To: MTB Distribution

From: Philippe Janson

Date: January 9, 1974

Subject; Design of a Dynamic Linker Running Outside Multics Security Kernel

I Introduction

Now that progress on my Master's thesis is to the point where the design of the dynamic linker running outside ring 0 is completed and the implementation is about to start, it seems worthywhile resting a moment to look back and think about all the implications of the design that is proposed. The purpose of this paper is to review the design, to justify the major tradeoffs between various of its aspects and to discuss its consequences. The motivations and goals of the design are reviewed -- the design is then outlined. Finally, we insist again on a few points developed in RFC-23 and add a discussion of several issues raised more recently during the last phase of the design. The architecture of the present dynamic linker is assumed to be known by the reader. It was briefly outlined in RFC-23.

II Motivations

In order to enforce the security of the information stored in a computer system, it is necessary to certify that the protection mechanism is correctly implemented so that there exists no uncontrolled access to the stored information. Certification requires that the security kernel be much simpler and smaller than the supervisors of the present general purpose operating systems, and of Multics in particular. Removing the dynamic linker from Multics security kernel is motivated by certification requirements: it is one step towards the reduction of the security kernel complexity.

There is also a protection motivation for having the dynamic linker running in the user environment. The dynamic linker handles information directly derived from source code provided by the users of the system. There is a fair chance that such code contains -- purposely or not -- inconsistencies capable of causing the linker to malfunction or perform unexpected operations. In order to certify the protection mechanism, we have to prove that malfunction or unexpected behavior cannot happen. It is hard to think of appropriate security checks on user code because potential inconsistencies in object code are numerous, vague and essentially unpredictable. Therefore, the only solution to enforce the protection is

to remove the linker from the security kernel so that its eventual malfunction cannot affect the protection mechanism of the whole system.

We have just explained two reasons why the linker should be removed from the security kernel. We still have to prove that it can be removed from the security kernel. We claim that the linker does not need the privileges of security kernel procedures and therefore, according to the least privilege principle, not only should but also can run in the user environment. We do not claim that this last assertion is obvious. Instead, it is the purpose of the rest of this paper to prove it.

As a last remark, we may well mention that our new design is more conceivable on the 6180 than on the 645 because frequent cross-ring calls resulting of this design are not more expensive than other calls, which was not true on the 645 machine.

III Goals

We think of the future linker as a set of procedures bound together in one single segment. This segment will be operational for any process in any user ring before that process needs to snap links in that ring. The linker will never be operational or known in ring 0. This implies that ring brackets be (1,5,5), with r, e access to *.*.* . (Of course, access could be restricted to some category of users later on if desired.)

The linker can be called in any ring to snap a link from a segment in that ring to any other segment in that -- and higher rings, or to gates into lower rings (provided the user has access to the segment or gates of course). On a linkage fault (link missing fault), the event discovered in ring 0 (fim) will cause the machine conditions to be passed as arguments.to the linker in the faulting ring. This sort of outward call will be handled by a mechanism similar to signalling although much simpler. The missing link will be snapped entirely in the faulting ring. On completion of this task, the control unit will be restored like in the signalling mechanism and execution will proceed. With the above design goals in mind, we will now carefully examine the potential problems, and explain how the design proposes to solve them. In a later paragraph, we will discuss the tradeoffs and implications of the proposed design.

IV Proposed Design: Problems and Solutions

The problems to be solved by the design can be classified into three categories with respect to their causes.

First of all, our goal stated that the linker has to be operational for any process in any ring before it is needed by that process in that ring. This design goal raises a whole set of problems further referred to as bootstrapping problems. These problems were solved by modifying or designing three ring 0 concepts: system initialization, process initialization and ring initialization.

Secondly, once the linker is operational, we want it to execute strictly in the faulting (calling) ring. This raises another set of problems further referred to as user ring problems because their solution requires modifying only user ring procedures and data bases.

Last, but not least, the design goals applied to cross-ring link handling have raised the need for minor hardware changes.

We will now describe the design without making any comment on its consequences and implications. This will be done in a later paragraph.

1) Bootstrapping features (ring 0)

What does it mean for the linker to be operational in some ring for some process? It means three things.

First, it implies that the linkage section of the bound linker contain no virgin (unsnapped) link. If this were not the case, execution of the linker when handling a linkage fault would cause other linkage faults to be taken thereby putting the process into an endless recursive loop.

