Chrysalis 4.0 Technical Notes

BUTTERFLY PLUS

Chrysalis 4.0 Technical Notes

January, 1988

RELEASE LEVEL

This manual conforms to the Final Version of the Chrysalis 4.0 operating system software for the Butterfly™ Plus Parallel Processor released in January of 1988.

Contents

Chapter 3 Use of the Butterfly Network Software

Chapter 4 Using Assembler Language

Tables

Chapter 1

Introduction

This document is a collection of technical information on release 4.0 of the Chrysalis operating system. It provides information on the following topics:

- interprocess calls
- using the Butterfly network software
- using the assembler language on the Butterfly
- using the DBX68 debugger
- Butterfly event-logging facility
- using the streams remote file system library

Chapter 2

Interprocess Calls in Chrysalis

To complement the SUN remote procedure call mechanism for inter-host procedure calls, Chrysalis has an inter-process procedure call mechanism, called *service calls*. The implementation was inspired by a combination of the SUN remote procedure call mechanism and the *msgio* library, but the details are somewhat different (for example, since we can assume homogenous hardware, there is no XDR mechanism, although some type-checking is done on arguments and return values).

DATA STRUCTURES AND CALLING CONVENTIONS

The service call routines use an opaque (meaning, the internals are invisible to user programs) data structure to coordinate their activities. This data structure, SCStream, is analogous to the FILE data structure used by the *stdio* package. It contains a description of the service-call stream, as well as the data buffered on the stream.

Service call routines throw on errors in transmitting or packaging messages.

WRITING A SERVER

Writing a server with the Chrysalis service-call mechanism is quite simple. A server consists of a preamble to register the service with the system, a dispatcher routine which dispatches to the user's routines based on the received message, and the subroutines which the server implements and exports to the user.

The routines provided for writing servers (and the proper sequence for calling them) are:

- Register the service with the system using *sc_create_service*:

  QH *sc_create_service(*char **name)*

  Registers a service with the name *name*. *sc_create_service* returns a queue handle which is used in subsequent calls to the service-call package. Some system services (e.g., the name server and the remote daemon) have their queue handles stored in a global data structure maintained by Chrysalis rather than in the name table. They define their queue during startup, and use that queue when calling *sc_server*.

- Call the *sc_server* routine:

void *sc_server(*QH *requestq,* **void-procedure** *dispatch, arg-to-dispatch)*

  sc_server does not return. It is passed the queue handle returned by *sc_create_service* and the address of a routine to call when a message arrives. This routine serves to dispatch the incoming message to the proper subroutine, and looks like this:

  void *dispatch(***SCStream** **scstr, arg-to-dispatch,* **int** *selection)*

  arg-to-dispatch is an argument which may be passed to the dispatch routine. If you write your own dispatch routine, *arg-to-dispatch* is not interpreted by the service-call mechanism. *selection* is a user-defined constant used to select which subroutine is invoked by the message.

- *dispatch* consists of doing a using *selection* to decide which subroutine to call, and calling the user's routine using *sc_invoke*:

void *sc_invoke(***SCStream** **scstr,* **procedure** *selected_function)*

sc_invoke unpackages the arguments to the user's procedure, *selected_function*, and calls *selected_function* with the right arguments. The first argument passed to *selected_function* is the **SCStream** pointer.

- To simplify writing servers even more, a standard dispatch routine is provided.

sc_standard_dispatch expects the user's routines to be numbered sequentially, starting at 0. The selection is used as an index into a null-terminated table of function pointers, the address of which is *sc_standard_dispatch*'s second argument. Here is an example declaration of a table, as well as showing how to invoke *sc_server* to call *sc_standard_dispatch*:

```
int do_Routine0(), do_Routine1(), do_Routine2(),
          do_Routine3(), do_Routine4(),
       do_Routine5(), do_Routine6(), do_Routine7();
int dispatch_table[] = {
    do_Routine0,
    do_Routine1,
    do_Routine2,
    do_Routine3,
    do_Routine4,
    do_Routine5,
    do_Routine6,
    do_Routine7,
    NULL,
};

 ...
extern  sc_standard_dispatch();
 ...
sc_server(service_queue, sc_standard_dispatch, dispatch_table);
 ...
```

- values are returned to the client program using:

  *sc_send_return_value(***SCStream** **scstr, arglist...)*

  This is a subroutine call, and calling it doesn't result in the server routine returning (that is, the server routine should contain its own return. *sc_send_return_value* takes an argument list similar to *sc_call* (see the section on writing a client), alternating *sc_type* specifications and pointers to values, terminated by a NULL *sc_type*.

  Should a server get a throw when processing a client's request, the *throwcode, throwvalue,* and *throwstring* are returned to the client. The service-call mechanism at the client end will then perform the throw. The throwstring has the string ''(SERVER):'' prefixed to it, to make it clear that the throw took place on the server end of the connection.

- Should a server need to exit, it should call *sc_delete_service*:

  *sc_delete_service(***char** **name,* **int** *exit_value)*

  This routine will notify the system that the service with name *name* is no longer offered, and, if *exit_value* is non-negative, will call *exit* with *exit_value* as an argument. If *exit_value* is negative (typically, -1), *sc_delete_service* will not call *exit*, but will return, instead.

- *throws* which occur while the server is executing a service call are transparently forwarded to the client if uncaught within the body of the user's routine or dispatch routine.

The subroutines exported by a server all look like normal subroutines, as they would be written in a user program, except that the first argument to the subroutine is a pointer to an **SCStream**, and, instead of calling *return*, the server routines call *sc_send_return_value*. For example:

```
    do_Routine1(scstr, str, number)
```

```
        SCStream *scstr;
        char *str;
        int number;
    {
        initial processing, calculating a result for the client...
        sc_send_return_value(scstr, SC_int, &number, RETURNEND);
        any further processing (the results of which aren't reported
            to the client)...
            (NOTE: the memory allocated for the string str gets
            unmapped by sc_send_return_value, so its contents
            are no longer accessible.)
    }
```

As is shown, results (perhaps a simple acknowledgement of receipt of the client's request) may be returned to the client before the service subroutine completes, allowing processing to continue in parallel. If you use such an approach, arguments which were passed to the routine on the stack (integers, characters, shorts, and doubles) are still available after one of the return-sending routines is called. Arguments which are passed to the routine as pointers on the stack (strings and buffers) point to storage which is freed by *sc_send_return_value*. In order to use string or buffer arguments after you send a return, you must copy the data to memory you manage yourself.

Note that nothing in the definition of the server routines restricts the server to a single process. Once the service's queue handle is registered with the system, any number of processes can execute *sc_server* in parallel, since a message arrives in the form of a single datum dequeued off of the service queue. Thus, a server need not worry about accessing the service-call data structures atomically, since atomicity is built in *on the server side*.

A simple example of a server is contained in a later section in this document.

WRITING A CLIENT

Finding the Server, Allocating Service Call Resources, and Freeing Them

In order to find a service, one can call *sc_open* to get an **SCStream** * for use in accessing the service:

> **SCStream** **sc_open(*char* *name)*

sc_open takes the name of the service being sought, and returns an **SCstream** pointer to use in calls to the service.

Note: SCStream structures aren't accessed atomically, and use *malloc* to allocate space for data, so it is important that each process or thread that a user's program has which wishes to use this service performs its own *sc_open* on it.

An **SCStream** * obtained using *sc_open* can be gotten rid of using *sc_close:*

> *sc_close(***SCStream** **scstr)*

sc_close deletes the data structures allocated by *sc_open*, and performs other clean-up tasks.

Of course, kernel routines (which also rely on the service-call mechanism) cannot use *malloc*, and make such frequent use of the services in question that they don't want to look them up by name all the time. Therefore, for processes which already know the queue handle the desired service is using, there exists another call:

> *sc_bind(***SCStream** **scstr,* **QH** *service_queue,* **long** **reply_queue_storage,* **int** *reply_queue_length)*

sc_bind takes a pointer to an **SCStream** allocated by the calling routine and packages it to connect to a service listening on *service_queue*.

Service connections are implemented using two dual queues, one listened to by the server (*service_queue*) and one listened to by the client waiting for return values. *Make_DualQ* is called with *reply_queue_storage* and its length, *reply_queue_length* to make the dual queue to which return values are sent.

As is clear from the invocation sequence, *sc_bind* uses a dual queue to transmit its return value messages. This is necessary if there is a possibility that multiple return value messages could be posted to the procedure call (as can be the case in asynchronous calls, see below). As creation of a dual queue is a moderately expensive operation, there is an additional call, which is used extensively in the kernel:

> *sc_bind_using_event(*SCStream **scstr,* QH *service_queue)*

This call will have messages containing return values posted as an event (using the *CurPCB->p_teh* event the system keeps lying around for such occasions), saving a good deal of time in binding the queue.

An SCStream * obtained using *sc_bind* or *sc_bind_using_event* can be gotten rid of using *sc_unbind:*

> *sc_unbind(*SCStream **scstr)*

Calling on the Server

Service calls are made using the *sc_call* routine:

> *sc_call(*SCStream **scstr,* **int** *routine_selector, arg-list..., ARGEND, retval-list..., RETURNEND)*

routine_selector is a user-defined flag which is used to select which routine is to be called (basically, *routine_selector* is passed to the server's dispatch routine, which calls the appropriate service routine). *arg-list* and *retval-list* are argument and return value lists, respectively. The lists are composed of alternating *sc_type* specifiers and pointers to values of the appropriate type, e.g.:

> **void** *sc_call(scstr, READ, SC_int, &fd, SC_buffer, &ptr_to_buffer, bufferlen, SC_int, &bufferlen, ARGEND, SC_int, &nread, RETURNEND);*

Which illustrates a routine written to imitate a UNIX-style read, where an integer file descriptor, a buffer and a buffer length is passed to the server, and the number of bytes actually read is returned in *nread. ARGEND* and *RETURNEND* are both defined as NULL, but the names are supplied to make one's code clearer and easier to read.

If the return list isn't empty, *sc_call* blocks waiting for the server to send return values. A call which doesn't require a return value need not block (of course, in this case, there is no guarantee that the server actually processes the message, since there is no acknowledgement returned of receipt of the message and there is no way to know that a call has thrown in the server).

If it is expecting a return value, *sc_call* will wait forever for a server to reply. If the server is hung or in an infinite loop, the client will also hang. This is, of course, the same as for any normal procedure call, however it can lead to programs hanging mysteriously. When one can predict an upper-bound for the time a routine will take, one can build a more robust application using *sc_call_with_timeout* to detect hung or looping servers:

> *sc_call_with_timeout(*SCStream **scstr,* **int** *timeout,* **int** *routine, arg-list...)*

The timeout is specified in seconds. If the server doesn't respond within *timeout* seconds, a throw occurs. A user program can detect this throw, and at least can print a message that helps you isolate the probelm with your program.

In addition, if you're going to be communicating with a server once, for a single call, you probably don't want to go through the overhead of doing a sequence of *sc_open*/*sc_call*/*sc_close* or *sc_bind*/*sc_call*/*sc_unbind*, which were designed for lengthy dialogs between a user program and a server. To facilitate a ''datagram'' or ''one-shot'' style of communicating with a server, there is *another* way to call on a server:

> *sc_call_using_queue(***QH** *servers_queue,* **int** *timeout,* **int** *routine, args...)*

This routine, used within the kernel to call on kernel services, combines the three steps (allocating resources for the service call, making the service call, and freeing the allocated resources) into one.

Using Asynchronous Procedure Calls

One may also perform asynchronous service calls, using *sc_call_start* and *sc_call_finish*.

> *sc_call_start(***SCStream** **scstr,* **EH** *event,* **int** *routine_selector, arg-list...)*

This routine sends the arguments to the server. The client process can then continue to run in parallel with the server's processing the call. When the server sends the return values, it also posts the event *event* if it's non-null. If one uses a separate event for each asynchronous call, one may have several asynchronous calls outstanding (either to the same server, or to several different servers). If this is done, care should be taken to read the return values in the order that the events are posted, otherwise replies will be lost.

Once the event has been posted, the return value can be retrieved using *sc_call_finish*:

> *sc_call_finish(***SCStream** **scstr, retval-list...)*

If you use asynchronous calls, you must have used *sc_open* or *sc_bind* to obtain a service call stream using a dual queue. Service-call streams constructed with *sc_bind_using_events* cannot be used with these asynchronous calls. An attempt to do so will result in a *CONSISTENCY* throw, with the warning that asynchronous calls require a reply queue.

