Pilot

Volume 1 of 2

Date: October 1980
Version: 5.0

XEROX
Office Products Division
System Development Department
Palo Alto, California

This document is for internal Xerox use only.

Pilot 5.0 (Mokelumne)

Mesa Defs

Long>BcdDefs
Long>BcdOps ←
LongBcdOps
MiscAlpha
Mopcodes
PrincOps
SDDefs
Long>Table[
Bases: LongBases]
TimeStamp

Public Definitions

. *BitBlt*
. *ByteBlt*
Checksum
Device
. *DeviceTypes*
Dialup
Environment
.. *File*
... *FileTypes*
..... *FloppyChannel*
.... *Heap*
. *Inline*
. *JLevelIVKeys*
. *Keys*
KeyStations
. *LevelIIIKeys*
. *LevelIVKeys*
.. *NetworkStream*
.... *PhoneNetwork*
.... *PhysicalVolume*
PilotClient
..... *PilotSwitches*
Process
.. *RS232C*
.. *RS232CCorrespondents*
. *RS232CEnvironment*
... *RS232CManager*
... *Runtime*
.... *Scavenger*
. *Socket*
... *Space*
. *Stream*
... *SubSys*
System
.... *TemporaryBooting*
TemporaryTransaction
TextBlt
Transaction
TTYPortEnvironment
.... *UserTerminal*
... *Volume*
.... *VolumeExtras*
. *Zone*

Special Defs

. *BootFile*
... *CountPrivate*
CPSwapDefs
Frame
PageMap
... *PerfPrivate*
.. *PerfStructures*
PilotLoadStateFormat
. . *PilotLoadStateOps*
PrincOpsRuntime
. *ProcessOperations*
PSB
SpecialSystem
StartList
Trap
VMMapLog

Communications Defs

. *BufferDefs*
CommUtilDefs
.. *DriverDefs*
DriverTypes
... *OISCPDefs*
OISCPTypes
.. *PupDefs*
PupTypes
. *SpecialCommunication*
StatsDefs
. *StatsOps*
. *TeledebugProtocol*

Faces

DisplayFace
EthernetFace
EthernetOneFace
KeyboardFace
MouseFace
PilotDisk
ProcessorFace
RS232CFace
RS366Face
. *SA4000Face*
. *SA800Face*
TTYPortFace

TestPilot **

Boot
. *BootSwap*
. *DebuggerSwap*
DeviceCleanup
DiskChannel
DriverStartChain
FilePageLabel
HeadStartChain
OISTransporter
PilotFileTypes
PilotMP
ProcessInternal
ProcessPriorities
ResidentHeap
RuntimeInternal
Snapshot
SoftwareTextBlt
SpecialFile
SpecialHeap
SpecialSpace
SpecialTransaction
. *SpecialVolume*
StartChainPlug
StoreDriverStartChain
TemporarySetGMT
Utilities

TestPilotKernel

CommunicationPrograms
KernelFile
MiscPrograms
PerformancePrograms
ResidentMemory
RuntimePrograms
StoragePrograms
SystemInternal
XferTrap

Control

ControlPrograms

- PilotControl
- SystemImpl

Store

.. *CachedRegion*
... *CachedRegionInternal*
. *CachedSpace*
.. *FileCache*
. *FileInternal*
.. *FilePageTransfer*
... *FilerException*
KernelSpace
.. *LabelTransfer*
. *MStore*
PhysicalVolumeFormat
.. *SimpleSpace*
.. *SubVolume*
... *SwapperException*
TransactionState
VM
VolumeInternal

FileMgr

FMPrograms
LogicalVolume
. *MarkerPage*
. *VolAllocMap*
. *VolFileMap*
. *VolumeImplInterface*

- FileImpl
- MarkerPageImpl
- PhysicalVolumeImpl
- ScavengeImpl
- VolAllocMapImpl
- VolFileMapImpl
- VolumeImpl

Filer

FilerPrograms
FileTask
RemotePageTransfer

- FileCacheImpl
- FilerControl
- FilerExceptionImpl
- FilerTransferImpl
- FileTaskImpl
- SubVolumeImpl

Swapper

PageFault
SwapBuffer
SwapperPrograms
SwapTask

- CachedRegionImplA
- CachedRegionImplB
- CachedSpaceImpl
- MStoreImpl
- PageFaultImpl
- ResidentMemoryImpl
- SimpleSpaceImpl
- SwapBufferImpl
- SwapperControl
- SwapperExceptionImpl
- SwapTaskImpl

Transactions

TransactionInternal

- TransactionImpl
- TransactionLogImpl
- TransactionStateImpl

VMMgr

Hierarchy
MapLog
Projection
SpaceImplInternal
STLeaf
. *STree*
VMMgrStore
VMMPrograms

- HierarchyImpl
- MapLogImpl
- ProjectionImpl
- SpaceImplA
- SpaceImplB
- STLeafImpl
- STreeImpl
- VMMControl

MesaRuntime

- BootSwapCross
- FrameImpl
- Instructions
- PilotNub
- Processes
- Signals
- SnapshotImpl
- Traps

RuntimeLoader

PilotLoaderOps

- BcdOperations #
- PilotLoaderCore
- PilotLoaderSupport
- PilotLoadState
- PilotUnloader

RuntimePerf

- PilotCounter
- PilotPerfMonitor

Misc

ZoneInternal

- HeapImpl
- ResidentHeapImpl
- StreamImpl
- UtilitiesImpl
- ZoneImpl

Communication

Checksums
CommunicationInternal
Echo
HalfDuplex
. *NetworkStreamInternal*
PacketStream
. *Router*
SocketInternal

Level0

- DispatcherImpl
- EthernetDriver
- EthernetOneDriver
- HalfDuplexImpl
- PhoneNetworkDriver
- PhoneNetworkImpl

Level1

- ChecksumsImpl
- RouterImpl
- RoutingTableImpl
- SocketImpl

Level2

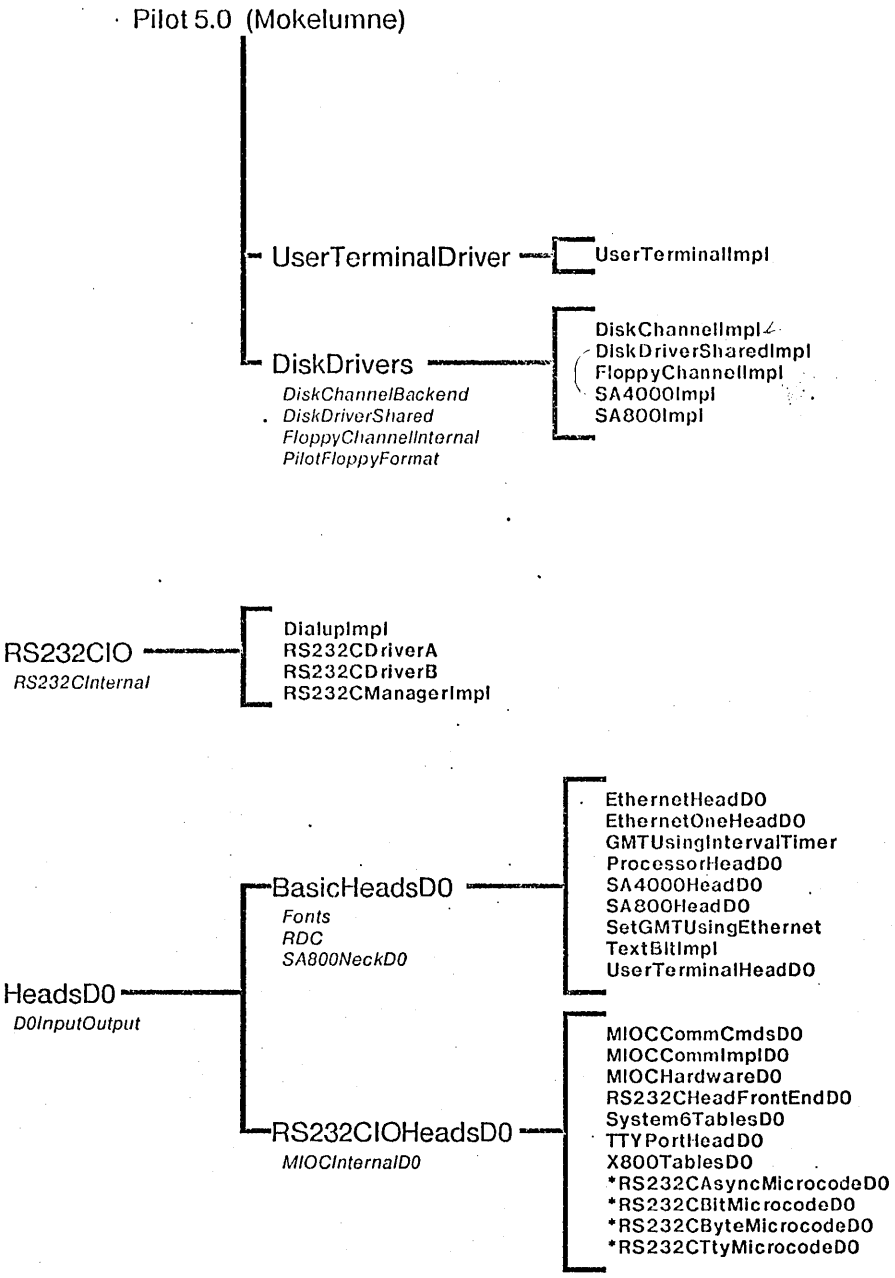
- EchoServerImpl
- NetworkStreamInstance
. NetworkStreamMgr
- PacketStreamInstance
. PacketStreamMgr

QueuePackage

- BufferPoolImpl
- QueueImpl
- SystemBufferPoolImpl

Boss
. CommunicationControl
PilotCommUtil
StatsHot

[Idun]<WPilot>Pilot1.sil



Large bold typeface: Configuration

Small plain typeface: Module

Small italic typeface: Interface/Definitions - scope is anything to the right in the configuration with which it appears

Dots indicate compilation order within levels: no-dots, then one-dots, etc.

* indicates microcode overlay.

** indicates packaged: Packager.bcd TestPilot[UnpackedTestPilot]/mb

# indicates recompiled Mesa-System source.

[], ← indicate command line compiler/binder options

```
-- Packed PilotDO.bootmesa (last edited by Luniewski on October 16, 1980  2:39 PM)

-- All RESIDENTDESCRIPTOR modules must also be IN!  (An implementation restriction of PilotControl.)
-- WARNING: If a procedure is RESIDENT, the entry vector for its containing module must also be RESIDEN
**T.  If a procedure is RESIDENTDESCRIPTOR, the entry vector for its containing module must be RESIDENT
**DESCRIPTOR or RESIDENT.  Finally, if a procedure is IN, the entry vector for its containing module mu
**st be IN or RESIDENT.

WART: PilotControl;

RESIDENT:
  BasicHeadsDO[EthernetOneHeadDO, EthernetHeadDO, GMTUsingIntervalTimer, ProcessorHeadDO,
          SA4000HeadDO, SA800HeadDO, UserTerminalHeadDO, SetGMTUsingEthernet,
          StartChainPlug],
  SPACE [Resident],
  FRAME [ResidentFrames];

RESIDENTDESCRIPTOR:
  FRAME [RandomCoolFrames],
  SPACE [SwappableSwapper],                    -- Yes, parts of the Swapper ARE swappable
  SPACE [FileAttributes, FileCreate, FileDelete, FileHelperProcess,
           Othello, OtherFileStuff, PhysicalVolume, Scavenger,
           VFMChangingSize, Volume],           --FileMgr
  SPACE [SpaceCodeHacking, SpaceCopying, SpaceCreate, SpaceDelete,
           SpaceMap, SpaceMisc, SpaceRemap,
           SpaceStatusOps, SpaceUnmap, VMMgrCommon, VMMHelperProcess], -- VMMgr
  SPACE [Frames, ProcessesHot, ProcessesCold],   -- MesaRuntime [FrameImpl, Processes]
  SPACE [SwappableSystem],                     -- Control [SystemImpl]
  SPACE [Abort, TransactionsCommon, TransactionsRunning];

IN:
  -- The following must be IN since they are RESIDENTDESCRIPTOR
  FRAME [RandomCoolFrames],
  SPACE [SwappableSwapper],                     -- Yes, parts of the Swapper ARE swappable
  SPACE [FileAttributes, FileCreate, FileDelete, FileHelperProcess,
           Othello, OtherFileStuff, PhysicalVolume, Scavenger,
           VFMChangingSize, Volume],            -- FileMgr
  SPACE [SpaceCodeHacking, SpaceCopying, SpaceCreate, SpaceDelete,
           SpaceMap, SpaceMisc, SpaceRemap,
           SpaceStatusOps, SpaceUnmap, VMMgrCommon, VMMHelperProcess], -- VMMgr
  SPACE [Frames, ProcessesHot, ProcessesCold],   -- MesaRuntime [FrameImpl, Processes]
  SPACE [SwappableSystem],                     -- Control [SystemImpl]
  SPACE [SwappableDiskDrivers], -- they make themselves resident when appropriate
  SPACE [Abort, TransactionsCommon, TransactionsRunning],
  -- The following are IN due to initialization requirements
  SPACE [CreateDelete],                        -- For ResidentHeap initialization
  SPACE [FileMgrInitialization],
  SPACE [ResidentInitialization],
  SPACE [TransactionsInitialization],
  SPACE [VMMgrInitialization];

NOTRAP: PilotControl, Traps;

GFT: 512;
MDSBASE: 0B;
PDABASE: 400B;
PDAPAGES: 2;
CODEBASE: 400B;
```

```
-- UnpackedPilotD0.bootmesa  (last edited by: Forrest on: October 4, 1980  3:46 PM)

-- Note:   This version is for the D0 only!  (Dlion version has different Heads.)

-- Note:   All RESIDENTDESCRIPTOR modules must also be IN!  (An implementation restriction of PilotCont
**rol.)

-- Anything called from inside the FileImpl / VolumeImpl / SpaceImpl monitors must be Resident or Resid
**entDescriptor because a pageFault would lock out the File Helper or VM Helper.

-- In the text below, configurations are in alphabetical order;  within a config, alphabetically by mod
**ule or instance.

WART: PilotControl;

RESIDENT:
  BasicHeadsD0[EthernetHeadD0, EthernetOneHeadD0, GMTUsingIntervalTimer,
        ProcessorHeadD0, SA4000HeadD0, SA800HeadD0, UserTerminalHeadD0,
        SetGMTUsingEthernet, StartChainPlug],
  Control[SystemImpl],  -- Has a cleanup procedure.
  DiskDrivers[DiskChannelImpl, StartChainPlug, DiskDriverSharedImpl, SA4000Impl],
  Filer[FileCacheImpl, FilerControl, FilerExceptionImpl, FilerTransferImpl,
        FileTaskImpl, SubVolumeImpl],
  -- Processes is temporarily resident until the replacement process is fixed
  MesaRuntime[BootSwapCross, Instructions, PilotNub, Processes, Signals, Traps],
  Misc[ResidentHeapImpl, StreamImpl, UtilitiesImpl, ZoneImpl],
  RuntimePerf[PilotCounter, PilotPerfMonitor],
  Swapper[CachedRegionImplA, CachedRegionImplB, CachedSpaceImpl,
        MStoreImpl, PageFaultImpl, ResidentMemoryImpl, SimpleSpaceImpl,
        SwapBufferImpl, SwapperControl, SwapperExceptionImpl,
        SwapTaskImpl],
  UserTerminal[UserTerminalImpl];

RESIDENTDESCRIPTOR:
  FileMgr[FileImpl, PhysicalVolumeImpl, VolAllocMapImpl, VolFileMapImpl, VolumeImpl,
          MarkerPageImpl, ScavengeImpl],
  Runtime[FrameImpl],
  RuntimePerf[PilotPerfMonitor],
  Transactions[TransactionImpl, TransactionLogImpl],  -- may be called from
                              -- inside the FileImpl monitor.
  VMMgr[MapLogImpl], HConfig[HierarchyImpl], PConfig[ProjectionImpl],
        VMMgr[SpaceImplA],  -- contains VM Helper process.
        -- SpaceImplB is not necessary for VM Helper process. ← WRONG!
        VVMMgr[STLeafImpl], HConfig[STreeImpl];

IN:
  Control[PilotControl],
  FileMgr[FileImpl, PhysicalVolumeImpl, VolAllocMapImpl, VolFileMapImpl, VolumeImpl,
        MarkerPageImpl, ScavengeImpl],
  Runtime[FrameImpl],
  RuntimePerf[PilotPerfMonitor],
  Transactions[TransactionImpl, TransactionLogImpl],
  VMMgr[MapLogImpl], HConfig[HierarchyImpl], PConfig[ProjectionImpl],
        VMMgr[SpaceImplA],
        VMMgr[SpaceImplB],  -- not necessary for VM Helper process.
        VMMgr[ STLeafImpl], HConfig[STreeImpl],
        VMMgr[VMMControl];  -- not necessary for VM Helper process.

NOTRAP: PilotControl, Traps;

GFT: 512;
MDSBASE: 0B;
PDABASE: 400B;
PDAPAGES: 2;
CODEBASE: 400B;


--       Change Log:
-- Knutsen   Aug 13, 1980 11:24 AM   Made PilotControl an IN; deleted TransactionStateImpl; SpaceImplB
**didn't need to be ResDesc
-- Jose/Knutsen   August 14, 1980  3:54 PM   Added SA800HeadD0 to HeadsD0;  VMMControl didn't need to b
**e ResDesc
-- McJones   August 19, 1980  5:04 PM   Renamed from Pilot.bootmesa; HeadsD0=>BasicHeadsD0
-- Luniewski August 22, 1980  4:49 PM  Made SystemImpl (temporarily?) resident
-- Luniewski August 27, 1980  9:10 AM Made Processes (temporarily?) resident
-- BLyon August 28, 1980  9:29 AM Renamed Ethernet* to EthernetOne*
```

```
-- McJones August 29, 1980  3:09 PM Deleted InterruptKey
-- HGM September 9, 1980  1:19 AM Add EthernetHead
-- McJones September 18, 1980  11:27 AM Changed PDABASE to 400B, PDAPAGES to 2, and GFT to 512
-- Forrest September 20, 1980  3:48 PM made from DO Version
```

```
-- UtilityPilot.bootmesa  (last edited by: McJones on: September 17, 1980  1:08 PM)

WART: PilotControl;
RESIDENT: ALL;
NOTRAP: PilotControl, Traps;

GFT: 256;
MDSBASE: 0B;
PDABASE: 400B;
PDAPAGES: 1;
CODEBASE: 400B;

-- Knutsen Jul 2, 1980 12:20 PM ?
-- McJones September 17, 1980  1:08 PM Changed PDABASE to 400B
```

```
-- DiagnosticPilot.bootmesa  (last edited by McJones on September 18, 1980  11:34 AM)

WART: PilotControl;
RESIDENT: ALL;
NOTRAP: PilotControl, Traps;

GFT: 256;
MDSBASE: 0B;
PDABASE: 400B;
PDAPAGES: 1;
CODEBASE: 400B;

-- Fay July 23, 1980  2:59 PM ?
-- McJones September 18, 1980  11:34 AM Changed PDABASE to 400B
```

```
-- Germ.bootmesa (last edited by: McJones on: July 11, 1980  4:18 PM)

WART: BootSwapGerm;
RESIDENT: ALL;
NOTRAP: BootSwapGerm;

GFT: 128;
MDSBASE: 7000B;
CODEBASE: 7002B;
PDAPAGES: 1;
FRAMEPAGES: 0;

FRAMEWEIGHT: 0,19;
FRAMEWEIGHT: 1,4;
FRAMEWEIGHT: 2,4;
FRAMEWEIGHT: 3,3;
FRAMEWEIGHT: 4,1;
FRAMEWEIGHT: 5,2;
FRAMEWEIGHT: 6,3;
FRAMEWEIGHT: 7,0;
FRAMEWEIGHT: 8,1;
FRAMEWEIGHT: 9,0;
FRAMEWEIGHT: 10,0;
FRAMEWEIGHT: 11,0;
FRAMEWEIGHT: 12,0;
FRAMEWEIGHT: 13,0;
FRAMEWEIGHT: 14,0;
FRAMEWEIGHT: 15,0;
FRAMEWEIGHT: 16,0;
FRAMEWEIGHT: 17,0;
FRAMEWEIGHT: 18,0;
```

