Using the Harmony Operating System: Release 3

Gentleman, W.M.; MacKay, Stephen; Stewart, Darlene; Wein, M.

81 (ERA-377), 65 PP. FEB. 1989.

National Research Council Canada
Division of Electrical Engineering

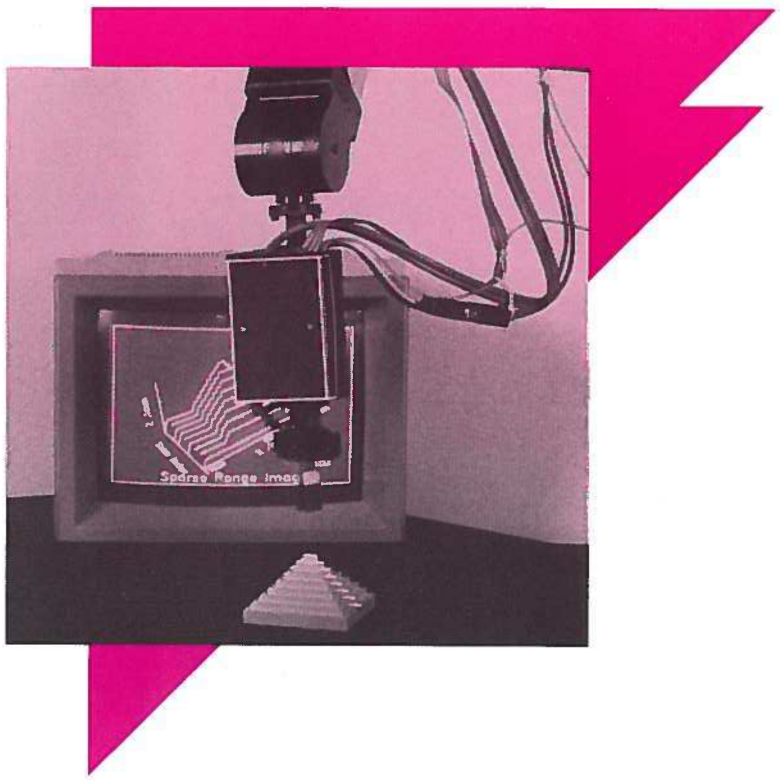
Conseil national de recherches Canada
Division de génie électrique

ERA-377

Using the Harmony Operating System: Release 3.0

W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein

February 1989



Canada

NRCC No. 30081

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

This report supersedes NRC/DEE report ERB-966, NRCC No. 27469, with the same title

Le present rapport remplace le rapport ERB-966, NRC/DEE, NRCC No. 27469, qui porte le même titre.

USING THE HARMONY OPERATING SYSTEM
Release 3.0

W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein

ERA-377

NRCC No. 30081

February 1989

TABLE OF CONTENTS

		Page
ABSTRACT/RÉSUMÉ		v
WHAT IS HARMONY? WHAT IS IT NOT?		1
WHAT IS REALTIME?		2
TASKS		3
COMMUNICATIONS AND SYNCHRONIZATION		6
STORAGE CLASSES		8
STREAM INPUT/OUTPUT		10
IMPLEMENTATION OF OTHER INPUT/OUTPUT MODELS		12
SERVERS AND CONNECTIONS		16
USER'S VIEW OF A PROGRAM		21
DEVELOPMENT ENVIRONMENT		22
Support Tools		22
debug		23
listing		23
examine		23
bound		23
fixaddr		24
fixexe		24
fixmsr		24
makemsr		24
listtree		24
mmake		24
PERFORMANCE		25
ACKNOWLEDGEMENT		26
REFERENCES		26
APPENDIX A.	A SAMPLE MULTITASK PROGRAM	29
APPENDIX B.	A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM	33
APPENDIX C.	DECLARATIONS AND SYSTEM FUNCTION CALLS	35
APPENDIX D.	USING SUPPLIED SERVERS	45
APPENDIX E.	SOFTWARE ENGINEERING CONSIDERATIONS: SOURCE MANAGEMENT AND THE ORGANIZATION OF THE HARMONY SOURCE TREE	61
APPENDIX F.	DEVELOPMENT SYSTEM USED AT NRC	65

ABSTRACT

Harmony is a multitasking, multiprocessing operating system for realtime control (such as is required in sensor-based robotics). It can also be used to implement "smart peripherals" to be used with itself or with some other operating system, and it provides a simple and inexpensive vehicle for experimenting with parallel computation. It is a portable system in that with moderate effort it can be realized on any computer of appropriate architecture. Our realizations run on several multiprocessor systems (VME and Multibus) based on the Motorola 680x0 family of processors. There is also a version for the Atari 520 or 1040 ST (MC68000) evaluation machine. Realizations exist also for other systems and for the Digital Equipment Corporation VAX. Written in C, Harmony is portable also across several development systems, of which three are supported directly. The one used in our Laboratory is based on the Macintosh personal computer and is described here. This report presents an overview of the operating system, the rationale for the design and the abstractions used in creating applications that can be targeted to either a single processor or a multiprocessor. The report also discusses the principal mechanisms used for task management and for intertask communication. Appendices in this report include examples of both uniprocessor and multiprocessor programs and listing of user callable functions.

RÉSUMÉ

Harmony est un système d'exploitation multitâche et multitraitement pour la commande en temps réel (notamment pour la comande de robots dotés de capteurs). On peut également s'en servir pour la gestion, sous Harmony ou un autre système d'exploitation, de périphériques intelligents. Il constitue un outil simple et peu coûteux pour faire des expériences en traitement parallèle. C'est un système transférable en ce sens qu'on peut l'implanter assez facilement sur n'importe quelle machine possédent l'architecture appropriée. Les versions que nous avons realisées tournent sur plusieurs systèmes multiprocesseur (VME et Multibus) architecturés autour de la famille de processeurs Motorola 680x0. Il existe également une version pour les machines Atari 520 ou 1040 ST (MC68000). Il existe également des versions pour d'autres systèmes et pour le VAX de Digital Equipment Corporation. Rédigé en C, Harmony peut être utilisé avec plusieurs systèmes de développement, dont trois sont supportés directement. Celui dont nous nous servons à notre laboratoire est basé sur l'ordinateur personnel Macintosh et est decrit ici. Le présent rapport présente une vue d'ensemble du système d'exploitation, des motifs qui ont présidé à sa conception ainsi que des principes sur lesquels repose la création de programmes d'application orientés monoprocesseur ou multiprocesseur. Le rapport étudie également les principaux mécanismes utilisés pour la gestion des tâches et pour les communications avec ces dernières. On trouvera en annexe des examples de programmes monoprocesseur et multiprocesseur ainsi qu'une liste des fonctions qui peuvent être appelées par l'utilisateur.

USING THE HARMONY OPERATING SYSTEM

Release 3.0

W.M. Gentleman, S.A. MacKay, D.A. Stewart, and M. Wein

WHAT IS HARMONY? WHAT IS IT NOT?

Harmony[1] is a *multitasking, multiprocessing* operating system for *realtime* control. (Each of these terms will be examined later.) It is a *portable* system, in that with moderate effort it can be realized on any computer of appropriate architecture. Our realizations run on several system types based on the Motorola 680x0 family of processors. There are two versions for a multiprocessor based on the VMEbus, one using the DY-4 DVME-134 (MC68020) single board computer (SBC) and the other using the Io Inc. V68/32 (MC68020) SBC. There is also a version for the Atari 520 or 1040 ST (MC68000) evaluation machine, as well as for the earlier Chorus multiprocessor, an MC68000-based Multibus multiprocessor using the Omnibyte OB68K1A SBC. Realizations exist also for other 680x0 family systems and for the Digital Equipment Corporation VAX. The operating system is written in C (plus a small amount of assembly language) and user programs at present must also be written in C or Fortran, although the system is designed to support eventually other languages such as Pascal. Care has been taken to avoid dialect or library dependence on any specific C compiler —currently, the compilers being used are Consulair Mac C, Apple MPW C 2.0, and A/UX C (a port by Unisoft of the PCC compiler) hosted on the Macintosh, and the Whitesmiths' C compiler hosted on the VAX/VMS.

Although C is used as the programming language, programs written to run in Harmony look, and are thought of, very differently from conventional C programs that are written to run in Unix[2] or some other conventional operating system. This difference goes beyond the question of what functions are available in the standard libraries and will be discussed at length later.

Harmony is an *open* system. This is a description that can be used at many levels. In the case of Harmony, this description is intended to imply that it is easy to use the system on many different configurations of hardware, and in particular to support peripherals not thought of when Harmony itself was originally designed. It is straightforward to integrate a new device into both the physical level of connection and the higher levels of software abstraction.

One major aspect is that Harmony is not designed to support a program development environment, and in particular it is not a multiuser timesharing system. Harmony devotes the entire resources of the machine to running a single multitask application program, and is bound in with that program at link time (unless a system in ROM is used). Program development is done on some host computer, running its own operating system, and test executions are performed by downloading executable images into target processors. (Production programs instead might well be in ROM and be started by power up or reset.) In particular, Harmony programs cannot assume that they have been started by a command interpreter like the Unix shell and that they have command line arguments and redirected I/O. Indeed, they may not have terminal I/O at all.

[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.

[2] Unix is a trademark of AT&T.

WHAT IS REALTIME?

The first of the terms used in the initial sentence to be examined is the term *realtime*. Realtime programming has two independent aspects. First, programs must interact with the real world, and second, time itself may be a critical resource.

What makes interacting with the real world different from other kinds of programming is primarily that relevant events in the real world can happen independently of, and asynchronously with, the state of computation of the program. Conventional programming assumes a master/slave relationship between the computer and the rest of the world — things only happen when requested by the computer and are then guaranteed to happen, typically synchronously with the computation. This relationship does not hold for programs that interact with the real world. Events in the real world are often spontaneous, as seen by a program, and anticipated events often do not happen. Often the program cannot be aware of all of the relationships between events occurring in the real world. Indeed, the situations that can arise in the real world are often so varied that it is unrealistic to attempt to anticipate them all, and thus programs must cope with unexpected events.

Unless we are willing to limit programs to tracking a single sequence of events and to busywait for each event in the sequence to happen, we need a mechanism for expressing asynchronism in the program, so that the asynchronism in the program can model the asynchronism in the real world and be synchronized with it. This mechanism must be understandable, well defined, and efficient, so programs can be designed using it and can be reasoned about so that their behaviour will be known. The mechanism that Harmony provides is *tasks* (also called processes in some other systems). It will be described below.

A secondary aspect of interacting with the real world is that exotic peripherals are often required to sense or affect things of interest. Not only do these exotic peripherals require special command interfaces, because they do not fit the traditional I/O model, but they often require complex algorithms in order to be used, because they are flakey: sensors do not reliably return correct values, actuators do not reliably accomplish the desired effect. Error recovery is thus also intrinsic to interacting with the real world. Both error recovery and coping with unanticipated events are considerably simplified if resources can be allocated and deallocated dynamically. Such resources include tasks themselves, where instantiations must be created and destroyed dynamically, as well as blocks of memory of various sizes, which on demand are allocated to and freed by tasks, and connections between tasks and peripherals, which on demand are opened and closed by tasks.

Working with time itself as a critical resource is different from other programming in several ways. In conventional programming, time is irrelevant except perhaps in terms of overall efficiency considerations, or for recording and metering. Time can become a critical resource if, for instance, an external event requires a control signal in response within a guaranteed time, or if an external event produces volatile data which must be accepted soon enough to avoid the data being lost, or if periodic activity must be performed at regular intervals with little jitter, or if performance requirements indicate need for explicit parallelism between computation and external activities or between different computations, or if parallelism is intrinsic to the computation (as when stepping motors in orthogonal directions must operate simultaneously to produce a diagonal movement). Of course time is only a critical resource when the constraints are sufficiently stringent, given the available computational power, that conformance must be planned for and will not happen by accident.

Sometimes, when time is a critical resource, time is of the essence and actually defines or redefines the computation. For example, in the above case involving periodic activity, if new sensor data are not available in time for the next cycle of computation, previous data must be extrapolated. Situations dealing with motion, and predicting trajectories, are often like this. More commonly, when time is a critical resource, the requirements translate into scheduling control. There is the choice of which computation to do first, when several are possible. When some computation becomes possible, there is the question of which current computations should be run to completion before starting this new one and which computations should instead be preempted. There can also be the necessity of ensuring the proper sequencing of otherwise independent computations. In Harmony, these things are achieved by dividing the program into tasks and using the rules for communication and synchronization between tasks, and the rules, including priority, which govern the scheduling of tasks on each processor. (These rules will be detailed below.)

One common characteristic when time is a critical resource is that no unanticipated delays in performing a computation are acceptable, that time performance must be guaranteed. This implies that only algorithms with execution time that can be bounded can be used. This stipulation apparently conflicts with the requirement (from interacting with the real world) that resources be dynamically allocated. The resolution of this dilemma is to preallocate those resources that will be required at critical times, doing the dynamic allocation at times that are not critical. For example, execution stacks are preallocated out of contiguous memory, rather than having stack frames allocated on demand from a heap, so that the cost of calling a function is bounded. Demand paging obviously is unacceptable in these circumstances.

Sensor-based robotics is an example of realtime computing that clearly illustrates both the real world aspect and the time critical aspect. In many robotic applications, there are a variety of objects in the robot's workspace. The robot must be aware of their positions, either to avoid them, or to make appropriate contact with them. Unless these objects are precisely located by mechanical fixturing, their exact locations, perhaps their shapes, and even their existence must be determined by sensors. Vision is one possible sensor, sensing the reactive forces on the robot's wrist is another, tactile and proximity sensing are yet others. The sensor measurements must be related to some abstract model of the environment for this data to be useful in planning robot motions. Note that this involves significant computation, using sophisticated algorithms. Some failure modes, and some reasons for unanticipated events, are physically obvious: lighting may be inadequate to visually resolve objects or objects may occlude each other, force exerted by the robot may cause objects to move or even break, proximity sensors may receive echoes from objects other than the one of primary interest. Having robots interact with objects in motion, even with another robot, is particularly challenging. The trajectory of the objects must be predicted and must be continually updated as new sensor data become available. The time at which things are computed becomes important, it may be necessary to predict how long various computations will take, and computations may have to be scheduled so that values will get computed soon enough. Not only does it become necessary to know how long it takes for sensors to obtain data, and for the computer to analyze the data, and for the robot and other effectors to perform operations, but time becomes of the essence in a program, such as having to specify when a robot arm should be at some point, and what speed it should have in what direction, in order to intercept some object, match the object's motion, and connect with it.

TASKS

At several points in the foregoing discussion, the concept of tasks has been alluded to, and it is now appropriate to define it. Tasks are a concept not used in conventional programming, not even in ordinary C programs in Unix, but they are the heart of programming in Harmony. A program in Harmony is made up of one or more, but typically many, tasks. A task is best thought of as like a subroutine, except that an instantiation of it must be explicitly created, and once it has been created, it executes independently of, and in parallel with, the task that created it. (Actually, tasks that are completely independent of each other are not of very much use, so a mechanism must be provided for tasks to communicate and synchronize with each other. The mechanism that Harmony provides is message passing, which will be described shortly.) Each task itself executes sequentially and synchronously, like a conventional program — the asynchronism and parallelism are strictly between tasks.

Executing in parallel is, of course, only actually possible when the computer is a multiprocessor, and the tasks are executed by different processors. There are two classical reasons for using multiprocessors: to increase total computing capacity, and to have redundancy to provide reliability in case of hardware failure. In realtime systems, there is a third reason: to guarantee responsiveness, by ensuring that a processor will be available to service an event soon enough. Harmony thus, as stated at the beginning, supports multiprocessing. When a task is created, the processor on which it will execute is specified, and this need not be the same as the processor on which the task requesting the creation is executing. A task competes for scheduling only with other tasks that run on the same processor. Note that the Harmony model of tasks tied to specific processors is different from the model used in some other systems, where any processor can execute any task. The Harmony model recognizes that processors may have local resources, such as memory or peripheral IC's, that can only be accessed by tasks executing on that processor, or that are much more efficient if so accessed. It also facilitates different processors executing different instruction sets. The hardware assumptions Harmony makes are discussed elsewhere [1].

Some simple multiprocessing systems allow only a single task to execute on each processor. This has the advantage of simplicity and makes context switching overhead negligible. However, it has significant disadvantages. The number of tasks can no longer represent the natural asynchronism of the computation but is constrained by the number of processors in the configuration, and failed processors must be allowed for. Moreover, processors, though cheap, are not free, and utilization may be poor when it is known that certain tasks will execute at disjoint times. Accessibility of peripherals can force several tasks to be executed by the same processor. Certain operating system services, required by each processor, are most conveniently implemented by separate tasks executing on that processor. Consequently, Harmony supports multiple tasks on each processor.

Since multiple tasks can be created to run on any given processor, Harmony must provide multitasking to share the processor amongst these tasks. The approach taken in timesharing systems to provide the analogous capability for multiprogramming is through timeslicing and round-robin scheduling. However, this approach would be inappropriate for realtime, given the requirements for control of scheduling as mentioned above. There are several scheduling schemes possible, such as deadline scheduling or clocked scheduling. The scheme in Harmony uses priority-based preemptive scheduling. Each processor supports a separate priority scheduling system. Each task has a fixed priority level. There is a separate FIFO ready queue for each priority level, and all tasks of that priority level which are not blocked are in that queue (blocking actions will be described later). The first task on the highest priority nonempty queue is the active task, i.e., the one which is executing. It continues to execute until it blocks or is preempted by a higher priority task becoming ready. That is, Harmony tasks run *natural break* and support *preemption*. Priority, as used in realtime systems, does not refer to *importance*, but rather to *urgency*. The ideal effect of priority is that the scheduling behaviour of a task can be understood by understanding only those tasks at the same or higher priority levels — the existence of lower priority tasks, much less their behaviour, can be ignored. In particular, if there is only one task at the highest priority, strict bounds can be placed on scheduling delays and jitter. Another important effect of priority is that nonblocking actions of high priority tasks can be viewed as atomic by lower priority tasks.

A task is thus the unit of sequential and synchronous execution. A task is also the unit of resource ownership. All resources, such as memory blocks, peripherals, or I/O connections, are owned and managed by some task.

For a task to be created, there must be a corresponding task template, which specifies the essential parameters of the task. (Multiple instantiations of the task defined by a given template can exist at the same time.) A task template is the following structure:

```
struct TASK_TEMPLATE
  {
     uint_32        GLOBAL_INDEX;
     task           (*ROOT)();
     uint_32        STACKSIZE;
     uint_32        PRIORITY;
  };
```

The type uint_32 is compiler-specific — see Appendix C. The fields in this structure are as follows:

GLOBAL_INDEX	is an integer identifying this template. It must be unique over all templates on all processors for this program. The range of these indices is 1 through _N_task_templates, a ROM external configuration parameter for which 200 is usually a satisfactory value. It is wise to use only larger values for GLOBAL_INDEX, say over 50, and also to use symbolic constants instead of literal integers. The indices 1, 2, and 3 are reserved: 1 is the first user task (i.e., it calls the procedure main()), 2 is the directory task (i.e., it provides task IDs corresponding to symbolic names), and 3 is the gossip task (i.e., it provides a general reporting and logging mechanism for other tasks). The system itself creates on processor 0 one instance of each of these three tasks.
ROOT	is the function of type task that is the code which the task will execute. (This function can, of course, call others.) It is a normal parameterless function with no return value. (A typedef defines task as void.) If the function ever returns, the task is destroyed; however, the parent is not notified. More than one task template might have the same root function.
STACKSIZE	Each task has its own stack, which is preallocated when the task is created. Unfortunately, most compilers do not generate code to check for stack overflow. A program development tool, the stack bounder, helps with the choice of this value.
PRIORITY	Task priority is specified by an integer; larger is lower priority. When Harmony is ported to a given target, these priorities are arranged to be consistent with whatever hardware priorities the target implements. For instance, on the Dy-4 DVME-134 multiprocessor, values 0 through 4 correspond to the Motorola MC68020 hardware interrupt levels 5 through 1. All lower priority levels correspond to subdivision of hardware priority 0. The number of priority levels is not fixed, but having too many levels gains little and adds overhead.