DATA TYPES

The routine-calling and return value sending routines both take lists of *sc_types*. Generally one must pass a *pointer to a value* to send the value, or a pointer to the place to put the value when receiving a value. This is analogous to SUN's XDR routines -- the same routine is used to send an integer as is used to receive an integer.

sc_types, and the way to pass them, are:

> *SC_int, &((int) datum)* -- send/receive the 32-bit integer, *datum.*
>
> *SC_short, &((short) datum)* -- send/receive the 16-bit integer, *datum.*
>
> *SC_char, &((char) datum)* -- send/receive the 8-bit character, *datum.*
>
> *SC_double, &((double) datum)* -- send/receive the 64-bit floating point number, *datum.*
>
> *SC_string, &((char *)datum)* -- send/receive the null-terminated string *pointed to* by **datum. Note: datum* is a pointer to *the address of* the string being sent or received, *not* a pointer to the string (see discussion below).
>
> *SC_buffer, &((char *)datum), bufferlen* -- send/receive the buffer pointed to by **datum.* In addition to a pointer *to the address* of the buffer (**not** *a pointer to the buffer), the length of the buffer is passed (see discussion below).*

The C data type, **long**, and the Chrysalis data types, **QH, OID, bits** and **EH,** are also represented (as puns on **int**).

Why Buffers and Strings Must Be Passed Using Double-Indirection

chars, **shorts**, and **ints** all may be returned on the stack. Strings and buffers, being arbitrarily long, aren't easily returned on the stack (and if they were, you would have to copy them somewhere before they are overwritten by calling the next subroutine), so space must be allocated for them somewhere else. If **datum* is NULL, space will be allocated for the string or buffer using *malloc*, and a pointer to the allocated space will be returned in *datum* (if **datum* is non-null, then the user has provided space for the buffer to be received (and is presumed prepared to take the consequences if there isn't enough room for the buffer in the provided space)). Therefore, instead of a pointer to the string, a pointer to a pointer to the string must be passed when receiving strings, and to be consistent, a pointer to a pointer must be passed when sending a string or a buffer.

The use of pointers to buffers or strings, instead of pointers to pointers to buffers or strings, is the single most prevalent programming error in using service calls. Here is an exammple of a correctly-formatted call:

```
char **strp;
char *str= "hello,there";
strp=&str;
sc_call(scstr, sc_string, strp, ARGEND, RETURNED);
```

A SAMPLE CLIENT AND SERVER

We present here an extended example of a client and a server, illustrating many of the variations for using the service-call mechanism. This example is taken from a pair of programs written to test the different features of the service-call mechanism, and contains examples of almost all the varieties of calls.

Another example of use of the service call mechanism may be found in the Chrysalis sources: the name daemon and its routines (*Name_Bind, etc.*) are implemented using service calls, as is the interface (*Load_Process*) with the loader daemon.

example.h:

First, we define the subroutine selections and some shared data structures in a header file to be included by users of the *example* service:

```
/* Routine name/index:            types of arguments and return-values: */
#define Routine1 1                /* string, int, returns int */
#define Routine2 2                /* struct s, no return */
#define Routine3 3                /* no argument, string return */
#define Routine4 4                /* no argument, tests asynch. calls */
#define SERVICE "example"            /* This service is named ``Example'' */

struct s {             /* A structure to test passing buffers */
   int a;
   short b;
   short c;
   char d[10];
};
```

example_client.c

This example is just one long routine, which calls several example routines to test various features of the service call mechanisms.

```
#include <stdio.h>
#include <sc_public.h>
#include "example.h"

main() {
   SCStream *strp;

   /*
   * Get a handle for the service.
```

```
  */
   strp = sc_open(SERVICE);

   /*
  * We do an infinite loop to stress resource allocation.  Run forever,
     or until we run out of some resource (F8 space, SARs, etc.).
  * Running out of something means the service call code isn't freeing data appropriately.
     while it's running you can keep an
  * eye on resources allocated using show and showproc to detect memory leaks.
  */
   while(true) { say_hello(strp); Sleep(1000); }
}

/*
 * Call most of the different varieties of routines possible, testing different combinations of
     argument types, and reply types.
 *
 * This program prints out messages marked by asterisks when something goes wrong.
     A message without asterisks indicates normal operation
 * or a normal status message.
 */
say_hello(strp)
   SCStream *strp;
{
   char *str = "hello, there";
   int number = 23;
   int result = 0;
   struct s s;
   struct s *sptr;
   static EH event = NULL;
   EH got_event;
   /*
  * Call Routinel, sending a string, an int, and returning an int. NOTE: ``&str'' is a char **!, not just a
    */
   catch
     printf("Calling Routinel(
     sc_call(strp, Routinel, SC_string, &str, SC_int, &number, ARGEND,
         SC_int, &result, RETURNEND);
     if(result == 92) printf("Win!  Result is %d0, result);
     else printf("0***What? result is %d0, result);
   onthrow
     when(true)
       printf("0***Routinel Lose!  %s(%X)@%X0, throwtext, throwvalue,
          throwlocation);
```

```
endcatch;

/*
* Call routine2: this also tests whether or not the sc_call waits for anything, or returns
 immediately after enqueuing the message.
*
* How to pass a structure to a server:
*/
s.a = -3;
s.b = -6;
s.c = -9;
sptr = &s;             /* Because we have to use a pointer to a pointer */
strcpy(s.d, "Hola!");
catch
  printf("Calling VOID Routine2(s{a:-3, b:-6, c:-9, d:
  sc_call(strp, Routine2, SC_buffer, &sptr, sizeof(s), SC_int, &numbe
      ARGEND, RETURNEND);
  printf("Win!0);
onthrow
  when(true)
    printf("0***Routine2 Lose!  %s(%X)@%X0, throwtext, throwvalue,
       throwlocation);
endcatch;

/*
* Call routine3: this tests routines with no arguments, and also tests returning a string value.
*/
str = NULL;            /* Want to test malloc. */
catch
  printf("Calling Routine3() should return
  sc_call(strp, Routine3, ARGEND, SC_string, &str, RETURNEND);
  printf("Win! Returned
  /*
  * NOTE: strings are returned as pointers to malloced areas.  THEREFORE, we have to
  *free the malloced thing, while we still have a pointer to it.
  */
  free(str);
onthrow
  when(true)
    printf("0***Routine3 Lose!  %s(%X)@%X0, throwtext, throwvalue,
       throwlocation);
endcatch;
```

```
    /*
    * Testing asynchronous calls.
    *
    * Make an event for this routine to tell us there are arguments waiting.
    */
    if(event == NULL) event = Make_Event(0, 0, RW_rw_, 0);
    catch
      printf("Calling Routine4(); ");
      sc_call_start(strp, event, Routine4, ARGEND);
      printf("waiting for response....");
      /*
      * Wait for the reply to arrive.
      */
      got_event = Wait();
      if(got_event == event) {
     char buffer[10];
     str = &buffer[0];  /* Provide pre-allocated space -- again, note the use of a pointer to a poi
     catch
       printf("Collecting response (should be
       sc_call_finish(strp, SC_string, &str, RETURNEND);
       printf("Routine4 Win! Returned
     onthrow
       when(true)
         printf("0***Routine4 lose in sc_call_finish! %s(%X)@%X0,
             throwtext, throwvalue, throwlocation);
     endcatch;
     Reset_Event(event);
      } else printf("0***Routine4 Lose! got wrong event!0);
    onthrow
      when(true) printf("0***Routine4 Lose! %s(%X)@%X0, throwtext,
               throwvalue, throwlocation);
    endcatch;
}
```

example_service.c:

A server is simply a loop listening for messages on the service queue. Since operations on a queue are atomic, a server could be implemented as many processes, each waiting to dequeue a request from the service queue.

```
#include <stdio.h>
#include <sc_public.h>
#include "example.h"

char *progname;

int do_Routine0(), do_Routine1(), do_Routine2(), do_Routine3(), do_Routi

int sc_standard_dispatch();

int dispatch_table[] = {
     do_Routine0,
     do_Routine1,
     do_Routine2,
     do_Routine3,
     do_Routine4,
     NULL,
};

main(argc, argv)
   int argc;
   char ** argv;
{
   QH sq;
   progname = argv[0];

   sq = sc_create_service(SERVICE);    /* register service with system */
   sc_server(sq, sc_standard_dispatch, dispatch_table);  /* run the service. */
}
```

```
/*
 * A non-entity because the table begins at 0, but the routine defs start at 1 (in studying
 * the anatomy of the software that has evolved, occasionally we find a vermiform appendix).
 */
do_Routine0(){}

/*
 * sc_invoke will pass the arguments on the stack just as they would be packaged by the client routine,
 * if we didn't have the baroque calling sequence with argument types.
 *
 * The arguments to Routine1 are a string followed by an integer. Note the natural handling of strings
 * in arguments (in contrast to the double-indirection forced on the caller).
 * Note also that integer arguments are direct, rather than indirect through a pointer.
 *
 * Returns an integer.
 */
do_Routine1(scstr, str, number)
   SCStream *scstr;
   char *str;
   int number;
{
   printf("Routine1:
   number *= 4;
   sc_send_return_value(scstr, SC_int, &number, RETURNEND);
}

/*
 * This routine has no return value. It tests passing a structure to a server.
 */
do_Routine2(scstr, str, number)
   SCStream *scstr;
   struct s *str;
   int number;
{
   printf("Routine2: number: %d, s.a: %d, s.b: %d, s.c: %d, s.d:
      number, str->a, str->b, str->c, str->d);
}

/*
 * This routine has no arguments, but does return a string. Note that when we're returning a string,
 * its back to double-indirection (&str, where str is a char *).
 */
do_Routine3(scstr)
   SCStream *scstr;
```

```
{
   char *str = "Hiya!";
   printf("Routine30);
   sc_send_return_value(scstr, SC_string, &str, RETURNEND);
}

/*
 * This routine tests asynchronous calls.  No arguments, but it returns a string (after waiting a perceptib
 */
do_Routine4(scstr)
   SCStream *scstr;
{
   char *str = "Hoya!";
   printf("Routine40);
   Sleep(5000);
   printf("Sending result0);
   sc_send_return_value(scstr, SC_string, &str, RETURNEND);
}
```

As noted in the discussion of *sc_server*, a standard dispatcher is provided. The example above makes use of it. Here is an illustration of a server defining its own dispatcher.

```
int do_Routine0(), do_Routine1(), do_Routine2(), do_Routine3(), do_Routine4();

/*
 * Given the constant which selects the routine, invoke the routine for the user.
 * This is presented as pedagogical proof that one can write one's own dispatch routir
 * dispatch the request to one of several tasks).
 *
 * Also illustrates the use of the sc_invoke routine, which unpackages the arguments se
 * the stack so server routines are written naturally.
 */
dispatch(scstr, ignored_argument, selection)
   SCStream *scstr;
   int selection;
{
   switch(selection) {
    case Routine0:
      sc_invoke(scstr, do_Routine0);
    case Routine1:
      sc_invoke(scstr, do_Routine1);
      break;
    case Routine2:
      sc_invoke(scstr, do_Routine2);
      break;
    case Routine3:
      sc_invoke(scstr, do_Routine3);
      break;
    case Routine4:
      sc_invoke(scstr, do_Routine4);
      break;
    default:
      printf("Got trash: claimed routine is %x\n", selection);
      break;
    }
}

main(argc, argv)
  int argc;
  char ** argv;
{
  QH sq;
  progname = argv[0];

  sq = sc_create_service(SERVICE);   /* register service with system */
  sc_server(sq, dispatch, NULL);  /* run the service. */
}

do_Routine1(scstr, ....)
```

CALL SUMMARY

Arglists and Supported Data Types

Arguments are passed in the call and return values returned from a call as a list of values representing the type of the argument followed by a pointer to the value of the argument. The exceptions to this are: strings, which are passed as a flag denoting a string argument, followed by a pointer to a string *pointer*; and arbitrary buffers of bytes, which are passed as a flag denoting a buffer, followed by a pointer to a buffer pointer, followed by the length, in bytes of the buffer. The list is terminated by a NULL appearing in a type specifier's place. For example:

SC_int, &intarg, SC_short, &shortarg, SC_buffer, &buffer_pointer, bufferlen, NULL

There are two defines, *ARGEND* and *RETURNEND* which may be used in place of *NULL* to terminate a list of arguments. Lists of this type will be referred to below as *arglists*.

Types (and how they are used) are:

SC_int, &((int) intarg),
SC_short &((short) shortarg),
SC_char &((char) chararg),
SC_double &((double) floatarg),
SC_string &((char *) string_pointer),
SC_buffer &((char *) buffer_pointer), buffer_length,

The C data type, **long**, and the Chrysalis data types, **QH**, **OID**, **bits** and **EH**, are also represented (as puns on **int**).

