Abstract

2B is an operating system based on BTRON2 specification which supports multimedia and distributed processing. 2B follows the basic design philosophy of BTRON2 and has a hierarchical structure. The hierarchy of 2B can be decomposed into three major layers: inner kernel, outer kernel, and shell. Each layer in the hierarchy consists of modules. The design policy of 2B, the internal structure of layers, and the implementation of 2B for TRON specification CPU are described.

1 Introduction

2B is the first implementation of the operating system (OS) based on BTRON2 specification. The BTRON2 specification for an operating system [1, 2] is a successor of BTRON1 specification [3]. The BTRON2 specification is defined by redesigning the system to improve uniformity, transparency, extensibility, security, reliability, and maintainability. It allows operating system designers to take advantage of abundant hardware resources where they are available. The BTRON2 specification has higher level of abstraction than the BTRON1 specification. However, care was taken not to widen the semantic gap between hardware resources and the abstraction in the specification. We believe an efficient implementation is possible in a real world setting.

As in the BTRON1 specification, the operating system is divided into two layers in the BTRON2 specification: kernel and shell. In the BTRON2 specification, kernel is further divided into two sublayers: outer kernel and inner kernel. The inner kernel provides basic resource management functions and intertask communication functions. Recent operating systems tend to be composed of small nucleus which has fundamental functionalities and processes which run in user mode [4, 5, 6, 7]. The idea of composition of those operating systems is the same as BTRON2.

Guidelines in the overview of BTRON2 [8] suggest that a subset of ITRON2 extended specification [9] be used as the inner kernel. The outer kernel provides the BTRON2 kernel program interface by utilizing the inner kernel functions. The outer kernel basically consists of independent managers for each resource of a different type. The shell provides complex and high-level functions using the outer kernel functions. The shell consists of managers and servers. The system architecture that follows the BTRON2 specification is shown in Figure 1.

Figure 1 Structure of BTRON2

A comparative chart of function levels of an operating system based on the BTRON2 specification and those of UNIX* operating system is shown in Figure 2. The inner kernel of the BTRON2 specification OS has lower functionality than UNIX kernel. The outer kernel of the BTRON2 specification OS has higher functionality than UNIX kernel in turn.

We don't explain the BTRON2 specification itself any further. For the specification and its survey, please refer to [1, 2, 8].

* UNIX is a registered trademark of UNIX System Laboratories Inc. GMICRO is a trademark GMICRO Group for the TRON specification microprocessors. Intel is a registered trademark of Intel Corporation.
Personal Media Corporation planned to develop a BTRON specification OS mainly for TRON specification microprocessors. Now it is called 2B. Some TRON specification microprocessors do not have a memory management unit (MMU). Hence we could not assume an MMU while designing 2B. We planned to develop an operating system for a wide range of hardware configurations: from small handy devices using a CPU without an MMU to workstations using CPUs with MMUs. We designed 2B [10] basically according to the guidelines on the structure of operating systems provided in the overview of BTRON2. We describes the objectives and design policy of 2B. Some design decisions and internal structure of 2B are then discussed. After that, we report the implementation of 2B.

![Comparison between BTRON2 and UNIX](image)

Figure 2 Comparison between BTRON2 and UNIX

## 2 Objectives and Design Policy of 2B

### 2.1 Objectives

In designing 2B, we assigned the following objectives and tried to meet them.

**Distributed Processing:** We expect that multi-processor systems will be used for improving throughput in many applications. The improvement in the fabrication technology of integrated circuits will make it possible to incorporate multiple CPUs in one chip. Hence, use of multiprocessing systems will become more popular than it is today. Moreover, the network technology will make it possible to connect more systems than it is possible today. One aim of ours is to provide distributed processing based on these hardware technologies, be it tightly coupled or loosely coupled systems.

**Improved Extensibility:** It is important to incorporate new devices, such as multimedia devices, and other new virtual resources. The multimedia capability available today is mainly for simple device control and the superimposing of video images on the screen. However, this is clearly not enough. We need to have extensibility at the core of the system.

