires workspace; if there are too many interrupts, an error condition is raised if there is insufficient space for the process.
No Node Defined	If code is generated for a node without defining the node number, an error is raised.
Node Allocation	Nodes are allocated dynamically – if there is insufficient memory, an error is raised.
Stack Overflow/Underflow	A finite amount of stack is provided to support local variables, parameter passing and recursion. If an attempt is made to place an item on the stack but there is insufficient space, a stack overflow exception is raised. Similarly, if an attempt is made to remove an item from an empty stack, a stack underflow exception is raised.
Multicast Error	In formulating the routing tables, each node has a list of its output nodes. If any attempt is made to transmit a packet to a node which is not expecting a packet from the source node, an error is raised.
Procedure main Missing	Each node starts with the procedure *main*. If this procedure is not defined for a node, a run-time exception is raised.
PC Out of Range	Each node has a finite code space – any attempt to access code outside this range is flagged.
Unknown Instruction	If a node attempts to execute an instruction which is not recognised by the interpreter, a run-time error is raised.
Link Allocation	The routing table information is allocated dynamically – if there is insufficient memory, an error is raised.
Data Recording Overflow	A finite amount of storage is provided to record data variables. If the space available is insufficient, this condition is raised.

Appendix 1

Fixed-point arithmetic

The use of *DAMSON* for the ARM processor can:

- Avoid using any floating-point software as a floating-point operation would require several hundred machine instructions.
- Allow the user to specify both integers (int) and floating point variables (float).
- Provide a reasonable resolution (accuracy) using a scaled fixed-point notation.
- Implement scaled fixed-point arithmetic to run on an ARM processor and to be emulated on a PC as fast as possible and in a straightforward way.

In *DAMSON*, the user simply defines a variable as an *int* or *float* in the normal way. How these variables are represented and operated upon is transparent to the user. Nevertheless, it would seem reasonable to use a similar mechanism in *DAMSON* to implement scaled fixed-point arithmetic, so that results from a PC emulation will have the same accuracy as results computed on an ARM processor. Therefore, the word size of variables and conventions for dealing with any scaled arithmetic should be consistent with the implementation on an ARM processor.

As the ARM processor does not have hardware floating-point, the following notation is proposed to represent floating-point arithmetic. Variables are represented as 32-bit variables with 16 bits for the signed integer part and 16 bits for the remainder, as shown in Figure A1.

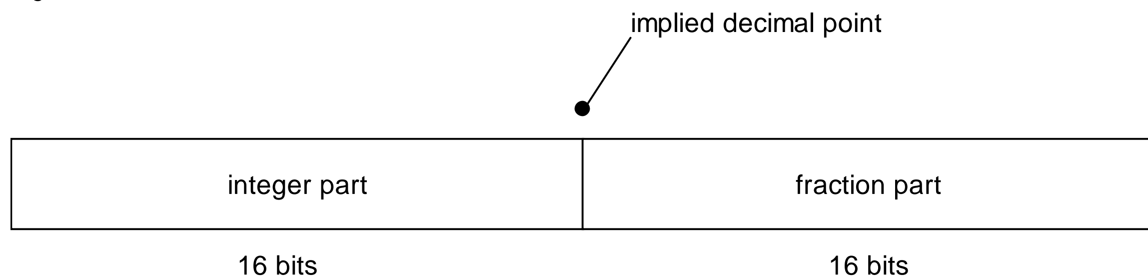


Figure A1 Scaled Fixed-point Arithmetic Notation

For example, the following scaled fixed-point numbers are represented in four bytes as follows:

```
2.0             00000000 00000010 00000000 00000000
-3.0            11111111 11111101 00000000 00000000
4000.0          11111010 00000000 00000000 00000000
0.25            00000000 00000000 01000000 00000000
-0.25           11111111 11111111 11000000 00000000
0.0084076       00000000 00000000 00000000 00000001
32767.0         01111111 11111111 00000000 00000000
```

In other words, the number range is -32768 to + 32767 and the resolution is approximately 8.4076×10^{-3}. It is assumed that, for the majority of applications appropriate to BIMPA, the range and accuracy implied by this notation will be sufficient. Using a notation of this form,

the addition and subtraction of 32-bit scaled fixed-point numbers is achieved using integer arithmetic, taking care to avoid overflow (numbers < -32768 or > 32767). Note that this 16:16 convention is just one representation of fixed-point arithmetic. It would be straightforward to extend this approach to 8:24 or 32:32 bit representation.

