# Harmony Application Notes: Release 3

National Research Council Canada
Conseil national de recherches Canada

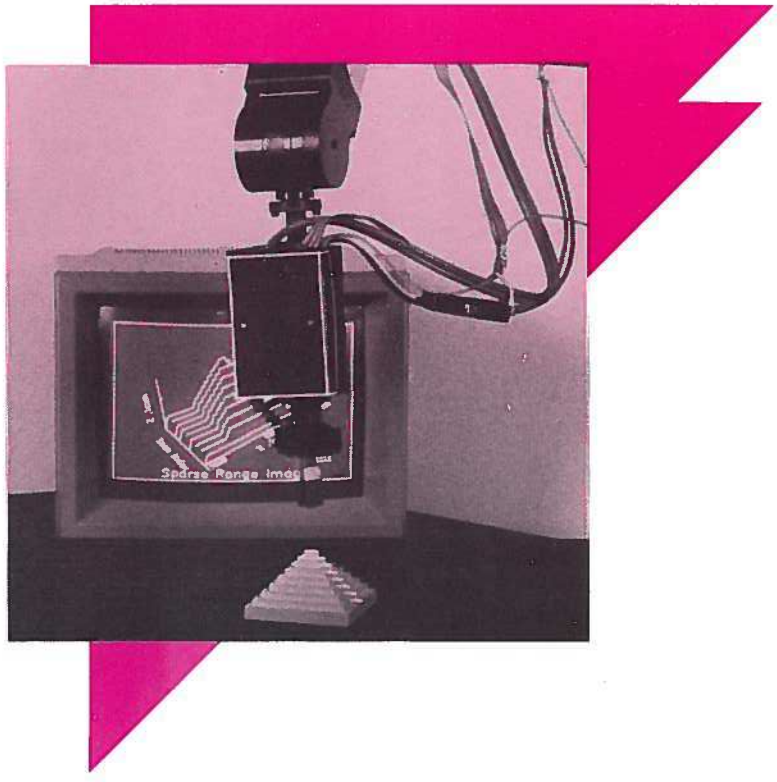
Division of Electrical Engineering
Division de génie électrique

ERA-378

# *Harmony Application Notes (Release 3.0)*

Edited by
D.A. Stewart and S.A. MacKay

February 1989




Canada

NRCC No. 30082

Additional copies are available free of charge from:

Editorial Office, Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R6

Des exemplaires supplémentaires peuvent être obtenus gratuitement à l'adresse suivante :

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R6

# HARMONY APPLICATION NOTES

(Release 3.0)

Edited by

Darlene A. Stewart and Stephen A. MacKay

ERA-378

NRCC No. 30082

FEBRUARY 1989

# TABLE OF CONTENTS

## Abstract

This document contains a series of application notes for Release 3.0 of Harmony,[1] a multitasking multiprocessor operating system for realtime control. A range of topics of particular interest to Harmony users is covered; the depth varies from conceptual level discussions, explaining why certain design decisions were made, to practical "how-to" usage descriptions. These notes supplement the material presented in Using the Harmony Operating System (W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein, Using the Harmony Operating System, Release 3.0, NRC/ERA-377, National Research Council of Canada, Ottawa, Ontario, February 1989.).

## Résumé

Le présent document contient une série de notes d'application de la version 3.0 de Harmony,[1] un système d'exploitation multiprocesseur multitâche pour les opérations en temps réel. Toute une gamme de sujets d'un intérêt particulier pour les utilisateurs du système Harmony sont abordés, allant de discussions sur le niveau concepteul, expliquant pourquoi certaines décisions de conception ont été prises, à des descriptions pratiques de l'utilisation du système. Ces notes viennent s'ajouter au matériel présenté dans la publication s'intitulant Using the Harmony Operating System (W.M. Gentleman, S.A. MacKay, D.A. Stewart et M. Wein, Using the Harmony Operating System, Release 3.0, NRC/ERA-377, Conseil national de recherches Canada, Ottawa (Ontario), février 1989.).

[1] Mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée. / Marque déposée réservée à l'usage exclusif de sa Majesté du chef du Canada par Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée.


# Application Note n001

**Title:** **Debugging Harmony Programs**

| **Revisions:** | | | |
|---|---|---|---|
| | 1986-08-05 | Darlene A. Stewart | |
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1988-09-09 | Darlene A. Stewart | (for Release 3.0) |
| | 1988-09-28 | Darlene A. Stewart | (for b035) |
| | 1989-01-10 | Darlene A. Stewart | (for b099) |
| | 1989-01-23 | Darlene A. Stewart | (more Release 3.0) |

**See Also:** W. M. Gentleman and D. A. Stewart, Debugging Multitask Programs, NRC/ERB-1008, National Research Council of Canada, Ottawa, Ontario, September 1987.

---

The Harmony debugger provides some services through its program interface and provides others through an interactive user interface. The interactive interface is invoked when a breakpoint is encountered, when an exception occurs, or when an abort occurs. The task in which the breakpoint, exception, or abort occurred is prevented from continuing (its normal) execution during the interactive dialogue with the user; other tasks, however, do continue their normal execution. The user can then investigate the situation using the interactive debug commands. Upon completion of his investigation, the user can restart the stopped task, thus terminating the interactive dialogue.

Breakpoints provide a mechanism for the programmer/user to temporarily stop an executing task at specified points to examine/modify its state and then allow it to continue with its normal course of execution. Two types of breakpoints are supported by the Harmony debugger: compiled breakpoints and plantable breakpoints. Breakpoints can be compiled in as subroutine calls using:

```
_Breakpoint( msg )
        char  *msg;    /* message displayed on debug device */
```

or they can be planted dynamically using the breakpoint (b) commands in the interactive debugger. Normally, the user program calls _Breakpoint during its initialization sequence to allow the user to plant additional breakpoints dynamically.

However, before any breakpoints can be planted dynamically, the debug control server must be created and initialized. The programming details regarding the creation and initialization of the debug control server are described below in the *Debug Control Server* section of this document.

The programmer can signal an abort to indicate an irrecoverable error with the subroutine call:

```
_Abort( msg )
        char  *msg;    /* message displayed on debug device */
```

Exception conditions, on the other hand, are automatically detected by the system. These include address errors, bus errors, illegal instructions, divide by zero, etc.

When one of these conditions (i.e., breakpoint, exception, abort) occurs, the debugger displays various information about what has happened including the ID of the task invoking the debugger, the reason, the current values of the stack pointer (sp), the stack frame pointer (a6), and the program counter (return pc), other relevant information for some types of exceptions, a register dump for planted breakpoints and exceptions, and a call traceback for the task causing the debugger to be invoked. Once this information is displayed, the prompt for the interactive debugger (debug>) is displayed, and the user can engage in an interactive dialogue with the debugger. The commands available in the interactive debugger are outlined in the *Interactive Debug Commands* section below. Additional debug facilities are described elsewhere in this document.

### Interactive Debug Commands

| | |
|---|---|
| **r** | Resume execution of the stopped task. |
| **R** | Restart Harmony (only for Atari ST). |
| **bs** [<pnumber>:]<addr> <task-id> | Set (plant) a breakpoint in a task (0 == all tasks). |
| **bc** <bp-number> | Clear specified planted breakpoint. |
| **bl** | List all planted breakpoints. |
| **s**[.<size>] [<pnumber>:]<addr> | Show contents of a location (with checking). |
| **S**[.<size>] [<pnumber>:]<addr> | Show contents of a location (without checking). |
| **<cr>** | Next location and show contents. |
| **n**[.<size>] | Next location and show contents (increment by old size). |
| **+**[.<size>] | Next location and show contents (increment by old size). |
| **^**[.<size>] | Previous location and show contents (decrement by new size). |
| **-**[.<size>] | Previous location and show contents (decrement by new size). |
| **.**[.<size>] | Same location and show contents. |
| **=**[.<size>] <value> | Assign a value to the current location. |
| **C** <global-index> | Create a task. |
| **D** <task-id> | Destroy a task. |
| **l** <pnumber> | List all the tasks on a processor. |
| **t** <task-id> | Display a task descriptor (TD structure). |
| **c** <task-id> | Display the call traceback for a task. |
| **m** <task-id> | Display the memory resources for a task. |
| **pb** [<pnumber>:]<addr> | Display a storage pool block descriptor. |

| | |
|---|---|
| **pb** 0 | Display the 1st storage pool descriptor on processor 0. |
| **pn** | Display the descriptor of the next storage pool block. |
| **pm** [<pnumber>:]<pool-addr> | Display a map of a storage pool. |
| **pm** 0 | Display the map of the storage pool on processor 0. |
| **u** [<pnumber>:]<addr> | Display a user connection block (UCB structure). |
| **xs** [<pnumber>:]<addr> | Display an XTRA_STREAMIO structure. |
| **vg** <pnumber> <#-bits> | Get (allocate) a use bit vector on specified processor. |
| **vc** <pnumber> | Clear the use bit vector for the specified processor. |
| **vd** <pnumber> | Display the use bit vector for the specified processor. |
| **w** | Display why the debugger was entered, including an exception/breakpoint/abort dump and call traceback. |

### Notes

Numbers can be entered following the C convention (i.e., 0 implies octal, 0x implies hexadecimal, otherwise decimal).

<size> is the number of bytes, which can be 1, 2, or 4. If a size is not specified, then the previous specified size is used. The initial size defaults to 4 bytes.

No whitespace is permitted around .'s and :'s in commands to the debugger.

For the Motorola 680x0 family of processors, register a6 is the stack frame pointer. For the Consulair Mac C compiler, the arguments are usually passed in the data registers (starting with register d0), but are stored on the stack by one of the first few instructions in the called routine; they are stored on the stack at negative offsets from register a6. (Exceptions to this rule include floating point arguments and arguments passed to a function that accepts a variable number of arguments. Consult the Consulair documentation for details on how such arguments are passed.) Likewise, all local variables are stored at negative offsets from register a6. The easiest way to determine the offset for a variable is to look at the disassembled code using Examine.

A call traceback for a task may contain nonsense if the task has an opportunity to run while the debugger is walking its stack. This problem will not hang the debugger because the stack is walked in a careful manner; it will merely produce erroneous results. (This problem has rarely been observed in practice.)

For the normal version of the debugger, debugger I/O is to the device whose server has registered the name "DEFAULT:". Normally, this is an alternate name for "TEXT_TERMINAL:".

The use of certain commands, such as the plantable breakpoint (b) commands and the use bit vector (v) commands, requires the presence of a debug control server and related tasks.

## Log Gossip

A very useful non-interactive debugging tool is Harmony's log gossip facility. The programmer can display messages on the debug device using the subroutine call:

```
_Log_gossip( msg )
        char  *msg;     /* message displayed on debug device */
```

_Log_Gossip sends the message to the Gossip task, which displays it on the debug device along with the id of the invoking task and then allows the task to continue execution without entering the interactive debugger. This facility has the advantage over direct I/O (i.e., using _Printf) that the debug messages from several tasks executing in parallel do not become intermingled and garbled. Note that the _Sprintf function can be used to construct a message that is then passed to _Log_gossip.

**Use Bits**

Provision has been made for a vector of use bits on each processor. Use bits can be used to denote important events or states in a program, the value of each bit indicating whether or not the corresponding event or state has occurred. Use bits are a very useful debugging aid, especially for tasks with strict time constraints.

The individual bits of a use bit vector can be set, reset, flipped, or tested using a very efficient set of functions. Each processor has its own use bit vector. A task can manipulate only the use bits on its own processor. The functions for manipulating use bits include:

```
_Set_use_bit( n )
        uint_32  n;

_Reset_use_bit( n )
        uint_32  n;

_Flip_use_bit( n )
        uint_32  n;

boolean _On_use_bit( n )
        uint_32  n;
```

For the Motorola 680x0 family, the use bit manipulation functions are implemented by very efficient inline C macros. When the use bit vector exists, use bit access time (e.g., _Set_use_bit) on a 10 Mhz MC68000 is estimated to be 8–14 µs depending on whether the bit number argument is a constant or a variable. If the vector does not exist, the same call is estimated to take 4–5 µs.

The use bit vectors can be allocated, cleared, and examined through the debug control server, either interactively by the user debugging the program or under program control. The debugger user interface in Gossip includes the following use bit vector commands:

```
vg <processor-#> <#-bits>
vc <processor-#>
vd <processor-#>
```

These commands get (allocate), clear, and display the use bit vector on the specified processor, respectively.

The debug control server user library functions providing the corresponding program interface include:

```
uint_32 _Get_use_bit_vector( dc_ucb, pnumber, num_bits )
        struct UCB  *dc_ucb;
        uint_32     pnumber;
        uint_32     num_bits;

boolean _Clr_use_bit_vector( dc_ucb, pnumber )
        struct UCB  *dc_ucb;
        uint_32     pnumber;

_Put_use_bit_vector( dc_ucb, pnumber )
        struct UCB  *dc_ucb;
        uint_32     pnumber;

uint_32 _Rd_use_bit_vector( dc_ucb, pnumber, buf, buf_size )
        struct UCB  *dc_ucb;
        uint_32     pnumber;
        uchar       *buf;
        uint_32     buf_size;
```

_Get_use_bit_vector allocates a use bit vector, returning the number of bits in the bit vector. The use bit vector for a processor may be allocated only once; subsequent calls do not allocate a new use bit vector. _Clr_use_bit_vector provides an atomic clear of the use bit vector; interrupts are disabled during the clear operation. _Rd_use_bit_vector reads the use bit vector and copies it into the buffer pointed to by buf, returning the number of bits copied; the use bits are packed into buf with use bit 0 being the high order bit of the first byte of buf.

The use bit manipulation functions (e.g., _Set_use_bit) are safe functions. They simply do nothing when called for a non-existent use bit (e.g., if the use bit vector has not been allocated). When there is no use bit vector the overhead for such calls is very low (just a few microseconds). Thus, it is reasonable to leave use bit manipulation calls permanently in the code throughout the development cycle or perhaps even in production code. These calls can then be enabled for debugging purposes simply by allocating a use bit vector from the debugger.

### Interpreting Call Tracebacks and Stack Frames

The discussion in this section is quite specific to the Motorola 680x0 family of processors.

One of the pieces of information that is displayed when a breakpoint, exception, or abort is encountered during execution of a task is a call traceback for that task. Alternatively, the user may request from the interactive debugger a call traceback display for any task executing on any processor. The call traceback allows one to quickly determine the current point of execution for a task and how it got there. It also provides information for locating the calling parameters and local variables for a function invocation.

A sample call traceback for a task might look something like:

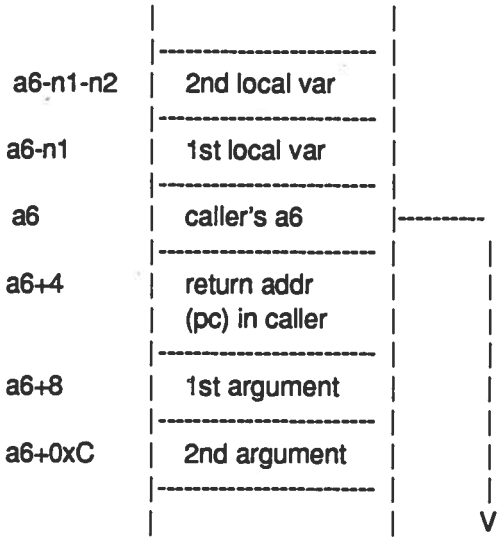
```
Call traceback for task 0x82800009:
                    returns to: 0x26DC
A6: 0x12A14         returns to: 0xF34
```

```
A6: 0x12A34     returns to: 0xB1B4
exception       returns to: 0xA9E
A6: 0x12B00     returns to: 0x27B2
```

In this example, task 0x82800009 is currently blocked, but when it becomes unblocked, it will continue execution at location 0x26DC. Using the Examine tool (a disassembler) on the executable image, we find out that 0x26DC is _Send+0x4A; i.e., the program counter will be at byte offset 0x4A from the start of the function _Send. (Note that we obtain this result simply by typing the absolute address as a command to Examine. If we are interested in what code corresponds to this address, we can do so by disassembling the _Send function.)

Associated with the execution of a function is a stack frame on the execution stack for the task. This stack frame is pointed to by the stack frame pointer (register a6). When a function is called, the stack frame pointer for the caller is pushed on the stack along with the return address in the calling function, a new stack frame is allocated for the called function, and the stack frame pointer is updated. The information that can be obtained if one knows the stack frame pointer for a particular invocation of a function includes: the stack frame pointer for the calling function (at a6), the return address in the calling function (at a6+4), the arguments to the called function (at negative or positive offsets from a6, depending on compiler), and the local variables for the called function (at negative offsets from a6).

A typical stack frame for many compilers (Mac C differs though):




In the call traceback example above, the stack frame pointer for _Send is 0x12A14 and the return address in the calling function is 0xF34 (_Buserr+0x2E). The stack frame pointer for _Buserr is 0x12A34 and the return address in the calling function is 0xB1B4 (_FlBuserr+0x1C). These stack frames are all associated with the handling of a bus error exception. The next line tells us that the exception (_FlBuserr) returns to address 0xA9E (_Child+0x7E). Thus the bus error actually occurred at the instruction just before this address in _Child, i.e., at location 0xA9C (_Child+0x7C). The stack frame pointer for _Child is 0x12B00, and _Child, being the root function of the task, returns to location 0x27B2 (_Suicide), if in fact it ever returns. We can index negatively in memory from location 0x12B00 to locate the local variables for _Child. Because _Child is the root function for the task, it has no arguments, but if it did, they could be located too.

*SPECIAL NOTE:* A call traceback for a task may contain nonsense if the task has an opportunity to run while the debugger is walking its stack. This problem will not hang the debugger because the stack is walked in a careful manner; it will merely produce erroneous results. (This problem has rarely been observed in practice.)

### The Debug Control Server

In order to be able to plant breakpoints dynamically (using the bs command) or to allocate/manipulate a use bit vector (using the v commands), the user's program must define task templates for the debug control server and related tasks, and it must create (and initialize) the debug control server; it should also create (and initialize) the clock server, although this is not essential to operation of the debug control server.

The following are sample task templates for the debug control server and related tasks on processor 0:

```
{ DBG_CONTROL, _Dbg_control, 800, 7, 0 },
{ DBG_SHADOW, _Dbg_shadow, 500, 7, 0 },
{ DBG_0AGENT, _Dbg_agent, 500, 5, 0 },
```

Task templates for debug agents to run on any other processors that will be debugged should be defined similarly:

```
{ DBG_1AGENT, _Dbg_agent, 500, 5, 0 }
{ DBG_2AGENT, _Dbg_agent, 500, 5, 0 }
```

Note that the manifest constants for the global indices (i.e., DBG_CONTROL, etc.) must be defined in the templates.h file.

The debug control server can be created in main (after the clock server has been created) as follows:

```
extern struct DC_INIT_REC Dbg_control_init;
   ...
_Server_create( DBG_CONTROL, (struct INIT_REC *)&Dbg_control_init );
```

where, for example, Dbg_control_init may be defined as:

```
struct DC_INIT_REC Dbg_control_init =
   {
      sizeof( struct DC_INIT_REC ), /* MSG_SIZE */
      DC_INIT,                      /* MSG_TYPE */
      0,                            /* no more init records */
      "DBG_CONTROL:",               /* dbg control server name */
      "CLOCK:",                     /* clock server name */
      DBG_SHADOW,                   /* debug shadow task index */
      DBG_0AGENT,                   /* processor 0 dbg agent index */
      DBG_1AGENT,                   /* processor 1 dbg agent index */
      DBG_2AGENT                    /* processor 2 dbg agent index */
   };
```

Note that debug agent tasks need only be defined for those processors containing code in which breakpoints might be placed.

### Hazards in Planting Breakpoints

Unfortunately the plantable breakpoint mechanism affects code, which can be shared among tasks. Even if one arranges that a time critical task is not stopped at the breakpoint, it can have its performance severely affected by the emulation of the breakpointed instruction. If the user wishes to place breakpoints in the code of some task and that code is shared with a time critical task, he is advised to duplicate the code to avoid the severe performance penalty induced by the presence of the breakpoint trap instructions.

Planted breakpoints violate stack bounds. The stack bounder, in addition to returning the minimum stack required for a task, also returns a larger result that takes into account the possibility of exception processing. This larger amount must be used in order to handle planted breakpoints safely.

There are a number of pieces of code where breakpoints cannot be handled properly. These include the routines _Send, _Td_service, _Block_signal_processor, _Block, _Signal_processor, _IP_int, _Add_ready, _Convert_to_td, _Enable, _Disable, any other piece of disabled code including all interrupt and exception handlers, and any code in ROM. Also, on the processor running the debugger, breakpoints cannot be planted in the debugger, the clock server (CLOCK:), the server responding to the name DEFAULT: and associated tasks, the connection routines, the stream I/O routines, and many kernel routines. What is worse, the debugger is unable to guard against the planting of breakpoints in these hazardous places. As a general rule, it is safe to plant breakpoints in one's own code, but it may be very dangerous to plant breakpoints in system code. (Normally planting breakpoints in one's own code prior to calling system code or after returning is adequate.)

There are some limitations on the specific instructions that can be breakpointed. First, it is the user's responsibility to determine that the breakpoint is in fact being placed at an instruction boundary; the results are disastrous if it is not. There may be some instruction types that should be avoided because the debugger has trouble emulating them correctly; for the Motorola MC68000 these (rarely used) instructions include: MOVE EA to SR, MOVE SR to EA, ANDI to SR, EORI to SR, and ORI to SR. Furthermore, breakpoints must not be placed on any instructions that are shorter in length than the breakpoint trap instruction; for the Motorola 680x0 family of processors there are no such instructions. The debugger contains no checks about the validity of breakpoint locations; the results are unpredictable if the user breaks the rules.

Finally, the user must remember to remove all breakpoints before resetting the Harmony target computer, unless he re-downloads before executing the program again.

### A Few Hints for Debugging Harmony Programs

The report *Debugging Multitask Programs* provides insight into debugging multitask programs in general. This section provides some additional pointers to make debugging Harmony programs, specifically, easier.

The best way to find out what is happening in a program, especially if it is *doing nothing*, is to set some breakpoints at strategic places to determine if the program is behaving as expected. During development, it is a good idea to call _Breakpoint (i.e., set a compiled breakpoint) near the start of main just after the DEFAULT: server has been created. It is also a good idea to configure a clock server and a

debug control server to permit the planting/clearing of breakpoints dynamically during program execution and to permit the use of use bit vectors. In this case, there should be a call to _Breakpoint immediately after these servers have been created to allow the planting of additional breakpoints and the allocation of use bit vectors. The precise addresses at which to place additional breakpoints can be determined by using the Examine tool to disassemble the code for the functions where breakpoints are to be placed. Function calls are easy to locate in the disassembly because they appear as jsr or bsr instructions with the symbolic function name displayed. Note that if any breakpoints are planted dynamically, they must be removed (using bc) before the reset button on the Harmony target computer is hit unless it is intended to redownload the code.