**Improved Portability:** We need to adapt to different hardware configurations easily. It is important that we can port the system to resource-poor systems and to resource-rich systems in an appropriate manner. One objective of the design is to improve portability. A system with improved portability makes it possible to limit (and remove) the functionality and to reduce needed resources on resource-poor specialized machines where we use only a limited set of known applications. A truly portable system should make it possible to change the implementation method to fit the environment in such cases.

**Improved Maintainability:** Operating systems have a long life cycle. It is inevitable that modifications are made, as suggested in discussing extensibility and portability, according to the changes of hardware environment and user expectations. Operating systems based on the BTRON2 specification is not as large as the operating systems for general-purpose mainframe computers, but still 2B is a complex system, and one aim is to make the maintenance easy.

**High Performance:** 2B is meant for human-machine interaction. It must provide a comfortable user interaction with reasonably good response accordingly. We have tried to make sure that the operating system is not using up computation resources, that enough computing power is left for the human-machine interaction, and that critical regions are minimized to quicken system response.

**Reasonable Size:** Usually, the lower is the functionality, the lower is the amount of required resources. If the operating system supports only low-level functionality, more computing resources are needed by the application that must offer a certain expected level of functionality at the end user level. In a multitasking system, if the level of functions provided by an operating system is high and multiple applications share and use the operating system functions, the total amount of computing resources required in the system would be low. Thus, in a multitasking system, we can't simply argue in favor of an operating system that requires lower amount of resources. We have tried to match the...
resource requirements (data, code, etc.) of the operating system to those of application programs.

2.2 Design Policy

We established guiding principles of internal design of 2B as follows.

Structured and Modular Programming: It is common wisdom to adopt structured and modular programming to improve maintainability. We tried to divide the system into components of manageable size, and to clarify the relation between these components. Hiding of the internal design of one component from the others was also a design principle. Structured and modular programming was generally adopted to improve maintainability, portability and extensibility.

Simplicity: We suspect that failures of large software projects such as operating system construction are often caused by the unnecessary (or untimely) complexity, for example, caused by early attempts for optimization. In order to avoid such complexity, unless there is a clear reason, simplicity is favored when we had to make choices. We took care so that we didn't divide the system into unnecessarily a large number of components since this large number would become the unnecessary complexity at the later stage of the project.

Parallelism: Preferred use of parallelism has forced us to adopt some coding style. In order to take advantage of parallelism, we should decrease the number of critical regions, and should avoid centralized control. We should write programs so that tasks can run concurrently, and we should not spare the use of tasks. We decided to place more importance on simplicity, timeliness and the use of parallelism on the future multiprocessing hardware than on the obscure saving of task switching time.

Clarification of Roles of Libraries and Managers: We decided to implement managers as independent processes basically to enhance modularity. Managers are not simple libraries.

It is important to clarify the roles played by libraries and managers. The simple sharing of code and constant data should be done by libraries. The sharing of data resources that requires mutual exclusion, synchronization and scheduling should be done by managers. Following these guidelines, managers control data resources that require exclusive access and synchronization, and libraries are used to share unmodified code or static (constant) data.

Procedural Interface: To facilitate coding, procedural call style interfaces were used where possible. We have adopted a remote procedure call as a communication primitive between processes in writing programs. By adopting a remote procedural call interface for invoking remote services instead of an explicit message passing interface, it is now easy to link directly (without the remote procedural call mechanism) different managers into one program to facilitate debugging.

3 Design of 2B

Here, we discuss some design decisions which are not discussed in the guidelines given in the BTRON2 overview.

3.1 Internal Structure of the Access Key

An access key is a handle to manipulate a real object, is obtained through an open basic operation and is 32 bit data (a real object is the abstraction of a system entity [11], and the open basic operation is listed in Table 1 later). We designed its structure as shown in Figure 3.

![Access Key Structure](image)

Figure 3 Access Key Structure

Mgrnum holds a value identifying a manager, and keynum holds an appropriate value for the manager to handle the access key. XXXX can be used differently by each manager for its internal use. E is set to 1 to show that this is an access key. In the BTRON2 kernel programming interface, certain arguments can be either an access key or a pointer to a structure. Since a structure is allocated on even byte boundary with the C compiler conforming to the BTRON2 specification and the least significant bit (LSB) of a pointer to such a structure is 0, setting the LSB of an access key to 1 makes it possible to distinguish between an access key and a pointer to a structure.