The executable image for each processor is compiled and linked separately in Harmony. For each processor, that image is defined, in part, by the declaration of a vector of task templates, _Template_list, that determines what tasks can be created on this processor. The vector is terminated by a null task template, i.e., one in which all fields are zero. Tasks are created by calls to the _Create primitive, specifying a template index:

```
id = _Create( global_index );
```
[3]

The _Create primitive returns a value id, which is a task ID used to identify this particular instantiation in future, e.g., for message passing, task destruction, etc. The value of id can be treated as simply a unique

[3] The form given in the text shows the usage of functions. Precise definition of type and of arguments appears in Appendix C.

collection of bits, which the programmer need not decode. Actually, it is an encoding identifying the processor and task. As a consequence, there is a limit of 4096 tasks that can be created on any one processor and a limit of maximum 1024 processors. The limits are high enough that one is not likely to reach them.

Two functions are provided to obtain other IDs that are often wanted: the ID of the task itself and the ID of the task that created this one:

```
id = _My_id();
id = _Father_id();
```

As described above, a task is destroyed if its root function returns. It can also destroy itself by calling the _Suicide primitive:

```
_Suicide();
```

More generally, any task can be destroyed by calling the _Destroy primitive:

```
_Destroy( id );
```

Destruction implies that the task is stopped, its id made invalid, and that its resources are returned to the system. Its connections (which will be discussed later) will be closed. Destruction of a task also implies destruction of all the descendants of the task, i.e., those tasks it created, those they created, etc. This is important in order to avoid the existence of orphans, tasks that hold resources but are useless because no other task knows of their existence. The _Destroy primitive is synchronous, which means that all this will be completed before the primitive returns, so the resources are immediately available for reuse.

COMMUNICATIONS AND SYNCHRONIZATION

Although the main purpose of using tasks is to allow them to proceed with their own computations independently and in parallel, each running at its own speed, in most practical programs the tasks need to communicate and synchronize with each other occasionally. In particular, if more than one task accesses the same location in storage, and if the value at that location can change, then there is a possibility of a critical race where the result of the program depends on when which task reads and when which task writes the value. Various systems have provided different mechanisms for dealing with the synchronization problem [2 – 4], including semaphores, monitors, conditional critical regions, rendezvous, and several kinds of message passing. Harmony addresses the problem by a paradigm for structuring multitask programs, using four primitives that provide message passing. A *message*, in Harmony, is a variable length contiguous block of storage, the first four bytes of which are a 32-bit unsigned integer specifying the length. Typically the type of the message is actually a struct or union, but the primitives declare the type for their arguments as char *, requiring a cast. The primitives are implemented as functions:

```
id = _Send( rqst, rply, id );
id = _Receive( rqst, id );
id = _Try_receive( rqst, id );
id = _Reply( rply, id );
```

The semantics of these primitives are as follows: A task wanting to send a message to another task sets up that message in space pointed to by the rqst argument, sets up space pointed to by the rply argument into which to receive a message in reply, and then calls the _Send primitive specifying in the id argument the ID of the desired correspondent task. A task wanting to receive a message must set up space pointed to by the rqst argument into which the contents of the sender's rqst message can be copied, and then calls either the _Receive or the _Try_receive primitive, specifying in the id argument either the ID of a particular correspondent task (receive specific) or 0, the latter indicating that a message will be accepted from any task (in fact FIFO). When the _Receive or _Try_receive primitive returns, the value id returned by the function is the ID of the correspondent task, and the sender's message will have been copied into the receiver's message. After the receiving task has finished processing the sender's request, it sets up, in space pointed to by the rply argument, the message to be replied back to the sender, and calls the _Reply primitive with the id argument specifying the sender. Specifying the sender is necessary because it is not required that a sender be replied to before receiving another message, nor that messages be replied to in the order they were received, in the inverse order to which they were received, or any other fixed order. A reply can only be made to a task from which a message has been received and to which a reply has not yet been made, however. The message being replied to the sender is copied from the receiver's space to the space provided by the sender before the _Send primitive returns. (Detailed definitions of the Harmony fundamental structures used to define the arguments are given in Appendix C).

A particularly important aspect of the semantics of these primitives is their blocking behaviour, for this is how tasks synchronize with each other. The sender blocks from when the _Send primitive is called, not just until the corresponding _Receive or _Try_receive primitive is called by the receiver but until the receiver calls the corresponding _Reply primitive. A receiver calling the _Receive primitive blocks if the id argument is 0 and no task has already called the _Send primitive to send a message to this receiver, or if the id specifies a particular task and that task has not yet called the _Send primitive to send a message to this receiver. The receiver calling the _Receive primitive unblocks when the message for which it is waiting becomes available. By contrast, the _Try_receive primitive is nonblocking and fails immediately, returning 0, if the desired message is not available when the receiver calls the primitive. The _Reply primitive is a nonblocking operation for the receiver (because the sender is known to be already blocked) and it also unblocks the sender, allowing it to run again. Of course if the sender and receiver happen to run on the same processor, the normal scheduling rules will dictate which actually runs first. Nothing else in the foregoing discussion is affected by whether the communicating tasks happen to be on the same processor or different processors.

Because tasks can be destroyed, it is possible that the primitives can fail. This is the reason why _Send and _Reply are also implemented as functions: all four primitives return the ID of the correspondent task if they succeed, and 0 if they fail. Another point to note is that the length of the message indicated in the space copied from and copied to may disagree. Only the shorter length is actually copied, but because the length copied from the source is the first thing copied into the destination space (overwriting the length originally there), it is possible to find out if the message was truncated, and how much space would be required to get it all.

There is a certain symmetry between _Send and _Reply in that both transfer data to another task. By turning around the usual role of these primitives, a nonblocking communication is possible between one task and another which has prearmed for the communication by sending to indicate it is ready to receive a message. That is, suppose task A has some data to transfer to task B. Normally, one would think of A sending a message to B. If this is done, A will block until B replies. If, however, it is important that A not block, the communication could be reversed and B could instead send to A and block. When A is ready with the data, it can transfer the data to B in a reply, with which A will not block but will continue to execute, and B will also resume execution in parallel. When B finishes processing the data, it can send to A again, returning any results and preparing to repeat the cycle.

The paradigm [2] is that, in general, tasks do not use resources directly; for example, several tasks will not all access a location in memory that can change. Instead, each such resource is *owned* by a single task, and operations on the resource are performed by that task on behalf of other tasks that have sent messages requesting these operations. The task given the responsibility for some resource is referred to as a *server*, since it is seen by other tasks as providing a service — the tasks using this service are referred to as *clients*. Mutual exclusion follows directly from the fact that the server is a single task, with a single thread of execution, and messages are received one at a time. Sometimes, parallelism can be exploited with respect to the resource the server controls. This is done by the server having *worker* tasks that employ the reverse communication described earlier — each worker task sends to the server whenever it has completed the last item of work it was given, and consequently is available to be given (by a reply) another item of work to do.

Occasionally, it is appropriate for closely cooperating tasks to share memory because the protocol by which they cooperate ensures this is safe and because the overheads of passing the data back and forth by message passing would be excessive. (Large constant tables are a classic example of where sharing is necessary and safe.) Harmony supports this, in that addresses mean the same to all tasks on the same processor, so pointers can be passed in messages and used by recipient tasks. When Harmony is implemented in a flat address space shared memory multiprocessor, this extends to pointers used by all tasks, whether on the same processor or not. However in a *thinwire* implementation, where processors communicate via reliable links but cannot directly access each other's memory, a pointer is only useful within a single processor. Because processor images are linked separately, externals are shared among tasks on the same processor, but only among tasks on that processor. In general, sharing memory requires great care and understanding, however, and should be avoided where possible. The tasks supplied with Harmony do not use shared memory, except for accessing _Dev_data_table [5] and in the debugger. In a few cases it has been necessary to provide a readable copy of a write-only device register that unfortunately must be shared by otherwise independent tasks.

The semantics of the call to _Send used by a client task to a server are identical to a procedure call, and so are familiar and comfortable to all programmers. Indeed, this semantics has become known in the literature as the remote procedure call. Because _Send and _Receive can block, there is some possibility of deadlock, but in practice it does not happen when the client/server paradigm is used because almost the only way to get a deadlock would be to have a cycle of tasks sending to each other. With a hierarchy of services implemented using lower level services, that never happens.

STORAGE CLASSES

A program in C running under Harmony has access to five storage classes: external, static, automatic, register, and dynamically allocated.

The first two can be useful, but have complicated aspects. Because the executable image for each processor is linked separately, externals or statics referenced by tasks on one processor are different from those referenced by tasks on another, even though the name in the source code is the same. On the other hand, all tasks on the same processor will access the same instance of a named external or static — which, unless the external or static is initialized and not subsequently assigned to, can lead to critical races. It is also worth noting that ROMability is often an objective of realtime systems, and that an external or static that is initialized by compile-time initialization is not ROMable if it is subsequently assigned to.

The situation is much simpler for variables of the automatic storage class. They are allocated on the stack of the task, dynamically as the task executes. There is a different instance at each function call, never mind for each task. The stack for the task itself is dynamically allocated when the task is created and recovered when the task is destroyed. Because local memory typically has faster access, the stack for a task is allocated from memory local to the processor on which the task will run.

Some of the hardest bugs to find involve stack overflow, where the stack required at some point in the nesting list is bigger than what was allocated for the task. Unfortunately, there is rarely adequate hardware checking for this when there are multiple stacks in the same address space, and most compilers do not generate code for a runtime software check on function entry. Harmony provides two aids to alleviate this unsatisfactory situation. The first is a function that can be called to determine if the stack for this task is already overflowed:

```
boolean = _Stackoverflow();
```

The other is a program development tool, `bound`, that determines the required stack size when the call graph for a task is simple (an acyclic digraph of known functions) and interactively aids the user in choosing an appropriate stack size when there is recursion or indirect function calls.

The register storage class of course has no problems for tasks.

A contiguous block of storage can be dynamically allocated by calling the primitive:

```
ptr = _Getvec( size );
```

where `size` is the number of bytes in the desired block, and the value returned is typed as pointer to `char`, so typically must be cast to the required type. The value 0 is returned if the allocation failed. Memory is allocated from a pool local to the processor executing the primitive. The memory obtained is not initialized to any particular value.

When a block of storage is no longer required, it can be returned to the storage pool by calling the primitive:

```
_Freevec( ptr );
```

When a task is destroyed, all of its allocated blocks are automatically returned to the storage pool.

Occasionally it is desirable to allocate an overly generous block of storage, then return the excess to the storage pool once the true requirements are known. Reducing the block to `size` and returning the excess is achieved by calling the primitive:

```
_Trimvec( ptr, size );
```

The size of a dynamically allocated block is available from the primitive:

```
size = _Sizeof( ptr );
```

A first fit storage allocation strategy is used for `_Getvec` because, although it does not provide the fastest allocation, the probability of denials is reduced since the *graded* distribution of idle blocks that develops over time preserves larger blocks. The classical problem with first fit is that each search for a suitable idle block examines whatever permanently allocated blocks are at the beginning of the pool as well as the blocks that are too small. Storage allocation for blocks `size` or larger may be faster if from time to time the starting point for the space search is reset by calling the primitive:

```
_Tune_Getvec( size );
```

STREAM INPUT/OUTPUT

The model of stream I/O is a powerful and uniform abstraction that is useful for doing input/output to a wide range of devices, from serial lines to terminals to tapes to disk files. A stream is an infinite sequence of bytes in which a current position is defined, and on which the operations of getting the next byte, putting the next byte, and changing the current position may be performed. Any higher level structure must be inferred from the value of the current byte, the context of the current byte, or the current byte position. For example, numbers are formed as sequences of bytes separated by delimiters; variable length lines of text are readily supported by separating the lines by a new-line symbol and transparent communications line control is achieved by ASCII control characters together with escapes inserted into the text to be transmitted.

Because the stream I/O model states that there is a first byte and each byte has a successor, Harmony does not use the concept of an end-of-file. Instead, a mechanism is introduced to indicate the end of useful information, whereby all successors to the last useful byte are NULL. Two functions are introduced to support this concept. The first function is _Only_nulls_left(), which returns TRUE to indicate end of useful information, and the second function is _Nullify(), which, on output, clears all successor bytes. The stream I/O model is based on a paradigm of a disk file. Therefore it is natural for a stream to be opened for read, write or read/write. A discussion of differences between the Harmony model of input/output and that of Unix can be found in Application Note n004 [5]. See also the discussion of the underlying I/O model on p.15.

Harmony supports *connection-oriented* stream I/O. (All types and manifest constants referenced below are declared in the standard header, sys.h, which should be included in any compilation of a Harmony program — see Appendix C.) A stream is made accessible by being opened by a task:

```
ucb = _Open( directive, userid );
```

where directive is a string that symbolically defines the stream of interest and will be discussed more fully later, and userid is an unsigned integer value that is an application-defined identification which servers may use to enforce access control. The value returned, ucb, points to a dynamically allocated struct called a *user connection block* that defines the server and connection number associated with this stream and points to the ucb_xtra, a second dynamically allocated struct that contains the buffer and supporting pointers. The user of stream I/O need not be aware of these details. The value 0 is returned if the open failed. There is no fixed limit as to how many streams may be open. In general, the same byte stream may be opened more than once (although some servers may disallow this), each with its own current position. This is typically done when several tasks use the same stream.

A stream which no longer needs to be accessible should be closed:

```
_Close( ucb );
```

Destruction of a task automatically closes all its open streams. It should be noted that closing a stream does not imply flushing any buffers (see below) and this must be done explicitly.

For each task, at most one of the opened streams is selected for input at any time, and all input comes from this stream until the selection is changed by:

```
prev_ucb = _Selectinput( ucb );
```

where the function returns the user connection block, prev_ucb, identifying the previously active stream.

Similarly, for each task at most one of the opened streams is selected for output, and all output goes to that stream until the selection is changed by:

```
prev_ucb = _Selectoutput( ucb );
```

In this way, input bytes can be obtained from the selected input stream without the cost and inconvenience of specifying the stream each time:

```
byte = _Get();
```

and similarly output bytes can be put to the selected output stream without the cost and inconvenience of specifying the stream each time:

```
byte = _Put( byte );
```

The byte output is returned as the value of _Put because of the convenience this often provides in program control structure.

There are also stream I/O primitives in terms of records, which offer significant notational advantage when entire records are to be moved. These functions refer to the currently selected stream and are:

```
error = _Get_record ( record, size );

error = _Put_record ( record, size );
```

where the record is a pointer to char and size is the number of bytes. The bytes in the stream will be in the same order as they are in memory.

Although the abstraction of stream I/O is attractive, it would be impractically expensive if it were not actually buffered so that _Get, _Put, _Get_record and _Put_record usually just reference a local buffer instead of actually doing I/O or even sending requests to a server. This buffering is transparent on input (although finding whether a byte is available for input is impossible without taking the risk of blocking) but not on output. The output buffer will be sent to the server to be output when it is full, but situations often arise where the buffer should be forced out without waiting to be full. This is done by:

```
_Flush();
```

(It is unfortunate that the phrase *flush the buffer* has two conflicting interpretations — instead of the one used here, there is a different one sometimes used in other systems to imply that the buffer contents should be discarded.)

The buffered input makes possible a function which is very convenient when breaking input tokens at delimiters, or when reading input that may be of several different forms. This is the action of putting the last byte read back into the input stream, so it will be read again by the next _Get. This action works only for the most recently read byte on the current input stream, although that can be any input stream:

```
_Unget();
```

For a more general mechanism, it is meaningful for some streams, such as sequential disk files, to change the current position to an arbitrary position in the stream, either indexed absolutely from the origin of the stream, or relative to the current position:

```
_Seek( ucb, position, relative );
```

Here position is an int_32, and relative is nonzero if the indexing is relative to the current position. This function can be used for streams that are opened as read or read/write. The function cannot be used for streams that are opened as write only, mainly as a consequence of, and interaction with, buffering.

On top of this basis, a few higher level functions are provided for convenience. Numbers are read free format, with the C convention where 0xNNN represents hexadecimal, 0NNN octal, and NNN with non-zero leading digit represents decimal, by:

```
number = _Get_number();
```
[4]

Numbers can be printed in decimal by:

```
_Put_decimal( number );
```

or in hexadecimal by:

```
_Put_hex( number );
```

and strings can be printed by:

```
_Put_string( string );
```

The above can all be combined by:

```
_Printf( format, item1, item2, ... );
```

which prints the variable number of items according to the format string. As in Unix, the characters in the format string are directly output, except that %% is printed as %, %c prints the corresponding item as a character, %i or %d prints the corresponding item as a decimal integer, %h or %x prints the corresponding item as a hexadecimal integer, and %s interprets the corresponding item as a pointer to a string and prints that string. Full format control as in Unix is not supported. In particular, there is currently no floating point conversion.

IMPLEMENTATION OF OTHER INPUT/OUTPUT MODELS

Not all I/O devices fit the stream I/O abstraction. Moreover, for some I/O devices, even though the abstraction can be forced to fit by embedding control in the stream of bytes, there are realtime applications that cannot afford to pay the performance penalty of such a uniform implementation. Consequently, it is necessary for Harmony to allow the user to design and implement his own I/O abstractions, and for tasks in a user program to have the ability directly to perform I/O operations and to handle interrupts. Even when the stream I/O abstraction is used, these facilities may be needed because, Harmony being an open system, the user may have unique devices for which he must therefore supply the tasks supporting the streams. In any case, the user will have to specify for any particular program which of the available servers are needed, because configuration of a control computer for realtime applications varies so widely.

To allow the user program to do I/O directly, with maximum performance, Harmony ignores the supervisor versus user state distinction present in most hardware, and runs the whole program in

[4] The names of the stream I/O functions are those in Release 3.0 of Harmony. Programs using Release 2.0 format can use an appropriate header file to alias the old name to the new one.

supervisor state. For memory mapped I/O, such as used with the Motorola MC68000, the task owning an I/O device can issue the I/O instructions directly in C — other processors, which have special I/O instructions, would require assembly language subroutines. The actual I/O commands depend, of course, on the device, and might involve either programmed I/O or DMA. Synchronizing the task with the I/O device can be done busywait by reading status registers, either pre-busywait by checking that the device is ready before issuing an I/O command, or post-busywait after an I/O command is initiated by checking for the device to have completed before going on. There are situations where busywait synchronization is the preferred solution, because the busy state is guaranteed to clear in less time than would be required for a context switch to another task. However, in general it is unsatisfactory for three reasons: because in uselessly consuming processor cycles it prevents other tasks of the same or lower priority from running, because it is difficult to do with more than one device (especially input devices), and because it can readily lead to deadlocks. Generally, better practice is for the task to initiate an I/O command to the device, to synchronize by blocking while waiting for an interrupt to signal completion of the command, then to retrieve any results. (A pre-wait version, similar to pre-busywait, is occasionally appropriate instead.)

Harmony provides a primitive for awaiting an interrupt:

```
_Await_interrupt( interrupt_id, rply_msg);
```

The semantics of this primitive are something like those of the _Receive primitive. The task executing the primitive blocks until the interrupt identified by interrupt_id occurs. The rply_msg argument of the primitive is provided because experience has shown that many peripheral devices have volatile data which must be recorded at the instant the interrupt is first detected, for the data will be lost by the time that the waiting task can be activated. It is important to note that defining this primitive for most efficient implementation requires that at most one task can be waiting for a given interrupt, that a task can be waiting for only one interrupt, and, of course, that the waiting task must run on the processor physically connected to that particular interrupt.