Secondly, if the linker ever is to be used in anything by a process, it has to have a descriptor, i.e., a segment number in the address space of the process.

Finally, even when the two first conditions are met, the linker will not be able to run in a ring unless its linkage section is combined in that ring i.e., copied into the combined linkage segment of that ring.

Each of the three above requirements has motivated one of the three following ring 0 design features.

1) At System Initialization Level

The prelinking requirement is fulfilled during system initialization. As far as system initialization is concerned, the bound linker is considered to be an initialization data base. It is a member of the second collection of MST segments. After the first two collections of segments are loaded and prelinked (as in the present Multics system), a special purpose initialization procedure is invoked to carry on the prelinking of the bound linker. The entire linkage section of the linker is scanned. Each virgin link is passed to the standard system prelinker (pre_link_2). Notice that upon return from the prelinker, the snapped links contain initialization segment numbers (SLT seg #). These numbers will not be valid for use in a user environment because the descriptor words for such numbers restrict access to the corresponding segments to ring 0 procedures. Therefore, a suitable amount of user segment numbers (KST seg #) is reserved for

the linker and some external segments it references. A mapping from initialization segment numbers to the user reserved segment numbers is maintained and every snapped link containing an initialization segment number is translated into a link containing the user segment number given by the mapping. When all virgin links are snapped and translated, the special purpose prelinking procedure returns to its caller and initialization resumes. All initialization segments are discarded, leaving behind only the linker linkage section template containing all snapped links and the table containing the mapping of SLT numbers (and names) to KST numbers.

The above procedure obviously implies that all segments referenced by the linker be available at system initialization. To be more specific, they all are in collection 2 of the MST. It will be shown further that none of these segments in turn has unsnapped links.

2) At Process Initialization Level

The requirement that the linker (and also the external segments it references) have a segment number before it is needed is fulfilled during process initialization. Just before a newly created process transfers from ring 0 to a user ring for the first time, and hence just before it can take any linkage fault, it initiates the linker (and the segments it references). In order to do so, it uses the segment numbers mapping table left around by the system initializer. Each segment is initiated with its pathname and segment number reserved for it.

On completion of this task, the process may transfer to the outer ring. In the present system it does so by calling the linker to get a pointer to the entry of the first user ring procedure. In the new design, as there will be no linker available in ring 0, it will initiate the first user ring procedure and transfer to it at a conventional offset.

3) Ring Initialization Level

By now, the hypothetical process we were considering is in the outer ring. It apparently can take linkage faults and yet the third requirement that we had established is not fulfilled. Namely the linker linkage section is not yet copied in the combined linkage segment of the outer ring. In fact, we proved in RFC-23 that no matter what the scheme of things looks like in the outer ring, and even if the process takes a linkage fault, the stack segment of the ring will be referenced before the combined linkage segment. This will yield a segment fault on the stack. As a consequence the process will execute the appropriate procedure to fabricate the missing stack. We take advantage of this segment fault to, at the same time, fabricate the combined linkage segment. We call the two operations ring initialization.

Making the stack is done in a way almost similar to the present design except that the procedure used for that purpose cannot call the linker in ring 0, like it used to, to get miscellaneous pointers destined for the stack header. Instead, the pointers have been fabricated and saved by the system initializer exactly like the linker linkage section template was fabricated and saved. When making a stack, the pointers just have to be copied into the stack header. In addition to the current pointer, a pointer to linker$linkage_fault is stored into the stack header for later use.

Making the combined linkage segment is very trivial. Once an appropriate segment is created for that purpose, its header is initialized. The linker linkage section template is copied into the segment. The appropriate lot entry is set and all other entries are set to all one's (see later). The search rules are initialized as well (see later).

On completion of this third bootstrapping step, the bound linker is fully operational in the user ring. Should the user process go to another virgin ring, the ring initialization operations will again happen just in the same way, thereby making the linker operational in the new ring.

2) Hardware Modification (1): Fault Handling

At this point of our design, the linker is ready to fix links on request and to handle linkage faults. Unlike in the present design, where a linkage fault means a link is missing (ft2), in the new design, a linkage fault may mean one of two things: a link is virgin but exists in the combined linkage segment, or a linkage section is missing i.e., it is not conbined. The first linkage fault, to be known as a link fault, is the good old fault tag 2 in the virgin link. The second linkage fault, to be called a lot fault, is a new type of fault be to recognized by the hardware.