3.2 Communication between Applications and Managers

We decided to use the rendezvous port of the inner kernel for communication between applications and managers. The rendezvous port is suited for a remote procedure call and the procedural interface we have adopted for programming. The rendezvous port also has forwarding facility which can be used effectively for forwarding of processing described later.

We didn’t use the message buffer function of the inner kernel for the communication since the number of system calls to implement a remote procedure call by the message buffer would be larger than that of using the
rendezvous port. The number of system calls in using the rendezvous port for a remote procedure call is three: cal_por for calling, acp_por for accepting the request, and rpl_por for returning a value (cal_por, acp_por and rpl_por are rendezvous port system calls). In the case of the message buffer, we need four system calls: snd_msg for calling, rcv_msg for accepting, snd_msg for sending a return value, and rcv_msg for receiving the return value (snd_msg and rcv_msg are message buffer system calls).

We have used the rendezvous port as the low-level communication mechanism. Programmers however do not have to deal with the rendezvous port directly because we have created a simple stub generator for the remote procedure call.

3.3 Dispatching Basic Operation Requests to Managers

We have a manager for each different type of real objects according to the guideline of BTRON2. We need to dispatch the operation requests to managers accordingly. Basic operations are listed in Table 1. There are 15 of such basic operations.

Dispatching to managers is done by
- looking at the real object pointer specified as an argument for the open operation, or
- looking at the access key specified as an argument for other operations.

We have introduced a manager called the name manager that can properly dispatch basic operation requests based on their argument information. The name manager is discussed in section 4.4.

Table 1 Basic Operations

<table>
<thead>
<tr>
<th>Type Of Operations</th>
<th>Meaning</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Object Operations</td>
<td>Open a Real Object</td>
<td>opn_obj</td>
</tr>
<tr>
<td></td>
<td>Close a Real Object</td>
<td>cls_obj</td>
</tr>
<tr>
<td></td>
<td>Copy a Real Object</td>
<td>cpy_obj</td>
</tr>
<tr>
<td>Record Operations</td>
<td>Get (read) Data from a Record</td>
<td>get_rec</td>
</tr>
<tr>
<td></td>
<td>Put (write) Data from a Record</td>
<td>put_rec</td>
</tr>
<tr>
<td></td>
<td>Make (create) a Record</td>
<td>mak_rec</td>
</tr>
<tr>
<td></td>
<td>Remove a Record</td>
<td>rmv_rec</td>
</tr>
<tr>
<td></td>
<td>Lock a Record</td>
<td>loc_rec</td>
</tr>
<tr>
<td></td>
<td>Search for Records</td>
<td>sch_rec</td>
</tr>
<tr>
<td></td>
<td>Map a Record</td>
<td>map_rec</td>
</tr>
<tr>
<td></td>
<td>Transcribe(copy or move) Records</td>
<td>trs_rec</td>
</tr>
<tr>
<td>Event Operations</td>
<td>Request to Raise an Event</td>
<td>ras_evt</td>
</tr>
<tr>
<td></td>
<td>Request to Notify an Event</td>
<td>ntf_evt</td>
</tr>
<tr>
<td>Access Key Operations</td>
<td>Duplicate an Access Key</td>
<td>dup_key</td>
</tr>
<tr>
<td></td>
<td>Change Attributes of an Access Key</td>
<td>chg_key</td>
</tr>
</tbody>
</table>

3.4 Interpretation of a Real Object Pointer during Open

In order to open a real object, a real object pointer must be interpreted to identify a target real object. A real object pointer is a combination of real object pointer elements shown in Table 2.

Table 2 Real Object Pointer Elements

<table>
<thead>
<tr>
<th>Access Key</th>
<th>Access Key Constant</th>
<th>Domain Containing Record</th>
<th>Real Object Containing Record</th>
<th>Domain ID</th>
<th>Media ID</th>
<th>Device ID</th>
<th>EntitySpecifier</th>
<th>Real ObjectSpecifier</th>
<th>Record NumberSpecifier</th>
<th>Link RecordSpecifier</th>
</tr>
</thead>
</table>

By concatenating record number specifiers (RNS) and link record specifiers (LRS), we can build a real object pointer that traverses the real object/virtual object network and identifies a real object.