Usually the task controlling a device will issue the I/O command, then call _Await_interrupt, and after the interrupt occurs and the primitive returns will access device registers to retrieve the results. It is important to notice that issuing the I/O command might cause the interrupt to happen immediately, before the task can call _Await_interrupt. Harmony treats interrupts as spurious when there is a protocol error, in that an interrupt occurs before there is a task waiting for it, as well as when noise or configuration errors result in interrupts occurring for which no handler has been provided. To ensure that this is not a problem, the task that will call _Await_interrupt for a particular interrupt must run at the Harmony priority corresponding to the hardware level at which that interrupt will occur because then the priority scheduling rules ensure that the hardware will mask the interrupt until this task is blocked and the priority falls. Another common task structure is that one task calls _Await_interrupt to catch the interrupt when it happens, but a different task issues the I/O command whose completion will cause the interrupt. This is used, for instance, where several disk spindles are under a common controller. To resolve the critical section with respect to controller registers, the same task issues all seek, read, and write commands, but because seeks, reads, and writes on the different spindles can overlap, a separate notifier task is used for each spindle to catch the command completion interrupt. The task that issues the commands must reply to the notifier so that the latter can call _Await_interrupt before the interrupt happens, which as in the earlier case implies that the task calling _Await_interrupt must run at the Harmony priority corresponding to the hardware level at which the interrupt will occur. In this case, however, one must also ensure that the reply to the notifier is executed before the interrupt can happen, which requires either that the task that issues the I/O command must also run at the priority of the notifier, or that the reply be made before the I/O command is issued. (There have been some computers where the design of controller hardware actually forced tasks issuing the I/O command and awaiting the interrupt to run at different hardware priorities.)

It would be nice only to have to go down to this level of abstraction, but because Harmony is an open system, indeed just because it is a configurable system, it is often necessary to descend further and implement the actual interrupt handler. Although doing so is not required, Harmony realizations typically use a two-level interrupt handling scheme. The first-level interrupt handler is invoked by the actual hardware interrupt, saves the registers on the stack of the interrupted task, and transfers to the proper second-level interrupt handler for the particular processor interrupted. Having a separate first level is useful, for instance, if the hardware does not provide distinct interrupt vectors for each processor, and the first-level interrupt handler must do that. (In this case the last step of state saving, saving the stack pointer in the task descriptor of the interrupted task, must be performed by the second-level interrupt handler.) It is also useful if various members of a processor family have different registers that need to be saved, but a device, and hence the second-level interrupt handler, might be used with any of these processors.

The second-level interrupt handler must first identify the interrupt source. Although this may be available immediately as a consequence of vectoring, or may be available in some register, quite often it is not. Interrupts that the task level of abstraction would like to treat as distinct are often multiplexed for hardware reasons. For example, many UART (*universal asynchronous receiver transmitter*) chip designs only generate a single type of interrupt, even though software structures usually want different tasks to wait for input, output, and modem control events. Demultiplexing typically involves reading status registers, although if many devices have been connected to the same interrupt vector it may involve polling. Once the interrupt source has been identified, the second-level interrupt handler must find whether there was a task waiting for that interrupt. The interrupt id used at the task level of abstraction is actually the byte offset in the vector _Int_table at which a pointer to the task descriptor of the waiting task, if any, is stored. (The interrupt id is also the byte offset into the vector _Dev_data_table at which is stored a pointer to any data that must be shared between the second-level interrupt handler and the waiting task, such as readable copies of write-only device registers. See also Application Note n008 [5].) If the interrupt was spurious, that is, if the entry of _Int_table was 0, the source of the spurious interrupt is recorded in the external _Spurdev and the external _Spurcount is incremented. Volatile data associated with the interrupt must next be saved in the rply_msg provided by the waiting task, and then the interrupt must be cleared. If the interrupt was not spurious, the waiting task is now put on its ready queue and dispatched, else the interrupted task is reactivated.

The second-level interrupt handler is typically code written in assembly language, and because the stack of the interrupted task cannot be used and the task to dispatch has not yet been determined, no stack is available. Interrupts are disabled throughout the second-level interrupt handler. Note that when the waiting task is put on its ready queue, advantage can be taken of the fact that it will be the only task on its queue: had there been another, the priority of the active task would have been at least that high, and the interrupt would not have been taken because it would have been masked by priority hardware.

Sometimes the above task level abstraction is not fast enough for the the arrival rate of the physical interrupts, and handling one interrupt might cause succeeding events to be missed. The problem might be that context switching itself is too slow, or it might be that too much immediate processing is required, or it might be that passing the processed data to another process by message passing is too slow. Often the real problem is that there is a clash in levels of abstraction: the task level of abstraction really wants to deal with high level events which occur at a rate that could be handled, but the hardware is causing physical interrupts for low level subevents which occur at a much higher rate — the problem could be finessed if the hardware device operated at the higher level. A classic example is the DDCMP communications protocol, which really is defined in terms of packets, but which is implemented with byte at a time interrupts through a standard UART. To solve such a problem in Harmony, we implement such a higher level device outside of the Harmony abstraction, and use the concept of *fake interrupts* to activate the waiting tasks at the task level of abstraction. Implementing outside the Harmony abstraction could mean with a separate processor, or it could mean in software on the same processor, but with an interrupt at a hardware priority even above the Harmony disable. Obviously software outside the Harmony abstraction cannot make use of Harmony services, and preempting even disabled Harmony code sacrifices some of the strong things that can be proved about normal Harmony programs. In the DDCMP example, moving the packet between the UART and a buffer, including header and trailer processing and byte stuffing or stripping for transparency, can be done by a software automaton, running perhaps at unmaskable interrupt level. This ensures that the byte level handling will happen if the processor is at all fast enough to handle the communications line at that speed, and the task level interaction with this automaton is all in terms of packets, providing a buffer containing a packet to transmit or a buffer in which to receive a packet, and awaiting an interrupt to indicate completion of processing of the packet. Of course there is no such real interrupt, but the interrupt can be faked by jamming the waiting task directly on its ready queue, and forcing a redispatch by causing an interprocessor interrupt (which will be serviced immediately or, if the processor is disabled, as soon as it becomes enabled, or, if _Td_service is already in progress, when it completes). Jamming a task directly on its ready queue, regardless of what Harmony is doing, is safe if that task is the only task that can run at that priority, provided that the tail pointer of the ready queue is updated before the head pointer — the ready queue must have been empty, the only operation Harmony could have been doing with this ready queue is examining the head pointer to determine whether there is a task to dispatch, and so atomicity of writing the head pointer ensures it is safe.

The machinery described in this section is sufficient to provide direct control over any physical device. It should be recognized, however, that the task structure of Harmony facilitates creating *smart peripherals* where software implemented as a controlling task provides an interface closer to the level of device actually desired, instead of what the raw hardware provides. Not only do the messages serviced by this controlling task define a higher level device to client tasks, but if the controlling task runs on a different processor from that of the client, the latter sees the computational load reduced as well. Eventual migration of the *smart peripheral* task into specialized hardware is also made more convenient if that hardware uses the same interface. An example using a conventional computer peripheral would be a *file store device* implemented by a task controlling an ordinary disk, where the task also did space allocation, catalog management, etc., so that the operations on the smart device were to create and destroy linear address spaces, grow or shrink them as necessary, and read or write them — independent of physical block location, let alone cylinder boundaries, bad tracks, etc. An even higher level would be a *file system device* implemented by a task using the *file store device*, where this task provided a hierarchical file system with access control, concurrency control, etc. Another example, this time from robotics, would be a *vision system device* implemented by a task controlling a TV camera and framegrabber, where the task responded to client tasks by indicating objects seen, together with their types, locations, and orientations.

Normally, a client task wanting to do I/O does it by exchanging messages with the server task which supports that I/O model for the desired device. The types of messages and the protocol of how they are used are what really define an I/O model. As discussed in the next section, an I/O server task must report for service to the directory task and must be prepared to receive open and close messages. What other messages it handles depend on the I/O model supported and on the device controlled. A simplified version of stream I/O, for instance, could be implemented by handling a read message to provide another buffer of input, a write message to output another buffer, a flush message to ensure that any buffers associated with the connection are flushed, as well as the open and close messages. The actual version of stream I/O provided with Harmony does not work this way, however, as the seek operation to move the current position in the stream has serious ramifications. Instead, the read and write requests are combined into a single advance window message, the model being that the stream should be thought of as the beginning part under control of the I/O server, concatenated with the window into the stream (the buffer) held by the client, concatenated with the ending part under control of the I/O server. When the user task is finished manipulating the current window (buffer), an advance window message is sent to the I/O server, returning the current window if it was modified, and requesting the next window if the connection is open for read.

Many server tasks may support the same I/O model. The server task may be the task directly controlling the hardware device or, as discussed above, a task implementing some higher level abstraction. The client task uses a symbolic name to establish, at run time, a connection with the appropriate server. This provides *device independence* within the class of servers supporting that I/O model.

SERVERS AND CONNECTIONS

Harmony is a connection oriented system. What this means is that tasks that communicate frequently maintain state about each other, in particular about the effect of earlier communication. Connections are often discouraged in distributed systems, partly because they are assumed to be expensive, and partly because of the difficulty in coping with the inconsistencies that can affect this state when messages are lost or damaged. However, connections in Harmony are inexpensive, and under either the shared memory or thinwire model of computation, damaged or lost messages are precluded. Most importantly, the effect of action requests in realtime systems is often to cause state changes in the real world (for example, the robot moves from its rest position, or the tracking radar locks on the selected target), and the program needs to reflect this real world state. A more detailed discussion of servers can be found in Application Note n002 [5] and in [2].

The function _Open was introduced in the discussion of stream I/O, in order to associate streams with servers supporting them. More generally, _Open is the mechanism for establishing a connection between a client task and a server. As stated earlier, _Open has two parameters, an open directive and a userid, as shown below:

```
ucb = _Open( directive, userid );
```

where directive is a string that symbolically defines the connection of interest and userid is an unsigned integer value that is an application-defined identification which can be used to enforce access control. The value returned, ucb, points to a dynamically allocated struct called a *user connection block* that defines the server and connection number associated with this connection and points to the ucb_xtra, a second dynamically allocated struct that contains the buffer and supporting pointers. The user of a connection need not be aware of these details. The value 0 is returned if the open failed. There is no fixed limit as to how many connections may be open. In general, the same connection may be opened more than once, each with its own current position. This is typically done when several tasks use the same stream or service.

A connection which no longer needs to be accessible should be closed:

```
_Close( ucb );
```

Destruction of a task automatically closes all its open connections. It should be noted that closing a connection does not imply flushing any buffers (see below) and this must be done explicitly.

The open directive is not just a symbolic name for the server, but is more like a command line in other operating systems: it is a string beginning with a pathname of possibly several components and containing options for the connection to be established. The first component of the pathname, which can be at most 30 characters and includes a terminating ':', is the symbolic name for the server. Additional components, which are separated by '/', are interpreted by the particular server, for example a file system server might treat them as subdirectory names and the file name, or an automatic calling unit server might treat them as parts of a telephone number. Options are position independent and are of three kinds: on toggles, off toggles, and value parameters. An on toggle is a '+' followed by a keyword, an *off toggle* is a '-' followed by a keyword, and a *value parameter* is a keyword and a value separated by an '='. A *keyword* is a sequence of characters delimited for on toggles or off toggles by white space and for value parameters by the '='. The *value* for value parameters is a sequence of characters delimited by white space. If a value containing white space is desired, the sequence of characters must be enclosed by quotation marks. (Note that if the open directive is a literal string, quotation marks enclosing a value inside the directive must be escaped.) The interpretation of on toggles and off toggles is left to the server, but the intended semantics are that a toggle has two values, on and off, and that there is possibly a default value which explicit appearance of the toggle overrides. The interpretation of value parameters is also left to the server, but the sequence of characters which form the value is made available as a string which the server may evaluate as the ASCII representation of a number, or may interpret as a symbolic name that can be looked up in an *evaluate table* to find a corresponding numeric value, or may parse further, or may simply use as text. Parsing open directives is supported by the library functions which encourage the use of position independent syntax:

```
svector = _Components( str );
svector = _On_toggles( str );
svector = _Off_toggles( str );
pvector = _Value_parameters( str );
number = _Evaluate( table, str );
check = _Valid_number( str );
_Freeparse( parse );
```

The function _Components() parses a string into pathname components. The delimiters are ':' and '/'. The ':' delimiter is part of the component; the '/' delimiter is not part of the component. The pathname is terminated by white space (i.e., blank, tab, or newline) or by the null byte. The structure returned is a vector of strings but is allocated as a single block.

The function _On_toggles() parses a string to collect the on toggles. Characters before the first white space (blank, tab, or newline) are ignored. The structure returned is a vector of keyword strings but is allocated as a single block. Similarly, the function _Off_toggles() parses a string to collect the off toggles.

The function _Value_parameters() parses a string to collect the value parameters. Characters before the first white space (blank, tab, or newline) are ignored. A value parameter is a keyword following white space, followed by a '=' character, followed by a value string. A keyword is a string of characters terminated by the '=' character. A value string is a string of characters terminated by white space or a null byte. The structure returned is a vector of VALUE_PARAMETER structs, but is allocated as a single block.

The function _Valid_number() checks to see whether the string str holds a valid ASCII representation of an integer number.

The function _Evaluate( ) searches a table for a given string. If it is found, the value associated with the string is returned. If the string is not found or a table not supplied, but the string is a valid ASCII representation of an integer number, the string is evaluated as such and the appropriate value returned. If none of the above is true, then 0 is returned. Ambiguity whether a returned value of 0 is a legitimate number is removed by the prior use of the _Valid_number() function. White space is presumed to have been removed previously.

The function _Freeparse() is provided to protect users from trying to free the strings of a parse individually, because these strings are allocated in a single block with the pointers to them.

The userid argument to _Open is provided because some servers, for example a database server, may want to enforce access controls on whether they will establish a requested connection. Having the user ID be available directly to the application program, with the consequent risk of forgery, may seem unusual compared to the practice in multiuser timesharing systems, but when protection is needed in realtime applications it is usually protection with respect to human operators interacting with an application program, not protection against the application program itself. Currently, requesting and validating the identity of an operator and encoding this as a user ID is completely left to the application program; no supporting functions are supplied. Most servers do not check the user ID, consequently a value 0 is often used for this argument.

Calling _Open causes two messages containing the open directive to be sent. The first is to the directory task. The directory task matches the first component of the pathname against the list of symbolic names of registered servers, and returns the corresponding server task ID if a match is found, otherwise returning 0. If a server was found, the second message is sent, this time to the server. The server sets up the connection or can deny it, based on the rest of the pathname, on the user ID, on what connections the server has already made, etc. The server then replies with a record containing a number identifying the connection, and the ID of task the client should communicate with for this connection (normally the server itself, but possibly an agent task instead), as well as the size of ucb_xtra block which this connection requires. (The ucb_xtra block is used, for instance, for buffers this connection requires on the client end.) Notice that by monitoring the success of this reply, the server can detect if the client task has been destroyed while the connection was being established, and thus can avoid leaving resources tied up in a connection which will not be used because the client no longer exists. Once a connection is established, messages from the client to the server must identify the connection, for a server may simultaneously be supporting multiple connections, even to the same client task — the file system task might open several files for the same client task, for instance.

A connection is closed by the client task sending a CLOSE_REQUEST message to the server; this will be done automatically for any connections still open when a client task is destroyed. This is the third major aspect to connections (the other two being runtime symbolic binding and textual initialization options). In many situations, it is important for one task to know when another task dies, in order to perform wrap-up, release resources, spawn replacement tasks, etc. This can be done without the connection mechanism, but only by creating vulture tasks that do a receive specific on the task to be monitored without ever being sent to by it, relying on the fact that the message passing primitives fail when the correspondent dies. The connection mechanism is much cheaper and more explicit.

The directory task initially has no built-in knowledge of which servers are available, so any server must send a REPORT_FOR_SERVICE message to the directory task reporting the symbolic name by which it is known. A mechanism is provided to allow servers to register under a secondary name, in addition to the primary name.

The reporting is conveniently accomplished by calling:

```
success = _Report_for_service( name, msg_type );
```

where name is a string indicating the symbolic name by which the server will be known and msg_type is one of REPORT_SECONDARY_NAME or REPORT_FOR_SERVICE. Any secondary names must be reported before the report for service. It is possible for a different task to register at a later time under the same name, thus temporarily or permanently taking on the responsibility of providing the services identified by that name.

To facilitate a server keeping track of those connections it has open with clients, routines are provided to allocate a connection table, allocate an entry in a connection table, lookup a connection table entry, and free a connection table entry:

```
table          =   _Alloc_connection_table( init_num_entries,
                        grow_num_entries, max_num_entries,
                        data_blk_size );

scb            =   _Get_connection( table, client, new_connection,
                        con_data_blk );

scb            =   _Lookup_connection( table, client, connection );

connection     =   _Free_connection( table, connection );
```

where init_num_entries is the number of entries initially (the table is grown later as needed), grow_num_entries is the amount by which the table is to be grown each time it grows, max_num_entries is the maximum size beyond which the table will not be grown (except that a 0 value indicates no limit), data_blk_size is the size needed for the server connection block for this server, table is a pointer to the allocated structure, client is the ID of the task with which the connection is being made, new_connection is a pointer to where the new connection number should be returned, con_data_blk, if supplied, is a pointer to the connection data block, scb is a pointer to the server connection block, connection is the number of the connection to be looked up or freed, and _Free_connection returns its connection argument as its function value or 0 if it fails.

The application program is responsible for creating whatever servers it will require. This is best done at the start of the function main(), in the first user task. Care must be taken to allow for the critical race where the server must report for service to the directory task before the first client task can open a connection with it. This is conveniently done using the _Server_create function:

```
id = _Server_create( global_index, init_list );
```

which creates an instantiation of the task template specified by the global_index, if init_list is nonzero supplies the initialization records on init_list, one at a time, as reply messages to the newly created server task as it requests them, and returns the ID of the server task after the task has sent a message confirming that reporting is complete. _Server_create returns 0 upon failure to create and initialize the server.

The *scatter/gather* aspect of init_list is very important in designing generic servers. A server is generic when several similar servers can be combined as instantiations of the same one by supplying the differences through initialization records. For instance, instead of having a different server for each terminal on a system, each with device addresses and characteristics hard coded into the functions the servers execute, the use of a generic server not only saves code space but also greatly assists a maintenance programmer reading the device assumptions a program makes and changing them. For reading and managing such configurations, a static description in a compile time initialized record structure is greatly preferable to attempting to combine all the device characteristics in a single record, or to building the configuration record at run time. Configuration record structures maintained in files and accessed as needed at run time also benefit from exploiting scatter/gather for modularity. The formats of the records in init_list are server dependent, but each must begin with the following struct:

```
struct INIT_REC
   {
      struct STD_RQST          STANDARD;
      struct INIT_REC          *IR_NEXT;
   };
```

where STD_RQST contains two unsigned fields: an unsigned 32-bit MSG_SIZE and an unsigned 16-bit MSG_TYPE, MSG_TYPE in this case being used to distinguish the various types of initialization records for a particular server. A server requiring initialization records should continue to request subsequent records until IR_NEXT is 0. If a server is going to register also under a secondary name, the name should normally be declared with the same initialization record type for all servers, i.e., a struct ALTNAME_INIT_REC is declared in sys.h and servers providing secondary names should use it.

The design and use of servers is a sufficiently important topic that although servers have been referred to extensively in the foregoing discussion, it is worth collecting together some of the issues. The first thing to recognize is that a server is analogous to a library subroutine. Whereas a private subroutine that is called from only a few well-known places can rely on undocumented aspects of the context in which it is called, a library subroutine must have clean, well-defined interfaces so that it will be useful to call from code not even thought of at the time the library subroutine was written. Like a library subroutine, the server task must provide a generally useful service, and the need for clean, well-defined interfaces applies to servers as well. What distinguishes a server from a library subroutine is that the server often must maintain state across service transactions, perhaps just across transactions for the same client, or perhaps even across clients. This state is inconvenient to represent in data carried by the clients, but can be conveniently represented in data local to the server task.

