Inter-Office Memorandum

To	Cedar Interest	Date	October 28, 1982
From	Paul Rovner	Location	Palo Alto
Subject	The Cedar Runtime System	Organization	PARC/CSL

XEROX

Filed on: [Indigo]<Cedar>Documentation>SafeStorage.tioga

For novice usage of the Cedar system it is not necessary to read this memo or use the features described in it.

INTRODUCTION

This memo is intended to be used as reference material by Cedar implementors and advanced users. It documents the runtime support for Cedar language features that deal with storage management, universal references (REF ANY) and ATOMs.

Useful information for Cedar Wizards appears below in this font.

The interfaces described herein are all exported by RT.bcd, which is included in CedarCore.bcd, which is included in the Cedar boot file.

This document has the following sections:

1. Storage Management

The Cedar runtime system provides support both for automatic storage management and for explicit storage management in the Mesa 6.0 style. The programmer indicates which of these he wants when he uses the NEW expression by (perhaps implicitly) specifying a *storage zone* from which to get new storage space for a Cedar data value. *The maximum allowed storage space for a Cedar data value is 64K words.*

The Cedar storage management facility is built on top of the Pilot *Space* machinery. Normally, a client of Cedar will not explicitly use Pilot Spaces for data storage.

In Cedar, there are two types of storage zone: ZONEs and UNCOUNTED ZONEs. The characteristic that distinguishes ZONEs from UNCOUNTED ZONEs is the deallocation method: storage from a ZONE is deallocated automatically; storage from an UNCOUNTED ZONE is deallocated explicitly by the client program via the FREE statement.

If a user does not specify a storage zone parameter to NEW, storage will be allocated from a distinguished, predefined (collectible) ZONE, called SystemZone (more below).

It is often convenient and natural to use different storage zones or storage management strategies for different purposes. The **SafeStorage** and **UnsafeStorage** interfaces provide facilities for creating distinct zones and for choosing among several alternate storage management strategies.

Each zone created by either **SafeStorage.NewZone** or **UnsafeStorage.NewUZone** gets its storage in units of *storage quanta* from a single *Base*. A zone's Base is specified as a parameter when the zone is created (see the descriptions of NewZone and NewUZone, below).

A *Base* represents a contiguous region of virtual memory. Bases are used to define regions of virtual memory from which zones get their data quanta. This is useful both for circumscribing and for limiting the storage that can be acquired by zones. A Base may not be moved or extended after it is created.

For an application that does not require circumscribed regions of virtual address space, the programmer need not use Bases explicitly. If the programmer does use one of the procedures that takes a Base parameter, he may opt for the pre-defined RootBase, defined below, by defaulting the parameter.

A *storage quantum* is a contiguous block of storage cells in virtual memory. The quantum size is a compile-time parameter of the runtime system; the location of a quantum is determined when the quantum is acquired from its Base (at runtime).

Each zone represents a set of (not necessarily contiguous) storage quanta. At any given time a zone may have as few as zero quanta or as many quanta as its Base owns (see the descriptions of ExtendZone and ExtendUZone, below). The identification of a zone's Base is made when the zone is created and does not change thereafter.

The runtime system maintains a table, called the *quantum map*, that has a one word entry for each quantum in virtual memory. This table is indexed by the most significant bits of a virtual memory address. There will be one non-null entry in the quantum map for each quantum that has been acquired by a ZONE. Each quantum map entry contains enough information about a quantum to find the ZONE to which it belongs, among other things.

(Notational point: we will occasionally use the word *object* below to mean *contiguous words of storage space for a Cedar data value.*)

1.1 The **SafeStorage** Interface

This is a SAFE interface to the features of the Cedar storage management facility that allow the programmer to deal with Bases, (counted) ZONEs, and the reference-counting collector.

ZONE management

SizeRepresentation: TYPE = {quantized, prefixed};

This is an argument used when a new zone is created by NewZone or NewUZone to indicate how size information is to be represented for objects from the zone.

