Extending the ITRON2 specification for MMU equipped processors

Tod K. Johnson, Chung-Lin Dai
R & D Department
Electronic Device Group
Mitsubishi Electronics America

Abstract

Support of MMU equipped processors is important to ITRON. MMU equipped processors can provide software protection from other software. Real-time software that runs directly on ITRON needs protection from software bugs. General purpose software that runs on top of ITRON, e.g. BTRON, needs protection from both software bugs, and malicious software.

This paper identifies the areas of the ITRON2 specification which need to be changed, and the issues that need to be resolved to extend the ITRON2 specification for MMU equipped processors. It also presents a proposed extension to the standard that resolves these issues.

1. Introduction

This paper extends the ITRON2 specification[1] in three areas: multi-application support, MMU support, and secure system support. The last two of these require a MMU equipped processor. The first one does not require a MMU equipped processor. It is included here because it is required for secure system support, and one aspect of multi-application support can be provided by MMU support. (Within this paper ITRON2 extended for MMU equipped processors is called ITRON/MMU. This is not an official name.) The motivation for extending the ITRON2 specification in these three areas follows.

Adding multi-application support fixes a deficiency in the ITRON2 specification. The ITRON2 specification can not support multiple applications on the same host. This problem is not limited to just multiple custom applications running on the same host. It prevents the development of standard packages which can be used as part of many custom applications.

Adding MMU support to ITRON has benefits which are not unique to ITRON, but common to any operating system. MMU support adds benefits in the areas of reliability, security, multi-application support, and functionality. MMU support can detect program errors, and limit the extent of damage of program errors. This results in more reliable applications, and supports writing applications that recover from program errors. Reliability is so crucial to real-time applications that this benefit alone is sufficient to justify adding MMU support to ITRON. However, in addition MMU support is needed to make it possible to implement a secure system on top of ITRON/MMU. A secure system requires memory protection at least to protect system code and data from application code. A general purpose system on top of ITRON even if single user, e.g. BTRON, needs computer security to protect it from computer viruses. MMU support removes one barrier to multi-application support: multiple applications with the same memory addresses. MMU support adds functionality needed for algorithms that require detecting access to memory areas. Finally, MMU support is needed to add virtual memory to an operating system.

Adding secure system support to ITRON broadens the scope of applications that can be built with ITRON. A secure system can not be built on top of an insecure system. At least an overlying secure system must be able to prevent an application from directly accessing an underlying insecure system. In a general programming environment this requires some minimal security support from the bottom level insecure system.

2. Structure of this paper

This paper has the following structure. Section 3 identifies the issues that must be resolved in order to extend the ITRON2 specification in these three areas. Section 4 provides extensions of the standard in these three areas. Each area is in a separate subsection. Several alternative extensions are mentioned. Two are selected and described in detail. One maximizes backwards compatibility with the ITRON2 specification. The other has slightly less backwards compatibility but offers some improved features. Finally, Section 5 presents a conclusion, status of this work, and directions for future work.

3. Issues that must be resolved in order to extend the ITRON2 specification in the three areas

3.1. Multi-application support

Multi-application support requires solving naming conflicts. Name is used in the most general sense. In ITRON it includes object ids, handler numbers, common exception handlers, SVC handlers, interrupt handlers, and memory addresses of both code and data. One solution to naming conflicts is multiple separate name spaces for resources, and address spaces for programs. Adding multiple name spaces to ITRON requires solving the following issues.
3.1.1. System calls that return object ids or handler numbers

System calls that return object ids or handler numbers can not be extended to multiple name spaces without creating backwards compatibility problems with the ITRON2 specification. Either system call interface changes, or changes to the way applications must be written are required. The root of the problem is a conflict between two elements of the design in the presence of separate name spaces: some functions return object ids, and object ids are normally chosen by the creator of an object, not allocated by the system. If any object id to be returned by a system call is not within the name space of the task making the system call, the system must allocate a new object id within the caller’s name space to refer to that object. Since both the system and the application are allocating object ids, each one must choose only object ids that have not already been allocated by the other. This requires the application to either keep track of the object ids allocated by the system, or retry create() with a different id if the id chosen is already allocated by the system. This programming problem is itself a backwards compatibility problem. If an application is not designed to choose object ids not used by the system, it can choose object ids already allocated by the system, not detect this error, and not perform properly. This programming problem can be removed, but only by changing the create() system calls which itself creates a backwards compatibility problem. This same problem occurs with the handler numbers of task specific alarm and cyclic handlers.

3.2. MMU support

The addition of MMU support is primarily the addition of support for multiple address spaces. Adding multiple address spaces to ITRON requires solving the following issues.

3.2.1. Mailboxes

These system calls can not work between multiple address spaces because the message size is not known. Without the size it is impossible to copy a message between address spaces. These system calls will work between any tasks if the message is in the shared semi-space, or between two tasks in the same logical space if the message is not in the shared semi-space. If the message is not in the shared semi-space, they will not work between two tasks in different logical spaces. In that case message buffers must be used.