-- Communication.pack, *last editted by BLyon, on* October 14, 1980 9:43 AM

```
LowestCommLevels: SEGMENT =
 BEGIN
 QueuesCantSleep: CODE PACK =
  BEGIN
  -- this stuff is always doing sometihing when communications is on
  DispatcherImpl [
   MainDispatcher, PutOnGlobalInputQueue, PutOnGlobalDoneQueue, DummyInputer];
  SystemBufferPoolImpl [ReturnFreeBuffer, GetInputBuffer];
  QueueImpl [Enqueue, Dequeue];
  StatsHot [StatIncr, StatBump];
  END;

 QueuesHot: CODE PACK =
  BEGIN
  -- this stuff is very active when some communications is running
  DispatcherImpl [SendToNextNetwork];
  SystemBufferPoolImpl [
   GetFreeBuffer, MaybeGetFreeBuffer, GetFreeOisBuffer, MaybeGetFreeOisBuffer,
   GetFreePupBuffer, MaybeGetFreePupBuffer];
  BufferPoolImpl [
   GetFreeSendBufferFromPool, GetFreeReceiveBufferFromPool,
   MaybeGetFreeSendBufferFromPool, MaybeGetFreeReceiveBufferFromPool,
   GetFreeSendOisBufferFromPool, GetFreeReceiveOisBufferFromPool,
   GetFreeSendSppBufferFromPool, GetFreeReceiveSppBufferFromPool,
   MaybeGetFreeSendOisBufferFromPool, MaybeGetFreeReceiveOisBufferFromPool,
   MaybeGetFreeSendSppBufferFromPool, MaybeGetFreeReceiveSppBufferFromPool,
   ReturnSendBufferToPool, ReturnReceiveBufferToPool, ReturnBufferToCorrectPool];
  END;

 EthernetCantSleep: CODE PACK =
  BEGIN
  -- this stuff is always active if communications is turned on
  EthernetDriver [Watcher, SmashCSBs, AddEntry];
  END;

 EthernetOneCantSleep: CODE PACK =
  BEGIN
  -- this stuff is always active if communications is turned on
  EthernetOneDriver [Watcher, SmashCSBs, AddEntry];
  END;

 SocketsAndRouterCantSleep: CODE PACK =
  BEGIN
  -- this stuff is always active if communications is turned on
  RoutingTableImpl [RoutingTableUpdater, FindNetworkNumber, RemoveEntry, AddEntry];
  END;

 EthernetOneHot: CODE PACK =
  BEGIN
  EthernetOneDriver [
   DecapsulateBuffer, InInterrupt, AddAddressPair, DeallocateEntry, Demon,
   SendRequest];
  END;

 EthernetHot: CODE PACK =
  BEGIN
  EthernetDriver [
```

```
    DecapsulateBuffer, InInterrupt, AddAddressPair, DeallocateEntry, Demon,
    SendRequest];
  END;

EthernetOneOut: CODE PACK =
  BEGIN
  EthernetOneDriver [
    OutInterrupt, EncapsulatePup, EncapsulateOis, ForwardBuffer,
    SendBuffer, SendBufferInternal, Translate, FindEntry, RemoveEntry];
  END;

EthernetOut: CODE PACK =
  BEGIN
  EthernetDriver [
    OutInterrupt, EncapsulateOis, EncapsulatePup, ForwardBuffer, SendBuffer,
    SendBufferInternal, Translate, FindEntry, RemoveEntry];
  END;

PhonesHot: CODE PACK =
  BEGIN
  PhoneNetworkDriver [
    FrameAny, OISFrameIt, PupFrameIt, TypeOfBuffer, SendFrame,
    SendWait, TranslateChannelStatus, PreemptedSending, ForwardFrame,
    Receiver, AwaitAndDisposeOfFrame, StatsNop, InterruptNop,
    DoGet, StatusChangeWait, ProcessStatusChange, StatIncr, StatBump,
    EnqueueSend, Sender, DequeueSend];
  END;

PhonesHalfDup: CODE PACK =
  BEGIN
  HalfDuplexImpl [
    TimerProcess, WaitTimer, StartTimer, SetTimer, WaitToSend,
    SendCompleted, CheckForTurnAround, NotifyOurTurnToSend,
    AwaitCTS, SendLTA, ClearOurTurn, AwaitNoSending, StatIncr,
    StatBump, MAIN, Initialize, Destroy];
  END;

SocketsAndRouterHot: CODE PACK =
  BEGIN
  SocketImpl [PackageIntoCrate];
  RouterImpl [ReceivePacket, DeliveredToLocalSocket, PacketHit];
  PilotCommUtil [CopyLong];
  ChecksumsImpl [TestChecksum];
  END;

SocketsReceive: CODE PACK =
  BEGIN
  SocketImpl [Get, TransferWait, TransferWaitAny, GetPacket];
  END;

SocketsAndRouterSend: CODE PACK =
  BEGIN
  SocketImpl [Put, PutPacket];
  RouterImpl [SendPacket, SendBroadcastPacket];
  RoutingTableImpl [
    FindNetworkAndTransmit, NotifyTransmitCompleted, ForwardPacket,
    FindDestinationRelativeNetID];
  ChecksumsImpl [SetChecksum];
  END;
```

```
Cool: CODE PACK =
  BEGIN
  QueueImpl [QueueInitialize, QueueCleanup];
  BufferPoolImpl [
    ReturnBufferToPool, ReturnReceiveBufferToCorrectPool, EnumerateBuffersInPool];
  SystemBufferPoolImpl [
    GetSystemBufferPool, GetWordsPerIocb, GetBufferParms, BuffersLeft,
    DataWordsPerPupBuffer];
  RouterImpl [AssignOisAddress];
  SocketImpl [AssignNetworkAddress];
  Boss [GetDeviceChain];
  EthernetOneDriver [GetEthernetHostNumber];
  END;

SocketCreation: CODE PACK =
  BEGIN
  RouterImpl [AddSocket, AssignDestinationRelativeOisAddress];
  SocketImpl [
    Create, CreateInternal, SetWaitTime, MilliSecondsToPulses, GetBufferPool];
  BufferPoolImpl [BufferPoolMake];
  PilotCommUtil [AllocateBuffers, AllocateIocbs, LockBuffers];
  END;

SocketDeletion: CODE PACK =
  BEGIN
  SocketImpl [Delete, Abort, Reset, GetStatus];
  RouterImpl [RemoveSocket];
  BufferPoolImpl [BufferPoolDestroy];
  SystemBufferPoolImpl [FreeQueueDestroy];
  PilotCommUtil [FreeIocbs, FreeBuffers, UnlockBuffers];
  END;

RouterCool: CODE PACK =
  BEGIN
  RouterImpl [
    FindMyHostID, GetOisPacketTextLength, SetOisPacketTextLength,
    SetOisPacketLength, SetOisStormy, SetOisCheckit, SetOisDriverLoopback];
  RoutingTableImpl [
    AddNetwork, AddNetworkLocked, RemoveNetwork, RemoveNetworkLocked,
    StateChanged, ProbeAnInternetRouter, FindNetwork];
  END;

QueuesCool: CODE PACK =
  BEGIN
  DispatcherImpl [
    DummyAddDelete, DummyBroadcaster, DummyStateChanged, GetPupRouter,
    GetOisRouter];
  SystemBufferPoolImpl [
    DataWordsPerRawBuffer, DataWordsPerOisBuffer, DataWordsPerSppBuffer,
    GetClumpOfPupBuffers];
  BufferPoolImpl [
    ReturnSendBufferToCorrectPool, BuffersLeftInPool, SendBuffersLeftInPool,
    ReceiveBuffersLeftInPool];
  QueueImpl [ExtractFromQueue];
  END;

PhonesCool: CODE PACK =
  BEGIN
```

```
  PhoneNetworkDriver [
    MAIN, ActivateTransporter, KillTransporter, CreatePhonePath, PreemptCleanup,
    Initialize, Destroy, CreateSendReceiveProcesses, CheckPath,
    RemoveSendReceiveProcesses, NotifySenderToGoAway, NotifyDialComplete,
    WaitDialComplete];
  PhoneNetworkImpl [
    MAIN, FindPhonePath, KnowAboutPhonePath, ForgetAboutPhonePath,
    AppendString];
  END;

CommonCold: CODE PACK =
  BEGIN
  PilotCommUtil [SecondsToTocks, MsToTocks];
  Boss [GetGiantVector, AddDeviceToChain, RemoveDeviceFromChain,
    SmashDeviceChain, GetNthDevice, GetNetworkID, SetNetworkID];
  DispatcherImpl [SetOisRouter, SetPupRouter];
  SystemBufferPoolImpl [AdjustBufferParms, SetWordsPerIocb];
  EthernetOneDriver [AdjustLengtoOfD0EthernetInputQueue];
  EthernetDriver [AdjustLengtoOfD0EthernetInputQueue];
  RoutingTableImpl [SetRouterFunction, GetRouterFunction];
  END;

CommunicationTurnOn: CODE PACK =
  BEGIN
  -- start-up + finish-up procedures
  CommunicationControl [OiscpPackageMake];
  Boss [OiscpPackageReady, CommPackageReady, CommPackageGo];
  EthernetDriver [
    MAIN, CreateDefaultEthernetDrivers, CreateAnEthernetDriver, SetupEthernetDriver,
    ActivateDriver];
  EthernetOneDriver [
    MAIN, CreateDefaultEthernetOneDrivers, CreateAnEthernetOneDriver,
    SetupEthernetOneDriver, ActivateDriver];
  PilotCommUtil [Zero];
  DispatcherImpl [DispatcherOn];
  SystemBufferPoolImpl [FreeQueueMake];
  RouterImpl [OisRouterOn];
  RoutingTableImpl [RoutingTableOn];
  SocketImpl [SocketOn];
  END;

CommunicationTurnOff: CODE PACK =
  BEGIN
  CommunicationControl [OiscpPackageDestroy];
  Boss [CommPackageOff];
  EthernetDriver [DeactivateDriver, DeleteDriver];
  EthernetOneDriver [DeactivateDriver, DeleteDriver];
  DispatcherImpl [DispatcherOff];
  RouterImpl [OisRouterOff];
  RoutingTableImpl [RoutingTableOff];
  END;

CommunicationDeepFreeze: CODE PACK =
  BEGIN
  -- in order of STARTs
  CommunicationControl [
    MAIN, InitializeCommunication, StartupCommunication];
  StatsHot [MAIN];
  PilotCommUtil [MAIN, CaptureErrors, DefaultErrorHandler, Glitch];
```

```
    Boss [MAIN];
    SystemBufferPoolImpl [MAIN];
    QueueImpl [MAIN];
    BufferPoolImpl [MAIN];
    DispatcherImpl [MAIN];
    ChecksumsImpl [MAIN];
    RouterImpl [MAIN];
    RoutingTableImpl [MAIN];
    SocketImpl [MAIN];
    END;

  SocketsAndRouterINRs: CODE PACK =
    BEGIN
    RoutingTableImpl [
      InternetRouterServer, SendRoutingInfoResponse, RoutingInformationPacket];
    ChecksumsImpl [IncrOisTransportControlAndUpdateChecksum];
    END;

  ShitCan: CODE PACK =
    BEGIN
    Boss [GetDoStats, GetUseCount];
    ChecksumsImpl [SlowlyComputeChecksum];
    SystemBufferPoolImpl [
      GetClumpOfRppBuffers, GetFreeRppBuffer, MaybeGetFreeRppBuffer];
    END;
  END;

HighCommLevels: SEGMENT =
  BEGIN
  Hot: CODE PACK =
    BEGIN
    PacketStreamInstance [
      TakeFromUser, GetForUser, WaitForAttention,
      ReturnGetSppDataBuffer, SendSystemPacket, PutPacketOnSocketChannel,
      Retransmitter, Receiver, GotSequencedPacket, AttentionPacketProcessed,
      UpdateAttnSeqTable, GotErrorPacket];
    NetworkStreamInstance [SendNow, FlushOutputBuffer];
    END;

  Warm: CODE PACK =
    BEGIN
    NetworkStreamInstance [
      GetByte, GetWord, GetBlock, PutBlock, PutByte, PutWord, SetSST,
      SendAttention, WaitAttention];
    END;

  Cool: CODE PACK =
    BEGIN
    PacketStreamInstance [
      MAIN, SuspendStream, SuspendStreamLocked, SetWaitTime,
      FindAddresses, GetSenderSizeLimit, EstablishThisConnection,
      ActivelyEstablish, WaitUntilEstablished, Delete, MilliSecondsToPulses];
    PacketStreamMgr [
      MAIN, Make, Destroy, GetUniqueConnectionID, InsertIntoConnectionTable,
      RemoveFromConnectionTable, ConnectionAlreadyThere];
    NetworkStreamInstance [MAIN, Delete];
    NetworkStreamMgr [
      MAIN, Create, AssignNetworkAddress, FindAddresses, SetWaitTime,
      CreateListener, DeleteListener, CreateTransducer, Close, CloseReply,
```

```
    Listen];
  END;

 TheEchoServer: CODE PACK =
  BEGIN
  EchoServerImpl [MAIN, CreateServer, DeleteServer, EchoServer];
  END;
 END;
```

-- **ControlPack** last edited by Luniewski September 2, 1980 12:11 PM

```
Control: SEGMENT =
BEGIN

ResidentSystem: CODE PACK =
    BEGIN
    SystemImpl [InitializeUIDCleanup]; -- A cleanup procedure
    END;

SwappableSystem: CODE PACK =
    BEGIN
    SystemImpl EXCEPT Initialization, ResidentSystem;
    END;

Initialization: CODE PACK =
    BEGIN
    PilotControl;
    SystemImpl [MAIN];
    END;
END;
```

-- **DiskDriversPack** last edited by Forrest September 20, 1980 8:16 PM

```
DiskDrivers: SEGMENT =
BEGIN

Resident: CODE PACK =
    BEGIN
    DiskChannelImpl EXCEPT Initialization;
    DiskDriverSharedImpl EXCEPT Initialization;
    SA4000Impl EXCEPT Initialization;
    DiskDrivers.StartChainPlug [Start];
    END;

SwappableDiskDrivers: CODE PACK =
    BEGIN
    FloppyChannelImpl EXCEPT Initialization;
    SA800Impl EXCEPT Initialization;
    END;

Initialization: CODE PACK =
    BEGIN
    DiskChannelImpl [MAIN];
    DiskDriverSharedImpl [MAIN];
    FloppyChannelImpl [MAIN, RegisterDrives];
    SA4000Impl [InitGlobals, MAIN, RegisterDrives];
```

```
    SA800Impl [MAIN, RegisterDriverProcs];
    DiskDrivers.StartChainPlug [MAIN];
    END;
END;
```

-- **FileMgrPack** last edited by Luniewski October 14, 1980 8:51 AM.

```
FileMgr: SEGMENT =
BEGIN

-- The hottest pack - in the center of the page fault loop

FileHelperProcess: CODE PACK =
    BEGIN
    FileImpl [FileHelperProcess, GetFileDescriptorSignals, GetFileDescriptor];
    VolAllocMapImpl [Close];
    VolFileMapImpl [Close, ContextSet, Find, Get, GetPageGroup, Lower, ReadNext];
    -- We include GetAttributes, GetLabelString, GetLogicalRootPage and GetStatus here as they fit and are (or are presumed
        tobe) called frequently called in the development environment
    VolumeImpl [Activate, Deactivate, GetAttributes, GetLabelString, GetLogicalRootPage, GetNext,
        GetStatus, LogicalVolumeFind, LVGetEntryPointer, LVGetStatus, VolumeAccess];
    END;

-- Procedures needed by the VMMHelperProcess

VMMHelperProcs: CODE PACK =
    BEGIN
    FileImpl [GetFileAttributes];
    END;

-- Volume File Map growing/shrinking

VFMChangingSize: CODE PACK =
    BEGIN
    VolAllocMapImpl [AccessVAM, AllocPageGroup, FreePageGroup];
    VolFileMapImpl [CreateVPage, Delete, DeletePageGroup, FreeVPage, Insert, InsertPageGroup,
        Merge, PutNext, Split, Xtra];
    END;

-- File Creation

FileCreate: CODE PACK =
    BEGIN
    FileImpl [Create, CreateWithID, CreateWithIDExternal, CreateWithIDInternal, TmpsEnter,
        TmpsGet, TmpsGrow];
    END;

-- File Deletion

FileDelete: CODE PACK =
    BEGIN
    FileImpl [Delete, DeleteCommon, DeleteFileOnVolumeInternal, DeleteImmutable,
        TmpsRemove];
    END;

-- File Attributes

FileAttributes: CODE PACK =
    BEGIN
    FileImpl [ChangeAttributes, ChangeAttributesInternal, GetAttributes, GetSize, IsOnVolume,
        MakeImmutable, MakeMutable, MakePermanent, MakeTemporary, SetSize, SetSizeInternal];
    END;

-- Used by map logging
```

```
MapLogging: CODE PACK =
    BEGIN
    FileImpl [GetFilePoint, GetVIDAndGroup];
    END;

-- Miscellaneous File stuff

OtherFileStuff: CODE PACK =
    BEGIN
    FileImpl [GetBootLocation, MakeBootable, MakeBootableOrUnbootable, MakeUnbootable,
        SetDebuggerFiles];
    END;

-- Transactions related stuff

TransactionRelated: CODE PACK =
    BEGIN
    FileImpl [LogContents, TxCreate, TxMakePerm, TxSetSize];
    END;

-- Scavenging stuff

Scavenger: CODE PACK =
    BEGIN
    ScavengeImpl[Vam, DeleteLog, DeleteLogFile, DeleteOrphanPage, GetLog, ReadBadPage,
        ReadOrphanPage, RewritePage, Scavenge, OpenScavengeContext, CreateLogFile,
        CloseScavengeContext, OpenLogFile, PutWords, EnumerateFiles, PutHeader, CloseLogFile,
        GetBadList, ScavengeVolume, VamFind, Vfm, VamSize, VamInit, VamMarkBadPage,
        VamPieceErase, VamPieceRebuild, VfmPieceRebuild];
    VolFileMapImpl [InitMap];
    VolumeImpl [BeginScavenging, EndScavenging];
    END;

-- Volume stuff - it should seldom be used

Volume: CODE PACK =
    BEGIN
    FileImpl [CloseVolumeAndFlushFiles, DeleteTempsInternal, DeleteTmpsInternalNew,
        DeleteTmpsInternalOld, OpenVolumeAndDeleteTemps];
    VolFileMapImpl [GetNextFile];
    VolumeImpl [Close, CloseLogicalVolume, GetRootFile, IsOnServer, Open, OpenLogicalVolume,
        OpenLogicalVolumeInternal, OpenVolume, PutRootFile, SetRootFile, SignalVolumeAccess];
    END;

-- PhysicalVolume stuff - it should seldom be used

PhysicalVolume: CODE PACK =
    BEGIN
    MarkerPageImpl [Enter, EnterMarkerID, Find, FindInternal, Flush];
    PhysicalVolumeImpl [AssertNotAPilotVolume, AssertPilotVolume, AwaitStateChange,
        CheckPhysicalRootLabel, DriveSize, FinishWithNonPilotVolume, GetAttributes, GetDrive,
        GetHandle, GetHints, GetNext, GetNextDrive, GetNextLogicalVolume, GetPVDrive,
        InterpretHandle, IsReady, Offline, PhysicalRootPageAccess, PhysicalRootPageAccessInternal,
        PhysicalRootPageCheck, PhysicalRootPageMap, PhysicalRootPageUnmap,
        PhysicalVolumeOffLineInternal, PhysicalVolumeOnLineInternal, RegisterPvInfo,
        RegisterSubvolumeMarker, ValidateDrive];
    VolumeImpl [CheckLogicalVolume, EnterLV, FindLogicalVolume, GetLVStatus,
        LogicalVolumeCheck, LogicalVolumeDebuggerCheck, LogicalVolumeLike,
        LogicalVolumeOffLine, LVDecrementPieceCount, PinnedFileEnter, PinnedFileFlush,
        ReadAndCheckLogicalRootLabel, RegisterLogicalSubvolume, RegisterVFiles,
        SubvolumeOffline, SubvolumeOnline, UnregisterVFiles, VFileEnter];
    END;

-- Code used primarily by Othello
```

```
Othello: CODE PACK =
    BEGIN
    MarkerPageImpl [CreateMarkerPage, GetNextPhysicalVolume, MarkerPageMap,
        MarkerPageUnmap, UpdateLogicalMarkerPages, UpdatePhysicalMarkerPages];
    PhysicalVolumeImpl [AccessPhysicalVolumeRootPage, CreateLogicalVolume,
        CreatePhysicalVolume, EraseLogicalVolume, GetContainingPhysicalVolume,
        GetNextBadPage, GetNextSubVolume, GetPhysicalVolumeAttributes,
        GetPhysicalVolumeBootFiles, GetSubVolumeAttributes, MarkPageBad,
        SetPhysicalVolumeBootFiles];
    VolumeImpl [DriveSize, GetContainingPhysicalVolume, GetLogicalVolumeBootFiles, GetType,
        LogicalVolumeCreate, LogicalVolumeErase, SetLogicalVolumeBootFiles];
    END;

-- The coldest pack - initialization code

FileMgrInitialization: CODE PACK = -- initially resident
    BEGIN
    FileImpl [InitializeFileMgr, MAIN, Move, Pin]; -- Pin is only called by PilotControl. Move is here only since it is
        unimplemented. ReplicateImmutable is also implicitly here as it is the same procedure body as Move.
    MarkerPageImpl [MAIN];
    PhysicalVolumeImpl [InitDisks, MAIN];
    ScavengeImpl [Initialize, MAIN];
    VolAllocMapImpl [MAIN];
    VolFileMapImpl [MAIN];
    VolumeImpl [OpenInitialVolumes, MAIN];
    END;
END;
-- FilerPack last edited July 10, 1980 1:20 PM by Luniewski.

Filer: SEGMENT =
BEGIN

-- Resident code - by definition!

Resident: CODE PACK =
    BEGIN
    FileCacheImpl EXCEPT Initialization;
    FilerExceptionImpl EXCEPT Initialization;
    FilerTransferImpl EXCEPT Initialization;
    FileTaskImpl EXCEPT Initialization;
    SubVolumeImpl EXCEPT Initialization;
    END;

-- Initialization code

Initialization: CODE PACK = -- initially resident
    BEGIN
    FileCacheImpl [Initialize, MAIN];
    FilerControl;
    FilerExceptionImpl [MAIN];
    FilerTransferImpl [MAIN];
    FileTaskImpl [MAIN];
    SubVolumeImpl [MAIN];
    END;
END;
-- MesaRuntimePack last edited by Luniewski October 16, 1980 11:33 AM

MesaRuntime: SEGMENT =
BEGIN

Resident: CODE PACK =
```

```
    BEGIN
    BootSwapCross EXCEPT Initialization;
    Instructions EXCEPT Initialization;
    PilotNub EXCEPT Initialization;
    Processes [Yield];
    Signals EXCEPT Initialization;
    Traps EXCEPT Initialization;
    -- make these resident so BootButton will always work
    -- and on the Dandelion the funny memory crockery will work
    SnapshotImpl [BootButton, InLoadFromBootLocation];
    END;

Frames: CODE PACK =
    BEGIN
    FrameImpl EXCEPT Initialization;
    END;

ProcessesHot: CODE PACK =
    BEGIN
    Processes [CheckProcess, Detach, End, Fork, GetPriority, MsecToTicks, Pause, SecondsToTicks,
        SetPriority, SetTimeout, TicksToMsec];
    END;

ProcessesCold: CODE PACK =
    BEGIN
    Processes EXCEPT Initialization, ProcessesHot, Resident;
    END;

InLoadOutLoad: CODE PACK =
    BEGIN
    SnapshotImpl [BootFromFile, BootFromVolume, BootFromPhysicalVolume, GetBootFileSize,
        GetDevice, InLoad, OutLoad];
    END;

SnapshotMisc: CODE PACK =
    BEGIN
    SnapshotImpl [InstallPhysicalVolumeBootFile, InstallVolumeBootFile, MakeBootable,
        MakeUnbootable];
    END;

Initialization: CODE PACK =
    BEGIN
    BootSwapCross [MAIN];
    FrameImpl [InitializeFrames, MAIN];
    Instructions [Initialize, MAIN];
    PilotNub [InitializeInterrupt, InitializePilotNub, Install, MAIN];
    Processes [Initialize, MAIN];
    Signals [Initialize, MAIN];
    SnapshotImpl [InitializeSnapshot, MAIN];
    Traps [Initialize, MAIN];
    END;
END;

-- MiscPack last edited by Luniewski October 16, 1980 2:08 PM

Misc: SEGMENT =
BEGIN
```

```
Resident: CODE PACK =
    BEGIN
    ResidentHeapImpl [FreeNode, LargeNode, MakeNode, NodeSize, RelativePointerToPointer,
        SplitNode, WordsToPages];
    StreamImpl [ByteBlt];
    UtilitiesImpl EXCEPT Initialization;
    ZoneImpl [AddSegment, FreeNode, MakeNode, Split, SplitNode, ValidateZone];
    END;

Streams: CODE PACK =
    BEGIN
    StreamImpl EXCEPT Initialization, Resident;
    END;

-- The following code packs are non-resident and depend upon the resident heap only creating, deleting, splitting nodes and adding
    segments.

Nodes: CODE PACK =
    BEGIN
    HeapImpl [Expand, ExpandHeap, ExpandMDS, FreeLargeOrRegularNode, FreeMDSNode,
        FreeNode, MakeLargeNode, MakeMDSNode, MakeNode, MakeSpace, Narrow,
        PagesForNewSegment];
    END;

HeapPruning: CODE PACK =
    BEGIN
    HeapImpl [EnumerateSpaces, Prune, PruneHeap, PruneMDS, SpaceFromLongPointer];
    ZoneImpl [GetAttributes, GetSegmentAttributes, RemoveSegment];
    END;

Attributes: CODE PACK =
    BEGIN
    HeapImpl [GetAttributes, GetAttributesHeap, GetAttributesMDS];
    ZoneImpl [NodeSize];
    END;

CreateDelete: CODE PACK =
    BEGIN
    HeapImpl [Create, CreateHeap, CreateMDS, Delete, DeleteHeap, DeleteMDS, FreeSpace,
        MakeResidentHeap, MakeResidentMDS, MakeSwappableHeap, MakeSwappableMDS];
    ZoneImpl [Create];
    END;

Miscellaneous: CODE PACK =
    BEGIN
    HeapImpl [CheckOwner, MakeResident, MakeSwappable];
    END;

Checking: CODE PACK =
    BEGIN
    HeapImpl [SetChecking, SetCheckingHeap, SetCheckingMDS];
    ZoneImpl [CheckNode, CheckZone, SetChecking];
    END;

Initialization: CODE PACK =
    BEGIN
    HeapImpl [InitializeHeap, MAIN];
    ResidentHeapImpl [InitializeResidentHeap, MAIN];
```

```
    StreamImpl [InitializeStream, MAIN];
    UtilitiesImpl [InitializeUtilities, MAIN];
    ZoneImpl [InitializeZone, MAIN];
    END;
END;
```