Calls Used by the Server

QH *sc_create_service(*char **name)*

Tells the system about the existence of a service named *name*. Returns a queue handle to be used in a following *sc_server* call.

void *sc_delete_service(***char** **name,* **int** *exit_value)*

Deletes a service created with *sc_create_service*, called by a server before it exits.

void *sc_server(*QH *requestq,* **int***dispatcher*, **bits** arg*)*

Does not return. Contains a loop which calls *dispatcher(***SCStream** **s, arg, int* selection_ from_user_request_message) repeatedly when service requests arrive on the *requestq* queue. Usually called using *sc_standard_dispatch*, in which case *arg* is a pointer to a null-terminated table of pointers to routines:

```
int do_Routine0(), do_Routine1(), do_Routine2(), do_Routine3(), do_R
int dispatch_table[] = {
    do_Routine0,
    do_Routine1,
    do_Routine2,
    do_Routine3,
    do_Routine4,
    NULL,
};
 ...
extern  sc_standard_dispatch();
 ...
sc_server(service_queue, sc_standard_dispatch, dispatch_table);
 ...
```

sc_standard_dispatch itself is never called directly by someone writing a server.

void *sc_send_return_value(*SCStream **s, arglist...)*

Sends a list of return values on the *SCStream* at *s*.

void *dispatch(*SCStream **s,* bits *arg_from_sc_server,* int *selection)*

dispatch is a user-provided routine (if the user intends to provide a dispatch routine other than the standard one.

void *service_routine(scstr, a, b, c)*
SCStream **scstr;*
int *a;*
char **b;*

short *c;*

This is an example of a service routine provided by a person writing a server. The first argument to a routine invoked by the service call mechanism is a pointer to an **SCStream**. The remaining arguments are standard arguments (*not* an *arglist*).

Calls Used by Client Programs

SCStream * *sc_open(*char **name)*

Finds a service with the name at *name*, allocates (using *malloc*) storage for an **SCStream** structure and creates a dual queue for use by the server in sending replies to the client, and initializes the **SCStream** to communicate with the service.

SCStream **sc_open_with_resources(*char **name,* SCStream **scstr,* long **reply_queue_storage,* int *queue_len)*

This routine is the equivalent of *sc_open*, except that it does not allocate any resources for the **SCStream**. It is used by the *msgio* library, which allocates the **SCStream** and other resources itself.

void *sc_close(*SCStream **scstr)*

Used to free the resources allocated by *sc_open* or *sc_open_with_resources.*

void *sc_bind(*SCStream **s,* QH *server_queue,* char **reply_queue_storage,* int *reply_queue_length_in_bytes)*

Used when a client knows what queue handle to use to communicate with a server, the calling program provides storage for an SCStream, and storage for a queue to use for the server to send replies to.

void *sc_bind_using_event(*SCStream **s,* QH *server_queue)*

Used when a client knows what queue handle to use to communicate with a server, *and* wishes to avoid the overhead of creating a dual-queue for replies. Can be used only when the connection isn't going to be used for asynchronous calls.

void *sc_unbind(*SCStream **s)*

Used to free the resources allocated by an *sc_bind* or *sc_bind_using_event* call in a long-lived process. These resources are automatically released when a process exits.

void *sc_call(*SCStream **s,* **int** *routine_selector, arglist, arglist)*

Calls service routine selected by *routine_selector*, an integer defined in a header file for the service (the routine selector is usually an index into a table of routines). The first *arglist* (terminated by *ARGEND*) is the list of arguments to the subroutine. The second *arglist* (terminated by *RETURNEND*) is the list of return values expected from the routine.

void *sc_call_with_timeout(*SCStream **s,* **int** *timeout,* **int** *routine_selector, arglist, arglist)*

Calls service routine selected by *routine_selector*. If the server doesn't respond within *timeout* seconds, this call will throw, with a *FAILED* throwcode. The first *arglist* (terminated by *ARGEND*) is the list of arguments to the subroutine. The second *arglist* (terminated by *RETURNEND*) is the list of return values expected from the routine.

void *sc_call_using_queue(*QH *requestq,* **int** *timeout,* **int** *routine, arglist, arglist)*

Like *sc_call_with_timeout*, but performs a subroutine call on a server with a minimum of overhead, provided that you know the queue on which the server is listening for requests. This call is used within the kernel to communicate with kernel services such as the name daemon and the loader daemon. *sc_call_using_queue* is the equivalent of calling *sc_bind_using_event, sc_call_with_timeout,* and *sc_unbind* in sequence. The first *arglist* (terminated by *ARGEND*) is the list of arguments to the subroutine. The second *arglist* (terminated by *RETURNEND*) is the list of return values expected from the routine.

void *sc_call_start(*SCStream **scstr,* EH *event,* **int** *routine, arglist)*

Used for asynchronous calls. *sc_call_start* returns immediately after sending the subroutine request to the server. When the server sends a reply to the request, it will also post *event*. You can have multiple outstanding asynchronous calls, but care must be taken to read the returns of the calls in the order in which they are posted, or results will be discarded.

void *sc_call_finish(*SCStream **scstr, arglist)*

Reads the return values of an asynchronous call started with *sc_call_start.*

Chapter 3

Use of the Butterfly Network Software

The network figures prominently in the use of the BBN Butterfly parallel processor. This chapter gives an overview of the Chrysalis network software.

ORGANIZATION OF THIS CHAPTER

The second section introduces the design concepts of the Butterfly network software, and also some of the terminology and acronyms used in discussing the network software. Section three provides examples that explain the use of the network routines in writing programs. Section four describes each of the components of the network software. The concluding section explains how to bring up the network software on your machine.

NETWORK PROTOCOLS: IP, UDP, RDP, TCP

The Butterfly network software implements the DOD standard Internet Protocol (IP). The IP layer provides a common internetwork addressing scheme for higher level protocols, such as the User Datagram Protocol (UDP), the Reliable Datagram Protocol (RDP) or the Transmission Control Protocol (TCP). Below the IP layer is the Ethernet layer.

UDP, RDP, and TCP sit in parallel on top of IP, and provide different types of service. UDP provides an "unreliable", low effort, low overhead datagram service. The unit of transmission in UDP is the individual packet or datagram. UDP users must implement their own means of packet delivery and packet sequencing. In UDP packets may be delivered out of sequence, delivered more than once, or may be dropped by one of the intermediate layers, with no notification provided to the user of a problem. RDP provides a reliable datagram service. TCP provides reliable, sequenced byte-streams to the user.

Other protocols (e.g., the File Transfer Protocol (FTP), the Network Terminal Protocol (TELNET), user-written applications) are layered on top of TCP, RDP, or UDP.

When a user writes with a UDP channel, the network server prefixes the user's data on a UDP header, which contains the length of the message, the destination port identifier, and the source port identifier. This prefixed packet is in turn handed to the IP layer, which adds a similar prefix. The IP layer then hands the packet to the Ethernet device driver, which prefixes a header that tells the Ethernet hardware how to deliver the packet to its destination.

TCP works similarly but, by providing sequencing information in its header, is able to present a reliable byte-stream abstraction.

Why so many layers? The IP layer is separated from the Ethernet layer because the final destination *may not be on this Ethernet.* Networks are joined by gateways, and the IP header vouchsafes the packet through gateways through one or more networks. The user rarely works directly with the IP layer. UDP, RDP, and TCP all sit in parallel on top of IP, and provide different kinds of services to the user.

More information about the whys and wherefores of the DOD network implementation can be obtained from the Network Information Center (NIC):

Network Information Center

SRI International
Menlo Park, California 94025 (NIC@NIC.ARPA)
(415)859-3695

SOCKETS

Like the Berkeley 4.2bsd UNIX system, the Butterfly network software is organized around *sockets.* A socket is a channel through which one communicates with the network software, and, in turn, the network.

There are some differences between a Berkeley UNIX socket and a Butterfly socket. The underlying abstraction for I/O on the Butterfly computer is a structure similar to the UNIX Standard-I/O library FILE structure (and similarly named). Also, a socket handle is a pointer on the Butterfly, instead of being a small integer as in 4.2bsd UNIX. Chrysalis FILEs are one-way (read or write) due to the buffering which takes place using FILEs. So in order to both read and write from a Butterfly socket you must make a copy of the socket once it is open. The Butterfly socket does not provide the *select* call, but it *does* have true asynchronous calls on the network (*accept_start, accept_finish, read_start, read_finish, recv_start, recv_next, send_start,* and *send_next* along with the Butterfly's event mechanism).

Except for the above considerations, writing a network-using program on the Butterfly is very similar to writing a network-using program on a 4.2bsd UNIX machine such as the SUN workstation.

WRITING NETWORK APPLICATIONS: A TCP EXAMPLE

TCP is the protocol of choice for most users who are just starting to write network applications. (Those who want to use UDP are expected to have more expertise.)

Network connections come in two halves--the *server* and the *user.* The distinction is that the server creates a socket and then *listens* passively (possibly for many connections from different hosts), while a user creates a socket and then actively *connects* (and can connect to only one host using the socket).

Server Program Example

A server schema looks like this (the catch blocks are left out of this example to focus layout attention on the network calls):

```
#include <public.h>
#include <stdio.h>
#include <net/types.h>
#include <net/in_addr.h>
#include <net/protonum.h>

#define MY_PORT_NUMBER 4321   /* arbitrarily chosen port
number */

FILE *socket(), *accept(), *MSGdup();
main() {
    FILE *socket_for_listening;
    FILE *socket_for_reading;
    FILE *socket_for_writing;

Server Program Steps:
    1) socket_for_listening = socket(TCPROTO, "r");
    2) bind(socket_for_listening, 0, MY_PORT_NUMBER, 0);
    3 listen(socket_for_listening, MAXIMUM_NUMBER_OF_LISTENERS);
    4) socket_for_reading = accept(socket_for_listening, NULL, NULL);
    /*
     * socket_for_reading may now be used as a file descriptor for
     * reading. In order to write on the network connection,
     * one must:
     */
    socket_for_writing = MSGdup(socket_for_reading, "w")
    . . .
```

Server Program Steps

1. A server program begins by obtaining from the network system a protocol socket on which to make network requests (*socket(TCPROTO,*

"r")). In this example, the server has requested that the network connection use TCP (*TCPROTO*).

2. The server program then gives the socket a name (*bind(socket_for_listening, 0, MY_PORT_NUMBER, 0)*). A "name" on the network consists of two parts: a 32-bit host address and a 16-bit number known as a *port.* In the case of the server, the host address is the same as the machine the server is running on, so the address argument to the *bind* call is left as 0, indicating that the network system can fill in this field with an appropriate value.

   Note that you can choose your port number according to the service you are providing. Certain port numbers (those below 1000) are reserved for network-wide functions. Also, Berkeley 4.2bsd UNIX systems will only allow the root to create a socket with a port number less than 1024, so choosing such a port number is a bad idea if you are debugging a program.

   One place to look for currently reserved port numbers is in the /etc/services file on your 4.2bsd machine. Another place to consult is the Network Information Center (NIC) at SRI (NIC@NIC.ARPA), or Jon Postel at USC-ISI (Postel@ISI.ARPA). The NIC maintains a list (updated periodically) of reserved network port numbers. Dr. Postel is in charge of allocating Internet resources such as reserved port numbers.

   If you are providing an entirely new kind of service, you are fairly safe from conflicts if you choose a random 16-bit number greater than 1000. All that really matters is that everyone who wants to talk to your service knows what port number to use, and that no one else is likely to use the port number for some other service.

3. Having given the socket a name, the server must *listen* for a process to actively connect to the socket. *listen* takes a second argument to specify the number of unaccepted connections the system should maintain. This is generally set to 5, signalling that socket initialization is complete and permitting adverisement of service.

4. Following the *listen*, the server does an *accept*. Here a server is able to specify from which hosts or ports on hosts it will accept a message; once *accept* is executed the system rejects attempts to connect by other hosts. The server in the example will accept connections from anyone. *accept* returns a FILE pointer which may be used in subsequent *reads* and *writes*, *fprintfs* and *fscanfs*.

As noted above, Butterfly FILE structures are one-way. To be able to both read and write a socket, one must use *MSGdup* on it to get a FILE structure to go the other way.

User Program Example

The schema for a user program--one which actively establishes a connection--looks like this:

```
#include <public.h>
#include <stdio.h>
#include <net/types.h>
#include <net/in_addr.h>
#include <net/protonum.h>

#define MY_PORT_NUMBER 4321 /* arbitrarily chosen port number */

FILE *socket(), *MSGdup();
main(argc, argv)
int argc;
char **argv;
{
    FILE *s;
    struct in_addr dstaddr;
    struct in_addr *atoIPa();
    s = socket(TCPROTO, "w");
    skip for client
     * Server willing to specify port and address
    dstaddr = *atoIPa(argv[1]);
    connect(s, dstaddr, MY_PORT_NUMBER);
    /*
     * At this point, the socket is a FILE pointer and may be
     * used for writing (have to do a MSGdup to use it for reading).
```

```
*/
```

User Program Step

Again, we create a socket and give it a name. The name we give it is not important, because no one will ever use it (in contrast, we must use the server's name in order to connect to it). The routine *atoipa* is in *libnet.a* and translates an ASCII representation of a host address, e.g., "128.11.7.1" into a *struct in_addr*, and returns a pointer to the result. (Here I've used structure assignment to perform the actual assignment of the structure elements.) After finding the host address to which we want to connect, we use the *connect* call to tell the system the name of the network connection we want to use.