The quest for good performance motivates a choice of size representation: quantized zones are good if there will be many objects of the same size and type, or of a few different sizes and types. Prefixed zones are more general (e.g. for TEXT objects), but

their allocator is slower, each object requires extra storage cells to carry size and type information, and prefixed zones are susceptible to fragmentation problems.

The size and type of an object from a zone that has a prefixed size representation is carried in 2 prefix words of the object. For purposes of allocation efficiency and flexibility in prefixed zones, the size of an object may be bigger than is required for a value of the object's type.

Each quantum in a zone that has a quantized size representation supplies objects of only one size and type. The size and type of an object from such a quantum is determined from the information in the quantum map. Effectively, the quantum map entry points to a descriptor for all quanta in the zone that have the same size and type. This descriptor identifies the size and type. (Note that a quantized zone can supply storage for objects of different sizes and types. Each quantum in a quantized zone is uniform with respect to size and type, but different quanta in the zone may have different object size and type attributes).

```
NewZone: PROC
    [ sr: SizeRepresentation _ prefixed,
      base: Base _ nullBase,  -- default will use the RootBase
      initialSize: LONG CARDINAL _ 0  -- words
    ] RETURNS[ZONE];
```

ZONEs are collectible objects. A ZONE will be reclaimed automatically if no references to it remain accessible and if no references remain accessible to storage in any of its quanta.

```
TrimZone: PROC[zone: ZONE];
```

This merges adjacent blocks on **zone**'s free lists and then releases all quanta that do not contain allocated storage. Released quanta are returned to the zone's Base.

```
TrimAllZones: PROC;
```

This invokes **TrimZone** for each ZONE.

```
MergeAllPrefixedZones: PROC[zone: ZONE] RETURNS[BOOL];
```

For each prefixed ZONE, merge adjacent free blocks on its free list.

```
IsZoneEmpty: PROC[zone: ZONE] RETURNS[BOOL];
```

This returns TRUE if there are no quanta assigned to the specified zone.

```
GetSystemZone: PROC RETURNS[ZONE];
```

This returns a predefined, prefixed ZONE in the RootBase. This zone is used for NEW expressions in which the zone argument is elided.

Controlling the garbage collector

```
SetCollectionInterval: PROC[newInterval: LONG CARDINAL --words--]
    RETURNS[oldInterval: LONG CARDINAL];
```

The collector is automatically invoked whenever n collectible words have been allocated since the last collection. SetCollectionInterval allows the client to specify n.

Garbage collection will also be invoked automatically by the reference-counting machinery when its tables become full or when more quanta than are available are required from a Base. If the tables overflow during a collection then reference counting and the incremental collector will be disabled. The trace-and-sweep collector will be invoked at this time to perform a more leisurely and thorough collection (recovering circular

structures and inaccessible objects with pinned reference counts) and reconstruct the reference count table.

```
SetMaxDataQuanta: PROC[nQuanta: CARDINAL]
    RETURNS[CARDINAL];
```

This establishes the maximum number of storage quanta that the Cedar runtime system will request from the PILOT Space machinery. When an allocation request occurs that would cause this limit to be exceeded, a collection followed by

```
TrimAllZones[]; TrimRootBase[];
```

will occur and the allocation request will then be retried. SetMaxDataQuanta returns the previous value of this threshold. A threshold value of 0 means that no limit is enforced by the Cedar runtime system (this is the default setting).

```
ReclaimCollectibleObjects: PROC[suspendMe: BOOLEAN _ TRUE,
                                traceAndSweep: BOOLEAN _ FALSE];
```

This causes a collection to be initiated (return to the caller will be delayed until completion only if suspendMe = TRUE). When the collection finishes, a pass is made over the free list of each prefixed zone to merge adjacent free blocks. If traceAndSweep = TRUE then a TraceAndSweep collection will occur rather than an incremental one. A TraceAndSweep collection causes the world to wedge for roughly 30 seconds.

The Cedar "garbage collection" strategy is based on the maintenance of reference counts by an incremental collector that runs concurrently with client programs. The system will do a trace-and-sweep collection if the incremental collector runs into trouble, for example if its tables overflow during a collection. A compactifying collector is planned, but is not yet designed.