Once in the debugger (because of a breakpoint, exception, or abort), a good place to begin in tracking down a problem is by looking at what tasks exist on each processor and what their states are. This can be accomplished using the list tasks (l) command.

Common states for a task include:

| | |
|---|---|
| ready | • the task is either executing or is ready to execute but is waiting because another task is running. |
| send_blocked | • the task is blocked, having sent a message to another task, which has not yet received it. |
| reply_blocked | • the task is blocked waiting for a reply from the task that has received its message already. |
| rcv_blocked | • the task, having called _Receive, is blocked waiting for a message from any task. |
| rcv_specific_blocked | • the task, having called _Receive, is blocked waiting for a message from a specific task. |
| await_interrupt | • the task, having called _Await_interrupt, is waiting for a specific logical interrupt. |

There exist other task states, but they are rarely observed from the debugger.

One can find out more information about the state of a task by examining its task descriptor using the t command. This command takes the task id as an argument. The task descriptor provides such information as the ID of the task, its STATE, whether it is DYING, its CORRESPONDENT (i.e., the task to whom it has sent or from whom it has received or is waiting to receive a message), pointers to the REQUEST and REPLY message buffers, its current top of stack pointer (STACKPTR), a pointer to the STACK, a pointer to the task's MEM_RESOURCES (list of blocks allocated using _Getvec), the id of the task's FATHER, a pointer to the CONN_RESOURCES (list of UCBs) for the task, the current INPUT_STREAM UCB, the current OUTPUT_STREAM UCB, and the TASK_ERROR_CODE (which indicates a variety of errors listed in the sys.h file).

One common source of error is stack overflow. This can sometimes be detected by examining the STACKPTR field in the task descriptor (i.e., it should be >= STACK for the Motorola 680x0 family of processors). Other times the stack has shrunk back to be within bounds by the time it is examined, but the damage has already been done. This problem often does not show up until long after the corruption has occurred, often manifesting itself as a bus error or a bad storage pool. The best way to check for and to guard against stack overflow is to use the Bound tool to determine appropriate stack sizes for the tasks.

Another common error is writing beyond the end of a buffer or some other data structure. This error often manifests itself as a bus error or a bad storage pool. This sort of bug usually damages the storage pool structures. Thus, it is a good idea to check the state of the storage pool using the pool map (pm) command.

The pool map (pm) command takes the address of the storage pool as an argument. This address of the start of the storage pool for a processor is stored in the external variable _Pool (whose address can be determined using Examine). The pool map command displays for each block in the storage pool a 1 if the block is allocated or a 0 if the block is free. It also indicates how many blocks there are in the pool and provides a list of addresses of the free blocks and their sizes. The pm command checks for bad blocks, displaying the address of the (first) bad block along with the address of the block immediately preceding the bad one. The pool map command terminates when it encounters a bad block.

Usually, but not always, the bad block results from some task writing beyond the end of a buffer in the block preceding the bad one. Locating the owner of this preceding block often finds the culprit. One can determine the owner of a block by examining the memory resource list for each suspect task using the m command, which displays a list of all the blocks allocated to a particular task. Once one knows which task has caused the corruption, it is often easy to find the bug either by looking at the source code or by re-running the program with additional breakpoints set in the suspect task to permit tracking its activity.

The pool map (pm) and memory resource (m) commands are also useful in locating space performance related bugs. Examining the memory resources can quickly reveal space hogs and which tasks forget to free the blocks with which they are finished. The display pool block descriptor commands (pb and pn) provide more detailed information about specific blocks in the storage pool; this is sometimes useful in tracking down memory allocation related problems.

If one is having a problem with the connections to servers from a task, it is useful to examine its connection resource (UCB) list using the u command. The CONN_RESOURCES field in the task descriptor contains the address of the first UCB in the list; each UCB points to the next one in the list, so one can examine the entire list by applying the u command to each UCB.

The programmer can provide his own extensions to the debugger by coding additional debug tasks (for example, to audit some data structure), which can be created (and thus invoked) from the debugger using the C command. Likewise, any undesirable task can be destroyed by issuing the D command to the debugger.

Other extremely useful facilities, including the plantable breakpoint (b) commands, use bit functions and the use bit vector (v) commands, the call traceback (c) command, and the _Log_gossip function, are described in detail in other sections of this document.

One final command worth noting is the why (w) command. After one uses these various commands to examine the state of the program, it is easy to forget just why the debugger was invoked (e.g., at which breakpoint we are). The why command redisplays the information detailing why the debugger was entered including any message that was displayed originally, the register dump, and the call traceback.

While the debugger is a far cry from being the ideal tool, it does contain many features that make debugging a Harmony program easier. Use of all of these features is strongly encouraged.

## Alternative Debugger Configurations

In the current configuration the interactive debugger executes within the Gossip task and it is invoked only when a breakpoint hit, an exception, or an abort occurs. At times it is desirable to be able to interact with the interactive debugger while allowing the program to run freely (and in parallel with this interaction). There are two easy ways of accomplishing this.

This first way is to write a simple debug task whose function body is simply:

```
for(;;)
    {
        _Breakpoint( "Debug Task" );
    }
```

Once created, this task will always breakpoint. This implementation has two disadvantages. Because Gossip processes its breakpoint/exception/abort messages sequentially, the user must issue the r command to resume execution of the stopped task in order to allow Gossip to process any queued messages. A second problem is that Gossip and the running program are often competing for use of a single terminal — this can be a fairly minor problem for output but it is an extreme nuisance for input. (This second problem is alleviated by use of the multiwindow UW server.)

An alternative is to supply a slightly more sophisticated debug task. In this case, the debug task would call the function _Debug, which is the interactive debugger. But first, it would have to do much the same thing as _Gossip does to open a connection to a tty or window (stream I/O) server (not the same tty or window as DEFAULT:) and to initialize a DBG_STATE data structure to pass to _Debug. Once created, this debug task would always be available for interacting with, while Gossip would handle the breakpoint hits, exceptions, and aborts. Such a debug task can be written easily by examining the code for the _Gossip function. (The file system example program provides an example of such an auxiliary debug task.)

In any of these configurations, if the debugger and corresponding tty/window server tasks are on the same processor as the tasks being debugged, then they should be assigned priorities that are higher than the priorities of the tasks being debugged. This permits interaction with the debugger even when the task being debugged runs astray into an infinite loop, for example.

## Normal versus Busywait Versions

Two versions of the interactive debugger exist: a normal version and a busywait version. The normal version is used by linking the application program with the normal debug library (i.e., debug.<lib-ext>). The busywait version can be used by linking the application program with the busywait debug library (i.e., bwdebug.<lib-ext>).

The normal version of Debug, which works exactly as described above, executes as part of the Gossip task. Whenever a breakpoint, exception, or abort occurs in a running task, the handler for the condition (_Breakpoint, _Abort, or an exception handler) sends a message to Gossip, thus causing the task to block and the interactive debugger to be entered. Upon termination of the interactive dialogue, Gossip replies to the stopped task allowing it to continue execution.

For the busywait version of Debug, calling _Breakpoint does not cause a message to be sent to Gossip, but rather the interactive debugger is executed as a subroutine called from _Breakpoint; that is, it

executes as part of the task being debugged. Also, it uses busywait I/O and so involves no sending of messages to any tasks, thus enabling debugging of the tty server and kernel software. Because of the inadequacies of busywait I/O, the busywait version of the debugger is of very limited value for other uses.

To facilitate debugging of terminal server code, the entire Gossip task can be configured easily to run without any interrupt-driven I/O. One must link the test program with the busywait debugger instead of the normal debugger. Second, one should not define DEFAULT: as an alternate name for the tty server. Then when _Gossip tries to open a connection to DEFAULT:, the name lookup by _Directory will fail so a connection will not be opened (which causes no harm). Since Gossip is the only task that tries to open a connection to DEFAULT:, this should have no other impact on the test program.

To facilitate initial checkout of the system, without running a terminal server, a busywait version of the srtest example program is provided. This version does not even require a terminal server to be linked with the program.


# Application Note n002

**Title:** **What is a Server?**

**Revisions:**

| | | |
|---|---|---|
| 1986-08-05 | W. Morven Gentleman | |
| 1987-11-16 | Darlene A. Stewart | (formatting) |
| 1989-01-24 | Darlene A. Stewart | (minor edits) |


**See Also:** W. M. Gentleman, S. A. MacKay, D. A. Stewart, and M. Wein, Using the Harmony Operating System (Release 3.0), NRC/ERA-377, National Research Council of Canada, Ottawa, Ontario, February 1989.

---

1. A *server* is a task that performs some service on behalf of arbitrary and unknown *client* tasks. It is analogous to a library subroutine.

2. What distinguishes a server from a library subroutine is that state must be maintained across service transactions, perhaps even across clients, or perhaps just across transactions for the same client. This state is inconvenient to represent by data associated with the clients, but can be conveniently represented in data local to the server task.

3. Because it accepts one request at a time, a server serializes access to a shared resource which it controls and thus can solve critical races and mutual exclusion problems. It need not process one request at a time, that is, it need not complete processing one request before accepting another. For simple nonsharable but serially reusable resources, having the server perform operations on the resource on behalf of the client may be enough. For complex sharable resources, such as an application data structure, the server may allow concurrent access by workers or by the clients themselves, while enforcing consistency conditions such as atomic updates, two phase commit, or access only to disjoint substructures.

4. Because the client will not, in general, have created the server, it cannot be expected to have inherited the task id from somewhere, so must identify the task by symbolic name. As Harmony tasks are identified by system-produced task ids (there are no "well-known names" in networking jargon), an explicit lookup of the symbolic name is required. This facilitates context dependent or time dependent binding of symbolic name to task id.

5. The server often needs to know if the client dies, perhaps while the service transaction is in progress, perhaps between service transactions. Hence maintaining a connection, where the system guarantees to supply CLOSE messages, is useful.

6. Because the server may maintain state for several clients, it must have a way to distinguish them. Connection ids, which the server can issue to clients, solve this and also allow a client to have more than one connection to the same server.

7. Because of the above, a server must support OPEN and CLOSE messages, must register with the _Directory task, and must provide connection numbers.

8. To reduce overhead of send-receive-reply on every service request, a server may require the client to provide a buffer in the client's space to support some caching. This buffer is the UCB_XTRA. Its use is dependent on the particular server and on any library functions used by the client to interact with that server.

9. Servers often have some parameters that must be set before normal client transactions can be handled. For example, a communications line speed must be set. Toggles and value parameters in the open directive can be used to ensure these parameters are set before use.




**Title:** **Coping with the Defect in Motorola 68000 Family Interrupt Handling**

| **Revisions:** | 1986-08-05 | W. Morven Gentleman | |
|---|---|---|---|
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1989-01-24 | Darlene A. Stewart | (minor edits) |

The processors in the Motorola 68000 family (MC68000, MC68008, MC68010, MC68012, and MC68020) share a serious design defect with respect to handling interrupts. The semantics defined for exception handling are that the only context switched on an exception is the status register and program counter: the current values (plus possibly some internal processor state to support mid-instruction interrupts) are pushed on the current stack, then a new program counter is taken from the interrupt vector and a new status register is generated from the interrupt level. However, the new context is not guaranteed to execute even just one instruction before it can be itself preempted by a yet higher priority interrupt.

The problem is that this automatically switched context is not the full context that must be preserved for the preempted program while responding to the interrupting event. For example, registers must be saved and global variables (such as _Active) may need to be set. More seriously, most modern operating systems, including Harmony, treat interrupts not as unexpected function calls but rather as events that make waiting tasks ready again. This means that the context switch includes changing stacks. Because the execution of even the first instruction of the interrupt handler is not guaranteed, even making this first instruction disable interrupts does not guarantee that the full context switch can be completed before subsequent interrupts are allowed. When the full context switch involves changing stacks, the consequence is that the state may be saved on the wrong stack.

Consider an example where a task A is active at a priority such that its processor level is 0. Suppose an interrupt, for which task B is waiting, occurs at processor level 2. Suppose further that handling this interrupt is then itself preempted by an interrupt, for which task C is waiting, occurring at processor level 4. What we would like is that the interrupt handler for the level 2 interrupt pushes the additional state of task A onto the stack of task A following the exception frame pushed there by the hardware, then switches the stack pointer to point to the stack of task B, and restores the additional state of task B. If the level 4 interrupt arrives after this, the exception frame will be pushed onto the stack of task B, and the interrupt handler for the level 4 interrupt can then push the additional state of task B onto the stack of task B, before setting the stack pointer to point to the stack of task C and restoring the additional state of task C. We can almost ensure that this happens by making the first instruction of each interrupt handler disable interrupts, and not enabling interrupts again until the state of the waiting task has been completely restored and the waiting task is dispatched. Unfortunately, if the level 4 interrupt occurs after the level 2 exception processing has begun (the exception frame is being pushed onto the stack of task A) but before the level 2 interrupt handler executes its first instruction, the second exception frame is pushed not on the stack of task B, but on the stack of task A. Then the additional state is saved on the stack of task A by the level 4 interrupt handler, which is indeed correct as the level 2 interrupt handler did not have a chance to switch this state for the state of task B. Task C is then activated and executes correctly.

The problem manifests itself because the readying and dispatching of tasks does not necessarily follow the simple nested discipline of stacks. In particular, when task C is active, actions it takes can ready other tasks, at priorities higher than that of task A, so that when task C ultimately blocks, these tasks will run before task A runs again. Let us consider only one such task D and assume it runs at a priority corresponding to processor level 1. Thus when C blocks, D will be dispatched, because it is of priority higher than A, which was apparently active when the level 4 interrupt happened. Actually B, which was waiting for an event (level 2 interrupt) that has now happened, is of higher priority than D, but because the level 2 interrupt handler did not have a chance to execute, the ready queue data structures were not updated to show that it is ready, and so the dispatcher does not know to dispatch it.

What will happen, on most hardware, is that the level 2 interrupt will not yet have been cleared, as the interrupt acknowledge processor cycle is not enough, and some explicit directive to the interrupt source must be issued. Consequently, the interrupt is still pending, and, since task D does not mask level 2 interrupts, the level 2 interrupt will be taken again. This time, let us say, B is dispatched and completes, D as next highest priority task is dispatched and completes, and eventually the system attempts to redispatch A. Unfortunately, the extra exception frame pushed onto the stack of A will make it attempt to re-execute the level 2 interrupt handler, for an event already handled. If we are lucky it may appear as a spurious interrupt but usually the effect is a crash. If the interrupt source has the unusual design whereby the interrupt acknowledge processor cycle itself does clear the interrupt, the symptom is different. In this case the event is now recorded only on the stack of A, so that regardless of priorities, the interrupt does not appear to happen until task A is dispatched again — D will happily run even though it is lower priority than the unserviced interrupt for B. Indeed, if task A is destroyed, the interrupt will never be serviced.

The window leading to this unfortunate sequence of events is quite narrow: if the level 4 interrupt happens too soon, the level 4 exception is taken instead of the level 2 happening first, whereas if the level 4 interrupt happens too late, the level 2 interrupt handler has disabled interrupts ensuring all is well. We have measured the window as about 2 μs on a 10 MHz MC68000 with no wait states.

This may mean that the problem is of sufficiently low probability to be ignorable. Alternately, external hardware can eliminate it by holding off subsequent interrupts until sufficiently long after the interrupt acknowledge cycle. However, crashes from this cause have been observed on commercial hardware, so it is worth knowing that interrupt handlers can be written to detect the occurrence of the problem and correct for it (although the standard interrupt handlers supplied with Harmony do not).

The key observation is that the exception frame records, in the saved status register, the processor level at the instant of interrupt. On the other hand, following _Active to locate the task descriptor of the apparently active task, and from there following MY_QUEUE to find the ready queue record for this task, allows us to find in ENABLE_SR what that level should have been (the only other level value the task might have is disable, but in that case the interrupt would not have happened). Thus if these are the same, no problem occurred and normal interrupt processing can continue; if they are different, the problem has been detected and corrective action is required.

What corrective action? As described above, for typical hardware the interrupt is still pending and will trigger again, so simply popping the extra exception frame off the stack when it is found is sufficient. For the other kind of hardware where the interrupt acknowledge processor cycle clears the interrupt, the corrective action is more messy as all stacks affected must be appropriately adjusted.


# Application Note n004

**Title:** **Differences between Harmony and Unix I/O**

| **Revisions:** | 1986-08-07 | W. Morven Gentleman | |
|---|---|---|---|
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1989-01-24 | Darlene A. Stewart | (reorganized) |

Harmony I/O and Unix I/O are very similar, both deriving from the stream I/O implemented in the BCPL programming language (even earlier antecedents may be identified). However, there are differences, some of which are described below. These differences can be classified four ways:

- those differences associated with I/O connections
- those differences associated with stream I/O models
- those differences associated with file system structure
- those differences associated with specific implementations of device servers.

**A. Differences Associated with I/O Connections**

1. A Unix child process inherits its parent's open files, whereas each Harmony task must open its own connections.

2. Harmony uses _Open() much more extensively than Unix uses open(). Whereas Unix opens files, and opens devices (which are treated as files), Harmony uses _Open() to establish connections not only to I/O, but to all services that might or might not have been configured. For instance, Unix assumes that there is always a clock, and always a file system, whereas Harmony assumes neither, although it does make the weaker assumption that there is always a directory service.

3. A Unix open() has two parameters: name, which is a file pathname, and mode, which is an integer indicating read or write or both. Unix has several related functions, such as fopen() and popen(), which open specific types of objects or could be regarded as supplying extra or different format parameters. A Harmony _Open() also has two parameters. However, the first is a directive, which is a string that includes, in addition to the pathname, toggles and value parameters specifying options such as the access mode or initial settings. The second is the active userid, in case the server wishes to apply access control. The same function, _Open(), is used for establishing all kinds of connections.

4. Unix devices are configured by a configuration record used at system startup, or by a record read from the file system. In either case the individual application program is not involved. Harmony application programs must supply the server (device) initialization records.

5. Unix devices are normally known to the system before the application program begins execution. Harmony applications must begin by creating the necessary servers, and must avoid the critical races of other tasks attempting to open connections to servers before the servers are created.

Redirecting I/O external to the process and inheriting files in Unix can be considered to be artifacts of this.

6. Unix uses signals to report asynchronous conditions. Harmony explicitly by design has no such construct, and the only asynchronous action that can be taken with respect to a task is to kill it. The most obvious use of this in Unix is killing interactive processes from the terminal. This can be done in Harmony, but is done quite differently. There is no default tty associated with a Harmony task.

**B. Differences Associated with Stream I/O Models**

7. The Unix ungetc() function can be applied an arbitrary number of times, inserting arbitrary characters ahead of what was the current file position, whereas the Harmony _Unget() function backs up the pointer in the current buffer, and hence is only guaranteed to work once, after a call to _Get(), and can only insert the character originally read.

8. Although the Unix getchar() and putchar() functions refer to the standard input and output streams (which may be redirected external to the process), the normal getc() and putc() functions in Unix require specification of the stream to use. The Harmony _Get() and _Put() functions can be redirected at any time, within the task, by calls to _Selectinput() and _Selectoutput(). Fixing the stream separately from the actual byte I/O directive reduces the cost of the latter.

9. Unix supports the concept of an end-of-file (EOF) whereas Harmony does not — all Harmony streams and files are semi-infinite.

10. The Unix printf() function uses a different set of conversion specifications from the Harmony _Printf() function. More importantly, the Harmony _Printf() does not support all the format conversions that the Unix printf() does, and specifically at this time it does not support formatting floating point values. In general the restrictions are to save code space, but the floating point restriction is because no conversion algorithm independent of machine representation is known.

11. By default, Unix flushes a terminal output stream when the character \n is put to it, whereas Harmony only flushes when the buffer is full or when _Flush() is called.

12. Unix fakes the put/get level by macros operating on top of the underlying read/write level. Harmony provides no read/write level, although bytes can be collected to provide records above the get and put level.

13. Unix automatically flushes buffers when a process is destroyed. Harmony does not, indeed Harmony cannot. When a Harmony task is destroyed, the system automatically closes all connections opened by that task (i.e., it sends CLOSE messages to the appropriate servers). Thus a server can flush any server-held buffers, but since Harmony connections are a general mechanism supporting multiple I/O models, not just stream I/O, the system cannot automatically flush any client-held buffers for a connection. Indeed, in Harmony, client-held buffers are not automatically flushed when a task calls _Close() for a connection.

14. Unix block I/O is done in terms of *natural* fixed size records when done at the read/write level. (Originally this was 512 byte blocks, but one of the major BSD4.2 improvements was to increase the block size.) Harmony I/O is done on a message size agreed upon between the client and server,

server and worker, etc. and need not be the same for each pair. It is explicitly hidden from the client.

### C. Differences Associated with File System Structure

**15.** Every Unix process has a current working directory, initially inherited from the parent, whereas a Harmony task has a current node if it uses a file system, but the current node is always initially the root of the file system.

**16.** The underlying data structures are different. The Unix file descriptor returned by open() is an integer which serves as an index into a (unchecked) fixed length vector. Thus the maximum number of open files is fixed at system generation. The file control block returned by fopen() is a pointer to a struct. The UCB returned by a Harmony _Open() is a pointer to a dynamically allocated struct. The same struct (of course not the same instance of the struct) is used for connections to all services. There is no limit on the number of connections that may be open simultaneously in Harmony.

**17.** Unix does not implement single writer/multiple reader consistency control on open files, and indeed several Unix practices rely on having multiple readers and writers. The Harmony implementation of stream I/O, as required by the Harmony file system, fundamentally requires this restriction. More precisely, a file in Harmony consists of the blocks stored, the blocks held by the file device server, and the blocks held by the client — so having multiple clients accessing a file, unless they were all read only, would necessitate avoiding any two of them accessing the same blocks.

**18.** The Unix access control permission structure is copied from the original DEC scheme, including owner, group, and world classes with read, write, and execute permissions. The Harmony permission structure is quite different. The concept of owner is different in Unix and Harmony, indeed the concept of userid is different in Unix and Harmony.

**19.** The Unix file system is an acyclic digraph, with shared nodes (inodes) preserved as long as some directory points to them. The Harmony file system is strictly a tree.

**20.** In the Unix file system, a directory is just a file in the sense that the file contents are the names and pointers defining the substructure. In the Harmony file system, a directory is a file in a different sense, in that every file potentially has both content (like Unix files) and substructure (like Unix directories).

**21.** Every Unix file is time-stamped to the nearest second. Harmony does not provide a clock with a wrap time greater than a month and a half, so Harmony files only have sequence numbers, not time stamps.

### D. Differences Associated with Specific Implementations of Device Servers

**22.** The Unix tty driver supplies *cooked* input with arbitrary typeahead, whereas the Harmony _Tty_server only accepts typeahead one line in advance of pending input requests.

**23.** Unix uses the ioctl() system call to set options on devices. Harmony has no such concept, although options may be set in the open directive of the call to _Open().

24. Unix provides *raw mode* I/O from tty's. The Harmony _Tty_server does not support raw mode.

25. Unix provides pipes. No server currently supplied with Harmony implements a facility like a pipe. On the other hand, pipes are usually used in Unix to provide interprocess communication, which in Harmony is directly and more generally supported by message passing. Berkeley BSD4.2 Unix provides another interprocess communication mechanism, sockets, which is not supported by AT&T System V Unix and has no equivalent in Harmony.