COMPONENTS

The programs providing network service on the Butterfly computer fall into four major categories. These categories are:

Device Drivers

The device drivers talk directly to the hardware or perform very low level functions.

Protocol Demons

The protocol demons implement the TCP, RDP, and UDP protocols, as well as the associated IP functions such as routing table maintenance.

Network Utilities

The network utilities are tools to configure and manage other network software.

Network Servers

The network servers provide higher level services such as virtual terminal emulation and file transfer.

Libraries

Finally, the libraries provide user programs with routines to access and control the rest of the network software.

DEVICE DRIVERS AND RELATED PROGRAMS

initmb	Initialize Multibus Adapter. This program sets up the parameters of the Butterfly Multibus Adapter card so that the software or the node can interract with devices on the Multibus. *initmb* must be run on the node to which the Multibus adapter is connected.
initnet	Initialize network data strucutres. *initnet* constructs a named memory obect to hold network process IDs and certain dual queues that take user requests. It is a rendezvous point. *initnet* must also be run on the node to which the Multibus adapter is connected.
startex	This parameter checking program is a preprocessor to the *excelan* driver. By using *startex* to start the excelan driver, you discard all the code used for parameter checking once the driver is running.
excelan	The Excelan Ethernet interface device driver. This program talks to the Ethernet chips on the Excelan board contained in the Multibus cage. *excelan* must be run on the node to which the Multibus adapter is connected.
loop	Loops all IP packets received back to the sender. This program pretends to be a network device driver, but actually acts as a "mirror" for all packets it sees. The loop is

primarily used for testing.

PROTOCOL DEMONS

arp	The "Address Resolution Protocol" demon. The address resolution protocol maps 32-bit Internet Protocol (IP) addresses to 48-bit Ethernet hardware addresses. When a user program asks to send an IP message, this process looks at the IP address of the message and supplies a corresponding Ethernet address from its table if such an address is listed. If *arp* does not have a corresponding Ethernet address, it queues the message, and broadcasts the unknown internet address to all hosts on the Ethernet with a request for address resolution. The intended host, whose internet address appears in the request for address resolution, then sends a reply directly to the Butterfly machine. *arp* intercepts this reply, puts the Ethernet address in *arp*'s table, and sends the original message in an Ethernet packet.
internet	Internet services demon. This process implements the TCP, RDP, and UDP protocols. It is by far the largest of the network processes. When the user makes requests of the network via the *MSG* library, this is the process that he or she will talk to.

NETWORK UTILITIES

cleanup	Releases sockets to the system; used after a server process dies.
ipreceive	Defines device drivers that *internet* should talk to. When *internet* is started up, it knows neither the network it is on nor the addresses recognized by that network hardware. *IPreceive* is used to communicate that information. The

	Butterfly software currently has only two device drivers that *internet* can talk to (the *excelan* driver and a software loopback driver called *loop* (q.v.)), but others will be added in the future. *ipreceive* keeps configuration and device-dependent knowledge out of the internet server.
netstat	Displays the status of the network connections.
route	Informs *internet* about gateways. This program tells the *internet* process which gateways *internet* can use for routing internetwork packets. If *route* weren't run, the network software could exchange packets only with hosts on its Ethernet.

NETWORK SERVERS

tftp	"Trivial" File Transfer Program. This program allows Butterfly users to move files to and from another host. This process is most useful to the *inet-loader* (see below). *tftp* is useful only if the host you are trying to talk to is running a *tftp server*. A *tftp* demon written by MIT for 4.2bsd is included in the distribution because the original *tftp* demon distributed with 4.2bsd does not correctly implement the protocol except on SUN workstations.
netloader	Internet loader. This program loads process templates over the network rather than over the serial line. Uses the *tftp* program (see above).
telnetd	TELNET demon. This program allows network users to have a virtual terminal on the Butterfly machine.
boottftp	Stub *tftp* implementation used to load programs quickly during network boot. This program allows programs to be loaded over the network before *netloader*, *tp*, and various supporting programs are loaded.

LIBRARIES

Originally, these library packages were separate libaries, but their functions are so useful that the packages have been added to the standard Chrysalis library, *libcs.a*.

streams-lib	STREAMS remote file system library. The modules in this library provide a user program with a way to read and write files on a remote host over the network. The remote host must be running the *streams-server*.
msgio	Message Passing Network Library. The modules in this library define the message-passing interface between the user's program and the *internet* demon. This library contains routines such as *socket, bind, listen, connect,* and *shutdown.*
net	Network Utility Library. This library contains useful auxiliary routines that deal with network entities. It has such routines as *atoea* (ASCII-to-Ethernet-address), *atoipa*, (ASCII-to-internet-protocol- address), *in_cksm*, and *read_ns*.

REMOTE ACCESS USING TELNET OR BEXEC

It is possible to access the Butterfly computer remotely through the network using the Internet virtual terminal protocol TELNET. The TELNET demon may be started once the network has been brought up:

```
run telnetd wm -login
```

TELNET provides a virtual terminal on the Butterfly machine across the network, running the window manager, just like the Butterfly console. To log off of the Butterfly machine, you should quit from the window manager using *wm's* **<control-g>q** command.

Users in their offices can log into a host computer and *telnet* from their host to the Butterfly machine. When the TELNET connection is opened, a window manager is started, a supercluster and cluster are created, and they can use the Butterfly machine.

Users may also use *blogin/bexec* for more transparent access to the Butterfly from a UNIX host.

ERROR CONDITIONS AND TROUBLESHOOTING

Three of the most common error conditions you may encounter are the ENOSOCKETS error, the *close:ENETRESET* error, and the broadcast packet error.

ENOSOCKETS Error

The ENOSOCKETS error may be caused by two separate conditions. See the *Chrysalis 4.0 Tutorial* for further details.

1. Hung processes may be unnecessarily hanging on to sockets.

   Using the *netstat s* command will show a list of all the sockets. The *cleanup* command will allow the release of unused sockets to the system.

2. There may be network processes that use up the allocated number of sockets.

   The network boot script can be changed to allocate more sockets when the network starts up.

ENETRESET Error

The ENETRESET error means that when the internet daemon tried to close the TCP connection, it found out the SPECIFIC CONNECTION had already been closed by the UNIX end. (This happens, for instance, if the front end program exits without closing a socket.) In the same situation, UNIX would return -1 and set *errno* to

This is a TCP error. It means that TCP cannot guarantee you that all the data you sent was read by the other process. You must rely on application-level semantics to be sure that both ends received everything they should have.

This is the kind of thing you should understand if you are using a TCP connection. See the 4.2/Sun IPC primer for more information, since Butterfly sockets follow the same model.

Broadcast Packet Error

The broadcast packer error typically displays a message like the following:

```
internet: ip_forward error: src 128.8.132.1 dst 128.8.132.255
```

This message may occur every few seconds.

This error occurs when you try to run subnets without properly configuring the network.

The current solution is:

1. Edit the file $CHRYSALIS/include/sitedefs.h

   At the end of this file is a discussion of subnetting, followed by the line:

   ```
   # define SUBNETS
   ```

   which is commented out. At your site, it should not be commented out: subnets should be defined.

2. Recompile and install internet.68:

   ```
   % cd $CHRYSALIS/net-src/internet
   % make ver=Chrys40        (or whatever your version is)
   ```

```
% make install
```

(By doing the “make” and “make install” separately you will not lose your old installed version of internet.68 if the make fails for any reason.)

Chapter 4

Using Assembler Language

This chapter is a quick overview of using assembler language on the Butterfly. It does not describe how to program the Motorola 68000 and 68020 processors. You should refer to the Motorola manual *MC68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual* for details on the 68000, or to the Motorola manual *MC68020 32-Bit Microprocessor User's Manual* for details on programming the 68020.

This chapter describes the differences between the syntax used by Motorola-format assemblers and *as68*, the assembler supplied with the Butterfly programming tools. It also gives a quick summary of all operations supported by *as68*.

ASSEMBLER MNEMONICS

as68 does not use standard Motorola mnemonics or syntax. There is, however, a set of simple rules for converting Motorola mnemonics to *as68* format:

- All *as68* mnemonics should be written in lower case. *as68* considers case significant (like C does), and all its built-in symbols are defined in lower case. User-defined symbols may be in upper, lower or mixed case as desired.

- If the Motorola mnemonic has a size suffix (e.g., the ".l" in "add.l"), drop the "." (e.g., "addl").
- If the instruction uses a special addressing form mnemonic (e.g., "addi" or "adda", but not "addx", which is a different operation entirely), drop the addressing suffix (e.g., "add"). *as68* does not recognize the special form mnemonics in most cases. **Exception:** *as68* does permit the "quick" forms of instructions: addq, subq, and moveq.
- The "move" instruction is written as "mov" in *as68*. "movem", "moveq", and "movep" stay the same, however.
- Branch instructions may have an "s" suffix to indicate a short branch. No suffix results in a short or long branch, at the whim of *as68* (based on its limited knowledge of where the target is). There is no way to force a long branch (other than hand assembly).
- There is a j_{cc} mnemonic corresponding to each b_{cc}, which generates a long branch if necessary, by using a branch/jump instruction pair. There is a similar "jbsr" form of "bsr".
- **Instructions supported by the 68020, but not the 68000, are not supported by as68.** This includes all the bit instructions, the long form of "b_{cc}", the long form of "link", "$TRAP_{cc}$", and others.

All Motorola instruction mnemonics which are different in *as68* are listed in Table 4-1. Instructions which are the same (except for leaving the "." out of size suffixes) are listed in Table 4-2. Instructions not supported by *as68* are listed in Table 4-3.

Table 4-1
Motorola Instruction Equivalents in *as68*

Motorola Mnemonic	*as68* Mnemonic	Comments
add, adda, addi	add	
and, andi	and	
B_{cc}	b_{cc}	not 32-bit form
cmp, cmpa, cmpi	cmp	
bra	bra	not 32-bit form
bsr	bsr	not 32-bit form
chk	chk	word form only, omit length
divs	divs	divs.w form only, omit length
divu	divu	divu.w form only, omit length
eor, eori	eor	
link	link	word form only, omit length
move, movea	mov	
muls	muls	muls.w form only, omit length
mulu	mulu	mulu.w form only, omit length
or, ori	or	
sub, suba, subi	sub	

Table 4-2
Motorola Instruction Mnemonics Identical in *as68*

abcd	jsr	roxr
addq	lea	rte
addx	lsl	rtr
asl	lsr	rts
asr	movem	sbcd
bchg	movep	s_{cc}
bclr	moveq	stop
bset	nbcd	subq
btst	neg	subx
clr	nop	swap
cmpm	not	tas[1]
db_{cc}	pea	trap
exg	reset	trapv
ext	rol	tst
illegal	ror	unlk
jmp	roxl	

1. Do not use TAS on the Butterfly

Table 4-3
Motorola Instructions Not Supported by *as68**

bfchg	bkpt	extb
bfclr	callm	movec
bfexts	cas	pack
bfextu	cas2	rtd
bfffo	chk2	rtm
bfins	cmp2	$trap_{cc}$
bfset	divsl	unpk
bftst	divul	

*All coprocessor instructions; All 32-bit branches

ADDRESSING MODES

as68 supports only those addressing modes supported by the 68000, and does not support the extended set of addressing modes of the 68020. Instructions using these modes must be hand assembled using **.word** directives. *as68* does not follow Motorola addressing mode syntax at all, except for register direct addressing. Register names are in lower case (i.e., "a0", not "A0"). Motorola syntax and the equivalent *as68* format for the supported address modes are shown in Table 4-4. *as68* does not support index scaling, base displacement, or indirect addressing modes.

Table 4-4
Motorola Addressing Syntax and as68 Equivalents

Motorola Syntax	*as68* Syntax
Dn	dn
An	an
(An)	an@
(An)+	an@+
-(An)	an@-
(d$_{16}$, An)	an@(d$_{16}$)
(d8,An,Xn)	an@(d8,xn)
(d8,An,Xn.W)	an@(d8,xn:w)
(d8,An,Xn.L)	an@(d8,xn:l)
(d16,PC)	pc@(d16)
(d8,PC,Xn)	pc@(d8,xn)
(d8,PC,Xn.W)	pc@(d8,xn:w)
(d8,PC,Xn.L)	pc@(d8,xn:l)
symbol	symbol
symbol.W	symbol.w
symbol.L	symbol.l
#value	#value

THE MOVEM INSTRUCTION

The "movem" instruction has a somewhat different argument format than that used by Motorola assemblers. Instead of using a register list, it takes an immediate operand which is the appropriate register mask. Additionally, this mask must be supplied in bit-reversed order if the instruction uses the pre-decrement addressing mode. Thus,

```
MOVEM.L D0-D3/A0/A4, foo
```

becomes

moveml #/110F, foo

in *as68*, while

MOVEM.L D0-D3/A0/A4, -(A7)

becomes

moveml #/F088, sp@-

in *as*.