Because the server may maintain state for several clients at any time, 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 open connection to the same server. Because the client will not, in general, have created the server, it cannot be expected to have inherited the task ID of the server from somewhere, and so must identify the server by symbolic name. Because 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. Given that the mechanism Harmony provides for such binding is connections,. a server must support OPEN_REQUEST and CLOSE_REQUEST messages, must register with the directory task, and must provide connection numbers. Servers often have some parameters that must be set before normal client transactions can be handled, so the toggles and value parameters in the open directive can be used to ensure these parameters are set before use. A server often needs to know if a client dies, perhaps while the service transaction is in service or perhaps between service transactions, so the system-supplied CLOSE_REQUEST messages on connections are important. To reduce the overhead of having a send–receive–reply on every service request, a server may require the client to provide a buffer in the client's space to provide some caching. This buffer would be in the UCB_XTRA, whose use is dependent on the particular server and on any library functions used by the client to interact with that server. It should be noted that because a server accepts one request at a time, it serializes access to any shared resource which it controls and thus can solve critical races and mutual exclusion problems. It need not process only 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 adequate. 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.

Several examples of servers are included in Harmony, including a calendar clock server, a server to provide explicit scheduling, servers to support interactive terminals and to act as virtual terminals to remote computers with multiwindow capability, a TCP/IP Ethernet server, a file device server and file system server and a null server.

USER'S VIEW OF A PROGRAM

Recapitulating and extending what was said above, a user sees a program as a set of processor images, one for each processor in the hardware configuration. A processor image is obtained by linking three fundamental user supplied externals with whatever user supplied externals and functions are brought in, directly or indirectly, by the references in the three fundamental externals, as well as with the Harmony kernel and support libraries. (In a ROM system, of course, the Harmony kernel and support libraries linked to are simply ROM entry points.) The preferred method of program development is thus that all user code and data, other than the three fundamental externals for each processor, are maintained in relocatable object code libraries.

The three fundamental externals for a processor image are:

```
unsigned _Pnumber
```

which must be initialized with a value specifying which processor this image is for;

```
struct TASK_TEMPLATE _Template_list[]
```

which must be initialized as a vector of all task templates for tasks that can be created on this processor; and

```
struct INT_PAIR _Interrupt_list[]
```

which must be initialized as a vector of all interrupt handlers required for this processor (excluding the interprocessor interrupt, which is always included). The two vectors must be terminated by a null TASK_TEMPLATE and a null INT_PAIR, respectively, i.e., in which all fields are 0. The INT_PAIR entries are the following struct:

```
struct INT_PAIR
   {
       uint_32 INT_PHYSICAL_ID;
       void    (*INT_HANDLER)();
   };
```

where the field INT_PHYSICAL_ID is an unsigned 32-bit integer that identifies the physical interrupt in whatever way is appropriate to that machine (interrupt priority level, bus vector, interrupt vector address, etc.) and INT_HANDLER is the second-level interrupt handler code to be executed when that interrupt is taken.

On processor 0, three special task templates must be included in _Template_list. A task template with GLOBAL_INDEX of 1 must be provided as the first user task, which the system starts and for which the root function is normally the user function main(). This is the start of the user's program. A task template with GLOBAL_INDEX of 2 must also be provided as the directory task, for which the root function would normally be _Directory, but which might conceivably have other than the default priority of 5. Finally, a task template with GLOBAL_INDEX of 3 must also be provided as the gossip task, for which the root function would normally be _Gossip, but which might conceivably have other than the usual priority of 5.

In choosing how to structure the source code, it is wise to remember that all compilation source should have an include file compiler.h to define compiler dialect differences and also a file sys.h, in order to have necessary structure and manifest constant declarations. It is also good practice to include a file giving all template index assignments to be used throughout the whole program. Remember to include the templates for all servers that will be used, as well as for the worker tasks they create. When choosing identifiers, note that Harmony uses the convention that all Harmony-defined objects, both internal and visible, start with an underscore in order that accidental name conflicts can be avoided and system use can be recognized. Another convention is that structure tags, field names, and manifest constants are all defined by upper case only names, and that externals and functions have names for which the first alphabetic is upper case and there is at least one lower case alphabetic.

Linker parameters are different for processor images for different processors, which is most conveniently handled by working with libraries, making linking an explicit step in program building, and using a specific command file for linking processor images for each processor. The usual arrangement is to have RAM starting on an arbitrary megabyte boundary on each processor, with each processor owning a range of addresses in the linear space. The processor image is linked with that as base. The first routine loaded into the image is forced to be _Pre_Setup which must be at a known address because control transfers there to start execution. This routine loads four addresses in the application program that the linker determines and the kernel needs to know (the locations of the three fundamental externals and the first location available for the storage pool) into registers and then transfers to _Setup. The latter, which can be anywhere including in ROM, sets up the C environment, then calls the Harmony kernel to build and initialize Harmony. Processor 0 initializes its Harmony kernel first, then opens the multiprocessor gates so the other processors can initialize their Harmony kernels. (A gate is a location used for synchronization.) In a downloading situation, on reset all processors other than processor 0 close their multiprocessor gate, then enter a busywait loop waiting for their multiprocessor gate to open, and processor 0 executes whatever code is necessary to accept a download of code first for the other processors, and then for itself. The end of downloading causes it to execute _Pre_Setup and initialize itself. Processor 0 must then open the multiprocessor gates for the other processors, then poll the gates to determine when all the other processors have completed initialization before it can start the first user task. The other processors initialize when their gates open.

DEVELOPMENT ENVIRONMENT

There are several environments for the development of Harmony and of applications intended to run under Harmony. The system in use in our laboratory is a network of Apple Macintosh personal computers. Appendix F describes, for the Macintosh environment, a recommended configuration of hardware and software, suitable as a development environment. Several tools described below are needed specifically for development on the Macintosh. Application Notes n007 and n013 [5] give a more detailed discussion of using the development system.

Development elsewhere is being carried out on Sun workstations or on time-shared VAX's under Unix. Development has also been carried out on a VAX/VMS using a Whitesmiths' C Compiler and it is still possible to use this environment, but it is not recommended, due to obsolescence of the compiler. As distributed, Release 3.0 of Harmony includes versions for the Macintosh development system under the Mac operating system, for A/UX (Unix) on the Macintosh, and for Whitesmiths' C on VAX/VMS.

Support Tools

Harmony currently provides the following support tools: debug, listing, examine, bound, fixaddr, fixexe, fixmsr, makemsr, listtree, and mmake. Only the first runs under Harmony itself. All the others run in the development environment. The versions that run on the Macintosh do not yet have a "Mac-like" user interface. Some tools are specific to the development system being used. For example, examine and bound are intimately involved with the exact form of a stored executable image and with the generated code from the chosen compiler. All the tools are written in a portable manner with well-identified dependencies on the development system and on the target system. Porting to a environment requires rewriting these dependent routines.

The tools fixaddr and listtree are aimed specifically at the development environment on the Macintosh. The tools fixmsr and mmake are used in Unix development systems. Several tools, including fixexe, are needed to augment or correct the capability to generate downloadable records under Whitesmiths' C. The last tool mmake provides a bridge between the source management being used in Harmony (see Appendix E and Application Note n005 [5]) and the Unix tool make.

debug

The question of how to debug a multitask program is still very much unanswered and may even require hardware support. For now, this very simple debugger is all that is available for Harmony. Breakpoints can be planted at run time or can be compiled into the source as calls to the function _Breakpoint. Values to be inspected must be identified by absolute address, and values are only available in decimal or hexadecimal. On the other hand, the debugger can print certain known system data structures, such as task descriptors, and works with interrupts enabled and with breakpoints on code in processors other than the one with the terminal. The debugger also supports *use bits*, a debugging technique for recording whether an event occurred that is sufficiently efficient to be left in a production program. Considerable evolution of the debugger can still be expected. The current version requires a stream I/O server "DEFAULT:" to be available. A more detailed description can be found in Application Note n001 [5] and in [6].

listing

The style of programming encouraged by Harmony consists of many small functions, each stored in a separate file, and of many header files, each defining structs and macros associated with a different abstraction. Compilations are done by compiling inclusion files that reference the appropriate source. A readable listing needs to be arranged as a book, with cover, table of contents and numbered pages with headings. This tool, given an inclusion file, produces this listing.

examine

It is sometimes necessary to examine a fully bound executable image, to determine overall characteristics such as the size of the text or data segments, to look up external symbols, to disassemble instructions or display initialized data, or even to patch erroneous values. This tool, given a .exe file produced by Whitesmiths' linker or by fixaddr, or a .out file produced by the Unix linker, does this.

bound

Bounding the stack required for a given task involves discovering the call graph of procedures called from the root function, or called by procedures called from the root function, etc. The stackframe size required for each of these procedures must be determined. All possible nesting lists can then be simulated to determine the maximum stack size required. Two things force this to be an interactive, rather than a wholly automatic, process. First, sometimes there are indirect procedure calls, i.e., calls to procedures that are formal arguments of the calling procedure or that are values returned by calls to other procedures or that are values of procedure-type variables. These cannot be known without running the program, so the possible procedures must be supplied interactively. Second, the call graph may involve a cycle, implying possible recursion, and while the stack requirement per cycle can be made available, the number of levels of recursion to support must be supplied interactively. This tool, given a .exe file or a .out file, allows the user to determine stack sizes for several root functions in that file.

fixaddr

This tool is used to convert from the format of the Macintosh executable image to the conventional Unix a.out form. On the Macintosh there is a software-based memory management scheme which assumes that the executable image is composed of segments and that all addressing within a segment is in a position-independent form (which is desirable for Harmony). Addressing between segments uses, by convention, register A5 as a base. Before the image can be downloaded the addressing is made absolute and the form is converted to the Unix a.out form. The tool fixaddr accomplishes this conversion.

fixexe

The tool fixexe converts a Whitesmiths' executable file from fixed-record-size-1 format to fixed-record-size-512 format. Not all programs and I/O routines can handle fixed-record-size-1 format. This may be due to a VMS bug. In particular, the Unity C I/O library cannot correctly read fixed-record-size-1 files.

fixmsr

The fixmsr tool corrects Motorola S-Record files generated by the A/UX hex command. Fixmsr inserts the null address field into S0, S1, S8, and S9 records (which is left out by hex) and corrects the length and checksum fields. Output is to the standard output stream.

makemsr

Most vendor-supplied BOOT ROM's require that the code to be downloaded be in the form of S-records. The tool makemsr produces an S-record file for a Harmony program. It is used on those systems that do not provide support for generating S-records. The tool converts the binary .out image into character encoded form (S-records) supported by Motorola. The form avoids forbidden bit combinations that could collide with communication software, but is verbose, hence makes downloading slow. An alternate method for downloading is to use the XMODEM protocol, if it is supported by the BOOT ROM. The XMODEM protocol does not require a separate translation step and requires fewer bytes to download a given image.

listtree

The listtree tool for the Macintosh development environment produces a list of all the files and folders in a specified subtree. The resultant document is useful for managing or verifying large source trees or for executing a command on every file in a subtree.

mmake

The mmake tool provides a bridge between the Harmony style source management and the more traditional approach using make. This tool uses an A/UX linkage editor script file (.link file) as a starting point and generates a Unix style make file. Mmake determines the source files upon which the target is dependent by following the target's link file, the inclusion files and shell scripts that created the object files. A makefile is then generated for the target, containing the complete dependency list. The resulting makefile has the suffix ".mk".

PERFORMANCE

It is difficult to make quantitative statements about performance for a portable system, because measured values depend upon tuning of the kernel code, which compiler is chosen, the microprocessor instruction set, the clock speed, the memory used, other aspects of the processor board design, etc. Nevertheless, qualitative statements about performance are important, to make sensible design tradeoffs in application programs. The measurements described here were made on a system with kernel code written for clarity rather than tuned to the target. The timing results were obtained by running the timing program, which is distributed with Harmony. There are versions of timing for all the main targets supported in Release 3.0.

The values shown below are for the Io Inc. V68/32 system, using a MC68020 processor running at 16 MHz with no wait states. The timing program and Harmony were compiled using the Consulair C compiler. (All versions of the kernel developed with the Consulair tools have the function _Convert_to_td recoded in assembly language.) The values are approximate and represent the lower bound, because the only tasks active were those associated with the timing program. Also, differences in byte alignment introduce some variability in the results.

_Create a task	1200 µs
send-receive-reply (short msg)	460 µs
_Copy_msg of 200 bytes	380 µs
_Open "NULL_SERVER: +r BUFFSIZE=20"	4300 µs
_Close	700 µs

The timings for stream I/O are have been measured (microseconds per byte) and fit the following relations:

$$\text{\_Get} = 20 + \frac{600}{\text{bufsize}} \ \mu\text{s}$$

$$\text{\_Get\_Record} = 8 + \frac{620}{\text{bufsize}} + \frac{7}{\text{recsize}} \ \mu\text{s}$$

$$\text{\_Put} = 22 + \frac{600}{\text{bufsize}} \ \mu\text{s}$$

$$\text{\_Put\_Record} = 8 + \frac{570}{\text{bufsize}} + \frac{12}{\text{recsize}} \ \mu\text{s}$$

An important characteristic is response time to an interrupt, i.e., delay from the moment of interrupt until the _Await_interrupt of the waiting task returns. This delay is measured as 39 µs. An indication of how good this really is, can be seen in the following comparison. The fastest assembly language interrupt handler, which does nothing more than to save the state of the interrupted task and to sort out which processor's interrupt vector to use, takes about 50% of that time.

The second important characteristic is the send–receive–reply cycle time. When both tasks are running in the same, otherwise idle, processor, the cost given in the table above is 460 µs. When the two tasks are running in different, otherwise idle, processors, the value is lower, because of parallelism in execution of the primitives on the two processors.

An important aspect of the message passing implementation is that it is distributed. Only the processor or processors that the communicating tasks are running on participate in the protocol: there is no central *switching* processor involved. The actual bus cycles used on the shared bus are so few that contention for these will not be a bottleneck. Message passing between tasks on the same processor is treated uniformly with message passing between tasks on different processors and not by special code.

The measured times given above are quite commensurate with the best task context switching times achieved by other systems on similar hardware, even uniprocessor systems. It is possible and, in fact desirable, to adapt Harmony to a specific family of single board computers and to rewrite several key kernel routines in assembly language so as to achieve significant improvement over the portable version described here. For example, Dy-4 Systems Inc. has done so for its family of boards. They have also implemented a version with restricted functionality, where tasks are not destroyed once they are created. This version has even better performance at the cost of reduced functionality. The intent of the NRC Release 3.0 is to have a portable evolving operating system, that lays the groundwork for local optimization, but does not include it.

In some situations, however, the resulting performance is not adequate. This problem has been recognized as being mainly due to inadequate hardware support for operating system primitives. One solution often proposed is to provide an operating system coprocessor for each processor or to provide microcode support for the primitives. Analysis of the Harmony kernel has identified that even with a coprocessor, an important remaining limitation is the bandwidth for memory accesses across the bus to the kernel data structures [7]. One proposed solution is to provide an operating system coprocessor with high speed static memory for maintaining kernel data structures. It turns out that a planned version of *thinwire* Harmony, in which processors do not have global shared memory, lends itself to implementation of a coprocessor with high speed local memory because kernel data structures are no longer accessed across the bus.

ACKNOWLEDGEMENT

Harmony, of course, is built upon concepts proven in earlier operating systems and languages, as well as new ideas evolved for it. Particular mention should be made of Thoth [2, 3], where this particular process abstraction originated, and BCPL, where the I/O abstraction originated and whose global vector was the inspiration for the global template list. (The I/O abstraction is also commonly associated with Unix, which copied it from BCPL).

REFERENCES

[1] W.M. Gentleman. "If only the hardware... (A software designer's lament)." *Proceedings of the IEEE International Workshop on Computer Systems Organization*, New Orleans, LA. March 29–31, 1983. pp. 88–95.

[2] W.M. Gentleman. "Message passing between sequential processes: the reply primitive and the administrator concept." *Software Pract. Exper.* 11(5): 435–466; 1981.

[3] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. "Thoth, a portable real-time operating system." *Commun. ACM*, 22(2): 105–115; 1979.

[4] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, and B.A. Wichmann. "Rationale for the Design of the Ada programming language." *SIGPLAN Notices*, 14, 6. Part B, June 1979.

[5] D.A. Stewart and S.A. MacKay (eds.), "Harmony Application Notes (Release 3.0)," NRC/ERA-378, National Research Council of Canada, Ottawa, Ont., February 1989.

[6] W.M. Gentleman and D.A. Stewart, "Debugging Multitask Programs," *Proceedings of the Conference of the Army Research Workshop on Parallel Processing and Medium Scale Multiprocessors*, January 6-8, 1986, Stanford, CA.

[7] D.A Green, "An investigation into the design of hardware accelerator for Harmony real-time operating system," NRC/ERB-981, National Research Council of Canada, Ottawa, Ont., March 1986.

APPENDIX A. A SAMPLE MULTITASK PROGRAM

The following program is a simple multitask program which illustrates many of the ideas and primitives discussed in this manual. In this Appendix, the program is run on only processor 0. The function main() is the root function of the first user task. It creates a server for a terminal attached to a serial port and opens a connection for input and output. It then creates a child task and goes into a loop, sending to the child task and printing the response and accompanying text to the selected terminal.

This program also illustrates the source management strategy discussed in Appendix E. The example is from /harmony/example/srtest/ in the distribution tree, and the pair of inclusion files shown are for the version of the program to run on a DY-4 DVME-134 board with the 68020 processor. The program is to be compiled on a Macintosh under the Mac operating system, using the Consulair Mac C compiler. Consequently, all the pathnames are written in the Macintosh syntax.

There are two inclusion files: one for externals and one for code. First, each inclusion file is compiled and then the objects of the two are linked together with appropriate version of the system libraries. Detailed steps necessary to build the library are decribed in Application Note 007 [5]. Following the inclusion files are the relevant included files.