Care is needed with multiplication and division. Clearly, the result of any multiplication must be in the range given above, while minimising loss of accuracy in the computation. Note that the identities $1.0\times1.0=1.0$ and $1.0\div1.0=1.0$ must hold. Consider the multiplication 3.0×2.0. In integer arithmetic, $3.0 = 3\times2^{16}$ and $2.0 = 2\times2^{16}$ giving a result 6×2^{32}. To restore the value to its correct magnitude for scaled fixed-point arithmetic, the result must be divided by 2^{16}, which is easily achieved by shifting the result 16 places to the right (arithmetically).

The 32×32 multiplication instruction on the ARM processor produces a 64-bit result. Shifting the 64-bit result 16 places to the right and copying the low 32 bits of this register to a 32-bit register restores the magnitude of the result. Care is need for both overflow and underflow. In overflow, any bits set in the most significant 16 bits of the 64-bit register are lost and the remnant will not hold a valid result. Similarly, if two small fractions are multiplied, the 16 least significant bits are discarded during the shift.
The C code a = b * c is equivalent to the following generic assembler code:

```
MOVE b,r1
MOVE c,r2
MULT r1,r2
SHIFTR r1,#16
MOVE r2,a
```

where r1:r2 are treated as a single 64-bit register pair.

Note that there is no need to treat the sign independently; it is correctly computed during the instructions.

Division can also be accomplished in a similar way. Consider the division $6.0\div2.0$. In integer arithmetic, $6.0 = 6\times2^{16}$ and $2.0 = 2\times2^{16}$ giving a result 3.0×2^{0}. However, if the dividend is multiplied by 2^{16} prior to the division, the correct result 3.0×2^{16} is achieved. Again, the multiplication by 2^{16} is implemented as an arithmetic shift (left) where the ARM division instruction computes integer division of a 64-bit number by a 32-bit number. The C code for the expression a = b / c is equivalent to the following generic assembler code:

```
MOVE b,r1
SHIFTL r1,#16
DIV r1,c
MOVE r1,a
```

assuming that r1 holds the 32-bit result and r2 holds the 32-bit quotient after the division.

The clear advantage is that the equivalent of floating point operations are performed in 4 or 5 integer instructions. Moreover, the compiler knows the type of each variable and is able to convert integer values to scaled fixed-point equivalent values if implied in the computation. For operations on integer-only values or floating-point-only values, there is no need for manipulation of the variables.

In the emulation of *DAMSON*, the types of each operand are known and the appropriate integer arithmetic is invoked. For an implementation on the ARM processor, the code generator will generate the necessary integer arithmetic instructions to be executed directly by the processor, derived from the intermediate code used by the *DAMSON* stack machine architecture.

Appendix 2

Intermediate Code Format

The intermediate code generated by *DAMSON* meets the following objectives:

- It has a compact form.
- It is a simple format that can be easily emulated or interpreted
- It provides the basis for an optimising code generator

A simple machine is assumed with a program counter (PC) pointing to the next instruction to be executed, a stack pointer (SP) pointing to the most recently pushed item on the stack and a frame pointer (FP) pointing to the start of the frame on the stack holding the local arguments. The machine has a linear word-addressed 32-bit memory starting from address 0. The global variables of a node start from memory address 1 and the stack needed by a node is dynamically allocated from its memory.

Within each simulated node, one process is allocated to the procedure *main* (and its procedures) and new processes are activated by the arrival of an interrupt. Processes are terminated when their parent procedure returns (including *main*).

Arithmetic instructions operate on the items at the top of the stack and consequently, have no arguments. For example, the *DAMSON* code

```
a = b + c
```

is implemented by the following instructions:

```
push b
push c
add
pop a
```

The add instruction adds the top two items on the stack and leaves the result of the addition on the top of the stack, decrementing the stack pointer, as shown in Figure A2, which depicts the state of the stack after each instruction.