3.3. Both multi-application support and MMU support

3.3.1. Interrupt handlers

An interrupt handler may be invoked directly by the microprocessor, or may require intervention by the operat-

<table>
<thead>
<tr>
<th>Specification</th>
<th>Changes made</th>
<th>MMU</th>
<th>Logical spaces</th>
<th>Interrupt handlers</th>
<th>Restrictions for Hardware invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITRON2</td>
<td></td>
<td></td>
<td></td>
<td>Invoked by</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hardware</td>
<td>1. Save &amp; restore registers to avoid need of A.</td>
</tr>
<tr>
<td>ITRON/MMU: Without multiple name spaces</td>
<td>Address spaces, MMU</td>
<td>Yes</td>
<td>One physical, One</td>
<td>Ids</td>
<td>Hardware or OS intermediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1. Save &amp; restore registers to avoid need of A.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2. Shared semi-space to avoid need of B.</td>
</tr>
<tr>
<td>ITRON/MMU: With multiple name spaces</td>
<td>Address spaces, MMU, Name spaces</td>
<td>Yes</td>
<td>Multiple virtual, Multiple</td>
<td>Capabilities</td>
<td>OS intermediate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Never</td>
</tr>
</tbody>
</table>

Table 1: ITRON Specifications: Comparison & Changes from one to the next.
ing system. The amount of operating system intervention depends on how many of the following three services are required. If none are required, the interrupt handler can be invoked directly by the hardware.

A. Set up for an interrupt handler written in a higher level language.
   This includes saving and restoring registers that are not saved and restored by the compiled high level-language interrupt handler, and passing one parameter to the interrupt handler. In all the ITRON specifications whether this is needed or not is specified by the interrupt handler attribute when the interrupt handler is defined.

B. Make address spaces either accessible or inaccessible.
   The interrupt handler must be in an address space accessible to the defining task. Optionally, other address spaces may be made inaccessible. Making other address spaces inaccessible guards against interrupt handler bugs that access address spaces other than the one in which the interrupt handler is defined. However, it does not guard against deliberately accessing other address spaces. Interrupt handlers can use privileged instructions to access any memory on the system.

C. Make name spaces either accessible or inaccessible.
   The interrupt handler must be in a name space accessible to the defining task. Optionally, other name spaces may be made inaccessible. Like address spaces, making other name spaces inaccessible guards against only bugs, not deliberately accessing other name spaces.

The ITRON specifications differ in the requirements an interrupt handler must meet in order to be directly invoked by the hardware. (See table 1 below.) ITRON2 without MMU requires in order to avoid the need for A that the interrupt handler save and restore all registers that are modified, and either be written in assembler, or have the same interface as a interrupt handler written in assembler. B and C are not required as only one address space and name space is supported. ITRON/MMU without multiple name spaces adds one more requirement in order to avoid the need for B: an interrupt handler’s code and data must be part of the shared semi-space. Even in this case operating system intervention is required if the unshared semi-space is to be made inaccessible. If the interrupt handler’s code or data are part of the unshared semi-space operating system intervention is required to switch the unshared semi-space. ITRON/MMU with multiple name spaces always needs operating system intervention to invoke interrupt handlers. The name space must be changed, C above, even if the two previous restrictions are met.

3.4. Secure system support

A secure system must provide both memory protection and resource protection. Memory protection is considered as part of MMU support. One way to provide resource protection is to control the access to names of resources. A resource that can not be named can not be operated upon. Issues related to multiple name spaces were considered as part of multi-application support. Additional resource protection issues in ITRON related to neither MMU support, or multi-application support are listed below.

3.4.1. Shared objects

If two name spaces share an object, they both have full access to the object. There is no way to allow just enough access to a shared object to do a job. For example, a server uses a message buffer to receive requests from clients. Clients must be able to send a message to the message buffer, but they should not be able to read a message from the message buffer, or delete the message buffer.

3.4.2. System calls that return object ids or handler numbers

System calls that return object ids or handler numbers can not be extended to multiple name spaces without creating security problems. acp_port() is the most important of these. Sending a message with cal_port() gives the caller’s task id to the message receiver. Due to the shared object problem this gives the message receiver full access to the task: read or write its state, control it, or terminate it. All the message receiver really should be able to do is reply to the message.

4. Extensions to the ITRON specification

This paper presents alternate extensions of the standard at two levels of backwards compatibility because in any system change there is always a conflict between backwards compatibility, and adding new features. Neither one achieves full binary compatibility with ITRON2. In both cases the start-up code must be modified as the data passed at start-up of ITRON/MMU has to change.

