Butterfly Project Report

Modula-2 on the BBN Butterfly Parallel Processor

Thomas J. Olson

January 1986

Computer Science Department
University of Rochester
Rochester, NY 14627
Modula-2 on the BBN Butterfly Parallel Processor†

Thomas J. Olson

Abstract

This report describes the current state of the Rochester Butterfly Modula-2 compiler and runtime environment. It covers use of the compiler, the standard runtime environment, and the (experimental) Butterfly environment. It also describes methods of extending the experimental environment and obtaining access to operating system functions.

† This work was supported in part by the Defense Advanced Research Projects Agency U. S. Army Engineering Topographic Labs under grant number DACA76-85-C-0001 and in part by the National Science Foundation under grant number DCR-8320136.
1. Introduction

This document provides an introduction to the Rochester Butterfly Modula-2 compiler. It assumes that you are familiar with the Modula-2 language and know enough about the Butterfly to compile and run C programs on it. A good source for information about Modula-2 is *Programming in Modula-2* by Niklaus Wirth [1]; for Butterfly information see BPR #1, "Getting Started with the BBN Butterfly Multiprocessor" [2].

The Butterfly Modula-2 compiler is an adaptation of the VAX compiler written by Mike Powell at DEC Western Research Laboratories. Neal Gafter of the University of Rochester retargeted the Powell compiler to the 68000, and the author provided Butterfly runtime libraries. The compiler itself knows nothing of the Butterfly and its architecture: all parallel processing services are provided through library calls.

Section 2 of this document contains instructions for using the compiler. In section 3 we describe the Butterfly version of the standard (VAX-compatible) Modula-2 libraries, with particular attention to features described in Wirth’s book and the Powell compiler documentation that are not available on the Butterfly. Section 4 presents an experimental library of modules that provide access to some functions of the Chrysalis operating system. Methods for extending the module library are discussed in section 5. The appendices contain definition modules for the libraries described in sections 3 and 4.

2. Using the Compiler

Compiling a Modula-2 program for the Butterfly is slightly more complex than compiling a C program, partly because the *xmake* utility does not know about Modula-2 and partly because some of the libraries discussed in this document are not built in to the compiler. In what follows we assume that the host environment is set up to compile C programs as described in [2].

In order to use the experimental libraries described in section 4, you must tell the mod68 compiler where to look for the definition modules. To do this, put the command

```
set MODPATH :/usr/bfly/UR/modula2/defs
```

into your .login file. Then, in your makefile, insert the following declarations ahead of any dependencies:
XLNK68LIBS = $x/local/libmod.a $x/local/libexpmod.a

.SUFFIXES: .068 .mod

.mod.068 :
mod68 -c $*.mod

This will tell xmake how to compile a Modula-2 program into an object file and how to find the libraries for the runtime environment. A sample Makefile is stored in /usr/bfly/UR/examples/modula2.

The compiler performs more or less as advertised in the VAX documentation: see man mod for details. Note however that the Powell compiler is only used to generate pcode, so some of the switches described in the manual are not relevant. Those that should work are:

- sx Use standard conventions for cardinal/integer compatibility, reserved word case and other Modula-2 rules the Powell compiler does not normally enforce. See the manual for legitimate values of x.
- C Generate runtime checks for array subscripts and pointer references.
- u Ignore case of reserved words and identifiers on input.

Note that you cannot use the -g and -pg switches, since the debuggers and profilers available on the Butterfly are not compatible with their UNIX counterparts. This may also be changed in later versions of Chrysalis.

One major weakness of the Butterfly compiler is that there is no support for floating point (types REAL and LONGREAL). The problem is that the Butterfly and Sun runtime libraries use different representations for floating point numbers. This will be fixed if we develop a Butterfly-specific version of the code generator.

A list of current known bugs and deficiencies is kept in /usr/bfly/UR/modula2/README.