26. Berkeley BSD4.2 Unix supports pseudoterminals, AT&T System V Unix does not. The _UW_server supports the equivalent concept in Harmony.

27. A Unix process can respond to a break from a terminal. The Harmony _Tty_server does not currently handle break, and if it did, the options of what it could do with break are quite different from what a Unix process can do.

28. Through an entry in the termcap or terminfo file, BSD4.2 Unix supports direct cursor addressing and other nonstandard aspects of specific tty devices. The same functionality is supported in AT&T System V Unix by the terminfo mechanism. The Harmony _Tty_server supports neither.


# Application Note n005

**Title:** A Harmony Source Code Style Sheet

| **Revisions:** | | | |
|---|---|---|---|
| | 1986-08-08 | W. Morven Gentleman | |
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1988-06-13 | Stephen A. MacKay | (QUED notes) |
| | 1989-01-25 | Darlene A. Stewart | (minor edits) |

**See Also:** Appendix E. Software Engineering Considerations: Source Management and the Organization of the Harmony Source Tree, *in* W. M. Gentleman, S. A. MacKay, D. A. Stewart, and M. Wein, Using the Harmony Operating System (Release 3.0), NRC/ERA-377, National Research Council of Canada, Ottawa, Ontario, February 1989.

---

Harmony source code is written in a different style from other C code; in particular, the style is different from that used in Unix. While we are not completely consistent, and in some aspects the style is still evolving, the style is distinctive and is based on experience and reasoned arguments, not just idiosyncratic whim. The following list is as yet incomplete:

### 1. Granularity

A Harmony program is not represented in a single file, but is split across several files in accordance with the source management scheme. Each function is in a separate file. Logically associated extern's, the same for all versions of a particular abstraction, are grouped together in a single file, often called extern.c. Similarly, declarations of #define's, struct's and union's that are logically associated, and the same for all variants of a particular abstraction, are grouped together in a single header file, i.e., one whose filename ends in .h. Alternate variants of any of these files may exist in parallel directories corresponding to more specific abstractions — the inclusion file for any particular realization of a program will have #include statements pointing to the appropriate variant of each file.

### 2. #include statements

No #include statements are permitted in the source code itself. Consequently, pathnames appear only in the inclusion files and nowhere in the source code. Thus although pathnames have different syntax in different host operating systems, dependency on the pathname syntax for a particular host operating system is contained entirely in the inclusion files. (Another possible mechanism, relying on search rules, is not viable because many operating systems do not let users set them.) Since a given inclusion file is specific to a particular host, particular target, and possibly to particular other attributes defining the version of the program, dependency on pathname format in inclusion files is not a problem. Related issues of absolute versus relative pathnames are also limited to inclusion files.

### 3. Conditional compilation

The use of #ifdef, #elseif, and #endif is strictly banned. This is for two reasons. First, if code implements several versions this way it becomes unreadable, and in particular following the logic of

any one version becomes impossible. Second, if code has successfully been expressed as largely version independent (e.g., portable code), then the code differing between versions will be a small portion of the total text, and one does not want to wade through the total text in order to find the differences or to be reassured that there are no differences in some part. Instead, Harmony uses the mechanism of inclusion files (and inclusion directories) to support multiple version code.

## 4. Portability considerations

Because, regardless of standards, there exist major compilers that do not implement certain features, the use of some features is banned. Among these are:

- enum
- functions returning struct (implemented wrong in PCC-based compilers anyway)
- passing struct's as arguments to functions
- assuming member name scope is local to the struct or union
- bit fields

## 5. Page layout

We attempt to keep lines shorter than 80 characters and functions on a single page, so that the listing program can produce a comprehensible volume on standard 8.5 in. by 11 in. paper.

## 6. Language features

Some language features are avoided as being not readable, prone to error during maintenance, or simply not often needed. One example is that the case body in a switch statement is always terminated with a break or return statement — we never allow control to fall through to the succeeding case.

## 7. Case rules for identifiers

Although the C language considers identifiers free form, using case consistently to denote attributes of identifiers often makes it possible to read something in the middle of a large piece of code without constantly turning back to look at the declarations. Three such rules are used to restrict choice of identifiers in Harmony source code:

- Local variables and types are all in lower case.
- Manifest constants (#define's), record types (struct and union tags), and record field names (member names) are all upper case.
- Function names and other extern names (names known to the linkage editor) have the first alphabetic character capitalized and contain at least one lower case alphabetic character.

## 8. Use of leading underscore

To avoid collisions between user-chosen identifiers and identifiers in system-supplied code, the identifier of every system-supplied function and every other system-supplied extern in Harmony starts with an underscore. (This same rule is used in Unix, except that in Unix the rule only applies to internal names, and user-visible names have no underscore, whereas in Harmony it applies to user-visible system names also.) The rule is unfortunately incomplete, in that it applies only to identifiers that the

linkage editor knows about. It does not apply to manifest constants, record types, or record field names. In general, because of separate compilation of user code and system code, these names cannot give rise to collisions, but there are exceptions, and of course all names in the standard system header file /harmony/sys/src/sys.h are reserved.

### 9. Identifier uniqueness

Identifiers must be unique in the first eight characters, first because the reference book for the language states that (B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, First Edition, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.), and second because many compilers and linkage editors require it (even though many more recent ones do not).

### 10. Concatenated words

Often the most meaningful identifier is a concatenation of shorter words. If possible, subject to the uniqueness restriction, we separate these words with underscores. If not, we coin new words by simply concatenating the shorter words. We have not used the convention popular on the Apple Macintosh where the first letter of each concatenated word is capitalized to demarcate the original words.

### 11. Declarations

The name of a function should appear on the first line of the function, following the type specifier for the return value if any. The parameter list follows on the same line. The declaration list for the parameters is on subsequent lines, indented with respect to the first line. Each type specifier is on a separate line, and there is a type specifier on each line, although more than one parameter of the same type may be on the line. Inside the function, the declarations of the extern's appear together, separated by a blank line from the declarations of local variables. Again each type specification is on a separate line, and there is a type specification on each line, although more than one variable of the same type may be declared on the same line. In general, no attempt has been made to use tabs to line up vertically all the identifiers being declared in a set of declarations (i.e., having a column of type specifiers followed by a column of identifiers). In header files, when declaring manifest constants, tabs are used to align vertically the values as well as the symbolic names. Similarly, when declaring struct's or union's, tabs are used to align vertically the field names as well as the field types. Indeed in struct declarations, each line has one type specification and exactly one field name, possibly followed by a comment explaining the usage of that field.

### 12. Extern declarations

Even though the scope rules for C are that an extern is in scope from its declaration through to the end of the file, Harmony style is to declare an extern in each function that uses it. (An exception to this rule is that user visible functions are declared in standard headers.) Use of the C scope rule makes reading code more difficult, as it is not obvious what external data a function may manipulate, and introduces errors if the functions are reordered in the source file. (Harmony source code is represented one function per file, but the C scope rule applies to the inclusion files that point to these source files, and reordering may be done in inclusion files.)

### 13. Static storage

The use of static is banned unless for records or arrays that are initialized but not subsequently changed. This is partly to simplify assuring ROMability, but primarily because all tasks on the same processor

that call a function with data declared static share the same copy, and critical races can easily be induced.

## 14. Brace style and indentation

The basic unit of indentation is an *indent level* (its size is defined below). The brace style is that each brace appears on a line by itself, open braces indented by half of one indent level from the previous level and each close brace is lined up beneath its matching open brace. Substatements, such as the body of a `for`, `if`, or `while` statement, are of course indented at a new level. `Case` labels inside a `switch` statement indent one level with respect to the `switch` statement itself, and the `case` bodies indent one level with respect to the `case` labels. Sometimes there are braces around a `case` body if it is more than a single statement followed by a `break` statement, but generally the `case` label and indenting are adequate to demarcate the `case` body and avoid unnecessary vertical expansion. For editors that support only mono-spaced text, the basic indent level is four spaces. Therefore braces are indented two spaces from the previous indent level. A tab should be substituted for a run of spaces that ends at a tab stop. Because of restrictions that exist in many development systems, tab stops are set every eight spaces. Readability of code can be improved through the use of proportionally spaced text in editors that support it. For ease of conversion between editors, the code should be formatted similarly to the mono-spaced code. For example, with the QUED/M editor from Paragon Concepts on the Mac, we use the Geneva 10 point screen font (with font substitution to Helvetica for listings on the Laserwriter). Each indent level is two tabs, with braces indented one tab from the previous indent level. Tabs are set every two *numeric character widths* (e.g., the width of the string "35") and are used wherever possible because with proportionally spaced text, a blank character does not have a fixed width. Note that the *AutoIndent* setting in QUED/M simplifies indenting.

## 15. Folding of long expressions

The rules for the folding of long expressions are more complicated than simple indenting, and rather than being syntax driven, they simply attempt to produce readable code. Thus an assignment statement folds so that the right-hand side is indented past where the "=" appears following the left-hand side. Similarly, function calls usually fold, if necessary, on the "," delimiting the arguments, and often if more than one argument folds, they all do. Logical expressions, controlling `if` or `while` statements, fold on "&&" or "||" and if one such fold occurs, all terms at this level fold. Initializers for arrays and structs use indenting and brace style similar to the conventions in code.

## 16. Comments

Comments in Harmony code generally are one of three kinds:

- *Function Specification*
  Unless the function is very short, and its purpose is obvious, every function must start with a comment describing its purpose. This comment follows the declaration list for the formal parameters and precedes the open brace for the compound statement which is the function body.

- *Section Heading*
  Often the code of a function consists of several fairly independent sections, each of which is accomplished in many statements. Beginning such a section with a heading comment, which

explains what the next section of code is intended to do is often useful. The comment appears on one or more lines by itself, indented the same as the section of code it describes.

- *Annotation*
  When something subtler is being done than is obvious by reading the statements executed, it can help to have a comment to the right of the statements executed, elaborating on what is really happening. Although usually one line, this sometimes can take several.

When a comment spans more than one line, the open and close comment symbols, "/*" and "*/", appear on lines by themselves and are lined up with "*" on the intervening lines to give a vertical line to the left of the comment. The usual rule for mono-spaced text is to start the comment at the current indent level with the "/*". The subsequent lines indent by a blank character; the text lines have the form "<blank>*<blank><blank>text" and the final line has the form "<blank>*/". For proportionally spaced text on the Mac, the comment begins indented by a space, "<blank>/* " and continues indented by a tab, "<tab>*<tab>text" and "<tab>*/".

## 17. White space

In general, white space is used to improve readability, and is not simply determined by syntax. Operators are preceded and followed by a blank. Commas separating arguments to a function are followed by a blank. Open parenthesis (round bracket) is generally followed by blank, in which case the matching close parenthesis (round bracket) is preceded by a blank. (Square brackets used for array declaration or subscripting are not similarly blank padded.) Function names, and keywords such as if and while, are followed directly by the open parenthesis with no intervening blank. Parenthesis-containing casts are not followed by a blank. Vertical white space is used to separate conceptual operations, which may be implemented with several statements.

## 18. Basic type sizes

The C language definition defines int, long int, short int, and char without being specific about the range of values these types can represent. The same applies to unsigned. Different compiler writers have made different choices, even for the same target machine. Harmony code cares that there are at least enough bits, consequently the standard types are not used in the code; instead int_32, int_16, char, uint_32, uint_16, and uchar are used. These types are defined, using #define or typedef, in the compiler header file (compiler.h) to arrange that at least the indicated number of bits are available for each type. (Harmony has a separate compiler.h file for each compiler supported.)

## 19. Virtual records

A machine independent construct used in many places in Harmony is the concept of virtual records. By overlaying a pointer to a list with a virtual instance of the type of record in the list, aligned so that the link pointer in the record coincides with the actual pointer to the list, code for insertion or deletion of the first or last record may become identical with code for insertion or deletion of interior records, thus avoiding special cases. (The effect is similar to Wirth's use of a sentinel record, but avoids the allocation of a possibly large record not otherwise used.) An ugliness of virtual records in C is that the casts required to computed field offsets in records are very messy. Eventually we may encapsulate this in macros.


# Application Note n006

**Title:** **Resources with Costly Context Switches**

| **Revisions:** | | | |
|---|---|---|---|
| | 1986-08-11 | W. Morven Gentleman | |
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1988-11-25 | W. Morven Gentleman | |
| | 1989-01-25 | Darlene A. Stewart | (minor edits) |

The processor is a resource whose state is saved and restored on every context switch. Some devices, such as a line printer, a tape drive, or a robot arm, are resources which have so much external state that it makes no sense to switch them between tasks: they have to complete what they are doing before starting something else. Such resources are normally owned by some server, which queues service requests from other tasks and issues them at feasible times.

Between these two extremes are resources which could be shared between tasks, but which have a context switch which is sufficiently expensive that we probably do not want to save and restore it on every task dispatch. Examples of such resources might be floating point coprocessors, signal processing peripheral processors, and array processors. The context switch can be costly in that saving and restoring the state may take many milliseconds (perhaps because the resource must reach a stable internal state before it can be halted and its state can be extracted). It can be costly in that saving the entire state might require hundreds of bytes of stack. It can be costly in that no direct synchronization exists between the host processor and the coprocessor, so that to avoid saving the state on the wrong stack, the host processor must busywait testing heuristically as to when the coprocessor has completed.

Rather than having the costly context switch on every task dispatch, whether or not the resource is actually in use, we could try to arrange that it happens only when necessary. One possibility is to modify the task descriptor to track whether such a resource is in use and only save the state when a task using the resource blocks, only restoring it when such a task is dispatched. A more complex possibility is to save and restore the state only when a task about to be dispatched actually needs the resource. This allows more processing overlap than the simpler scheme, but requires a more complex implementation because the state of the resource cannot be saved with the task being blocked as in general the resource will instead have been activated by a task that was running some time earlier.

In many cases, the basic causes of costly context switching arise from the need to support preemption. If the resource is instead allowed to come to a natural stopping point, it will be in a quiescent state, with little or no state to save and restore, and little or no time required to do that. Furthermore, in many cases, preemption is unnecessary, in that the processor may indeed need to be preempted to execute some urgent task that has now become ready to run, but this urgent task will not involve the resource with the costly context switch. Where such behaviour can be assured, we may be able to accommodate such resources with no change to the Harmony task management, simply by relating the natural stopping points of the resource to the natural break dispatching that happens within a single Harmony priority level.

Specifically, if all the tasks that use a resource run at the same priority level, then when a task starts to use the resource, it will continue to be active, thus preventing other tasks from attempting to use the resource, until it comes to a natural break, i.e., takes an explicit action to yield the processor. The exceptions to this, i.e., preemption of this task by higher priority tasks, by definition do not affect the resource, so its state need not be saved and restored. The natural breaks involve sending and receiving messages, e.g., when doing I/O, which are points when a resource like a floating point coprocessor is quiescent anyway, so possibly letting some other task take over and use it is fine. The tasks using the resource will share it in a round robin, and indeed to force the next task to have a turn we could have tasks that use the resource periodically send messages to an echo task, which simply replies, but thereby rotates the round robin. In other words, unless someone explicitly tries to be clever and exploit processor and coprocessor overlap (which conflicts with task switching) running all tasks that use a resource at the same priority can be a complete and simple solution to the resource sharing problem, requiring no special code.

Experience actually using this approach to control access to the Motorola MC68881 floating point coprocessor in Motorola MC68020 systems has not been completely satisfactory. First, many situations have arisen where it was necessary for a task that used floating point to preempt another task that also used floating point, so the use of multiple priority levels is essential to the problem specification. More surprisingly, it was found that compilers that assume state will be preserved in the resource across function calls are awkward because they assume this even when the function is one that blocks and causes a context switch to another task that also uses floating point. The Freesoft GNU C compiler is a case in point. Either explicit source code level saving and restoring of floating point registers must be used across function calls that block, or floating point register save and restore must be built into the context switch (at least for tasks that use floating point), which is exactly what running at the same priority all those tasks that use floating point was supposed to avoid.

**Title:** **Harmony Development on a Mac for the Dy-4 DVME-134**

| **Revisions:** | | | |
|---|---|---|---|
| | 1987-06-18 | Darlene A. Stewart | |
| | 1987-11-16 | Darlene A. Stewart | (formatting) |
| | 1988-11-24 | Darlene A. Stewart | (Release 3.0) |
| | 1989-01-26 | Darlene A. Stewart | (more Release 3.0) |

This application note describes some details about Harmony development for the Dy-4 DVME-134 target computer using an Apple Macintosh with the Consulair Mac C compiler, linker, and associated tools, and the inTalk terminal emulator. It explains how to compile and link the Harmony system libraries, how to compile, link, and generate .msr files for the dummy programs and for the srtest example application program, and how to download srtest to a DVME-134 system. This document assumes familiarity with use of the Macintosh system, in particular, use of the Finder and MultiFinder, launching applications, and SFGetFile boxes. We also do not attempt to explain the details of using the various tools — refer to their user manuals for detailed information.

## A. Installing the Necessary Tools

1. The Harmony tree must be installed under Master, so that the HFS pathnames to Harmony files are of the form *Master:harmony:...*. The Harmony tree under Master contains only code (both source and inclusion files).

   We keep any (source or inclusion) code files on which we are currently working under Working, but only those that differ from the ones under Master (i.e., Working contains a sparse Harmony code tree); only when we have completed (and tested) a project (whether a small one, such as a bug fix, or a big one, such as writing a new server), do we integrate (copy) the code from Working back into Master and then distribute a complete new copy of Master to all our Macs.

   Deletions are represented under Working using *-null-file-* and *-null-dir-* files, which are text files with no contents having special icons; the name of a -null-file- or -null-dir- file matches the name of the file or directory being deleted. Thus, Working can represent all changes, additions, and deletions that must be applied to Master at an integration point.

   All output files (.hlib, .Rel, .MAP, .out, .exe, .msr) are kept under Derived, along with copies of the inclusion files used to generate the output files. Derived may also contain other derived files, including source for one-time tests. Derived is used to hold all throw-away items that are never copied to Master, whereas Working is used to hold new and changed items that will eventually be integrated into Master.

   The inclusion files in Working point only to source files in Master as do the inclusion files in Master. The inclusion files in Derived, however, point to source files in both Master and Working, as appropriate.

On our Macs, Master, Working, and Derived are MacServe volumes on the hard disk, so they appear as disks to the system. If it is not possible to create volumes on the hard disk (such as with MacServe), one can simply create three folders at the root of the HFS on the hard disk called Master, Working, and Derived and use these folders as described above. If Master, Working, and Derived are folders instead of volumes, it will be necessary to edit all Harmony inclusion files changing Master to <hard disk>:Master, where <hard disk> is the name of the hard disk.

The layered scheme described above can be extended easily to include more layers too, e.g., Application-Master, Application-Working, etc.

2. The System Folder must contain the Paths.Rsrc file for Harmony. Both the source (Harmony.path) and the compiled (Paths.Rsrc) files for this path resource are distributed on the Harmony Tools disk. Paths.Rsrc is produced by compiling Harmony.path using the Consulair Path Manager.

3. To use the Consulair Mac C compiler with MultiFinder, one must arrange that Mac C is loaded in the low 512K of memory. Two small patches make this easier to accomplish. MultiFinder already contains a patch in it to force load programs with certain creators in the first megabyte. One such program is Microsoft EXCEL. Thus, patching the MultiFinder file (e.g., with Fedit+) to change all (usually two or three) instances of the string XCEL to MACC will cause MultiFinder to force load the Mac C compiler in the first megabyte. Then, the Application Memory Size (in the Get Info box) for the Consulair C application must be set to 350K. Next, we arrange that the upper half of the first megabyte is full by changing (e.g., with ResEdit) the creator of the Consulair Link application to MACC too and setting the Application Memory Size for Link to 475K. (The size required for Link to make this trick work may vary from system to system.) Then, if one simply follows the procedure of always loading the Link application before the C application under MultiFinder, the Mac C compiler will run as well with MultiFinder as without MultiFinder. It may be most convenient to move the C and Link applications to the Mac desktop.

4. The Harmony tools — *bound*, *examine*, *fixaddr*, *listing*, *listtree*, and *makemsr* — are also distributed on the Harmony Tools disk. These programs should be placed in a convenient place, say in a Harmony Tools folder on the hard disk. Also on the Harmony Tools disk are two empty files named -null-file- and -null-dir- having the special icons to represent deletions. Documentation for the various Harmony tools is located in the Master:harmony:doc:tools folder.

## B. Building Harmony

5. Using the Finder, create the folder Derived:harmony:sys:inc. While holding the option key down, drag the folder Master:harmony:sys:inc:dy134cmac to the newly created Derived:harmony:sys:inc. This creates an inclusion tree for the version of Harmony for the DY-4 DVME-134.

6. Compile Harmony using the following procedure. Launch the Consulair Mac C compiler; this can be done by double clicking on the C application in the Mac C folder or on the desktop (or by transferring to it from QUED using the Transfer menu, if not using MultiFinder). Once in the compiler, walk the Derived:harmony:sys:inc:dy134cmac tree, using the SFGetFile box presented by the compiler, and compile every .c and .asm file. A file can be compiled by double clicking on it or by selecting it and clicking the *Compile* button. Note that in the compiler SFGetFile box, one must select whether to see C files or assembly language files by selecting the corresponding radio

button; it will not display both C files and assembly language files at the same time, so be careful not to miss compiling the assembly language files under the kernel, the debugger, and the devices. Upon completion of compiling all the files, one can either exit the compiler by selecting *Quit* from the File menu (or by transferring to another program, such as the linker, using the Transfer menu, if not using MultiFinder).