In the present design, the linkage section of a segment is combined by the linker when the segment is first referenced. In the proposed design, this is no more the case: the linkage section of a segment is combined at the very last moment, only when it is needed. To be even more explicit, when a procedure is entered, it loads (PL/I_operators_ loads) the processor lp register with the packed pointer contained in the appropriate lot entry. As we already mentioned, ring initialization sets all lot entries to all ones. The result of doing an lprplp on an all one's register is to be a fault recognized by the hardware. The need for this new feature is caused by a desire to handle gate and non gate segments homogeneously, and by the initial goal of having the linker running in the faulting ring.

To make things clear, suppose a ring four procedure calls a gate entry into ring three. On the first call, this causes a link fault. In the present design, where the linker runs in ring 0, it has the capability of setting the validation level to three and to combine the linkage section of the gate being referenced for the first time. But in the new design, how could the

linker running in ring four (faulting ring) have the capability to combine the linkage section in ring three? This is impossible and justifies wanting the linkage section to be combined when a procedure is first entered and not when it is first referenced. In our example, after the call to the gate is actually performed, the gate will attempt to load its linkage pointer. In doing so, it will cause a lot fault resulting in the linker being called in ring three (faulting ring) to combine the linkage section, which it can do now since it executes in ring three.

Both linkage faults cause the fault intercepting mechanism to save the machine conditions and call the standard signaller. The signaller sets up a stack frame in the faulting ring and transfers to linker$linkage_fault through the pointer stored in the stack header furing ring initialization. This entry point of the linker recognizes one or the other type of fault and call link_snap or link_man accordingly. On completion of the fault handling, the linker returns to the signaller. The signaller does a few checks on the machine conditions and then switches back to ring 0 and restores the control unit.

3) User ring features

All design features outlined in this paragraph result of the linker ring brackets being (1,5,5). This puts some restrictions on the type of external references, calls and arguments the linker can use.

1. Reference problems

As the linker is removed from ring 0, it will no longer have access to ring 0 data bases like kst, pds, active_hardcore_data, etc. Even before the new design was completed, we had the feeling that these restrictions were insignificant because the linker conceptually needs no access to ring 0 data bases. This feeling has proved to be correct: whenever the present linker references ring 0, it does so for no real good protection reason. In the case of the kst, the linker references the search rules. As search rules are per ring data, there is no reason why they should be in ring 0 in our new design. Hence we moved them to the user rings combined linkage segments (see later). In the case of the pds, the items referenced by the linker are either per ring items (wdir) in which case they can be moved like the search rules, or redundant copies of information already kept in outer rings (pit).

The case of active_hardcore_data is more difficult to handle. The present linker references the data base to do some metering of linking activity. Clearly if the linker is removed from ring 0, there is no way (and no reason) for it to save metering data in ring 0. Yet we would like to be able to keep metering data, as in the past, on a system wide basis, and it is impossible to do so in user rings as any user could overwrite, willfully or not, the metering data. One potential solution is to have the metering done by the signaller. On handling linkage faults, it could just increment counters and meter time and paging use of the linker.

In addition, some metering can be done by the linker itself on a per process basis. The problems of metering both in ring 0 and the user ring is currently being examined. No definite design is proposed yet. Suggestions are welcome.

2. Call problems

Not only does the current linker reference ring 0 data bases, but it also calls ring 0 procedures, which it won't be able to do. Again we had felt from the beginning on that this should cause no trouble: either the functions called by the linker could be accessible to user rings through gate or the linker does not really need to call the function. This feeling has also proved to be true: all of the primitives needed by the linker will indeed be accessible to it through hcs_, other primitives invoked by the present linker (e.g., kst_man) are not needed by the new proposed linker.