A. Application program—Binary compatibility

If the application program is compiled or assembled separately from the start-up, the application program need only be linked with the new start-up code. This requires that both the assembler and C interface specifications be unchanged provided attribute values defined within the ITRON2 specification are used. The interface could be different when new attribute values, that were illegal before, are specified.
Figure 1: Logical spaces and address spaces: A comparison of ITRON2 and ITRON/MMU

B. Application program—C source level compatibility, Simple changes needed in assembler source

ITRON2 application programs written in C do not need source modifications, and do not need to be recompiled. However, they need to be linked with a new ITRON2 compatible C system call interface library. This library conforms to the ITRON2 C interface specification, and the new ITRON/MMU as-
system call interface library.) ITRON2 application programs written in assembler would need simple editing to supply values for any added parameters required by system calls. This requires that the new ITRON/MMU specification be a super set of the ITRON2 specification, and the C interface and assembler interface specifications be the same except for addition of new parameters, or new attribute values.

4.1. MMU support

4.1.1. Tasks, and Logical spaces

The addition of MMU support is primarily the addition of support for multiple address spaces. Address spaces are not represented as entities by themselves, but are part of a logical space. A logical space defines an execution environment. A logical space contains:

- An address space which is composed of two semi-spaces: an unshared semi-space, and the shared semi-space. The address space contains only memory that has been allocated to either of its semi-spaces.
- Zero or more tasks.
- Definitions of common exit and exception handlers, and extended system call handlers. (See section 4.2.2.)
- A name space if multiple name spaces are supported. (See B in section 4.2.1.)

Tasks always execute within the logical space in which they were created. Task names can be moved between multiple logical spaces, if multiple name spaces are supported. However, the task itself does not move, but remains within the logical space in which it was created.

Tasks can either share all memory and resources, or have separate memory and, if separate name spaces are supported, separate resources. All tasks within the same logical space share all memory and resources. Tasks within different logical spaces share all shared memory, i.e. memory allocated by get_blk(), and local memory only if sharing was enabled by inh_spcl() or cre_map(). Tasks within different logical spaces share all resources if multiple name spaces are not supported. If they are, tasks within different logical spaces share resources only if sharing was enabled. (See B in section 4.2.1.)

The ITRON2 specification does not have logical spaces. Rather all tasks share all memory and resources. The two are compared in figure 1. This is equivalent to using ITRON/MMU and creating all tasks within the same logical space. This single logical space is created at system start-up, and all needed code is mapped into it at that time. Creating additional tasks only requires the two steps required by ITRON2: creating, and starting the task.

Several system calls are added to create and delete logical spaces. Create space, cre_spcl(), creates a new logical space without any tasks. Delete space, del_spcl(), deletes a logical space and all tasks contained within it, if any. Delete task with space, mdel_tskcl(), deletes a task and the logical space that contains it as if del_spcl() was called. Exit and delete task with space, mexit_tskcl(), causes the calling task to exit, and deletes both the calling task and the logical space that contains it as if del_spcl() was called. Normally delete task, del_tskcl(), is used instead of mdel_tskcl(), and exit and delete task, exit_tskcl(), is used instead of mexit_tskcl(). A logical space is deleted automatically when the last task within it is deleted.

One system call is either added or modified in order to create a task within a logical space. One alternative is to not change the create task system call, cre_tskcl(), and add system call create task within space, cre_stskcl(). The cre_tskcl() system call always creates a task in the logical space of the caller. The cre_tskcl() system call is exactly the same except it has an additional parameter, spcid, which specifies the logical space within which to create the task. This alternative has "Application program—Binary compatibility" with ITRON2. A second alternative is to not add cre_tskcl(), but add the logical space parameter directly to cre_tskcl(). This alternative has only "Application program—C source level compatibility. Simple changes needed in assembler source" with ITRON2.

4.1.2. Allocation and sharing of memory

Memory can be allocated, and shared in several ways. New memory blocks can be allocated from a local memory pool, or shared memory pool to a single logical space, or all logical spaces, respectively. Ranges of already allocated memory can be shared between specified logical spaces, but only in a hierarchy, or at multiple addresses within the same logical space at different protection levels. Physical memory not managed by the kernel, i.e. not part of a memory pool, can be allocated to one or more logical spaces.

The specification of the memory pool operations are nearly the same as in ITRON2. The shared memory pool system calls themselves do not change. However, the logical memory addresses which can be returned by get_blk() may be restricted. In particular when implemented on a TRON processor get_blk() is likely to be implemented to return only addresses in the shared semi-space. The local memory pool system calls are changed to allow allocating memory blocks in any logical space. Either a logical space parameter is added to the get local memory block, get_l_b1(), and release local memory block, rel_l_b1(), or get_l_b1(), and rel_l_b1() are not changed and new system calls, mget_l_b1(), and mrel_l_b1(), that are the same but have the new logical space parameter.

Memory shared by all logical spaces can not be efficiently implemented in a loosely-coupled multiprocessor environment. It probably will not be available on a loosely-coupled multiprocessor version of ITRON. (A restricted form could be available that is shared only between two logical spaces on the same host. This would still be adequate for fast interrupt handlers, and cyclic handlers that do not require address space switches.) Application programs that may be run on a loosely-coupled multipro-
cessor must not use memory that is shared by all logical spaces.