Master:harmony:example:srtest:inc:dy134cmac:single:externs0.c

```
#include    "Master:harmony:sys:src:compiler:macc:compiler.h"
#include    "Master:harmony:sys:src:sys.h"
#include    "Master:harmony:sys:src:servers:tty:ttyinit.h"
#include    "Master:harmony:sys:src:devices:mc68901:dvme134:m68901int.h"
#include    "Master:harmony:example:srtest:src:templates.h"
#include    "Master:harmony:example:srtest:src:dvme134:single:srtest0.c"
#include    "Master:harmony:example:srtest:src:dvme134:ttyinit.c"
```

Master:harmony:example:srtest:inc:dy134cmac:single:code0.c

```
#include    "Master:harmony:sys:src:compiler:macc:compiler.h"
#include    "Master:harmony:sys:src:sys.h"
#include    "Master:harmony:sys:src:servers:tty:ttyinit.h"
#include    "Master:harmony:example:srtest:src:templates.h"
#include    "Master:harmony:example:srtest:src:child.c"
#include    "Master:harmony:example:srtest:src:main.c"
#include    "Master:harmony:example:srtest:src:default:createand.c"
```

Master:harmony:example:srtest:src:templates.h

```
/*
 *   Global indices for task templates:
 *
 *   Note that indices 1-3 are reserved:
 *           main            1
 *           _Directory      2
 *           _Gossip         3
 */


#define     TTI              4
#define     TTO              5
#define     TTY              6
#define     CHILD           10
```

Master:harmony:example:srtest:src:dvme134:single:srtest0.c

```
extern   task   main();
extern   task   _Directory();
extern   task   _Gossip();
extern   task   Child();
extern   task   _Tty_server();
extern   task   _SPi_mc68901();
extern   task   _SPo_mc68901();
extern   void   _Mc68901_int();

uint_32  _Pnumber  =  0;

struct TASK_TEMPLATE _Template_list[] =
   {
      {  MAIN,          main,               1000,    7   },
      {  DIRECTORY,     _Directory,         1000,    7   },
      {  GOSSIP,        _Gossip,            2000,    5   },
      {  CHILD,         Child,              1000,    6   },
      {  TTY,           _Tty_server,        2000,    5   },
      {  TTI,           _SPi_mc68901,       1000,    0   },
      {  TTO,           _SPo_mc68901,        800,    0   },
      {  0,             0                      0,    0   }
   };

struct INT_PAIR _Interrupt_list[] =
   {
      {  5,      _Mc68901_int        },
      {  0,      0                   }
   };
```

Master:harmony:sys:src:servers:tty:ttyinit.h

```
/*
   *   Tty initialization messages
   */

typedef struct TTYI_PORT_INIT_REC
   {  struct INIT_REC        TTYI_HDR;
      char                   TTYI_NAME[32];        /* server name */
      uint_32                TT_DEV_GRP;           /* device group this port is a member of */
      uint_32                TT_GRP_MEMBER;        /* member number in device group */
      uint_32                TTI_DEV_CODE,         /* input logical interrupt */
                             TTI_INDEX,            /* input task template */
                             TTO_DEV_CODE,         /* output logical interrupt */
                             TTO_INDEX,            /* output task template */
                             TTETC_DEV_CODE,       /* etc logical interrupt */
                             TTETC_INDEX;          /* etc task template */
      char                   *TTY_ADDR;            /* I/O address of tty port */
      uint_16                TTY_BAUD_RATE;        /* baud rate, ignored for ports *
                                                   /* without soft baud select */
   };

#define TTYIT_ALTNAME_INIT     1
#define TTYIT_PORT_INIT        2
```

Master:harmony:example:srtest:src:dvme134:ttyinit.c

The following are typical initialization values. This port also has the secondary name DEFAULT. The first declaration is needed to satisfy compiler forward reference.

```
extern struct ALTNAME_INIT_REC      Default_init;

struct TTYI_PORT_INIT_REC                         Tt0_init   =
   {
      sizeof( struct TTYI_PORT_INIT_REC ),          /* MSG_SIZE */
      TTYIT_PORT_INIT,                               /* MSG_TYPE */
      I_CAST(struct INIT_REC *)&Default_init,        /* next initialization record */
      "TEXT_TERMINAL:",                              /* tty server name */
      0,                                             /* no device group */
      0,                                             /* therefore no group member number */
      RECV_0,                                        /* tti device code */
      TTI,                                           /* tti template index */
      XMIT_0,                                        /* tto device code */
      TTO,                                           /* tto template index */
      ETC_0,                                         /* ttetc device code not used */
      0,                                             /* ttetc template index not used */
      I_CAST(char *)MC68901_ADDR,                    /* I/O address of tty port */
      9600                                           /* baud rate */
   };

struct ALTNAME_INIT_REC                      Default_init      =
   {
      sizeof( struct ALTNAME_INIT_REC ),             /* MSG_SIZE */
      TTYIT_ALTNAME_INIT,                            /* MSG_TYPE */
      0,                                             /* no more initialization records */
      "DEFAULT:",                                    /* tty server name */
   };
```

Master:harmony:example:srtest:src:default:createand.c

```
void Create_and_open_tty()
   {
      extern    struct TTYI_PORT_INIT_REC    Tt0_init;

      _Server_create( TTY, (struct INIT_REC *)&Tt0_init );
      _Selectinput( _Open( "TEXT_TERMINAL: +r", 0 ) );
      _Selectoutput( _Open( "TEXT_TERMINAL: +w", 0 ) );
   }
```

Master:harmony:example:srtest:src:main.c

```
task main()
   {
      extern    void        Create_and_open_tty();

      uint_32               n, limit;
      uint_32               child;
      struct STD_RQST  request;
      struct STD_RPLY  reply;

      Create_and_open_tty();

      _Put_string( "Parent creates child.\n" );
      _Flush();
      if( ! (child = _Create( CHILD )) )
         {
            _Put_string( "*** Creation of CHILD failed.\n" );
            _Flush();
        }

    limit = 100;
    for(;;)
        {
            for( n = 0; n < limit; ++n )
                {
                    request.MSG_SIZE = sizeof( request );
                    reply.MSG_SIZE = sizeof( reply );
                    if( ! _Send( (char *)&request, (char *)&reply, child ) )
                            _Put_string( "*** Send to child failed.\n" );
                    else
                            _Printf( "Child replied %i.\n", reply.RESULT );
                }
            _Put_string( "How many more message exchanges do you want? " );
            _Flush();
            limit = _Get_number();
            while( _Get() != '\n' ) ;
        }
};
```

Master:harmony:example:srtest:src:child.c

```
task Child()
    {
        uint_32    counter, requestor;
        struct STD_RQST   request;
        struct STD_RPLY   reply;

        counter = 0;

        for(;;)
            {
                request.MSG_SIZE = sizeof( request );
                requestor = _Receive( (char *)&request, 0 );

                reply.MSG_SIZE = sizeof( reply );
                reply.RESULT = counter++;
                _Reply( (char *)&reply, requestor );
            }
    };
```

APPENDIX B. A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM

This program is the same as the program of Appendix A, but the child task has been moved to run on processor 1. Note that the only difference in the program is that the task template for the child task has been moved from the _Template_list for processor 0 to the _Template_list for processor 1. Had the ttyinit record, main(), and Child() been compiled into a relocatable object library, the inclusion files defining the processor images could have been even simpler, essentially just referring to the files containing the three fundamental externals.

There are four inclusion files: for each processor there is one for externals and one for code. First, each inclusion file is compiled. Then, a binary image for each processor is linked using the compiled objects for that processor, together with appropriate version of the system libraries. Detailed steps necessary to build the library are described in Application Note 007 [5]. The file dummymain.c exists only to satisfy the Consulair linker. Other systems may not require it.

Master:harmony:example:srtest:inc:dy134cmac:double:externs0.c

```
#include "Master:harmony:sys:src:compiler:macc:compiler.h"
#include "Master:harmony:sys:src:sys.h"
#include "Master:harmony:sys:src:servers:tty:ttyinit.h"
#include "Master:harmony:sys:src:devices:mc68901:dvme134:m68901int.h"
#include "Master:harmony:example:srtest:src:templates.h"
#include "Master:harmony:example:srtest:src:dvme134:double:srtest0.c"        /* for double */
#include "Master:harmony:example:srtest:src:dvme134:ttyinit.c"
```

Master:harmony:example:srtest:inc:dy134cmac:double:externs1.c

```
#include    "Master:harmony:sys:src:compiler:macc:compiler.h"
#include    "Master:harmony:sys:src:sys.h"
#include    "Master:harmony:example:srtest:src:templates.h"
#include    "Master:harmony:example:srtest:src:dvme134:double:srtest1.c"     /* for double */
```

Master:harmony:example:srtest:inc:dy134cmac:double:code0.c

```
#include    "Master:harmony:sys:src:compiler:macc:compiler.h"
#include    "Master:harmony:sys:src:sys.h"
#include    "Master:harmony:sys:src:servers:tty:ttyinit.h"
#include    "Master:harmony:example:srtest:src:templates.h"
#include    "Master:harmony:example:srtest:src:main.c"
#include    "Master:harmony:example:srtest:src:default:createand.c"
```

Master:harmony:example:srtest:inc:dy134cmac:double:code1.c

```
#include    "Master:harmony:sys:src:compiler:macc:compiler.h"
#include    "Master:harmony:sys:src:sys.h"
#include    "Master:harmony:example:srtest:src:child.c"
#include    "Master:harmony:example:srtest:src:macc:dummymain.c"             /* for double */
```

The following are the included files which are in addition to those in Appendix A and are specific to the two-processor configuration:

Master:harmony:example:srtest:src:dvme134:double:srtest0.c

```
extern   task   main();
extern   task   _Directory();
extern   task   _Gossip();
extern   task   _Tty_server();
extern   task   _SPi_mc68901();
extern   task   _SPo_mc68901();
extern   void   _Mc68901_int();
uint_32  _Pnumber  =  0;

struct TASK_TEMPLATE _Template_list[] =
   {
      {  MAIN,            main,              1000,   7  },
      {  DIRECTORY,       _Directory,        1000,   7  },
      {  GOSSIP,          _Gossip,           2000,   5  },
      {  TTY,             _Tty_server,       2000,   5  },
      {  TTI,             _SPi_mc68901,      1000,   0  },
      {  TTO,             _SPo_mc68901,       800,   0  },
      {  0,               0,                    0,   0  }
   };

struct INT_PAIR _Interrupt_list[] =
   {
      {  5,      _Mc68901_int        },
      {  0,      0                   }
   };
```

Master:harmony:example:srtest:src:dvme134:double:srtest1.c

```
extern   task   Child();
extern   void   _Mc68901_int();

uint_32  _Pnumber  =  1;

struct TASK_TEMPLATE _Template_list[] =
   {
      {  CHILD,           Child,             1000,   6  },
      {  0,               0,                    0,   0  }
   };

struct INT_PAIR _Interrupt_list[] =
   {
      {  5,      _Mc68901_int        },
      {  0,      0                   }
   };
```

Master:harmony:example:srtest:src:macc:dummymain.c

```
task main()
   {
   }
```

APPENDIX C. DECLARATIONS AND SYSTEM FUNCTION CALLS

Compiler-related Definitions

Source: /harmony/sys/src/compiler/macc/compiler.h

The following typedef's allow us to minimize portability problems due to the various C compilers (even for the same processor) not agreeing on the sizes of int and short int. Note that these particular definitions are specific to the Consulair Mac C compiler and may be different for other compilers.

```
typedef unsigned char  uchar;    /* unsigned characters      */
typedef          short int_16;   /* 16-bit signed integers   */
typedef unsigned short uint_16;  /* 16-bit unsigned integers */
typedef          long  int_32;   /* 32-bit signed integers   */
typedef unsigned long  uint_32;  /* 32-bit unsigned integers */
```

Selected data structure definitions

Source: /harmony/sys/src/sys.h

Note that sys.h makes use of some basic types (and macros) that are compiler-specific. These types are found in the compiler.h file for each compiler. The appropriate compiler.h file must be included before sys.h in every Harmony inclusion file.

```
typedef    uchar boolean;
#define    TRUE      1
#define    FALSE     0

typedef    void      task;

struct INT_PAIR
  {
     uint_32  INT_PHYSICAL_ID;
     void     (*INT_HANDLER)();
  };

struct TASK_TEMPLATE
  {
     uint_32  GLOBAL_INDEX;
     task     (*ROOT)();
     uint_32  STACKSIZE;
     uint_32  PRIORITY;
  };

struct STD_RQST
  {
     uint_32  MSG_SIZE;
     int_16   MSG_TYPE;
  };

struct STD_RPLY
  {
     uint_32  MSG_SIZE;
     uint_32  RESULT;
  };
```

An INIT_REC structure must be at the start of any server initialization record. The size and format of the remainder of the initialization record is server dependent. The MSG_TYPE field in the INIT_REC structure is used by a server to distinguish the various types of initialization records it may receive.

```
struct INIT_REC                                   /* basic server initialization record */
   {
      struct     STD_RQST STANDARD;
      struct     INIT_REC    *IR_NEXT;            /* next record in initialization list */
   };

struct ALTNAME_INIT_REC
   {
      struct INIT_REC        INIT_REC_HDR;
      char                   ALTNAME[32];         /* alternate server name */
   };
```

Global task indices reserved for specific purposes:

```
#define  MAIN            1
#define  DIRECTORY       2
#define  GOSSIP          3
```

The remainder of this Appendix lists user-callable functions and, where appropriate, identifies supplied tasks.

Task Creation and Destruction

Task creation and destruction — supplied tasks

_Local_task_manager()	root function for a task which is responsible for creating and destroying tasks on its processor
_Explicit_scheduler()	root function for a scheduling task that uses a recurring interrupt to define epochs, and to facilitate other tasks being run after n epochs, as well as defining the order in which tasks waiting for an epoch will run

Application-callable functions

```
uint_32 _Create( task_index )
     uint_32   task_index;
```

_Create creates a task using the task template whose GLOBAL_INDEX is identical to the value of the *task_index* parameter. It returns a task ID if successful. If the creation is unsuccessful, 0 is returned and the reason for the failure is indicated by the task error code.

```
void     _Destroy( id )
     uint_32   id;
```

_Destroy destroys the task specified by the *id* parameter. All resources owned by the task are returned to the system. Its connections are closed. Any descendants of the task are also destroyed. The _Destroy primitive is sychronous, which means that the destruction is complete before the primitive returns.

```
void     _Suicide()
```

_Suicide destroys the task invoking the primitive.

```
uint_32  _Father_id()
```

_Father_id returns the ID of a task's creator.

```
uint_32  _My_id()
```

_My_id returns the task ID of the task invoking the primitive.

Message Passing

Application-callable functions

```
uint_32 _Send( rqst_msg, rply_msg, id )
        char        *rqst_msg,  *rply_msg;
        uint_32     id;
```

_Send sends a message specified by the rqst_msg parameter to the task specified by the *id* parameter and waits for a reply message in the *rply_msg* parameter. _Send returns *id* if successful, 0 otherwise.

```
uint_32 _Receive( rqst_msg, id )
        char        *rqst_msg;
        uint_32     id;
```

_Receive attempts to receive a message from the task specified by the *id* parameter, unless *id* is 0, in which case a message from any task will be accepted. The sender's message is copied into the *rqst_msg*. If the task specified by *id* has not called the _Send primitive to send a message to this receiver, or if *id* is 0 and no task has, the receiver will block. The _Receive primitive unblocks when a message for which it is awaiting becomes available. _Receive returns the identifier of the task from which the message was received.

```
uint_32 _Try_receive( rqst_msg, id )
        char        *rqst_msg;
        uint_32     id;
```

_Try_receive functions similarly to the _Receive primitive except that it is nonblocking. If the desired message is not available when the receiver calls _Try_receive then it will fail immediately and return 0. Otherwise, it functions in a manner indistinguishable from _Receive.

```
uint_32 _Reply( rply_msg, id )
        char        *rply_msg;
        uint_32     id;
```

_Reply transmits a reply message specified by the *rply_msg* parameter to the task specified by *id* from which a message has been received. _Reply returns *id* if successful, 0 otherwise.

Interrupts

Application-callable functions

```
void _Await_interrupt( interrupt_id, rply_msg )
        uint_16     interrupt_id;
        char        *rply_msg;
```

_Await_interrupt blocks the task invoking the primitive until the interrupt specified by *interrrupt_id* occurs. The message specified by *rply_msg* may be filled by the interrupt handler to record information about the interrupt.

```
void _Enable()
```

_Enable allows interrupts to occur.

```
void _Disable()
```

_Disable masks off interrupts.

Memory Management

Application-callable functions

```
char *_Getvec( size )
     uint_32    size;
```

_Getvec allocates *size* bytes of storage. If successful, it returns a pointer to the block of storage allocated to the task. Otherwise, it returns 0 and sets the task error code to indicate the reason for the failure. The task error code is set to OUT_OF_MEMORY if the request is not satisfied because a large enough block was not found.

```
void _Freevec( block )
     char       *block;
```

_Freevec releases the block of storage pointed to by the *block* parameter, which must have been allocated previously with the _Getvec primitive. The task error code will be set if *block* does not point to a block of storage owned by the task.

```
void _Trimvec( block, size )
     char       *block;
     uint_32    size;
```

_Trimvec reduces the block of storage specified by *block*, previously allocated by _Getvec, to *size* bytes in size, returning excess bytes to the storage pool.

```
uint_32 _Sizeof( block )
     char       *block;
```

_Sizeof returns the allocated size (in bytes) of *block* allocated with _Getvec.

```
void _Tune_Getvec( size )
     uint_32    size;
```

_Tune_Getvec tunes _Getvec by arranging for searches to start at the first idle block of the indicated size. This accelerates the search because it arranges to skip permanently allocated blocks at the beginning of the pool, and because it takes into account the statistical behaviour of first fit storage management that block sizes become graded in size, with blocks earlier in the pool being generally smaller.

Using Servers

Using servers — supplied tasks

_Directory()	root function for the directory task used to establish connections between clients and servers

Application-callable functions

```
uint_32 _Server_create( task_index, init_list )
     uint_32              task_index;
     struct   INIT_REC    *init_list;
```

_Server_create creates a server task, whose task template is specified by *task_index*. The *init_list* parameter is a list of initialization records. The size and format of the initialization records is server-dependent, but the struct INIT_REC must be at the start of each record. A server may indeed be provided with several types of initialization records which are distinguished by the MSG_TYPE field in the INIT_REC struct. The server knows the last initialization record has been provided when the IR_NEXT field is 0. NULL (0) may be passed to _Server_create if no initialization is needed for the server.

```
struct   UCB *_Open( directive, user_id )
     char       *directive;
     uint_16    user_id;
```

_Open returns a pointer to a UCB (user connection block) if a connection can be opened to a server with the specified directive. Because the UCB parameters are filled in atomically by the OPEN_RPLY message, the server can detect if the client has died while the open was in progress, and thus can avoid losing resources associated with that connection. Note that a server may get a CLOSE_REQUEST message for a connection which the server has been informed that the client died before open completed. Since _Open cannot know what I/O model the client and server are using, it cannot initialize whatever model dependent structures there may be in the UCB_XTRA. Instead, these must be initialized when the ucb is used. However, if UCB_XTRA is allocated, _Open does indicate that initialization is required by setting UCB_MAIN.STD_REPLY.RESULT to UCB_UNINITIALIZED. The *user_id* parameter can be used as a primitive protection mechanism, but for most servers it will be 0.

```
void _Close( ucb )
     struct      UCB  *ucb;
```

_Close closes the open connection of this task specified by *ucb*. Note that it does NOT flush the buffer first. Note also that a server may get two CLOSE_REQUEST messages, if the client dies while executing _Close. Additional functions needed for implementing servers are listed in the section "Implementing servers".

Stream I/O Functions

Application-callable functions

```
struct   UCB   *_SelectInput( ucb )
     struct   UCB  *ucb;
```

_Selectinput selects the stream specified by *ucb* to be the task's current input stream. It returns the task's previously selected input stream.

```
struct   UCB   *_Selectoutput( ucb )
     struct   UCB  *ucb;
```

_Selectoutput selects the stream specified by *ucb* to be the task's current output stream. It returns the task's previously selected output stream.

```
void _Seek( ucb, position, relative )
     struct  UCB  *ucb;
     int_32       position;
     boolean      relative;
```

_Seek changes the current position in the stream specified by *ucb*. If *relative* is TRUE then the position is moved *position* bytes relative to the current position. If *relative* is FALSE then the *position* is set *position* bytes relative to the start of the stream.

```
char _Get()
```

_Get returns the next character available on the selected input stream.

```
char _Put( byte )
     char    byte;
```

_Put writes *byte* to the selected output stream. It returns *byte.*

```
void _Flush()
```

_Flush forces the contents of the output buffer to be sent to the server.

```
void _Unget()
```

_Unget guarantees to unget the last character read, only if the previous operation on the stream was a _Get.

```
int_32  _Get_number()
```

_Get_number reads a number from the selected input stream. The number must be in the format 0xNNN (hexadecimal), 0NNN (octal), or NNN (decimal). _Get_number returns the value of the number read.

```
uint_32 _Get_string( s, n )
        char      *s;
        uint_32   n;
```

_Get_string reads a line of characters from the currently selected input stream into *s*, stopping when either a newline ('\n') is encountered or *n*-1 characters have been read. _Get_string appends a null character to *s* to terminate it, so it may write a maximum of *n* characters into *s*. _Get_string returns the number of characters read, excluding the null terminator.

```
void  _Put_decimal( n )
        int_32    n;
```

_Put_decimal writes *n* on the selected output stream in decimal.

```
void  _Put_hex( n )
        uint_32   n;
```

_Put_hex writes *n* on the selected output stream in hexadecimal format.

```
void  _Put_string( s )
        char      *s;
```

_Put_string writes the string *s* on the selected output stream.

```
boolean _Get_record( record, size )
        char      *record;
        uint_32   size;
```

_Get_record reads *size* bytes from the selected input stream into the storage pointed to by *record*. It returns TRUE if successful, FALSE otherwise.

```
boolean _Put_record( record, size )
        char      *record;
        uint_32   size;
```

_Put_record writes *size* bytes to the selected output stream from the storage pointed to by *record*. It returns TRUE if successful, FALSE otherwise.

```
void  _Printf( fmt, x DEFN_VARARGS )
        char    *fmt;
        int_32  x;
```

_Printf performs formatted output. The *fmt* parameter is a string which controls the output format. The *x* parameter is a marker indicating that a variable number of arguments may be present.

```
boolean _Only_nulls_left()
```

_Only_nulls_left is Harmony's nearest equivalent of end-of-file. It 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.

```
void    _Nullify()
```

_Nullify 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.