As a result, we do not need to add any entry to hcs_. Unfortunately, we must delete a lot of entries. It turns out that many (over 10%) of hcs_ entries are directed to some procedure to be included in the bound linker. Such entries will therefore become obsolete. Yet, there probably exists a large number of user ring procedures which call the soon to be obsolete gates. For compatibility we must keep these gates around and have them call the linker in the user ring. This poses an outward call problem. Actually the trouble we are facing has a much broader scope than our paper suggests. It is a problem any system programmer will encounter when trying to modify the ring 0 interface. The solution of the problem is to split hcs_ into two segments, one of which is a user ring transfer vector and is called hcs_. This transfer vector may direct some calls somewhere else in the calling ring (e.g., to our linker) or down to the second part of hcs_ called hcs_gate which is the actual gate segment into ring 0. We will give no more details about this feature as it was extensively discussed by M. D. Schroeder in MTB-024.

Finally, we discuss the case of calls to "all rings" procedures. Clearly, if a procedure called by the linker must have ring brackets (0,5,5), it cannot be part of the bound linker which has brackets (1,5,5). On the other hand, the reader will remember that during linker initialization, we snap links only in the linker linkage section. Hence the linker which is entirely prelinked might call "all rings" procedures which are not: this might lead to undesirable recursion in linkage faults. Fortunately, only two "all rings" procedures are called by the linker and a very elegant solution can be designed to avoid recursion in both cases. The linker calls object_info but uses only a very little bit of the information returned by object_info. Therefore, we can write a much simplified version of object_info and include it into the bound linker without duplicating too much code. Object_info

in turn calls no other "all rings" procedure. As to the second procedure called by the linker, area_, the solution is very simple. Area_ is called only when snapping type 6 links. But no link representing a symbolic reference in the tree of references generated from area_ is of type 6. Therefore, if we simply pre-link the linker to area_ and leave area_ with a virgin linkage section, all that could happen is taking a type 3 or 4 link fault on a virgin link of area_ while processing a type 6 link fault in a user program. This recursion is guaranteed to be of at most level two and is therefore authorized.

3. Search Rules

The present linker handles the search rules directories by means of pointers to them. When the linker tries to initiate some segment involved in a link pair, it does so by passing a directory pointer and an entryname to initiate. However, we will show later that the use of directory pointers may violate security and generality unless precautions are taken (see discussion later), therefore, we must designate directories by pathnames instead of by pointers.

Consequently search rules are a set of pathnames in the combined linkage segment of each ring. The default search rules are initialized in the combined linkage segment header during ring initialization. The search rules pointer in the header is set to point to them initially. When a user specifies his own set of search rules (up to 22 pathnames), the rules are stored in some available slot in the combined linkage segment of the current ring. The search rules pointer is set to point to them but the default search rules are never erased from the ls header. They can be restored as the current search rules at any time.

4. Hardware modification (2): The case of gates

We have already studeid and solved one of the problems raised by cross ring links where we developed the lot fault concept. Yet another problem is raised in the very same context of snapping a link to a gate into a lower ring. To translate the offset part of the symbolic reference, the linker has to search the definition section of the target gate. Bearing in mind the fact that the linker is running in the faulting ring, i.e., the high ring, it would not have read access to the gate segment on the present 6180 hardware. To give it access we simply suggest that the read access mode bit in the descriptor of a segment be associated with the R3 ring bracket instead of the R2 ring bracket. This will give the required access to the linker. Again the discussion of such a feature will be postponed for the moment.

V. Proposed Design: Tradeoff and Implications

In the above chapter we have outlined the design without arguing about anything. Of course lots of features have important implications on the behavior of the system and implied sometimes hard choices. We hope we have been able to guess what all the potential relevant questions of the reader may be and we will now try to answer them.

1. A system wide prelinking of the linker

At the time we are proposing to remove the linker from the security kernel, it may seem paradoxical to prelink it at system initialization in a protection environment which has the same capabilities as the security kernel has during system operation. The reason of this choice was a pure matter of economy. If the operation of prelinking had to be performed outside ring 0, it would have to be performed for each single process, once in each ring where the process goes. Since the result of the prelinking would always be the same, and since it is a "long" computation, doing it so often is clearly not desirable.

Moreover, doing the prelinking during system initialization allows us to use the standard prelinker thereby saving the trouble of having to keep some other prelinker around during system operation.

Notice that the system wide prelinking of the linker implies that any newly created process will know no other linker than the standard system provided one. But this prevents in no way the user to replink another linker of his own with the standard linker and to then discard the standard linker and replace it by his own (even reusing the same segment number).

A direct implication of the system wide prelinking of the linker is that the linker and any segment referenced by it will have the same segment numbers in the kst's of all processes.