The Butterfly runtime environment for Modula-2 includes most of what is available on the VAX. Appendix A contains definition modules for those modules which have changed significantly. The unix and signals modules are missing, of course, and some modules (particularly io) are not fully implemented. The math module is missing because of the lack of floating point support in general. Unfortunately the definition modules for io, system, and the other 'built-in' modules are hard-coded into the front end of the compiler. This means that if you use one of the unsupported functions, your mistake will not be caught until link time. The current list of known discrepancies is as
XLNK68LIBS = $x/local/libmod.a $x/local/libexpmod.a

.SUFFIXES: .068 .mod

.mod.068 :
    mod68 -c $*.mod

This will tell xmake how to compile a Modula-2 program into an object file and how to find the libraries for the runtime environment. A sample Makefile is stored in /usr/bfly/UR/examples/modula2.

The compiler performs more or less as advertised in the VAX documentation: see man mod for details. Note however that the Powell compiler is only used to generate pcode, so some of the switches described in the manual are not relevant. Those that should work are:

-sx Use standard conventions for cardinal/integer compatibility, reserved word case and other Modula-2 rules the Powell compiler does not normally enforce. See the manual for legitimate values of x.

-C Generate runtime checks for array subscripts and pointer references.

-u Ignore case of reserved words and identifiers on input.

Note that you cannot use the -g and -pg switches, since the debuggers and profilers available on the Butterfly are not compatible with their UNIX counterparts. This may also be changed in later versions of Chrysalis.

One major weakness of the Butterfly compiler is that there is no support for floating point (types REAL and LONGREAL). The problem is that the Butterfly and Sun runtime libraries use different representations for floating point numbers. This will be fixed if we develop a Butterfly-specific version of the code generator.

A list of current known bugs and deficiencies is kept in /usr/bfly/UR/modula2/README.


The Butterfly runtime environment for Modula-2 includes most of what is available on the VAX. Appendix A contains definition modules for those modules which have changed significantly. The unix and signals modules are missing, of course, and some modules (particularly io) are not fully implemented. The math module is missing because of the lack of floating point support in general. Unfortunately the definition modules for io, system, and the other 'built-in' modules are hard-coded into the front end of the compiler. This means that if you use one of the unsupported functions, your mistake will not be caught until link time. The current list of known discrepancies is as
follows.

3.1. System

System.iotransfer as described by Wirth is not available on either the Butterfly or the VAX. The system.CPUTime function returns the value of the local Butterfly node's real-time clock, rather than the elapsed CPU time since the process was started. A special note about the coroutine support (system.transfer and system.NewProcess): These routines work exactly as on the VAX, i.e., they set up Modula-2 processes running in the address space of the parent Chrysalis process. The processes created by these calls do not run in parallel on remote nodes. You can create parallel processes using the Butterfly specific libraries described in the next section.

3.2. Io

Functions dealing with files (Open and Close) are not supported, since Chrysalis has no file system. This means that you are limited to the files the system gives you (i.e., input, output, and terminal). The binary input and output functions Readb and Writeb are not supported.

3.3. Memory and Storage

Because of quirks in the Chrysalis memory object management system, requests for more than 64Kbytes of memory will always fail. (This applies to New and Dispose as well as Allocate and Deallocate.) This limitation should be fixed in Chrysalis 2.3, which will support extended memory objects.

4. Butterfly-Specific libraries

One of the most unpleasant features of the Butterfly is the amount of code required to make use of its potential parallelism. This is because the C runtime library gives the programmer a very flexible but very low-level view of the machine. It seems clear that a straightforward interface to the Chrysalis library is not what we want for Modula-2. This section describes a preliminary set of services at what I believe to be a more appropriate level of abstraction. We stress their experimental nature; suggestions for modifications or extensions are invited. See Appendix B for definition modules.

4.1. Processes

The Processes module allows you to run other processes, to freeze and thaw other processes, and to sleep for a specified number of milliseconds. It also virtualizes node numbers, so that the programmer need not be concerned with how the Butterfly is configured (i.e., which nodes are up and what their switch addresses are). This is intended to replace the For_All_Nodes macro used under Chrysalis by C programs.
4.2. Events

This module is a fairly straight translation of the Chrysalis calls related to event handling. Procedures exist to create, destroy, reset, post, and wait or poll for events. The Chrysalis multiple-wait call Mwait() is not supported, since its argument (a null-terminated array) is alien to the Modula-2 style.