Library of Utilities

Application-callable functions

```
void _Copy( src, dest, num_bytes )
     char       *src, *dest;
     uint_32    num_bytes;
```

_Copy copies *num_bytes* bytes from the storage pointed to by *src* to the storage pointed to by *dest.*

```
void _Zero( ptr, num_bytes )
     char       *ptr;
     uint_32    num_bytes;
```

_Zero sets *num_bytes* bytes of storage pointed to by *ptr* to 0.

```
void _Sprintf( str, fmt, x DEFN_VARARGS )
     char    *str, *fmt;
     int_32  x;
```

_Sprintf functions similarly to _Printf but instead of writing output to a stream it places the output bytes in the storage pointed to by *str.*

```
int_32  _Sgetnum( s )
     char  *s;
```

_Sgetnum computes an integer value from an ASCII string, *s*, returning 0 if the string does not represent a valid number. We assume white space is stripped.

```
int_16  _Str_compare( s1, s2 )
     char *s1, *s2;
```

_Str_compare compares two null terminated strings, *s1* and *s2.* If the strings are equal, 0 is returned. If the *s1* is lexicographically less than *s2*, a negative value is returned. If *s1* is lexicographically greater than *s2*, a positive value is returned.

```
uint_32 _Str_length( s )
     char *s;
```

_Str_length returns the length of a null-terminated string, *s.* The '\0' terminator is not included in the length.

```
void _Str_copy( from, to, max )
     char       *from;
     char       *to;
     uint_16    max;
```

_Str_copy copies the null-terminated string pointed to by *from* into the string pointed to by *to.* The *max* parameter specifies the maximum number of bytes to copy including the null-terminator ('\0'), which is always appended to *to.*

```
boolean    _Str_equal( s1, s2 )
     char *s1, *s2;
```

_Str_equal compares two null-terminated strings for lexical equivalence. If either or both of the string pointers are NULL, the comparison evaluates as FALSE.

```
uint_16 _Swap_16( in_16 )
     uint_16   in_16;
```

_Swap_16 swaps the upper byte of *in_16* with the lower byte and returns the result.

```
uint_32 _Swap_32( in_32 )
     uint_32   in_32;
```

_Swap_32 reverses the order of bytes of *in_32* and returns the result.

Parsing Utilities

Application-callable functions

```
char **_Components( s )
     char *s;
```

_Components parses a string into pathname components. The delimiters are ':' and '/'. The ':' delimiter is part of the component; the '/' delimiter is not part of the component. The pathname is terminated by white space (i.e. blank, tab, or newline) or by the NULL byte. The structure returned is a vector of strings but is allocated as a single block.

```
char **_On_toggles( s )
     char *s;
```

_On_toggles parses a string to collect the "on" toggles. Characters before the first white space (blank, tab, or newline) are ignored. An on toggle is a '+' character following white space, followed by a keyword. A keyword is a character string terminated by white space or a null byte. Using on toggle keywords containing '=' characters is ill advised. The structure returned is a vector of keyword strings but is allocated as a single block.

```
char **_Off_toggles( s )
     char *s;
```

_Off_toggles parses a string to collect the "off" toggles. Characters before the first white space (blank, tab, or newline) are ignored. An off toggle is a '-' character following white space, followed by a keyword. A keyword is a character string terminated by white space or a NULL byte. Using off toggle keywords containing '=' characters is ill advised. The structure returned is a vector of keyword strings but is allocated as a single block.

```
struct  VALUE_PARAMETER*_Value_parameters( s )
     char *s;
```

_Value_parameters parses a string to collect the value parameters. A value parameter is defined as an expression in the form "keyword=value_string". A keyword is a string of characters terminated by the '=' character. A value string is a string characters terminated by white space or a null byte. Note that characters before the first white space (blank, tab, or newline) are ignored. The structure returned is a vector of VALUE_PARAMETER structs, but is allocated as a single block.

```
Int_32    _Evaluate( table, s )
     struct  EVALUATE_ENTRY  table[];
     char    *s;
```

_Evaluate searches a table for a given string. If found, the value associated with the string is returned. If the string is not found or a table not supplied, but the string is a valid ASCII representation of an integer number, the string is evaluated as such and the appropriate value returned. If none of the above, then 0 is returned. White space is presumed to have been removed previously.

```
boolean _Valid_number( s )
     char *s;
```

_Valid_number checks to see whether the string *s* holds a valid ASCII representation of an integer number.

```
void _Freeparse( parse )
     char **parse;
```

_Freeparse is provided to protect users from trying to free the strings of a parse individually, because these strings are allocated in a single block with the pointers to them.

Error Handling

Application-callable functions

```
boolean   _Stackoverflow()
```

_Stackoverflow returns TRUE if the task's stack is currently overflowed.

```
uint_32  _Set_task_error_code(  error_code  )
         uint_32     error_code;
```

_Set_task_error_code sets the task error code to *error_code*. It returns *error_code*.

```
uint_32 _Task_error_code()
```

_Task_error_code returns the task error code.

```
void  _Abort( s )
         char *s;
```

_Abort terminates the execution of a task, closes all connections and releases all allocated resources.

Debug

Debugger — supplied tasks

_Dbg_control()	root function for the debug control server that manages planted breakpoints
_Dbg_agent()	root function for the debug agent task which is instantiated on each target processor to perform operations on behalf of the debug control task.
_Dbg_shadow()	root function for the debug shadow task which provides timeouts for the debug control server for operations being performed by the debug agent tasks
_Gossip()	root function for the general logging and reporting task, used in particular to report processor faults

Application-callable functions

```
void  _Breakpoint( s )
         char *s;
```

_Breakpoint is a breakpoint compiled into the code, rather than being planted dynamically. The string *s* is printed by the debugger before entering the debugger's interactive mode.

```
void  _Log_gossip( s )
         char *s;
```

_Log_gossip logs the string *s* to the _Gossip task.

The following functions for manipulating use bits are actually implemented as macros:

```
void  _Set_use_bit(  n  )
         uint_32     n;
void  _Reset_use_bit(  n  )
         uint_32     n;

void  _Flip_use_bit(  n  )
         uint_32     n;

boolean _On_use_bit(  n  )
         uint_32     n;
```

Implementing Servers

Callable functions

```
uint_32 _Report_for_service( name, msg_type )
     char       *name;
     uint_16    msg_type;
```

_Report_for_service is called by a server to register itself with the _Directory task. The argument *name* is a string of length < 32, whose last character is a ':'. The *msg_type* parameter is one of REPORT_FOR_SERVICE or REPORT_SECONDARY_NAME. Any secondary names must be reported prior to the REPORT_FOR_SERVICE request, and only one REPORT_FOR_SERVICE may be done during a server task's lifetime. Violation of this protocol may result in a deadlock. On a REPORT_FOR_SERVICE call, _Report_for_service also informs the creator whether the server has successfully registered with _Directory by sending it a REPORT_COMPLETED or REPORT_UNSUCCESSFUL message.

```
struct   CON_TABLE  *_Alloc_connection_table( init_num_entries, grow_num_entries,
                                              max_num_entries, data_blk_size )
     uint_32    init_num_entries, grow_num_entries, max_num_entries;
     uint_32    data_blk_size;
```

_Alloc_connection_table allocates a connection table. The *init_num_entries* specifies the initial size of connection table to be allocated. The *grow_num_entries* argument specifies the amount by which the table is to be grown each time it is grown. If it is zero, the table will not be grown. The *max_num_entries* argument specifies a maximum size beyond which the table will not be grown. A zero value for *max_num_entries* is used to indicate that there is to be no limit on the maximum size of the table. If the *data_blk_size* is 0, the connection routines will leave the responsibility for managing connection data block storage to the server, and allow the server to specify a pointer to the connection data block in the call to _Get_connection. The returned value is a pointer to the connection table, or 0 if the table could not be allocated.

```
char *_Get_connection( table, client, new_connection, con_data_blk )
     struct   CON_TABLE  *table;
     uint_32             client,  *new_connection;
     char                *con_data_blk;
```

_Get_connection allocates new connections in a connection table. The table is grown if necessary. If the connection data block size for the table is 0, the server may pass a pointer to the connection data block in *con_data_blk*; in this case, the connection routines do not allocate/free the connection data blocks, but use this pointer instead. _Get_connection returns the server's connection data block for the newly allocated connection if the allocation is successful, or 0 if the operation was unsuccessful. This routine also returns the connection number of the new connection in *new_connection*.

```
char *_Lookup_connection( table, client, connection )
     struct   CON_TABLE  *table;
     uint_32             client, connection;
```

_Lookup_connection looks up the specified existing connection in a connection table. It verifies the client and connection and returns the server's connection data block for the specified connection if the lookup is successful, or 0 if the lookup was unsuccessful.

```
uint_32 _Free_connection( table, connection )
     struct   CON_TABLE  *table;
     uint_32             connection;
```

_Free_connection frees a connection entry, connection, in a connection table, *table*. The CON_DATA_BLK is freed, and the entry is made available for re-use. _Free_connection validates *connection*. It returns the connection number freed if it was valid, otherwise it returns 0.

APPENDIX D. USING SUPPLIED SERVERS

This appendix describes the supplied tasks and shows how selected servers are used.

Server Tasks

Examples of creating the particular server are under /harmony/example/. In each case the related basic structure definitions are under /harmony/sys/src/servers/. Once the server is created, an application task can use the callable functions, listed in this Appendix and which can be found in a directory /userlib/, under the appropriate directory (identified below).

Clock — supplied tasks

_Clock_server()	root function for an alarm clock task that replies after a specified interval or at a specified time, and allows a waiting task to be explicitly awakened, as well as facilitating reading and setting of time
_Clock_notifier()	root function for a task receiving interrupts from a timer

The example task that creates a clock server for the DY-4 DVME-134 system can be found in /harmony/example/clock/. The definition of the clock initialization record is in /harmony/sys/src/servers/clock/clockinit.h.

```
struct CLK_INIT_REC                              /* Clock server initialization record */
   {
      struct INIT_REC   CI_HDR;
      char              CI_NAME[32];               /* clock server name */
      uint_32           CI_DEV_GRP;                /* device group for this clock */
      uint_32           CI_GRP_MEMBER;             /* member number in device group */
      uint_32           CI_X_NOTIFIER;             /* global index of notifier */
      char              *CI_CLK_IO_ADDR;           /* (even) I/O address of clock chip */
      uint_32           CI_TIMER;                  /* which timer on clock chip */
      uint_32           CI_DEV_CODE;               /* 1st logical interrupt for clock */
      uint_16           CI_ALARM_RESOLUTION;       /* in milliseconds      */
   };

#define CIT_CLK_INIT_REC           1
```

A typical set of initialization values for the _Clock_server can be found under /harmony/example/clock/src/dvme134/. In particular, the actual values, assigned to the CLK_INIT_REC, as listed below, are in clockinit.c.

```
struct CLK_INIT_REC Clock_init =
   {
      sizeof( struct CLK_INIT_REC ),      /* MSG_SIZE */
      CIT_CLK_INIT_REC,                   /* MSG_TYPE */
      0,                                  /* no more initialization records */
      "CLOCK:",                           /* clock server name */
      0,                                  /* no device group */
      0,                                  /* therefore no group member number */
      CLOCK_NOTIFIER,                     /* global index for clock notifier */
      I_CAST(char *)MC68901_ADDR,         /* I/O address for clock chip */
      MFP_A_TIMER,                        /* Use timer 3 on the clock chip */
      TIMER_A,                            /* 1st logical interrupt for clk chip */
      50                                  /* alarm resolution in milliseconds */
   };
```

The following records are typical of those needed for a processor that is to have a clock server and are illustrated in the example in the file clock0.c.

Assuming the following declarations:

```
extern    task   _Clock_server();
extern    task   _Clock_notifier();
extern    task   _Mc68901_int();
```

these records appear in the _Template_list:

```
{  CLOCK,                _Clock_server,    1000,  6  },
{  CLOCK_NOTIFIER,       _Clock_notifier,   500,  0  },
```

and this record appears in the _Interrupt_list:

```
{  5,     _Mc68901_int        }
```

Application-callable functions

Functions that interact with the clock server are listed below. They are defined in /harmony/sys/src/servers/clock/userlib. The argument clock_ucb is returned by a prior call to _Open().

```
uint_16 _Getmonth( clock_ucb )
    struct UCB    *clock_ucb;
```

_Getmonth normally returns a 16 bit unsigned integer stored in the clock state. This integer is initialized to 0 when a clock server is created, but may change later using the _Setmonth function. If *clock_ucb* is 0, _Getmonth will return 0 and set the task error code to INVALID_CONNECTION. If _Getmonth is unable to communicate with the clock server corresponding to *clock_ucb*, the value returned is undefined.

```
uint_16 _Setmonth( clock_ucb, month )
    struct UCB    *clock_ucb;
    uint_16       month;
```

_Setmonth stores a 16 bit unsigned integer in the clock state, for subsequent retrieval using the _Getmonth function. This value will not change over time, except via more calls to the _Setmonth function. If *clock_ucb* is 0, then _Setmonth will return 0 and set the task error code to INVALID_CONNECTION. If _Setmonth is able to successfully modify the integer in the clock state (through the clock server), then it will return *month*. Otherwise, the value returned is undefined.

```
uint_32 _Gettime( clock_ucb )
    struct UCB    *clock_ucb;
```

_Gettime returns the current time in milliseconds if successful. If *clock_ucb* is 0, then 0 will be returned and the task error code will be set to INVALID_CONNECTION. If communication with the clock server is unsuccessful, then the value returned is undefined.

```
uint_32 _Settime( clock_ucb, millisec )
    struct UCB    *clock_ucb;
    uint_32       millisec;
```

_Settime sets the clock server's current time to *millisec* if successful. If *clock_ucb* is 0, then 0 will be returned and the task error code will be set to INVALID_CONNECTION. If communication with the clock server is unsuccessful, then the value returned is undefined. Otherwise, _Settime returns the current time after setting it.

```
uint_32 _Delay( clock_ucb, millisec )
    struct UCB    *clock_ucb;
    uint_32       millisec;
```

_Delay blocks the invoking task for *millisec* milliseconds. That is, *millisec* specifies a delta from the current time to the time at which the task wishes to awake. If *clock_ucb* is 0, then _Delay will return 0 and will set the task error code to INVALID_CONNECTION. If _Delay is successful in communicating with the clock server (and thus delaying the task), it will return 0, unless the task is awakened with the _Wakeup function, in which case the value returned with be the time at which the task was awakened by the clock server. Otherwise, the return value is undefined.

```
uint_32 _Sleep( clock_ucb, millisec )
     struct UCB    *clock_ucb;
     uint_32       millisec;
```

_Sleep blocks the invoking task until the current time is *millisec*. Note that changing the current time using the _Settime function can thus affect when the task will wake up. If *clock_ucb* is 0, then _Sleep will return 0 and set the task error code to INVALID_CONNECTION. If _Sleep is successful in communicating with the clock server (and thus putting the task to sleep), it will return 0, unless the task is awakened with the _Wakeup function, in which case the value returned with be the time at which the task was awakened by the clock server. Otherwise, the return value is undefined.

```
uint_32 _Wakeup( clock_ucb, wake_id )
     struct UCB    *clock_ucb;
     uint_32       wake_id;
```

_Wakeup attempts to wake up the task whose task identifier is *wake_id* if that task is currently blocked on the clock server as a result of invoking the _Delay or _Sleep functions. If *clock_ucb* is 0, _Wakeup will return 0 and set the task error code to INVALID_CONNECTION. If _Wakeup is unsuccessful in communicating with the clock server, the return value is undefined. If _Wakeup is successful in communicating with the clock server but the task specified by *wake_id* is not currently delaying or sleeping, the _Wakeup will return 0. If _Wakeup successfully wakes up the specified task, _Wakeup will return the time at which the task was awakened.

```
uint_32 _Time_from_vec( vec )
        uint_16    vec[5];
```

_Time_from_vec calculates a time in milliseconds from a time specified in days, hours, minutes, seconds, and milliseconds. The *vec parameter* is an array of 5 integers which are, in order from *vec[0]* to *vec[4]*: milliseconds, seconds, minutes, hours, and days. The time in milliseconds is returned.

```
void    _Time_to_vec( time, vec )
        uint_32    time;
        uint_16    vec[5];
```

_Time_to_vec performs the inverse transformation of _Time_from_vec, calculating the days, hours, minutes, seconds, and milliseconds of a time from a *time* in milliseconds. The result is stored in the *vec* array.

Null server — supplied task

_Null_server()	root function for a task supporting minimal server services

The definition of the null server initialization record is in /harmony/sys/src/servers/null/nullinit.h

```
struct NULL_INIT_REC
   {
      struct INIT_REC     NULLI_HDR;
      char                NULLI_NAME[32];        /* server name */
   };

#define    NULLSERVER_INIT    1
```

A typical set of initialization values for the _Null_server can be found under /harmony/example/timing/src. In particular, the actual values assigned to the NULL_INIT_REC, as listed below, are in nullinit.c.

```
struct NULL_INIT_REC      Null_server_init =
    {
        sizeof( struct NULL_INIT_REC ),      /* MSG_SIZE */
        NULLSERVER_INIT,                     /* MSG_TYPE */
        0,                                   /* no more initialization records */
        "NULL_SERVER:"                       /* null server name */
    };
```

The following records are typical of those needed for a processor that is to have a clock server and are illustrated in the example in the subdirectory atarist/tty in the file timing0.c.

Assuming the following declaration:

```
    extern    task    _Null_server();
```

this record appears in the _Template_list:

```
            {    NULL_SERVER,    _Null_server,    6000,  5 }
```

Application-callable functions

Applications access the services provided by this task through the stream I/O functions.

Tty server — supplied tasks

_Tty_server()	root function for the tty server task
_SPi_mc68901() or IC-specific alternate	root function for the input notifier task
_SPo_mc68901() or IC-specific alternate	root function for the output notifier task

The definition of the tty server initialization record is in /harmony/sys/src/servers/tty/ttyinit.h.

```
struct TTYI_PORT_INIT_REC
    {
        struct INIT_REC     TTYI_HDR;
        char                TTYI_NAME[32];          /* server name */
        uint_32             TT_DEV_GRP;             /* device group this port is a member of */
        uint_32             TT_GRP_MEMBER;          /* member number in device group */
        uint_32             TTI_DEV_CODE,           /* input logical interrupt */
                            TTI_INDEX,              /* input task template */
                            TTO_DEV_CODE,           /* output logical interrupt */
                            TTO_INDEX,              /* output task template */
                            TTETC_DEV_CODE,         /* etc logical interrupt */
                            TTETC_INDEX;            /* etc task template */
        char                *TTY_ADDR;              /* I/O address of tty port */
        uint_16             TTY_BAUD_RATE;          /* baud rate, ignored for ports */
                                                    /* without soft baud select */
    };

#define TTYIT_ALTNAME_INIT    1
#define TTYIT_PORT_INIT       2
```

A typical set of initialization values for the _Tty_server can be found under /harmony/example/timing/src/atarist/tty. In particular, the actual values assigned to the TTYI_PORT_INIT_REC, as listed below, are in ttyinit.c.

```
struct TTYI_PORT_INIT_REC Tt0_init    =
    {
        sizeof( struct TTYI_PORT_INIT_REC ),       /* MSG_SIZE */
        TTYIT_PORT_INIT,                           /* MSG_TYPE */
        I_CAST(struct INIT_REC *)&Default_init,    /* next initialization record */
        "TEXT_TERMINAL:",                          /* tty server name */
        0,                                         /* no device group */
        0,                                         /* therefore no group member number */
        RECV_0,                                    /* tti device code */
        TTI,                                       /* tti template index */
        XMIT_0,                                    /* tto device code */
        TTO,                                       /* tto template index */
        ETC_0,                                     /* ttetc device code not used */
        0,                                         /* ttetc template index not used */
        I_CAST(char *)MC68901_ADDR,                /* I/O address of tty port */
        19200                                      /* baud rate */
    };

struct ALTNAME_INIT_REC   Default_init  =
    {
        sizeof( struct ALTNAME_INIT_REC ),         /* MSG_SIZE */
        TTYIT_ALTNAME_INIT,                        /* MSG_TYPE */
        0,                                         /* no more initialization records */
        "DEFAULT:",                                /* tty server name */
    };
```

The following records are typical of those needed for a processor that is to have a clock server and are illustrated in the example in the file timing0.c.