-- **RuntimeLoaderPack** last edited by Sandman October 7, 1980 10:47 AM

```
RuntimeLoader: SEGMENT =
BEGIN

LoaderUnloaderCommon: CODE PACK =
    BEGIN
    BcdOperations [ProcessModules];
    PilotLoaderSupport [CloseLinkSpace, FreeSpace, GetSpace, OpenLinkSpace, ReadLink,
        WriteLink];
    PilotLoadState [AcquireBcd, EnterModule, GetMap, GetModule, InputLoadState, ReleaseBcd,
        ReleaseLoadState, ReleaseMap];
    END;

Loader: CODE PACK =
    BEGIN
    BcdOperations [ProcessConfigs, ProcessExports, ProcessImports, ProcessSegs];
    PilotLoaderCore [AssignControlModules, BadVersion, BindImports, CheckTypes, ConvertLink,
        EqualNames, EqualVersions, LoadModules, New, NextMultipleOfFour, ProcessFramePacks,
        ProcessLinks, ProcessTypeMap, GetModule, SetLink];
    PilotLoadState [MapConfigToReal];
    PilotLoaderSupport [AllocateCodeSpaces, AllocateFrames, AppendName, BadFile, DestroyMap,
        FindCode, FindMappedSpace, FindSegments, FindSPHandle, InitBinding, InitializeMap,
        InvalidModule, Load, LoadBcd, LoadConfig, NewConfig, ReleaseBinding, ReleaseCode,
        ReleaseFrames, Run, RunConfig, SgiToLP];
    PilotLoadState [BcdExports, BcdExportsTypes, BcdUnresolved, UpdateLoadState];
    END;

Unloader: CODE PACK =
    BEGIN
    PilotLoadState [EnumerateBcds, MapRealToConfig, RemoveConfig];
    PilotUnLoader EXCEPT RuntimeLoaderInitialization;
    END;

Misc: CODE PACK =
    BEGIN
    BcdOperations [FindName, ModuleVersion, ProcessExternals, ProcessFiles, ProcessNames,
        ProcessSpaces];
    PilotLoadState [EnumerateModules, ForceDirty];
    END;

RuntimeLoaderInitialization: CODE PACK =
    BEGIN
    BcdOperations [MAIN];
    PilotLoaderCore [MAIN];
    PilotLoaderSupport [MAIN];
    PilotLoadState [InitializePilotLoadState, MAIN];
    PilotUnLoader [MAIN];
    END;
END;
```

-- **RuntimePerfPack** last edited by Forrest September 20, 1980 7:30 PM

```
RuntimePerf: SEGMENT =
BEGIN

Resident: CODE PACK =
    BEGIN
    PilotCounter EXCEPT Initialization;
    PilotPerfMonitor EXCEPT Initialization;
    END;

Initialization: CODE PACK = -- Initially swapped in
    BEGIN
    PilotCounter [InitializePilotCounter, MAIN];
    PilotPerfMonitor [InitializePilotPerfMonitor, MAIN];
    END;
END;
```

-- **SwapperPack** last edited August 27, 1980 4:49 PM by Luniewski.

```
Swapper: SEGMENT =
BEGIN

Resident: CODE PACK =
    BEGIN
    CachedRegionImplA EXCEPT Initialization;
    CachedRegionImplB EXCEPT Initialization;
    CachedSpaceImpl EXCEPT Initialization;
    MStoreImpl EXCEPT Initialization;
    PageFaultImpl EXCEPT Initialization;
    ResidentMemoryImpl EXCEPT Initialization;
    SwapBufferImpl EXCEPT Initialization;
    SwapperControl EXCEPT Initialization;
    SwapperExceptionImpl EXCEPT Initialization;
    SwapTaskImpl EXCEPT Initialization;
    END;

SwappableSwapper: CODE PACK =
    BEGIN
    SimpleSpaceImpl EXCEPT Initialization;
    END;

Initialization: CODE PACK =
    BEGIN
    CachedRegionImplA [InitializeInternal, InitializeRegionCacheA, InitializeRegionCacheB, MAIN];
    CachedRegionImplB [Initialize, MAIN];
    CachedSpaceImpl [MAIN];
    MStoreImpl [Initialize, InitializeMStore, MAIN];
    PageFaultImpl [MAIN];
    ResidentMemoryImpl [InitializeResidentMemoryA, InitializeResidentMemoryB, MAIN];
    SimpleSpaceImpl [AllocateVM, Create, DescribeSpace, DescribeSpaceInternal,
        DisableInitialization, HandleFromPage, InitializeSimpleSpace, MAIN, SuperFromPage];
    SwapBufferImpl [InitializeSwapBuffer, MAIN];
    SwapperControl [InitializeSwapper, MAIN];
    SwapperExceptionImpl [MAIN];
    SwapTaskImpl [MAIN];
    END;
END;
```

-- **TransactionsPack** *(Last edited by Gobbel October 2, 1980 11:13 AM)*

```
Transactions: SEGMENT =
BEGIN
```

```
RunningTransaction: CODE PACK =
    BEGIN
    TransactionLogImpl [AssureLogRoom, LogInternal];
    TransactionStateImpl [AddToTransaction, Log, MakeNode, MaybeCrash, NullProc,
        UpdateStateFile, WithdrawFromTransaction];
    END;

AbortBeginCommitCommon: CODE PACK =
    BEGIN
    TransactionImpl [ReleaseLog, ReleaseLogInternal];
    TransactionStateImpl [FreeNode, NoMoreOperations, ReleaseTransaction];
    END;

BeginTransaction: CODE PACK =
    BEGIN
    TransactionStateImpl [Begin];
    END;

Abort: CODE PACK =
    BEGIN
    TransactionImpl [Abort, RestoreFilesInTransaction];
    END;

Commit: CODE PACK =
    BEGIN
    TransactionImpl [Commit];
    TransactionStateImpl [RecordCommit];
    END;

TransactionsInitialization: CODE PACK =
    BEGIN
    TransactionImpl [CheckState, CompareStateEdition, DisableTransactions,
        InitializeTransactionData, MAIN, RecoverState, RecoverTransactions, ValidateLogFile];
    TransactionLogImpl [InitializeLogsA, InitializeLogsB, MAIN];
    TransactionStateImpl [InitializeStateA, InitializeStateB, MAIN, SetCrashProcedure];
    END;
END;
```

-- **UserTerminalDriverPack** last edited by Luniewski July 29, 1980 4:41 PM

```
UserTerminalDriver: SEGMENT =
BEGIN

Resident: CODE PACK =
    BEGIN
    UserTerminalImpl EXCEPT Initialization;
    END;

Initialization: CODE PACK =
    BEGIN
    UserTerminalDriver.StartChainPlug [MAIN, Start]; -- Explicit qualification is needed to avoid ambiguous
        references when all of Pilot is packaged via a merged segment.
    UserTerminalImpl [MAIN, Start];
    END;
END;
```

-- **VMMgrPack** last edited by Luniewski October 16, 1980 2:59 PM

```
VMMgr: SEGMENT =
BEGIN

VMMHelperProcess: CODE PACK =
```

```
    BEGIN
    HierarchyImpl [Touch, LoadDesc];
    ProjectionImpl [Touch, Get];
    SpaceImplA [VMMHelperProcess];
    STLeafImpl [Close, Open];
    STreeImpl [Get, KeyFromPDesc, SearchLeaf, SearchRoot, Update, UpdateRoot];
    END;

SpaceOpsCommon: CODE PACK =
    BEGIN
    -- we include the operations needed to get space handles from long pointers here as most clients will keep pointers to spaces
        instead of space handles, thus necessitating a translation before calling into the Space operations. GetAttributes is
        included here: 1) it fits, and 2) in the development environment, it is frequently called from
        HeapImpl.SpaceFromLongPointer which is called from HeapImpl.PruneMDS which is frequently called. The presence of
        GetAttributes here => FindFirstWIthin must also be here.
    HierarchyImpl [GetDescriptor, FindFirstWithin, GetInterval, Update];
    ProjectionImpl [TranslateLevel];
    MapLogImpl [WriteLog];
    ProjectionImpl [ForceOut];
    SpaceImplA [ApplyToInterval, ForAllRegions, GetAttributes, GetHandle,
        PageFromLongPointer];
    SpaceImplB [EnterSpace, GetSpaceDesc, JoinTransaction, ProcessWindow,
        ReleaseDefaultWindow];
    END;

SpaceStatusOps: CODE PACK =
    BEGIN
    SpaceImplA [Activate, ApplyToSpace, Deactivate, ForceOut, Kill];
    SpaceImplB [MakeReadOnly, MakeWritable];
    END;

SpaceMap: CODE PACK =
    BEGIN
    SpaceImplB [Map];
    STLeafImpl [AllocateWindow];
    END;

SpaceUnmap: CODE PACK =
    BEGIN
    SpaceImplB [Unmap, UnmapInternal];
    STLeafImpl [DeallocateWindow]; -- Also used by SpaceImpl.Remap
    END;

SpaceCreate: CODE PACK =
    BEGIN
    HierarchyImpl [Insert];
    ProjectionImpl [Split];
    SpaceImplA [Create, CreateAligned, CreateUniformSwapUnits, CreateInternal];
    STLeafImpl [Create];
    STreeImpl [AllocateRoot, ExpandRoot, Insert];
    END;

SpaceDelete: CODE PACK =
    BEGIN
    HierarchyImpl [Delete];
    ProjectionImpl [DeleteSwapUnits, Merge];
    SpaceImplA [Delete, DeleteSwapUnits, DeleteInternal];
    STLeafImpl [Delete];
    STreeImpl [Delete];
```

```
    END;

SpaceCopying: CODE PACK =
    BEGIN
    SpaceImplB [CopyIn, CopyOut, GetCopyInfo];
    END;

SpaceTransactions: CODE PACK =
    BEGIN
    SpaceImplB [ReleaseFromTransaction];
    END;

SpaceRemap: CODE PACK =
    BEGIN
    SpaceImplB [Remap];
    END;

SpaceCodeHacking: CODE PACK =
    BEGIN
    SpaceImplA [CreateForCode, MakeGlobalFrameResident, MakeGlobalFrameSwappable,
        MakeProcedureResident, MakeProcedureSwappable, MakeResident, MakeSwappable,
        MaxIntervalForCode, SpaceForCode];
    END;

SpaceMisc: CODE PACK =
    BEGIN
    HierarchyImpl [ValidSpaceOrSwapUnit];
    SpaceImplA [LongPointerFromPage, GetWindow, LongPointer, Pointer, VMPageNumber];
    END;

VMMgrInitialization: CODE PACK =
    BEGIN
    HierarchyImpl [MAIN];
    MapLogImpl [Initialize, MAIN];
    ProjectionImpl [MAIN];
    SpaceImplA [InitializeInternal, InitializeSpace, MAIN];
    SpaceImplB [HandleWriteProtectTrap, InitializeSpaceImplB, MAIN, WriteProtectTrap];
    STLeafImpl [AllocateMapLogFile, InitVMMgrStore, InitSTLeaf, MAIN];
    STreeImpl [Initialize, MAIN];
    VMMControl [CreateSpace, FillHierarchyAndProjection, InitializeVMMgr, IsDiagnosticPilot,
        MAIN];
    END;
END;
```

-- **TestPilotPack** last edited by Luniewski October 16, 1980 2:11 PM

```
TestPilotResident: SEGMENT MERGES Control, DiskDrivers, Filer, MesaRuntime, Misc, RuntimePerf,
    Swapper, UserTerminalDriver =
BEGIN
-- This segment merges all of the resident code so as to minimize resident breakage

Resident: CODE PACK =
    BEGIN
    Control.ResidentSystem;
    DiskDrivers.Resident;
    Filer.Resident;
    MesaRuntime.Resident;
    Misc.Resident;
    RuntimePerf.Resident;
```

```
    Swapper.Resident;
    UserTerminalDriver.Resident
    END;
Control.SwappableSystem;
DiskDrivers.SwappableDiskDrivers;
MesaRuntime.Frames;
MesaRuntime.ProcessesHot;
MesaRuntime.ProcessesCold;
MesaRuntime.InLoadOutLoad;
MesaRuntime.SnapshotMisc;
Misc.Nodes;
Misc.HeapPruning;
Misc.Attributes;
Misc.Checking;
Misc.CreateDelete;
Misc.Miscellaneous;
Misc.Streams;
Swapper.SwappableSwapper;

ResidentInitialization: CODE PACK =
    BEGIN
    Control.Initialization;
    DiskDrivers.Initialization;
    Filer.Initialization;
    MesaRuntime.Initialization;
    Misc.Initialization;
    RuntimePerf.Initialization;
    Swapper.Initialization;
    UserTerminalDriver.Initialization
    END;
END;

SwappableStore: SEGMENT MERGES FileMgr, Transactions, VMMgr =
BEGIN
-- This segment merges all of the swappable code in the Store config
FileMgr.FileHelperProcess;

VMMHelperProcess: CODE PACK =
    BEGIN
    VMMgr.VMMHelperProcess;
    FileMgr.VMMHelperProcs;
    END;

FileMgr.VFMChangingSize;
FileMgr.FileCreate;
FileMgr.FileDelete;
FileMgr.FileAttributes;
FileMgr.OtherFileStuff;

FileMgr.Scavenger;
FileMgr.Volume;
FileMgr.PhysicalVolume;
FileMgr.Othello;
Transactions.Abort;

VMMgrCommon: CODE PACK =
    BEGIN
    FileMgr.MapLogging;
    VMMgr.SpaceOpsCommon;
    END;
VMMgr.SpaceStatusOps;
```

```
VMMgr.SpaceMap;
VMMgr.SpaceUnmap;
VMMgr.SpaceCreate;
VMMgr.SpaceDelete;
VMMgr.SpaceCopying;
VMMgr.SpaceRemap;
VMMgr.SpaceCodeHacking;
VMMgr.SpaceMisc;

TransactionsRunning: CODE PACK =
    BEGIN
    FileMgr.TransactionRelated;
    Transactions.RunningTransaction;
    VMMgr.SpaceTransactions;
    END;

TransactionsCommon: CODE PACK =
-- Merged from AbortBeginCommitCommon, BegTransaction, Commit, VMMgr.SpaceTransactions. This assumes that commiting
    a transacion is the normal mode of operation and that aborting is "rare".
    BEGIN
    Transactions.AbortBeginCommitCommon;
    Transactions.BeginTransaction;
    Transactions.Commit;
    END;
FileMgr.FileMgrInitialization;
Transactions.TransactionsInitialization;
VMMgr.VMMgrInitialization;
END;

ResidentFrames: FRAME PACK =
    BEGIN
    -- Control
    SystemImpl;
    PilotControl; -- Must be resident due to its exported variables
    -- DiskDrivers
    DiskChannelImpl;
    DiskDriverSharedImpl;
    FloppyChannelImpl;
    SA4000Impl; -- Must be resident until Pilot knows how to dynamically make it resident
    SA800Impl; -- Must be resident until Pilot knows how to dynamically make it resident
    -- Filer
    FileCacheImpl;
    FilerExceptionImpl;
    FilerTransferImpl;
    FileTaskImpl;
    SubVolumeImpl;
    -- MesaRuntime
    BootSwapCross;
    Instructions;
    PilotNub;
    Processes;
    Signals;
    SnapshotImpl;
    Traps;
    -- Misc
    ResidentHeapImpl;
    UtilitiesImpl;
    ZoneImpl;
    -- RuntimePerf
    PilotCounter;
```

```
    PilotPerfMonitor;
    -- Swapper
    CachedRegionImplA;
    CachedRegionImplB;
    CachedSpaceImpl;
    MStoreImpl;
    PageFaultImpl;
    ResidentMemoryImpl;
    SwapBufferImpl;
    SwapperControl;
    SwapperExceptionImpl;
    SwapTaskImpl;
    -- UserTerminalDriver
    UserTerminalImpl;
    -- In addition, due to the need to swap writable frame packs to a temporary file and not to the boot file, it is necessary that
        anything that PilotControl calls before mapping initially out spaces have their frames be resident (even though, in a strictly
        logical sense, they need not be resident). At the moment, this means everything in the Store config.
    SimpleSpaceImpl;
    FileImpl;
    VolAllocMapImpl;
    VolFileMapImpl;
    VolumeImpl;
    HierarchyImpl;
    MapLogImpl;
    ProjectionImpl;
    SpaceImplA;
    SpaceImplB;
    STLeafImpl;
    HConfig.STreeImpl;
    PConfig.STreeImpl;
    MarkerPageImpl;
    PhysicalVolumeImpl;
    ScavengeImpl;
    Transactions;
    FrameImpl;
    StreamImpl;
    -- The following are included here as they fit are hot in the development environment
    HeapImpl;
    END;


CommunicationLowestLevelFrames: FRAME PACK =
  BEGIN
  Communication.PilotCommUtil;
  Communication.Boss;
  Communication.CommunicationControl;
  Communication.QueuePackage;
  Communication.Level0;
  Communication.Level1;
  Communication.Level2 [PacketStreamMgr, NetworkStreamMgr, EchoServerImpl];
  END;

CommunicationStatsFrame: FRAME PACK =
  BEGIN
  Communication.StatsHot;
  END;

CommunicationInstanceFrames: FRAME PACK =
  BEGIN
```

```
  Communication.Level2 [PacketStreamInstance, NetworkStreamInstance];
  END;

RandomCoolFrames: FRAME PACK =
  BEGIN
    BcdOperations;
    PilotLoaderSupport;
    PilotLoadState;
    PilotLoaderCore;
    PilotUnLoader;
    -- The following are initialization only frames. They are included here since they do not increase the size of this swap unit, thus
        saving one page of MDS
    DiskDrivers.StartChainPlug;
    FilerControl;
    UserTerminalDriver.StartChainPlug;
    VMMControl;
    END;
```

```
Pilot: CONFIGURATION
    IMPORTS
        -- device faces -- DisplayFace, EthernetFace, EthernetOneFace, HeadStartChain, KeyboardFace, MouseFace,
        ProcessorFace, RS232CFace, RS366Face, SA4000Face, SA800Face, TemporarySetGMT,
        -- client -- PilotClient
    EXPORTS
        -- public -- ByteBlt, Dialup, File, FloppyChannel, Heap, NetworkStream, PhoneNetwork, PhysicalVolume, Process,
        RS232C, RS232CManager, Runtime, Scavenger, Socket, Space, SpecialCommunication, Stream, System,
        TemporaryBooting, TemporaryTransaction, Transaction, UserTerminal, Volume, Zone,
        -- communication private -- OISCPDefs, StatsDefs, StatsOps, SpecialSystem,
        -- PUP package only -- BufferDefs, CommUtilDefs, DriverDefs, PupDefs,
        -- insiders only -- DiskChannel, FloppyChannelInternal, PilotSwitches, SpecialVolume, SubSys, VolumeExtras,
        -- heads only -- DeviceCleanup, ProcessInternal, ResidentHeap, RuntimeInternal, SpecialHeap, SpecialSpace

= { TestPilot; RS232CIO; SpecialTransactionStub LINKS: CODE; }.
```

LOG

Time: February 5, 1980 9:54 AM *By: Garlick* *Action: Export PhoneNetwork*

Time: April 28, 1980 1:54 PM *By: Danielson* *Action: Move RS232C driver and head up from TestPilot*

Time: July 19, 1980 3:20 PM *By: McJones* *Action: Export Heap, Transaction, SpecialTransaction; StatsPrivateDefs = >StatsOps*

Time: August 1, 1980 4:01 PM *By: Gobbel* *Action: SpecialTransaction = >TemporaryTransaction*

Time: August 6, 1980 2:46 PM *By: Gobbel* *Action: Move HeadsD0 out*

Time: August 6, 1980 2:46 PM *By: McJones* *Action: Don't export ByteBltDefs; import HeadStartChain, SA800Face; move RS232CIOHeadsD0 out*