Memory can be shared between specified logical spaces in only a hierarchy. This restricts logical spaces that share memory to be on the same host. This facilitates extending ITRON to a loosely-coupled multiprocessor without requiring kernel support of memory shared between hosts without shared physical memory.

Memory sharing between specified logical spaces can only be established when a new logical space is created. First, when a new logical space is created it inherits already allocated memory from the logical space of the task that created it according to memory inheritance values set by \texttt{inh_spec()}. Memory inheritance values are associated with memory regions at the page level. Memory is either shared, copied, or not inherited. Memory regions that are not allocated within the creating logical space are not allocated within the new logical space. Memory regions allocated within the creating logical space that are not inherited, shared, or copied are not allocated, allocated and shared, or allocated and copied, respectively, within the new logical space. Second, system calls that transfer, or share memory mappings are restricted to have the same logical space for both source and destination.

Memory can be shared within the same logical space at different addresses with different protection settings. First, the memory is allocated. Then \texttt{sha_map()} is used to map a range of memory at another address with a new memory protection setting. This is useful for algorithms in which access to memory must be denied to clients at the same time access is allowed to servers.[2]

Physical memory not managed by the kernel, i.e. not part of a memory pool, can be allocated to one or more logical spaces, one at a time, by \texttt{cre_map()}. An attempt to map physical memory that is managed by the kernel fails. A common use of this system call is to map read-only memory that contains code into a logical space.

### 4.1.3. Memory maps: Memory protection, Transfer, and Deletion

Existing memory maps may be deleted by \texttt{del_map()}, transferred to a different address in the same logical space by \texttt{trf_map()}, and the memory protection changed by \texttt{set_ptr()}. If the last mapping of a memory block from either a local memory pool, or a shared memory pool is deleted the memory block is released back to the memory pool.

### 4.1.4. Handle memory protection faults

Memory exception handlers for tasks, SVC, and the task independent portion may be defined, and examined. Some applications use the MMU to detect reads or writes to memory areas.[2] All of these require user supplied memory exception handlers.

1. This restriction, and the idea of memory inheritance properties was adopted from Mach.[3]

### 4.1.5. Inter-task communication, and Copying data between logical spaces

Message buffers, and ports are exactly the same specification as in ITRON2. The implementation may be different. In ITRON2 implementations they always copy data. An ITRON/MMU implementation could avoid the copy by mapping the message copy-on-write into the receiver provided that both the message and the receiver’s data message buffer are page aligned and an even multiple of the page size.

Mailboxes are the same within a semi-space, but do not work between multiple unshared semi-spaces, see section 3.2.1. They should not be modified in order to work. They are optimized for efficiency. Messages are not copied but passed by reference. Messages queued waiting a receiver are linked into a doubly-linked list via fields in the message header provided by the sender, not allocated by the operating system. Any modification to make them work between multiple address spaces would make them less efficient, and incompatible with the current mailboxes as the message header format would have to change. Rather than making them work they should be modified to fail and return error code E_ILLEGAL if the message sent is neither in the shared semi-space nor in the same logical space in which the mailbox was created.

### 4.1.6. Creating and starting a new task: Examples of how the MMU support calls are used

Several steps are necessary to create and start a new task. Most of these are preparing the logical space in which the task is to be created. If the logical space is already fully prepared a task is created and started by the same two steps used in ITRON2. All steps must be performed if a task is created and started in a new logical space. Many variations are possible. Three scenarios are given below as examples of how the MMU support system calls are used.

- The code of the new task is stored in ROM.

  1. Select which read/write memory is shared between the current logical space and the new logical space. Set the memory inheritance properties of the current logical space with \texttt{inh_spec()}

  2. Create the new logical space with \texttt{cre_spec()}. The address space of the new logical space contains only the memory specified to be copied or shared in step 1.

  3. Map ROM resident code to the new logical space with \texttt{cre_map()}

  4. If multiple name spaces are supported, prepare to share resources between the current logical space and the new logical space. Copy ids of objects to be shared into the new logical space. (See B in section 4.2.1.)

  5. Create the new task with \texttt{cre_stsk() or cre_tsk()}

  6. Start the new task with \texttt{sta_tsk()}

---

30
• The code of the new task is stored in a file. The code will fit in the current address space.

1. Allocate memory in the current logical space to contain all the code of the new task using get_lblo(). This memory need not be allocated at the proper address for execution.

2. Read the code of the new task from disk into the memory allocated above.

3. Set the inheritance of the memory containing the code from to shared using inh_spco().

4. Select which read/write memory is shared between the current logical space and the new logical space. This is the same as step 1 in the ROM scenario.

5. Create the new logical space with cre_spco(). This is the same as step 2 in the ROM scenario.

6. Transfer the memory that contains the code from the allocated address to the execution address within the new logical space using tr_map().