Assuming the following declarations:

```
extern    task    _Tty_server();
extern    task    _SPi_mc68901();
extern    task    _SPo_mc68901();
```

these records appear in the _Template_list:

```
{   TEXT_TERMINAL,  _Tty_server,       2000, 5  },
{   TTI,                 _SPi_mc68901, 500,  0  },
{   TTO,                 _SPo_mc68901,500,  0  },
```

and this record appears in the _Interrupt_list:

```
{   6,       _Mc68901_int          }
```

Application-callable functions

Applications access the services provided by this task through the stream I/O functions.

UW server — supplied task

_UW_server() root function for the UW server task

The definition of the UW initialization record is in /harmony/sys/src/servers/uw/uwinit.h.

```
struct UWI_PORT_INIT_REC
  {
      struct INIT_REC     UWI_HDR;
      char                UWI_NAME[32];                /* server name */
      uint_32             UW_DEV_GRP;                  /* device group this port is a member of */
      uint_32             UW_GRP_MEMBER;               /* member number in device group */
      uint_32             UWI_DEV_CODE;                /* input logical interrupt */
      uint_32             UWI_INDEX;                   /* input task template */
      uint_32             UWO_DEV_CODE;                /* output logical interrupt */
      uint_32             UWO_INDEX;                   /* output task template */
      uint_32             TTETC_DEV_CODE;              /* etc logical interrupt */
      uint_32             TTETC_INDEX;                 /* etc task template */
      char                *UW_ADDR;                    /* I/O address of uw port */
      uint_16             UW_BAUD_RATE;                /* baud rate, ignored for ports */
                                                       /* without soft baud select */
  };

#define UWIT_ALTNAME_INIT    1
#define UWIT_PORT_INIT       2
```

A typical set of initialization values for the _UW_server can be found under /harmony/example/uwdemo/src/dvme134/demo/. In particular, the actual values, assigned to the UWI_PORT_INIT_REC, as listed below, are in uwinit.c

```
struct  UWI_PORT_INIT_REC     UW0_init =
  {
      sizeof( struct UWI_PORT_INIT_REC ),     /* MSG_SIZE */
      UWIT_PORT_INIT,                         /* MSG_TYPE */
      I_CAST(struct INIT_REC *)&Tt_init,      /* next initialization record */
      "UW0:",                                 /* UW server name */
      0,                                      /* no device group */
      0,                                      /* therefore no group member number */
      RECV_0,                                 /* SPi device code */
      UWI,                                    /* SPi template index */
      XMIT_0,                                 /* SPo device code */
      UWO,                                    /* SPo template index */
      ETC_0,                                  /* SPetc device code not used */
      0,                                      /* SPetc template index not used */
      I_CAST(char *)MC68901_ADDR,             /* I/O address of uw port */
      9600                                    /* baud rate */
  };
```

The following records are typical of those needed for the UW server and are illustrated in the example in the file uwdemo0.c.

Assuming the following declaration:

```
      extern    task    _UW_server();
```

this record appear in the _Template_list:

```
              {    UW,   _UW_server,     2000,     5   },
```

and this record appears in the _Interrupt_list:

```
            {   5, _Mc68901_int   },
```

Application-callable functions

Applications access the services provided by this task through the streamio functions.

EHVT server — supplied tasks

Echo Handshake Virtual Terminal

_Ehvt_server()	root function of EHVT server task
_Ehvtic()	root function of EHVT input consumer task
_Ehvtip_mc68901() or IC-specific alternate	root function of EHVT input producer task

The definition of the EHVT initialization record is in /harmony/sys/src/servers/ehvt/ehvtinit.h.

```
    struct EHVT_PORT_INIT_REC
      {
        struct INIT_REC EHVT_HDR;
        char          EHVT_NAME[32];              /* EHVT server name */
        uint_32       EHVT_DEV_GRP;               /* device group this port is a member of */
        uint_32       EHVT_GRP_MEMBER;            /* member number in device group */
        uint_32       EHVT_WAKEUP_DEV_CODE;       /* _Ehvtic wakeup logical int */
        uint_32       EHVTIC_INDEX;               /* EHVT input consumer task */
        uint_32       EHVTI_DEV_CODE;             /* input logical interrupt */
        uint_32       EHVTIP_INDEX;               /* EHVT input producer task */
        uint_32       EHVTO_DEV_CODE;             /* output logical interrupt */
        uint_32       EHVTO_INDEX;                /* output task template */
        uint_32       EHVTE_DEV_CODE;             /* etc logical interrupt */
        uint_32       EHVTE_INDEX;                /* etc task template */
        char          *EHVT_ADDR;                 /* I/O address of serial port */
        uint_32       EHVTI_BUF_SIZE;             /* input buffer size */
        uint_16       EHVT_BAUD_RATE;             /* baud rate, ignored for ports */
                                                  /*   without soft baud select */
    };

    #define EHVTIT_ALTNAME_INIT    1
    #define EHVTIT_PORT_INIT       2
```

A typical set of initialization values for the Ehvtic server can be found under /harmony/example/uwdemo/src/dvme134/. In particular, the actual values, assigned to the EHVT_PORT_INIT_REC, as listed below, are in ehvtinit.c

```
    struct EHVT_PORT_INIT_REC Ehvt1_init =
      {
        sizeof( struct EHVT_PORT_INIT_REC ),      /* MSG_SIZE */
        EHVTIT_PORT_INIT,                         /* MSG_TYPE */
        0,                                        /* no more initialization records */
        "EHVT:",                                  /* ehvt server name */
        0,                                        /* no device group */
        0,                                        /* therefore no group member number */
        EHVTIC_WAKEUP,                            /* _Ehvtic wakeup logical int */
        EHVTIC,                                   /* EHVT input consumer task index */
        RECV_0,                                   /* tti device code */
        EHVTIP,                                   /* EHVT input producer task index */
        XMIT_0,                                   /* tto device code */
        EHVTO,                                  /* output task template index */
        ETC_0,                                  /* ttetc device code not used */
        0,                                      /* ttetc template index not used */
        I_CAST(char *)MC68901_ADDR,             /* I/O address of serial port */
        2000,                                   /* input buffer size */
        9600                                    /* baud rate */
    };
```

Assuming the following declarations:

```
extern    task    _Ehvt_server();
extern    task    _Ehvtip_mc68901();
extern    task    _Ehvtic();
extern    task    _Sspo_mc68901();
```

these records appear in the _Template_list:

```
    {   EHVT,          _Ehvt_server,         1500, 5   },
    {   EHVTIP,        _Ehvtip_mc68901,      850,  0   },
    {   EHVTIC,        _Ehvtic,              750,  6   },
    {   EHVTO,         _Sspo_mc68901,        750,  0   },
```

and this record appears in the _Interrupt_list:

```
    {   5, _SMc68901_int }
```

Application-callable functions

Applications access the services provided by this task through the stream I/O functions.

File system — supplied tasks

The file system consists of three tasks:

_File_system()	root function for the file system server task
_File_device()	root function for the file device server task
_File_device_format()	root function for the file device format server task

The example task that creates a file system server (and the underlying file device server) for the Atari 520 ST system floppy disk, can be found in /harmony/example/fsys/. The definition of the file system initialization records is in /harmony/sys/src/servers/fsys/fsysinit.h.

```
struct FS_BASIC_INIT_REC              /* file system server basic initialization record */
    {
        struct INIT_REC       FSI_HDR;
        char                  FSI_NAME[32];             /* server name */
        char                  FSI_FDEV_NAME[32];        /* file device server name */
        uint_32               FSI_ROOT_LDRV_NO;         /* root logical drive */
        uint_16               FSI_OWNER_ROOT;           /* owner of "" */
        uchar                 FSI_O_PERMS_ROOT;         /* owner perms for "" */
        uchar                 FSI_W_PERMS_ROOT;         /* world perms for "" */
        uint_16               FSI_SE_CACHE_SIZE;        /* # substructure entries */
        uint_16               FSI_SB_CACHE_SIZE;        /* # substructure buffers */
    };
```

```
struct FS_LDRV_INIT_REC                  /* logical drive initialization record */
   {
      struct INIT_REC        FSI_HDR;
      uint_32                FSI_LDRV_NO;                /* logical drive number */
      uint_32                FSI_PDRV_NO;                /* physical drive number */
      uint_32                FSI_NUM_BLKS;               /* number of blocks */
      uint_32                FSI_BLK_SIZE;               /* multiple of sector size */
      uint_32                FSI_CYL_OFFSET;             /* cylinder offset: 1st blk */
      char                   FSI_LABEL[32];              /* label string */
   };

#define FSIT_BASIC_INIT_REC     1
#define FSIT_LDRV_INIT_REC      2
```

A typical set of initialization values for the file system can be found under /harmony/example/fsys/src/atarist/floppy/ in the file fsysinit.c. This file defines the configuration for the file system servers for an Atari ST floppy drive system. Fs0_init is the first record in the initialization record list passed to _Server_create for "FILESYS0:" The extern declaration is included to satisfy the compiler requirement for forward references.

```
extern struct FS_LDRV_INIT_REC Fs0_l0_init;

struct FS_BASIC_INIT_REC Fs0_init =
   {
      sizeof( struct FS_BASIC_INIT_REC ),         /* MSG_SIZE */
      FSIT_BASIC_INIT_REC,                        /* MSG_TYPE */
      I_CAST(struct INIT_REC *)&Fs0_l0_init,      /* next initialization record */
      "FILESYS0:",                                /* file system server name */
      "FILEDEV0:",                                /* corresponding fdev server name */
      0,                                          /* root logical drive */
      ROOT_OWNER,                                 /* owner of root ("*") */
      ROOT_O_PERMS,                               /* owner perms for root ("*") */
      ROOT_W_PERMS,                               /* world perms for root ("*") */
      40,                                         /* # substructure entries cached */
      2                                           /* # substructure buffers cached */
   };

struct  FS_LDRV_INIT_REC   Fs0_l0_init =
   {
      sizeof( struct FS_LDRV_INIT_REC ),          /* MSG_SIZE */
      FSIT_LDRV_INIT_REC,                         /* MSG_TYPE */
      0,                                          /* no more initialization records */
      0,                                          /* logical drive number */
      0,                                          /* physical drive number */
      585,                                        /* number of blocks */
      512,                                        /* blk size - multiple of sector size */
      0,                                          /* cylinder offset of first block */
      "Filesys0-ldrv0",                           /* label string */
   };

extern     struct  FS_LDRV_INIT_REC Fs1_l1_init;
extern     struct  FS_LDRV_INIT_REC Fs1_l2_init;
```

The following record is typical of that needed for creating a file system server. A file system server requires the creation of a file device server described also in this Appendix.

Assuming the following declaration:

```
        extern     task   _File_system_server();
```

this record appears in the _Template_list (in the file fsys0.c):

```
                {    FILE_SYSTEM,          _File_system_server,  3000, 6,  },
```

Application-callable functions

The functions that interact with the file system server are listed below. They are defined in /harmony/sys/src/servers/fsys/fsuserlib/.

File system control

```
char *_Copy_file( from_name, to_name, user_id )
      char          *from_name,  *to_name;
      uint_16       user_id;
```

_Copy_file copies a file for the specified user. Errors are reported on the currently selected output stream. _Copy_file returns *to_name* if successful, and 0 if not.

```
char *_Copy_tree( fsys_ucb, from_root, to_root, user_id )
      struct UCB     *fsys_ucb;
      char           *from_root,  *to_root;
      uint_16        user_id;
```

_Copy_tree copies an entire tree of nodes. The parameter *fsys_ucb* is the connection block for file system containing the source tree (from_root). Errors and status are reported on the currently selected output stream. _Copy_tree returns *to_root* if successful, and 0 if not.

```
char *_Get_current_node( fsys_ucb, pathname )
      struct UCB     *fsys_ucb;
      char           *pathname;
```

_Get_current_node appends the current node for the specified file system to *pathname*. The *pathname* string should be large enough to append up to 256 characters to it. _Get_current_node returns *pathname* if it was successful, and 0 if not.

```
char *_Get_access_perms( fsys_ucb, pathname, owner, o_perms, w_perms )
      struct UCB     *fsys_ucb;
      char           *pathname;
      uint_16        *owner;
      uchar          *o_perms;
      uchar          *w_perms;
```

_Get_access_perms returns in *owner, o_perms*, and *w_perms*, the owner and permissions for the specified node in the specified file system. _Get_access_perms returns *pathname* if it was successful, and 0 if not.

```
char *_Make_node( fsys_ucb, pathname, o_perms, w_perms   )
      struct UCB     *fsys_ucb;
      char           *pathname;
      uchar          o_perms,  w_perms;
```

_Make_node makes a new node in the file system tree. _Make_node returns *pathname* if successful, and 0 if not.

```
struct SUBSTRUCTURE_ENTRY *_Next_substructure_entry( substr_entry )
      struct  SUBSTRUCTURE_ENTRY  *substr_entry;
```

_Next_substructure_entry returns the next substructure entry.

```
struct RP_U_BITMAP_READ *_Read_bitmap( fsys_ucb, ldrv_no, bm_length, bm_buf )
    struct UCB                    *fsys_ucb;
    uint_32                       ldrv_no, bm_length;
    struct    RP_U_BITMAP_READ    *bm_buf;
```

_Read_bitmap returns in *bm_buf* the bitmap for the specified logical drive in the specified file system; *bm_length* is not the allocated size of *bm_buf*, but rather the size of the bitmap data area (*bm_buf*->UBR_BITMAP); _Read_bitmap returns *bm_buf* if successful, and 0 if not.

```
struct   SD *_Read_space_descriptor( fsys_ucb, pathname, sd )
    struct UCB *fsys_ucb;
    char *pathname;
    struct SD *sd;
```

_Read_space_descriptor returns in *sd* the space descriptor for the specified node in the specified file system. The space descriptor contains information about a node in the file system tree. _Read_space_descriptor returns *sd* if successful, and 0 if not.

```
struct   RP_U_SUBSTRUCTURE_READ *_Read_substructure( fsys_ucb, pathname,
                                                     sb_length, substr_buf )
    struct  UCB                        *fsys_ucb;
    char                               *pathname;
    uint_32                            sb_length;
    struct  RP_U_SUBSTRUCTURE_READ     *substr_buf;
```

_Read_substructure returns in *substr_buf* the substructure for the specified node in the specified file system. The routine _Next_substructure_entry can be used to walk the substructure buffer. The *sb_length* parameter is not the allocated size of *substr_buf*, but rather the size of the substructure data area (*substr_buf*->USSR_DATA). _Read_substructure returns *substr_buf* if successful, and 0 if not.

```
char *_Rename_node( fsys_ucb, pathname, new_node_name )
    struct UCB    *fsys_ucb;
    char          *pathname,  *new_node_name;
```

_Rename_node renames the specified node in the specified file system. The node's position in the file system tree cannot be changed, only its name (within its parent's substructure) can be changed. Thus *new_node_name* is just a component name of size < MAX_COMPONENT_SIZE, not a full pathname. _Rename_node returns *pathname* if successful, and 0 if not.

```
char *_Rm_node( fsys_ucb, pathname )
    struct UCB    *fsys_ucb;
    char          *pathname;
```

_Rm_node removes the specified node in the specified file system. _Rm_node returns *pathname* if successful, and 0 if not.

```
char *_Rm_tree( fsys_ucb, root )
    struct UCB    *fsys_ucb;
    char          *root;
```

_Rm_tree removes an entire tree of nodes. Errors and status are reported on the currently selected output stream. _Rm_tree returns *root* if successful, and 0 if not.

```
char *_Set_current_node( fsys_ucb, pathname )
    struct UCB    *fsys_ucb;
    char          *pathname;
```

_Set_current_node makes pathname the current node in the specified file system. _Set_current_node returns *pathname* if successful, and 0 if not.

```
char *_Set_access_perms( fsys_ucb, pathname, change_owner, new_owner,
                              o_on, o_off, w_on, w_off, owner, o_perms, w_perms )
      struct UCB     *fsys_ucb;
      char           *pathname,  change_owner;
      uint_16        new_owner,  *owner;
      uchar          o_on,  o_off,  w_on,  w_off,  *o_perms,  *w_perms;
```

_Set_access_perms changes the owner and/or access permissions for the specified node in the specified file system. This routine returns in *owner*, *o_perms*, and *w_perms*, the new owner and permissions for the node. _Set_access_perms returns *pathname* if it was successful, and 0 if not.

```
char *_Shrink_file( pathname, user_id )
      char         *pathname;
      uint_16      user_id;
```

_Shrink_file shrinks the file specified by *pathname* so that it occupies the minimum number of blocks on disk required to represent all of the information contained in the file. This routine writes the file size status or errors to the currently selected output stream. _Shrink_file returns *pathname* if successful, and 0 if not.

```
char *_Walk_tree( fsys_ucb, pathname, walk_data, traversal )
      struct UCB     *fsys_ucb;
      char           *pathname;
      char           *walk_data;
      uint_32        traversal;
```

_Walk_tree is used to walk a tree of nodes in the file system to obtain the full pathname of each node in the tree. On the first call to _Walk_tree the client passes 0 for walk_data, and the root of the tree to be walked in pathname. Traversal may be PRE_ORDER or POST_ORDER indicating the type of tree traversal to be done. On this and subsequent calls, Walk_tree returns a pointer to its internal data structure (*walk_data*) and copies the full pathname of a node of the tree into *pathname*. _Walk_tree returns NULL (0) for the pointer to its internal data when the walk is completed. On each call to _Walk_tree, the client should pass in *walk_data* the pointer returned on the previous call to _Walk_tree. Errors are reported on the currently selected output stream.

File system information

```
char *_File_size( pathname, user_id )
      char *pathname;
      uint_16  user_id;
```

_File_size writes the file size information about a file to the currently selected output stream. Errors are reported on the currently selected output stream.; _File_size returns *pathname* if successful, and 0 if not.

```
uint_32 _Fsys_space( fsys_ucb, init_list )
      struct UCB        *fsys_ucb;
      struct INIT_REC  *init_list;
```

_Fsys_space displays space usage statistics for the specified file system on the currently selected output stream; *init_list* is a pointer to the list of initialization records which describe the configuration of the specified file system; _Fsys_space returns 1 if it is successful, and 0 if not.

```
char *_List_file( pathname, user_id, break_pages )
      char         *pathname;
      uint_16      user_id;
      char         break_pages;
```

_List_file lists a text file on the currently selected output stream. If *break_pages* is set, _List_file pages (i.e. prompts for input from the currently selected input connection every 23 lines). Lines are assumed to be terminated by '\n', and the file is assumed to be terminated by '\0'. _List_file returns *pathname* if successful, and 0 if not.

```
char *_List_nodes( fsys_ucb, pathname, check_substr )
     struct UCB     *fsys_ucb;
     char           *pathname, check_substr;
```

_List_nodes lists the substructure for the specified node on the currently selected output stream. The list is formatted with several nodes per line. If *check_substr* is set, nodes which have substructure are tagged by a '+'; nodes whose substructure cannot be check are tagged by a '?'. This routine provides the equivalent of 'ls' or 'lc' in Unix systems. Errors are reported on the currently selected output stream. _List_nodes returns *pathname* if successful, and 0 if not.

```
char *_Put_tree( fsys_ucb, root )
     struct UCB     *fsys_ucb;
     char           *root;
```

_Put_tree lists the full pathnames for all the nodes in the specified file system tree on the currently selected output stream. The output is paged (i.e. the currently selected input stream is prompted after every 23 lines of output). Errors are reported on the currently selected output stream. _Put_tree returns *root* if successful, and 0 if not.

```
char *_Stat_node( fsys_ucb, pathname, user_id )
     struct UCB     *fsys_ucb;
     char           *pathname;
     uint_16        user_id;
```

_Stat_node writes the space descriptor information for a node to the currently selected output stream. _Stat_node returns *pathname* if successful, and 0 if not.

```
int_32  _Lsb_index( file_ucb )
     struct UCB     *file_ucb;
```

_Lsb_index returns the index of the last significant byte in the file specified by *file_ucb*. The "last significant byte" is defined as the last non-zero byte in the file. -1 is returned if the file contains no significant bytes.