Time: August 27, 1980 4:58 PM *By: Forrest* *Action: Export stuff for heads*

Time: August 29, 1980 1:20 PM *By: BLyon* *Action: renamed EthernetFace to EthernetOneFace.*

Time: September 3, 1980 11:27 AM *By: Forrest* *Action: Delete PPData, Add special communication.*

Time: September 9, 1980 5:36 PM *By: Forrest* *Action: Add EthernetFace.*

Time: October 2, 1980 1:46 AM *By: Gobbel* *Action: Add stub for SpecialTransaction.*

Time: October 2, 1980 5:51 PM *By: Jose* *Action: Export FloppyChannelInternal, DiskChannel, SpecialVolume.*

Time: October 3, 1980 5:52 PM *By: Forrest* *Action: Import TemporarySetGMT; export PilotSwitches.*

Time: October 9, 1980 1:58 PM *By: Luniewski* *Action: Export VolumeExtras. Move SubSys to "For insiders" section.*

```
-- Note that the configuration name is TestPilot; this serves as input to the Packager to produce TestPilot.bcd
-- It may also be used by using binder command line options/Directory Statements

PACK -- resident code; some modules correspond to PACK within a smaller configuration
    BootSwapCross, -- TestPilotKernel
    UserTerminalImpl, -- UserTerminalDriver
    DiskDriverSharedImpl; -- DiskDrivers

TestPilot: CONFIGURATION
    IMPORTS Dialup, DisplayFace, EthernetFace, EthernetOneFace, HeadStartChain, Inline, KeyboardFace, MouseFace, PilotClient,
        PilotDisk, ProcessorFace, RS232C, SA4000Face, SA800Face, SpecialTransaction, TemporarySetGMT
    EXPORTS
        -- public -- ByteBlt, File, Heap, NetworkStream, OISTransporter, PhoneNetwork, PhysicalVolume, Process, Runtime,
        Scavenger, Socket, Space, SpecialCommunication, Stream, System, TemporaryBooting, TemporaryTransaction, Transaction,
        UserTerminal, Volume, Zone,
        -- head support -- DeviceCleanup,
        -- Communication private -- Checksums, OISCPDefs, SocketInternal, StatsDefs, StatsOps,
        -- for PUP package only -- BufferDefs, CommUtilDefs, DriverDefs,
        PupDefs,
        -- insiders -- BcdOps, DebuggerSwap, DiskChannel, DiskChannelBackend, DiskDriverShared, FloppyChannel,
        FloppyChannelInternal, KernelFile, PacketStream, PilotLoadStateOps, PilotSwitches, ProcessInternal, ResidentHeap,
        ResidentMemory, Router, RuntimeInternal, Snapshot, SpecialFile, SpecialHeap, SpecialSpace, SpecialSystem,
        SpecialTransaction, SpecialVolume, SubSys, SystemInternal, Utilities, VolumeExtras =

BEGIN

TestPilotKernel[Dialup, DiskChannel, DriverStartChain1, EthernetFace, EthernetOneFace, HeadStartChain, KeyboardFace, PilotClient,
    ProcessInternal, ProcessorFace, RS232C, RuntimeInternal, SpecialTransaction, StoreDriverStartChain, TemporarySetGMT];

[DiskChannel, DiskChannelBackend, DiskDriverShared, DriverStartChain1, FloppyChannel, FloppyChannelInternal, PhysicalVolume,
    StoreDriverStartChain] ← DiskDrivers[DriverStartChain, Inline, PhysicalVolume, PilotDisk, Process, ProcessInternal, ResidentHeap,
    RuntimeInternal, SA4000Face, SA800Face, Space, SpecialSpace, Transaction, Utilities];

[DriverStartChain, UserTerminal] ← UserTerminalDriver[DisplayFace, KeyboardFace, MouseFace, PilotSwitches, Process,
    ProcessInternal, ResidentHeap, Runtime, Space, SpecialSpace, System, Transaction];

END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

*Time: April 16, 1980 5:33 PM By: McJones Action: Add SetGMTUsing*Ethernet

Time: April 13, 1980 11:00 PM By: Forrest Action: Don't export IOCS

Time: April 28, 1980 1:52 PM By: Danielson Action: Remove RS232 head and channel

Time: April 29, 1980 8:57 AM By: Schwartz Action: Don't export Dialup and PhoneNetwork; export DOInputOutput

Time: May 2, 1980 12:51 PM By: Forrest Action: Change packs

Time: June 11, 1980 11:47 AM By: Luniewski Action: Export PhysicalVolume as public

Time: June 19, 1980 2:03 PM By: Luniewski Action: Export Scavenger as public

Time: June 26, 1980 2:16 PM By: Luniewski Action: Export BcdOps and PilotLoadStateOps to Pilot insiders

Time: July 16, 1980 11:25 PM By: Gobbel Action: Export Transaction, SpecialTransaction

Time: July 19, 1980 3:12 PM By: McJones Action: Export Heap, SpecialHeap, DiskChannelBackend, DiskDriverShared, OISTransporter, PhoneNetwork, SocketInternal, Transaction; import Dialup, RS232C; StatsPrivateDefs = >StatsOps, OISProcessorFace = >ProcessorFace

Time: July 29, 1980 11:29 AM By: McJones Action: Remove HeadsDO

Time: August 12, 1980 10:11 AM By: BLyon Action: Removed ByteBltDefs.

Time: August 13, 1980 5:48 PM By: Jose Action: Import Inline, PilotDisk, and SA800Face, export FloppyChannel, link Floppy into DriverStartChain

Time: August 26, 1980 10:53 AM By: BLyon Action: renamed EthernetFace to EthernetOneFace.

Time: August 29, 1980 7:23 PM By: Forrest Action: Killed PPData Export.

Time: September 6, 1980 3:00 PM By: HGM Action: IMPORT EthernetFace.

Time: September 15, 1980 11:11 AM By: Forrest Action: Links: CODE.

Time: September 20, 1980 3:05 PM By: Forrest Action: Change outer name to TestPilotUnpacked.

Time: September 25, 1980 4:48 PM By: Gobbel Action: Added import and export of SpecialTransaction.

Time: October 2, 1980 5:47 PM By: Jose Action: Export FloppyChannelInternal.

Time: October 3, 1980 5:45 PM By: Forrest Action: Import TemporarySetGMT.

Time: October 9, 1980 1:55 PM By: Luniewski Action: Add Export of VolumeExtras. Move SubSys to "For insiders" section.

```
PACK -- resident code; each module corresponds to a PACK in a smaller configuration
    BootSwapCross, -- MesaRuntime
    PilotCounter, -- PilotPerf
    ResidentHeapImpl, -- Misc
    FilerControl; -- Store

TestPilotKernel: CONFIGURATION
    IMPORTS
        Dialup, DiskChannel, DriverStartChain, EthernetFace, EthernetOneFace, HeadStartChain, KeyboardFace, PilotClient,
        ProcessInternal, ProcessorFace, RS232C, RuntimeInternal, SpecialTransaction, StoreDriverStartChain, TemporarySetGMT
    EXPORTS
        -- public -- ByteBlt, File, Heap, NetworkStream, OISTransporter, PhoneNetwork, PhysicalVolume, Process, Runtime, Scavenger,
        Socket, Space, SpecialCommunication, Stream, System, TemporaryBooting, TemporaryTransaction, Transaction, Volume,
        Zone,
        -- for heads and drivers Only -- DeviceCleanup, PilotSwitches, ProcessInternal, ResidentHeap, RuntimeInternal, SpecialSpace,
        Utilities,
        -- Communication private -- Checksums, OISCPDefs, SocketInternal, StatsDefs, StatsOps,
        -- for PUP package only -- BufferDefs, CommUtilDefs, DriverDefs,
        PupDefs,
        -- for insiders only -- BcdOps, DebuggerSwap, KernelFile, PacketStream, PilotLoadStateOps, ResidentMemory, Router,
        Snapshot, SpecialFile, SpecialHeap, SpecialSystem, SpecialTransaction, SpecialVolume, SubSys, SystemInternal,
        VolumeExtras =

BEGIN

Control;

MesaRuntime;

RuntimePerf;

Misc;

Store;

RuntimeLoader;

Communication;

END.
```

LOG

(For earlier log entries see Pilot 3.0 archive version.)
Time: April 16, 1980 5:35 PM By: McJones Action: Add import of TemporarySetGMT
Time: February 25, 1980 1:19 PM By: Forrest Action: Remove IO
Time: April 30, 1980 4:55 PM By: Forrest Action: Add RuntimePerf; change packs
*Time: May 2, 1980 5:06 PM By: HGM/Blyon Action: Don't import DOInputOutput (Ethernet*Driver *now uses* Ethernet*Face)*
Time: June 11, 1980 1:20 PM By: Luniewski Action: Export of PhysicalVolume as a public interface
Time: June 19, 1980 2:02 PM By: Luniewski Action: Export Scavenger as a public interface
Time: June 26, 1980 2:16 PM By: Luniewski Action: Export BcdOps and PilotLoadStateOps to Pilot insiders
Time: July 16, 1980 11:23 PM By: Gobbel Action: Export Transaction, SpecialTransaction
Time: July 19, 1980 3:11 PM By: McJones Action: Export Heap, SpecialHeap, OISTransporter, PhoneNetwork, SocketInternal; import Dialup, RS232C; StatsPrivateDefs = > StatsOps, OISProcessorFace = > ProcessorFace
Time: August 1, 1980 3:58 PM By: Gobbel Action: Change SpecialTransaction to TemporaryTransaction.
Time: August 12, 1980 10:08 AM By: BLyon Action: Removed ByteBltDefs.
Time: August 26, 1980 10:50 AM By: BLyon Action: renamed EthernetFace to EthernetOneFace.
Time: September 6, 1980 3:00 PM By: HGM Action: IMPORT EthernetFace.
Time: September 25, 1980 4:46 PM By: Gobbel Action: Added import and export of SpecialTransaction.
Time: October 9, 1980 1:53 PM By: Gobbel Action: Added Export of VolumeExtras. Moved SubSys to "For insiders" section.

```
Control: CONFIGURATION LINKS: CODE
    IMPORTS BootSwap, CommunicationPrograms, DebuggerSwap, DeviceCleanup, DriverStartChain, HeadStartChain, Inline,
            KernelFile, MiscPrograms, PerformancePrograms, PilotClient, Process, ProcessorFace, Runtime, RuntimeInternal,
            RuntimePrograms, Space, StoragePrograms, StoreDriverStartChain, TemporarySetGMT, Transaction, Volume
    EXPORTS DebuggerSwap, PilotSwitches, SpecialSystem, StoragePrograms, System, SystemInternal =

BEGIN

PilotControl;

SystemImpl;

END.
```

LOG

(For earlier log entries, see Pilot 4.0 archive version.)

Time: April 8, 1980 8:38 AM	*By: Knutsen*	*Action: Export StoragePrograms; don't import MStore, ProcessInternal; don't export ResidentMemory*
Time: April 16, 1980 5:44 PM	*By: McJones*	*Action: Import TemporarySetGMT instead of exporting ControlPrograms and importing DOInputOutput*
Time: May 1, 1980 4:55 PM	*By: Forrest*	*Action: Delete pack statements*
Time: June 27, 1980 3:03 PM	*By: Knutsen*	*Action: Don't import SpecialSpace*
Time: July 11, 1980 12:06 PM	*By: McJones*	*Action: Import Transaction; don't import EthernetFace; OISProcessorFace = > ProcessorFace*
Time: August 29, 1980 5:25 PM	*By: Forrest*	*Action: Don't use PPData*
Time: September 15, 1980 10:40 AM	*By: Forrest*	*Action: change to use code links*

```
PACK -- resident code; each component corresponds to a PACK in a smaller configuration
    FilerControl, -- Filer
    SwapperControl; -- Swapper

Store: CONFIGURATION
    IMPORTS DiskChannel, PilotSwitches, Process, ProcessorFace, ResidentHeap, Runtime, RuntimeInternal, SpecialTransaction,
            StoragePrograms, System, SystemInternal, Utilities, Zone
    EXPORTS File, KernelFile, PhysicalVolume, ResidentMemory, Scavenger, Space, SpecialFile, SpecialSpace, SpecialTransaction,
            SpecialVolume, StoragePrograms, TemporaryTransaction, Transaction, Volume, VolumeExtras =

BEGIN

Filer;
Swapper;
FileMgr;
VMMgr;
Transactions;

END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

Time: April 11, 1980 4:41 PM By: Knutsen Action: Import (rather than exporting) ResidentMemory; import StoragePrograms
Time: May 21, 1980 11:12 AM By: Luniewski Action: Import Zone; export PhysicalVolume
Time: June 19, 1980 2:00 PM By: Luniewski Action: Export Scavenger
Time: June 30, 1980 11:20 PM By: Gobbel Action: Export Transaction
Time: July 16, 1980 11:21 PM By: Gobbel Action: Export SpecialTransaction
Time: August 1, 1980 3:56 PM By: Gobbel Action: SpecialTransaction = > TemporaryTransaction.
Time: August 12, 1980 12:20 PM By: McJones Action: Import ProcessorFace
*Time: August 29, 1980 5:27 PM By: Forrest Action: Kill PPD**
Time: September 25, 1980 4:44 PM By: Gobbel Action: Added import and export of SpecialTransaction.
Time: October 9, 1980 1:49 PM By: Luniewski Action: Added export of VolumeExtras.

```
PACK
    FileImpl, VolFileMapImpl, VolumeImpl;
    -- MarkerPageImpl, PhysicalVolumeImpl, ScavengeImpl, and VolAllocMapImpl will only be called infrequently, hence are not
       packed.

FileMgr: CONFIGURATION LINKS: CODE

    IMPORTS DiskChannel, FileCache, FilePageTransfer, FilerException, LabelTransfer, MStore--temporary--, PilotSwitches, Process,
            SimpleSpace, SubVolume, System, SystemInternal, Transaction, TransactionState, Utilities

    EXPORTS File, KernelFile, LogicalVolume, PhysicalVolume, Scavenger, SpecialFile, SpecialVolume, StoragePrograms, VolFileMap,
            Volume, VolumeExtras =
BEGIN

FileImpl;

MarkerPageImpl;

PhysicalVolumeImpl;

ScavengeImpl;

VolAllocMapImpl;

VolFileMapImpl;

VolumeImpl;

END.
```

LOG

For previous log entries, see version archived with Pilot 3.0.

January 28, 1980 4:48 PM	*Forrest*	*Suppressed export of Temporary Booting*
March 10, 1980 10:25 AM	*Forrest*	*Deleted Import of Space*
April 18, 1980 4:04 PM	*Luniewski*	*Added Export of Transaction.*
May 29, 1980 1:35 PM	*Luniewski*	*Added export of PhysicalVolume. Include PhysicalVolumeImpl and deleted VolumeImplB.*
June 17, 1980 4:39 PM	*Luniewski*	*Import Space. Export Scavenger.*
July 6, 1980 6:09 PM	*Gobbel*	*Import TransactionInternal.*
July 22, 1980 11:41 AM	*Fay*	*Import Transaction.*
August 1, 1980 3:54 PM	*Gobbel*	*Change TransactionInternal to SpecialTransaction.*
August 28, 1980 1:04 PM	*Knutsen*	*Change SpecialTransaction to TransactionState.*
September 9, 1980 5:52 PM	*Luniewski*	*Delete Import of Space.*

Time: September 15, 1980 10:40 AM By: Forrest Action: change to use code links

Time: October 9, 1980 1:47 PM By: Luniewski Action: Export VolumeExtras

```
PACK
    --AltoDiskImpl,
    FileCacheImpl,
    FilerControl,
    FilerExceptionImpl,
    FilerTransferImpl,
    FileTaskImpl,
    --RemotePageTransferImpl,
    SubVolumeImpl;

Filer: CONFIGURATION LINKS: CODE
    IMPORTS DiskChannel, Process, ResidentHeap, ResidentMemory, RuntimeInternal, Zone
    EXPORTS --AltoDisk,-- FileCache, FilePageTransfer, FilerException, LabelTransfer, StoragePrograms, SubVolume =
BEGIN

--AltoDiskImpl; + + archived on [Iris]<APilot>30>

FileCacheImpl;

FilerControl;

FilerExceptionImpl;

FilerTransferImpl;

FileTaskImpl;

--RemotePageTransferImpl; + + archived on [Iris]<APilot>30>

SubVolumeImpl;

END.
```

LOG
Time: January 30, 1980 4:17 PM By: Gobbel Action: AltoDiskImpl and RemotePageTransferImpl commented out
Time: May 19, 1980 5:17 PM By: Luniewski Action: Imports of ResidentHeap and Zone added
*Time: August 29, 1980 5:47 PM By: Forrest Action: Kill PPD**
Time: September 15, 1980 10:54 AM By: Forrest Action: Trim Long: Links: CODE.

```
PACK
    -- Resident Modules:
    CachedRegionImplA,
    CachedRegionImplB,
    CachedSpaceImpl,
    MStoreImpl,
    PageFaultImpl,
    ResidentMemoryImpl,
    SwapBufferImpl,
    SwapperControl,
    SwapperExceptionImpl,
    SwapTaskImpl;

    -- SimpleSpaceImpl is not logically a part of the Swapper, and does not need to be resident for its operation.  Thus is not packed
       with the resident modules.

Swapper: CONFIGURATION LINKS: CODE

    IMPORTS FilePageTransfer, PilotSwitches, Process, ProcessorFace, ResidentHeap, RuntimeInternal, StoragePrograms, Utilities

    EXPORTS CachedRegion, CachedRegionInternal, CachedSpace, MStore, ResidentMemory, SimpleSpace, SpecialSpace,
            StoragePrograms, SwapperException =
BEGIN

CachedRegionImplA; -- the region cache.
CachedRegionImplB; -- region operations.
CachedSpaceImpl;
MStoreImpl;
PageFaultImpl;
ResidentMemoryImpl;
SimpleSpaceImpl;
SwapBufferImpl;
SwapperControl;
SwapperExceptionImpl;
SwapTaskImpl;

END.
```

LOG

(For earlier log entries, see Pilot 4.0 archive version.)

April 16, 1980 12:31 PM	*Knutsen*	*import StoragePrograms*
July 22, 1980 11:46 AM	*Fay*	*Import Transaction*
August 12, 1980 12:09 PM	*McJones*	*Import ProcessorFace*
September 2, 1980 6:18 PM	*Knutsen*	*Don't import Transaction. Eleminate PPD*.*
September 15, 1980 10:55 AM	*Forrest*	*Links: CODE*

```
PACK
    TransactionLogImpl,
    TransactionStateImpl;
    -- TransactionImpl is not packed with the above two modules since its code is only needed at Commit[] and Abort[] time, whereas
       the above modules are needed continuously whenever a transaction is in progress.

Transactions: CONFIGURATION LINKS: CODE

    IMPORTS File, KernelFile, KernelSpace, PilotSwitches, Runtime, ResidentHeap, SimpleSpace, Space, SpecialTransaction, Volume

    EXPORTS SpecialTransaction, StoragePrograms, TemporaryTransaction, Transaction, TransactionState

    CONTROL -- (to initialize exported variables)
        TransactionImpl,
        TransactionStateImpl,
        TransactionLogImpl =

BEGIN

TransactionImpl LINKS: FRAME; -- since this imports SpecialTransaction.ClientStart into some boot files, it can't use code links

TransactionLogImpl;

TransactionStateImpl;

END.
```

LOG

June 15, 1980 10:42 PM	*Gobbel*	*Created file.*
September 2, 1980 5:44 PM	*Knutsen*	*Added TransactionStateImpl.*
September 19, 1980 4:12 PM	*Gobbel*	*Removed import of SystemInternal.*
September 25, 1980 4:43 PM	*Gobbel*	*Added import and export of SpecialTransaction.*
October 3, 1980 6:13 PM	*Gobbel*	*Added import and export of SpecialTransaction.*
October 3, 1980 6:15 PM	*Forrest*	*October 3, 1980 6:15 PM.*

```
PACK
    HierarchyImpl,
    MapLogImpl,
    ProjectionImpl,
    SpaceImplA,
    SpaceImplB,
    STLeafImpl,
    STreeImpl;
    -- VMMControl is not packed since it is only needed during initialization, and will be swapped out forever after that.

VMMgr: CONFIGURATION LINKS: CODE

    IMPORTS CachedRegion, CachedSpace, File, KernelFile, PilotSwitches, Process, ResidentHeap, Runtime, RuntimeInternal,
            SimpleSpace, StoragePrograms, SwapperException, SystemInternal, Transaction, TransactionState, Utilities, Volume

    EXPORTS KernelSpace, Space, SpecialSpace, StoragePrograms =

BEGIN

-- Put the two instances of STreeImpl in separate configurations to get around debugger Set Octal Context bug

HConfig: CONFIGURATION
    IMPORTS CachedSpace, ResidentHeap, STLeaf, Transaction, Utilities EXPORTS Hierarchy, STree, VMMPrograms =
    BEGIN
    HierarchyImpl[CachedSpace, STree, Transaction];
    STree ← STreeImpl[];
    END;

PConfig: CONFIGURATION
    IMPORTS CachedRegion, ResidentHeap, STLeaf, Transaction, Utilities EXPORTS Projection, STree, VMMPrograms =
    BEGIN
    ProjectionImpl[CachedRegion, STree];
    STree ← STreeImpl[];
    END;

[Hierarchy, STreeHier: STree, VMMPrograms] ← HConfig[];

[Projection, STreeProj: STree, VMMPrograms] ← PConfig[];

MapLogImpl;

SpaceImplA;

SpaceImplB;

STLeafImpl;

VMMControl[CachedRegion, CachedSpace, Hierarchy, MapLog, Projection, SimpleSpace, StoragePrograms, STreeHier, STreeProj,
           VMMPrograms];

END.
```

LOG

For entries previous to Dec 17, 1979, please see version archived with Pilot 3.0.