Both of an RNS and an LRS will specify a link record in a real object, and the interpretation of an RNS and an LRS must follow the content of the record, which content is a real object pointer (Figure 4). In BTRON2, real objects that hold link records are not limited to
storage-type real objects (files) and the formats of some real object pointer elements (for example, domain ID, real object specifier) can be different among real objects. Hence, it is generally impossible to perform the complete interpretation of a real object pointer in one manager that handles a type of real objects.

It is possible to build a central manager that requests the interpretation to another manager and receives the result from the manager. However, that central manager will hold many foci of control during operations and can be a bottleneck. This is opposed to the use of parallelism, one of major principles of 2B design.

We adopted the following approach. The interpretation of real object pointers is performed within a manager as far as possible, and only when the interpretation can't be done further, it is passed to another manager. This approach is good for load balancing as well.

The problem is how to ask another manager for interpretation. A manager can call another manager when no further interpretation is possible. However, if this simple calling is used, when the open processing is finished after the interpretation, all managers that are called during the interpretation must be traversed back to return the final result to the originator.

Another way for asking the interpretation is to forward the processing when no further interpretation is possible in a local manager. When the open processing is finished, the original caller is directly called back disregarding the managers involved at intermediate steps.

Forwarding is attractive from the viewpoint of message transfer. Let us assume that asking for interpretation, and returning of the result can be contained in one message each, and the number of managers consulted is N. When we use calling, 2N message transfers are necessary. On the other hand, (N+1) messages transfers are necessary when we use forwarding. Figure 5 shows the case of two managers involved in forwarding the processing. In a network environment, the overhead of one message transfer itself can be large.

Owing to the background explained above, we have adopted the forwarding for the pointer interpretation request. In a stand-alone machine environment, this forwarding can be implemented by using the forwarding function of a rendezvous port.

4. Structure of 2B

The whole structure of 2B is shown in Figure 6. We now explain the internal structure of 2B.

4.1 Kernel

The inner kernel has functionalities of a subset of ITRON2 extended specification. The inner kernel consists of modules, each addressing specific functions: task management, task dependent synchronization, interrupt handling, exception management, memory pool management, timer management, system management, extended synchronization and communication, forced exceptions management, resource management support functions, timer handler functions, and debugging support.

The outer kernel of 2B consists of managers for real objects, the name manager which controls naming in general, and the portion manager which controls memory blocks that can be purged from primary storage and written to secondary storage. In the following, the managers for real objects and their internal processing flow are briefly described. After that, the name manager, and the portion manager are explained.

4.2 Outer Kernel Managers For Real Objects

Managers for real objects are classified as follows: process/task manager, storage manager, interprocess communication (IPC) manager, device manager, timer manager, and estate manager.

The process/task manager controls process real objects and task real objects. This consists of the process manager and the task manager.

The storage manager controls storage-type real objects and plays similar role played by a file system component in other operating systems. 2B currently provides the memory manager and the file manager as the storage manager. We can use either of them according to application requirements. The memory manager aims for high-speed, and provides volatile storage-type real objects in main memory. That manager is simple since it does not have to deal with secondary storage. The file manager provides non-volatile storage-type real objects on secondary storage devices such as disks.

The IPC manager is a generic name given to managers that control real objects used for interprocess or intertask communication. The queue manager, the rendezvous manager, the semaphore manager are examples of the IPC manager.
The device manager is a generic name given to managers that control various device real objects. Each device real object of a different type is controlled by a different device manager: the keyboard manager, the pointing device manager, the disk manager, the serial line manager, the printer manager, etc.

The timer manager controls timer real objects. They can be used to generate events after a given time passes.

The estate manager controls real objects that have the functionality of a window. The estate manager supports windows of any shape, having multiple views and capable of drawing from TAD (TRON Application Databus [11]) data directly.