4.3. SharedMem

This module creates a contiguous block of globally shared memory. The amount allocated depends on the state of the Chrysalis memory manager at initialization time; the module tries to allocate 64K per node. This module exports no functions, but does provide a pointer to the shared block and an indication of its size. Note that any process that imports SharedMem must have enough SARs to map in the shared block. On large Butterfly configurations this implies that there can be only one such process per node.

4.4. SharedHeap

The SharedHeap module uses SharedMem to create a block of shared memory as described above: it then manages it as a concurrently accessible heap, exporting functions Allocate and Deallocate. The interface is thus identical to the standard Storage module, but pointers to storage allocated by SharedHeap can be safely passed to processes on remote nodes (provided that they also import SharedHeap).

4.5. FetchAndPhi

Work done on the NYU Ultracomputer project [3] has shown that a class of atomic Fetch-and-Operate instructions make it possible to write code for many classical parallel programming problems without explicit critical sections. The Butterfly supports these operations on 16-bit integers; support for 32-bit operands has been promised by BBN. The FetchAndPhi module provides access to these special operations from within Modula-2. Using them is a bit tricky because of the lack of a 16-bit data type in Modula-2. The FetchAndPhi functions have been defined on a type called "short", which is the integer subrange [0..65535]. This means that many published Fetch-and-Phi algorithms may have to be modified to work correctly. In particular, tests for negative values and comparisons must be examined closely.

5. Extending the Butterfly-Specific Libraries

The Butterfly-specific modules described in the previous section are clearly only a beginning. Extending them is not hard, though experience with C and Chrysalis is strongly recommended. This section is intended to provide information and advice about how to do this. The Powell compiler documentation is also recommended reading, particularly "Using Modula-2 with Unix C and Berkeley Pascal" [4].
5.1. Calling C functions from Modula-2

There are two ways to call C functions from inside Modula-2. To understand them, the programmer must know a little bit about how the Powell compiler and the VAX and Butterfly C compilers treat identifiers. By convention, all UNIX compilers prepend an underscore to every public variable. For example, the C statement "foo = 1" would be compiled to something like "movl 1, _foo" by a UNIX 68000 compiler, and an entry would be made in the linker symbol table associating the string "_foo" with some address. Unfortunately the Butterfly C compiler, cc68, does not adhere to the prepended underscore convention; the statement "int foo" creates a symbol table entry called "foo". In Modula-2, the situation is complicated by the fact that there may be other "foo" variables in other modules. They must be given different names so that the linker won't attempt to give them the same address. The Powell compiler handles this by prepending the module name to the name of the variable. For example, if foo is exported by module baz it will appear in the symbol table as "_baz_foo". The module main body (ie. its runtime initialization routine, if it is not the main program) will be referenced as "_baz_init" (note the two underscores).

The Powell compiler provides a standard convention for calling C functions from user code. The programmer must provide a definition module using the attribute '@externar' after the keyword 'procedure' or before the list of variables to be imported from a C program. What this does is suppress the attachment of the module name to the identifier so that, in the example above, the symbol table entry is "_foo" rather than "_baz_foo". The prepended underscore still appears as per UNIX convention.

Suppose definition module "baz.def" references a function "foo", as described above. If the @external keyword is present, the C implementation for the Butterfly should contain a definition of the form 'int _foo();': if not, it should be 'int _baz_foo();'. Note again the leading underscores; on the VAX these would not be necessary. There must also be a routine called _baz_init() to resolve the initialization reference, even if no initialization is actually needed. If more than one module in the program imports baz, the initialization routine will be called more than once. The initialization routine should protect itself if this is a problem. Within a single process this can be done by declaring a static variable initialized to 0. The initialization routine tests this variable; if it is zero it sets it to one and continues, otherwise it exits.

Appendix C contains a sample definition module and C implementation.