April 8, 1980 5:20 PM	*Knutsen*	*VMMControl now imports StoragePrograms. Don't PACK VMMControl.*
April 18, 1980 4:07 PM	*Luniewski*	*Added Import of Transaction.*
May 29, 1980 3:49 PM	*Knutsen*	*Don't import PPData, PPDStorage.*
June 27, 1980 5:56 PM	*Gobbel*	*SpaceImpl split into SpaceImplA and SpaceImplB.*
August 1, 1980 11:22 AM	*Knutsen*	*Add Transaction to STreeImpl IMPORTs in H/PConfig and to HierarchyImpl, export KernelSpace.*
August 28, 1980 12:51 PM	*Knutsen*	*SpaceInternal renamed to KernelSpace, SpecialTransaction to TransactionState.*
September 15, 1980 11:02 AM	*Forrest*	*Links: CODE.*

```
PACK -- these modules are resident
  BootSwapCross,
  Instructions,
  PilotNub, -- can some of this be swappable eventually?
  Processes, -- resident because Swapper's ReplacementProcess is using Yield
  Signals, -- should be residentDescriptor or swappable
  Traps;

MesaRuntime: CONFIGURATION LINKS: CODE
  IMPORTS DebuggerSwap, File, KernelFile, KeyboardFace, PhysicalVolume,
    PilotSwitches, ProcessorFace, ResidentMemory, Space, SpecialFile, SpecialVolume,
    StoragePrograms, TemporarySetGMT, Transaction, Utilities
  EXPORTS BootSwap, DeviceCleanup, Process, ProcessInternal, Runtime,
    RuntimeInternal, RuntimePrograms, Snapshot, TemporaryBooting =

BEGIN

BootSwapCross;
FrameImpl;
Instructions;
PilotNub;
Processes;
Signals;
SnapshotImpl;
Traps;

END.
```

LOG

Time: April 8, 1980 1:20 PM	*By: Knutsen*	*Action: MStore = > StoragePrograms*
Time: April 16, 1980 5:38 PM	*By: McJones*	*Action: ControlPrograms = > TemporarySetGMT*
Time: April 30, 1980 4:58 PM	*By: Forrest*	*Action: Move perf stuff to RuntimePerf*
Time: June 26, 1980 10:38 AM	*By: McJones*	*Action: OISProcessorFace = > ProcessorFace; import Transaction*
Time: August 1, 1980 2:28 PM	*By: Luniewski*	*Action: Import PhysicalVolume; Frames = > FrameImpl*
Time: August 26, 1980 10:36 AM	*By: McJones*	*Action: Merge InterruptKey with PilotNub*
Time: September 17, 1980 2:35 PM	*By: McJones*	*Action: Don't import SystemInternal*

```
-- RuntimeLoader.config (last edited by: Sandman on: September 15, 1980 11:46 AM)

RuntimeLoader: CONFIGURATION LINKS: CODE
 IMPORTS File, Heap, Runtime, RuntimeInternal, RuntimePrograms,
  Space, SpecialSpace, Transaction
 EXPORTS BcdOps, PilotLoadStateOps, Runtime, RuntimePrograms, SubSys
 CONTROL PilotLoadState =
 BEGIN
 PilotLoaderCore;
 PilotLoaderSupport;
 PilotLoadState;
 PilotUnLoader;
 BcdOperations;
 END.

LOG
Time: January 31, 1980 12:48 AM      By: Forrest    Action: Create from Tajo>PilotLoader.config
Time: June 26, 1980 2:04 PM          By: Luniewski  Action: Export BcdOps and
  PilotLoadStateOps
Time: June 26, 1980 3:01 PM          By: McJones    Action: Import Transaction
Time: August 15, 1980 3:46 PM        By: Fay        Action: Replaced PilotBcdOperations by BcdOperations (from Mesa group).
Time: Aug 27, 1980 11:30 AM          By: Sandman    Action: Import Heap.
Time: September 15, 1980 11:47 AM    By: Howard     Action: LINKS: CODE.
```

```
PACK
  PilotCounter,
  PilotPerfMonitor;

RuntimePerf: CONFIGURATION LINKS: CODE
  IMPORTS ProcessInternal, ProcessorFace, ProcessOperations, Space, SpecialSpace, System, Transaction
  EXPORTS PerformancePrograms =

BEGIN
PilotCounter;
PilotPerfMonitor;
END.
```

LOG

Time: April 30, 1980 4:40 PM	*By: Forrest*	*Action: Create from MesaRuntime*
Time: June 26, 1980 10:44 AM	*By: McJones*	*Action: Import Transaction*
Time: July 29, 1980 11:25 AM	*By: McJones*	*Action: Don't import KeyboardFace or TemporaryBooting*
Time: August 29, 1980 5:20 PM	*By: Forrest*	*Action: Kill PPDataImpl*
Time: September 19, 1980 11:14 AM	*By: Johnsson*	*Action: delete PerfBreakHandler; add imports*

```
PACK -- resident code
    ResidentHeapImpl, StreamImpl, UtilitiesImpl, ZoneImpl;

Misc: CONFIGURATION LINKS: CODE
    IMPORTS PilotSwitches, Process, ResidentMemory, Runtime, RuntimeInternal, Space, SpecialSpace, SystemInternal, Transaction
    EXPORTS ByteBlt, Heap, MiscPrograms, ResidentHeap, SpecialHeap, Stream, Utilities, Zone =

BEGIN

HeapImpl;

ResidentHeapImpl;

StreamImpl;

UtilitiesImpl;

ZoneImpl;

END.
```

LOG

(For earlier log entries see Teak archive version.)
Time: January 27, 1980 6:26 PM By: Forrest Action: Export Space
Time: January 28, 1980 1:33 PM By: Forrest Action: Eliminate export of Space
Time: January 28, 1980 4:47 PM By: Forrest Action: Import SystemInternal for new StreamImpl
Time: April 30, 1980 4:51 PM By: Forrest Action: Move PPDataImpl to RuntimePerf.config
Time: June 26, 1980 9:48 AM By: McJones Action: Add HeapImpl; import PilotSwitches, Transaction
Time: July 22, 1980 4:12 PM By: Fay Action: Export SpecialHeap.
Time: September 15, 1980 11:05 AM By: Forrest Action: Links: CODE.

```
Communication: CONFIGURATION LINKS: CODE
 IMPORTS
  ByteBlt, Dialup, EthernetFace, EthernetOneFace, Heap, Inline,
  PilotSwitches, Process, ProcessInternal, ResidentHeap, RS232C,
  Runtime, Space, SpecialSpace, SpecialSystem, StatsDefs,
  Stream, System, Transaction
 EXPORTS
  -- public communication --
   NetworkStream, OISTransporter, CommunicationPrograms,
   PhoneNetwork, Socket, SpecialCommunication,
  -- pup private -- BufferDefs, CommUtilDefs, DriverDefs, PupDefs,
  -- communication private --
   OISCPDefs, StatsDefs, StatsOps, SocketInternal,
  -- OISCP private -- PacketStream, Router, Checksums
 CONTROL CommunicationControl =
BEGIN

 QueuePackage: CONFIGURATION
  IMPORTS
   BufferDefs, CommUtilDefs, DriverDefs, Process, PupDefs, StatsDefs
  EXPORTS BufferDefs, DriverDefs, OISCPDefs, PupDefs
  CONTROL SystemBufferPoolImpl =
   BEGIN
   QueueImpl; -- STARTED by SystemBufferPoolImpl.MainlineCode
   BufferPoolImpl; -- STARTED by SystemBufferPoolImpl.MainlineCode
   SystemBufferPoolImpl; -- STARTED by Boss.MainlineCode
   END;

 LevelO: CONFIGURATION
  IMPORTS
   BufferDefs, CommUtilDefs, Dialup, DriverDefs, EthernetFace,
   EthernetOneFace, HalfDuplex, Heap, Inline, PilotSwitches, Process,
   ProcessInternal, RS232C, Runtime, SpecialSystem,
   StatsDefs, System
  EXPORTS CommUtilDefs, DriverDefs, OISTransporter, PhoneNetwork
  CONTROL DispatcherImpl =
   BEGIN
   EthernetDriver; -- not STARTED ? (CreateDefaultEthernetDrivers from Boss)
   EthernetOneDriver; -- not STARTED ? (CreateDefaultEthernetOneDrivers from Boss)
   DispatcherImpl; -- STARTED by Boss.MainlineCode
   -- Follow three need frame links because they have imports that come into .boot files.
   PhoneNetworkDriver LINKS: FRAME; -- not STARTED
   PhoneNetworkImpl LINKS: FRAME; -- not STARTED
   HalfDuplexImpl LINKS: FRAME; -- not STARTED
   END;

 Level1: CONFIGURATION
  IMPORTS
   BufferDefs, ByteBlt, CommUtilDefs, DriverDefs, Heap, Inline,
   OISCPDefs, Process, SpecialSystem, StatsDefs, System
  EXPORTS
   Checksums, CommunicationInternal, OISCPDefs, Router, Socket,
   SocketInternal, SpecialCommunication
  CONTROL ChecksumsImpl =
   BEGIN
   RouterImpl; -- STARTED by CommunicationControl.InitializeCommunication
   RoutingTableImpl; -- STARTED by CommunicationControl.InitializeCommunication
   SocketImpl; -- STARTED by CommunicationControl.InitializeCommunication
   ChecksumsImpl; -- STARTED by CommunicationControl.InitializeCommunication
   END;

 Level2: CONFIGURATION
  IMPORTS BufferDefs, ByteBlt, DriverDefs, Heap, OISCPDefs, Process, Router,
   Runtime, Socket, SocketInternal, StatsDefs, Stream, System
  EXPORTS CommunicationInternal, Echo, NetworkStream, PacketStream
  CONTROL EchoServerImpl =
   BEGIN
   EchoServerImpl; -- STARTED by CommunicationControl.InitializeCommunication
   NetworkStreamMgr; -- STARTED by CommunicationControl.InitializeCommunication
   NetworkStreamInstance; -- multiple instances STARTED by NetworkStreamMgr
```

PacketStreamMgr; -- *STARTED by CommunicationControl.InitializeCommunication*
PacketStreamInstance; -- *multiple instances STARTED by PacketStreamMgr*
END;

CommunicationControl; -- *STARTED by PilotControl*
PilotCommUtil; -- *STARTED by CommunicationControl.InitializeCommunication*
Boss; -- *STARTED by CommunicationControl.InitializeCommunication*
StatsHot; -- *STARTED by CommunicationControl.InitializeCommunication*
QueuePackage;
Level0;
Level1;
Level2;

END.

LOG

Time: January 8, 1980 1:18 PM By: Dalal Action: conversion to Pilot 4.0.
Time: May 6, 1980 6:19 PM By: BLyon Action: Temp removed CommunicationPrograms from the EXPORTS. Removed D0InputOutput from IMPORTS; replaced D0EthernetDefs with EthernetOneFace; *Added CONTROL clause so that communications can be built/started seperately from TestPilotKernel .. tajo.*
Time: July 2, 1980 11:53 AM By: Garlick Action: Added PhoneNetworkDriver to Level 0 and PhoneNetworkImpl to main config.
Time: July 7, 1980 11:18 AM By: BLyon Action: Made config for Pilot integration.
Time: August 11, 1980 4:56 PM By: BLyon Action: Put in CONTROLs for start trapping and changed ByteBltDefs to ByteBlt.
Time: September 6, 1980 2:57 PM By: HGM Action: IMPORT EthernetFace and PilotSwitches.
Time: September 15, 1980 10:38 AM By: Forrest Action: Code Links.

```
PACK UserTerminalImpl, StartChainPlug;

UserTerminalDriver: CONFIGURATION LINKS: CODE
    IMPORTS DisplayFace, KeyboardFace, MouseFace, PilotSwitches, Process, ProcessInternal, ResidentHeap, Runtime, Space,
        SpecialSpace, System, Transaction
    EXPORTS DriverStartChain, UserTerminal =

BEGIN

UserTerminalImpl[DisplayFace, DriverStartChain1, KeyboardFace, MouseFace, PilotSwitches, Process, ProcessInternal, ResidentHeap,
        Runtime, Space, SpecialSpace, System, Transaction];

[DriverStartChain1, HeadStartChain, StoreDriverStartChain] ← StartChainPlug[];

END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)
Time: April 11, 1980 4:53 PM By: Forrest Action: Fix imports in BitBltImpl
Time: June 26, 1980 4:24 PM By: McJones Action: Import Transaction into config and UserTerminalImpl
Time: July 26, 1980 12:43 AM By: Forrest Action: Kill BitBltImpl
Time: July 29, 1980 11:18 AM By: McJones Action: Import MouseFace into config and UserTerminalImpl

```
PACK
    DiskChannelImpl,
    DiskDriverSharedImpl,
    SA4000Impl,
    StartChainPlug;

PACK
    FloppyChannelImpl,
    SA800Impl;

DiskDrivers: CONFIGURATION LINKS: CODE
    IMPORTS DriverStartChain, Inline, PhysicalVolume, PilotDisk, Process, ProcessInternal, ResidentHeap, RuntimeInternal,
        SA4000Face, SA800Face, Space, SpecialSpace, Transaction, Utilities
    EXPORTS DiskChannel, DiskChannelBackend, DiskDriverShared, DriverStartChain1, FloppyChannel, FloppyChannelInternal--for use
    by disk formatters only--, PhysicalVolume, StoreDriverStartChain1 =

BEGIN

DiskChannelImpl;

DiskDriverSharedImpl;

StoreDriverStartChain1 ← SA4000Impl[DiskChannelBackend, DiskDriverShared, Inline, Process, ProcessInternal, ResidentHeap,
    RuntimeInternal, SA4000Face, StoreDriverStartChain, Utilities];

[DiskChannel, DriverStartChain1, FloppyChannel, FloppyChannelInternal, PhysicalVolume] ←
    FloppyChannelImpl[DiskChannelBackend, DriverStartChain2, Inline, PhysicalVolume, Process, ProcessInternal, ResidentHeap,
    SA800Face, Space, SpecialSpace, Transaction]; -- FloppyChannelImpl must be started before SA800Impl

[DriverStartChain2, PhysicalVolume] ← SA800Impl[DiskChannel, DiskChannelBackend, DriverStartChain, FloppyChannel,
FloppyChannelInternal, Inline, PilotDisk, Process, RuntimeInternal, SA800Face, Space, SpecialSpace, Transaction, Utilities];

-- ... other disk drivers ...

[DriverStartChainDontCare, HeadStartChainDontCare, StoreDriverStartChain] ← StartChainPlug[];

END.
```

LOG

Time: April 13, 1980 10:24 PM By: Forrest Action: Trim log to Amarsosa; move in DiskChannelImpl; delete comments refering to Model 31
Time: July 19, 1980 1:42 PM By: Jose Action: Export DiskDriverShared.
Time: August 13, 1980 4:10 PM By: Jose Action: Add Floppy Disk driver.
Time: September 15, 1980 5:40 PM By: Jose Action: Add FloppyChannel to SA800Impl imports.
Time: September 20, 1980 3:30 PM By: Forrest Action: Code links.
Time: October 1, 1980 12:42 PM By: Jose Action: Export FloppyChannelInternal.

```
-- LastEdited: October 10, 1980  4:39 PM  By: Artibee

RS232CIO: CONFIGURATION
    LINKS: CODE
    IMPORTS Heap, OISTransporter, Process, RS232CFace, RS366Face
    EXPORTS Dialup, RS232C, RS232CManager
    CONTROL RS232CDriverA =
BEGIN
-- channel and reserving it
    RS232CDriverA;
    RS232CDriverB;
    RS232CManagerImpl;
-- dialing
    DialupImpl;
END.
```

LOG

Time: June 6, 1979 11:29 AM By: VSS Action: Created file
Time: May 23, 1980 5:57 PM By: VSS Action: removed all pre-Amargosa log entries
Time: May 23, 1980 5:57 PM By: VSS Action: Replaced RS232CFrontEnd with RS232CDriverA and RS232CDriverB, as part of code to handle multiple RS232C Channels.
Time: June 24, 1980 4:07 PM By: VSS Action: Remove OISTransporterImpl and PhoneNetworkImpl from this configuration (they go to Communication.config).
Time: July 22, 1980 3:12 PM By: CRF Action: Add Transaction to the IMPORTS for RS232CHeapImpl's use.
Time: August 4, 1980 2:56 PM By: VSS Action: Remove RS232CHeapImpl, and IMPORT Heap instead. Remove IMPORTs of Space, SpecialSpace and Zone.
Time: October 10, 1980 4:39 PM By: Artibee . Action: Added LINKS: CODE.

```
HeadsD0: CONFIGURATION
    IMPORTS ByteBlt, DeviceCleanup, Heap, Process, ProcessInternal, ResidentHeap, Runtime, RuntimeInternal, Space, SpecialHeap,
        SpecialSpace, Zone
    EXPORTS DisplayFace, EthernetFace, EthernetOneFace, HeadStartChain, KeyboardFace, MouseFace, ProcessorFace,
        RS232CFace, RS366Face, SA4000Face, SA800Face, TTYPortFace, TemporarySetGMT =

BEGIN

BasicHeadsD0;

RS232CIOHeadsD0;

END.
```

LOG

Time: April 21, 1980 5:34 PM By: McJones Action: Create file

Time: August 26, 1980 10:58 AM By: BLyon Action: renamed EthernetFace to EthernetOneFace

Time: September 6, 1980 3:47 PM By: HGM Action: EXPORT EthernetFace

Time: October 3, 1980 5:33 PM By: Forrest Action: Remove SoftwareTextBlt export; add TemporarySetGMT

```
PACK -- resident
    ProcessorHeadD0,
    EthernetHeadD0,
    EthernetOneHeadD0,
    GMTUsingIntervalTimer,
    SA4000HeadD0,
    SA800HeadD0,
    UserTerminalHeadD0,
    StartChainPlug,
    SetGMTUsingEthernet;

BasicHeadsD0: CONFIGURATION LINKS: CODE
    IMPORTS DeviceCleanup, Inline, ProcessInternal, RDC, RuntimeInternal, SA800NeckD0
    EXPORTS D0InputOutput, DisplayFace, EthernetFace, EthernetOneFace, HeadStartChain1, KeyboardFace, MouseFace,
            ProcessorFace, SA4000Face, SA800Face, TemporarySetGMT =

BEGIN

[D0InputOutput, ProcessorFace1] ← ProcessorHeadD0[]; -- NOT in head start chain

[EthernetFace, HeadStartChain1] ← EthernetHeadD0[D0InputOutput, DeviceCleanup, HeadStartChain2, Inline];

[EthernetOneFace, HeadStartChain2] ← EthernetOneHeadD0[D0InputOutput, DeviceCleanup, HeadStartChain3, Inline];

[HeadStartChain3, ProcessorFace2] ← GMTUsingIntervalTimer[DeviceCleanup, HeadStartChain4, ProcessInternal, ProcessorFace];

[HeadStartChain4, SA4000Face, System] ← SA4000HeadD0[D0InputOutput, DeviceCleanup, HeadStartChain5, Inline, RDC];

[SA800Face, HeadStartChain5] ← SA800HeadD0[D0InputOutput, DeviceCleanup, HeadStartChain6, Inline, SA800NeckD0, System];

[DisplayFace, HeadStartChain6, KeyboardFace, MouseFace, ProcessorFace3] ← UserTerminalHeadD0[D0InputOutput, DeviceCleanup,
        HeadStartChain, Inline, ProcessorFace, RuntimeInternal];

ProcessorFace ← ProcessorFace1 THEN ProcessorFace2 THEN ProcessorFace3;

StartChainPlug;

SetGMTUsingEthernet;

END.
```

LOG

(For earlier log entries see Amargosa archive version.)

Time: April 21, 1980 5:34 PM By: McJones Action: Delete D0Ether import to EthernetHeadD0 and commented-out Diablo31 stuff

Time: April 28, 1980 2:04 PM By: Danielson/Schwartz Action: Remove RS232 head and its imports from config

Time: May 2, 1980 12:48 PM By: Forrest Action: Add PrincOpsBitBltImpl (temporary)

Time: May 14, 1980 6:40 PM By: McJones Action: Remove OISProcessorHeadD0 from HeadStartChain

Time: May 19, 1980 12:47 PM By: McJones Action: Define OISProcessorFace with THEN to avoid conflict with HeadStartChain.Start

Time: May 30, 1980 2:40 PM By: Forrest Action: Remove PrincOpsBitBltImpl

Time: June 26, 1980 10:33 AM By: McJones Action: OISProcessorFace = >ProcessorFace; don't import OISDisk; import RDC; SA4000HeadD0 exports System; UserTerminalHeadD0 doesn't import DisplayFace or KeyboardFace

Time: July 29, 1980 11:09 AM By: McJones Action: Export MouseFace from UserTerminalHeadD0 and config

Time: August 13, 1930 5:57 PM By: Jose Action: Add SA800HeadD0.

Time: August 15, 1980 3:54 PM By: Fay Action: Add TextBltImpl

Time: August 18, 1980 5:43 PM By: McJones Action: Renamed from HeadsD0

Time: August 20, 1980 6:29 PM By: BLyon Action: renamed Ethernet to EthernetOne&.*

Time: September 6, 1980 1:44 PM By: HGM Action: Add EthernetHeadD0, EXPORT EthernetFace.

Time: September 15, 1980 11:10 AM By: Forrest Action: Links: CODE.

Time: October 3, 1980 5:30 PM By: Forrest Action: remove software textBlt temporarily. This will be put back in when operable, but won't have to be exported since the unimplemented trap is now in the ProcessorHead.

```
RS232CIOHeadsD0: CONFIGURATION

    LINKS: CODE
    IMPORTS ByteBlt, DeviceCleanup, D0InputOutput, Heap, Process, ProcessInternal, ResidentHeap, Runtime, RuntimeInternal,
            Space, SpecialHeap, SpecialSpace, Zone
    EXPORTS RS232CFace, RS366Face, TTYPortFace =

BEGIN
RS232CHeadFrontEndD0;
MIOCCommCmdsD0;
MIOCCommImplD0;
MIOCHardwareD0;
TTYPortHeadD0;
X800TablesD0;
System6TablesD0;
RS232CAsyncMicrocodeD0;
RS232CByteMicrocodeD0;
RS232CBitMicrocodeD0;
RS232CTtyMicrocodeD0;
END.
```

LOG

Time: October 15, 1979 4:47 PM By: Bill Danielson Action: Created file
Time: April 29, 1980 8:50 AM By: Victor Schwartz Action: (See Amargosa archive for all pre-Amargosa-release log entries) Change name to RS232CIOHeadsD0.config as part of removal of RS232C stuff from TestPilot.config.
Time: May 1, 1980 1:03 PM By: Victor Schwartz Action: Added TTY microcode variant.
Time: July 22, 1980 3:15 PM By: Chuck Fay Action: Added Transaction to IMPORTS for RS232CHeapImpl's use.
Time: August 4, 1980 3:03 PM By: Danielson Action: Added TTYPort head to configuration.
Time: October 10, 1980 4:39 PM By: Artibee Action: Added LINKS: CODE.