The list of files to compile is:

```
Derived:harmony:sys:inc:dy134cmac:connect:connect.c
Derived:harmony:sys:inc:dy134cmac:connect:contable.c
Derived:harmony:sys:inc:dy134cmac:connect:directory.c
Derived:harmony:sys:inc:dy134cmac:connect:report.c
Derived:harmony:sys:inc:dy134cmac:debug:breakpnt.c
Derived:harmony:sys:inc:dy134cmac:debug:bwdculib.c
Derived:harmony:sys:inc:dy134cmac:debug:bwdebug.c
Derived:harmony:sys:inc:dy134cmac:debug:bwio.c
Derived:harmony:sys:inc:dy134cmac:debug:dbgagent.c
Derived:harmony:sys:inc:dy134cmac:debug:dbgcntrl.c
Derived:harmony:sys:inc:dy134cmac:debug:dbgshadow.c
Derived:harmony:sys:inc:dy134cmac:debug:dcusrlib.c
Derived:harmony:sys:inc:dy134cmac:debug:debug.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:bw.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:imc68901.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:m68901int.asm
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:mfptimer.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:s68901int.asm
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:spi.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:spo.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:sspi.c
Derived:harmony:sys:inc:dy134cmac:devices:mc68901:sspo.c
Derived:harmony:sys:inc:dy134cmac:devices:valalter:valalter.c
Derived:harmony:sys:inc:dy134cmac:devices:valalter:vavpint.asm
Derived:harmony:sys:inc:dy134cmac:gossip:gossip.c
Derived:harmony:sys:inc:dy134cmac:gossip:gosusrlib.c
Derived:harmony:sys:inc:dy134cmac:kernel:kernel.c
Derived:harmony:sys:inc:dy134cmac:kernel:kernelasm.asm
Derived:harmony:sys:inc:dy134cmac:kernel:presetup.asm
Derived:harmony:sys:inc:dy134cmac:kernel:ramextern.c
Derived:harmony:sys:inc:dy134cmac:kernel:romextern.c
Derived:harmony:sys:inc:dy134cmac:kernel:storepool.c
Derived:harmony:sys:inc:dy134cmac:kernel:taskabs.c
Derived:harmony:sys:inc:dy134cmac:lib:lib.c
Derived:harmony:sys:inc:dy134cmac:lib:parse.c
Derived:harmony:sys:inc:dy134cmac:servers:clock:clkusrlib.c
Derived:harmony:sys:inc:dy134cmac:servers:clock:clockserv.c
Derived:harmony:sys:inc:dy134cmac:servers:ehvt:ehvtextrn.c
Derived:harmony:sys:inc:dy134cmac:servers:ehvt:ehvti.c
Derived:harmony:sys:inc:dy134cmac:servers:ehvt:ehvtserv.c
```

Derived:harmony:sys:inc:dy134cmac:servers:null:nullextern.c
Derived:harmony:sys:inc:dy134cmac:servers:null:nullserver.c
Derived:harmony:sys:inc:dy134cmac:servers:tty:tty.c
Derived:harmony:sys:inc:dy134cmac:servers:tty:ttyextern.c
Derived:harmony:sys:inc:dy134cmac:servers:tty:ttyserver.c
Derived:harmony:sys:inc:dy134cmac:servers:uw:uw.c
Derived:harmony:sys:inc:dy134cmac:servers:uw:uwextern.c
Derived:harmony:sys:inc:dy134cmac:servers:uw:uwserver.c
Derived:harmony:sys:inc:dy134cmac:streamio:streamio.c

*NOTE:* Walking the tree compiling many files (without quitting the compiler) eventually causes the compiler to go into a snit, where it cannot find files correctly. If this problem occurs, simply quit from the compiler, re-launch it, and continue. Occasionally, the snit manifests itself in other ways, for example, by causing the compiler to quit with a system error; in these cases, the compiler often leaves locked .Cer and .ASM files in the folder of the inclusion file being compiled when the snit occurred. These locked files must be unlocked in order for the compiler to work correctly; this can be done by attempting to open the culprit .Cer and .ASM files using QUED (rebooting will also unlock any locked files).

7. Build the Harmony libraries using the following procedure. Launch the Consulair Mac C linker; this can be done by double clicking on the Link application in the Mac C folder (or by transferring to it from the Mac C compiler or QUED using the Transfer menu, if not using MultiFinder). Use the SFGetFile box presented by the linker to invoke the linker on the .link files in the Derived:harmony:sys:inc:dy134cmac folder. A file can be linked by double clicking on it or by selecting it and clicking the *Link* button. Upon completion of linking all the files, one can exit the linker by selecting *Quit* from the File menu (or by transferring to another program, such as QUED, using the Transfer menu, if not using MultiFinder).

   The list of files to link is:

   Derived:harmony:sys:inc:dy134cmac:bwdebug.link
   Derived:harmony:sys:inc:dy134cmac:debug.link
   Derived:harmony:sys:inc:dy134cmac:externs.link
   Derived:harmony:sys:inc:dy134cmac:servers.link
   Derived:harmony:sys:inc:dy134cmac:system.link

## C. Building the Dummy Programs

8. Using the Finder, create the folder Derived:harmony:example:dummy:inc. While holding the option key down, drag the folder Master:harmony:example:dummy:inc:dy134cmac to the newly created Derived:harmony:example:dummy:inc. This creates an inclusion tree for the dummy programs. (A dummy program must run on each processor not being used by the Harmony application. :normal:dummy1, :normal:dummy2, and :normal:dummy3 are dummy Harmony programs for processors 1, 2, and 3, respectively. :fast:dummy1, :fast:dummy2, and :fast:dummy3 are minimal dummy programs for processors 1, 2, and 3, respectively; they keep the processors out of the way and are very fast to download.)

9. Compile the dummy programs using the following procedure. Launch the Consulair Mac C compiler. Use the SFGetFile box to compile the .c and .asm files in the Derived:harmony:example:dummy:inc:dy134cmac tree. (See item 6 for details.)

The list of files to compile is:

Derived:harmony:example:dummy:inc:dy134cmac:fast:code1.asm
Derived:harmony:example:dummy:inc:dy134cmac:fast:code2.asm
Derived:harmony:example:dummy:inc:dy134cmac:fast:code3.asm
Derived:harmony:example:dummy:inc:dy134cmac:normal:code1.c
Derived:harmony:example:dummy:inc:dy134cmac:normal:code2.c
Derived:harmony:example:dummy:inc:dy134cmac:normal:code3.c
Derived:harmony:example:dummy:inc:dy134cmac:normal:externs1.c
Derived:harmony:example:dummy:inc:dy134cmac:normal:externs2.c
Derived:harmony:example:dummy:inc:dy134cmac:normal:externs3.c

**10.** Link the dummy programs using the following procedure. Launch the Consulair Mac C linker. Use the SFGetFile box to link the .link files in the Derived:harmony:example:dummy:inc:dy134cmac tree. (See item 7 for details.)

The list of files to link is:

Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy1.link
Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy2.link
Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy3.link
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy1.link
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy2.link
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy3.link

**11.** Run fixaddr on the dummy programs using the following procedure. Fixaddr converts the Mac executable output (.out file) produced by the linker to a Unix style executable (.exe file) that Harmony tools can deal with. Launch fixaddr by double clicking the fixaddr application in the Harmony Tools folder. Fixaddr displays an SFGetFile box for the file to be converted; select the dummy1.out file in the Derived:harmony:example:dummy:inc:dy134cmac:fast folder. Fixaddr then prompts for the text offset (the start address for the Harmony program); this should be 0x100500 for processor 0, 0x200000 for processor 1, 0x300000 for processor 2, etc. Enter "0x200000" followed by a <cr>. Fixaddr generates numerous information messages as it processes the file; any warning messages about unloaded segments can be ignored. Additional warnings about magic instructions are generated for the :fast:dummy programs; these are expected and can be ignored. Finally, after the file is converted, fixaddr prompts for whether another file is to be converted; enter "y" to convert the next file (enter "n" or <cr> to quit).

The files to be converted using fixaddr with their corresponding text offsets are:

| | |
|---|---|
| Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy1.out | 0x200000 |
| Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy2.out | 0x300000 |
| Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy3.out | 0x400000 |
| Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy1.out | 0x200000 |
| Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy2.out | 0x300000 |
| Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy3.out | 0x400000 |

A typical display from fixaddr after converting a file might look like:

```
------ FIXADDR ------
Select input file:
Enter text offset: 0x100500
Link file srtest0.out opened.
Map file srtest0.MAP opened.
Exe file srtest0.exe opened.
Initialization complete.
Resource fork successfully read.
Reading map file...
Map file read.
Building symbol table...          Symbol table built.
WARNING: Unloaded jump table entry (seg 5 addr fe20)
WARNING: Unloaded jump table entry (seg 5 addr fe62)
Copying code segments...          1 2 3 4 5 6       Code copied.
Building data segment...          Data segment built and initialized.

Done. - Another image to fix? ('y' or 'n' - default 'n')
```

12. If XMODEM will not be used to download to the target, then makemsr must be run on the dummy programs using the following procedure. (Using XMODEM, one can download the .exe file directly.) Makemsr generates a .msr file from the .exe file that can be downloaded into the DVME-134 system. Launch makemsr by double clicking the makemsr application in the Harmony Tools folder. Makemsr prompts for the .exe file to work on using an SFGetFile box; select the dummy1.exe file in the Derived:harmony:example:dummy:inc:dy134cmac:fast folder. After the file is generated, makemsr prompts for whether another .msr file is to be generated; enter "y" to generate the next file (enter "n" or <cr> to quit).

    The list of files to be run makemsr on is:

```
Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy1.exe
Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy2.exe
Derived:harmony:example:dummy:inc:dy134cmac:fast:dummy3.exe
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy1.exe
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy2.exe
Derived:harmony:example:dummy:inc:dy134cmac:normal:dummy3.exe
```

### D. Building the Srtest Example

13. Using the Finder, create the folder Derived:harmony:example:srtest:inc. While holding the option key down, drag the folder Master:harmony:example:srtest:inc:dy134cmac to the newly created Derived:harmony:example:srtest:inc. This creates an inclusion tree for the srtest applications.

14. Compile srtest using the following procedure. Launch the Consulair Mac C compiler. Use the SFGetFile box to compile the .c files in the Derived:harmony:example:srtest:inc:dy134cmac tree. (See item 6 for details.) There are three versions of srtest: a single processor version in the :single folder, a dual processor version in the :double folder, and a single processor busywait I/O version in the :busywait folder. On the Mac, global data (externs) must be compiled separately from

functions (because externs must be placed in segment 1 in the linked Mac executable); therefore, for example, srtest0 consists of two inclusion files: code0.c and externs0.c.

The list of files to compile is:

Derived:harmony:example:srtest:inc:dy134cmac:busywait:code0.c
Derived:harmony:example:srtest:inc:dy134cmac:busywait:externs0.c
Derived:harmony:example:srtest:inc:dy134cmac:double:code0.c
Derived:harmony:example:srtest:inc:dy134cmac:double:code1.c
Derived:harmony:example:srtest:inc:dy134cmac:double:externs0.c
Derived:harmony:example:srtest:inc:dy134cmac:double:externs1.c
Derived:harmony:example:srtest:inc:dy134cmac:single:code0.c
Derived:harmony:example:srtest:inc:dy134cmac:single:externs0.c

**15.** Link the various versions of srtest using the following procedure. Launch the Consulair Mac C linker. Use the SFGetFile box to link the .link files in the Derived:harmony:example:srtest:inc:dy134cmac tree. (See item 7 for details.)

The list of files to link is:

Derived:harmony:example:srtest:inc:dy134cmac:busywait:srtest0.link
Derived:harmony:example:srtest:inc:dy134cmac:double:srtest0.link
Derived:harmony:example:srtest:inc:dy134cmac:double:srtest1.link
Derived:harmony:example:srtest:inc:dy134cmac:single:srtest0.link

**16.** Run fixaddr on srtest using the following procedure. Launch fixaddr. Fixaddr d[illegible]n SFGetFile box for the file to be converted; select a .out fil[illegible] Derived:harmony:example:srtest:inc:dy134cmac tree. Then enter the text offset for the[illegible] response to fixaddr's prompt (e.g., 0x100500 for a processor 0 image). Finally, a[illegible] conversion is completed, respond to fixaddr's prompt for another file (enter "y", "n[illegible] (See item 11 for details.)

The files to be converted using fixaddr with their corresponding text offsets are:

| | |
|---|---|
| Derived:harmony:example:srtest:inc:dy134cmac:busywait:srtest0.out | 0x[illegible] |
| Derived:harmony:example:srtest:inc:dy134cmac:double:srtest0.out | 0x[illegible] |
| Derived:harmony:example:srtest:inc:dy134cmac:double:srtest1.out | 0x2[illegible] |
| Derived:harmony:example:srtest:inc:dy134cmac:single:srtest0.out | 0x1[illegible] |

Before one runs an application program, it is highly recommended that the stack sizes be checked to ensure that they are large enough. This can be accomplished by running the Harmony bound tool (found on the Harmony Tools disk) on the .exe files for the application. Documentation for the bound tool is found in the Master:harmony:doc:tools folder.

**17.** If XMODEM will not be used to download to the target, then makemsr must be run on srtest using the following procedure. (Using XMODEM, one can download the .exe file directly.) Launch makemsr. Makemsr uses an SFGetFile box to prompt for the .exe file to work on; select a .exe file in the Derived:harmony:example:srtest:inc:dy134cmac tree. After the file is generated, respond to makemsr's prompt for another file (enter "y", "n", or <cr>). (See item 12 for details.)

The list of files to be run makemsr on is:

```
Derived:harmony:example:srtest:inc:dy134cmac:busywait:srtest0.exe
Derived:harmony:example:srtest:inc:dy134cmac:double:srtest0.exe
Derived:harmony:example:srtest:inc:dy134cmac:double:srtest1.exe
Derived:harmony:example:srtest:inc:dy134cmac:single:srtest0.exe
```

## E. Downloading Srtest to a Two Processor DVME-134 System

**18.** Make sure the RS-232 port for processor 0 on the DVME-134 System (J1) is connected to the Mac's modem port using an appropriate cable.

**19.** Launch the inTalk (formerly inTouch) terminal emulator (or an inTalk document) by double clicking on it. Make sure that the *Communications* settings are 9600 baud, 8 data bits, 1 stop bit, no parity, xon/xoff flow control, modem connector, no carrier detect; that the *Terminal Preferences* are set to 80 columns, line wrap OFF, local echo off, incoming cr only, outgoing cr only, incoming lf only, backspace key generates DELETE; that the *Text Transfers* setting is for *Standard Flow Control*; and that the *Binary Transfers* setting is for *inTalk*. These settings can be saved in an inTalk document (select *Save* from the File menu); double clicking this document will then launch inTalk and set all the options appropriately.

A very useful mode of operation for testing Harmony programs is to use MultiFinder to switch among inTalk communicating with the DVME-134 system, the *examine* tool for resolving addresses to symbols (and vice versa) and disassembling code, and QUED to look at the source code. Unfortunately, a 1-Mbyte Mac may not have enough memory to run all these applications at once. However, with MultiFinder, one can quit one application and launch another quickly.

**20.** Press the reset (red RST) button on the DVME-134 system. The following message should appear in the inTalk window:

```
DVME-134 Multiprocessor
Harmony Bootloader   Vers 2.0
NRC Canada

>
```

The ">" is a command prompt. If you enter a "?", the ROM monitor will display a list of the commands that it understands.

**21.** Two methods of downloading are supported by the NRC Boot Loader for the Dy-4 DVME-134: downloading of Motorola S-Records (.msr file) using a straight text transfer and downloading of the binary executable file (.exe file) using the XMODEM protocol. Either method of downloading may be used. The XMODEM download is significantly faster, however.

To download using XMODEM, use the following procedure. Enter "x2" followed by <cr>. This tells the ROM monitor to receive a download of two .exe files. ("x" followed by <cr> downloads to one processor). The ROM monitor will then display the message:

```
Switch to 19.2kbaud, then strike any key to continue:
```

Change the *Communications* baud rate setting in inTalk to 19200 baud, then enter <cr>. The ROM monitor will then display the message:

```
Ready for transfer:
```

Then, select *Send Binary File...* from the *Transfer* menu and using the SFPutFile box, choose (double click) the dummy1.exe file from the Derived:harmony:example:dummy:inc:dy134cmac:fast folder. This sends the file; a progress indicator is displayed along the bottom of the inTalk window during the file transfer as well as a button for stopping the transfer. When the transfer is completed, the ROM monitor will again display the message:

```
Ready for transfer:
```

Next, select *Send Binary File...* from the *Transfer* menu again and this time choose the srtest0.exe file from the Derived:harmony:example:srtest:inc:dy134cmac:single folder. This downloads the single processor version of srtest into processor 0. This time when the transfer is completed the ROM monitor will display the message:

```
Switch to 9.6kbaud, then strike any key to continue:
```

Change the *Communications* baud rate setting in inTalk to 9600 baud, then enter <cr>. The ROM monitor will then display the ">" command prompt.

To download the .msr files, instead, use the following procedure. Enter "d2" followed by a <cr>. This tells the ROM monitor to receive a download of two .msr files. ("d" followed by <cr> downloads to one processor). Then, select *Send Text File...* from the *Transfer* menu and using the SFPutFile box, choose (double click) the dummy1.msr file from the Derived:harmony:example:dummy:inc:dy134cmac:fast folder. This sends the file; a progress indicator is displayed along the bottom of the inTalk window during the file transfer as well as buttons to stop/pause the transfer. Next, select *Send Text File...* from the *Transfer* menu again and this time choose the srtest0.msr file from the Derived:harmony:example:srtest:inc:dy134cmac:single folder. This downloads the single processor version of srtest into processor 0.

22. The .msr file contains a few extra characters at the end that are not recognized by the monitor, which may cause it to dump them on the screen with some ?'s — ignore this, and enter <cr> to get the ">" prompt. Enter "g" followed by a <cr> to start the program running. It should produce output like:

```
parent creates child
child replied 1
child replied 2
    etc.
child replied 100
How many more message exchanges do you want?
```

Entering a number in response to this question causes srtest to exchange the requested number of messages between its main and child tasks outputting the replied result for each exchange.

23. Repeat steps 20–22, but this time download the .exe or .msr files for the dual processor version of srtest:

    Derived:harmony:example:srtest:inc:dy134cmac:double:srtest0.exe
    Derived:harmony:example:srtest:inc:dy134cmac:double:srtest1.exe

    or

    Derived:harmony:example:srtest:inc:dy134cmac:double:srtest0.msr
    Derived:harmony:example:srtest:inc:dy134cmac:double:srtest1.msr

**Title:** Using _Dev_data_table

**Revisions:** 1987-08-19 W. Morven Gentleman
1989-02-06 W. Morven Gentleman

Harmony identifies logical interrupt sources on a given processor by interrupt_id. For many purposes interrupt_id can be regarded just as a symbol, the value of which is an arbitrary integer given by #define; however, it is actually the byte offset of a 4 byte slot in the vector of possible logical interrupts, _Int_table.

The vector _Dev_data_table is provided by Harmony parallel to the vector _Int_table, i.e., for every slot in _Int_table, there is a corresponding slot in _Dev_data_table. The entry in _Dev_data_table is intended to hold a pointer to a device dependent record that contains information unique to the device capable of causing that interrupt. In many cases devices capable of causing different interrupts are independent in the Harmony abstraction, but not in the physical implementation. These are called device groups, and even though there may be no interrupt for the group as a whole, the init_rec for a device that is part of a device group contains a logical interrupt number to identify the device group, as well as providing a member number for this device. The entry in _Dev_data_table for the logical interrupt number for the device group will be a pointer to a device dependent record shared among the device group members. It may be useful for this record to have pointers to the individual device dependent records corresponding to the logical interrupts of the individual members of the group, or for the individual device dependent records to have pointers to the shared record.

To see how the record located through _Dev_data_table could be used, we begin by noting that, for many devices types, there is some data that must be shared between the (second level) interrupt handler and the task that will wait on this interrupt. (This task may simply be a notifier, i.e., a task whose only function is to send a message when an interrupt occurs, but it often has other responsibilities.) A typical example is where the device control register is write only, and the task that will await the interrupt must write to the control register to enable the interrupt, and the interrupt service routine must write to the control register in order to clear the interrupt. If the control register contains controls other than just the clear and enable for the interrupt, both writes must preserve whatever other controls were already set. Because the control register itself cannot be read to determine these other controls, a readable copy accessible both to the task and to the interrupt handler must be maintained. It is important to note in connection with this usage that the reason for shared storage is that the (second level) interrupt handler is not a task and only executes when the interrupt occurs. Therefore, the interrupt handler cannot execute initialization for the device, cannot obtain shared values by message passing, and cannot do other things that could be done by code executed by some task — but it can read and write shared storage.

The first reason why use of the record located through _Dev_data_table may be necessary is to support multiple instances of the same device type. The readable copy described in the previous paragraph could be maintained in a global variable, bound by the linkage editor both to the task and to the interrupt handler; however, if there were more than one instance of the device, so that the task was

instantiated for each instance of the device, the static binding by a linkage editor could not produce a different global variable for each instance, as needed. Indexing by logical interrupt source through _Dev_data_table does provide unique bindings, because the logical interrupt source, interrupt_id, will be unique for each instance. (The actual storage for the shared variable, or more generally for the shared record, may be on the stack of the task, may be separately allocated by _Getvec, or whatever.)

The second reason why use of the record located through _Dev_data_table may be necessary is to support generic devices. Not only do we want to handle multiple instances of the same device type simply by multiply instantiating the controlling task, but we want to use as much common code as possible when handling different device types in the same class of devices. Even handling multiple instances of the same device type requires that configuration parameters that may differ for different instances of the device type, e.g., interrupt_id, must be supplied separately for each instantiation of the controlling task. In Harmony, this is done by initialization records on the init_list argument to _Server_create. Sometimes different device types within the same class can also be accommodated by different parameter values, but often they require code specific to the particular device type. By encapsulating such code in subsidiary tasks such as notifiers, and by having the global indices of the appropriate subsidiary tasks be contained in the initialization records, the actual server code may be valid for any device type in the class. For example, the _Tty_server is written in a way that it can be used with any of a wide class of different serial port designs, although the appropriate _SPi and _SPo tasks for each serial port type depend on the UART used. A communications problem exists, however, because the generic server receives the device specific initialization record and must make this data available to the subsidiary tasks and to the (second level) interrupt handler without even knowing what this data might be. Always copying the initialization record data to a record indexed by _Dev_data_table is a solution that has several advantages, such as not requiring every subsidiary task to allocate stack space or space from _Getvec (as would be required if the initialization record data were passed on by messages) for what is often constant data.

Another aspect of support for generic devices is that the record located through _Dev_data_table can be used to facilitate ancillary tasks accessing the device to control features not supported by the generic server. A case in point is software controlled serial port line speed, which the Harmony supplied _Tty_server does not support because that feature is rarely possible and is rarely needed. Some serial ports do have it, however, and some applications do require it. An ancillary task can easily be provided to change the line speed while the application program is running, requiring only that the ancillary task can locate the data from the initialization record and can have storage associated with the device to record the current state. Both these are readily accomplished with the record located through _Dev_data_table.

The third reason why use of the record located through _Dev_data_table may be necessary is to support thinwire implementations, i.e., where the processors do not share memory. For portability, it is an objective of Harmony that all system supplied code, except for the very lowest level of message passing, is identical for shared memory and thinwire implementations. Since, in general, we cannot assume that the initialization record is on the same processor as the server is, addresses in the initialization record cannot be assumed to be accessible to the server. Thus the readable copy shared variable referred to above cannot be supplied in the initialization record, the initialization record cannot contain functions to be called for device initialization, and so forth. Any data from the initialization record that must be saved for later use by the server must be saved by the server, and any data that must be made available to subsidiary tasks must either be explicitly passed to those subsidiary tasks by message passing or, if it is known that the server, subsidiary tasks, and interrupt handler share address space, can be saved in the record located through _Dev_data_table.