Monitoring the garbage collector

The procedures below are intended for use by a "collector watcher" process (see the example).

```
ReclamationReason: TYPE =
        {clientRequest, clientTAndSRequest, clientNoTraceRequest,
          rcTableOverflow, allocationInterval, quantaNeeded};

IsCollectorActive: PROC RETURNS[active: BOOLEAN,
                                previousIncarnation: CARDINAL];

WaitForCollectorStart: PROC
    RETURNS[incarnation: CARDINAL,
                --identifies the collection event
            reason: ReclamationReason,
            wordsAllocated: LONG CARDINAL,
              -- since previous collection was initiated
            objectsAllocated: LONG CARDINAL --ditto--];
```

This returns control sometime (soon) after the collector becomes active. Returns immediately if the collector is already active.

```
WaitForCollectorDone: PROC
    RETURNS[incarnation: CARDINAL,
            reason: ReclamationReason,
            wordsReclaimed: LONG CARDINAL,
            objectsReclaimed: LONG CARDINAL];
```

This returns control sometime (soon) after the collector ceases activity. Returns immediately if the collector is inactive.

Example of a "collector watcher" process:

```
...
CollectorWatcher: PROC =
 { DO
     ni, incarnation: CARDINAL;
     reason: ReclamationReason;
     wordsAllocated: LONG CARDINAL;
           --since prev collection start
     objectsAllocated: LONG CARDINAL;
     wordsReclaimed: LONG CARDINAL;
     objectsReclaimed: LONG CARDINAL;

     [incarnation, reason, wordsAllocated, objectsAllocated]
          _ WaitForCollectorStart[];
     IF stopPlease THEN {cwStopped _ TRUE; RETURN};
     WriteString[
          SELECT reason FROM
            clientRequest => "[clientRequest",
            clientTAndSRequest => "[clientTAndSRequest",
            clientNoTraceRequest => "[clientNoTraceRequest",
            rcTableOverflow => "[rcTableOverflow",
            allocationInterval => "[allocationInterval",
            quantaNeeded => "[quantaNeeded",
           ENDCASE => ERROR];
     WriteString[" Collection initiated after allocating "];
     WriteLongOctal[wordsAllocated];
     WriteString[" words, "];
     WriteLongOctal[objectsAllocated];
     WriteLine[" objects]"];
     [ni, , wordsReclaimed, objectsReclaimed]
          _ WaitForCollectorDone[];
     IF stopPlease THEN {cwStopped _ TRUE; RETURN};
     IF ni # incarnation THEN

WriteLine["<missed collection end, another intervened:>"];
     WriteString["["];
     WriteString[" Collection finished, "];
     WriteLongOctal[wordsReclaimed];
     WriteString[" words reclaimed, "];
     WriteLongOctal[objectsReclaimed];
     WriteLine[" objects reclaimed]"]
    ENDLOOP};

...
Process.Detach[FORK CollectorWatcher[]];
```

Statistics

NWordsAllocated: PROC RETURNS[LONG CARDINAL];

This returns the number of words of collectible storage that have been requested since the beginning of time.

NWordsReclaimed: PROC RETURNS[LONG CARDINAL];

This returns the number of words of collectible storage that have been reclaimed since the beginning of time.

Bases

RTBasic. Base: TYPE ...

A **Base** represents a contiguous region of virtual memory. Bases are used to define the origin for relative reference types (which are not implemented yet) and to define regions of virtual memory from which zones get their data quanta. This latter property is useful both for circumscribing and for limiting the storage that can be acquired by zones. A Base may not be moved or extended after it is created.

GetRootBase: PROC RETURNS[Base];

This predefined Base represents the entire address space, not including the MDS. It is the default Base for creating new Bases and zones.

NewBase: PROC[size: LONG CARDINAL,
baseParent: Base _ nullBase] RETURNS[Base];

size is in words, rounded to the nearest number of storage quanta.

A *quantum* is a section of the address space that includes addresses [j*Q..(j+1)*Q) where Q is a fixed value called the *quantum size* (currently 4 pages) and j is called a *quantum index*. A quantum is the smallest amount of address space that a Base can occupy.