We do not believe that this is a limitation of any sort on the generality of kst segment numbers. As we mentioned above the user can use the standard linker to prelink a linker of his own. Therefore he can just as well use the standard linker to prelink another copy of the standard linker using another segment number. Generality is restored for its fans!

2. A User ring initialization performed in ring 0

The reader may wonder why the only piece of the linking mechanism which has not been removed from ring 0 is the procedure that fabricates a combined linkage segment for a virgin ring.

There is no way in which it could be removed and we can prove this very simply. Fabricating a combined linkage segment requires (at least) one call to ring 0 to make this segment. This call implies the existence of a link to the appropriate gate. Obviously if the link is to be useful, it has to be in some linkage section of the combined linkage segment of the current ring. This is absurd since we are precisely trying to fabricate this segment!

Therefore the segment has to be fabricated in ring 0 for the user ring. The procedure used for this purpose is invoked by the procedure which fabricates the stack on the assumption that where a stack is fabricated, a combined linkage segment will most likely be needed.

3. Restricting the external references of the linker

We have talked about the case of the "all rings" procedure area_. We now would like to discuss in general what sort of references exactly the linker can use to avoid infinite recursive linkage faults.

Firstly, it can call any ring 0 gate as these are prelinked in ring 0. Secondly, it can reference any (non ring 0) data base as these have no linkage sections and yield no further symbolic references. Thirdly, we notice that the linker has several somehow distinct functions: managing linkage sections, snapping type 3 or 4 (external) links, snapping type 1 or 5 links (self referencing) and snapping type 6 (grow if not found) links. From this distinction, we can generalize the case of area_ by the following assertions:

- Any ("all rings") procedure called by the linker to manage linkage sections must be made part of the bound linker, otherwise trying to make a combined linkage section would yield other lot faults and go into endless recursion;
- The prelinking of any procedure called only to snap type 1 or 5 links can be avoided if it contains itself no type 1 or 5 links (which will be the case in general).
- The prelinking of any procedure called only to snap type 6 links can be avoided if it contains no type 6 links itself (again this is most likely to be the case in general).
- Any procedure called only to snap type 3 or 4 links must be part of the bound linker, unless it contains no type 3 or 4 links (which is not the case in general).

4. Pathnames vs. Segment Numbers for search rules

If search rules were segment numbers, both protection and generality would be violated. If we used segment numbers, the procedure which would initiate the search rules, and obviously would run in user rings, would need a gate into the supervisor which when given a pathname would return a segment number. But if such a gate existed, it could be used by other procedures to find out information about protected segments (e.g., does such a pathname exist or is the corresponding segment number within the bounds of segment numbers initiated by a given protected subsystem?) This is a violation of protection: although it could probably be monitored, it would add complexity to the kernel, which we are precisely trying to avoid.

Secondly, search rules being segment numbers would reflect the mapping of numbers to names at the time the search rules were initiated. Suppose a directory being part of the search rules were deleted during the life of a process. Then the remaining segment numbers would be inconsistent

and meaningless. Instead a pathname is always meaningful as such: it may or may not designate an existing directory, but it can always be tried as a search rule. Using segment numbers makes search rules dependent on the meaning, scope, extent and implementation of segment numbers, which is not general.

In an attempt to guess the questions of the reader, we have investigated the practical differences between directory pointers and pathnames. Although one may expect a lot ot be said due to the conceptual differences, the practical differences evaluate to nil. In the present system, when a search rule pointer points to a segment, or is inconsistent, it is simply not used. In our scheme, the corresponding pathname would be tried unsuccessfully. The net practical result is the same. Moreover, the segment number/directory pointer implementation contains several built in errors like the following for instance. If a search rule directory is terminated or deleted during the existence of a process, and another directory is initiated with the segment number of the previous one, this new directory would be searched as if it were the old one. This would not happen with pathnames of course.

It is now time to mention that the use of pathnames instead of pointers in the search rules is recommended only in a transient design phase. As a matter of fact, we mentioned that pathnames have to be used instead of pointers unless some precautions are taken. Because such precautions are not implemented in the present system, we cannot use pointers. However a a parallel project being worked on by R. G. Bratt is currently on its way towards implementing the required features. The goal of the project is to remove the mapping from pathnames to segment numbers from the kst and to store it on a per ring basis. As a result of this major design change, the search rules can and will again be segment numbers without lack of generality or security. In designing the new linker for the transient use of pathnames we will bear in mind the future use of segment numbers so as to simplify the modifications required later.