The devices and servers abstractions supplied in Release 3.0 of Harmony illustrate these uses of _Dev_data_table. The use of a readable copy of a write-only register is illustrated in the device abstraction for the Motorola MC6840, in particular in _I_ptm and _PtmCRset. The use of a readable copy of a write-only register, together with sharing the record between two logically independent interrupt sources (the SPI interrupt and the SPO interrupt), is illustrated in the device abstraction for the Motorola MC6850, in particular in _I_mc6850_serial_port, _SPi_mc6850, _SPo_mc6850, and _Mc6850_int. The use of a device group is illustrated in the device abstraction for the Signetics SCN2681, in particular in _I_scn2681_serial_port, _PtTenable, _SPi_scn2681, _SPo_scn2681, _Scn2681_int, _Sspi_scn2681, _Sspo_scn2681, _Ehvtip_scn2681 and _Ss2681_int. These devices can be used with either the _Tty_server or the _UW_server server abstraction. The situation is not entirely satisfactory, however. Initialization is being done by fixed code called from _SPi and cannot be tailored by the init_rec. The Intel 8274, for instance, has two almost independent channels A and B, each of which could be controlled by an instantiation of _Tty_server. However, the *almost* is important because both channels must be initialized in order to use either, and of course once one channel is in use it cannot be reinitialized when the server for the other is created. This can be solved by device groups, so the records located through _Dev_data_table for these two channels point to a shared structure to indicate whether the initialization has been done yet. However, the Intel 8274 is an example where the initialization record should have many more fields than the initialization record for the Motorola MC6850 requires, as there are more options in the UART that must be specified, but the existing generic _Tty_server has no way to pass these fields on to _SPi for it to use in device initialization.

The _File_device_server provides another example of the use of the record located through _Dev_data_table, but not essentially different from what is described above. No thinwire implementation of Harmony is running yet, so we have had no opportunity to confirm that the precautions taken to support thinwire implementations are complete.


# Application Note n009

**Title:** **Programming without End-Of-File**

**Revisions:** 1987-08-26 W. Morven Gentleman
1988-01-25 Darlene A. Stewart (code fragment error)
1989-01-26 Darlene A. Stewart (Release 3.0)

A programmer coming to Harmony from other systems will be surprised to find that Harmony has no concept of end-of-file. The purpose of this note is to explain why Harmony has been designed this way, and to show how the abstraction provided does enable standard programming idioms, albeit done slightly unconventionally.

The Harmony abstraction of a stream, or a file, is a semi-infinite sequence of bytes. That is, there is an initial byte and every byte has a successor byte. All bytes have defined values; after some point, all bytes in the stream are null bytes. A concept of next position in the stream (usually misleadingly called current position) is maintained for any open connection to the stream. The index of some byte in the sequence is recorded as the next position, and input and output occur starting at that byte, advancing the next position. It may be possible to set the next position arbitrarily, but the interface does not make the explicit value of the next position available.

This abstraction is in contrast with the more common abstraction where a stream or file is a finite sequence of bytes where there may be an initial byte and if there is, then there is a last byte, and every byte has a successor byte, except the last byte. Again, any connection to the stream maintains a concept of next position in the stream, but this next position may actually be the index of a byte beyond the last byte. Some systems allow the next position to be the index of any byte beyond the last byte, but others only allow the index of the byte immediately beyond the last byte. When the next position is beyond the last byte, writing produces a new last byte, but reading produces an end-of-file error. This error is detected on some systems (e.g., Pascal) by a function which must be called to ascertain whether reading will produce it, on other systems (e.g., the standard C library function read()) it is detected by a read returning fewer bytes than requested, and on yet other systems (e.g., the standard C library function getc()) it is detected by the returned value being a metacharacter, EOF, that is an illegal byte value.

As an abstraction, the Harmony stream has considerable advantage over the conventional file, because the uniformity of knowing that all bytes in the file always exist and are initialized to null finesses many arbitrary design details that must otherwise be memorized. Examples of questions that are simply not relevant include: whether the next position is constrained to at most one past the last byte, what values are returned for fields when a record is read that is at least partly past the last byte, what happens when a byte is read that was not previously written (as can happen if next position is set arbitrarily), what happens when EOF is "pushed back" on a stream by ungetc(), whether record fields and array elements must be declared as integer rather than character to allow for the EOF metacharacter, how to represent EOF on a stream (such as TTY input) that is naturally infinite, and whether a foreshortened read on a packet stream occurred because of packet forwarding or because the connection was dropped.

However, the primary reason why Harmony uses this abstraction is a low level efficiency issue. Maintaining the end-of-file, i.e., the index of the last byte, is expensive. Of course, incrementing the value of the index is not the problem, but if output is buffered, the current value must be passed when the buffer is flushed, and to be crash resilient the value must be written to disk whenever the buffer is written to disk. File systems based on variable length records, such as VAX/VMS or GCOS, can record end-of-file fairly cheaply in the record control word of the last record. However, file systems such as Multics or Unix or Harmony, where the file contents are a homogeneous sequence of blocks of bytes, need to record it elsewhere, and that involves an extra disk I/O, possibly even an extra seek. Of course any file system will record the physical blocks actually allocated to the file, but this presumably changes much less frequently than the position of the end-of-file.

The obvious question is whether giving up end-of-file results in a significant loss of functionality. Practical experience is that in most cases it does not, either because the end of the data to be read can be determined some other way (as in the Unix a.out executable image format, where a fixed size header describes the sizes of items in the file), or because the file contains text and is terminated by the first null byte in the file, or because processing is sequential and should stop when there are only null bytes left in the file.

Ascertaining that there are only null bytes left in the selected input stream is done by calling the function _Only_nulls_left(). _Only_nulls_left() returns TRUE if the server for the currently selected input stream can prove that all remaining bytes in the stream will be null bytes; it returns FALSE if the server can prove that there exists at least one byte which is not null remaining in the stream, or if the server is unable to establish whether there may be bytes that are not null remaining in the stream. If the input stream were a communications line, for instance, the only circumstance where _Only_nulls_left() returns TRUE would be when the line was disconnected.

As an example, in section 1.5 (page 15) of *The C Programming Language* (B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, First Edition, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.), there is a classic C program:

```
main()          /* copy input to output; 2nd version */
{
    int c;

    while ((c = getchar() ) != EOF)
        putchar(c);
}
```

In Harmony the body of this program would be written as:

```
while( _Put( _Get() ) || !_Only_nulls_left() ) ;
```

As another example, the common operation in Unix of concatenating files makes no sense in Harmony because streams are infinite. However, forming a stream whose contents are a string that is the concatenation of all the strings contained in the given files is straightforward to do in Harmony, and is usually equivalent in effect to the Unix concatenation of files.

An important implementation observation is that most streams or files are represented on block devices, and that different block devices may have different block sizes. For many reasons (e.g., use of DMA

transfers) one would like to move data in integral units of blocks. Consequently, although the last byte is probably in the middle of some block, the whole block will be moved, effectively meaning the file is padded out to the end of block. When copying files between devices with different block sizes, a system with end-of-file inserts or deletes these padding bytes as required. Similarly, when copying files between devices with different block size in Harmony, the number of explicitly represented null bytes in the last block is adjusted as necessary.

Several users of Harmony have implemented servers that provide Harmony streams to files that are actually stored in the file system of some other operating system, for instance, Unix. There should be no problem in such a server supporting the Harmony stream with no end-of-file despite the stored file requiring end-of-file, as long as the server makes appropriate translations.

Another important implementation observation is that it is necessary to be able to preallocate blocks to files, both to avoid fragmentation and to avoid the runtime delays that might occur if blocks were allocated only when they were first written. Allocating contiguous blocks, if possible, is even better. A simple way to accomplish this in Harmony is to seek to the maximum byte position to be preallocated and write a byte, as in the following sample code:

```
_Seek( ucb, desired_size, 0 );
_Put( '\0' );
_Flush();
_Seek( ucb, 0, 0 );
```

A companion to _Only_nulls_left() in the Harmony stream I/O model is the function _Nullify(), which nullifies any (currently) remaining bytes in an output stream. That is, all bytes from the current byte position to the end of the stream become null. This does not affect bytes written in the future, only ones written in the past. This function is most useful for read-write streams (on which _Seek() can be used), as this state is, by definition, always true for write-only streams. This function is particularly useful for the file device server, where it can be used to zero all or a tail portion of a file and to notify the server to release any unneeded blocks (i.e., to shrink the file). (This is actually a case where _Nullify() is useful for a write-only stream. Upon completion of writing data to the file, the client task can call _Nullify() to throw away any extra blocks to minimize disk space.)

As a postscript, in Unix the output functions putc(), putchar(), fputc(), and putw() return EOF on error. In practice, this result is rarely checked, so there is no effective difference from Harmony, where _Put() always returns its argument. Errors in Harmony can be detected by calling _Task_error_code().

Unfortunately, the functions _Only_nulls_left() and _Nullify() were originally thought to be part of the file system and were inadvertently overlooked when the file system was implemented (as were the request messages to the server and the corresponding replies). These functions should be regarded as part of stream I/O; they are supported in Release 3.0 of Harmony, but not in prior releases.


# Application Note n010

**Title:** **Partial Shared Memory Implementation of Harmony**

**Revisions:** 1988-05-20 W. Morven Gentleman
1989-01-23 W. Morven Gentleman

The initial implementation of Harmony assumed all processors saw a single flat shared memory space. A *thinwire* implementation, which assumes no shared memory, has been planned for some time. Between these is a partial shared memory implementation, where the hardware architecture includes a large block of memory that is accessible to all processors (and which all processors see at the same addresses), but where all other memory is private to a particular processor. Making a partially shared memory version of Harmony from the fully shared memory version entails four easy steps.

1. The file /harmony/sys/src/kernel/mmm/romextern.c, for any machine mmm, contains the following pointers that must point at addresses in the shared memory:

```
char     *_MP_gate
uint_32  *_Ltm_id
uint_16  *_Max_task_number
uint_32  *_Loc_directory_id
uint_32  *_Loc_gossip_id
uint_32  *_Loc_first_user_task_id
uint_32  **_Ltm_for_template
uint_32  *****_Td_table
```

   Not only must _Ltm_for_template point to a location in shared memory, but when _I_externs for processor 0 allocates a block of memory which the location in shared memory points to, that block must be allocated in shared memory. Not only must _Td_table point to an array of locations in shared memory, but when _I_td_table and _Grow_td_table put blocks in the td tree for any processor, those blocks must be in shared memory. With some code redesign, it would be possible to have some of these quantities be in private memory instead of shared memory, because once they are initialized they never change, and each processor could keep and refer to its own private copy. This applies to _Ltm_id, _Loc_directory_id, _Loc_gossip_id, _Loc_first_user_task_id, and _Ltm_for_template. Initializing the private copies is the code redesign referred to and is not elaborated here because we are assuming that there is enough shared memory for it to be unnecessary.

2. _Get_td and _Free_td must be changed so that the td is allocated in shared memory.

3. Message copying must be made to work. Of course, if the sender's request and reply messages were always allocated in the shared memory, the current code for _Send and _Receive would work. (The call to _Copy_msg is in _Receive and _Reply, so it does not matter if the receiver's messages are in shared memory because the code will be executed by the receiver.) This would be possible if the sender's messages were always allocated with _Getsvec as discussed below.

However, that would require extensive changes to our code for servers, stream I/O, etc., as well as relying on user code to always use the right allocator, so is probably not practical.

Consequently, we must assume that the sender's request and reply messages are in the private memory of the sender's processor, and the request must be copied into buffers in the shared memory in order for the receiver to read it, and the reply must be copied back from buffers in the shared memory. This is best accomplished by modifying _Send to do this (before and after the call to _Block_signal_processor). The alternative would be to change _Copy_msg to copy packets back and forth by an interrupt protocol using _Block_signal_processor on the receiving processor and _Td_service on the sending processor. I have tried this in the past on other systems I have built, and the overhead is horrendous. (The attraction is that one can get away with two packet buffers per processor pair, and buffer exhaustion cannot happen.)

In modifying _Send to allocate buffers in the shared memory, we need to worry about induced overhead on message passing and about unbounded disable time, but even more we need to worry about properly freeing the buffers, not just after successful send–receive–reply cycles but also if either the sender or receiver is destroyed. This will happen if _Getsvec is implemented in the same way as _Getvec, as suggested below, and if the destruction code in _Local_task_manager also frees this storage. If a different way of managing the shared storage is chosen, care must be taken to guarantee the release of the buffers when a task is destroyed.

4. Minor changes in the debugger, primarily to add messages to the user agents to read and write local memory since the debugger itself will not be able to do this.

### Managing the Shared Memory

The simplest way to manage shared memory would be to duplicate the code for _Getvec, etc., adding an "s" (for shared memory) to the names (e.g., _Getsvec ), and making the _SPool for each processor be a disjoint section of the shared memory. This could be expected to lead to high overheads for memory allocation. Several alternatives exist.

First, caching blocks of popular sizes might make the standard Harmony memory management strategy sufficiently fast. Second, an allocation scheme based on linked fixed-size blocks could be used. Such an allocator can be found in the TCP/IP server under /harmony/sys/src/servers/tcpip/. Of course both _Send and _Receive, or at least _Copy_msg, would need to be changed to cope with messages that are linked segments instead of contiguous memory. Another possible refinement might be to let the receiver free the request message allocated by the sender, and conversely the sender free the reply message from the receiver.

### Doing the Port

It is obvious that as a strategy in doing the port to a partially shared memory machine, the first thing to do is to bring up a system using only the shared memory, i.e., where the pool for each processor is allocated in the shared memory, and only when this is working begin to move the normal pool for each processor into that processor's private memory, leaving the task descriptors and the message buffers allocated in shared memory.

### Downloading

This discussion does not address the question of how programs are to be downloaded into memory prior to execution.

**Title:** **Hardware Configuration of DVME-134**

**Revisions:** 1988-02-19 D. Green
1989-01-26 Darlene A. Stewart (minor edits)

This document describes the configuration requirements for the DY-4 DVME-134 processor boards in order to support the Harmony operating system. In general, the jumpers to be described can be classified as either Harmony related, VMEbus related, board related, or user related. For example, the interrupt configuration jumper (JB10) is Harmony related, VMEbus related, board related, and user related while the EPROM size selection jumper (JB7) is only board related.

**JB1** **Interrupt Acknowledge** *(Harmony, user related)*

Determines the interrupt acknowledge mechanism to be used for each of the seven interrupt levels.

| | | | |
|---|---|---|---|
| required: | JB1 | 13 to 14 | interprocessor interrupt is autovectored at level 6 |
| | JB1 | 11 to 8 | MC68901 mfp chip is autovectored at level 5 (_Tty_server, _UW_server, _Clock_server) |
| optional: | all others | | choice of 901IACK (mfp generated IACK) or VIACK (VME bus vectored) |

**JB2** **VMEbus Arbiter** *(VMEbus related)*

| | | | |
|---|---|---|---|
| required: | JB2 | out | fixed priority bus arbitration |

**JB3** **VMEbus Requestor Mode** *(VMEbus related)*

| | | | |
|---|---|---|---|
| required: | JB3 | in | release on request |

**JB4** **DRAM Access Speed** *(board related)*

| | | | |
|---|---|---|---|
| required: | JB4 | 4 to 5 | 120ns RAMs, see manual for alternatives |
| | JB4 | 6 to 8 | |

**JB5** **VMEbus Time-Out** *(VMEbus related)*

Only applies to processor 0, the system controller board.

| | | | |
|---|---|---|---|
| required: | JB5 | 3 to 5 | 64 µs time-out (processor 0 only!) |

**JB6 BI-mode Master** *(board related)*

A BI-mode master can remove itself from BI-mode. This is required for proper system startup.

| | | | |
|---|---|---|---|
| required: | JB6 | 1 to 2 | is BI-mode master |

**JB7 EPROM Size Selection** *(board related)*

| | | | |
|---|---|---|---|
| required: | JB7 | 1 to 2 | |
| | JB7 | 5 to 6 | 27512 EPROM, see manual for alternatives |

**JB8 CPU Cache Enable** *(board related)*

| | | | |
|---|---|---|---|
| required: | JB8 | out | cache can be software enabled and disabled |

**JB9 Watchdog Time-Out** *(board related)*

Determines the response to a time-out.

| | | | |
|---|---|---|---|
| required: | JB9 | no jumpers installed | no response generated (neither SYSFAIL nor IRESET) |

**JB10 Interrupt Handler** *(Harmony, VMEbus, board, user related)*

Determines hardware interrupt levels for both on board and bus generated interrupts. Additional interrupts may be added by the user.

| | | | |
|---|---|---|---|
| required: | JB10 | 10 to 18 | MC68901 (mfp chip) interrupt to CPU level 5 |
| | JB10 | 12 to 20 | location monitor interrupt (interprocessor interrupt) to CPU level 6 |

**JB11 P2 Reserved Input Enable** *(board related)*

The VMEbus supports single only cycle read–modify–write operations (i.e., the MC680x0 TAS instruction). However, the MC68020 can generate multicycle read–modify–write operations (CAS and CAS2). DY-4 supports these multicycle read–modify–write operations by propagating the MC68020's RMC/ signal onto the P2 backplane on a reserved signal line. Since Harmony does not use r–m–w cycles, this feature can be ignored.

| | | |
|---|---|---|
| | JB11 | do not care |

**JB12 P2 Reserved Output Enable** *(board related)*

See JB11 comments above.

| | | |
|---|---|---|
| | JB12 | do not care |

**JB13 Front Panel Push Button** *(board related)*

Determines whether the push button resets the board or performs no function at all.

| | | |
|---|---|---|
| JB13 | in | generate a local reset only. |

**JB14 VMEbus Request Level Selection** *(VMEbus related)*

Selects VMEbus Bus request level (0 to 3).

| | | | |
|---|---|---|---|
| required: | JB14 | 3 to 5 | BR3* used only. |

**JB15 Bus Grant Selection** *(VMEbus related)*

Determines the bus grant level for this board. Must be identical to the request level chosen by JB14.

| | | | |
|---|---|---|---|
| required: | JB15 | 1 to 2 | pass BG0* |
| | JB15 | 5 to 6 | pass BG1* |
| | JB15 | 7 to 8 | pass BG2* |
| | JB15 | 9 to 11 | monitor BG3IN* |
| | JB15 | 10 to 12 | drive BG3OUT* |

**JB16 System Control Enable** *(VMEbus, board related)*

Determines whether or not this board is THE system controller. The VMEbus requires that the board installed in slot 1 perform the system controller functions. Therefore only one board in a system can be configured as the system controller.

required (see note above):

| | | |
|---|---|---|
| JB16 | in | is system controller (processor 0 only!) |
| | out | is NOT system controller |

**JB17 P2 BITE Output/Termination Enable** *(board related)*

The "Built-In Test Equipment" or BITE* signal is used to force a board into the "Board Isolation Mode" (BI-mode) for diagnostic purposes. We have not incorporated this feature into our systems.

| | |
|---|---|
| JB17 | no jumpers installed |

**JB18 Dual-Port Base Address Selection** *(Harmony, VMEbus, board, user related)*

This feature, which can be implemented by backplane jumpers in place of JB18, permits the selection of the dual-port RAM base address. We have chosen to use JB18 since our applications often require a slightly different configuration for each board within a system. The use of backplane jumpers makes sense when all boards are configured identically except for

their RAM base addresses. This jumper pattern is also used by the common bootROM to automatically determine the board id during system startup.

| | | | Processor# | | | |
|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | etc. |
| required: | JB18 | 1 to 2 | in | in | in | |
| | JB18 | 3 to 4 | out | in | out | |
| | JB18 | 5 to 6 | in | out | out | |
| | JB18 | 7 to 8 | in | in | in | |
| | JB18 | 9 to 10 | in | in | in | |


# Application Note n012

**Title:** **The Harmony TCP/IP Network Service**

| **Revisions:** | 1988-10-25 | R. K. Parr | |
|---|---|---|---|
| | 1988-11-17 | Darlene A. Stewart | (formatting) |
| | 1989-02-06 | Darlene A. Stewart | (minor edits) |

A Harmony TCP/IP server consists of a collection of tasks and primitives which, together, implement and provide a user interface to the ARPANET network architecture. The tasks involved include the _TCP_server, _TCP_timer, _IP_Rx, _IP_Tx, and _IP_timer. By interacting with a peripheral device controller, IP datagrams bearing user data encapsulated within TCP segments are transmitted to and received from a physical network medium. The interface between the tasks of the TCP/IP server and the physical controller is a collection of device-dependent functions.

### The Physical Network Interface

A network interface peripheral is made known to the system by an entry for its interrupt service routine in the _Int_table of the processor which services the device. In addition, the processor must be provided with another table called the _Net_dev_table whose entries are the addresses of the NET_INTERFACE structures corresponding to the network controllers it services. For example, a _Net_dev_table for a processor servicing a single Interlan NI3010A Ethernet interface might be represented as follows:

```
extern struct NET_INTERFACE _Ether_NI3010_interface;

struct NET_INTERFACE *_Net_dev_table[] =
   {
      &_Ether_NI3010_interface
   };
```

The device table index of a network peripheral is the offset of its NET_INTERFACE in _Net_dev_table and is specified in the DEV_TAB_INDEX field of the NET_INIT_REC corresponding to the interface. In the above case, the device table index is 0.

The fields of the NET_INTERFACE are the addresses of a standard set of functions through which the system may interact with the device:

```
typedef  uint_16  (*uint16_funcp)();
typedef  uint_32  (*uint32_funcp)();
typedef  void     (*void_funcp)();

struct NET_INTERFACE
   {
      uint32_funcp   INIT_NET_DEVICE;        /* initialize net device */
      uint32_funcp   GO_ON_LINE;             /* put net station on line */
```

```
    uint32_funcp    GO_OFF_LINE;            /* take net station off line */
    void_funcp      ENAB_RECV_INT;          /* enable device recv interrupt */
    void_funcp      DSAB_DEV_INT;           /* disable device interrupts */
    uint32_funcp    SEND_NET_DATA;          /* send data to net device and tx */
    uint32_funcp    RECV_NET_DATA;          /* recv data from net device */
    uint32_funcp    DECODE_NET_FRAME;       /* decode net frame header */
    uint32_funcp    SGET_NET_ADDR;          /* conv text net address to binary */
    void_funcp      GET_BROADCAST_ADDR;     /* get net broadcast address */
    uint16_funcp    GET_NET_PRO;            /* conv user to net protocol id */
    uint16_funcp    GET_USER_PRO;           /* conv net protocol to user id */
};
```

What is commonly called a "device driver" is therefore, in the Harmony network context, a collection of user-written procedures to implement the functions above with respect to a specific peripheral. The Interlan NI3010A NET_INTERFACE is defined in the file Master:harmony:sys:src:servers:tcpip:ni3010:externs.c as follows:

```
struct NET_INTERFACE _Ether_NI3010_interface =
  {
    _Pwrup_NI3010,              /* initialize net device */
    _GoOnline_NI3010,           /* put net station on line */
    _GoOffline_NI3010,          /* take net station off line */
    _EnabRecvInt_NI3010,        /* enable device recv interrupt */
    _DsabDevInt_NI3010,         /* disable device interrupts */
    _Send_NI3010,               /* send data to net device and tx */
    _Recv_NI3010,               /* recv data from net device */
    _Decode_NI3010,             /* decode net frame header */
    _SGetEtherAddr,             /* conv text net address to binary */
    _EtherBroadcastAddr,        /* get ethernet broadcast address */
    _UserToEtherPro,            /* conv user to net protocol id */
    _EtherToUserPro             /* conv net protocol to user id */
  };
```

Note that the final four functions of the above NET_INTERFACE are not NI3010A-specific but rather are applicable to any Ethernet interface and reside in ethernet.hlib.