4.3 Processing inside Outer Kernel Managers

Tasks that implement managers for real objects follow basically the processing pattern shown in Figure 7.
An outer kernel manager task waits for receiving a request at a rendezvous port. When a request is sent to the rendezvous port, the task receives and analyzes the request and then performs the processing. It replies a return value, and begins waiting for further requests.

In principle, one such task would be sufficient. However, if the individual processing takes long time, or if the task enters into a wait state during the processing, we can't take advantage of the inherent parallelism of requests coming from many sources, and there is possibility for deadlock. Hence we prepare multiple tasks internal to the manager so that multiple requests can be serviced within the manager.

Having different tasks for receiving, analyzing, and executing requests is also a good way for a proper scheduling. We have created separate tasks for executing requests in some managers. The rendezvous port is used for intertask communication within a manager. Forwarding facility of the rendezvous port is used to reduce the overhead of communication.

4.4 Name Manager

The name manager offers the following capabilities: registration of names, dispatching and forwarding of basic operation requests, and assignment of IDs to inner kernel resources.

Registration of Names: The name manager has under its control
- IDs for managers,
- Type names and subtype names for resources controlled by such managers, and
- IDs for communication in inner kernel (rendezvous port ID).

The name manager can assign a manager ID (number) to identify a manager. Outer kernel managers that control real objects are assigned a manager ID when they are installed into the system. Each outer kernel manager in turn registers the type name, and the subtype name of the resource that the manager controls, and it also registers ID for communication into a name manager database.

A manager number is used as a part of an access key stated earlier. A type name and a subtype name are used as a part of identification information of real objects controlled by such managers. The ID for communication is used by the name manager when it dispatches a basic operation request, or forwards such a request.

Dispatching and Forwarding of Basic Operation Requests: When a basic operation is called for, the request is first passed to the name manager. The name manager tries to find a manager responsible for the operation by looking at the access key or the real object pointer that is a part of the arguments of the basic operation request. The manager found by the name manager is asked to perform the requested operation.

In the case of an open operation, the responsible manager is determined by the first type name and subtype name that are contained in the real object pointer of the arguments. For other basic operations, the manager number in the access key of the arguments is used to decide which manager is responsible for an operation. When the responsible manager is identified, the request for processing the operation is sent to the manager via a registered ID for communication for that manager.

Figure 8 shows the flow of processing when an application or a shell invoked an operation.

As was discussed earlier, when each manager can't analyze the identification information any further during an open operation, the remaining processing request is forwarded. A forwarding request of the remaining processing is asked to the name manager. The name manager analyzes such request as an ordinary request for dispatch. If it can identify a responsible manager, the name manager forwards the request to the manager. In a stand-alone machine environment, the rendezvous of originating task is forwarded to the rendezvous port of the target manager.

Assignment of IDs to Inner Kernel Resources: The inner kernel doesn't have the facility to generate a unique ID for an inner kernel object and hence we need to assign an ID when we create an object. The object IDs
allowed in the inner kernel must be an integer within a contiguous range. Thus, when an outer kernel manager creates an inner kernel object, each outer kernel manager must assign a unique ID for itself. A mechanism agreed among outer kernel managers is needed to uniquely assign IDs.

We could conceivably give each manager a fixed range of IDs. However, if the method is used, we can't easily accommodate the situation where one manager requires more IDs than expected and runs out the ID range after a revision. Moreover, that pre-assignment method would possibly result in the waste of the ID space because some managers, which have been assigned the ID range, may not be installed into the system at run time.

Owing to the reasons above, we decided to avoid fixed pre-allocation of IDs to each manager. The name manager offers the required dynamic ID assignment facility. Each outer kernel manager tells the name manager what type of resources and how many such resources the outer kernel manager needs for management of real objects. After that, the manager is given the required number of IDs. The outer kernel manager uses only those assigned IDs for subsequent services.

4.5 Portion Manager

The portion manager is a memory management manager with the following features:
- Efficient memory management using blocks as unit
- Memory can be written to secondary storage and then purged.