```
-- To fully pack UtilityPilot, one should utter...
-- PACK PilotControl, FileImpl, FilerControl, UtilitySpaceImpl, SwapperControl, BootSwapCross, ResidentHeapImpl,
      CommunicationControl, UserTerminalImpl;
PACK
    PilotControl, SystemImpl;
PACK
    UserTerminalImpl, -- UserTerminalDriver
    DiskDriverSharedImpl, SA800Impl; -- DiskDrivers

UtilityPilot: CONFIGURATION
    IMPORTS
        Dialup, PilotClient, RS232C, RuntimePrograms--Pilot Loader--,
        --heads-- DisplayFace, EthernetFace, EthernetOneFace, HeadStartChain, KeyboardFace, MouseFace, ProcessorFace,
        SA4000Face, SA800Face, TemporarySetGMT,
        -- inlines to make Binder happy -- Inline, PilotDisk
    EXPORTS
        -- public -- ByteBlt, File, Heap, NetworkStream, OISTransporter, PhoneNetwork, PhysicalVolume, Process, Runtime, Scavenger,
        Socket, Space, Stream, System, TemporaryBooting, Transaction, UserTerminal, Volume, Zone,
        -- Communication private -- OISCPDefs, StatsDefs, StatsOps,
        -- for PUP package only -- BufferDefs, CommUtilDefs, DriverDefs, PupDefs,
        -- for test puposes/insiders/heads only -- DebuggerSwap, DeviceCleanup, DiskChannel, FloppyChannel, FloppyChannelInternal,
        KernelFile, PilotSwitches, ProcessInternal, PacketStream, ResidentHeap, ResidentMemory, Router, RuntimeInternal,
        Snapshot, SpecialFile, SpecialHeap, SpecialSpace, SpecialSystem, SpecialVolume, SystemInternal, Utilities, VolumeExtras =

BEGIN

UtilityPilotKernel[Dialup, DiskChannel, DriverStartChain1, EthernetFace, EthernetOneFace, KeyboardFace, HeadStartChain, PilotClient,
    ProcessorFace, RS232C, RuntimePrograms --(for loader)--, StoreDriverStartChain, TemporarySetGMT];

[DiskChannel, DiskChannelBackend, DiskDriverShared, DriverStartChain1, FloppyChannel, FloppyChannelInternal, PhysicalVolume,
    StoreDriverStartChain] ←
    DiskDrivers[DriverStartChain, Inline, PhysicalVolume, PilotDisk, Process, ProcessInternal, ResidentHeap, RuntimeInternal,
    SA4000Face, SA800Face, Space, SpecialSpace, Transaction, Utilities];

[DriverStartChain, UserTerminal] ←
    UserTerminalDriver[DisplayFace, KeyboardFace, MouseFace, PilotSwitches, Process, ProcessInternal, ResidentHeap, Runtime,
    Space, SpecialSpace, System, Transaction];

END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

Time: April 16, 1980 6:21 PM	*By: McJones*	*Action: Add SetGMTUsingEthernet*
Time: April 13, 1980 11:01 PM	*By: Forrest*	*Action: Don't export IOCS*
Time: April 21, 1980 4:01 PM	*By: Luniewski*	*Action: Delete SetGMTUsingEthernet from leading comment*
Time: May 1, 1980 9:55 AM	*By: Forrest*	*Action: Add SetGMTUsingEthernet to packs*
Time: May 1, 1980 9:55 AM	*By: Forrest*	*Action: Change UtilityHeadsD0 to HeadsD0*
Time: June 11, 1980 5:46 PM	*By: Luniewski*	*Action: Export PhysicalVolume*
Time: June 18, 1980 10:36 AM	*By: Luniewski*	*Action: Export Scavenger*
Time: June 27, 1980 11:19 AM	*By: McJones*	*Action: Export Heap, OISTransporter, PhoneNetwork, Transaction; import Dialup, RS232C; OISProcessor = > Processor, StatsPrivateDefs = > StatsOps*
Time: July 30, 1980 8:36 AM	*By: Forrest*	*Action: Move out heads, StatsPrivateDefs = > StatsOps*
Time: August 18, 1980 5:35 PM	*By: McJones*	*Action: Link floppy driver into DriverStartChain; import SA800Face; don't export ByteBltDefs*
Time: August 29, 1980 8:55 AM	*By: BLyon*	*Action: renamed EthernetFace to EthernetOneFace.*
Time: September 9, 1980 5:37 PM	*By: Forrest*	*Action: add EthernetFace.*
Time: September 15, 1980 11:18 AM	*By: Forrest*	*Action: LINKS: CODE. Add SpecialHeap to exports.*
Time: October 3, 1980 2:18 PM	*By: Jose*	*Action: export FloppyChannel and FloppyChannelInternal.*
Time: October 3, 1980 10:35 PM	*By: Forrest*	*Action: Import TemporarySetGMT.*
Time: October 9, 1980 2:02 PM	*By: Luniewski*	*Action: Export VolumeExtras.*

```
PACK BootSwapCross, FrameImpl, SnapshotImpl; -- MesaRuntime
PACK HeapImpl, ResidentHeapImpl; -- Misc
PACK QueueImpl, BufferPoolImpl, SystemBufferPoolImpl, EthernetDriver, EthernetOneDriver, DispatcherImpl, PhoneNetworkDriver,
PhoneNetworkImpl, HalfDuplexImpl, RouterImpl, RoutingTableImpl, SocketImpl, ChecksumsImpl, EchoServerImpl, NetworkStreamMgr,
NetworkStreamInstance, PacketStreamMgr, PacketStreamInstance, CommunicationControl, PilotCommUtil, Boss, StatsHot; --
Communication

UtilityPilotKernel: CONFIGURATION
    IMPORTS
        Dialup, DiskChannel, DriverStartChain, EthernetFace, EthernetOneFace, KeyboardFace, HeadStartChain, PilotClient,
        ProcessorFace, RS232C, RuntimePrograms --(for loader)--, StoreDriverStartChain, TemporarySetGMT
    EXPORTS
        -- public -- ByteBlt, File, Heap, NetworkStream, OISTransporter, PhoneNetwork, PhysicalVolume, Process, Runtime, Scavenger,
        Socket, Space, Stream, System, TemporaryBooting, Transaction, Volume, Zone,
        -- for Drivers Only -- DeviceCleanup, PilotSwitches, ProcessInternal, ResidentHeap, RuntimeInternal, SpecialSpace, Utilities,
        -- Communication private -- OISCPDefs, StatsDefs, StatsOps,
        -- for PUP package only -- BufferDefs, CommUtilDefs, DriverDefs, PupDefs,
        -- for insiders only -- DebuggerSwap, KernelFile, PacketStream, ResidentMemory, Router, Snapshot, SpecialFile, SpecialHeap,
        SpecialSystem, SpecialVolume, SystemInternal, VolumeExtras =

BEGIN
Control;
MesaRuntime;
-- RuntimeLoader is not used.
-- RuntimePerf is not used.
Misc;
UtilityStore;
Communication;
END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

Time: April 13, 1980 10:21 PM	*By: Forrest*	*Action: Remove IO*
Time: April 17, 1980 2:43 PM	*By: Knutsen*	*Action: Actually remove IO*
Time: May 1, 1980 9:54 AM	*By: Forrest*	*Action: Fix up packs*
Time: June 11, 1980 5:48 PM	*By: Luniewski*	*Action: Export PhysicalVolume*
Time: June 18, 1980 10:37 AM	*By: Luniewski*	*Action: Export Scavenger*
Time: June 27, 1980 11:16 AM	*By: McJones*	*Action: Export Heap, OISTransporter, PhoneNetwork, Transaction; import Dialup, RS232C; OISProcessorFace = > ProcessorFace, StatsPrivateDefs = > StatsOps*
Time: August 1, 1980 2:38 PM	*By: Luniewski*	*Action: Frames = > FrameImpl*
Time: August 18, 1980 3:56 PM	*By: McJones*	*Action: Don't export ByteBltDefs*
Time: August 29, 1980 9:40 AM	*By: BLyon*	*Action: renamed Ethernet* to EthernetOne*.*
Time: September 9, 1980 5:31 PM	*By: Forrest*	*Action: Fix up packs*
Time: September 17, 1980 10:07 AM	*By: Forrest*	*Action: SpecialHeap added to exports*
Time: September 18, 1980 1:27 PM	*By: McJones*	*Action: Don't PACK Processes*
Time: October 9, 1980 2:00 PM	*By: Luniewski*	*Action: Export VolumeExtras*

```
PACK
    -- FileMgr:
    FileImpl,
    PhysicalVolumeImpl,
    ScavengeImpl,
    VolAllocMapImpl,
    MarkerPageImpl;

PACK
    -- Swapper:
    SwapTaskImpl,
    SimpleSpaceImpl;

PACK
    -- UtilityVMMgr:
    UtilitySpaceImpl,
    UtilityVMMControl;

UtilityStore: CONFIGURATION
    IMPORTS DiskChannel, PilotSwitches, Process, ProcessorFace, ResidentHeap, Runtime, RuntimeInternal, SpecialVolume,
            StoragePrograms, System, SystemInternal, Utilities, Zone
    EXPORTS File, KernelFile, MStore, PerformancePrograms, PhysicalVolume, ResidentMemory, RuntimePrograms, Scavenger,
            Space, SpecialFile, SpecialSpace, SpecialVolume, StoragePrograms, Transaction, Volume, VolumeExtras =

BEGIN

Filer;

FileMgr;

Swapper;

-- UtilityVMMgr:

    UtilitySpaceImpl LINKS: CODE;

    UtilityVMMControl LINKS: CODE;

END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

Time: April 17, 1980 2:32 PM	*By: Knutsen*	*Action: Import StoragePrograms, export ResidentMemory. Add UtilityVMMControl.*
Time: April 23, 1980 11:29 AM	*By: Knutsen*	*Action: Export RuntimePrograms*
Time: April 30, 1980 6:36 PM	*By: Forrest*	*Action: Export MiscPrograms, PerformancePrograms, PPData*
Time: June 4, 1980 3:56 PM	*By: Luniewski*	*Action: Import Zone*
Time: June 11, 1980 5:45 PM	*By: Luniewski*	*Action: Export PhysicalVolume*
Time: June 17, 1980 4:53 PM	*By: Luniewski*	*Action: Export Scavenger*
Time: June 27, 1980 11:57 AM	*By: McJones*	*Action: Export Transaction*
Time: August 18, 1980 3:54 PM	*By: McJones*	*Action: Import ProcessorFace*
Time: August 29, 1980 5:28 PM	*By: Forrest*	*Action: Kill PPD*; Links: CODE*
Time: October 9, 1980 1:59 PM	*By: Luniewski*	*Action: Export VolumeExtras*

Time: September 2, 1980 2:15 PM *By: McJones* *Action: Remove Interrupt Key*
Time: October 6, 1980 1:24 PM *By: Forrest* *Action: Export SpecialSpace (for DonateDedicatedRealMemory)*

```
PACK
    BootSwapGerm,
    BootSwapCross,
    TeledebugImpl,
    BootChannelDisk,
    BootChannelEther,
    MiniEthernetDriver,
    ProcessorHeadD0,
    SA4000HeadD0,
    EthernetOneHeadD0;

Germ: CONFIGURATION LINKS: CODE
    IMPORTS Boot, DeviceCleanup, Frame, Inline, PageMap, PilotDisk, PrincOpsRuntime, ProcessOperations, RDC, Trap =
BEGIN

[BCnull, HSCnull, ResidentMemory] ← BootSwapGerm[Boot, BC0, BootSwap, Frame, HSC0, Inline, ProcessorFace, PageMap,
                                                 PrincOpsRuntime, ProcessOperations, ResidentMemory, Teledebug, Trap];
BootSwapCross;

-- Teledebugger
TeledebugImpl;

-- BootChannel implementations

-- The modules implementing various types of BootChannel's are daisy chained together.
-- BootSwapGerm is the "real" importer of BootChannel and also exports BCnull, a dummy BootChannel to terminate the chain.
-- To add a new driver named "BootChannelNew" to the chain:
--   1) Find the end of the chain, i.e. "BCn ← BootChannelX[...BCnull...];"
--   2) Change it to "BCn ← BootChannelX[...BCn + 1...];"
--   3) Append after it "BCn + 1 ← BootChannelNew[...BCnull...];"

-- (Similar remarks apply to the HeadStartChain (HSCn) linking heads.)
BC0 ← BootChannelDisk[Boot, BC1, Inline, PilotDisk, SA4000Face];
BC1 ← BootChannelEther[Boot, BCnull, --EtherFace,-- MiniEthernetDefs, ProcessorFace, ResidentMemory];

-- Communication
MiniEthernetDriver;

-- Heads
ProcessorHeadD0; -- NOT in HeadStartChain
[HSC0, SA4000Face, System] ← SA4000HeadD0[D0InputOutput, DeviceCleanup, HSC1, Inline, RDC];
[EthernetOneFace, HSC1] ← EthernetOneHeadD0[D0InputOutput, DeviceCleanup, HSCnull, Inline];

END.
```

LOG

Time: February 7, 1980 2:54 PM	*By: Knutsen*	*Action: Create file from GermDisk.config of February 6, 1980*
Time: February 15, 1980 1:51 PM	*By: McJones*	*Action: OISProcessorFaceD0Impl = > OISProcessorHeadD0*
Time: March 11, 1980 11:16 AM	*By: Forrest*	*Action: Commented out Diablo31*
Time: April 10, 1980 12:29 PM	*By: Forrest*	*Action: Added Teledebug*
Time: April 22, 1980 5:56 PM	*By: McJones*	*Action: MiniD0Driver = > MiniEthernetDriver + EthernetHeadD0*
Time: May 14, 1980 6:44 PM	*By: McJones*	*Action: Remove OISProcessorHeadD0 from HeadStartChain*
Time: July 1, 1980 1:21 PM	*By: McJones*	*Action: 48-bit processor ids, OISProcessor... = > Processor...*
Time: July 22, 1980 2:53 PM	*By: Fay*	*Action: Reordered BootSwapGerm[...] to get PageMap and PrincOpsRuntime in the correct order.*
Time: August 21, 1980 10:05 AM	*By: McJones*	*Action: Renamed file Germ.config = > GermD0.config*
Time: August 29, 1980 11:20 AM	*By: BLyon*	*Action: Renamed Ether* to EthernetOne*.*

-- Last edited: September 3, 1980 12:33 AM By: Mary Artibee

```
TTYPortChannel: CONFIGURATION
    IMPORTS Heap, Process, ProcessInternal, TTYPortFace
    EXPORTS TTYPort CONTROL TTYPortDriverA, TTYPortDriverB =

BEGIN
TTYPortDriverA;
TTYPortDriverB;
END.
```