7. Remove the code for the new task from the current logical space with del_map().

8. If multiple name spaces are supported, prepare to share resources between the current logical space and the new logical space. This is the same as step 4 in the ROM scenario.

9. Create the new task with cre_stsk() or cre_tsk(). This is the same as step 5 in the ROM scenario.

10. Start the new task with sta_tsk(). This is the same as step 6 in the ROM scenario.

• The code of the new task is stored in a file. The code will not fit in the current address space.

1. Select which read/write memory is shared between the current logical space and the new logical space. This is the same as step 1 in the ROM scenario.

2. Create the new logical space with cre_spco(). This is the same as step 2 in the ROM scenario.

3. Allocate memory in the new logical space to contain all the code of the new task using mget_lblo(). This memory must be allocated at the proper address for execution.

4. Allocate a memory buffer in the current logical space with get_lblo().

<table>
<thead>
<tr>
<th>Thing—Name</th>
<th>ITRON2</th>
<th>ITRON/MMU without multiple name spaces</th>
<th>ITRON/MMU with multiple name spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Solution A</td>
<td>Solution B or C</td>
</tr>
<tr>
<td>Interrupt handlers—Vector number</td>
<td>User</td>
<td>User</td>
<td>User</td>
</tr>
<tr>
<td>Exit and exception handlers for task independent portion</td>
<td>System</td>
<td>System</td>
<td>Logical space</td>
</tr>
<tr>
<td>Objects—Ids</td>
<td>User</td>
<td>OS</td>
<td>User or OS</td>
</tr>
<tr>
<td>Alarm and cyclic handlers—Handler number</td>
<td>User</td>
<td>OS</td>
<td>User or OS</td>
</tr>
<tr>
<td>Common exit and exception handlers for tasks</td>
<td>System</td>
<td>Logical space</td>
<td>User</td>
</tr>
<tr>
<td>Common exit and exception handlers for SVC handlers</td>
<td>Logical space</td>
<td>User</td>
<td></td>
</tr>
<tr>
<td>Extended system call handlers (SVC handlers)—Function codes</td>
<td>User</td>
<td>User</td>
<td></td>
</tr>
<tr>
<td>Exit and exception handlers for SVC handlers</td>
<td>Task</td>
<td>Task</td>
<td></td>
</tr>
<tr>
<td>Exit and exception handlers for tasks</td>
<td>Task</td>
<td>Task</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Name spaces: A comparison of ITRON2 and ITRON/MMU
4.2. Multi-application support

The ITRON2 specification does not provide a multi-application environment. Conflicts between applications can occur in several areas of the specification. They are listed below grouped according to the type of solution. Each is discussed in greater detail in the following sections. The system calls that create or define these are given at the end of each list entry.

- "Creator" assigned identifiers. Several alternate solutions given.
  - Object ids: cre Xxx().
  - Handler numbers of both alarm and cyclic handlers: def alm(), and def cyc().
- Per application entities. Two solutions are given.
  - Common exit and exception handlers that are used by any task which does not have a specific handler defined: def exit(), def cex(), def fex(), and def sex().
  - Common exit and exception handlers that are used by any extended system call handler which does not have a specific handler defined: sdef exit(), sdef cex(), sdef fex(), and sdef sex().
  - Extended system call handlers: def s ve().
- Per system entities. Conflict not important and not resolved.
  - Exit and exception handlers for interrupt processing, the ITRON specification calls this the task independent portion: idef exit(), idef cex(), idef fex(), and idef sex().
  - Interrupt handlers: def int().

Multi-application support can be added to ITRON2 without adding MMU support. All the conflicts between applications listed above have nothing to do with whether a MMU is supported. There is one additional conflict between applications that must be solved in order to have a multi-application environment: applications must not overlap in memory. This conflict is outside the scope of the ITRON2 specification because the specification does not include a program loader. Yet even this conflict can be solved without a MMU by using position independent code for all applications.

Multi-application support is included in ITRON/MMU because the whole problem is now within the scope of the specification. Adding MMU support brings application memory overlap within the scope of the extended specification. This conflict can be solved by providing multiple virtual address spaces.

4.2.1. "Creator" assigned identifiers: Object ids, and handler numbers

These are the most serious as they are the most common. The causes are: (1) object ids, and handler numbers are specified when objects are created, and handlers are defined, and (2) object ids, and handler numbers are global to all tasks. The solution is to remove either or both of the causes of the problem. Several alternate solutions are listed below and summarized in table 1 below.