OTHER DIFFERENCES FROM MOTOROLA SYNTAX

- Comments start with a ";" or "|" character.
- Labels must be followed by a ":" character.
- Local labels are named "*<decimal number>*$", and are forgotten anytime a normal label is encountered. An example:

```
foo:
; start of scope of 1$
        movl    d0, d1
        bmi     1$
        movl    d0, d2
1$:
        subql   #1, d3
; end of scope of 1$
bar:
        addql   #1, d2
        bne     1$        ; *** error: 1$ is no longer defined
bletch:
```

- Constants are assumed to be decimal unless otherwise indicated. A "^" prefixes octal constants, a "0x" or "/" prefixes hexadecimal ones, and a "%" prefixes binary constants. Hexadecimal constants may use either upper or lower case letters. A trailing "." indicates a decimal constant.

- Expressions are normally evaluated strictly left-to-right. "[" and "]" may be used to group expressions like parentheses are used in C. The C binary operators +, -, *, /, &, <<, and >> are supported in *as68*. The C "%" operator (modulo) is "^" in *as68*, and C "|" (bit or) is "!". The unary C operators +, -, and ~ are also supported.
- A symbol may be defined by assignment from an expression with the "=" operator (e.g., **foo** = **bar+3**).
- Most of the pseudo-ops are different. Table 4-5 summarizes the available pseudo-ops.

Table 4-5

as68 Pseudo-ops *<*delim*>is any character not in <*string*>

Pseudo-op	Arguments	Function
.ascii	*<delim><string><delim>*	assemble string constant;*
.asciz	*<delim><string><delim>*	zero-terminated string constant
.blkb	*<length>*	reserve <*length*> bytes of storage
.blkw	*<length>*	reserve <*length*> words of storage
.blkl	*<length>*	reserve <*length*> longwords of storage
.zerob	*<length>*	reserve and zero <*length*> bytes
.zerow	*<length>*	reserve and zero <*length*> words
.zerol	*<length>*	reserve and zero <*length*> longs
.byte	*<expr>[,<expr>]...*	store 8-bit value of expr(s)
.word	*<expr>[,<expr>]...*	store 16-bit value of expr(s)
.long	*<expr>[,<expr>]...*	store 32-bit value of expr(s)
.insrt	"*<filename>*"	include contents of file <*filename*>
.even		force word alignment
.text		assemble into text segment of program
.data		assemble into data segment of program
.bss		assemble into bss segment of program
.end	*[<start address>]*	stop assembly of current file with optional start address
.endpass		stop assembly immediately
.globl	*<symbol>,[<symbol>]*	defines <*symbol*>(s) as global(s)
.comm	*<name>,<expr>*	define common block name and size
.defrs	*<symbol>,<register>...*	define <*symbol*>(s) as register name(s)
.stabs	*"<string>", <type>, <other>, <desc>, <value>*	*enter* <*string*> in symbol table
.stabn	*<type>, <other>, <desc>, <value>*	enter unnamed symbol in symbol table
.stabd	*<type>, <other>, <desc>*	enter unnamed symbol in symbol table with value equal to current location

USER CONSIDERATIONS

- *as68* sometimes "optimizes" your instructions. It is especially liable to turn "addl #k, xn" into "addql #k, xn" if it can. It also will turn "movl #0, foo" into "clr foo", which actually produced different behavior on the 68000 (because it implemented "clr foo" as "and #0, foo"). This will normally only bother you if you think the instruction should have been a different length than the one it produced. It may also startle you if you read the hex output in .168 files closely.
- Expression evaluation is left-to-right–no precedence! Use "[" and "]" to force a different order of evaluation.
- Local labels have a scope only between real labels. This can often produce mysterious complaints about undefined expressions.
- Undefined expressions often produce the error message "span dependency error".
- Global or extern symbols should not be used in immediate expressions in instructions which use short constants (e.g., btst, addq), because the linker can't cope with them. If you do this, you may get unedifying complaints from the assembler. One easy way to inadvertently cause this is to reference an undefined symbol in such an expression when the "-defineall" switch has been turned on.
- The TAS, CAS, and CAS2 instructions (i.e., the instructions that use indivisible read-modify-write memory operations) are not supported by the Butterfly. TAS is recognized by the assembler, but no warning is given if it is used. On the Butterfly Plus, these instructions may be used on local memory only. Remote references by those instructions will cause bus errors. Unfortunately, a strange interaction between the 68020 and the 68881 PMMU will also cause a bus error (even on a local reference) if the PMMU needs to table-walk during the RMW instruction.

LINKING ASSEMBLY LANGUAGE TO C

It is relatively simple to call assembly language from C, and vice-versa. First, you should understand C argument-passing conventions. The Green Hills C compiler passes all arguments on the stack, with the first argument lowest on the stack (just above the return PC). See Kernighan and Ritchie for the details on exactly how various types of arguments are converted when passed as arguments. If your routine will return a value, it should be in d0 when you exit your function. Double-precision floats are passed back in d0 and d1, with the least-significant part of the mantissa being in d1. (If you enable the -X130 switch to the Butterfly C compiler, float results are returned in fp0, instead. This option is disabled by default.)

When the subroutine is entered, the SP will be pointing at the return address, with the first argument (if any), directly above it. Normal procedure is to use a6 as a frame pointer. If you fail to follow this convention, nothing will break, but you may confuse *ddt* or *dbx* if it has to do a stack backtrace through your routine. If you do use the frame linkage, the first instruction in your routine should be "link a6,#-<*size of locals*>", and you should put "unlk a6" just before the rts at the end of your routine. Don't forget that the value in the link instruction should be negative– this is an easy error and results in a badly trashed stack.

Subroutine entry points callable from C must start with "_", which is added by the C compiler to all symbol names. This name must also appear in a **.globl** directive, so that the symbol is marked as externally visible.

A simple example which increments its long integer argument by one and returns the result (called from C as "addone(value)"):

```
         .globl   _addone
_addone:
         link     a6,#-4
         movl     a6@(8), a6@(-4)
         addl     #1, a6@(-4)
         movl     a6@(-4), d0
         unlk     a6
         rts
```

Chapter 5

Getting Started With DBX68

This document is intended to help you get started using the source level debugging and execution capabilities of dbx68. Dbx68 is a tool for debugging programs under Chrysalis^{TM} on the Butterfly^{TM} machine. Dbx68 runs on the host machine (VAX or SUN) and uses the Loader Debugger Protocol (LDP) to interact with the *ldpserver* process that must be running on the Butterfly.

The following is an example debugging session with the "oops" program which is discussed in the *Tutorial for Programming in the C Language.*

NOTATION

In the example shown below, the user input appears after the front end system prompt "#>" where # is an integer, or after the dbx68 prompt "(dbx68)". The rest of the interaction consists of output generated by dbx68. The "[#" and "#]" are delimiters surrounding comments to explain the dbx68 commands that the user has typed.

This tutorial is intended for users of Chrysalis version Chrys4.0, and describes dbx68 version 3.21.175.

STARTING THE LDPSERVER

Before starting the ldpserver, you must start up the network. In order to determine whether the network is running, run "netstat s". Netstat throws if the network is not up and running. Otherwise, it reports on the sockets. If the network is not already running on your Butterfly machine, the following command will start the network:

```
take <machine-name>.netboot
```

At most one ldpserver process may be running on the Butterfly machine. The following command will start the ldpserver:

```
run -kernel ldpserver &
```

If someone already started the ldpserver on the Butterfly machine, an error message like the one following will appear:

```
LDPServer> Unable to complete bind operation to establish listen connection. LDPServer> There is another LDPServer running. LDPServer> LDPServer exiting.
```

When the ldpserver starts up, it disowns itself to the NETSYSTEM object. Therefore, if the netoff program is run, the ldpserver will be killed.

COMPILING FOR DBX68

In order to use all of the debugging features of dbx68, you must compile and link your program with the -g flag to produce the symbol information in the object file. If your program is not compiled with -g, you may use the machine level debugging facilities of dbx68.

Now let's compile the program we want to debug:

```
13> cd ~
14> cp /usr/butterfly/chrys/Chrys40/tools/oops.c68 .
15> setenv CHRYSALIS /usr/butterfly/chrys/Chrys40/BF_PLUS
16> bcc -g oops.c68
```

Please note that the compiler flag to disable the allocation of local variables to registers (usually -X18) is helpful for reliable inspection of locals. Alternatively, the register variable may be examined near a reference to the variable.

CREATING THE BUTTERFLY ENVIRONMENT

In the multi-user environment of Chrysalis 4.0, the supercluster forms the individual user's context. You must create a supercluster via the remote shell's blogin program or via telnet. In either case, the name you supply to the "login:" prompt identifies your supercluster. You may set up any environment variables required by your program before you begin debugging.

If you should forget to set up the Butterfly environment and you've already run dbx68, you may use the dbx68 "sh" command. This command allows you to submit a command line to the shell for execution. In this manner, you may run blogin to create a supercluster.

Here is the source code for the oops program:

```
/* erroneous demo program */

#include <public.h>

subr(x,s)
int x;
char * s;
{
    printf("  in subr: x=%d, s=
    throw(CHECK, s, x);       /* cause a throw to show off backtrace */
}

main(argc, argv)
int argc;
char * argv[];
{
    short foo;          /* place to put uninteresting results */
    char *odd_ptr;      /* pointer to odd address */

    if (argc != 2) {
    printf("usage: oops {batsl}0);
    }
    else switch (argv[1][0]) {
    case ´b´:
        printf("oops: causing bus error0);
        foo = *((short *) NULL);   /* don´t have read access here */
        break;
    case ´a´:
        printf("oops: causing address error0);
        odd_ptr = (char *) 0xFD0001;  /* pointer to odd address */
        foo = *((short *) odd_ptr);       /* word read at odd address */
        break;
    case ´t´:
        printf("oops: throwing away0);
        throw(FAILED, "oops: requested throw", 0);
        break;
    case ´s´:
        printf("calling subr(20,
        subr(20, "foobar");
        break;
    case ´l´:
        while (TRUE) ; /* loop forever */
```

```
            break;
        default:
            printf("oops: invalid argument ´%s´0, argv[1]);
            break;
        }
}
```

DEBUGGING

Let's indicate to dbx68 that we want to run the program immediately by specifying the -r switch. Dbx68 will exit if the program terminates successfully.

```
17>dbx68 -r oops.68
dbx68 version 3.21.175 of 12/21/87 12:06 (socrates.bbn.com).
Type ´help´ for help.
enter target host name or Internet address: jolt.bbn.com
enter target supercluster name (default is ´jvd´):
connected to existing supercluster: jvd
loading program ...
usage: oops {batsl}
18>
```

Now let's debug the program without running it immediately. Notice that the ".68" file name extension is optional.

```
18>dbx68 oops
dbx68 version 3.21.175 of 12/21/87 12:06 (socrates.bbn.com).
Type ´help´ for help.
reading symbolic information ...
reading `/usr/butterfly/chrys/Chrys40/BF_PLUS/chrys.syms´ ...
(dbx68)
```

When starting up, dbx68 reads the Chrysalis symbols found in the file "chrys.syms". In order to locate the chrys.syms file, dbx68 searches the B_PATH environment variable.