LOG

Time: July 30, 1980 8:20 AM By: Mary Artibee Action: Created.
Time: August 4, 1980 3:53 PM By: Mary Artibee Action: Import TTYPortFace (from RS232CIOHeadsD0).
Time: August 5, 1980 6:55 PM By: Mary Artibee Action: Import ResidentHeap, Zone.
Time: September 3, 1980 12:33 AM By: Mary Artibee Action: Remove ResidentHeap, Zone.

```
-- PilotControl.mesa (last edited by: Forrest on: October 10, 1980 4:41 PM)

DIRECTORY
  Boot USING [Location, LP, LVBootFiles, nullDiskFileID, pRequest, Request],
  BootSwap USING [
    GetPStartListHeader, Initialize, InitializeMDS, mdsiGerm, sPilotSwitches],
  CommunicationPrograms USING [InitializeCommunication],
  ControlPrograms USING [SystemImpl],
  DebuggerSwap USING [Parameters],
  Device USING [nullType, Type],
  DriverStartChain USING [Start],
  Environment USING [
    maxPagesInMDS, PageCount, PageNumber, PageOffset, wordsPerPage],
  File USING [Capability, ID, nullID, PageNumber, Permissions, read, write],
  Frame USING [Free, GetReturnFrame, SetReturnLink],
  HeadStartChain USING [Start],
  KernelFile USING [Pin],
  MiscPrograms,
  PerformancePrograms USING [InitializePilotCounter, InitializePilotPerfMonitor],
  PilotClient USING [Run],
  PilotMP USING [
    Code, cBadBootFile, cClient, cCommunication, cControl, cMap, cStorage],
  PilotSwitches USING [],
  PrincOps USING [StateVector, TrapLink],
  Process USING [Detach],
  ProcessOperations USING [WriteWDC],
  ProcessorFace USING [SetMP, Start],
  Runtime USING [CallDebugger],
  RuntimePrograms USING [
    InitializeFrames, InitializeInterrupt, Instructions, InitializePilotLoadState,
    InitializePilotNub, Processes, Signals, InitializeSnapshot, Start, Traps],
  SDDefs USING [SD],
  Space USING [
    CopyIn, Create, defaultWindow, Delete, Handle, Kill, LongPointer, Map,
    nullHandle, PageNumber, WindowOrigin],
  StartList USING [
    Entry, Index, Header, SpaceIndex, SpaceType, StartIndex, SwapUnitInfo, Base,
    VersionID],
  SpecialSystem USING [NetworkAddress, nullNetworkAddress],
  StoragePrograms USING [
    DescribeSpace, empty, free, HandleFromPage, InitializeFileMgr,
    InitializeFiler, InitializeMStore, InitializeRegionCacheA,
    InitializeRegionCacheB, InitializeResidentMemoryA, InitializeResidentMemoryB,
    InitializeSimpleSpace, InitializeSwapper, InitializeTransactionData,
    InitializeVMMgr, IsUtilityPilot, LongPointerFromPage, nullSpaceHandle, outlaw,
    PageFromLongPointer, RecoverTransactions, ReplacementProcess, SpaceOptions,
    StartListProc, SuperFromPage],
  StoreDriverStartChain USING [Start],
  System USING [],
  SystemInternal USING [Unimplemented],
  TemporaryBooting USING [Switches], -- Trap USING [WriteXTS],
  VMMapLog USING [
    Descriptor, EntryPointer, PatchTable, PatchTableEntry,
    PatchTableEntryPointer],
  XferTrap USING [WriteXTS];

PilotControl: PROGRAM
  IMPORTS
    Boot, BootSwap, CommunicationPrograms, ControlPrograms, DriverStartChain,
    Frame, HeadStartChain, KernelFile, MiscPrograms, PerformancePrograms,
    PilotClient, Process, ProcessOperations, ProcessorFace, Runtime,
    RuntimePrograms, Space, StoreDriverStartChain, StoragePrograms,
    SystemInternal, XferTrap

  EXPORTS DebuggerSwap, PilotSwitches, StoragePrograms, System
  SHARES File, RuntimePrograms =
  BEGIN OPEN Environment;
  PEntry: TYPE = POINTER TO StartList.Entry; -- for debugger

  --
  -- PilotSwitches
  switches: PUBLIC TemporaryBooting.Switches;

  --
  -- StoragePrograms
  countVM: PUBLIC PageCount ← 40000B;
  -- size of VM implemented on a D0. Should come from ProcessorFace! ("←" due to compiler glit
**ch)
  pageMDS: PUBLIC Environment.PageNumber; -- first page of Pilot's (only) MDS.
  lpMDS: PUBLIC LONG POINTER; -- to Pilot's (only) MDS.
  pageHyperspace: PUBLIC Environment.PageNumber;
  -- first page of non-MDS non-Germ VM. (Hyperspace is assumed to follow MDS.)
  countHyperspace: PUBLIC Environment.PageCount;
  pageMDSGerm: PUBLIC Environment.PageNumber; -- first page of Germ's MDS.
  tableBase: PUBLIC StartList.Base; -- base pointer of the Start List.

  --
  -- System (exported here until directed exports appear)
  NetworkAddress: PUBLIC TYPE = SpecialSystem.NetworkAddress;
  nullNetworkAddress: PUBLIC NetworkAddress ← SpecialSystem.nullNetworkAddress;

  --
  -- DebuggerSwap
  canSwap: PUBLIC BOOLEAN ← FALSE; -- can we get to the debugger
  parameters: PUBLIC DebuggerSwap.Parameters;

  -- Parameters:
  request: Boot.Request ← LOOPHOLE[Boot.LP[
    highbits: BootSwap.mdsiGerm, lowbits: Boot.pRequest], LONG POINTER TO
    Boot.Request]↑; -- parameter to the germ which loaded us

  -- Assorted global variables:
  h: POINTER TO StartList.Header;
  -- Map log and patch table descriptor (see also CachedRegionImpl):
  mapLogDescriptor: VMMapLog.Descriptor ←
    [self:, writer: FIRST[VMMapLog.EntryPointer],
      reader: FIRST[VMMapLog.EntryPointer], limit: FIRST[VMMapLog.EntryPointer],
      patchTable: @dummyPatchTable.patchTable]; -- overwritten by CachedRegionImpl
  DummyPatchTable: TYPE = RECORD [
    patchTable: VMMapLog.PatchTable,
    entries: ARRAY [0..5) OF VMMapLog.PatchTableEntry];
  dummyPatchTable: DummyPatchTable ←
    [[limit: FIRST[VMMapLog.PatchTableEntryPointer],
      maxLimit:
      FIRST[VMMapLog.PatchTableEntryPointer] + 5*SIZE[VMMapLog.PatchTableEntry],
      entries:], ];
```

```
Bug: PRIVATE --PROGRAMMING--ERROR [type: BugType] = CODE;
BugType: TYPE = {bug0, bug1, bug2, bug3, bug4, bug5, bug6, bug7, bug8, bug9};

-- Heart of PilotControl
ProcessStartList: PROCEDURE = -- Assume called from main body
  BEGIN
  pSwitches: LONG POINTER TO TemporaryBooting.Switches = LOOPHOLE[Boot.LP[
    highbits: BootSwap.mdsiGerm, lowbits: @SDDefs.SD[BootSwap.sPilotSwitches]]];
  lvBootFiles: Boot.LVBootFiles;
  dT: Device.Type; -- device type for debugger boot files
  dO: CARDINAL; -- device ordinal for debugger boot files
  state: PrincOps.StateVector;
  pageEndHyperspace: Environment.PageNumber;
  pageExcessRealMemory: PageNumber; -- real pages not backing bootloaded stuff.
  pageFreeVM: PageNumber; -- VM following that allocated by StartPilot.
  Frame.Free[Frame.GetReturnFrame[]]; -- Get key switch settings.
  switches ← pSwitches↑;
  switches.u ← up; -- only Pilot sets UtilityPilot-ness

  ProcessorFace.SetMP[PilotMP.cControl];

  -- Get start traps working first.
  RuntimePrograms.Start[LOOPHOLE[RuntimePrograms.Traps]];

  -- Initialize processor face.
  ProcessorFace.Start[];

  BootSwap.InitializeMDS[]; -- sets pMon
  h ← BootSwap.GetPStartListHeader[];
  tableBase ← h.table;
  IF h.version ~= StartList.VersionID THEN Punt[PilotMP.cBadBootFile];
  -- not the version of StartList we were compiled with
  pageMDSGerm ← BootSwap.mdsiGerm*maxPagesInMDS;
  --pageMDS ← StoragePrograms.PageFromLongPointer[LOOPHOLE[1, POINTER]]; + + ("1" s
**ince "0" collides with NIL) (StoragePrograms version can be used in Mesa 6)
  pageMDS ← StoragePrograms.PageFromLongPointer[LOOPHOLE[1, POINTER]];
  lpMDS ← StoragePrograms.LongPointerFromPage[pageMDS];
  pageHyperspace ← pageMDS + maxPagesInMDS; -- follows MDS.
  pageEndHyperspace ←
    IF pageMDSGerm > pageHyperspace THEN pageMDSGerm
    ELSE FIRST[PageNumber] + countVM; -- ends at Germ or end of VM.
  countHyperspace ← pageEndHyperspace - pageHyperspace;
  pageFreeVM ← h.lastVMPage + 1;
  pageExcessRealMemory ← h.lastBootLoadedPage + 1;
  MiscPrograms.InitializeUtilities[]; -- (IMPORTed by Frames.)

  -- Initialize Mesa runtime.
  RuntimePrograms.InitializeFrames[];
  START RuntimePrograms.Instructions;
  Frame.SetReturnLink[PrincOps.TrapLink]; -- so it can be set to Processes.End
  START RuntimePrograms.Processes[pagePDA: h.pdaBase, countPDA: h.pdaPages];
  START RuntimePrograms.Signals;
  RuntimePrograms.InitializePilotNub[
    pageLoadState:
    tableBase[tableBase[h.initLoadState].parent].vmpage + tableBase[
      h.initLoadState].base, countLoadState: tableBase[h.initLoadState].pages,
    pVMMapLog: @mapLogDescriptor];
  BootSwap.Initialize[mdsiOther: BootSwap.mdsiGerm];
```



```
  InitializeDebuggerSwap[]; -- read in germ for debugger's world.
  -- InterruptKey is started below after KeyboardFace implementation is started.
  -- SnapshotImpl and PerformancePrograms are started below after the virtual memory is runni
**ng.

  -- Mesa runtime now fully operational; can swap if debugger pointers set with Othello comman
**d.
  IF switches.zero = down THEN Runtime.CallDebugger["Key Stop 0"];

  -- Get basic real and virtual memory facilities working.
  StoragePrograms.InitializeMStore[];
  StoragePrograms.DescribeSpace[
    StoragePrograms.free, pageExcessRealMemory,
    pageEndHyperspace - pageExcessRealMemory, Space.defaultWindow];
  -- gives excess real memory to MStore.
  StoragePrograms.InitializeSimpleSpace[];
  -- (enumerates non-empty initially resident spaces, initializes VM allocator).
  StoragePrograms.DescribeSpace[
    StoragePrograms.empty, pageFreeVM, pageEndHyperspace - pageFreeVM,
    Space.defaultWindow];
  -- tells SimpleSpace about unused hyperspace VM in which it can create SimpleSpaces.
  StoragePrograms.InitializeRegionCacheA[];
  -- creates region cache, initializes AllocateRuthlessly.
  StoragePrograms.InitializeResidentMemoryA[];

  -- ResidentMemory now operational.
  MiscPrograms.InitializeZone[];
  MiscPrograms.InitializeResidentHeap[];
  HeadStartChain.Start[]; -- start heads

  RuntimePrograms.InitializeInterrupt[];
  -- now that KeyboardFace.keyboard is defined
  START ControlPrograms.SystemImpl;

  -- Initialize more of Misc configuration.
  MiscPrograms.InitializeStream[];
  StoreDriverStartChain.Start[]; -- start drivers used by Store

  -- Initialize the Store configuration.
  BEGIN OPEN StoragePrograms;
  ProcessorFace.SetMP[PilotMP.cStorage];
  switches.u ← IF IsUtilityPilot[] THEN down ELSE up;
  InitializeFiler; -- Initialize Swapper.
  InitializeSwapper[@mapLogDescriptor]; -- SimpleSpace I/O is now operational.
  -- Describe initallyResident, cachedDescriptor, and outlaw Spaces to Swapper:
  InitializeRegionCacheB[]; -- describes region cache space.
  InitializeResidentMemoryB[]; -- describes resident memory spaces.
  DescribeSpace[
    StoragePrograms.outlaw, pageMDSGerm, maxPagesInMDS, Space.defaultWindow];
  -- describes Germ MDS space.
  DescribeSpace[
    StoragePrograms.outlaw, FIRST[PageNumber], 1, Space.defaultWindow];
  -- plant outlaw space on unmapped page 0 (LONG[NIL]↑).
  IF pageMDS # FIRST[PageNumber] THEN
    DescribeSpace[StoragePrograms.outlaw, pageMDS, 1, Space.defaultWindow];
  -- plant outlaw space on unmapped mds page 0 (NIL↑).
  EnumerateStartList[DescribeInitiallyResidentSpaces];
  -- describes non-empty initallyResident spaces.
```

```
-- Start the virtual memory replacement process.
BEGIN
countThreshold: PageCount ← 4;
-- for replacement algorithm in MStore.AwaitBelowThreshold.
Process.Detach[
  FORK ReplacementProcess[IF IsUtilityPilot[] THEN 0 ELSE countThreshold]];
END;

-- Part one of Transactions initialization: exported variables and simple spaces.
InitializeTransactionData[];

-- Start FileMgr, which will find system volume and debugger booting information.
[debuggerDeviceType: dT, debuggerDeviceOrdinal: dO] ← InitializeFileMgr[
  @LOOPHOLE[request.location, disk Boot.Location], @lvBootFiles];
IF ~switches.u = down THEN
  EnumerateStartList[PinResidentDescriptorPageGroups];
IF switches.t = up AND
  (dT ≠ Device.nullType OR lvBootFiles[debuggee].da ≠ Boot.nullDiskFileID.da)
  THEN switches.m ← down --disable map logging if no debugger--

ELSE
  BEGIN
  parameters.locDebugger ←
    [deviceType: dT, deviceOrdinal: dO,
      vp: disk[diskFileID: lvBootFiles[debugger]]];
  parameters.locDebuggee ←
    [deviceType: dT, deviceOrdinal: dO,
      vp: disk[diskFileID: lvBootFiles[debuggee]]];
  canSwap ← TRUE; -- now can get to debugger
  END;

InitializeVMMgr[countVM, @mapLogDescriptor];
-- also MapLogs the unitary and family spaces in the boot file

ProcessorFace.SetMP[PilotMP.cMap];
EnumerateStartList[CreateInitiallyOutSpaces];
-- creates swapUnits of non-initallyResident, non-cachedDescriptor spaces

-- Now everything allocated by StartPilot has been placed in the VMMgr databases and MapLo
**gged.
IF switches.one = down THEN Runtime.CallDebugger["Key Stop 1"];

RecoverTransactions[];
END; -- of OPEN StoragePrograms

PerformancePrograms.InitializePilotCounter[];
PerformancePrograms.InitializePilotPerfMonitor[];

RuntimePrograms.InitializeSnapshot[]; -- not initially resident
MiscPrograms.InitializeHeap[]; -- uses Space
DriverStartChain.Start[]; -- start remaining drivers
ProcessorFace.SetMP[PilotMP.cCommunication];
IF switches.c = up THEN CommunicationPrograms.InitializeCommunication[];
RuntimePrograms.InitializePilotLoadState[
  tableBase[tableBase[h.initLoadState].parent].vmpage + tableBase[
    h.initLoadState].base, tableBase[h.initLoadState].pages];
Space.Kill[
```



```
  -- so that UtilityPilot can do the Delete of the space
    StoragePrograms.HandleFromPage[
    StoragePrograms.PageFromLongPointer[tableBase]]];
  Space.Delete[
    StoragePrograms.HandleFromPage[
    StoragePrograms.PageFromLongPointer[tableBase]]]; -- delete the StartList

  -- Ready to start the client.
  ProcessorFace.SetMP[PilotMP.cClient];
  IF switches.two = down THEN Runtime.CallDebugger["Key Stop 2"L];
  state.instbyte ← state.stkptr ← 0;
  state.dest ← PilotClient.Run;
  state.source ← Frame.GetReturnFrame[];
  RETURN WITH state; -- transfer to PilotClient.Run, freeing our frame as we go

  END;

-- DescribeInitiallyResidentSpaces: Creates entries in the Swapper caches for (non-empty) initall
**yResident spaces.
-- The motivations for this section are as follows: The facility provided by the Swapper's Describ
**eSpace procedure is the only way to create and initialize space and region descriptors for data t
**hat are already in memory. It is also the only way to create cachedDescriptor space and region
**descriptors. Therefore, all initiallyResident or cachedDescriptor spaces and swapUnits must be
**processed by DescribeSpace, before the VMMgr is initialized. To simplify our life, we require th
**at all CachedDescriptor items be initiallyResident, thus restricting the processing here to only in
**itiallyResident items. (We don't process non-initiallyResident spaces and swapUnits here so as
**to minimize the amount of region cache space required during initialization, and to simplify the
**coding.)
-- VMMControl.InitializeVMMgr uses the space and region cache entries created here to constru
**ct the initial Hierarchy and Projection databases, and to MapLog the mapped spaces. For this p
**rocess, the requirements on the caches are: there must be a space cache entry for any space t
**hat is (itself) mapped; there must be a region cache entry for every swapUnit.

DescribeInitiallyResidentSpaces: StoragePrograms.StartListProc --[index]-- =
  -- Creates and maps spaces for: all initiallyResident unitary or family spaces.
  -- Creates regions for: all initiallyResident swap units, and unitary spaces.
  -- For the convenience of CreateInitiallyOutSpaces, the handle of each space entry is initialize
**d to Space.nullHandle.
  -- Note: We assume that a family is entirely tiled with swap units (no holes).
  BEGIN
  defaultOptions: StoragePrograms.SpaceOptions =
    [ -- (the most probable options)
      -- The program logic below assumes these options as defaults!
      createSpace: FALSE, -- assume.
      pinSpaceD: FALSE, -- assume.
      mapped: TRUE, -- any non-empty space or swap unit is mapped.
      createRegion: TRUE, -- assume.
      pinRegionD: FALSE, -- assume.
      subspace: FALSE, -- assume.
      initiallyResident: TRUE, pinned: FALSE]; -- assume.
  options: StoragePrograms.SpaceOptions ← defaultOptions;
  -- set up the most probable options.
  WITH e: tableBase[index] SELECT FROM
    space => -- create space descriptor:
      BEGIN
      IF e.vmpage + e.pages > h.lastVMPage + 1 THEN ERROR Bug[bug1];
      e.handle ← LOOPHOLE[StoragePrograms.nullSpaceHandle];
      -- for CreateInitiallyOutSpaces. (Note: Space.nullHandle is not initialized yet.)
```

```
    IF ~e.bootLoaded THEN RETURN;
    WITH t: e.type SELECT FROM
      empty => RETURN; -- (handled separately by InitializeSimpleSpace)

      unitary =>
        options.pinSpaceD ←
          (tableBase[t.swapUnit].info.state = residentDescriptor);
      family => options.pinSpaceD ← t.anyResidentDescriptorChildren;
      ENDCASE => ERROR Bug[bug2];
    options.createSpace ← TRUE;
    options.createRegion ← FALSE;
    StoragePrograms.DescribeSpace[
      options, e.vmpage, e.pages, WindowForSpace[LOOPHOLE[index]]];
    END;
  swapUnit => -- create region descriptor:
    BEGIN
    readOnlyWindow: Space.WindowOrigin =
      [file: [File.nullID, File.read], base: 0];
    readWriteWindow: Space.WindowOrigin =
      [file: [File.nullID, File.read + File.write], base: 0];
    IF tableBase[e.parent].bootLoaded THEN
      BEGIN
      options.subspace ← (tableBase[e.parent].type.class = family);
      SELECT e.info.state FROM
        resident => options.pinned ← TRUE;
        residentDescriptor => options.pinRegionD ← TRUE;
        swappable => --options.mapped ← TRUE--NULL;
        ENDCASE => ERROR Bug[bug0];
      StoragePrograms.DescribeSpace[
        options, tableBase[e.parent].vmpage + e.base, e.pages,
        IF e.info.readOnly THEN readOnlyWindow ELSE readWriteWindow];
      END;
    END;
  ENDCASE => ERROR Bug[bug3];
END;

PinResidentDescriptorPageGroups: StoragePrograms.StartListProc --[index]-- =
  BEGIN
  WITH e: tableBase[index] SELECT FROM
    space => NULL;
    swapUnit =>
      IF e.info.state = residentDescriptor THEN
        KernelFile.Pin[
          file: [request.location.diskFileID.fID, File.read],
          page: tableBase[e.parent].backingPage + e.base, count: e.pages];
    ENDCASE => ERROR;
  END;

-- Creates and maps non-initiallyResident spaces.

CreateInitiallyOutSpaces: StoragePrograms.StartListProc --[index]-- =
  -- Note: DescribeInitiallyResidentSpaces has initialized the handle of each space StartList.entr
**y to Space.nullHandle.
  BEGIN
  WITH e: tableBase[index] SELECT FROM
    space => NULL; -- defer creating ALL spaces so the space can be
    -- correctly mapped to either the boot file or a temporary file
```



```
    swapUnit => -- We ASSUME that all swapUnit's of a space have the same
      -- resident and readOnly attributes!!!
      BEGIN OPEN e;
      IF tableBase[parent].bootLoaded THEN RETURN;
      IF LOOPHOLE[tableBase[parent].handle, Space.Handle] = Space.nullHandle
        THEN CreateParent[parent, e.info.readOnly OR e.info.state = resident];
      -- (sets handle)
      IF tableBase[parent].type.class = family THEN
        [] ← Space.Create[e.pages, LOOPHOLE[tableBase[parent].handle], base];
      END;
    ENDCASE => ERROR Bug[bug4];
  END;

CreateParent: PROCEDURE [parent: StartList.SpaceIndex, mapToBootFile: BOOLEAN] =
  BEGIN OPEN p: tableBase[parent];
  parentsParent: Space.Handle;
  pageParentsParent: Space.PageNumber;
  window: Space.WindowOrigin;
  IF p.type.class = empty THEN ERROR Bug[bug5];
  [parentsParent, pageParentsParent] ← StoragePrograms.SuperFromPage[p.vmpage];
  p.handle ← LOOPHOLE[Space.Create[
    p.pages, parentsParent, p.vmpage - pageParentsParent]];
  window ←
    IF mapToBootFile THEN WindowForSpace[parent] -- map to boot file
     ELSE Space.defaultWindow; -- map to temporary file
  Space.Map[LOOPHOLE[p.handle], window];
  IF ~mapToBootFile THEN -- must copy contents into the space in this case
    BEGIN
    --TEMPORARY BUG WORKAROUND. We first touch the space before doing the
    -- CopyIn as CopyIn doesn't read the file if the space is out and dead as ours is.
    ptr: LONG POINTER TO WORD = Space.LongPointer[LOOPHOLE[p.handle]];
    ptr↑ ← 0; -- THE WORKAROUND - touch the space
    Space.CopyIn[LOOPHOLE[p.handle], WindowForSpace[parent]];
    END;
  WITH t: p.type SELECT FROM
    family =>
      IF t.anyResidentDescriptorChildren THEN
        ERROR SystemInternal.Unimplemented;
    ENDCASE;
  END;

WindowForSpace: PROCEDURE [space: StartList.SpaceIndex]
  RETURNS [Space.WindowOrigin] =
  BEGIN
  RETURN[
    Space.WindowOrigin[
      file: File.Capability[
      fID:
      SELECT tableBase[space].backingStore FROM
        null => File.nullID,
        self => request.location.diskFileID.fID,
        ENDCASE => ERROR Bug[bug7],
      permissions:
      IF tableBase[space].readOnly THEN File.read ELSE File.read + File.write],
      base:
      tableBase[space].backingPage + request.location.diskFileID.firstPage]]
  END;
```

```
  EnumerateStartList: PUBLIC PROCEDURE [Proc: StoragePrograms.StartListProc] =
    BEGIN OPEN StartList;
    index: Index;
    size: CARDINAL;
    FOR index ← StartIndex, index + size UNTIL tableBase[index].option = stop DO
      Proc[index];
      size ←
        WITH tableBase[index] SELECT FROM
          space => SIZE[space Entry],
          swapUnit => SIZE[swapUnit Entry],
          ENDCASE => ERROR Bug[bug8];
      ENDLOOP;
    END;

  InitializeDebuggerSwap: PROCEDURE =
    BEGIN OPEN parameters;
    locDebuggee ← LOOPHOLE[h.locDebuggee];
    locDebugger ← LOOPHOLE[h.locDebugger];
    -- Now can get to debugger if info was in boot file.
    IF locDebuggee.diskFileID.da ~= LOOPHOLE[LONG[0]] THEN canSwap ← TRUE;
    pMicrocodeCopy ← pGermCopy ← NIL; -- until microcode swapping installed

    END;

  -- When all else fails . . .

  Punt: PROCEDURE [c: PilotMP.Code] = INLINE {ProcessorFace.SetMP[c]; DO ENDLOOP};

  -- Main Body code:

  ProcessOperations.WriteWDC[1];
  -- interrupts off (necessary for now on Dandelion)
  XferTrap.WriteXTS[off];
  ProcessStartList[]; -- never returns!

  END.
```

(For earlier log entries see Pilot 4.0 archive version.)

April 16, 1980 9:18 AM Knutsen Unbundle ResidentMemory into separate module. Expl
***icitly start MStore, SimpleSpace, ResidentMemory, ResidentHeap, and CachedRegionImplA, ea*
***rly. Exports countVM, etc. Unbundle Store initialization functions. Changes for moving to nonz*
***ero MDS. Changes for eliminating DiagnosticPilotControl.*
April 30, 1980 5:35 PM Forrest Change to turn interrupts off at startup, on after startin
***g heads*
May 3, 1980 1:00 PM Forrest ControlDefs = >PrincOps, FrameOps = >Frame, ProcessOps = >Pro
***cessOperations*
May 14, 1980 6:22 PM McJones Start processor face implementation; move starting of i
***nterrupts back to Processes*
May 17, 1980 9:04 PM Forrest ?
July 9, 1980 7:15 PM Knutsen/Luniewski New StartPilot/StartList
July 14, 1980 4:36 PM Gobbel Add transaction initialization
July 19, 1980 1:57 PM McJones OISProcessorFace = >ProcessorFace; initialize Heap
August 15, 1980 11:49 AM McJones Allow map logging when teledebugging
August 28, 1980 4:39 PM McJones START InterruptKey = >InitializeInterrupt[]; add PinResi
***dentDescriptorPageGroups and add firstPage in WindowForSpace*
August 29, 1980 9:02 PM Forrest Eliminate references to PPData
September 2, 1980 2:43 PM Forrest Temporarily remove Trap
September 8, 1980 9:32 PM Johnsson Add XferTrap