5.2. Problem Areas

Initialization - The method of handling multiple initializations described above works well for single processes. It may not work for multiple processes, since Chrysalis gives each process a separate address space. This only matters if the initializer alters the global
state (e.g. by creating named objects). Trouble can arise as follows: suppose module **foo** has a critical initialization step which checks for a named OID, maps it in if it exists, or creates and names it if it doesn’t. The main program imports **baz** and **foo**. At runtime, the main program will call **_baz_init()** and **_foo_init()** in that order. Suppose that **_baz_init()** uses the **Process** module to create children on other nodes, and that those children also import **foo**. Their calls to **_foo_init()** may overlap each other’s or the main program’s. In either case there is danger that two or more processes will find the name unbound and proceed with the initialization. This will cause either the object creation or the name binding to fail. The only guaranteed way to avoid this problem that we know of is to a) forbid process creation by initialization modules and b) be sure that the main program imports every module that is imported by processes it wants to spawn. This latter restriction is particularly unwieldy, since it requires us to violate the opacity of the spawned processes. The former restriction is not as unpleasant but does require programmer discipline.

**Argument Passing** - Many Chrysalis calls (e.g. MakeProcess) take structures as arguments. No effort has been made to make these isomorphic to Modula-2 records, so interfacing to these routines is tricky. It can be done by being rigorous about information hiding (i.e., making all types opaque and providing functions to read and set those fields the user needs access to). This works but carries an obvious performance penalty. The same strategy can be used for the large number of Chrysalis calls that are implemented as macros.

**Error handling** - Chrysalis provides a rather sophisticated set of calls (**catch**, **throw** and associated functions) for handling unexpected situations from bus errors to queue overflow and underflow. It makes heavy use of them internally and expects the user to do so as well. Modula-2 has no such mechanism, and adding one would be a significant extension of the language. One of the most important considerations in designing a Modula-2/Chrysalis utility is that of how throws from the underlying Chrysalis routines should be handled.

In cases where an exception really indicates a serious error (e.g. a bad memory reference), it is reasonable to print a message and halt. The message should be phrased in terms of the abstraction presented by the associated routine, rather than in terms of the underlying implementation. A message like "invalid dual queue handle" is no help to a user who thinks he is writing to a file, and doesn’t know that dual queues are involved. Letting the exception propagate out to the default handler in the shell is a bad idea, since Chrysalis messages tend to be cryptic.

Unfortunately Chrysalis uses **throw** and **catch** for routine flow of control as well as for fatal errors, most notably in the case of dual queue overflows and multiple event postings.
This type of error should probably return an error code rather than killing the process that encounters it, since the user may be able to ignore or recover from it.

Acknowledgements

The Butterfly Modula-2 compiler is the product of many people's labors. First among these is Mike Powell of the Digital Equipment Corporation, who wrote the original VAX compiler and made it available to us. Tom LeBlanc of the University of Rochester systems faculty conceived the project, provided advice and encouragement and read many drafts of this document. Neal Gafter wrote the 68000 code generator with assistance from Ken Yap, who provided the 'mod68' driver. Liudvikas Bukys contributed his extensive experience with Chrysalis.

References

definition module io;

(* This is a dummy definition module to indicate what is in the built-in *)
(* io module. Note that case is ignored in built-in identifiers *)

type File; (* Open file variable type *)

var

(* Standard files connected to Chrysalis standard input, output, and error *)
input, output, terminal: File:

procedure Readf(f: File; format: (* constant *) array of Char: var arg1 : ArgTypel; var arg2 : ArgType2; ...): integer:
(* read a list of values from a file according to a format string *)
(* f: an open file; format: constant string format (like Unix scanf) *)
(* argn: variable for corresponding format item, type must match *)
(* return value: number of values read, <0 for end of file *)

procedure Writef(f: File; format : (* constant *) array of Char: arg1 : ArgTypel; arg2 : ArgType2: ...):
(* write a list of values to a file according to a format string *)
(* f : an open file; format: constant string format (like Unix printf) *)
(* argn: value for corresponding format item, type must match *)

procedure Readc(f: File; var c : Char) : integer;
(* read the next character from the file *)
(* f: an open file; c: variable to read next char into *)
(* return value: >= 0 if read OK, <0 if end of file *)

procedure Writec(f: File; c : Char);
(* write a character to a file *)
(* f: an open file; c: value for next char to write *)