A. Remove cause (1)—This is chosen in this paper for both compatibility levels.

The system optionally assigns object ids when objects are created, and handler numbers when handlers are defined. This requires changing all the object creation and exception handler definition system calls. Add an object attribute, and handler attribute, TA_ALCID which is 0x8000. If this parameter is not set the system call behaves identically as before. If this is set, the system call allocates the object id or handler number and returns it. The TA_ALCID bit of the object attribute, or the handler attribute is set to zero before the attribute is stored. The C interface and two sample calls to one system call is given below as an example of the change required. The object id or handler number argument is changed to a union.

```c
/* ITRON2: Before change */
ER cre fgl(ID fglid, ATR fgattr); /* Create event flag */
error = cre fgl(idf) 3, (ATR) 0); /* Caller supplied id */
/* ITRON/MMU: After change */
typedef union t id or_ptr

IDid; /* Not TA_ALCID: Caller supplied. */
IDid p id; /* TA_ALCID: System assigned. */
) T ID OR_PTR;```
This provides full source compatibility with ITRON2 for one application whether it is written in C or assembler. At most one ITRON2 application which specifies its own object ids and handler numbers can be run along with any number of new applications that use system assigned object ids and handler numbers.

TA_ALCID does not exactly follow the meaning of an object or handler attribute. Whether it is specified or not does not change the object or handler, only the manner in which the object’s id or handler’s number is selected. Another parameter selecting whether system allocation is performed would be cleaner. However, TA_ALCID is better. One system call that would have to be changed, def alm(), can not have another parameter added. It already has the maximum number of parameters allowed by ITRON2. In addition, adding another parameter would not provide source compatibility with ITRON2.

This method does not provide security. All object ids and handler numbers and all operations upon them are accessible to every task in the system.

B. Remove cause (2).

Object ids and handler numbers are local to each logical space, i.e. each logical space has its own local name space. Several alternate changes one of which would be required by this are discussed below.

1. Add a parameter to all existing system calls that require a object id or handler number as a parameter. The new parameter specifies which logical space contains the object or handler. Three such system calls can not have this new parameter added: accept port for rendezvous, acpsro, cyclic wakeup task, cyc_wup(), and define alarm handler, def alm(). These system calls already have 6 parameters, the maximum number of parameters allowed by the ITRON2 specification. This does not provide any secure method to transfer a object id to a server.

<table>
<thead>
<tr>
<th>Compatibility level—A is higher, but B is more useful. (See section 4.)</th>
<th>A. Application program—Binary compatibility</th>
<th>B. Application program—C source level compatibility, Simple changes needed in assembler source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-application support for object ids and handler numbers (See section 4.2.1.)</td>
<td>A. The system optionally assigns object ids when objects are created, and handler numbers when handlers are defined.</td>
<td>A. The same as left.</td>
</tr>
<tr>
<td>Create a task in another logical space (See section 4.1.1.)</td>
<td>Do not modify cre_tsk(). Add cre_stsk().</td>
<td>Do not add cre_stsk(). Modify cre_tsk() to do the function of cre_stsk().</td>
</tr>
<tr>
<td>Get/Release a local memory block in another logical space (See section 4.1.2.)</td>
<td>Do not modify get_lbl(), and rel_lbl(). Add mget_lbl(), and mrel_lbl().</td>
<td>Do not add mget_lbl(), and mrel_lbl(). Modify get_lbl(), and rel_lbl() to do the function of mget_lbl(), and mrel_lbl(), respectively.</td>
</tr>
<tr>
<td>Multi-application support for common exit and exception handlers, and extended system call handlers (See section 4.2.2.)</td>
<td>A. A logical space inherits all the common exit and exception handlers, and extended system call handlers of the logical space which is current when it is created.</td>
<td>B. Add a logical space parameter to each system call to either define or get either a common exit and exception handler or a extended system call handler. For exit and exception handlers the logical space parameter is ignored if not specifying a common handler.</td>
</tr>
</tbody>
</table>

Table 3: Changes in ITRON/MMU from ITRON2 which depend on compatibility levels: With minimal security features, and without multiple name spaces
2. Add system calls to manipulate ids and handler numbers: move them from one name space to another, delete them, read the contents of a name space, transfer them in messages, etc.

3. Do both of the above—This is the preferred solution if multiple name spaces are provided.

This method can be extended to provide security. The local name spaces in each logical space contain capabilities rather than object ids. (It is not necessary to use capabilities for handler numbers. Only one operation upon handlers may need protection: Activate cyclic handler, act_cyc(). This one case does not warrant the complexity of introducing capabilities for handlers.) However, this method is complicated. Many new system calls need to be added, and nearly all existing system calls need to be modified.