If any of the list of functions to be implemented do not apply to a given network device, a null or stub function must be provided in its place in the NET_INTERFACE. Each function is expected to return zero if the call is successful or an appropriate non-zero error code.

It is thus a straightforward matter to interface Harmony TCP/IP to a new physical medium by providing the above set of functions, along with an interrupt service routine, to drive its associated peripheral controller. The functions written for the Interlan NI3010A Ethernet controller also reside in ethernet.hlib.

### The Null Net Interface

A null net interface is also provided for testing of TCP/IP applications before the network interface hardware is available. The _Null_net_interface actually resides in IP's externs file and simply consists

of a list of null-returning stub functions. All IP datagrams destined for either a loopback address (any address with net number 127) or the local host address will pass through a software loopback mechanism at the IP/net level to be received as an incoming datagram by the originating TCP/UDP.

```
extern struct NET_INTERFACE _Null_net_interface;

struct NET_INTERFACE *_Net_dev_table[] =
   {
      &_Null_net_interface
   };
```

While it is possible to have more than one network interface per Harmony system and, indeed, more than one interface serviced by a single processor, the gateway part of the design has not yet been implemented to connect multiple interfaces.

### The Network Interface Initialization Record Parameters

Several of the fields of the NET_INIT_REC are associated with the physical network protocol. IP datagrams are considered to be carried as network packets or frames, encapsulated within a possibly variable-sized header and/or trailer. In the fields NET_PREFIX_MAX and NET_SUFFIX_MAX must be specified the maximum amount of information that may be prepended and/or appended to a user datagram by the network controller. Note that this must include not only the header/trailer data that are actually transmitted in the frame, but also any status information that is required by the network controller for transmission or that is returned by the controller into memory upon reception from the net. These two quantities, together with the net maximum transmission unit as specified in NET_MTU_SIZE, determine the amount of memory that is allocated at initialization time for working buffer space in the _IP_Tx and _IP_Rx tasks.

The maximum transmission unit of the network is the maximum amount of user data that may be transmitted in a single frame. The MTU of the Ethernet, for example, is 1500. The NET_MTU_SIZE specified in the NET_INIT_REC, however, may well be smaller than that which the physical medium may actually be able to carry. An IP implementation must be able to handle datagrams of at least 576 bytes but is not required to accept, and especially ought not transmit, larger datagrams unless explicit knowledge of the capacity of the target host is known. This is especially true if the destination is off-network. However, most hosts on Ethernet are likely to be configurable to larger datagram sizes.

If a network frame that is larger than the MTU specified in the net initialization record is received, Harmony TCP/IP will read it from the net controller in a series of MTU-sized segments and discard each segment until the entire frame has been accepted. It is assumed that the total frame length may be decoded from information in the frame header.

Also included in the NET_INIT_REC are fields for the network physical address length, in octets, and the particular network station address by which the station/interface is known. An Ethernet address is 48 bits or 6 octets in length. The local net address is necessary for the use of the Address Resolution Protocol, the mechanism by which IP host addresses are translated to corresponding physical network addresses. If address resolution is not operative, specification of the local net address is not required.

The NET_STN_ADD field need not be specified as well if the net controller is capable of reading its own factory-programmed address. If so, the device powerup or initialization function, identified in the

INIT_NET_DEVICE field of the net interface, must write the station address into the NET_STN_ADD field of the initialization record. On the other hand if the net station address is specified, one may do so in either text or binary form according to the expectations of the powerup function with the proviso that it must appear in binary form in NET_STN_ADD when the powerup function completes.

Interaction with the network controller and its accompanying interrupt service routine is made possible by specifying the device system bus address in the DEV_BUS_ADD field of the initialization record along with three _Int_table offsets: DEV_RECV_INT, the offset at which _IP_Rx awaits device receive interrupts, DEV_USER_INT, the offset at which the task currently in control of the device, either _IP_Tx or _IP_Rx, waits for device I/O to complete, and DEV_FREE_INT, the offset at which either of the tasks waits for the other to relinquish control of the device. If device status information must be maintained, the offset within _Dev_data_table of the address of such data is specified in the DEV_DDT_OFF field. For the Interlan NI3010A, for example, this is a copy of the 8-bit write-only interrupt enable register.

The task indices of _IP_Tx and _IP_Rx are specified in the IP_TXTASK_INDEX field and the IP_RXTASK_INDEX field of NET_INIT_REC. These two tasks are created by the TCP task at initialization time and must be assigned the same task priority levels and be instantiated on the same processor since they both interact with the net interface device whose interrupt is wired to a single processor.

Finally, several toggle switches relevant to the operation of the physical interface may be specified in the NET_MODE_FLAGS field:

```
#define  _REVERSE_ARP_ENABLED       0x00000100
#define  _MODULE_LOOPBACK_MODE      0x00001000
#define  _INTERNAL_LOOPBACK_MODE    0x00002000
#define  _HW_LOOPBACK_DISABLED      0x00004000
#define  _NET_BROADCAST_DISABLED    0x00008000
#define  _SET_PROMISCUOUS_MODE      0x00010000
```

The Reverse Address Resolution Protocol (RARP) is a mechanism by which hosts may dynamically discover their own IP addresses at initialization time from a host-to-net address translation cache that is maintained at a remote central location. In the Harmony context this facility may be especially useful in that it would enable the specification of a general and ROMable network initialization record applicable over many hosts. If the _REVERSE_ARP_ENABLED switch is set, the _IP_boot procedure will call the primitive _IP_host_addr with a zero address as argument, thereby triggering the broadcast of a RARP request onto the network referencing the local station address. The remote RARP server, having resolved the net address to an IP host address, will then send the appropriate RARP reply. Note, however, that at the current time a Harmony host is not capable of acting as a RARP server, only as a RARP client.

Under certain test conditions it may be appropriate to operate the network interface in a hardware loopback mode in which all send requests are looped back within the net controller and/or physical medium access hardware. The _MODULE_LOOPBACK_MODE switch and the _INTERNAL_LOOPBACK_MODE switch are significant to the Interlan NI3010A Ethernet controller. Unless otherwise specified by mode switches like those above or the presence of the null net interface, the default operating mode is online.

While online a loopback function is still required for frames destined for the local host. Some net interfaces, however, such as serial line drivers may not be capable of operating in loopback mode. In those cases where hardware loopback is not possible, setting the _HW_LOOPBACK_DISABLED switch will enable the software loopback mechanism.

For network media that do not support broadcasting or for point-to-point links on those that do but on which broadcasting is unneeded, broadcasting may be disabled by setting the _NET_BROADCAST_DISABLED toggle.

In order to act as a network monitor it may be appropriate for a station to receive frames destined for any net address. This behaviour is known as *promiscuous mode* and may be specified, if appropriate for the underlying medium, through the _SET_PROMISCUOUS_MODE switch. Currently, however, Harmony TCP/IP does not support frame-level monitoring.

### The Internet Address

An IP host address is a 32-bit field that is conceptually divided into two variable-size subfields, the most significant representing the *network number* and the least significant the *local host address*. The division takes five forms or classes, denoted "A" through "E".

Textually, an IP address is expressed in the *dotted decimal* notation, in which the 32-bit address is divided into four 8-bit fields, each specified as a decimal number and separated by periods. The IP address of NOAH.ARC.CDN, for example, is 128.144.1.10 in this notation.

The class of an IP address is easily recognized by the value of its leftmost field in the above notation. A class A address has an 8-bit network number and a 24-bit local address. The most significant bit of the network number is set to 0 thus allowing a 7-bit effective network number or 128 class A networks, numbered 1 through 126. Net number 0 is reserved and 127 denotes a loopback address.

A class B address has a 16-bit net number and a 16-bit local address. The two highest-order bits of the net number are set to 10, thus allowing a 14-bit effective net number. In the dotted decimal notation, a class B address is recognized by its leftmost field being in the range 128 to 190. Net number 191 is reserved. Currently only class B addresses of the form 128.rrr.rrr.rrr have been assigned.

A class C address has a 24-bit net number and an 8-bit local address. The three highest-order bits of the net number are set to 110, thus allowing a 21-bit effective net number. In the dotted decimal notation, a class C address is recognized by its leftmost field being in the range 192 to 223. Currently only class C addresses of the form 192.rrr.rrr.rrr have been assigned.

Class D addresses, with the four highest-order bits set to 1110, are multicast addresses. Class E addresses, with the four highest-order bits set to 1111 are reserved.

The local host IP address is specified in the IP_HOST_ADDR field of the NET_INIT_REC in the dotted decimal notation and is converted internally to the 32-bit binary format at initialization time. While it may be desirable in some cases for a host to have more than one IP address associated with a given physical interface, Harmony TCP/IP currently provides for the specification of only one local host address.

### Explicit Subnetting

A network consisting of an interconnected collection of local nets may be explicitly subnetted by assigning certain bits of its host address field to represent a *subnet number* within the larger network. The remaining bits of the host field then represent the local host address within the subnet. This technique is used to prevent the explosion in size of routing tables, as all host addresses on a given subnet may be routed by a single entry. The class B networks of medium-to-large-scale organizations are the usual candidates for explicit subnetting. With any subnet may be associated a subnet mask: the logical AND of an IP address with the subnet mask yields the subnet number. Suppose it is agreed that the most significant eight bits of the host address field of a given class B network are to represent a subnet number. Thus each subnet has an 8-bit local host address field and the subnet mask would be 255.255.255.0. If explicit subnetting is desired, the subnet mask may be specified in dotted decimal notation in the field IP_SUBNET_MASK of the NET_INIT_REC.

The subnet mask may also be discovered dynamically at boot time by setting the bit switch _REQUEST_SUBNET_MASK in the NET_MODE_FLAGS field of the initialization record. If this toggle is set, _IP_boot calls the _IP_addr_mask primitive with a mask argument of 0, thereby generating a request to the TCP/IP server to broadcast an address mask request message in the Internet Control Message Protocol (ICMP). Any host that hears the request and is authorized to respond (usually a gateway) will send the appropriate ICMP mask reply. Such authorization may be granted to a Harmony host by setting the _REPLY_SUBNET_MASK switch in the NET_MODE_FLAGS field. If a host is unable to get a response to an address mask request it assumes that explicit subnetting is not enabled. This may, however, not be true if the host that replies to mask request broadcasts happens to be down. It is common, therefore, for a host that services mask requests to broadcast a mask reply to its subnet when it boots, thereby turning on subnetting for all its client hosts. This action may be enabled by setting _BROADCAST_SUBNET_MASK in the NET_MODE_FLAGS field of the initialization record. Clearly, dynamic address mask discovery is only applicable for broadcast networks.

### Broadcast Addresses

Note that for all address classes, host addresses consisting of all 0 bits and all 1 bits have special meanings. A network number concatenated with a host address of all 1's represents the internet broadcast address for that network. This address will be resolved to the broadcast address of the underlying technology and will be propagated by gateways throughout the domain of the network. Similarly, a subnet number concatenated with a subnet address of all 1's represents the subnet broadcast address and will not be propagated beyond the extent of the subnet. An IP address of all 1's, or 255.255.255.255, is called the local broadcast address and is not propagated by any host or gateway. No broadcast address of any form may appear in the source host address field of the header of an IP datagram. IP addresses in which the net number and/or host address fields are zero may not be used except as the source address of a datagram generated by a dynamic discovery procedure to determine a host's address. For example, the Bootstrap Protocol (BOOTP) is an alternative to the combination of a RARP request and an ICMP address mask request and employs the User Datagram Protocol (UDP) to broadcast an IP datagram to a remote BOOTP server listening at a well-known UDP port. Since the source host address is likely not known, the source address field of the datagram is set to zero.

### Gateways

If a network is not isolated, a mechanism must be provided to allow datagrams bound for hosts on indirectly connected networks or subnets to be routed to intermediate hosts called gateways that bridge

adjacent networks to form the links in a path between the source and target hosts. A gateway forwards a datagram destined for a remote host by consulting a routing table of entries each of which represents a destination net/subnet number or host address. The target address of a datagram is matched against the keys of the routing database and the next directly connected network and destination host is selected from among possibly several choices. The intelligence upon which this decision is based is provided by the exchange of information with neighbouring gateways in an agreed manner such as that of the Routing Information Protocol (RIP). It is often sufficient for a host to know only the address of a nearby gateway to which it routes, by default, any datagram destined for an address that is not on the local net or subnet. The address of a default gateway for a Harmony host may be specified in dotted decimal form in the IP_DEFAULT_GWY field of the net initialization record. More complex routing capability is not possible at this time, although provisions for the future addition of routing and gateway functions have been allowed for in the overall design.

### Routing

It is envisioned that in the Harmony network context, such routing functions would be provided by the addition of two new tasks, a routing task and a gateway task. It is clear that some hosts may require more complex routing capability than that afforded by knowing a default gateway address yet not be gateways themselves. Indeed, many hosts are RIP *listeners*, using the protocol only to gather information about neighbouring gateways while not broadcasting any intelligence of their own. The routing task would be capable of accepting and parsing textual route specification strings and inserting the compiled entries into a routing database. If the destination address of an outbound TCP or UDP datagram is not on the attached net or subnet and there is no default gateway, then the TCP server will send the datagram to the routing task for identification of the next destination and the appropriate interface through which it should be transmitted. The task id of the routing task would be made known to the TCP server courtesy of its associated IPTX/IPRX task pair. At initialization time, IPRX attempts to connect to the routing task and, if successful, passes its identity back to TCP as part of the startup procedure.

If a host is to perform gateway functions as well, the routing task would be passed the task index of, and in turn create, its associated gateway task at initialization time. The gateway task would be instantiated on the same processor as the routing task and be granted read access to the routing database which would be owned by the latter. A datagram whose final destination is not the local host address would be passed on by IPRX to the gateway task for further routing.

When the IPRX task receives a datagram whose destination address is not the local host address, it would be passed on to the gateway task. The gateway task, upon consulting the routing tables it shares with the routing task would then forward the datagram out the appropriate interface or, alternatively, deliver it to a TCP task attached to another interface within the same node. Similarly, messages may be exchanged between applications connected to different TCP/IP servers attached to different net interfaces within the same node. A datagram bound for a different, but internal, host on a different, but attached, network would be sent by the routing task to the appropriate net interface but marked as a loopback request on that link.

The routing task would also be responsible for both processing and generating ICMP redirect messages. As the routing task has not as yet been implemented, Harmony TCP/IP currently does not respond to ICMP redirects.

### The Internet Control Message Protocol

Within the gateway task would also be situated much of the machinery of the Internet Control Message Protocol (ICMP). Error messages regarding unreachable hosts and networks, datagram header and fragmentation problems, time-to-live expirations, and gateway redirects would be handled by the gateway task. As we have already noted, subnet mask requests and replies are also part of ICMP but are implemented within the host IPTX/IPRX task pair, as is the ICMP echo reply function. Currently there is no local user ICMP interface so echo requests cannot be generated from a Harmony TCP/IP host. Such a capability would be an addition to the TCP task and would supplement its existing TCP and UDP interfaces. The only other ICMP functions that a host, rather than a gateway, might implement are the port and protocol unreachable error messages. These are not supported at this time.

### The Routing Information Protocol

A possible implementation of RIP would consist of two tasks, a sender and a receiver, for each net interface. As RIP is UDP-based and uses UDP port 520, the sender would open a UDP connection specifying port 520 and the local broadcast address as destination socket while the receiver would open a connection with port 520 and the local host address as the source socket. Both tasks would also open connections to the routing task in order to update the database with newly-received information and to obtain routing data for broadcast to other gateways.

### Transparent Subnetting

Another approach to routing and subnetting that does not involve an explicit routing protocol or separate routing tables is the notion of *transparent subnetting* or *proxy ARP*. In this scheme the gateways of an interconnected set of physically distinct subnets conspire to lead the hosts of the net to believe that the overall topology of the net is a single broadcast domain. When an ARP request is broadcast by a host, each gateway on that subnet rebroadcasts the request on each of its connected subnets. In this way the ARP request is propagated throughout the network. When the target host finally receives the request and responds, the ARP reply will be sent to the last gateway that rebroadcast the request which, in turn, caches the address translation and resends the reply to the gateway/host from which it originally received the request but with its own physical net address in place of that of the responding host. Ultimately, the requesting host will receive an ARP reply resolving the target IP address to the net address of the gateway that last handled the reply. This gateway is thus acting by proxy for the destination host. The proxy ARP mechanism also implements routing in the form of the sequence of cached proxy address translations in the gateways that separate the source and destination hosts. If an incoming ARP request received by the Harmony IPRX task is not for the local host address and there is a proxy ARP task present, then the request will be forwarded to it for further processing. This task would attempt to resolve the target host address in the address translation cache associated with each of its directly connected interfaces. If successful, proxy ARP would respond to the requestor with the appropriate translation. If resolution failed, however, proxy ARP would then rebroadcast the request on each of its attached links and queue an entry referencing the target host address. This entry would be granted a certain lifetime and would prevent the regeneration of the request if a loop in the network topology resulted in the rebroadcast eventually returning to the gateway. The proxy ARP task could replace the routing task in the Harmony network architecture (the IPRX task of each net interface would then connect to it as if to the routing task) or, perhaps, coexist with the routing task.

### The IP Checksum

The IP component of a TCP/IP host which does not implement routing and gateway functions is fairly modest. Its main task is to prepend a TCP or UDP segment with the IP header to form an IP datagram. Since the interface between the transport (TCP) and the network (IP) layers requires the specification of most of the information from which the IP header is constructed, the IP header is actually formed by the Harmony TCP task. It falls to the IPTX task, then, only to calculate the IP header checksum and to obtain the resolution of the destination IP address to its associated physical address, either from the ARP cache or by initiating an ARP request. Calculation of the header checksum may be disabled by setting the _IP_CHECKSUM_DISABLED switch in the NET_MODE_FLAGS field of the net initialization record. This should be done only if it can be guaranteed that the networks that any datagram will traverse implement an acceptable form of error checking at the data link level. This, for example, is true of Ethernet which generates, and attaches as a trailer, a 32-bit CRC for every frame.

### The Address Resolution Cache

Address resolution is enabled by specifying a non-zero ARP address translation cache size in the ARP_CACHE_SIZE field of NET_INIT_REC. The ARP_HW_TYPE is a code that identifies the physical network and is carried in the ARP/RARP packet. The most common medium on which ARP is implemented is Ethernet and its hardware type code is 1. ARP cache entries are allotted a certain lifetime, measured in minutes, in order to allow the changing of network interface controllers for maintenance and repair purposes without requiring manual intervention in the translation caches of all the hosts that may have been communicating with an affected host. A one-second interval timer task, _IP_timer, is provided for the IP host tasks and each cache entry's lifetime field is decremented on the timer's regular ticks. The task index of the timer is specified in the IP_CLOCK_INDEX field of the initialization record.

Address translations may be manually entered into the cache with a call to the _IP_bind_addr primitive, with the name of the net interface as known to the Directory task and a text string representing the entry as arguments. The latter consists of an IP address in dotted decimal form followed by the physical network address in whatever notation is expected by the format conversion function for that interface as identified in its NET_INTERFACE.

```
uint_32 _IP_bind_addr( net, addr )
        char *net;
        char *addr;
```

For example, an IP-to-Ethernet cache entry might appear as follows:

```
_IP_bind_addr( "NET:", "128.144.1.170 02:07:01:00:38:82" );
```

### Fragmentation and Reassembly

The IP timer is also used to time out the reassembly of received fragmented datagrams. Since a given datagram may traverse networks of different physical technologies en route to its destination, it may have to be repackaged at a gateway into a sequence of smaller fragments, each of which becomes a datagram in its own right, if the size of the original datagram is larger than the maximum transmission unit of the next link. Indeed, fragments may even have to be fragmented. The reassembly of all the resulting fragments is done in the destination host and has 15 s to complete before dynamic resources

acquired for the process are released. The fragments of a datagram in reassembly are identified by their source and destination addresses, the higher-level protocol, and the datagram id carried in the IP header. Harmony TCP/IP currently allows up to four datagrams to be in reassembly concurrently.

While fragmentation is usually done in a gateway a host may also have to perform what is termed *source fragmentation*. For example, the downloading of a host at boot time using the Bootstrap Protocol (BOOTP) and the Trivial File Transfer Protocol (TFTP) is a common procedure. In TFTP file blocks are transferred in 512-byte segments with the process completing when the target host receives a block of less than 512 bytes. Clearly a host must be able to send a 512-byte datagram even when the MTU of the physical network is smaller than this, as in the SATNET or packet radio networks. If the 512-byte blocks of the sending host are appropriately fragmented, the pieces will be reassembled at the receiving end into the proper segment size required by TFTP.

### The Maximum Datagram Size

Fragmentation and, especially, reassembly are expensive procedures so it is wise to avoid them if possible. This can be accomplished, in part, by a judicious choice of the maximum datagram size to be handled by the host and specifiable in the IP_MAX_DGM_SIZE field of the initialization record. The minimum value for this field is 576: every IP host must be able to receive a datagram (header plus data) of at least this size and should not send a datagram of greater than this size unless explicit knowledge of the capacities of all intermediate and final destination hosts are known. For example, if all the hosts on a given Ethernet cable are capable of accepting datagrams up to the size of the Ethernet MTU of 1500 bytes, then the IP_MAX_DGM_SIZE can safely be set to this value with the knowledge that datagrams bound for destinations off the subnet may be fragmented at the local gateway.

### Connecting Directly to the IP Interface

Currently it is not possible for an application to connect directly to the IP level of the Harmony TCP/IP service. Such a capability, however, would be useful for several reasons and worthy of consideration for future releases. This would make it possible for transport-level protocols other than TCP and UDP to be implemented and tested on top of the IP datagram service. This was, in fact, one of the prime motivations behind the UNIX TCP/IP *raw socket* service. Also, this would make it possible for an ICMP task with its own user interface to be developed in order to implement the Packet InterNet Groper (PING) facility which makes use of ICMP echo and is often used for network reachability tests and debugging.

Currently only one TCP task can be associated with a single IP net interface. By enabling the IP level to accept open requests it could be arranged that more than one TCP task be attached to a given interface, thereby implementing the concept of multiple logical hosts or *multihoming*.

The converse of this would be the ability to associate one TCP task with more than one physical interface. An example of this would be the use of several serial interfaces to increase the bandwidth of a point-to-point link. This feature could be implemented by moving the interface-specific information in NET_INIT_REC to NET_INTERFACE, that is, the device bus address and its _Int_table and _Dev_data_table offsets.