The inner kernel has memory pool management function, but its unit is one byte and we deem it a little inefficient for management of blocks. Moreover, the inner kernel memory management function doesn't support memory management with purging and writing to secondary storage. Some managers, such as the storage manager, need such memory management function.

We implemented the portion manager to meet such needs. In a virtual memory system, the portion manager would be called the page manager. However, because the unit of allocation is not a page, we call the manager the portion manager to avoid confusion. The unit of allocation is a portion which is 512 bytes in the current configuration. Used/unused state and resident/purgable state of portions are managed with bitmaps. Those states can be manipulated efficiently through bit manipulation operations of TRON specification CPU. Currently, the portion manager supports writing to secondary storage as a way of purging memory, and reading from secondary memory as a way of loading into memory. By allocating a memory block with a secondary storage block number, we can load the data from the data block in the secondary storage, and when the memory is purged to make place for another purpose, the memory data is written to the secondary storage block automatically. The least recently used (LRU) algorithm is used as a replacement algorithm of purgable portions. Resident portions is regarded as being referenced.

5 Shell

Shell of 2B now consists of a BTRON1-compatible window server, a BTRON1-compatible real/virtual object server and a Kana-kanji translation server.

The BTRON1-compatible window server implements the windowing functions for programs together with BTRON1-compatible libraries. This window server uses the estate manager in the outer kernel. When an application calls BTRON1-compatible window library functions, the window server is called. Functions of the estate manager are invoked if necessary.

The BTRON1-compatible real/virtual object server supports the functionality of the BTRON1 specification.

The Kana-kanji translation server does what its name implies. It translates a Japanese kana character string to a Japanese kanji character string through user's help. It is an input server of a sort. If we want to introduce another type of input server such as a server to support hand-written character recognition from pen input, we can implement such a server as a shell server like this kana-kanji translation server.

The BTRON1-compatible libraries, which are linked into application programs, and outer kernel servers together provide compatibility with the BTRON1 specification and a compatible human-machine interface for applications.

6 Implementation

6.1 Hardware

2B has been implemented on an evaluation hardware platform [12] based on GMICRO series, GMICRO/100, GMICRO/200 and GMICRO/300, all of which are TRON specification microprocessors. The outline of the hardware platform is shown in Table 3.

<table>
<thead>
<tr>
<th>Table 3 Hardware Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td><strong>RAM</strong></td>
</tr>
<tr>
<td><strong>Hard Disk</strong></td>
</tr>
<tr>
<td><strong>Floppy Disk</strong></td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Optional I/O</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
6.2 Kernel

In order to achieve high performance, the inner kernel is programmed in assembly language. The total size of source files is about 12000 lines including comment lines. The total amount of object code and static data is about 60 kilobytes.

The C++ language is used to implement managers and libraries in the outer kernel. Structuring and modularization within managers was done using class facility of C++. The C language and an assembly language are used for some device driver codes. The C++ system that was used is a preprocessor system based on AT&T's cfront. For C, we used a proprietary one and GNU C compiler system for TRON specification CPU [13, 14]. We have used a set of original runtime libraries developed by us.

We think that amount of source codes could be reduced with the C++ language. If we use the C language, we had to write many similar codes or we had to use many complex and tricky macros. The C++ language helps us to program straightforwardly and alleviates unessential programing effort. We hope future maintenance does not need much effort.

Bugs of the C++ compiler annoyed us. Some proper source codes could not be compiled and some source codes were compiled into wrong codes. We had to rewrite programs to avoid compiler's bugs. The rewriting needs much effort. Extensions of the C++ language specification have also demanded rewriting. The extensions included incompatible modifications with the old version and the compiler.

We developed simple program generators to avoid monotonous repetitive programming and to improve maintainability. For coding remote procedure calls, we developed a stub generator that takes as input C++ class definitions and generates required stub routines. The C++ system we used during implementation did not support class definition with type arguments (parameterized class or template class). Hence we built a simple source generator that can take a C++ class definition and an implementation body as input, and then generates a modified class definition and body after substituting the "type arguments," thus we effectively could use the class definition with type arguments. This saved us a lot of repetitive rewriting of essentially the same code when we tried to adapt it to support different data types. We also developed a simple tool which inserts debug statements into sources. Those debug statements print a function name and arguments or a return value when the function is called and is left. This tool helped us to debug programs.