C. Remove both causes (1) and (2).

All of the B.x and C.x solutions have backwards compatibility problems with the current ITRON2 specification due to either system call interface changes, or to changes in the way an application must be written. The root of the problem is a conflict between two elements of the design in the presence of separate name spaces: some functions return either object ids or handler numbers, and object ids or handler numbers are not allocated by the system. Method B.1 requires changing all system calls that create objects, or define exception handlers, alarm handlers, and cyclic handlers, but not system calls that define interrupt handlers and extended system call handlers. Method B.2 does not require changing any system calls. However it requires application programming changes. An application must either keep track of the object ids and handler numbers allocated by the system, or re-try the create and define system calls with a different object id or handler number if the object id or handler number tried is already allocated by the system.

4.2.2. Per application entities: Common exit and exception handlers, and extended system call handlers

These are less serious than the above, but still need a solution. The only solution is to make them local to each logical space. Two ways to do this are listed below.

A. A logical space inherits all the common exit and exception handlers, and extended system call handlers of the logical space which is current when it is created. Any changes made later in that logical space do not effect the newly created logical space. System calls are neither changed nor added. This is source compatible with the current ITRON2 specification—This is chosen in this paper for compatibility A. Application program—Binary compatibility.

B. Add a logical space parameter to each system call to either define or get either a common exit and exception handler or a extended system call handler. The new logical space parameter LSP_SELF with value 0 refers to the current logical space. The new logical space parameter need only be used for common handlers, and could be ignored otherwise. However, if a logical space contains a local name space the new logical space parameter should be used all the time. This is less confusing, and could be useful in a debugger. This solution is not source compatible with the current ITRON2 specification—This is chosen in this paper for compatibility B. Application program—C source level compatibility, Simple changes needed in assembler source.

4.2.3. Per system entities: Exit and exception handlers for interrupt processing, and Interrupt handlers

These entities should have only one definition per system. Thus conflicts are not possible. All interrupt vectors have the same exit and exception handlers. In addition, the interrupt processing exit and exception handlers are invoked also if any fatal error or error of unknown cause occurs in the operating system. Due to these two facts, they must be one per system. Second, each interrupt vector has only one interrupt handler in order to reduce the overhead in invoking an interrupt handler. (An alternative would be a chain of interrupt handlers per interrupt vector. Each interrupt handler would return whether it had handled the interrupt or not. ITRON would invoke each interrupt handler in an interrupt vector's chain in turn until the interrupt is handled. However, this increases the overhead in invoking an interrupt handler.)

4.3. Security

ITRON/MMU is defined to be a secure system. (ITRON2 is not, and can not be as it does not include memory protection.) Security must be added so that a secure system can be built on top of ITRON. It is not possible to build a secure system on top of an insecure system.

The security features added to ITRON/MMU are minimized. Many, but not all, applications of ITRON are single applications per machine that do not require security. The security features are kept minimal so that these applications do not need to pay execution time or memory space for security features they do not need.

4.3.1. Denial of access to all ITRON system calls

ITRON/MMU includes only very broad grain security: denial of access to all ITRON system calls except those made by the system on top. This pushes all security mechanisms into the overlying system, and adds the minimum to ITRON/MMU itself. All ITRON system calls are made by first invoking the overlying system. It then
makes the ITRON system call on behalf of the caller. The overlaying system can implement any security mechanism and policy.

A new system task attribute is defined: TA_NOSYSSCALL which is 2. Every system call aborts and returns error E_SCACV if the system call is made from a task, and TA_NOSYSSCALL is set in that task. If the system call is either not made from a task, or the calling task does not have TA_NOSYSSCALL set the system call executes. Whenever a task is created the TA_NOSYSSCALL attribute of the new task is set if either it is specified, or the calling task has TA_NOSYSSCALL set. This prevents a task with TA_NOSYSSCALL set getting around the system call restriction by creating another task without TA_NOSYSSCALL set.

**TRON specification chip interface**

A TRON chip implementation would allow system calls if either the previous ring was ring 0, in which case the system call was not from a task, or TA_NOSYSSCALL is not set in the calling task. The overlaying system would run in ring 0.

5. Conclusion, status, and directions for future work