The new Base obtains its storage quanta from its parent Base. Use of the **nullBase** default for the **baseParent** argument causes RootBase to be used as the parent.

Bases are collectible objects. A Base will be reclaimed automatically if no references to it remain accessible (this can be true only if there are no zones in it). Its quanta will be returned to its parent Base.

TrimRootBase: PROC RETURNS[nSpacesDeleted: CARDINAL];

TrimRootBase attempts to find and delete Pilot spaces that are assigned to the RootBase but from which no storage quanta are currently assigned to any ZONE. After a collection finishes, the collector invokes TrimAllZones[] followed by TrimRootBase[] if the collection was initiated because virtual space is exhausted.

Unusual operations on ZONEs for sophisticated clients

ZoneFullProc: TYPE = PROC[zone: ZONE, size: LONG CARDINAL];

ExtendZone: ZoneFullProc;

This adds storage quanta to a ZONE. The quanta come from **zone**'s Base. **size** is a number of words to be added, and is rounded up to the next multiple of the quantum size. If the Base cannot supply the needed quanta, a garbage collection followed by TrimAllZones (see below) will occur.

SetZoneFullProc: PROC
[zone: ZONE,
proc: ZoneFullProc
] RETURNS[oldProc: ZoneFullProc];

When the allocator finds that inadequate space is left in the specified ZONE to satisfy a new request, it calls the ZoneFullProc for the ZONE, and then re-attempts the allocation. The initial (default) ZoneFullProc for every ZONE is ExtendZone. SetZoneFullProc establishes **proc** as **zone**'s ZoneFullProc.

SIGNALs and ERRORs

InvalidSize: ERROR[size: LONG CARDINAL];

Raised by the allocator and by NewBase, NewZone, and ExtendZone if their size argument is too big.

MemoryExhausted: ERROR[base: Base];

Raised by the allocator and by ExtendZone and ExtendUZone.

NarrowRefFault: ERROR[ref: REF ANY, targetType: Type];

Raised by NARROW.

NarrowFault: ERROR;

Raised by NARROW.

UnsafeProcAssignment: SIGNAL[proc: PROC ANY RETURNS ANY];

1.2 The **UnsafeStorage** Interface

This is an UNSAFE interface to features that allow the programmer to deal with UNCOUNTED ZONEs.

```
NewUObject: PROC
    [ size: CARDINAL,  -- words
      zone: UNCOUNTED ZONE,
      type: RTTypesBasic.Type _ RTTypesBasic.nullType
    ] RETURNS [LONG POINTER];
```

This is a useful alternative to the Mesa NEW expression for allocating objects from an UNCOUNTED ZONE that carry their types.

```
GetHeapReferentType: PROC[ptr: LONG POINTER]
    RETURNS[type: Type];
```

This returns the Type that is associated with the specified object. THIS IS UNSAFE. It is intended only for objects that have been allocated from an UNCOUNTED ZONE that was created by NewUZone with typeRepresentation = TRUE. If **ptr** = NIL, GetHeapReferentType returns nullType.

```
NewUZone: PROC
    [ initialSize: LONG CARDINAL _ 0  -- words
      sr: SizeRepresentation _ prefixed,
      typeRepresentation: BOOL _ FALSE
    ] RETURNS[UNCOUNTED ZONE];
```

NewUZone creates a new UNCOUNTED ZONE that will get its quanta from the RootBase.

FreeUZone: PROC[uz: UNCOUNTED ZONE];

FreeUZone explicitly deallocates **uz**, which is assumed to be an object that was created by **NewUZone** .

ExtendUZone: UZoneFullProc;

ExtendUZone adds quanta to an UNCOUNTED ZONE that was created by NewUZone. The quanta come from the RootBase. **size** is a number of words that is rounded up to the next multiple of the quantum size. A garbage collection followed by TrimAllZones is invoked if RootBase cannot supply the needed quanta.