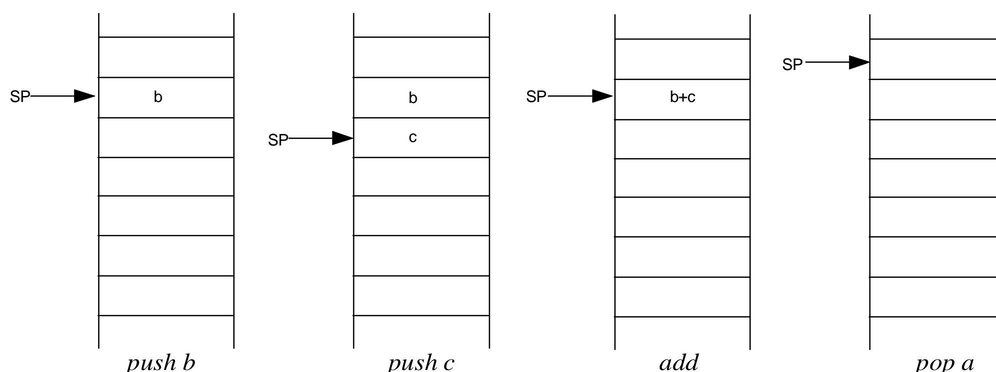


Figure A2 Execution of an Addition Instruction for a Stack Machine

Table 1 summarise the 42 intermediate code instructions used by the *DAMSON* compiler and emulator. Where no argument is defined, the instruction is simply a single byte.

Op code	Argument	Description
PUSHG	Memory address	Push a global variable onto the stack
PUSHL	Offset from FP	Push a local variable onto the stack
PUSHC	Constant value	Push a constant onto the stack
PUSHSTR	String address	Push a string address onto the stack
PUSHCLK		Push the local clock value onto the stack
PUSHFP		Push the frame pointer FP onto the stack
PUSHTOS		Push the top item on the stack onto the stack
PUSHI		Replace the top item on the stack with the contents it points to
POPG	Memory address	Pop a global variable from the stack
POPL	Offset from FP	Pop a local variable from the stack
POPI		Pop the item on the top of the stack with the address given by the adjacent item
SWAP		Swap the top two items on the stack
FLOAT		Convert the item on the top of the stack to a fixed-point integer
INT		Convert the item on the top of the stack to an integer
COMP		Compare the top two items on the stack, leaving the Boolean result on the stack
OR		Boolean OR of the top two items on the stack, leaving the result on the stack
AND		Boolean AND of the top two items on the stack, leaving the result on the stack
ADD		Integer addition of the top two items on the stack, leaving the result on the stack
SUB		Integer subtraction of the top two items on the stack, leaving the result on the stack
MULT		Integer multiplication of the top two items on the stack, leaving the result on the stack
MULTF		Fixed-point multiplication of the top two items on the stack, leaving the result on the stack
DIV		Integer division of the top two items on the stack, leaving the result on the stack
DIVF		Fixed-point division of the top two items on the stack, leaving the result on the stack
MOD		Integer modulo of the top two items on the stack, leaving the result on the stack
LOGICAND		Bitwise logic AND of the top two items on the stack, leaving the result on the stack
LOGICOR		Bitwise logic OR of the top two items on the stack, leaving the result on the stack
LOGICXOR		Bitwise logic XOR of the top two items on the stack, leaving the result on the stack
LSHIFT		Left shift of the top two items on the stack, leaving the result on the stack
RSHIFT		Right shift of the top two items on the stack, leaving the result on the stack
NEG		Integer negation of the top item on the stack, leaving the result on the stack
NOT		Boolean inversion of the top item on the stack, leaving the result on the stack
ONESCOMP		Bitwise logic inversion of the top item on the stack, leaving the result on the stack
JT	Label	jump to label if the top item of the stack is true, removing the item from the stack
JF	Label	jump to label if the top item of the stack is false, removing the item from the stack
JUMP	Label	jump to a label (unconditionally)
TEST	Label	Compare the top two items on the stack for equality, jump to label if equal
CALL	Label	Call the local procedure at label (return address automatically pushed onto the stack)
RETURN	Label	Return from the local procedure defined at label
ENTRY	Label	Set up the local frame from the pushed arguments
SYSCALL	Variable list	System call
VCOPY		memcpy defined by the top three items on the stack, which are popped from the stack
PLOT	Channel number	Plot the top item on the stack on the given channel (item is copied, not removed)

Table 1 Intermediate Code Instruction Format

For dyadic operations, where the order of the arguments is important (e.g. SUB, DIV and COMP instructions), the result is given by <next to top of stack item> - <top of stack item>, which is readily emulated at run-time (for a SUB operation) as follows:

```
x1 = stackpop()
x2 = stackpop()
stackpush(x2 – x1)
```

The use of this approach to compilation simplifies the code generation phase. The code generator attempts to keep the machine registers loaded with the items close to the stop of the stack, reducing the need for memory access addressing modes and maximising the speed of execution. The simplicity of the intermediate code format also enables the emulator to execute relatively fast.