The ITRON2 specification can be readily extended to provide MMU support, multi-application support, and secure system support. MMU support consists of providing multiple address spaces, one per logical space. Tasks may be created in the current logical space, or any other logical space. A task executes within only one logical space, the one in which it was created. For compatibility, and, especially, for efficiency, mailbox system calls are restricted to contain only messages within one unshared address space, and messages within the shared address space. Multi-application support requires the removal of naming conflicts between applications. This can be done by system assignment of names, and/or local name spaces within each logical space. The first is less complicated and has been chosen in this paper. The second can be extended to provide capability security features. It was not chosen because most of ITRON applications do not need such a full computer security system. Instead, access control for all ITRON system calls is added so that an overlaying system can implement a full security system for the applications that require it. Alternatives were chosen to extend ITRON2 in these three areas at two levels of backwards compatibility. The alternatives chosen are summarized in table 1 below.

This project is ongoing. A detailed specification has been prepared for the extensions described in this paper. The implementation of the Application program—Binary compatibility alternative has begun. The extensions described in this paper provide a full in-memory MMU system. They do not provide paging. Additional extensions could be developed to support paging resulting in a full virtual memory system. We have focused on an in-memory system first as it is useful to more systems.

**References**


# APPENDIX

## A. System calls for MMU support

<table>
<thead>
<tr>
<th>System call</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create space</td>
<td>ER cre_sp(H spcid, ATR spcatr)</td>
</tr>
<tr>
<td>Delete space</td>
<td>ER del_sp(H spcid)</td>
</tr>
</tbody>
</table>

Table 4: MMU Support: Create and delete logical spaces

<table>
<thead>
<tr>
<th>System call</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create task within space</td>
<td>ER cre_stsk(H tskid, ATR tskatr, FP stadr, TPRI itskpri, W stksz, H spcid) or cre_tsk() with the same definition.</td>
</tr>
<tr>
<td>Delete task with space</td>
<td>ER mdel_tsk(H tskid)</td>
</tr>
<tr>
<td>Exit and delete task with space</td>
<td>void mexit_tsk(void)</td>
</tr>
</tbody>
</table>

Table 5: MMU Support: Create and delete tasks within a logical space

<table>
<thead>
<tr>
<th>System call</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set inheritance of logical space</td>
<td>ER inh_sp(H spcid, W logadr, UW len, UW incode)</td>
</tr>
<tr>
<td>Get shared memory block—Unchanged</td>
<td>ER get_blk(VP *p_blk, H mplid, W bcnt, TMO tmout)</td>
</tr>
<tr>
<td>Release shared memory block—Unchanged</td>
<td>ER rel_blk(H mplid, VP p_blk)</td>
</tr>
<tr>
<td>Get local memory block to space</td>
<td>ER mget_lbl(VP p_blk, H lplid, W bcnt, TMO tmout, H spcid, UW option) or get_lbl() with the same definition.</td>
</tr>
<tr>
<td>Release local memory block from space</td>
<td>ER mrel_lbl(VP p_blk, H lplid, H spcid) or rel_lbl() with the same definition.</td>
</tr>
<tr>
<td>Create logical mapping</td>
<td>ER cre_map(W phyaddr, W len, ATR prcode, H dstspid, W *p_dstdadr, UW option)</td>
</tr>
<tr>
<td>Share logical mapping</td>
<td>ER sha_map(H spcid, W srcadr, W *dstadr, W len, ATR UW option)</td>
</tr>
<tr>
<td>Transfer logical mapping</td>
<td>ER trf_map(H spcid, W srcadr, W *dstadr, W len, ATR prcode, UW option)</td>
</tr>
<tr>
<td>Delete logical mapping</td>
<td>ER del_map(H srcspid, W srcadr, W len)</td>
</tr>
<tr>
<td>Set memory protection</td>
<td>ER set_prt(H spcid, W logadr, UW len, ATR prtr, UW option)</td>
</tr>
</tbody>
</table>

Table 6: MMU Support: Add, modify, and remove memory maps within a logical space
### Table 7: MMU Support: Handle memory protection faults

<table>
<thead>
<tr>
<th>System call</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define memory exception handler</td>
<td>ER def_mex(ID spcid, ID tskid, HATR exhatr, FP mexhdr)</td>
</tr>
<tr>
<td>Define Memory exception handler for SVC</td>
<td>ER sdef_mex(ID spcid, FN s_fncd, HATR exhatr, FP mexhdr)</td>
</tr>
<tr>
<td>Define Memory exception handler—Task independent</td>
<td>ER idef_mex(HATR exhatr, FP mexhdr)</td>
</tr>
<tr>
<td>Get Memory Exception Handler</td>
<td>ER get_mex(ID spcid, HATR *p_exhatr, FP *p_mexhdr, ID tskid)</td>
</tr>
<tr>
<td>Get Memory Exception Handler for SVC</td>
<td>ER sget_mex(ID spcid, HATR *p_exhatr, FP *p_mexhdr, FN s_fncd)</td>
</tr>
<tr>
<td>Get Memory Exception Handler—Task independent</td>
<td>ER iget_mex(HATR *p_exhatr, FP *p_mexhdr)</td>
</tr>
</tbody>
</table>

### Table 8: MMU Support: Move data between logical spaces

<table>
<thead>
<tr>
<th>System call</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive message from mailbox—Modified</td>
<td>ER rcv_msg(T.MSG **pk_msg, ID mbxid, TMO tmout)</td>
</tr>
<tr>
<td>Send message to mailbox—Modified</td>
<td>ER snd_msg(ID mbxid, T.MSG *pk_msg)</td>
</tr>
<tr>
<td>Receive message from message buffer—Unchanged</td>
<td>ER rcv_mbf(VP pk_bmsg, W *p_bmsgsz, ID mbfid, TMO tmout)</td>
</tr>
<tr>
<td>Send message to message buffer—Unchanged</td>
<td>ER snd_mbf(ID mbfid, VP pk_bmsg, W bmsgsz)</td>
</tr>
<tr>
<td>Move data to another space</td>
<td>ER mov_dat(ID srcspid, W srcadr, ID dstspid, W dstadr, W len)</td>
</tr>
</tbody>
</table>