procedure SReadf(s : array of Char; format: (* constant *) array of Char; var arg1 : ArgTypel; var arg2 : ArgType2; ...): integer:
(* read a list of values from a string according to a format string *)
(* s : a string; format : constant string format (like Unix scanf) *)
(* argn : variable for corresponding format item, type must match *)
(* return value : number of values read *)

procedure SWritef(s : array of Char; format : (* constant *) array of Char;
arg1 : ArgType1; arg2 : ArgType2; ...);
(* write a list of values to a string according to a format string *)
(* s : a string; format : constant string format (like Unix printf) *)
(* argn : value for corresponding format item, type must match *)

end io.
definition module Events:

export Event, MakeEvent, DestroyEvent, PostEvent, WaitEvent, GetEvent, 
WaitAnyEvent, GetAnyEvent, EventData;

(****************************************************************
(* This module provides access to a limited version of the
(* Chrysalis event mechanism. Routines are provided which
(* allow you to
(* - create an event object (MakeEvent)
(* - delete " " (DestroyEvent)
(* - post an event with data (PostEvent)
(* - reset an event so that it can
(*  be posted again (ResetEvent)
(* - Receive an event (blocking) (WaitEvent)
(* - Receive an event (non-blocking) (GetEvent)
(* - Receive any event (blocking) (WaitAnyEvent)
(* - Receive any event (non-blocking) (GetAnyEvent)
(* - read the data posted with an event (EventData)
(*
(****************************************************************)

procedure MakeEvent(VAR e : Event) : boolean;
(* Creates a new event block. Event creation can fail; this
(* procedure returns True on success, False on failure. *)

procedure DestroyEvent(e : Event):
(* Deletes an existing event. The delete will fail silently
(* if the event does not exist, or if a different process
(* created it.*)

procedure PostEvent(e : Event; data : Cardinal);
(* Posts the event with the data. It is an error to post an
(*) event which has been posted already. *)

procedure ResetEvent(e : Event);
(* Reset the event so that it can be posted again. *)

procedure WaitEvent(e : Event);
(* Causes the calling process to sleep until event e is posted.
(* It is an error to Wait for an event that you did not create. *)

procedure WaitAnyEvent(VAR e : Event):
(* Causes the caller to sleep until one of its events is posted.*)

procedure GetAnyEvent(VAR e : Event) : boolean;
(* If an event created by this process has been posted, this
(* procedure returns True and sets e to the posted event;
(* otherwise it returns False.*)

procedure GetEvent(e : Event) : boolean:
(* returns True if this event has been posted. *)

procedure EventData(e : Event) : Cardinal;
(* Returns the value of the data posted with event e. *)

end Events.
definition module FetchAndPhi;

from System import Address;

export short, FetchAndADD, FetchAndAND, FetchAndOR,
            StorePtr, ReadPtr, StoreInt, ReadInt, StoreCard, ReadCard;

 teardown {
  (* This module provides atomic fetch-and-phi operations in
  (* the style of the NYU Ultracomputer. Because of microcode
  (* limitations in the Butterfly, these operations cannot be
  (* done on full words. They apply only to the data type
  (* "short".
  (* Also included are atomic 32-bit reads and writes, since
  (* the butterfly does not guarantee atomicity.
  (* Separate versions are provided for pointers and integers;
  (* other types can be handled with casts.
  (***********************************************)

type short = [0..65535];

  (* FetchAndPHI(x,y) atomically sets x := PHI(x,y) and returns
  (* the previous value of PHI. Variables of type short can
  (* be treated as Cardinal for most purposes. Beware of using
  (* them as addresses, though!  *)

procedure FetchAndADD(VAR x : short; y : short) : short;

procedure FetchAndAND(VAR x : short; y : short) : short;

procedure FetchAndOR(VAR x : short; y : short) : short;