UZoneFullProc: TYPE = PROC[zone: UNCOUNTED ZONE, size: LONG

CARDINAL];

SetUZoneFullProc: PROC
[zone: UNCOUNTED ZONE,
proc: UZoneFullProc
] RETURNS[oldProc: UZoneFullProc];

When **zone**'s allocator finds that there is inadequate storage space to satisfy a new request, it calls **zone**'s UZoneFullProc and then re-attempts the allocation. The initial (default) UZoneFullProc for every UNCOUNTED ZONE is ExtendUZone. SetUZoneFullProc establishes the specified procedure as the specified UNCOUNTED ZONE's UZoneFullProc.

MergeUZone: PROC[zone: UNCOUNTED ZONE];

MergeUZone merges **zone**'s free lists. For prefixed UNCOUNTED ZONEs.

TrimUZone: PROC[zone: UNCOUNTED ZONE];

TrimUZone merges **zone**'s free lists and then releases all quanta that do not contain allocated objects. Released quanta are returned to the operating system.

IsUZoneEmpty: PROC[zone: UNCOUNTED ZONE] RETURNS[BOOL];

IsUZoneEmpty returns TRUE if there are no quanta assigned to the specified zone.

GetSystemUZone: PROC RETURNS[UNCOUNTED ZONE];

This returns a predefined, prefixed UNCOUNTED ZONE that will get its quanta from the RootBase. It has no type representation.

InvalidPointer: ERROR[ptr: LONG POINTER];

Raised by FREE.

2. Universal References and Runtime Types: The **RTTypesBasic** Interface

One of the "delayed binding" features of safe Cedar is the ability to manipulate general references without either compile-time knowledge of the referent type or the use of a LOOPHOLE to specify the referent type when it comes time to de-reference. Such references are termed "universal references" and have type REF ANY. The Cedar language facilities (REF ANY, NARROW, WITH ... SELECT) for dealing with universal references are supported at runtime by a system that can allocate an object of a given type, determine the referent type of a specified universal reference, and determine whether two given types are equivalent.

In addition to such basic support for compiled programs, the Cedar runtime system provides an implementation for the RTTypesBasic interface described below.

Another client of the runtime type system is the garbage collector, which must be able to determine for each object in memory where within the storage for that object there are references to other objects. For this purpose, the runtime type system maintains a data structure that represents this information (called a "reference containing map") for each type for which a collectible object has been created.

2.1 Types

Type: TYPE = RTBasic.Type

A **Type** is a datum that represents a Cedar TYPE at runtime. Conceptually, every collectible object includes a **Type** representing the TYPE of the data value. This is stored with the object.

The Cedar NEW expression automatically attaches the appropriate **Type** to the newly created object. The NEW expression is the only way in the language to create a collectible object.

A Cedar expression of the form CODE[<<type expression>>] returns a **Type** for the specified TYPE. (NOTE: this is an interim syntax. There will be better syntax in Cedar for acquiring Types from TYPE expressions).

nullType: Type = ...

By convention, this is a distinguished, predefined Type that is the Type of the referent of NIL. Other distinguished, predefined Types are listed below:

```
unspecType   -- the distinguished type of UNSPECIFIED
fhType       -- the distinguished type of localFrames
gfhType      -- the distinguished type of globalFrames
```

GetReferentType: PROC[ref: REF ANY] RETURNS[type: Type];

This returns the Type that is carried by the specified object. If ref = NIL, GetReferentType returns nullType.

GetCanonicalType: PROC[type: Type] RETURNS[Type];

This computes the *canonical Type* for the specified Type. Each Type has a canonical Type. Canonical Types are used for testing Type equivalence efficiently (if t1 and t2 are canonical Types, then t1=t2 IFF t1 and t2 represent equivalent TYPEs). Under normal circumstances, the Cedar client will not use this function. Note that GetCanonicalType[nullType] = nullType.

GetCanonicalReferentType: PROC[ref: REF ANY]
RETURNS[type: Type];

This is included for convenience. It returns
GetCanonicalType[GetReferentType[ref]] .

EquivalentTypes: PROC[t1, t2: Type]
RETURNS[BOOL];

This is included for convenience. It returns
GetCanonicalType[t1] = GetCanonicalType[t2]
This will be TRUE if the corresponding Cedar TYPEs are equivalent..