File device

While applications do not communicate with the file device server, the application programmer needs to create the server. The following are the definitions for initialization record of the underlying file device server, as well as the specific initialization values for the initialization record.

```
struct FD_BASIC_INIT_REC                    /* basic initialization record */
   {
     struct INIT_REC     FDI_HDR;
     char                FDI_NAME[32];                /* server name */
     char                *FDI_CONTROLLER_ADDR;        /* I/O address of controller */
     uint_32             FDI_CMD_DONE_INT;            /* cmd done logical interrupt */
     uint_32             FDI_X_CMD_NOTIFIER;          /* global index */
     uint_32             FDI_X_SEEK_NOTIFIER;         /* global index */
     boolean             FDI_REPORT_DISK_ERR;         /* report disk errors in detail? */
   };

struct FD_PDRV_INIT_REC                     /* physical drive initialization record */
   {
     struct INIT_REC     FDI_HDR;
     uint_32             FDI_DRV_NO;                  /* physical drive number */
     uint_32             FDI_DEV_CODE;                /* seek done logical interrupt */
     uint_16             FDI_TYPE_DRV;                /* winchester/floppy */
     uint_16             FDI_UNIT;
     uint_16             FDI_HEADS;                   /* number of heads/tracks */
     uint_16             FDI_CYLINDERS;               /* number of cylinders */
```

```
        uint_16                 FDI_SECTORS_PER_TRACK;
        uint_16                 FDI_BYTES_PER_SECTOR;
        uint_16                 FDI_NUM_ALT_CYLS;          /* or floppy encoding */
    };
```

File system server initialization record types:

```
    #define FDIT_BASIC_INIT_REC     1
    #define FDIT_PDRV_INIT_REC      2
```

Specific initialization values for the file device example are under /harmony/example/fsys/src/ataristfloppy/ in the file fdevinit.c. This file defines the configuration for the file device server for an Atari ST floppy drive system.

This file defines the configuration for the _File_device_server and the _FD_Format_server for an Atari ST floppy drive system. Fd_init is the first record in the initialization record list passed to _Server_create for the _File_device_server. The initialization records are replied to the file device server during the server creation sequence (_Server_create). Note that only the first initialization record differs between the _File_device_server and the _FD_Format_server. The physical disk drives are the same for both servers.

The following is included to satisfy the compiler requirement for forward references.

```
extern struct FD_PDRV_INIT_REC Fd_d0_init;

    struct FD_BASIC_INIT_REC Fd_init =              /* for _File_device_server */
        {
            sizeof( struct FD_BASIC_INIT_REC ),     /* MSG_SIZE */
            FDIT_BASIC_INIT_REC,                    /* MSG_TYPE */
            I_CAST(struct INIT_REC *)&Fd_d0_init,   /* next initialization record */
            "FILEDEV0:",                            /* file device server name */
            I_CAST(char *)DMA_CTRL_ADDR,            /* I/O address of controller */
            DISK_CMD_DONE,                          /* cmd done logical interrupt */
            FD_CMD_NOTIFIER,                        /* cmd notifier global index */
            FD_SEEK_NOTIFIER,                       /* seek notifier global index */
            TRUE                                    /* report disk errors in detail? */
        };

    struct FD_PDRV_INIT_REC Fd_d0_init =        /* config for physical drive 0 */
        {
            sizeof( struct FD_PDRV_INIT_REC ),      /* MSG_SIZE */
            FDIT_PDRV_INIT_REC,                     /* MSG_TYPE */
            0,                                      /* no more initialization records */
            0,                                      /* physical drive number */
            DISK_SEEK_DONE,                         /* dev code (seek done logical int) */
            FLOPPY,                                 /* device type */
            0,                                      /* unit number */
            1,                                      /* number of heads */
            80,                                     /* number of cylinders */
            9,                                      /* number of sectors per track */
            512,                                    /* number of bytes per sector */
            DOUBLE_DENSITY                          /* floppy encoding */
        };
```

The following task template definitions and related definitions for the file device server are grouped together with those for the file system server, in the file fsys0.c.

Assuming the following declarations:

```
extern   task   _File_device_server();
extern   task   _FD_Cmd_notifier();
extern   task   _FD_Seek_notifier();
```

these records appear in the _Template_list:

```
{  FILE_DEVICE,        _File_device_server,  4000,  5,},
{  FD_CMD_NOTIFIER, _FD_Cmd_notifier, 500, 0,},
{  FD_SEEK_NOTIFIER, _FD_Seek_notifier, 500, 0,}
```

and this record appears in the _Interrupt_list:

```
{  0x11c, _Disk_int     }
```

The third server is the file device format server _FD_Format_server. For the sake of brevity, the details of this server are not given here. However, the Harmony source tree contains the files for the file device format server.

TCP/IP server — supplied tasks

_IP_Rx()	root function for ip receive agent task
_IP_Timer()	root function for ip timer task
_IP_Tx()	root function for ip transmit agent task
_TCP_server()	root function for tcp server task
_TCP_timer()	root function for tcp timer task

The task templates and initialization record definitions can be found in /harmony/sys/src/servers/tcpip/userlib/tcpipuser.h.

Application-callable functions

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

uint_32  _IP_send( ucb, buff, len, flags )
         struct   UCB          *ucb;
         struct   BUFF_MSGIO  *buff;
         uint_16                len;
         uint_16                flags;

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

uint_32  _Listen( ucb )
         struct   UCB  *ucb;

uint_32  _IP_addr_mask( net, addr_mask )
         uint_32    net;
         uint_32    addr_mask;

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

```
uint_32  _IP_host_addr(  net,  addr  )
         uint_32    net;
         uint_32    addr;

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

APPENDIX E. SOFTWARE ENGINEERING CONSIDERATIONS: SOURCE MANAGEMENT AND THE ORGANIZATION OF THE HARMONY SOURCE TREE

The problem of managing Harmony source is different from that addressed by most software development tools, such as Make, or SCCS, or RCS on Unix. Those tools are intended to manage a program as it evolves. The concern is that as the program changes with time two things may happen:

- parts of the program on which other parts depend may change, necessitating recompilation, redoing linkage editing, or other processing, in order to rebuild a correct and consistent version of the program. This is what Make addresses.

- conflicting requirements of production use and further development dictate a regime of a sequence of frozen versions of the program, termed releases, which production users can count on as being stable while developers continue to evolve the subsequent release. The frozen versions may be expected to have minor changes as errors are repaired, leading to sub-releases. If the program is big, and worked on by several developers, they may need to be protected from tripping over each other, for instance by preventing accidental and incompatible changes to the same module. These problems are what SCCS (Source Code Control System) and RCS (Revision Control System) address.

The software management problem facing Harmony is completely different, and although the two problems mentioned above do exist, they are minor by comparison. The problem of Harmony is that it is not one program, but a family of programs. The problem arises because Harmony is portable, configurable, and open, and because development is supported in many different computing environments. This has the following implications for source code:

- It is not known what computer system will be used for development of Harmony itself or application programs to be used in Harmony. The first consequence is that though the source for Harmony is organized in a tree, the form of pathnames identifying a particular unit cannot be stated in general! Another consequence is that the tools available are in general unknown, much less how the tools are invoked — unless they are tools provided with Harmony itself. Finally, the mechanism for executing command sequences on a given development system is unknown.

- It is not known what target microprocessor the Harmony system will be for. Care in using portable techniques means that this is surprisingly insignificant. Of course the functions written in assembler need redoing for a new microprocessor, but apart from that, the main things dependent on instruction set are the data structures describing the registers and other information saved on the stack when a task is interrupted, and for the same reason the information hand-crafted onto the stack when a task is created.

- It is not known what target microcomputer the target microprocessor is used in. This can affect what resources are available, what interrupt priority levels have been preallocated, how interprocessor interrupts are generated, what addresses are preallocated and what are available (what the memory map must be), what powerup sequence and other initialization is required, etc. Both assembly language and C code can be affected.

- It is not known what compiler and assembler will be used on the source. The compiler matters because there are minor differences in the languages accepted. For example, one such difference is that any compiler based on C as in Unix version 6 or earlier, does not permit the casting of external initializers to match the type of the variable being initialized — whereas compilers based on C as in Unix version 7 or later require the casting for type conformance. Another potential problem is caused by the manner in which structures are returned from functions. Many compilers generate nonreentrant code, which is catastrophic in a realtime system. With care, these problems can generally be avoided. The assembler matters rather more, as the syntax of different assemblers may be quite different. Register and subroutine linkage assumptions made by the compiler may require different assembly language code. Field offsets in structs must be calculated by hand for the assembly code, as assemblers rarely interpret C header files. Most annoying, the external symbols generated by C compilers rarely are exactly the identifiers used in C programs, so the assembly code must be changed to generate the appropriate symbols itself.

- The pathetic linkers and library editors used in the industry often are incapable of resolving backward references to modules occurring earlier in a library but not already forced to be loaded. This means that a unit of compilation bigger than a function must be used, so all such references can occur within a single linkage module.

- Limits on table sizes in various compilers mean that the unit of compilation must be limited to be smaller than the total source. In particular, many compilers will not accept all the header files as input in a single compilation, because the number of structs and unions, the number of members of structs and unions, or the number of defines, exceeds the maximum for which table space exists.

- The intended market of board level computers means that not only is the specific configuration unknown, e.g., device addresses, but the actual devices can differ in functionality and interface. A Motorola 6850 ACIA is vaguely equivalent to a Signetics 2681 DUART or an Intel 8251A USART, but the detail differences do affect the code for a terminal server. Similarly, the differences among timer chips or alarm clock chips do affect the code for the clock server. Peripherals implemented as separate boards, such as disk controllers, also have differences that are not transparent to the code for servers, such as the file device server, and might be used with any microcomputer implemented for the same bus. However, for a complex server, most of the code of the server is independent of these differences, and should be common across versions.

- Since many realtime systems are dedicated or highly specialized, Harmony is both configurable and open, in that any part of the supplied software not needed can be omitted, and customized servers required for the application can readily be added, whether they are modifications of supplied servers or something wholly new. Different modes of system use must be supported also, from programs downloaded into target systems for execution, to programs linked with Harmony then burned in ROM, to programs linking to a silicon operating system version of Harmony itself supplied in ROM.

- Many application programs must have access to header files from Harmony itself, i.e., data structure definitions and manifest constant values. This is true both of programs intended to run with Harmony on the target environment, and tools intended to run on whatever development environment is used to produce Harmony programs. For the latter, problems such as byte-swapping must be kept in mind.

These sorts of issues call for a new kind of source management. The solution used with Harmony evolved from the solution used with Thoth to a similar (but simpler) problem. The Harmony source management strategy is to regard the Harmony code tree as a database of source code, from which appropriate items are extracted to make the version of interest at the moment. The granularity of the database is that one function is stored in each file, although there are also files containing logically grouped external declarations, and logically grouped sets of `#define`, `struct` and `union` definitions. The extraction is done by files containing only `#include` statements. Because the extraction is for a specific version, where the file naming convention for the specific development host are known, file pathnames can be expressed in the appropriate format. An *inclusion file*, as these are known, defines a compilation unit which should be compiled together so the resulting object module can be stored in a library without the ordering concerns mentioned above. The source explicitly makes no use of the `#ifdef` conditional compilation mechanism, first because experience shows that if many alternate versions of code are combined this way the code becomes unreadable, and second because finding code with version differences then requires searching the complete text of the code, rather than just identifying files. The tree structure of the file system is heavily used in organizing the database.

The first branching of the Harmony code directory identifies major categories of items. These include the system itself (/sys/),[1] the supplied tools (/tools/), example programs (/example/), documentation (/doc/), and support for ROM (/bootrom/). The /tools/ and /example/ subdirectories are divided at the next level into specific tools and specific examples. Documentation for all the tools is grouped under the directory /doc/tools/. Command files for VMS and for A/UX are in directory /scripts/ under /tools/.

The source code directory (/src/) may contain files of source code directly. Or, if the program or library is made up of major configurable abstractions, it will be divided into subdirectories corresponding to these abstractions. Only files common to all abstractions (like certain header files) will appear at the top level. Abstraction subdirectories themselves may have subdirectories if the abstraction subdivides naturally into sub-abstractions, as /sys/src/servers/ does. The source code directly under /src/ or directly under an abstraction such as /sys/src/connect/ is only that code in C which is identical for all versions. If some attribute dictates a special variant, another level of subdirectory named for that attribute is interposed above the source code. Clearly the contents below different variant-discriminating subdirectories are parallel to each other. Often, there is a single version of the system that requires a small piece of code to be different from the other versions. In these cases, we create a variant directory /default/, which applies to the majority and a specific variant directory for the exception. Similarly, when the code is written in assembler instead of C, a subdirectory is interposed naming the specific assembler. This is not completely satisfactory, as it implies an assembler is used with only one compiler. Every variant subdirectory contains a file variant.doc describing what characterizes the need for that variant.

For a specific program or library, such as /harmony/example/srtest/ or /harmony/sys/, the first directory level distinguishes the source code database (/src/) from the directories of inclusion files (/inc/), for compiling various versions of this entity.[2] Every version has a corresponding directory (with possible subdirectories) of inclusion files, the structure in general paralleling the structure of the source directory (described above). The following versions are supplied with release 3.0:

Directory name	Target board	Microprocessor	Host	Compiler
dy134mpw2	DY-4 DVME-134	MC68020	Macintosh	Apple MPW C 2.0
dy134cmac	DY-4 DVME-134	MC68020	Macintosh	Consulair Mac C
dy134aux	DY-4 DVME-134	MC68020	Macintosh	A/UX C
dy134wvms	DY-4 DVME-134	MC68020	VAX/VMS	Whitesmiths' C
iovcmac	Io Inc. V68/32	MC68020	Macintosh	Consulair Mac C
atstcmac	Atari 520/1040ST	MC68000	Macintosh	Consulair Mac C
atstmpw2	Atari 520/1040ST	MC68000	Macintosh	Apple MPW C 2.0
ch00cmac	Omnibyte OB68K1A	MC68000	Macintosh	Consulair Mac C
ch00wvms	Omnibyte OB68K1A	MC68000	VAX/VMS	Whitesmiths' C
ch10wvms	Omnibyte OB68K1A	MC68010	VAX/VMS	Whitesmiths' C

An inclusion directory should contain a file version.doc describing exactly what version this directory corresponds to, i.e., what variants are included. Under an inclusion directory, there will be subdirectories corresponding to the abstraction subdirectories (if any) under the corresponding source directory.

[1] The "/token/" syntax is intended to be generic and is not that of Unix. The leading / does not imply the root.

[2] We use the term *variant* in the source branch of the tree and the term *version* on the inclusion side of the tree.

Primarily, the files under the inclusion directory are inclusion files, to be compiled. An inclusion file can also be used to generate listings, since the listing tool works on inclusion files.

Our software management scheme is based on layers or parallel trees. It is assumed that the developer is working with at least three parallel trees. On the Macintosh these trees are mapped to three logical volumes. The configuration in our laboratory for Harmony development uses the following logical volumes:

Master
: read-only copy of Harmony source

Working
: updated sources to be integrated into Master

Derived
: temporary files, temporary inclusion files that point to sources in Working, intermediate files produced by the compiler, linker and tools

Our normal working procedure is for each user to have a read-only copy of the master source code. Any Harmony system developer that needs to make a change to a file on the Master places the changed (or new) file in the corresponding directory in a sparse tree on his Working volume. Deletions are marked in the Working tree by special null files. In order to modify and to recompile a particular part of Harmony the following steps are taken. The appropriate inclusion directories are copied from the Master volume (or from Working in the case of altered inclusion files) to the corresponding node on the Derived volume. Because the pathnames in the inclusion file are absolute, each pathname in the inclusion file on the Derived volume still references all the source files in the Master volume. Then, to activate the revised files that have been modified and deposited in Working, specific pathnames are changed to point to these revised copies. The inclusion files on Derived volume are then compiled with the resulting object modules being left in the inclusion directory of the file that was compiled. Likewise, when the object modules are combined into libraries and into executable program images, all the generated files are left on the Derived volume. Similar procedures are followed by Harmony application programmers, who may introduce their own layers between Master and Derived.

APPENDIX F. DEVELOPMENT SYSTEM USED AT NRC

One of many characteristics that distinguish personal computers from workstations is the fact that a working system based on a personal computer is made up of many third party software products. These products include compilers, editors, utilities and even modules that augment the original operating system. There are several choices for each component. The exercise of selecting a consistent set is non-trivial. The configuration of a Macintosh system for Harmony development that is given below represents one set of choices. There is no intention to imply that the chosen products are the best; merely that they have worked well in our laboratory. Also, many of the choices were made in 1986, when the Macintosh was selected, other choices may be more relevant today. Our system is evolving in step with the technology.

The following shows the range of systems used in our laboratory.

- Apple Macintosh Plus personal computer, 1 MB and 20 MB disk (minimum configuration)

- Macintosh SE with Radius full page display, 2.5 MB, 60 MB disk (typical configuration)

- Macintosh II or IIx with 4 MB, 40 - 80 MB disk (alternate typical configuration)
  (Development under A/UX requires an upgraded Macintosh II or a IIx.)

All the development machines in our laboratory are networked using LocalTalk and Phonenet, and are running a file- or volume-sharing package, MacServe, from Infosphere. MacServe also offers a facility for partitioning the disk into logical volumes. Harmony source management is based on a layered structure which is well supported by the use of logical volumes. Other groups are using TOPS as the networking software. In that case the volume partitioning is achieved with one of the volume partitioning utilities. Elsewhere, even if there is only one user, more than one machine will likely be used. One machine typically would be with the realtime target, while another might be in the office. Consequently, it is convenient to use either MacServe or TOPS.

Consulair Mac C v.5.0 compiler is being used at present. Migration to the Apple MPW system and its C compiler is in progress and is expected to be completed later in 1989. A preliminary version of the port to MPW 2.0 for the Atari evaluation system is included in Release 3.0 of Harmony.

Source editing is done mainly using Paragon Concepts QUED/M programmer's editor with macros for navigating in parallel trees on different volumes. However, there is a distressing trend in some development systems to enforce the use of the built-in editor, making it difficult to use an alternative editor, such as QUED/M, which is often more powerful or more friendly than the built-in alternative.

InTalk virtual terminal from Palantir Software is being used because it has a macro facility, and also because it supports XMODEM protocol, which is used for downloading to a target system. There are several alternatives to InTalk available. Also, Apple MacTerminal is sometimes useful because of its faithful emulation of the DEC VT100.

Other useful tools include MacTree from SRT Software, useful for displaying and managing large file trees, particularly during the integration phase and HFS Navigator which modifies the behaviour of the file access dialog box in the Macintosh operating system, so as to facilitate rapid switch of the current directory.

Target systems in use in our laboratory, for which there are versions included in Release 3.0 of Harmony, are listed in Appendix E.