Now dbx68 waits for further commands:

```
(dbx68) conn jolt.bbn.com [# Establish connection to ldpserver #]
connected to existing supercluster: jvd
(dbx68) func main  [# Identify the function name #]
(dbx68) stop if argc != 2  [# Set up a conditional breakpoint #]
[1] stop if oops.main.argc <> 2
(dbx68) trace main [# Ask dbx68 to trace entry/exit of main #]
[2] trace main
(dbx68) run [# Run the program with NO arguments #]
loading program ...
calling main(argc = 1, argv = 0xfb000710, 0xfb000790) from function
       process_startup
[2] stopped in main at line 21 in file "oops.c68"
   21       if (argc != 2) {
(dbx68)
(dbx68) print argc
1
(dbx68) status
[1] stop if oops.main.argc <> 2
[2] trace main
(dbx68)
(dbx68) delete 1   [# Remove breakpoint/trace item number 1 #]
(dbx68) cont
usage: oops {batsl}
returning 0 from main

execution completed, exit code is 0
(dbx68) delete 2   [# Remove breakpoint/trace item number 2 #]
(dbx68) status
(dbx68)
```

Let's try some other dbx68 commands:

```
(dbx68) list 14,24 [# Examine the source file #]
    14   main(argc, argv)
    15   int argc;
    16   char * argv[];
    17   {
    18       short trash; /* place to put uninteresting results */
    19       char *odd_ptr;    /* pointer to odd address */
    20
    21       if (argc != 2) {
    22           printf("usage: oops {batsl}\n");
    23           }
    24       else switch (argv[1][0]) {
(dbx68) stop at 21 [# Set a breakpoint at line 21 #]
[4] stop at "oops.c68":21
(dbx68) status [# Examine active stop/trace commands #]
[4] stop at "oops.c68":21
(dbx68) run t  [# Run program with argument ``t´´ #]
[4] stopped in main at line 21 in file "oops.c68"
    21      if (argc != 2) {
(dbx68) where  [# Print procedure call stack trace #]
main(argc = 2, argv = 0xfb000710, 0xfb000790), line 21 in "oops.c68"
process_startup(0x24eb4430, 0x0) at 0x4004e
process_seed() at 0xfd00c2ca
(dbx68)
(dbx68) print argc, *argv
2 "oops.68"
(dbx68) next   [# Execute up to next source line #]
stopped in main at line 24 in file "oops.c68"
    24      else switch (argv[1][0]) {
(dbx68) where
main(argc = 2, argv = 0xfb000710, 0xfb000790), line 24 in "oops.c68"
process_startup(0x24eb4430, 0x0) at 0x4004e
process_seed() at 0xfd00c2ca
(dbx68)
(dbx68) cont   [# Continue execution from where it stopped #]
oops: throwing away

(oops) (OID=0x24062680): throw SYS-FAILED , Number 0x0 @ 04037e
      "oops: requested throw": 0x0

program frozen by throw 4224
```

```
throw 00001080: oops: requested throw (00000000) in main at line 36
            in file "oops
   36                  throw(FAILED, "oops: requested throw", 0);
(dbx68) quit
19>
```

Note: The throw code is 4224, hexadecimal 1080, (SYS-FAILED) and the throw value is 0.

Dbx68 also allows you to connect to an existing process. Let's try executing oops on the Butterfly machine and then asking dbx68 to debug the process.

Butterfly machine:

```
    (cluster 8) [24] oops 1    [# Start oops running #]
```

Different Butterfly Window:

```
(cluster 8) [24] ps -x -on 24 [# Find oops process id #]
Process   State Name

240115f0:   W   (/telnetd.68)
240117d0:   W   (/telnetd.68)
24043800:   R   (/usr/jvd/oops.68)
240d2170:   W   (nbshell.68)
24034080:   W   (nbshell.68)
240441c0:   R   (ps.68)
24001a90:   Pr  (wm.68)
240012b0:   W   <epoch scheduler>
24001430:   W   <remote demon>
240011a0:   W   <startup>
(cluster 8) [24]
```

Dbx68 interaction:

```
19>dbx68 oops
dbx68 version 3.21.175 of 12/21/87 12:06 (socrates.bbn.com).
Type 'help' for help.
reading symbolic information ...
reading `/usr/butterfly/chrys/Chrys40/BF_PLUS/chrys.syms' ...
(dbx68) alias jolt "conn jolt.bbn.com"    [# Define an alias #]
(dbx68) jolt    [# Connect to jolt.bbn.com #]
connected to existing supercluster: jvd
(dbx68) process 24043800   [# Start debugging the running process #]
No filename provided, default is oops.68
reading symbolic information ...
reading `/usr/butterfly/chrys/Chrys40/BF_PLUS/chrys.syms' ...
(dbx68)
(dbx68) where
main(argc = 2, argv = 0xfb000710, 0xfb000790), line 43 in "oops.c68"
process_startup(0x24792290, 0x0) at 0x4004e
process_seed() at 0xfd00c2ca
(dbx68)
(dbx68) list 40,45
   40                subr(20, "foobar");
   41                break;
   42           case '1':
   43                while (TRUE) ;       /* loop forever */
   44                break;
   45           default:
(dbx68)
(dbx68) print argc, argv[1][0][# Print out some variables #]
2 '1'
(dbx68) stop at 43
[1] stop at "oops.c68":43
(dbx68) status
[1] stop at "oops.c68":43
(dbx68) cont
[1] stopped in main at line 43 in file "oops.c68"
   43                while (TRUE) ;       /* loop forever */
(dbx68) where
main(argc = 2, argv = 0xfb000710, 0xfb000790), line 43 in "oops.c68"
process_startup(0x24792290, 0x0) at 0x4004e
process_seed() at 0xfd00c2ca
(dbx68)
(dbx68) cont
[1] stopped in main at line 43 in file "oops.c68"
   43                while (TRUE) ;       /* loop forever */
(dbx68)
```

Let's execute oops on the Butterfly machine with a different option and then look at that process:

Butterfly machine:

```
(cluster 8) [24] oops s
calling subr(20, "foobar")
  in subr: x=20, s="foobar"

(/usr/jvd/oops.68) (OID=0x24112060): throw SYS-CHECK , Number 0x0 @
    0402c2 "foobar": 0x14
```

Dbx68 interaction:

```
(dbx68)  process 24112060
No filename provided, default is oops.68
reading symbolic information ...
reading `/usr/butterfly/chrys/Chrys40/BF_PLUS/chrys.syms´ ...
(dbx68) where
_catch_df() at 0xfd001ebe
_throw(0x2080, 0x208b4, 0x14) at 0xfd00fd88
subr(x = 20, s = "foobar"), line 10 in "oops.c68"
main(argc = 2, argv = 0xfb000710, 0xfb000790), line 40 in "oops.c68"
process_startup(0x14ed2520, 0x0) at 0x4004e
process_seed() at 0xfd00c2ca
(dbx68)
(dbx68) list subr
    4
    5   subr(x,s)
    6   int x;
    7   char * s;
    8   {
    9       printf("  in subr: x=%d, s=
   10       throw(CHECK, s, x);       /* cause a throw to show off backt
   11   }
   12
   13
   14   main(argc, argv)
(dbx68) print x,s
20 "foobar"
```

```
(dbx68)
(dbx68) quit    [# Exit dbx68 #]
```

This introduction should give you enough information to begin debugging using dbx68. For further details, see the dbx68 manual page (under Host Tools) and the ldpserver manual page (under Butterfly Tools).

Chapter 6

Butterfly Event Logging Facility

The Butterfly Event Logging Facility can be used to produce a timestamped log of events that occur during the execution of a Butterfly program. The resulting event or history log can then be used to understand dynamic aspects of the program's behavior. This can be an aid to debugging and to improving program performance.

This chapter describes the event logging facility and how to use it.

DESCRIPTION

There are three parts to the Butterfly Event Logging Facility:

1. A collection of event logging functions.

   To use the facility with a Butterfly program, it is necessary to instrument the program by inserting calls that invoke event logging functions. These event logging functions are accessible to C language and FORTRAN programs.

2. The *sendelog* utility.

   Execution of a Butterfly program instrumented with calls that invoke event logging functions produces an event log on the Butterfly. The Butterfly utility *sendelog* can be used to transfer the event log to the front end host for display and analysis. *sendelog* uses the Butterfly "streams package" to transfer the event log.

3. The *gist* display program.

   The *gist* program can be used to display a Butterfly history log graphically. Gist uses the graphics capabilities of the X window system. Consequently, the X window system must be running on the host when *gist* is used.

The following steps are necessary to use the event logging facility with a Butterfly program:

1. The program must be instrumented by inserting calls to event logging functions.
2. The instrumented program must be run on a Butterfly to produce an event log.
3. The *sendelog* utility must be run on the Butterfly to move the event log to a host for display.
4. The *gist* program must be run on the host to display the event log.

The sections that follow describe each of these steps.

INSTRUMENTING A PROGRAM

To use the package, calls to a "log event" procedure are inserted into a program. The file **/usr/butterfly/src/gist/example.c** is an example of a C program that has been instrumented in this way.

Each entry in an event log contains

- A time stamp (the real time clock value when the event was recorded);
- An integer event code that serves to identify the event;
- An arbitrary 32 bit data item quantity. The data item is included in the log entry for use by analysis and presentation programs, such as *gist*.

The event log produced by the logging package is implemented by a related collection of Chrysalis objects.

> There is a "log" object for each processor. The object for a processor log contains the entries for each event logged on that processor.
>
> There is a "directory" object that serves to define the event log. It catalogs the individual processor logs that make up the log.

The following must be done to instrument a Butterfly program:

1. Include *elog.h*

   #include <elog.h>

2. A call must be inserted into the program to make the directory object for the event log. The directory object must be created before any other event logging functions are invoked by the program.

   From C this is done by

   ```
   ELOG_INIT (name)
   ```

   name is a character string used to generate a name for the event log; the name generated is "name.elog".

   From FORTRAN this is done by

   ```
   CALL ELOG_INIT (NAME, CLEAR)
   ```

3. Each type of event to be logged must be defined so that the event type can be handled properly by display and analysis programs.

   From C an event is defined by

   ```
   ELOG_DEFINE (event_code, event_name,
   data_format_string)
   ```

event_code is an integer which uniquely identifies the event type. The use of small integers is recommended. *event_name* is a character string. It is typically used by display programs, such as *gist*, to label events. *data_format_string* is a "printf" format string which specifies how to print the data item logged with the event.

From FORTRAN this is done by

```
CALL ELOG_DEFINE (EVENT_CODE, EVENT_NAME,
DATA_FORMAT_STRING)
```

For example, the C statement:

```
ELOG_DEFINE (4, "State variables updated", "Iteration
%d");
```

defines event type 4 for which the data item logged is an integer representing an iteration count.

4. Prior to logging any events on a processor, the object for the processor log must be created.

   From C this is done by

   ```
   ELOG_CREATE (name)
   ```

   nameo is a character string which specifies an event log ("name.elog") previously initialized by ELOG_INIT.

   From FORTRAN this is done by

   ```
   CALL ELOG_CREATE (NAME)
   ```

   NOTE: In the present implementation, only one process on a processor may execute ELOG_CREATE for a given event log (and log events to that log).

5. After the directory for an event log has been created, the event types defined, and the log for a processor created, the processor may log events.

   From C an event is logged by

   ```
   ELOG_LOG (event_code, data)
   ```

   event_code specifies an event type previously defined by ELOG_DEFINE, and *data* is an arbitray 32-bit data item that is recorded along with the time and the event_code.

   From FORTRAN this is done by

   ```
   CALL ELOG_LOG (EVENT_CODE, DATA)
   ```

The event logging facility includes functions beyond the four (ELOG_INIT, ELOG_DEFINE, ELOG_CREATE, ELOG_LOG) described above. These additional functions are described in the last section of this chapter.

NOTE: The C language interface described above is implemented by macros. The compile time symbol ELOG controls how the macros expand. If ELOG is defined, the macros expand to invoke the appropriate event logging functions. If ELOG is undefined, the macros expand to null statements. This makes it possible to control at compile time whether an instrumented program creates an event log. If ELOG is undefined, then no runtime overhead is incurred by the presence of the event log instrumentation.

THE SENDELOG UTILITY

The *sendelog* utility is used to move event logs from the Butterfly to a host for analysis.

To use *sendelog*:

1. Make sure the streams-server is running on the target host and the necessary Butterfly enviroment variables (STREAMS_HOST, STREAMS_PASSWORD, etc.) are set properly.
2. Run sendelog on the Butterfly. To transfer an event log named "program.elog" to the file "/usr/jones/data/program.elog":

   ```
   (new) [f] sendelog program /usr/jones/data/program
   ```

   sendelog responds by printing its progress as it sends the event log.

THE *GIST PROGRAM*

The best way to learn how to use *gist* is to try it. The file /chrys/Chrys40/Examples/gist_example.elog is an example event log that can be used with *gist.*

gist manages 3 separate windows.

1. Trace Window.
   The trace window is the largest window, and occuppies most of the screen. It contains an event trace for each processor. The set of processor traces make a 2 dimensional processor-event versus time display. Time advances on the horizontal axis and the processor axis is the vertical axis. Each processor trace is a horizonal line. Logged events are displayed as event boxes on the appropriate processor trace.

   The user can scroll forward and backward along the time axis, as well as zoom in and out. It is also possible to zoom and scroll the processor axis.
2. Legend Window.
   The legend is a small window that appears to the right of the trace window. It contains information about the log currently being displayed and can be used to control which event types are displayed. The legend window contains an entry for each event type (defined via ELOG_DEFINE). The entry includes code for the event and its name.

In addition, part of the entry is a box that indicates whether that event type is currently displayable; the box is a "button" that may be toggled via a mouse click to control whether the event type is displayable.

3. Prompt Window.
The Prompt Window appears beneath the Legend Window. It is used to help the user by prompting for expected actions.