5. New hardware fault

There is not much to say about this topic, except that such a tool is really needed. Why should it be impossible to fault on a packed pointer set to all ones just like we fault on a regular pointer with some fault tag.

We have tried to avoid the use of a new fault, but all other solutions would have involved a cascade of difficulties and expensive software modifications: setting full pointers in the lot, using negative segment numbers in packed pointers, adding instructions to test packed pointers in entry sequences, etc., all such ideas brought up problems much more complex and expensive than the proposed hardware improvement can ever be.

6. Readable gates

We conclude this chapter by discussing the second hardware modification. Associating read access mode with the R3 bracket of a segment to calculate the access to the segment has actually no effect at all on non gate segments.

As a matter of fact, R2 is identical to R3, by definition, on non gate segments. Therefore associating the r bit with R3 instead of R2 has no effect on the protection achieved for such segments.

As far as gate segments are concerned, the effect of the modification is just to make them readable from outer rings where they use to be only callable from. Of course, subsystem programmers should be aware of this and not store secret information in the gates of their protected subsystems. We also feel that the modification is conceptually justified: a user running in the outside world should be entitled to have read access to gates into a protected subsystem it can call. Given that a gate is a door for the outside world, it seems conceptually reasonable to allow the outside world to see where that door is and how it looks like. What we are trying to protect is what is behind the gate and not the gate itself.

VI. Achievements of the design

1. Performance of the linker

As the goal of the design was not to improve the performance of the linker, we will not talk about performance improvement but instead, we will compare the performance of the proposed design to the performance of the present design.

To start with the comparison, we can simply say that except for a few name changes irrelevant to performance evaluation, the new linker is an identical, textual copy of the old linker. So far the performance of both should be identical. The only differences are due to the use of pathnames instead of pointers as search rules, and to the invocation of the signaller for each linkage fault.

Using pointers as search rules required initiating pathnames once and then using pointers directly for the search. Using pathnames also requires initiating these only once but every time they are used as search rules, there is a little extra overhead involved in ring 0 to retrieve the pointer corresponding to the pathname. The overhead evaluates to whatever amount of time is spent in kstsrch to translate the search rule into a pointer. It is essentially variable depending on how fast kstsrch can find the match and depending on how many search rules are tried for one link on the average. In the worst case, it could eventually reach 10% of the total processing time of the link. Of course, it is desirable to avoid such an overhead. As a matter of fact it will only exist during the transient period when pathnames are used. As soon as reference names are pulled out of ring 0 (see R. G. Bratt's project) the problem will disappear.

The overhead due to the signaller is of about 100 instructions per linkage fault, which is totally negligible w.r.t the 10,000 instructions a normal link needs to be snapped.

On the other hand, the new design will yield some performance improvement at the level of the linkage sections manager. The new version of the program is about 1/3 of the length of the present program. Moreover, it will not be invoked as often. In the present linker, it is invoked every time a segment is first referenced no matter whether the segment has a linkage section or not and no matter whether it will use it or not. In our design, the linkage section manager is invoked on a lot fault, at the very last moment, only where a linkage section is actually needed.

2. Protection and Certification

Let us now see what we have achieved in the design. Protection wise, if nothing else we have proved that the linker can run in user rings. This argument alone justifies that it should run in user rings according to the least privilege principle.

Certification wise, we may have reduced the size of the security by no more than 15,000 words out of 300,000 (8,000 text words out of 150,000) i.e., about 5%. But the complexity -- whatever measuring unit is used for that -- is incredibly reduced as the following figures show:

2500	very complex PL/I lines (handling lots of pointers)	
650	alm instructions	
30	entry points out of 1200	(2.5%)
15	procedures out of 300	(5%)
18	gates out of 160	(11%)

have been removed from the security kernel. Not only are the figures significant, but the code of the linker is a particularly complex and hard to audit one.

VII. Conclusions

We have given the motivations for proposing the project. We have proved it to be feasible. We have discussed its issues and achievements. We hope that the reader is convinced of the positive value of such a project on the way towards a certifiable secure system, even if certain details of the proposal may be incompletely settled or subject to minor design changes.