IsReferentType: PROC[ref: REF ANY, type: Type]
RETURNS[BOOL];

This is included for convenience. It returns
(EquivalentTypes[GetReferentType[ref], type])

NarrowRef: PROC[ref: REF ANY, type: Type] RETURNS[REF ANY];

This is useful for checking, and is included for convenience.
If ref = NIL, NarrowRef returns NIL.
Otherwise, NarrowRef returns ref if
GetCanonicalType[type] = GetCanonicalReferentType [ref]
or raises SafeStorage.NarrowRefFault if not.

2.2 Type attachments

PutTypeAttachment: PROC[type: Type, attachment: REF ANY];

GetTypeAttachment: PROC [type: Type] RETURNS[REF ANY];

These provide a way to associate an arbitrary object with a specified Type, for example a user-defined print procedure. This facility is viewed as an interim one until there are appropriate language features for useful applications .

2.3 Finalization of collectible objects

It is often convenient to specify actions to take when an object of a particular Type is no longer accessible to any client of the package that created it. Such *package finalization* actions might include (for example) removal of a reference to the object from a package-maintained cache.

A built-in facility for object finalization is more than a convenience. In applications where a package must maintain copies of the object handles that it gives to clients, the alternatives to package finalization are expensive, and duplicate in one way or another the storage management functions that the collector provides. The Cedar finalization mechanism is specifically tailored to such applications and to the requirements of efficiency and safety. Though one can imagine more general facilities for finalization, we believe that the features described below are a powerful and useful set. They are inexpensive and SAFE.

BEWARE: Until appropriate language changes are made, it is necessary to establish finalization for a Type BEFORE allocating any objects of the Type. Allocate objects of the Type ONLY in the program module that makes the calls to establish finalization. Use the exact same TYPE expression in the finalization-establishing calls and in NEW expressions for objects of the Type.

Procedures

```
EstablishFinalization: PROC
    [type: Type,
     npr: [1..maxNPackageRefs],
     fq: FinalizationQueue
    ];
```

type is a Type for which finalization actions are to be enabled. **fq** is a queue on which the collector will put REFs to finalizable objects of that Type. Note that finalization is a property of the object Type, not of the REF Type.

The collector puts a REF to an object on the queue for its Type when it finds that exactly **npr** references remain to the object in other objects. In other words, **npr** is the number of references that are considered "package references" for this purpose. Note that there must be at least one package reference for a finalizable Type.

Small points: (1) If the finalization queue is full, the collector will try again next time a collection occurs. (2) References on the stack (i.e. in local variables of a procedure) are not included in the counting to determine whether an object should be finalized. References in global frames (i.e. in program variables) ARE included in this counting. If any references to the object appear on the stack, the object will not be finalized.

Typically, a client will FORK a process that invokes FQNext to dequeue REFs from **fq** and then performs the desired finalization actions. FQNext will wait until **fq** is non-empty, so this process will be inactive most of the time. The user procedure that invokes FQNext should be external to the monitor that protects the data structure being finalized. This procedure should then call an ENTRY procedure to do the finalization actions. See the example below.

```
ReEstablishFinalization: PROC
    [type: Type,
     npr: [1..maxNPackageRefs],
     fq: FinalizationQueue
    ];
```

This is useful when (for example) a new version of a program that establishes

finalization for the specified type is loaded and started in the same environment in which the old one ran. Generally, of course, there may be other problems that preclude forward progress in this situation, but this procedure is useful for the particular problem of package finalization. In such a situation, the program can catch **CantEstablishFinalization**, unwind, and then invoke **ReEstablishFinalization**. The signal will be raised anew if no finalization has been specified for type or if npr is different. The effect of succesful execution is to replace the new fq for the old. Subsequently, objects found by the collector to be finalizable will be put on the new fq.

maxNPackageRefs: CARDINAL = 2;

FinalizationQueue: TYPE ...

NewFQ: PROC[length: CARDINAL _ 10] RETURNS[FinalizationQueue];

Creates a new FinalizationQueue. **length** specifies the maximum number of REFs that it can hold.

FQNext: PROC[fq: FinalizationQueue] RETURNS[REF ANY];

dequeue the next REF to be finalized from **fq**. Wait (on a condition variable that is NOTIFY'd when the collector queues a REF on fq) if fq is empty.