STARTING *GIST*

gist is started on the host by:

```
shell> gist [eventlog]
```

for example:

```
shell> gist /usr/butterfly/chrys/Chrys40/Examples/gist/gist_example
```

Note that the .elog filename is not specified.

If no event log is specified, *gist* simply prompts for an event log name:

```
shell> gist
Enter event log name:
```

After opening the event log, *gist* displays the initial portion of it.

Commands to *gist* are entered by means of a popup menu. To popup the command menu, move the mouse to the Trace Window and press the middle mouse button and hold it down. To select a command from the menu, move the mouse over the command desired and release the middle mouse button; to abort the command selection, move the mouse out of the menu and release the middle button.

Some commands gather parameters by means of dialog boxes. To select a particular parameter to specify, move the mouse over the entry and click any mouse button. For "text" parameters, the desired text may then be entered. The specification of single line "text" parameters can be terminated by a carriage return or by moving the mouse out of the text region and clicking it; the specification of multiple line "text" parameters can be terminated only by moving the mouse out of the text region and clicking. The dialog box can be completed by clicking the mouse over the "OK" or "Cancel" buttons, or, if no entry is selected, by typing a carriage return.

Controlling the Trace Display

When an event log is opened, an initial trace display is created. The initial trace display contains a processor trace for each processor in the log, beginning at time = 0 (the time of the first event in the log). The space between processor traces is chosen so that every trace can be displayed. The scale of the time axis is arbitrarily chosen to display about 3000 microseconds of the event log.

Logged events are indicated by boxes on processor traces. If there is enough space between traces, event codes are displayed within the event boxes. (Generally, if there are 32 or fewer processors, codes can be displayed within the boxes; if there are 64 or more processors they cannot.)

Controlling the Events Displayed

When *gist* constructs the initial event log display, it displays event boxes for all event types. The Legend Window can be used to control which event types *gist* displays. The small square boxes to the left of the event descriptions can be used to control whether or not events of a particular type are to be displayed. A box can be "toggled" by moving the mouse over the box in the Legend Window and clicking any mouse button. After the desired event types to be displayed have been specified, moving the mouse over the box labeled *Make Event Display Changes* and clicking any mouse button will cause *gist* to change the event types that are displayed.

Measuring Time Between Events

The Time Ruler menu command can be used to measure the time between 2 positions in the Trace Window. After selecting the Time Ruler command, the mouse is used to specify a fixed "anchor" point for the ruler. After the anchor point is fixed, *gist* "tracks" mouse movements and displays the time between the anchor point and the current mouse position.

Zooming and Scrolling the Time Axis

The bottom right of the trace display contains 6 mouse sensitive buttons used to control the time region of the event log appearing on the screen. Clicking any mouse button while the mouse is positioned above one of these buttons will invoke the indicated function.

The "zoom in" and "zoom out" buttons control the scale of the time axis.

The 4 scroll buttons "start", "scroll left" (labeled with a left arrow), "scroll right" (labeled with a right arrow) and "end" can be used to move back and forth along the time axis.

The scroll left and scroll right buttons scroll the display by half the width of the onscreen time axis.

Sometimes it is desirable to more precisely control the onscreen portion of the event log. There are a variety of menu commands that allow you to exercise such control, including:

Search For Event

> This command allows you to specify an event to find. You can specify the event, the processor for the event (any processor or a specific processor), the search direction (forward or reverse in the log), the time from which to begin the search, and the action to take if the event is found (scroll to the event or zoom the time axis to include the event).

Set Onscreen Min To ...

This command scrolls as required to provide a new onscreen minimum for the time axis. The time value for the new minimum can be specified in terms of a specific onscreen event (via the mouse), a specific onscreen time (via the mouse), or a specific time.

Set Onscreen Max To ...

This command scrolls as required to provide a new onscreen maximum for the time axis. The time value for the new maximum can be specified in terms of a specific onscreen event (via the mouse), a specific onscreen time (via the mouse), or a specific time.

Scale By ...

This command changes the scale of the time axis such that the origin (the onscreen minimum) remains unchanged. The amount to scale can be specified in terms of a specific onscreen event (via the mouse), in which case the scale will be changed so that the event is the new onscreen maximum, or in terms of a specific onscreen time (via the mouse), in which case the specified time becomes the new onscreen maximum.

Set Center For Zoom To ...

This command changes the time value used for centering the zoom operations invoked by the zoom in and zoom out buttons. The new zoom center can be specified in terms of an onscreen event (via the mouse) or an onscreen time (via the mouse).

Zoom ...

This command can be used to zoom in or out on the time axis is a more controlled fashion than that provided by the zoom in and zoom out buttons. The zoom operation can be specified in terms of an onscreen area that is to occupy the entire Trace Window (zooming in), or in terms of a compression (zooming out) or expansion (zooming in) of an onscreen area anchored by the onscreen minimum.

Zooming and Scrolling the Processor Axis

The lower right side of the trace display contains 6 buttons used to control the scale of the processor axis and the processors traces appearing on the screen (for logs containing sufficently many processors).

The "zoom in" and "zoom out" buttons control the scale of the processor axis.

The 4 scroll buttons "start", "scroll up" (labeled with an up arrow), "scroll down" (labeled with a down arrow) and "end" can be used to move up and down along the processor axis.

NOTE: Zooming out both the time and the processor axes often reveals patterns of events not so readily apparent in other more "normal" views.

Obtaining More Detailed Views

Events

Positioning the mouse over an event box and clicking the right mouse button pops up an event detail box that contains information about the event, including the event time, the event code, the event name, and the data logged with the event (displayed in the format specified by the ELOG_DEFINE call used to define the event).

When an event detail box is up, moving the mouse over it or over the corresponding event box and clicking the left mouse button will cause the event detail box to be taken down.

Positioning the mouse over a processor trace line and clicking the right mouse button pops up an event detail box for the event nearest the mouse (if any), and causes the program to enter a mode where clicking the left mouse button pops up event detail boxes for successively earlier (onscreen) events and clicking the right mouse button pops up event detail boxes for successively later (onscreen) events. Clicking the middle mouse button exits this mode. To indicate entering this mode, the mouse cursor, normally an "X" with a hole in the center, changes shape to a small box between left and right arrows; upon exiting the mode, the normal mouse cursor is restored.

Positioning the mouse over a processor trace line and clicking the left mouse button will cause any event detail boxes for that processor trace to be taken down.

Processors

When there is enough space between processor traces, the traces are labeled. The traces are not labeled if they are too closely spaced, either because there are too many (typically > 100) or because the user has zoomed out too far.

If the processor traces are not labeled, positioning the mouse to the left of the processor axis above the time axis and clicking the right mouse button pops up the label for the nearest processor trace, and causes the program to enter a mode where clicking the left mouse button pops up labels for traces successively above and clicking the right mouse button pops up labels for traces successively below. Clicking the middle mouse button exits this mode. To indicate entering this mode, the mouse cursor changes to a small box between arrows pointing to the upper left and lower right; upon exiting the mode, the normal mouse cursor is restored.

Obtaining Hardcopy

The *Dump Screen To File* menu command can be used to write the contents of the Trace Window and the Legend Window to a file in PostScript format. The resulting file can then be printed on a PostScript printer. The *Dump Screen To File* allows the user to specify a caption for the screen dump.

Exiting *gist*

To exit *gist*, position the mouse over the QUIT button at the bottom center of the trace window and click any mouse button. A small window will pop up asking for confirmation.

Alternatively, the "quit" command may be selected from the command menu. Confirmation will be requested.

Upon exiting, the Trace, Legend, and Prompt windows are destroyed.

THE EVENT LOGGING FUNCTIONS

The following summarizes the *include* file and the functions required to instruument a Butterfly program to produce an event log.

#include <elog.h>

> *elog.h* contains definitions required by the event logging routines.

ELOG_INIT (name)

> ELOG_INIT is a macro in the C environment and a subroutine in the FORTRAN environment. It creates a directory object for the log "name.elog". ELOG_INIT must be called before any other event logging functions are used. In the C environment ELOG_INIT expands into a call to the routine elog_init if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

ELOG_CREATE (name)

ELOG_CREATE is a macro in the C environment and a subroutine in the FORTRAN environment. It creates a log object for the processor that calls it for the log "name.elog". ELOG_CREATE must be called on a processor before any events are logged by the processor.

In the C environment ELOG_CREATE expands into a call to the routine elog_create if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

ELOG_DEFINE (event_code, event_name, data_format_string)

ELOG_DEFINE is a macro in the C environment and a subroutine in the FORTRAN environment. It is used to define an event type. *event_code* is an integer which uniquely identifies the event type. The use of small integers is recommended. *event_name* is a character string. It is typically used by display programs, such as *gist*, to label events. *data_format_string* is a "printf" format string which specifies how to print the data item logged with the event.

ELOG_DEFINE should not be used until ELOG_INIT has been called.

In the C environment ELOG_DEFINE expands into a call to the routine elog_define if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

ELOG_LOG (event_code, data)

ELOG_LOG is a macro in the C environment and a subroutine in the FORTRAN environment. It is used to log an event of type *event_code* along with a 32 bit *data* item.

A processor log object can hold 5461 events. ELOG_LOG performs a range check prior to logging the event. If the log object for the processor is full, a warning message is printed and events are no longer logged for the processor.

In the C environment ELOG_LOG expands into code that logs the event if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

ELOG_FAST_ELOG (event_code, data)

ELOG_FAST_LOG is a macro in the C environment and not available in the FORTRAN environment. It is similar to ELOG_LOG but is somewhat faster (and less safe) since no range check is performed prior to logging the event.

ELOG_FAST_LOG expands into code that logs the event if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

ELOG_RESET

ELOG_RESET is a macro in the C environment and a subroutine in the FORTRAN environment. It empties the event log for a processor.

In the C environment ELOG_RESET expands into a code that resets the event log for the processor if the compile time symbol ELOG is defined; otherwise, it compiles into a null statement.

Chapter 7

STREAMS Remote File System Library

As applications for the Butterfly have developed, the need for ready access to files resident upon other hosts has become apparent. This need was first served by the *rfs library*. *STREAMS* is an outgrowth of the concepts contained in the rfs library, but provides higher performance (on the order of 250 000 bits/sec throughput on an unloaded system), and the ability to have files open on multiple remote hosts simultaneously. The server has been similarly enhanced to use multiple processes, perform limited authentication, and to permit access control on a per-file basis. It also is designed to work on both VAXs and SUNs without modification. The price paid for this increased flexibility is the ability to do arbitrary seeks on remote files.

In addition to providing a new library for use in programs requiring high-speed streams, new utilities are introduced with this library. These utilities permit redirection of both input and output to remote files for arbitrary Butterfly programs. These new programs provide the user with the ability to do journalling and to run command scripts from remote host files, all without the user's programs having to know about the file system. Other new utilities allow the user to measure the throughput of the streams package at any time.

SYSTEM DESCRIPTION

As with the rfs library, the streams library provides basic functions for opening, reading, writing, and closing files. On the Butterfly end, these functions are contained in a single module, *streams-lib.o68* which is in libcs.a, the default C language function library. On the host end, these functions are performed by a program called *streams-server.* To the Butterfly user, these programs operate as a single system.

The streams library provides eight basic functions that the user can call. The functions are as follows: n_initialize_net_files (); fileptr = n_open (filename, access); n_read (fileptr, buffer_address, byte_count); n_write (fileptr, buffer_address, byte_count); n_close (fileptr); n_set_host (host-name); n_set_quiet (boolean) n_set_port (portnumber) n_set_password (password);

Manual pages for each function can be found in this section of the manual.

n_initialize_net_files is a function that must be called before any other streams function can be invoked. Its purpose is to initialize the data structures that are used for keeping track of the open files on the Butterfly side of the system. *n_open* takes arguments like the UNIX *fopen* function, and returns a pointer to a structure of type *FILE* in the communications segment of the process. *n_read* and *n_write* both operate like *read* and *write* in UNIX; they attempt to transfer a buffer of length *byte_count* and return the actual number of bytes read or written. *n_close* closes the file and sends an end-of-file indication to the other end.

Note that no *n_seek* operation is provided. This omission is due to the way that the streams library is implemented. Every file opened by the user results in a unidirectional TCP stream being opened between the user and the server. No overhead is added to this stream once it is opened; all *n_read* and *n_write* operations simply map into read and write operations on the corresponding TCP connection. This lack of overhead is the reason that the data can be transferred between the two systems so quickly. *n_open* and *n_close* operations are negotiated over a separate control TCP connection. Since the control information is separated from the data information, synchronization of the two streams is difficult. Since most users polled want simply to read and write data without seeking, we have decided to ignore the synchronization problem by not providing seeks.