### Creating a TCP Server

A TCP server is created by a call to _Server_create, specifying an initialization record of structure NET_INIT_REC and the TCP task index as arguments. The NET_INIT_REC is of form:

```
struct NET_INIT_REC
  {
    struct INIT_REC   STD_INIT;                 /* standard init rec header */
    char              NET_DIRECTORY_NAME[ 32 ]; /* directory name */
    uint_32           DEV_TAB_INDEX;            /* device table index */
    char              *DEV_BUS_ADD;             /* device bus address */
    uint_32           DEV_RECV_INT;             /* device receive data interrupt */
    uint_32           DEV_USER_INT;             /* user data transfer interrupt */
    uint_32           DEV_FREE_INT;             /* user device free interrupt */
    uint_32           DEV_DDT_OFF;              /* device status block ddt offset */
    uint_16           NET_PREFIX_MAX;           /* max data prefix */
    uint_16           NET_MTU_SIZE;             /* net max user data */
    uint_16           NET_SUFFIX_MAX;           /* max data suffix */
    uint_16           NET_ADD_LEN;              /* net address length */
    char              NET_STN_ADD[ 32 ];        /* net station hardware address */
    uint_32           NET_MODE_FLAGS;           /* net mode flags */
    uint_32           IP_TXTASK_INDEX;          /* ip tx task index */
    uint_32           IP_RXTASK_INDEX;          /* ip rx task index */
    char              IP_HOST_ADDR[ 16 ];       /* in ip four-tuple form */
    char              IP_SUBNET_MASK[ 16 ];     /* in ip four-tuple form */
    char              IP_DEFAULT_GWY[ 16 ];     /* in ip four-tuple form */
    uint_32           IP_MAX_DGM_SIZE;          /* max ip dgm (at least 576) */
    uint_32           IP_CLOCK_INDEX;           /* ip interval timer task index */
    uint_32           ARP_HW_TYPE;              /* arp hardware type */
    uint_32           ARP_CACHE_SIZE;           /* arp cache max size */
    uint_32           TCP_TASK_INDEX;           /* tcp task index */
    uint_32           TCP_CLOCK_INDEX;          /* tcp timer task index */
    uint_16           TCP_USERS_MAX;            /* maximum connection count */
    uint_16           TCP_SEND_QUEUE_SIZE;      /* max segs queued for send */
    uint_16           TCP_RECV_QUEUE_SIZE;      /* max out-of-order segs qu'd */
    uint_16           TCP_WINDOW_SIZE;          /* max data in sndq or rbuf */
    uint_16           USER_MSGIO_MAX_DATA;      /* max data in user msg io rqst */
    uint_16           USER_STREAM_BUF_SIZE;     /* harmony stream buf size */
    uint_16           TCP_BUFF_DATA_SIZE;       /* segmented buffer data size */
    uint_16           TCP_TRACE_BUF_SIZE;       /* max trace buffer entries */
  };
```

The NET_DIRECTORY_NAME is the name that is reported to the Harmony Directory task and the name referenced by all application tasks wishing to open connections to the server and gain access to its services.

### The _IP_boot Primitive

Once the server has been successfully created, the net interface to which it is connected must be turned online by a call to the _IP_boot primitive, with the net initialization record and the server's task id as arguments. _IP_boot is a convenient mechanism for accomplishing two objectives. First, if more than one interface is being created in a gateway, it allows each of the TCP/IP interfaces to be created and to connect to the gateway-routing server before any of them go online. As routing entries may be

generated when each TCP/IP connects to the routing server, this ensures that at least the local routing tables are complete before any of the interfaces begin handling datagrams. Second, _IP_boot provides a framework for the initiation of boot-time network functions such as acquisition of the local host address or the subnet address mask and, especially, for the possible future implementation of the Bootstrap Protocol. A new _IP_boot could be written that executes BOOTP and, having extracted the local host address and the subnet address mask from the boot server reply, can then set these values for the local interface by calling the _IP_host_addr and _IP_addr_mask primitives with the appropriate arguments.

```
uint_32 _IP_boot( net, id )
        struct NET_INIT_REC *net;
        uint_32 id;

uint_32 _IP_host_addr( id, addr )
        uint_32 id;
        uint_32 addr;

uint_32 _IP_addr_mask( id, addr_mask )
        uint_32 id;
        uint_32 addr_mask;
```

In the above three primitives, the id argument refers to the task id of the TCP server.

### Opening a Connection to the TCP Server

The large majority of relationships between two TCP users will be that of client and server so the format for opening a Harmony connection to the network service at either end and then establishing a TCP connection between the two correspondents reflects that relationship. A server at a particular host *listens* at a TCP port whose number is widely known to be associated with the service it provides while, conversely, a remote client wishes to *connect* to that port at that host for the purpose of obtaining that service. The open directive to the Harmony TCP service for creating the client–server connection usually will assume the following two forms:

*Client:* `ucb = _Open( "net name: port number@host address +tcp", 0 );`

*Server:* `ucb = _Open( "net name port number +tcp", 0 );`

A TCP connection is identified by a unique pair of source and destination sockets, each socket consisting of a 32-bit IP address and a 16-bit port number. Certain port numbers have already been assigned to well-known services such as the Telnet Protocol which is a TCP application and is associated with port 23. Port numbers 0 through 255 are reserved for assignment by the Internet Network Information Centre. User-defined port numbers are best assigned from the range 256 through 1023 although care must be taken as many of these ports have also already been allocated to various other standard services. Port numbers 1024 and above are (generally) unreserved and recyclable.

The port number is expressed as a decimal number in the range 0 to 65 536 and the host address in the dotted decimal notation. Assuming that the Harmony TCP/IP server is known by the directory name "NET:", the open directives for a client wishing to interact with a server at TCP port 1234 on host 128.144.1.170 would be:

*Client:* ucb = _Open( "NET: 1234@128.144.1.170 +tcp", 0 );

*Server:* ucb = _Open( "NET: 1234 +tcp", 0 );

### The _Listen and _Connect Primitives

Note that for the server no host address need be specified in the directive: the local host address is assumed by default. The _Open primitive does not generate any network traffic, only establishing a connection between the calling task and the local server and initiating the acquisition of connection resources. The server task declares itself available for service by issuing a call to the TCP _Listen primitive:

```
uint_32 _Listen( ucb );
    struct UCB *ucb;
```

*Server:* err = _Listen( ucb );

The primitive blocks the caller until such time as a TCP connection has been successfully established with a remote correspondent. For its part, the client task solicits a connection with a remote server by issuing a call to the TCP _Connect primitive:

```
uint_32 _Connect( ucb );
    struct UCB *ucb;
```

*Client:* err = _Connect( ucb );

On behalf of the client, the TCP server selects an unused port number in the unreserved range as the source port and completes the source socket with the local host address. The _Connect request is immediately rejected if the destination socket is not fully specified with both target host and port. _Connect blocks the caller until a TCP connection has either been established or fails due to time-out. If the target is down and there is no response to the TCP synchronization (SYN) segment, the SYN will be retransmitted MAXRETRANSMISSIONS times and then return a TIMEOUT error indication. If the host is up but the target server is not listening at the the specified port, a reset segment will be returned to the local host. In response to this, the _Connect primitive will delay CONNRETRYMSECS and then retry the connection for a total of CONNMAXTRIES before returning a RESET indication. These manifests are defined in the include file tcpipuser.h.

### The _IP_send Primitive

Having successfully established a TCP connection with the remote server, the client task may now send data to the server via the _IP_send primitive. The call returns immediately with an indication of the success or failure of the send request. Note that the successful completion of the _IP_send call does not imply that the user data have been received by either the remote TCP or the remote correspondent or even that they have yet been sent, but rather that internal connection resources are sufficient for the buffering and queuing of the data.

```
uint_32 _IP_send( ucb, buff, len, flags );
    struct UCB *ucb;
    struct BUFF_MSGIO *buff;
```

```
uint_16 len;
uint_16 flags;
```

The len bytes of user data to be transferred are expected to be in the text buffer of the BUFF_MSGIO structure, buff. The flags argument will be discussed shortly.

```
struct BUFF_MSGIO
  {
    struct STD_MSG  BUFF_HDR;      /* message header */
    uint_32         BUFF_CON;      /* connection number */
    uint_16         BUFF_LEN;      /* data length */
    uint_16         BUFF_FLG;      /* tcp flags */
    char            BUFF_TEXT[1];  /* user data */
  };
```

### The _IP_recv Primitive

Symmetrically, the completion of the _Listen primitive indicates to the server that a remote client has connected and will presently be sending a request. The server awaits the arrival of data from the client by issuing an _IP_recv request to the local TCP server:

```
uint_32 _IP_recv( ucb, buff, len );
    struct UCB *ucb;
    struct BUFF_MSGIO *buff;
    uint_16 len;
```

The space into which the data are to be transferred is the text buffer of buff and is expected to be of length len.

The flags argument of _IP_send may be specified as 0, PUSH, URGENT, or PUSH | URGENT. With the URGENT flag, TCP offers a weak mechanism for initiating the processing of *out-of-band* data. An URGENT indication returned in the BUFF_FLG field after an _IP_recv should prompt the user to process the received data with dispatch and continue issuing _IP_recv requests until the URGENT flag is turned off. Unless the sender specifies the PUSH flag, an _IP_recv call does not return until len bytes of data have been received by the local TCP. Data received under the PUSH flag can satisfy any _IP_recv request. The number of bytes delivered is returned in the BUFF_LEN field of the BUFF_MSGIO specified in the _IP_recv. Note that the PUSH indication, however, is only of significance to the sender and is not returned in BUFF_FLG following _IP_recv.

### Closing a TCP Connection

When the client has no more requests to make of the server, the closing of the connection is initiated by a call to the Harmony _Close primitive which causes a TCP termination or FIN segment to be generated. A PUSH of any remaining undelivered data is implied when the FIN segment is received at the remote TCP. An _IP_recv request issued by the server against a closing connection with an empty receive buffer returns an error indication of END_OF_DATA. This should prompt the server to _Close his end of the connection, causing it to enter the last phase of the graceful termination sequence. Following the exchange of segments involved in the closing protocol and the release of all dynamic connection resources, a call to _Close returns.

Depending on the circumstances, a _Close may not initiate the graceful, bilateral closing sequence but unilaterally abort the connection with a TCP reset segment. The latter is the case if deliverable data remain in the connection's receive buffer or if the send queue is full and the termination (FIN) segment cannot be queued. Note that in future Harmony TCP releases this second condition should be removed so that a connection would be aborted only if internally buffered data had not been received. A connection which has been reset enters the *closed* state and all its dynamic resources, except the connection data block itself, are released. Subsequent user requests against such a connection will be returned with an appropriate error indication. The user should then _Close the connection.

### Call Queuing

Only the task that opens a connection to the Harmony TCP server may issue _IP_send and _IP_recv requests against it. Consequently, a server which must be able to accept and service requests from many remote clients concurrently must create an agent task to open the TCP server, listen for a connection, and receive requests. When a connection is received, the worker would report the event to the server which, in turn, would create a new worker to listen for further connect requests. This clearly leaves a small window during which no agent of the server is listening for connections. A connect request received by the TCP server during this window would cause the generation of a TCP reset segment to its remote peer. If the remote TCP was in a Harmony node, the retry mechanism of the _Connect primitive would simply generate another connect request after an interval. As this practice is wasteful of connect requests and does not guarantee that the remote client will ever eventually get connected to the server, *call queuing* support should be added to future Harmony TCP implementations.

### TCP Initialization Record Parameters

Several TCP-related parameters are included in the NET_INIT_REC for user-specification. The _TCP_timer is a 128-ms interval timer used in counting down queued segments for retransmission, connections in time-wait state, and burst-mode delayed acknowledgements. Its task index must be provided in the TCP_CLOCK_INDEX field of the initialization record and a corresponding entry made in _Template_list. The TCP clock task is created by the TCP server task as part of the initialization procedure. The TCP_SEND_QUEUE_SIZE limits the number of outstanding, unacknowledged segments on any given connection. The maximum TCP segment size is calculated as the maximum datagram size minus the standard TCP header size (20). If the maximum user data buffer that may be sent to, or replied from, the TCP server as specified in the USER_MSGIO_MAX_DATA is larger than the maximum TCP segment size then a user _IP_send request will be segmented into smaller pieces of appropriate size. Note then, that the TCP send queue size limit may be reached before the same number of user _IP_send's have been issued.

The maximum amount of data that may be queued on a given TCP connection is further limited by the parameter TCP_WINDOW_SIZE, in bytes. _IP_send requests are rejected after this amount of data has been queued or TCP_SEND_QUEUE_SIZE has been reached, whichever comes first. Although a TCP connection is considered to be an unstructured, bidirectional, sequenced data data stream, outbound TCP data are queued as whole segments for retransmission purposes. The Harmony TCP retransmission policy is *first only*, with out-of-order segments being queued internally. This policy minimizes the amount of retransmission traffic. The maximum number of segments that may be retained on the out-of-order receive queue is specified as the TCP_RECV_QUEUE_SIZE field of NET_INIT_REC. The unstructured nature of inbound TCP data is, however, retained as all received data are queued FIFO in an internal buffer whose maximum size is TCP_WINDOW_SIZE.

In order that all data buffering and unbuffering be done bounded time and to prevent excessive fragmentation of the system dynamic memory pool, all buffered data are kept in linked chains of fixed-size buffers acquired at initialization time and allocated from an internally-managed list. The size of these fixed-size links is specified in the TCP_BUFF_DATA_SIZE field of the NET_INIT_REC. Typical choices for this parameter would be 32 or 64.

The TCP header of each TCP datagram that is transmitted contains a checksum that is computed over the TCP header and the user data as well as a 12-byte conceptual *pseudo-header* that includes the source host address, the destination host address, the transport protocol number, and the TCP length (header plus data). If it is desired to disable the calculation of the checksum, the _TCP_CHECKSUM_DISABLED switch may be set in the NET_MODE_FLAGS field of the initialization record.

### Harmony Stream I/O Interface to the TCP Server

As an alternative to the message-oriented I/O implemented by _IP_send and _IP_recv, data may be transferred to and from a TCP connection using the primitives of the Harmony stream I/O model. A TCP stream I/O connection is strictly one-way, that is, it may be opened either for read or for write but not both, by specifying the on-toggle "+r" or "+w" in place of "+tcp". Note that in order to implement the client-server model of interaction using stream I/O, both client and server will have to maintain two separate connections each, one for read and one for write. Pseudocode fragments for a Harmony stream I/O client and server pair might be written as follows:

*Server:*

```
request_ucb = _Open( "NET: request port +r", 0 );
_Listen( request_ucb );
_Selectinput( request_ucb );

for(;;)
    {
        if( _Getstr( client_request, request_size ) )
            {
                ... process request ...

                if( !reply_ucb )
                    {
                        reply_ucb = _Open( "NET: reply port +w", 0 );
                        _Listen( reply_ucb );
                        _Selectoutput( reply_ucb );
                    }

                _Putstr( reply_to_client );
            }
        else if( _Only_nulls_left() )
            {
                _Close( request_ucb );
                _Close( reply_ucb );
                break;
            }
    }
```

*Client:*

```
request_ucb = _Open( "NET: request port@server host +w", 0 );
_Connect( request_ucb );
_Selectoutput( request_ucb );
_Putstr( request_to_server );
reply_ucb = _Open( "NET: reply port@server host +r", 0 );
_Connect( reply_ucb );
_Selectinput( reply_ucb );
_Getstr( server_reply, reply_size );
_Close( request_ucb );
_Close( reply_ucb );
```

Observe that the *call queuing* facility described above would be very useful in the timely connection of the client to the server's reply port. If the client's connect request was received by the server's TCP before the server had issued a _Listen against the reply port, the request would be waiting and ready for acceptance as soon as the _Listen was done. Note also that while it is tempting to have the server actively connect to the client's reply port, one must realize that the machinery for returning the connecting host address to the server task, converting it to ASCII, and splicing it into an open directive would have to be developed. This is simply a nuisance, not an impossibility, and could easily be implemented if experience using Harmony TCP indicated that it would be preferable to the above model.

The size of the TCP stream I/O data buffer is specified in the USER_STREAM_BUF_SIZE field of the initialization record.

### The User Datagram Protocol

Within the Harmony TCP server is also implemented the User Datagram Protocol (UDP), which is accessed by specifying the "+udp" toggle in the open directive. User interaction with UDP is message-oriented and through the _IP_send and _IP_recv primitives as with TCP. No stream I/O calls may be issued against a UDP connection. UDP provides a simple and unreliable datagram service with no retransmissions and no flow control.

Client tasks interact with UDP-based services through well-advertised port numbers in much the same way as with TCP services. However, the _Listen and _Connect primitives, through which the sequenced octet streams between TCP ports are synchronized and established, are not valid in UDP. The _IP_send and _IP_recv primitives may be issued as soon as the connection with the UDP server is open. In order that _IP_send be successful, the destination socket must have been fully specified. On the other hand, a user may call _IP_recv against an unspecified destination much in the same way a TCP user issues a _Listen. Incoming datagrams are queued intact internally so that an _IP_recv is guaranteed to return only the data borne by the first datagram on the receive queue. If the user receive buffer is too small to accept the entire datagram, the overflow data will be lost.

With each UDP datagram is associated a checksum that is calculated in the same manner as that of a TCP datagram. Disabling of this calculation can be accomplished by setting the switch _UDP_CHECKSUM_DISABLED in the initialization record.

### Complete Socket Pair Specification

The two standard paradigms described above for specifying the source and destination sockets in a TCP or UDP open directive, namely providing either only the source port or both the destination port and host, are not sufficient to cover many connection requirements. Indeed, the TCP/UDP specification is much more complete in this regard, the standard forms above being only a more commonly encountered subset. The Harmony implementation for the full specification of a source/destination socket pair in an open directive uses the keywords "s=" and "d=" as follows:

```
"net name: s=source port@source host d=destination port@destination host +pro"
```

where "net name" is the name by which the TCP/UDP server is known to the Harmony Directory task, "s=" is the keyword introducing the source socket, "d=" is the keyword introducing the destination socket, and "+pro" is the protocol, one of "+tcp", "+udp", "+r", or "+w". Recall that currently, however, Harmony TCP admits only one source host address. A situation in which the fully-specified notation would be required is that of the Bootstrap Protocol (BOOTP): the BOOTP protocol uses UDP and two reserved port numbers, one for the BOOTP client (68) and the other for the BOOTP server (67). The client sends requests using 67 as the destination port (this is usually an IP broadcast) and the server sends replies using 68 as the destination port and 67 as the source port. Thus, the Harmony open directives for the BOOTP tasks to UDP server "NET:" would be:

*Server:* `ucb = _Open( "NET: s=67 d=68 +udp", 0 );`

*Client:* `ucb = _Open( "NET: s=68 d=67@255.255.255.255 +udp", 0 );`

### UDP Broadcasting

Under certain circumstances, such as for the BOOTP client above, it may be meaningful to specify an IP broadcast address as a destination host address. Valid broadcast addresses are the local broadcast address, 255.255.255.255, the network broadcast address (the net number followed by all ones), and, if applicable, the subnet broadcast address (the subnet number followed by all ones). Each such IP broadcast address is resolved to the broadcast address of the underlying technology prior to transmission on the physical medium. Note that broadcasting should be used only with the UDP protocol.

### Historical Trace Debugging

A historical TCP/UDP trace mechanism that does not perturb the timing of the arrival and departure of datagrams has been provided for protocol debugging purposes. Each TCP or UDP datagram that is transmitted or received, all data that are delivered from a connection to a user task, and all segments enqueued and dequeued from the send queue may be logged in a circular trace buffer of a maximum of TCP_TRACE_BUF_SIZE entries. If tracing is not required, the trace buffer size may be set to zero. Which of the trace functions are desired are selected by setting switches in the NET_MODE_FLAGS field of the initialization record.

```
#define  _DEL_TRACE_ENABLED     0x08000000
#define  _SND_TRACE_ENABLED     0x10000000
#define  _RCV_TRACE_ENABLED     0x20000000
#define  _ENQ_TRACE_ENABLED     0x40000000
#define  _DEQ_TRACE_ENABLED     0x80000000
```

Access to the historical buffer is obtained through the Harmony debugger "z" command. By entering this command, followed by the task id of the TCP server, the trace entries are output to the debugging terminal one at a time, each prompted by a <return> from the keyboard. If a character other than <return> is entered, control is returned to the command level of the debugger. There are three trace formats, one for segments sent and received, one for segments enqueued and dequeued, and one for data delivered from a connection:

```
1:      ENQ SID=10 TIM=28 PRO=6
        SRC=128.144.1.170, 1024 DST=128.144.1.10, 1234
        SEQ=0x5A0001 CTL=0x5010 LEN=1024

2:      DEQ SID=10 TIM=28 PRO=6
        SRC=128.144.1.170, 1024 DST=128.144.1.10, 1234
        SEQ=0x5A0001 CTL=0x5010 LEN=1024

3:      SND SID=10 TIM=28 PRO=6
        SRC=128.144.1.170, 1024 DST=128.144.1.10, 1234 LEN=1024
        SEQ=0x5A0001 ACK=0x345678 CTL=0x5010 WND=4096 URG=0 MSS=0

4:      RCV SID=10 TIM=28 PRO=6
        SRC=128.144.1.170, 1024 DST=128.144.1.10, 1234 LEN=1024
        SEQ=0x5A0001 ACK=0x345678 CTL=0x5010 WND=4096 URG=0 MSS=0

5:      DEL LEN=1024 TIM=28 PRO=6
        SRC=128.144.1.170, 1024 DST=128.144.1.10, 1234
```

*Notes on the interpretation of a trace buffer entry:*

The SID field of the SND/RCV/ENQ/DEQ entries is the segment identification number carried in the IP header of a datagram.

The TIM field is the internal TCP server time in 128-ms ticks.

The PRO field is the transport protocol id carried in the IP header: TCP=6, UDP=17

The SRC and DST fields specify the source host address, source port number and destination host address and destination port number fields of the socket pair respectively. The LEN field is the amount of TCP or UDP user data carried by the datagram.

The SEQ, ACK, CTL, WND, and URG fields are formatted only for a TCP datagram and represent the segment sequence number, the acknowledgement sequence number, control bit fields, receive window size, and urgent data size of the given segment. For TCP synchronization (SYN) segments for which the TCP maximum segment size option is set, the MSS field indicates this value. Otherwise, MSS is zero.

ENQ and DEQ entries are made for TCP only.

## The Harmony TCP/IP Example Test Program

Two versions, one message-oriented and one stream-oriented, of a test program have been included in the Harmony example subtree in order to provide a guide to understanding how a TCP/IP server is created and its application interface primitives used. The programs also have an interactive prologue during which host addresses and values for certain key parameters are solicited from the user. In this way they may be constrained to execute in either software or hardware loopback modes or, alternatively, to communicate with a user-specified remote host. Also, the effects of changing parameters such as the maximum datagram and window sizes on the performance of the test programs may be observed.

In the test, an *active* task connects to a *passive* task and then sends a user-specified number of 1K byte buffers on the connection as fast as possible. The data consists of 32-bit unsigned integers starting at 0. When the passive task receives a buffer of data it verifies that the next expected sequence of bytes has, in fact, been delivered.

1. Enter max datagram size:

   The maximum datagram size must be at least 576 and should not exceed the maximum transmission unit of the physical net unless source fragmentation is really required. For Ethernet, the MTU is 1500.