FQEmpty: PROC[fq: FinalizationQueue] RETURNS[BOOL];

Returns TRUE if fq is empty.

Sample program fragment

```
FooObject: TYPE = ...
fooFQ: FinalizationQueue = NewFQ[];
...
FinalizerProcess: PROC[fooFQ: FinalizationQueue] =
 { DO
     foo: REF FooObject = NARROW[FQNext[fooFQ]];
     FinalizeFooObject[foo];
         -- FinalizeFooObject is an ENTRY procedure
         -- that will nilify foo's package REF
    ENDLOOP};
...
EstablishFinalization[CODE[FooObject], 1, fooFQ];
Process.Detach[FORK FinalizerProcess[fooFQ]];
...
...NEW[FooObject...
```

2.4 SIGNALs and ERRORs

RTTypesBasic. InvalidType: ERROR[type: Type];

RTTypesBasic. CantEstablishFinalization: ERROR[type: Type];

Raised by EstablishFinalization.

3. ATOMs: The **AtomsPrivate** Interface

This is the lowest layer of Cedar runtime support for ATOMs. Other documents, notably the language manual and the Atom and List interfaces, are more relevant than this one as reference material for conventional usage of ATOMs in Cedar.

ATOM is a predefined REF type in Cedar. This means that an ATOM can be widened to a REF ANY (i.e. passed as a REF ANY parameter or assigned to a REF ANY variable), that

ATOM can appear as the TYPE expression in an arm of a
WITH <<REF ANY>> SELECT
construct, and that NIL can be assigned to an ATOM variable or passed as an ATOM parameter.

The Cedar runtime system maintains an ATOM dictionary. This is a name space that is global to the current Cedar execution. It is used as a unique map between printnames and ATOMs.

The compiler recognizes as an ATOM constant any identifier that has $ as its first character. ATOMs are acquired from the ATOM dictionary for such constants by the Cedar runtime loader.

The AtomsPrivate interface provides the following operations:

GetAtom: PROC[pName: Rope.ROPE]
RETURNS[ATOM];

GetAtom is used to acquire the unique ATOM with the specified printname. A new ATOM will be created if necessary.

UnsafeMakeAtom: PROC[pName: LONG POINTER TO READONLY TEXT]
RETURNS[ATOM];

Like GetAtom, but unsafe. Included for convenience.

EnumerateAtoms: PROC[callee: PROC[ATOM] RETURNS[stop: BOOL]]
RETURNS[ATOM];

EnumerateAtoms may be used to enumerate the ATOMs that are known to the runtime system. It will invoke the specified procedure ("callee") exactly once for each ATOM. EnumerateAtoms will return either when an invocation of callee returns TRUE or when there are no more ATOMs to enumerate. In the former case, the value returned by EnumerateAtoms will be the ATOM that was passed to callee. In the latter case, the value returned by EnumerateAtoms will be NIL. Never fear: no monitor lock is held by EnumerateAtoms when **callee** is invoked.

While the call on EnumerateAtoms is active, ATOMs that are created either by explicit calls on MakeAtom or by loading a program module that defines new ATOM constants may or may not be included in the enumeration.

An ATOM value can be LOOPHOLEd to a REF AtomsPrivate.AtomRec, which is defined as follows:

```
AtomRec: TYPE =
  RECORD[ pName: Rope.Text, -- the arg to GetAtom
          propList: REF ANY _ NIL,
          link: ATOM _ NIL        -- used by GetAtom
        ];
```

4. Configuration Parameters

The current storage manager uses 1024-word quanta (4 pages), a 22-bit address space, and requires roughly 65K words of storage for its internal data structures, which include the reference count table, the quantum map, and various auxilliary tables to support runtime types. Only one page of this data is pinned. The code is designed to work with any quantum size that is a power of 2 between $2^{\wedge 8}$ and $2^{\wedge 15}$, and with an address space of any size between 16 and 28 bits. Changing any of these parameters requires modification of the runtime system and the Cedar microcode, but does not require client recompilation.