We have taken advantage of high level instructions of TRON specification CPU. We used such instructions in libraries that handle character strings, bit-strings, and lists(queues). Moreover, the drawing to bitmaps is done by bitmap manipulation instructions of TRON specification CPU. High level instructions of TRON specification CPU accelerate the operation of 2B.

In the current implementation, the queue manager, the rendezvous manager and the semaphore manager are linked into one executable image as the IPC manager. This is because they share large amount of the same code.

Furthermore, the IPC manager code is executed by an application task directly. An IPC manager task tends to be in a wait state during its operation. If we follow the design principle of providing as many tasks as are required, we probably need tasks that are as many as the tasks that can call the IPC manager. In order to reduce the number of manager tasks and the communication overhead, we let application tasks execute the IPC manager code without task switching. We can create independent tasks, if necessary. In the current stand-alone environment, neither the name manager nor the portion manager contain an independent task.

6.3 Shell, BTRON1-compatible Libraries and Applications

The shell and BTRON1-compatible libraries were implemented using C language. System applications of 2B include basic editors (basic text editor and basic figure editor). These application programs are ported from BTRON1. BTRON1-compatible libraries are used for porting.

Here we summarize some tips for porting an application for BTRON1-based systems to 2B.

For efficient data access, the BTRON1-compatible libraries now use 32 bits for a word as opposed to 16 bits in the BTRON1 specification. Accordingly, some programs require rewriting. In the case of basic editors, the change was more or less mechanical and was not a big job.

An endian of TRON specification CPU is different from that of Intel 80286 CPU on which BTRON1-based OS was implemented. This difference can cause problems for some programs. Basic editors already paid attention to portability issues caused by endians, and thereby porting was not difficult in this regard, too.

Compiler systems can cause problems as usual. So-called alignment tends to cause problems for us. If a BTRON1 application assumes a certain alignment property from a C compiler system, some rewriting may be necessary. In the case of basic editors, we have some alignment dependent parts in data structures that handle TAD data. However, we solved most of the problems by rewriting structure declarations to match the new compiler alignment requirement.
6.4 Implementation Summary and Future Plan

The inner kernel, the outer kernel, the shell, the BTRON1-compatible libraries, and application programs were decomposed into modules of manageable sizes. This division of the project into modules allowed us to proceed with small sub-projects concurrently. We intend to perform performance evaluation and plan to tune the implementation for higher performance and to augment the quality for reliability.

We have found some functions of outer kernel managers should be incorporated in the inner kernel. We will incorporate the inner kernel resource ID assignment function of the name manager into the inner kernel. We plan to enhance and generalize functions of the portion manager and then incorporate them into the inner kernel.

We have found there are purgable portions in some managers (for example, the estate manager) which do not perform disk I/O. It is desirable to purge the portions in a small system. Thus we plan to improve the portion manager so that we can define purge handlers and load handlers for portions. A purge handler is called when a portion should be purged, and performs the purging of the portion data. A load handler is called when a non-resident portion is accessed, and loads data in the portion. The current portion replacement algorithm is also found to degrade the performance in a system having small amount of memory. Large data I/O can purge important management information and can cause to reload the information frequently (like a thrashing). If we let the information be resident, the performance problem could be solved but usable memory would be restricted. We have been investigating to introduce a resident priority function in the portion manager so that outer kernel managers can control memory residency appropriately. Utilizing the enhanced inner kernel, we will implement a distributed virtual memory system.

Acknowledgements

We would like to acknowledge Prof. Sakamura's suggestions and encouragement during the development. We would like to thank those who have helped us in setting up the hardware and development environment, especially people in GMICRO group, and those who developed and provided us with a GNU compiler system for the TRON specification CPU at Sakamura laboratory at University of Tokyo, and finally to those who have been involved in the development of 2B for their continuing commitment.

References

[8] Sakamura, K., Overview of BTRON2 Kernel, Sakamura Laboratory, Department of Information Science, University of Tokyo, 1990 (In Japanese).