2. Enter max window size:

   The example program constrains the max window size to be at least 1024, for otherwise a single send buffer could not be queued. A suggested value for the window size is 4096 for good performance.

3. Enter mode option word:

   The mode option word is a 32-bit hexadecimal number each of whose bits represents one switch in the NET_MODE_FLAGS field of NET_INIT_REC. Each of these binary toggles is discussed above. For a complete listing of their definitions, see tcpipuser.h. For example, if complete historical tracing is required in module loopback mode, "0xF8001000" would be entered in response to the above prompt.

4. Enter number of trace buffer entries:

   If historical tracing is not required, specify 0. Otherwise, estimate the size of the wrap-around buffer that is needed. A minimum size of 50 is suggested. The trace buffer is allocated as a contiguous block from system pool so care should be taken when specifying a large number. Each trace entry is 44 bytes in size.

5. Enter source host internet address:

   If <return> is entered, the local host address defaults to that in the IP_HOST_ADDR field of the example program initialization record: 128.144.1.170. While this host address is appropriate for a node on the Class B Alberta Research Council Ethernet, it may not be on other non-isolated networks. If an address other than the default is desired, it should be entered in the standard dotted-decimal notation.

6. Breakpoint after TCP server created? (y/n : default n):

   If "y" is entered a compiled _Breakpoint call is executed following the creation of the TCP/IP tasks. This provides a convenient place at which to set further dynamic breakpoints before the test program starts.

7. Null network interface? (y/n : default n):

   If there is no network interface controller TCP/IP application tasks may be tested in software loopback mode through the null net interface.

8. Create passive task to listen at tcp port 1234? (y/n : default n):

   If the test is being executed in any loopback mode both the active and passive tasks will be instantiated within the Harmony node and interact with the same TCP server. If, however, it is intended that the active task connect to a passive task on a remote node, answer "n" to this question.

9. Create active task to connect to tcp port 1234? (y/n : default n):

   Similarly, if the passive task is to wait for a connect request from an active task on another node, the local active task should not be created so answer "n" to this question.

10. Enter target host internet address:

    If the local active task is created and the target node is remote, then enter the destination host address in dotted-decimal notation. If <return> is entered the default local host address, 128.144.1.170, is used. Once the destination address has been determined it is spliced into the active task's open directive text string.

11. Enter target host ethernet address:

    If the physical medium is the Ethernet, the station address of the target host may be entered in the form "AA:BB:CC:DD:EE:FF". This forces an address resolution entry for the target host into the ARP cache using the _IP_bind_addr primitive. This is useful if the destination node does not implement the ARP protocol.

12. Enter send count (in units of 1K bytes):

    Enter the number of 1K buffers to be sent. The system time is reset to 0 before the first buffer is sent and then the elapsed time is acquired and displayed after the last buffer has been sent.

13. Enable TCP push function? (y/n : default n):

    The TCP *push* flag is automatically associated with all sends in stream I/O mode but is optional for message-oriented I/O.


# Application Note n013

**Title:** **Harmony Development on a Mac for the Atari (520 or 1040) ST**

**Revisions:** 1988-11-25 Darlene A. Stewart
1989-01-26 Darlene A. Stewart (minor edits)

This application note describes some details about Harmony development for the Atari (520 or 1040) ST target computer using an Apple Macintosh with the Consulair Mac C compiler, linker, and associated tools, and the inTalk terminal emulator. It explains how to compile and link the Harmony system libraries, how to compile, link, and generate .msr files for the srtest example application program, and how to download srtest to an Atari ST system. This document assumes familiarity with use of the Macintosh system, in particular, use of the Finder and MultiFinder, launching applications, and SFGetFile boxes. We also do not attempt to explain the details of using the various tools — refer to their user manuals for detailed information.

## A. Installing the Necessary Tools

1. The Harmony tree must be installed under Master, so that the HFS pathnames to Harmony files are of the form *Master:harmony:....* The Harmony tree under Master contains only code (both source and inclusion files).

   We keep any (source or inclusion) code files on which we are currently working under Working, but only those that differ from the ones under Master (i.e., Working contains a sparse Harmony code tree); only when we have completed (and tested) a project (whether a small one, such as a bug fix, or a big one, such as writing a new server) do we integrate (copy) the code from Working back into Master and then distribute a complete new copy of Master to all our Macs.

   Deletions are represented under Working using *-null-file-* and *-null-dir-* files, which are text files with no contents having special icons; the name of a -null-file- or -null-dir- file matches the name of the file or directory being deleted. Thus, Working can represent all changes, additions, and deletions that must be applied to Master at an integration point.

   All output files (.hlib, .Rel, .MAP, .out, .exe, .msr) are kept under Derived, along with copies of the inclusion files used to generate the output files. Derived may also contain other derived files, including source for one-time tests. Derived is used to hold all throw-away items that are never copied to Master, whereas Working is used to hold new and changed items that will eventually be integrated into Master.

   The inclusion files in Working point only to source files in Master as do the inclusion files in Master. The inclusion files in Derived, however, point to source files in both Master and Working, as appropriate.

   On our Macs, Master, Working, and Derived are MacServe volumes on the hard disk, so they appear as disks to the system. If it is not possible to create volumes on the hard disk (such as with

MacServe), one can simply create three folders at the root of the HFS on the hard disk called Master, Working, and Derived and use these folders as described above. If Master, Working, and Derived are folders instead of volumes, it will be necessary to edit all Harmony inclusion files changing Master to <hard disk>:Master, where <hard disk> is the name of the hard disk.

The layered scheme described above can be extended easily to include more layers too, e.g., Application-Master, Application-Working, etc.

2. The System Folder must contain the Paths.Rsrc file for Harmony. Both the source (Harmony.path) and the compiled (Paths.Rsrc) files for this path resource are distributed on the Harmony Tools disk. Paths.Rsrc is produced by compiling Harmony.path using the Consulair Path Manager.

3. To use the Consulair Mac C compiler with MultiFinder, one must arrange that Mac C is loaded in the low 512K of memory. Two small patches make this easier to accomplish. MultiFinder already contains a patch in it to force load programs with certain creators in the first megabyte. One such program is Microsoft EXCEL. Thus, patching the MultiFinder file (e.g., with Fedit+) to change all (usually two or three) instances of the string XCEL to MACC will cause MultiFinder to force load the Mac C compiler in the first megabyte. Then, the Application Memory Size (in the Get Info box) for the Consulair C application must be set to 350K. Next, we arrange that the upper half of the first megabyte is full by changing (e.g., with ResEdit) the creator of the Consulair Link application to MACC too and setting the Application Memory Size for Link to 475K. (The size required for Link to make this trick work may vary from system to system.) Then, if one simply follows the procedure of always loading the Link application before the C application under MultiFinder, the Mac C compiler will run as well with MultiFinder as without MultiFinder. It may be most convenient to move the C and Link applications to the Mac desktop.

4. The Harmony tools — *bound*, *examine*, *fixaddr*, *listing*, *listtree*, and *makemsr* — are also distributed on the Harmony Tools disk. These programs should be placed in a convenient place, say in a Harmony Tools folder on the hard disk. Also on the Harmony Tools disk are two empty files named -null-file- and -null-dir- having the special icons to represent deletions. Documentation for the various Harmony tools is located in the Master:harmony:doc:tools folder.

5. The NRC Harmony Boot Loader for the Atari ST is distributed on an Atari disk. It can be booted by inserting the disk in the Atari ST floppy drive and turning the power on or pressing the reset button on the back of the Atari keyboard unit if the power is already on. The Harmony Boot Loader file can be copied to another disk using Atari TOS operating system facilities. (See the Atari owner's manual for details.)

## B. Building Harmony

6. Using the Finder, create the folder Derived:harmony:sys:inc. While holding the option key down, drag the folder Master:harmony:sys:inc:atstcmac to the newly created Derived:harmony:sys:inc. This creates an inclusion tree for the version of Harmony for the Atari ST.

7. Compile Harmony using the following procedure. Launch the Consulair Mac C compiler; this can be done by double clicking on the C application in the Mac C folder or on the desktop (or by transferring to it from QUED using the Transfer menu, if not using MultiFinder). Once in the

compiler, walk the Derived:harmony:sys:inc:atstcmac tree, using the SFGetFile box presented by the compiler, and compile every .c and .asm file. A file can be compiled by double clicking on it or by selecting it and clicking the *Compile* button. Note that in the compiler SFGetFile box, one must select whether to see C files or assembly language files by selecting the corresponding radio button; it will not display both C files and assembly language files at the same time, so be careful not to miss compiling the assembly language files under the kernel, the debugger, and the devices. Upon completion of compiling all the files, one can either exit the compiler by selecting *Quit* from the File menu (or by transferring to another program, such as the linker, using the Transfer menu, if not using MultiFinder).

The list of files to compile is:

Derived:harmony:sys:inc:atstcmac:connect:connect.c
Derived:harmony:sys:inc:atstcmac:connect:contable.c
Derived:harmony:sys:inc:atstcmac:connect:directory.c
Derived:harmony:sys:inc:atstcmac:connect:report.c
Derived:harmony:sys:inc:atstcmac:debug:breakpnt.c
Derived:harmony:sys:inc:atstcmac:debug:bwdculib.c
Derived:harmony:sys:inc:atstcmac:debug:bwdebug.c
Derived:harmony:sys:inc:atstcmac:debug:bwio.c
Derived:harmony:sys:inc:atstcmac:debug:dbgagent.c
Derived:harmony:sys:inc:atstcmac:debug:dbgcntrl.c
Derived:harmony:sys:inc:atstcmac:debug:dbgreset.asm
Derived:harmony:sys:inc:atstcmac:debug:dbgshadow.c
Derived:harmony:sys:inc:atstcmac:debug:dcusrlib.c
Derived:harmony:sys:inc:atstcmac:debug:debug.c
Derived:harmony:sys:inc:atstcmac:devices:ataridisk:ataridisk.c
Derived:harmony:sys:inc:atstcmac:devices:ataridisk:diskint.asm
Derived:harmony:sys:inc:atstcmac:devices:mc68901:bw.c
Derived:harmony:sys:inc:atstcmac:devices:mc68901:imc68901.c
Derived:harmony:sys:inc:atstcmac:devices:mc68901:m68901int.asm
Derived:harmony:sys:inc:atstcmac:devices:mc68901:mfptimer.c
Derived:harmony:sys:inc:atstcmac:devices:mc68901:spi.c
Derived:harmony:sys:inc:atstcmac:devices:mc68901:spo.c
Derived:harmony:sys:inc:atstcmac:devices:null:nullint.asm
Derived:harmony:sys:inc:atstcmac:devices:ramdisk:ramdisk.c
Derived:harmony:sys:inc:atstcmac:gossip:gossip.c
Derived:harmony:sys:inc:atstcmac:gossip:gosusrlib.c
Derived:harmony:sys:inc:atstcmac:kernel:kernel.c
Derived:harmony:sys:inc:atstcmac:kernel:kernelasm.asm
Derived:harmony:sys:inc:atstcmac:kernel:presetup.asm
Derived:harmony:sys:inc:atstcmac:kernel:ramextern.c
Derived:harmony:sys:inc:atstcmac:kernel:romextern.c
Derived:harmony:sys:inc:atstcmac:kernel:storepool.c
Derived:harmony:sys:inc:atstcmac:kernel:taskabs.c
Derived:harmony:sys:inc:atstcmac:lib:lib.c
Derived:harmony:sys:inc:atstcmac:lib:parse.c
Derived:harmony:sys:inc:atstcmac:servers:clock:clkusrlib.c
Derived:harmony:sys:inc:atstcmac:servers:clock:clockserv.c

Derived:harmony:sys:inc:atstcmac:servers:fdev:fdbm.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdcmdnot.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdformat.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdfrqst.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdrqst.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdsd.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdseeknot.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fdsupport.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:fduserlib.c
Derived:harmony:sys:inc:atstcmac:servers:fdev:filedev.c
Derived:harmony:sys:inc:atstcmac:servers:fsys:filesys.c
Derived:harmony:sys:inc:atstcmac:servers:fsys:fsextern.c
Derived:harmony:sys:inc:atstcmac:servers:fsys:fsuserlib.c
Derived:harmony:sys:inc:atstcmac:servers:null:nullextern.c
Derived:harmony:sys:inc:atstcmac:servers:null:nullserver.c
Derived:harmony:sys:inc:atstcmac:servers:tty:tty.c
Derived:harmony:sys:inc:atstcmac:servers:tty:ttyextern.c
Derived:harmony:sys:inc:atstcmac:servers:tty:ttyserver.c
Derived:harmony:sys:inc:atstcmac:servers:uw:uw.c
Derived:harmony:sys:inc:atstcmac:servers:uw:uwextern.c
Derived:harmony:sys:inc:atstcmac:servers:uw:uwserver.c
Derived:harmony:sys:inc:atstcmac:streamio:streamio.c

If one does not want to use the Harmony file system, it is not necessary to compile the fdev and fsys servers.

*NOTE:* Walking the tree compiling many files (without quitting the compiler) eventually causes the compiler to go into a snit, where it cannot find files correctly. If this problem occurs, simply quit from the compiler, re-launch it, and continue. Occasionally, the snit manifests itself in other ways, for example, by causing the compiler to quit with a system error; in these cases, the compiler often leaves locked .Cer and .ASM files in the folder of the inclusion file being compiled when the snit occurred. These locked files must be unlocked in order for the compiler to work correctly; this can be done by attempting to open the culprit .Cer and .ASM files using QUED (rebooting will also unlock any locked files).

8. Build the Harmony libraries using the following procedure. Launch the Consulair Mac C linker; this can be done by double clicking on the Link application in the Mac C folder (or by transferring to it from the Mac C compiler or QUED using the Transfer menu, if not using MultiFinder). Use the SFGetFile box presented by the linker to invoke the linker on the .link files in the Derived:harmony:sys:inc:atstcmac folder. A file can be linked by double clicking on it or by selecting it and clicking the *Link* button. Upon completion of linking all the files, one can exit the linker by selecting *Quit* from the File menu (or by transferring to another program, such as QUED, using the Transfer menu, if not using MultiFinder).

   The list of files to link is:

Derived:harmony:sys:inc:atstcmac:bwdebug.link
Derived:harmony:sys:inc:atstcmac:debug.link
Derived:harmony:sys:inc:atstcmac:diskctrl.link

Derived:harmony:sys:inc:atstcmac:externs.link
Derived:harmony:sys:inc:atstcmac:ramdisk.link
Derived:harmony:sys:inc:atstcmac:servers1.link
Derived:harmony:sys:inc:atstcmac:servers2.link
Derived:harmony:sys:inc:atstcmac:servers3.link
Derived:harmony:sys:inc:atstcmac:system.link

If one does not want to use the Harmony file system, it is not necessary to link the servers2.link and servers3.link files.

## C. Building the Srtest Example

9. Using the Finder, create the folder Derived:harmony:example:srtest:inc. While holding the option key down, drag the folder Master:harmony:example:srtest:inc:atstcmac to the newly created Derived:harmony:example:srtest:inc. This creates an inclusion tree for the srtest applications.

10. Compile srtest using the following procedure. Launch the Consulair Mac C compiler. Use the SFGetFile box to compile the .c files in the Derived:harmony:example:srtest:inc:atstcmac tree. (See item 7 for details.) There are two versions of srtest: a single processor (normal I/O) version in the :single folder, and a single processor busywait I/O version in the :busywait folder. On the Mac, global data (externs) must be compiled separately from functions (because externs must be placed in segment 1 in the linked Mac executable); therefore, for example, srtest0 consists of two inclusion files: code0.c and externs0.c.

    The list of files to compile is:

    Derived:harmony:example:srtest:inc:atstcmac:busywait:code0.c
    Derived:harmony:example:srtest:inc:atstcmac:busywait:externs0.c
    Derived:harmony:example:srtest:inc:atstcmac:single:code0.c
    Derived:harmony:example:srtest:inc:atstcmac:single:externs0.c

11. Link the various versions of srtest using the following procedure. Launch the Consulair Mac C linker. Use the SFGetFile box to link the .link files in the Derived:harmony:example:srtest:inc:atstcmac tree. (See item 8 for details.)

    The list of files to link is:

    Derived:harmony:example:srtest:inc:atstcmac:busywait:srtest0.link
    Derived:harmony:example:srtest:inc:atstcmac:single:srtest0.link

12. Run fixaddr on srtest using the following procedure. Fixaddr converts the Mac executable output (.out file) produced by the linker to a Unix style executable (.exe file) that Harmony tools can deal with. Launch fixaddr by double clicking the fixaddr application in the Harmony Tools folder. Fixaddr displays an SFGetFile box for the file to be converted; select a srtest0.out file in the Derived:harmony:example:srtest:inc:atstcmac tree. Fixaddr then prompts for the text offset (the start address for the Harmony program); this should be 0x500 for the Atari ST, which is a single processor system. Enter "0x500" followed by a <cr>. Fixaddr generates numerous information messages as it processes the file; any warning messages about unloaded segments can be ignored.

Finally, after the file is converted, fixaddr prompts for whether another file is to be converted; enter "y" to convert the next file (enter "n" or <cr> to quit).

The files to be converted using fixaddr with their corresponding text offsets are:

| | |
|---|---|
| Derived:harmony:example:srtest:inc:atstcmac:busywait:srtest0.out | 0x500 |
| Derived:harmony:example:srtest:inc:atstcmac:single:srtest0.out | 0x500 |

A typical display from fixaddr after converting a file might look like:

```
------ FIXADDR ------
Select input file:
Enter text offset: 0x500
Link file srtest0.out opened.
Map file srtest0.MAP opened.
Exe file srtest0.exe opened.
Initialization complete.
Resource fork successfully read.
Reading map file...
Map file read.
Building symbol table...          Symbol table built.
WARNING: Unloaded jump table entry (seg 5 addr fe20)
WARNING: Unloaded jump table entry (seg 5 addr fe62)
Copying code segments...          1 2 3 4 5 6      Code copied.
Building data segment...          Data segment built and initialized.

Done. - Another image to fix? ('y' or 'n' - default 'n')
```

Before one runs an application program, it is highly recommended that the stack sizes be checked to ensure that they are large enough. This can be accomplished by running the Harmony bound tool (found on the Harmony Tools disk) on the .exe files for the application. Documentation for the bound tool is found in the Master:harmony:doc:tools folder.

13. Run makemsr on srtest using the following procedure. Makemsr generates a .msr file from the .exe file that can be downloaded into the Atari ST system. Launch makemsr by double clicking the makemsr application in the Harmony Tools folder. Makemsr prompts for the .exe file to work on using an SFGetFile box; select a srtest0.exe file in the Derived:harmony:example:srtest:inc:atstcmac tree. After the file is generated, makemsr prompts for whether another .msr file is to be generated; enter "y" to generate the next file (enter "n" or <cr> to quit).

The list of files to be run makemsr on is:

Derived:harmony:example:srtest:inc:atstcmac:busywait:srtest0.exe
Derived:harmony:example:srtest:inc:atstcmac:single:srtest0.exe

## D. Downloading Srtest to an Atari ST System

14. Make sure the RS-232 modem port for the Atari ST is connected to the Mac's modem port using an appropriate cable.

**15.** Launch the inTalk (formerly inTouch) terminal emulator (or an inTalk document) by double clicking on it. Make sure that the *Communications* settings are 19200 baud, 8 data bits, 1 stop bit, no parity, xon/xoff flow control, modem connector, no carrier detect; that the *Terminal Emulation* setting is VT-52; that the *Terminal Preferences* are set to 80 columns, line wrap OFF, local echo off, incoming cr only, outgoing cr only, incoming lf only, backspace key generates DELETE; and that the *Text Transfers* setting is for *Standard Flow Control.* These settings can be saved in an inTalk document (select *Save* from the File menu); double clicking this document will then launch inTalk and set all the options appropriately.

A very useful mode of operation for testing Harmony programs is to use MultiFinder to switch among inTalk communicating with the DVME-134 system, the *examine* tool for resolving addresses to symbols (and vice versa) and disassembling code, and QUED to look at the source code. Unfortunately, a 1-Mbyte Mac may not have enough memory to run all these applications at once. However, with MultiFinder, one can quit one application and launch another quickly.

**16.** Boot the Harmony Boot Loader on the Atari ST by inserting its disk in the Atari ST floppy drive and turning the power on or, if the power is already on, pressing the reset button at the left corner of the back of the Atari processor/keyboard unit. The following menu should appear in the inTalk window:

```
ATARI LOADER FOR THE HARMONY OPERATING SYSTEM

                AVAILABLE COMMANDS

A)bort
D)elete file from ATARI disk
K)eep mode, switch on
L)oad image from disk
Q)uick mode, switch on
R)ename a file on the Atari disk
Available memory : 419586 bytes.

READY TO ACCEPT DOWNLOAD, OR COMMAND
>
```

The ">" is a command prompt. A command is entered by entering its first letter. Because inTalk presents a progress indicator during file transfers, it is desirable to turn the Quick mode on in the Atari loader by entering "q" in order to turn off the loader's own progress indicator. One can arrange that the next file being downloaded will also be stored on an Atari disk by turning Keep mode on; this is done by entering "k" and then responding to the questions asked by the Atari loader (the loader provides straight forward prompts). Once a program (.msr file) is stored on an Atari ST disk, it may be loaded from disk using the Load command by entering "l" and answering the prompt for a file name; loading from disk is much faster than downloading from the Mac. (Note that it takes slightly longer to download a program when storing it on an Atari disk too than when not storing it on disk.) Other commands are provided in the loader for managing the files on the Atari disk (e.g., deleting or renaming files). Documentation about the Atari loader and its commands can be found in Master:harmony:doc:bootrom:atarist.doc.

17. The only method of downloading supported by the NRC Boot Loader for the Atari ST is downloading of Motorola S-Records (.msr file) using a straight text transfer.

    To download the srtest0.msr file (single version) use the following procedure. Select *Send Text File...* from the *Transfer* menu and using the SFPutFile box, choose (double click) the srtest0.msr file from the Derived:harmony:example:srtest:inc:atstcmac:single folder. This sends the file; a progress indicator is displayed along the bottom of the inTalk window during the file transfer as well as buttons to stop/pause the transfer. During the download the Atari loader displays the following message below the prompt for the command menu:

    ```
    Accepting file from serial link...
    ```

    Once the download is (successfully) completed, it replaces the message will the following:

    ```
    Download operation succeeded.
    ```

    Also, two new commands appear in the command menu:

    ```
    C)lear memory
    G)o, run image
    ```

    and the display of the amount of available memory is updated. If the download failed, say because there was not enough memory or not enough space to store the program on disk, an error message is displayed instead of the "download succeeded" message and the Go command is not added to the command menu. Entering a command clears any informational message from the menu display.

18. Enter "g" to start the program running. It should produce output like:

    ```
    parent creates child
    child replied 1
    child replied 2
        etc.
    child replied 100
    How many more message exchanges do you want?
    ```

    Entering a number in response to this question causes srtest to exchange the requested number of messages between its main and child tasks outputting the replied result for each exchange.