One of the enhancements provided by the streams library is the ability to open files on many different hosts simultaneously. The user can specify the host she wishes to connect to by prefixing the *filename* parameter of the *n_open* subroutine with a four-octet internet address followed by a colon (:). Therefore, the call

```
n_open ("128.11.1.1:/etc/motd", "r");
```

opens the file */etc/motd* on host 128.11.1.1.

Because servers running on different hosts may not be listening on the same TCP ports, a similar mechanism is provided for specifying the port number to which to connect. If the host address is followed by a comma and a number in the filename of the *n_open*, this number is used as the foreign port address for future connection attempts by the user. Therefore, the call

```
n_open ("128.11.1.1,25252:/etc/motd", "r");
```

will attempt to connect to the server on port 25252 of host 128.11.1.1 and then open file */etc/motd.*

To avoid confusion resulting from several users having streams servers running on the same host, each server has a password associated with it that the user end must supply when opening a file. This password may also be specified in the file name after the host address by enclosing it in square brackets ([]). Therefore, the call

```
n_open ("128.11.1.1,12345[MyPassword]:/etc/motd", "r");
```

will open a connection to the streams server listening on port 12345 on host 128.11.1.1, specify a password of *MyPassword*, and open file

/etc/motd for reading.

Host address, port number, password, and file name must be specified in that order.

Most users will not want to be bothered typing in all these substrings just to open a simple file. For this reason, all of the above filename parameters are optional. If a parameter is not specified by the user in the filename, the user's environment variables are searched for a corresponding streams variable. Host addresses are specified by the variable *STREAMS_HOST*, port numbers by the variable *STREAMS_PORT*, and passwords by the variable *STREAMS_PASSWORD*. Therefore, the effect of the last command mentioned above could have been achieved by preceding the *n_open* call with the following shell commands (shown with the shell prompt):

(new) [f] setgenv STREAMS_HOST 128.11.1.1 (new) [f] setgenv STREAMS_PORT 12345 (new) [f] setgenv STREAMS_PASSWORD MyPassword

and then invoking a program that executes

```
n_open ("/etc/motd", "r");
```

Should a program wish to override the environmental defaults for these parameters, three subroutines are provided to allow it to do so: *n_set_host*, *n_set_port*, and *n_set_password*. *n_set_host* and *n_set_password* accept strings as arguments; *n_set_port* takes an integer. A parameter changed with one of these three routines will stay changed with a program until either the next occurrence of such a call or the next time the user overrides the defaults with a filename specification. If any of these routines are provided NULL or zero as an argument, they will change the default back to the appropriate environment variable.

If host, port, and password parameters are not specified in either the environment or in the filename, then, as a last resort, compiled-in defaults are used. The streams host will be whatever the system's BOOTHOST variable is set to. The port is STREAMS_SERVER_PORT as defined in <net/streams.h>. (The distributed value is 12345.) If the user neglects to assign a password in either the environment or filename, a password of "" (the null string) will be supplied.

These three techniques give the user a great deal of flexibility regarding which hosts, ports, and passwords are to be used when opening files. *n_set_quiet* suppresses library informational messages if given a non-zero argument, and re-enables those messages if given a zero argument.

THE SERVER

The *streams-server* program implements the "back-end" of the streams system. This program accepts TCP connections generated by the streams library functions, opens disk files on behalf of the user, and then forks a copy of itself for each open file.

The server produces logging messages on its standard output file. These messages are time stamped and monitor the opening and closing of files by the server. Most users will wish to redirect the standard output to a file for use should a problem occur.

The only user parameter accepted by the server currently is an alternate port number. If started with no arguments, the streams server will listen on TCP port 12345. Should this port be busy (for example, it may be in use by another streams server on the same machine), it may be desirable to start the streams server on a different port. If the streams server is started with one argument, the argument specifies the port number that should be used for accepting connections.

When the streams server is started, it will prompt you for a password for use in accepting connections from Butterfly users. Echoing will be turned off while you enter the password. Once the password has been set, the streams server will fork a subprocess of itself running in the background. Should you wish to kill the streams server at a later time, the process number of the child process is printed at the time of startup and is in the first line of the log file.

AN EXAMPLE

The following is a simple program that reads a file and prints it on the standard output using the streams library.

```
/*
 * type.c68
 *
 * Open a streams file and print it on the standard output.  e.g.,
 *    type 128.11.1.1:/etc/motd
 */

#include <public.h>
#include <stdio.h>

char buffer[2048];          /* place into which to read data */
FILE *read_file;            /* network file */
FILE *n_open ();            /* returns a FILE * */

main (argc, argv)
     int argc;
     char **argv;
{
    register int count;

    n_initialize_net_files ();
    if (argc < 2)           /* Was name supplied? */
    {
        printf ("No file name given\n");
        exit (1);
    }
    if ((read_file = n_open (argv[1], "r")) == NULL)
    {
        printf ("Cannot open file %s for reading\n", argv[1]);
        exit (2);
    }

    /*
     * Read and print until EOF
     */
```

```
        while (count = n_read (read_file, buffer, sizeof (buffer)))
            write (stdout, buffer, count);
        fflush (stdout);
        n_close (read_file);
        exit (0);
    }
```

The program begins by initializing the network file system via a call to *n_initialize_net_files*. It then checks to make certain it was given an argument; if so, it opens the argument string for reading via the streams library call *n_open*. Once the file is open, *n_reads* are done until the byte count returned by *n_read* is 0, indicating end-of-file. Then the output is flushed, the file is closed, and the program exits.

This file is compiled using bcc on the VAX host. No flags or special libraries are required because the streams library is part of libcs.a, the standard C library searched by bcc.

Once the program is compiled, we start up the streams-server using the following sequence:

$ streams-server >streams.log Please type in a password for remote users (echo will be suppressed): *abcdef* Thank you, streams-server starting as process 25932 $

This operation would normally be performed only once a session.

Now we connect to the Butterfly host via a serial line, the telnet program, or **blogin.** Once we are logged onto the Butterfly, two ways of running this program are possible. We could fully specify the server host, port, and password, as well as the file name by typing

    (new) [f] type 128.11.1.2,12345[abcdef]:/etc/motd

However, if we wished to run the program again, we would have to type in the host address, port number, and password, as well as the new file name. Therefore, in a case where the user wishes to use the streams library extensively, it is better to set environment variables that define the host and server parameters. Our Butterfly console session would then look like this:

(new) [f] setgenv STREAMS_HOST 128.11.1.2 (new) [f] setgenv STREAMS_PASSWORD abcdef (new) [f] type /etc/motd

Notice that we did not set the variable STREAMS_PORT (although we could have), because we started the streams-server using the default port.

Once the user's environment is set up, running the program with a different file is easy, e.g.

```
    type .login
```

MULTIPLE FILES

Making a few simple modifications to the *type* program results in a program that copies files on a single machine or between two machines. The program is as follows:

```
/*
 * cp.c68
 *
 * Open a streams file and write it on another streams file.  e.g.,
 *    cp /etc/motd 128.11.1.2:mymotd
 */

#include <public.h>
#include <stdio.h>

char buffer[2048];              /* place into which to read data */
FILE *read_file;                    /* network file */
FILE *write_file;                   /* ditto */
FILE *n_open ();                    /* returns a FILE * */

main (argc, argv)
     int argc;
     char **argv;
```

```
{
    register int count;

    n_initialize_net_files ();
    if (argc < 3)           /* Were names supplied? */
    {
        printf ("Usage: cp file1 file2\n");
        exit (1);
    }
    if ((read_file = n_open (argv[1], "r")) == NULL)
    {
        printf ("Cannot open file %s for reading\n", argv[1]);
        exit (2);
    }
    if ((write_file = n_open (argv[2], "w")) == NULL)
    {
        printf ("Cannot open file %s for writing\n", argv[2]);
        exit (2);
    }

    /*
     * Read and write until EOF
     */

    while (count = n_read (read_file, buffer, sizeof (buffer)))
            n_write (write_file, buffer, count);
    n_close (write_file);
    n_close (read_file);
    exit (0);
}
```

Provided that the user's streams environment variables are set as they were in the previous example, and the streams-server is still running, typing

```
cp /etc/motd mymotd
```

will result in the file /etc/motd on host 128.11.1.2 being copied to the file mymotd in the directory where the streams-server was started. However, a more powerful result can be accomplished by typing

```
    cp /etc/motd 10.1.0.82:/usr/cdh/mymotd
```

This will copy the file /etc/motd on 128.11.1.2 to /usr/cdh/mymotd on 10.1.0.82 (provided that a streams-server is listening there on the default service port with the same password).

The reverse is not true however. Typing

```
    cp 10.1.0.82:/usr/cdh/mymotd /etc/motd
```

will simply move the file /usr/cdh/mymotd to /etc/motd on host 10.1.0.82. This is because every time a host, port, or password is changed in a file name, it is remembered by the streams library functions for future use. Since the first argument of cp is opened first, its host address is remembered for use in the open of the second argument. To achieve the effect of moving the file from 10.1.0.82 to 128.11.1.2, either the host name must be specified in both file names, or the STREAMS_HOST environment variable changed to 10.1.0.82 before the program is run. Since most users perform most of their work on one host, it seems more useful to have the library remember the host name when it was not the default, than to have the system always return to the default. Note that defaults are overridden in this way only for the life of the program in which the open of the new host happens (i.e., it does not change the STREAMS_HOST environnment variable).

NEW TOOLS FOR REDIRECTING INPUT AND OUTPUT

With the streams library, six new tools have been distributed that make use of the streams server. There are *stdin*, *stdout*, *teein*, *teeout*, *streams-source*, and *streams-sink*. *stdin* and *stdout* perform functions analogous to the less-than (<) and greater-than (>) operators in the UNIX shell. They fork a new process with the standard input or the standard output of that process directed at a streams library file. Therefore, to record the output of the Butterfly *MatrixMultiply* command in a file *results* on the STREAMS_HOST, you would simply type

```
stdout results MatrixMultiply
```

To redirect the output to a different host, you would need only prefix the file name argument with a host address (and password prefix if the server password were different).

stdin works similarly when you wish to redirect the standard input.

> *teeout* is directly analogous to the *tee* command in UNIX; it not only redirects the output to a file, but also prints the output as it does this. *teein* does the same thing with standard input; it prints the data from the file as it is sent to the user program.

Due to a limitation in interprocess I/O in Chrysalis, these programs currently do all their input and output one character at a time; therefore, performance when using these programs will not be comparable to that achievable when using the streams library directly. Nonetheless, data rates of 20 000 bits/second are easily achievable.

The last two programs, *streams-source* and *streams-sink*, are test programs for use with the streams server in measuring the throughput of the streams system on your network. The command syntax is as follows:

```
streams-source filename write-size number-of-bytes
```

Therefore to test how fast you can move 200000 bytes of data to your host 3400 bytes at a time, you could use the command

```
streams-source /dev/null 3400 200000
```

To do a similar process using *streams-sink* to transfer data from the remote host to the Butterfly, a source file of sufficient size is required (my personal favorite is /usr/dict/words); the process will terminate when either end-of-file is reached or the count is exhausted.

When they complete their transfers, streams-source and -sink print out statistics of how well they did. Here is some sample output recorded with *stdout* from bbn-atlas, a 3-processor Butterfly talking to a VAX 11/785 over a 10 MBit Ethernet.

```
Trying control connection ...
Open
Trying data connection ...
Open
Starting transfer of 200000 bytes, 3400 bytes at a time.
Closing . . . .Transferred 200600 bytes in 5 seconds
buffer size was 3400
Data rate was 34232 bytes per second, 273856 bits per second
Closing connection . . . .
Closed
```

CAVEATS

The following items should be kept in mind when writing and running programs using the streams library (including programs like *stdout*):

1. If two users wish to use the streams-server at the same time on the same host, and they don't wish to share a common directory, one of the servers will have to be started with a non-default port number.

2. Authentication is only a string comparison, and the strings are sent in the clear.

3. Performance of the package is highly dependent upon the size of blocks that you transfer. Block sizes between 2048 and 3400 bytes seem to be the most efficient on our Ethernet. Larger sizes sometimes suffer due to buffering constraints, fragmentation, and collisions. Smaller sizes get swamped with packet overhead.

4. Performance is also highly dependent upon the traffic on your Ethernet, which most users cannot control.

5. No *n_fread* and *n_fwrite* functions are yet provided, requiring the user to provide her own buffering to achieve the kind of block sizes

mentioned above. These functions may be provided in a future release.

6. If you are transferring binary data to or from a VAX using this package, you will have to do a four-byte reversal of all integers contained in objects greater than 1 byte long.

7. If you want to know why an operation failed, you have to look at the error number returned by the server. While this information is available in the library routines (it's in reply_buffer.n_reply_errno), there is currently no way to communicate this information to the user. The only way to verify if a transfer really completed correctly now is to compare the value returned by *n_close* (which is the number of bytes successfully transferred) with the number of bytes you attempted to transfer.