  (* Store<type>(x, y) atomically stores value y into variable
  (* x. It is rather slow, since it uses the pnc block transfer
  (* mechanism to guarantee atomicity.         *)

procedure StorePtr(VAR x : Address; y : Address);
procedure ReadPtr(VAR x : Address) : Address;

procedure StoreInt(VAR i : Integer; i : Integer);

procedure ReadInt(VAR x : Integer) : Integer;

procedure StoreCard(VAR x : Cardinal; i : Cardinal);

procedure ReadCard(VAR x : Cardinal) : Cardinal;

end FetchAndPhi.
definition module Processes;

export Process, NullProcess, MyProcessID, ProcNode, NumberOfNodes,
   NewProcess, FreezeProcess, ThawProcess, Sleep;

(* This module provides access to a few selected operating
(* system services relating to process creation and management.
(* Routines are provided which allow you to
(* - run a program with arguments on some node
(* - Suspend yourself or another process
(* - Awaken a suspended process
(* - Sleep for a specified interval
(*

(type Process = CARDINAL:

var
   NullProcess : Process;
   MyProcessID : Process;
   (* Set at initialization time to the ID of this process. *)
   ProcNode : CARDINAL;
   (* Set at initialization time to the node this process is on. *)
   NumberOfNodes : CARDINAL;
   (* Set at initialization time to the number of nodes in
   (* the system. The logical node for a process spawn is
   (* always between 0 and (NumberOfNodes - 1). *)

procedure NewProcess(Name, Args : array of CHAR;
   Node, NoSARs : CARDINAL) : Process;
   (* Creates the specified program on the specified logical Node
   (* with appropriate args and number of SARs. If NoSARs is 0,
   (* you will get the Chrysalis default, currently 16.
   (* NewProcess returns NullProcess on failure. *)

procedure FreezeProcess(p : Process);
   (* Causes p to be suspended. P can be the current process. *)

}
procedure ThawProcess(p : Process);
(* Causes p to be made runnable. P need not be frozen; if so
(* the Thaw will be lost. *)

procedure Sleep(millisec : CARDINAL);
(* causes you to sleep for the specified interval, possibly
(* minus a few hundred microseconds. *)

end Processes.
definition module SharedMem:
(***********************************************************************
(* This module establishes a contiguous block of 
(* globally shared memory. The amount allocated depends on 
(* the state of the Chrysalis memory manager at initialization 
(* time; the system tries to allocate 64K per node. The 
(* amount allocated and its location are exported. 
(***********************************************************************

from System import ADDRESS;
export Start, Size;

var Size : CARDINAL;
(* Set at initialization time to the amount of memory in the block. *)

var Start : ADDRESS;
(* Where in the running process' memory space the shared block begins.*)

end SharedMem.

definition module SharedHeap:
(***********************************************************************
(* This module creates and manages a global, concurrently 
(* accessible heap. 
(***********************************************************************

from System import ADDRESS;

procedure Allocate(VAR a : ADDRESS; size : CARDINAL);
(* Allocates size bytes from the global heap, returning a pointer in var parameter a. *)

procedure DeAllocate(VAR a : ADDRESS; size : CARDINAL);
(* Frees size bytes from the global heap starting at address a. *)
(* The storage should have been allocated by Allocate. *)

end SharedHeap.
Appendix C

Sample C implementation of a Modula-2 Definition Module

As an example we present a module which provides access to the Chrysalis block transfer function Do_bt(). Note that we cannot use the '@external' directive, because Do_bt is a macro.

```
definition module BlockTrans;
(* Access to the pnc block transfer operation *)

from system import ADDRESS;
export BlockTransfer;

procedure BlockTransfer(from, to : ADDRESS; nbytes : CARDINAL);
end BlockTrans.
```

The C implementation is straightforward:

```
#include <chrys.h>
#include <stdio.h>

void _BlockTrans_init() {} /* no initialization required */

void _BlockTrans_BlockTransfer(from, to, size)
    char *from, *to;
    int size:
{
    catch
        Do_bt(from, to, size);
    onthrow
        when(TRUE) { /* only error we can get is a bus error */
            printf("BlockTrans : Bus Error :");
            catch_prt(stderr, throwlocation, throwcode,
                throwtext, throwvalue);
            fflush(stdout); fflush(stderr); exit(1);
        }
    endcatch;
}
```