September 15, 1980 11:49 AM Luniewski Make CreateParent map writable, swappable spaces to
***a temporary file and not the boot file.*
October 10, 1980 4:40 PM Forrest Unmap Page 0 pageMDS (if # 0).

```
-- SystemImpl.mesa (last edited by: Forrest on: October 3, 1980 8:21 PM)
-- THINGS TO DO:
-- 1) Get rid of call to TemporarySetGMT.SetGMT after through with gmt clock simulation

DIRECTORY
  ControlPrograms USING [],
  DeviceCleanup USING [Await, Item, Reason],
  Environment USING [Long],
  Inline USING [LongDiv, LongDivMod, LongMult],
  Process USING [MsecToTicks, SetTimeout],
  ProcessorFace USING [
    GetGreenwichMeanTime, gmtEpoch, microsecondsPerHundredPulses, PowerOff,
    processorID, ResetAutomaticPowerOn, SetAutomaticPowerOn],
  RuntimeInternal USING [WorryCallDebugger],
  SpecialSystem USING [ProcessorID],
  System USING [
    GetClockPulses, GreenwichMeanTime, Microseconds, Pulses, TimerHandle,
    UniversalID],
  SystemInternal USING [UniversalID],
  TemporarySetGMT USING [SetGMT],
  Volume USING [Close, GetNext, ID, nullID];

SystemImpl: MONITOR
  IMPORTS
    DeviceCleanup, Inline, Process, ProcessorFace, RuntimeInternal, System,
    TemporarySetGMT, Volume
  EXPORTS SpecialSystem, System, SystemInternal, ControlPrograms =
  BEGIN OPEN System; -- Interval timers
  GetIntervalTime: PUBLIC PROC [t: TimerHandle] RETURNS [Microseconds] = {
    RETURN[PulsesToMicroseconds[[GetClockPulses[] - LOOPHOLE[t, Pulses]]]]};

  PulsesToMicroseconds: PUBLIC PROC [p: Pulses] RETURNS [Microseconds] =
    BEGIN -- (p*msPerHp)/(100units/hundred)
    RETURN[MultThenDiv[p, ProcessorFace.microsecondsPerHundredPulses, 100]]
    END;

  MicrosecondsToPulses: PUBLIC PROC [m: Microseconds] RETURNS [Pulses] =
    BEGIN -- (microseconds*100units/hundred)/microsecondsPerHundredPulses
    RETURN[[MultThenDiv[m, 100, ProcessorFace.microsecondsPerHundredPulses]]]
    END;

  MultThenDiv: PROC [m1: LONG CARDINAL, m2: CARDINAL, dv: CARDINAL]
    RETURNS [result: LONG CARDINAL] =
    BEGIN OPEN Inline, mm1: LOOPHOLE[m1, num Environment.Long];
    t: MACHINE DEPENDENT RECORD [
      SELECT OVERLAID * FROM
        separate => [low, mid, high: CARDINAL],
        lower => [lowlong: LONG CARDINAL, junk: CARDINAL],
        higher => [junk: CARDINAL, highlong: LONG CARDINAL],
        ENDCASE];
    t.lowlong ← LongMult[mm1.lowbits, m2];
    IF mm1.highbits # 0 THEN
      BEGIN
      t.highlong ← LongMult[mm1.highbits, m2] + t.mid;
      IF t.high # 0 THEN
        BEGIN OPEN q: LOOPHOLE[result, num Environment.Long];
        -- have to do triple divide
        IF t.high >= dv THEN t.high ← t.high MOD dv;
```



```
        -- overflow; lowbits will be right
        [quotient: q.highbits, remainder: t.mid] ← LongDivMod[t.highlong, dv];
        q.lowbits ← LongDiv[t.lowlong, dv];
        RETURN;
        END;
      END;
    -- t.high is 0, so let mesa do the work...
    RETURN[t.lowlong/LONG[dv]];
    END;

  -- GMT

  -- This is an entry procedure to serialize access to ProcessorFace.GetGreenwichMeanTime

  GetGreenwichMeanTime: PUBLIC ENTRY PROC RETURNS [GreenwichMeanTime] = {
    RETURN[LOOPHOLE[ProcessorFace.GetGreenwichMeanTime[]]]};

  -- Processor identification (SpecialSystem)

  GetProcessorID: PUBLIC PROC RETURNS [SpecialSystem.ProcessorID] = {
    RETURN[LOOPHOLE[ProcessorFace.processorID]]};

  -- Universal identifiers

  UniversalID: PUBLIC TYPE = SystemInternal.UniversalID;

  -- Generate new universalID from the processorID and universal id counter.
  -- Sequence field of resultant value always less than
  -- SecondsSinceEpoch[GetGreenwichMeanTime].

  GetUniversalID: PUBLIC ENTRY PROC RETURNS [UniversalID] =
    BEGIN
    secondsSinceEpoch: LONG CARDINAL;
    nextUID: SystemInternal.UniversalID;
    -- If clock isn't set, GetGreenwichMeanTime returns gmtEpoch, so uidCounter = 0
    DO
      secondsSinceEpoch ←
        ProcessorFace.GetGreenwichMeanTime[] - ProcessorFace.gmtEpoch;
      IF secondsSinceEpoch = 0 THEN
        RuntimeInternal.WorryCallDebugger["GMT clock not set: GetUniversalID"L];
      IF ~uidCounterValid THEN
        BEGIN
        uidCounter ← secondsSinceEpoch; -- clock set after initialization
        uidCounterValid ← TRUE;
        END;
      IF uidCounter < secondsSinceEpoch THEN EXIT;
      WAIT oneSecond;
      ENDLOOP;
    nextUID ←
      [processor: LOOPHOLE[ProcessorFace.processorID], sequence: uidCounter];
    uidCounter ← uidCounter + 1;
    RETURN[nextUID]
    END;

  oneSecond: CONDITION;
  uidCounter: LONG CARDINAL; -- always <= SecondsSinceEpoch[GetGreenwichMeanTime[]]
  uidCounterValid: BOOLEAN ← FALSE;
```

```
InitializeUIDCleanup: PROC =
  BEGIN OPEN DeviceCleanup;
  item: Item;
  reason: Reason;
  DO
    reason ← Await[@item];
    SELECT reason FROM turnOff => uidCounterValid ← FALSE; ENDCASE;
    ENDLOOP;
  END;

-- System Power Control

PowerOff: PUBLIC PROC =
  BEGIN
  vID: Volume.ID ← Volume.nullID;
  UNTIL (vID ← Volume.GetNext[vID]) = Volume.nullID DO
    Volume.Close[vID] ENDLOOP;
  ProcessorFace.PowerOff[];
  END;

SetAutomaticPowerOn: PUBLIC PROC [
  time: GreenwichMeanTime, externalEvent: BOOLEAN] = {
  ProcessorFace.SetAutomaticPowerOn[time, externalEvent]; };

ResetAutomaticPowerOn: PUBLIC PROC = {ProcessorFace.ResetAutomaticPowerOn[]; };

-- SystemInternal

Unimplemented: PUBLIC SIGNAL = CODE;

-- Initialization

TemporarySetGMT.SetGMT[]; -- move this call to PilotControl?
Process.SetTimeout[@oneSecond, Process.MsecToTicks[1024]];
-- set timeout to (approx) one second
InitializeUIDCleanup[];
END.
```

LOG
(For earlier log entries see Teak archive version.)
Time: February 4, 1980 3:56 PM By: McJones Action: Move gmt-keeping below
**OISProcessorFace
Time: February 26, 1980 12:22 PM By: McJones AR1840: Reset uidCounter on ret
**urn from debugger
Time: April 15, 1980 3:11 PM By: McJones Action: Move InitializeGMT to another modul
**e
Time: June 24, 1980 4:39 PM By: McJones Action: Convert to 48-bit processor ids and 8
**0-bit universal ids; OISProcessorFace = >ProcessorFace
Time: October 3, 1980 7:56 PM By: Forrest Action: Convert to use 48 bit arithmetic in us
**ec< = >pulses

```
-- FileImpl.mesa (last edited by: McJones on: September 26, 1980 4:29 PM)

-- Note on monitors in this module: All procedures in this module that must read or change entries
**in the FileMgr's caches do their actual work through LogicalVolume.VolumeAccess, and are thu
**s protected by VolumeImpl's monitor lock from inconsistencies that might otherwise result if ca
**che accesses were interleaved. Simply (I hope) stated: VolumeImpl's monitor is used to insure t
**he consistency of individual cache entries, FileCacheImpl's monitor insures consistency of the
**global state of the caches, and FileImpl's monitor is used to serialize access to files at a higher l
**evel (i.e., above the level of the caches). There is at least one procedure, GetFileDescriptor, whi
**ch must be outside of FileImpl's monitor (because it is used by the FileHelper), but which must a
**ccess the FileMgr caches.

DIRECTORY
  --AltoFileDefs USING [FA, LD],
  Boot USING [Location, LVBootFiles],
  Device USING [Type],
  DiskChannel USING [
    Address, Handle, Drive, GetAttributes, GetDriveAttributes, GetDriveTag],
  Environment USING [PageCount, PageNumber, wordsPerPage],
  File USING [
    Capability, delete, ErrorType, grow, ID, lastPageNumber, nullCapability,
    nullID, PageCount, PageNumber, Permissions, read, shrink, Type, write],
  FileCache USING [
    FlushFile, GetFilePtrs, GetPageGroup, SetFile, SetPageGroup, ReturnFilePtrs],
  FileInternal USING [
    Descriptor, FilePtr, LocalFilePtr, maxPermissions, PageGroup],
  FilePageTransfer USING [Initiate, Request],
  FilerException USING [Await],
  FMPrograms USING [
    MarkerPageImpl, PhysicalVolumeImpl, VolAllocMapImpl, VolFileMapImpl],
  Inline USING [BITAND, LowHalf],
  KernelFile USING [defaultPageCount, GetNextFile, GetRootFile],
  LabelTransfer USING [ReadLabel, VerifyLabels, WriteLabelAndData],
  LogicalVolume USING [
    CloseLogicalVolume, FileVolumeStatus, FreeVolumePages, Handle, nullVolumePage,
    OpenLogicalVolume, PageNumber, PutRootFile, VolumeAccess, VolumeAccessProc,
    VolumeAccessStatus],
  MStore USING [Promise],
  PilotDisk USING [
    Address, GetLabelFilePage, Label, LabelCheckSum, SetLabelFilePage],
  PilotFileTypes USING [PilotRootFileType, PilotVFileType, tTempFileList],
  PilotSwitches USING [switches --f--],
  Process USING [Detach],
  Runtime USING [CallDebugger],
  SimpleSpace USING [Create, ForceOut, Handle, Map, Page, Unmap],
  Space USING [defaultWindow],
  SpecialFile USING [Link],
  SpecialVolume USING [GetLogicalVolumeBootFiles, SetLogicalVolumeBootFiles],
  StoragePrograms,
  SubVolume USING [Find, GetPageAddress, Handle],
  System USING [GetUniversalID, VolumeID],
  SystemInternal USING [ --altoFPSeries,--Unimplemented],
  Transaction USING [Handle, nullHandle],
  TransactionState USING [FileOps, Log, LogEntry],
  Utilities USING [LongPointerFromPage],
  VMMapLog USING [Entry],
  VolAllocMap USING [AllocPageGroup, FreePageGroup],
  VolFileMap USING [DeletePageGroup, GetPageGroup, InsertPageGroup],
```



```
  Volume USING [
    GetNext, ID, InsufficientSpace, NeedsScavenging, NotOpen, nullID, systemID,
    TypeSet, Unknown];

--VolumeInternal USING [altoVolume, PageNumber];

FileImpl: MONITOR
  IMPORTS
    DiskChannel, FileCache, FilePageTransfer, FilerException, FMPrograms, Inline,
    KernelFile, LabelTransfer, LogicalVolume, MStore, PilotDisk, PilotSwitches,
    Process, Runtime, SimpleSpace, SpecialVolume, SubVolume, System,
    SystemInternal, Transaction, TransactionState, Utilities, VolAllocMap,
    VolFileMap, Volume
  EXPORTS File, LogicalVolume, KernelFile, SpecialFile, StoragePrograms
  SHARES DiskChannel, File, System, SystemInternal =
  BEGIN OPEN File, FileInternal;
  VolProc: TYPE = LogicalVolume.VolumeAccessProc;

  -- One-page buffer used by MakePermanent, MakeImmutable, MakeBootable:
  pageBuffer: SimpleSpace.Handle = SimpleSpace.Create[1, hyperspace];
  bufferPage: Environment.PageNumber = SimpleSpace.Page[pageBuffer];
  bufferPointer: LONG POINTER = Utilities.LongPointerFromPage[bufferPage];

  -- Temporary File list used by DeleteTemps and EnterInTempFileList
  --         (these values assume list entries do not overlap pages)
  tmpsBuffer: SimpleSpace.Handle ← SimpleSpace.Create[1, hyperspace];
  tmpsVolume: Volume.ID ← Volume.nullID;
  tmpsFile: File.Capability;
  tmpsFileSize: File.PageCount;
  tmpStart: LONG POINTER TO File.ID = Utilities.LongPointerFromPage[
    SimpleSpace.Page[tmpsBuffer]];
  tmpsLast: LONG POINTER TO File.ID =
    tmpStart + (Environment.wordsPerPage/SIZE[File.ID])*SIZE[File.ID];
  tmps: LONG POINTER TO File.ID;

  nullTransactionHandle: Transaction.Handle = Transaction.nullHandle;

  AttributeAction: TYPE = {immutable, permanent, mutable, temporary};

  Error: PUBLIC ERROR [type: ErrorType] = CODE;
  InvalidParameters: PUBLIC ERROR = CODE;
  LabelError: SIGNAL = CODE;
  Unknown: PUBLIC ERROR [file: Capability] = CODE;
  ExistingFile: PUBLIC ERROR = CODE;
  FileImplError: ERROR [
    {
    disappearedOrRemoteTempFile, fileWithHole, illegalAttributeAction,
    impossibleFileType, makeBootablePageGroupNotFound,
    makeBootableImpossibleError, missingPage0, missingPageGroupSetSize,
    missingPinnedPageGroup, remoteTmpsFile, remoteTxLog,
    SubvolumeNotFoundInGetFilePoint, tmpsFileProblem, tmpsFileWentAway,
    tmpsVolumeWentAway, volumeWentAwayDuringDeleteTemps,
    setSizeImpossibleError}] = CODE;

  -- MONITOR EXTERNAL PROCEDURES

  Create: PUBLIC PROCEDURE [
    volume: System.VolumeID, initialSize: PageCount, type: Type,
```

```
  transaction: Transaction.Handle] RETURNS [file: Capability] =
  BEGIN
  IF type IN PilotFileTypes.PilotVFileType THEN ERROR Error[reservedType];
  file ← [[System.GetUniversalID[]], FileInternal.maxPermissions];
  CreateWithIDExternal[volume, initialSize, type, file.fID, transaction];
  END;

CreateWithID: PUBLIC PROCEDURE [
  volume: System.VolumeID, initialSize: PageCount, type: Type, id: ID] =
  -- Create file given the file identifier
  {CreateWithIDExternal[volume, initialSize, type, id]};

Delete: PUBLIC PROCEDURE [file: Capability, transaction: Transaction.Handle] = {
  DeleteCommon[@file, NIL, transaction]};

DeleteImmutable: PUBLIC PROCEDURE [
  file: Capability, volume: System.VolumeID, transaction: Transaction.Handle] =
  {DeleteCommon[@file, @volume, transaction]};

FileHelperProcess: PROCEDURE =
  BEGIN
  countInitiated: Environment.PageCount;
  helpCount: CARDINAL ← 0;
  fileD: FileInternal.Descriptor;
  mappedOffMessage: STRING = "Mapped off File (Helper)"L;
  -- save the frame space in Helper1
  req: FilePageTransfer.Request;
  DO
    req ← FilerException.Await[];
    GetFileDescriptorSignals[@req.file, @fileD];
    -- file cache updated as side effect
    WITH fileD SELECT FROM
      local =>
        BEGIN
        Helper1: VolProc =
          BEGIN
          fPtr: FileInternal.FilePtr;
          success: BOOLEAN;
          group: FileInternal.PageGroup;
          updateMarkers ← FALSE;
          [success, fPtr] ← FileCache.GetFilePtrs[1, fileD.fileID];
          IF ~success THEN RETURN;
          -- fell out of cache, let excepton happen again
          [success, group] ← VolFileMap.GetPageGroup[
            volume, @fileD, req.filePage];
          IF (success ← success AND (req.filePage < group.nextFilePage)) THEN
            FileCache.SetPageGroup[req.file.fID, group, FALSE];
          -- page is within file?
          FileCache.ReturnFilePtrs[1, fPtr];
          IF ~success THEN Runtime.CallDebugger[mappedOffMessage]
          END;

        SELECT LogicalVolume.VolumeAccess[@fileD.volumeID, Helper1] FROM
          ok => NULL;
          ENDCASE =>
            Runtime.CallDebugger[
              "Volume vanished between Descriptor and PageGroup (Helper)"L];
        END;
      remote => NULL;
```



```
    ENDCASE;
  countInitiated ← FilePageTransfer.Initiate[req];
  IF req.promise THEN MStore.Promise[countInitiated];
  helpCount ← helpCount + 1;
  ENDLOOP
END;

GetAttributes: PUBLIC PROCEDURE [
  file: Capability, transaction: Transaction.Handle]
  RETURNS [type: Type, immutable, temporary: BOOLEAN, volume: System.VolumeID] =
  BEGIN
  f: FileInternal.Descriptor;
  GetFileDescriptorSignals[@file, @f];
  WITH f SELECT FROM
    local => RETURN[type, immutable, temporary, f.volumeID];
    ENDCASE => --RETURN[File.Type[0], FALSE, FALSE, VolumeInternal.altoVolume]
      ERROR SystemInternal.Unimplemented
  END;

GetBootLocation: PUBLIC PROCEDURE [
  file: File.Capability, filePage: File.PageNumber]
  RETURNS [
    deviceType: Device.Type, deviceOrdinal: CARDINAL, link: SpecialFile.Link] =
  BEGIN
  v: Volume.ID;
  grp: FileInternal.PageGroup;
  channel: DiskChannel.Handle;
  GetVIDAndGroup[@v, @grp, @file, filePage]; -- set vID, group
  [channel, LOOPHOLE[link, DiskChannel.Address]] ← SubVolume.GetPageAddress[
    v, grp.volumePage + (filePage - grp.filePage)];
  [deviceType: deviceType, deviceOrdinal: deviceOrdinal] ←
    DiskChannel.GetDriveAttributes[DiskChannel.GetAttributes[channel].drive];
  END;

GetFileDescriptor: PROCEDURE [
  file: POINTER TO READONLY File.Capability, fileD: FileInternal.FilePtr,
  vP: POINTER TO READONLY Volume.ID ← NIL] RETURNS [found: BOOLEAN] =
  -- Look up file in cache and then in given volume or all volumes (vP = NIL),
  -- and set the descriptor. Return success
  BEGIN
  fPtr: FileInternal.FilePtr;
  success: BOOLEAN;
  v: Volume.ID;
  getAll: Volume.TypeSet =
    [normal: TRUE, debugger: TRUE, debuggerDebugger: TRUE];
  GetFileDescriptor1: VolProc =
    BEGIN
    group: FileInternal.PageGroup;
    label: PilotDisk.Label;
    lf: local FileInternal.Descriptor ← [file.fID, v, local[, , , ]];
    updateMarkers ← FALSE;
    [success, group] ← VolFileMap.GetPageGroup[volume, @lf, 0];
    IF ~success THEN RETURN;
    label ← LabelTransfer.ReadLabel[lf, 0, group.volumePage];
    IF label.fileID # lf.fileID OR PilotDisk.GetLabelFilePage[@label] # 0 THEN
      SIGNAL LabelError;
    lf.body ← local[
      immutable: label.immutable, temporary: label.temporary,
      size: VolFileMap.GetPageGroup[
```

```
      volume, LONG[@lf], File.lastPageNumber].group.filePage, type: label.type];
    -- success??
    fileD↑ ← lf;
    DO
      FileCache.SetFile[lf, FALSE];
      [success, fPtr] ← FileCache.GetFilePtrs[1, lf.fileID];
      IF success THEN
        BEGIN
        FileCache.SetPageGroup[lf.fileID, group, FALSE];
        FileCache.ReturnFilePtrs[1, fPtr];
        RETURN;
        END;
      ENDLOOP;
    END;

    IF file.fID = File.nullID THEN RETURN[FALSE];
  --IF LOOPHOLE[file.fID, SystemInternal.UniversalID].series = SystemInternal.altoFPSeries THE
**N
  --IF GetAltoSize[file] = 0 THEN ERROR Unknown[file↑] ELSE RETURN[[file.fID, VolumeInternal.
**altoVolume, remote[]]];
    [success, fPtr] ← FileCache.GetFilePtrs[1, file.fID];
    IF success THEN
      BEGIN
      fileD↑ ← fPtr↑;
      FileCache.ReturnFilePtrs[1, fPtr];
      IF vP = NIL OR fileD.volumeID = vP↑ THEN RETURN[TRUE];
      END;
    v ← IF vP = NIL THEN Volume.GetNext[Volume.nullID, getAll] ELSE vP↑;
    WHILE v # Volume.nullID DO
      IF LogicalVolume.VolumeAccess[@v, GetFileDescriptor1] = ok AND success THEN
        RETURN[TRUE];
      IF vP # NIL THEN EXIT ELSE v ← Volume.GetNext[v, getAll];
      ENDLOOP;
    RETURN[FALSE];
    END;

  GetFileDescriptorSignals: PROCEDURE [
    file: POINTER TO READONLY File.Capability, fileD: FileInternal.FilePtr,
    vP: POINTER TO READONLY Volume.ID ← NIL] =
    BEGIN
    IF ~GetFileDescriptor[file, fileD, vP].found THEN ERROR Unknown[file↑];
    END;

  GetFilePoint: PUBLIC PROCEDURE [
    pEntry: LONG POINTER TO VMMapLog.Entry, pFile: POINTER TO File.Capability,
    filePage: File.PageNumber] =
    BEGIN
    countMax: CARDINAL = 4096;
    vID: Volume.ID;
    group: FileInternal.PageGroup;
    lvPage: LogicalVolume.PageNumber;
    success: BOOLEAN;
    svH: SubVolume.Handle;
    label: PilotDisk.Label;
    GetVIDAndGroup[@vID, @group, pFile, filePage];
    lvPage ← filePage - group.filePage + group.volumePage;
    -- file => logical vol page
    [success, svH] ← SubVolume.Find[vID, lvPage];
    IF ~success THEN ERROR FileImplError[SubvolumeNotFoundInGetFilePoint];
```



```
    label.fileID ← pFile.fID;
    PilotDisk.SetLabelFilePage[@label, filePage];
    pEntry↑ ←
      [page:, writeProtected:, fill:,
        count: MIN[countMax, Inline.LowHalf[group.nextFilePage - filePage]],
        filePoint: disk[
        driveTag: DiskChannel.GetDriveTag[
        DiskChannel.GetAttributes[svH.channel].drive],
        diskPage: lvPage - svH.lvPage + svH.pvPage,
        -- logical vol => physical vol page,
        labelCheck: PilotDisk.LabelCheckSum[@label, 0]]];
    END;

  GetSize: PUBLIC PROCEDURE [file: Capability, transaction: Transaction.Handle]
    RETURNS [size: PageCount] =
    BEGIN
    fileD: FileInternal.Descriptor;
    GetFileDescriptorSignals[@file, @fileD];
    RETURN[
      WITH fileD SELECT FROM
        local => size,
        ENDCASE => --GetAltoSize[@file]--ERROR SystemInternal.Unimplemented]
    END;

  IsOnVolume: PUBLIC PROCEDURE [file: Capability, volume: System.VolumeID] =
    BEGIN
    fileD: FileInternal.Descriptor;
    IF ~GetFileDescriptor[@file, @fileD] THEN
      FileCache.SetFile[[file.fID, volume, remote[]], FALSE];
    END;

  MakeBootable: PUBLIC PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber, count: File.PageCount,
    lastLink: SpecialFile.Link] RETURNS [firstLink: SpecialFile.Link] =
    BEGIN
    RETURN[
      LOOPHOLE[MakeBootableOrUnbootable[
        bootable, @file, firstPage, count, LOOPHOLE[lastLink]]]];
    END;

  MakeImmutable: PUBLIC PROCEDURE [
    file: Capability, transaction: Transaction.Handle] = {
    ChangeAttributes[@file, immutable, transaction]};

  MakeMutable: PUBLIC PROCEDURE [file: Capability] = {
    ChangeAttributes[@file, mutable]};

  MakePermanent: PUBLIC PROCEDURE [
    file: Capability, transaction: Transaction.Handle] = {
    ChangeAttributes[@file, permanent, transaction]};

  MakeTemporary: PUBLIC PROCEDURE [file: Capability] = {
    ChangeAttributes[@file, temporary]};

  MakeUnbootable: PUBLIC PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber, count: File.PageCount] = {
    [] ← MakeBootableOrUnbootable[unbootable, @file, firstPage, count, ]};
```

```
Move, ReplicateImmutable: PUBLIC PROCEDURE [
  file: Capability, volume: System.VolumeID, transaction: Transaction.Handle] =
  {SIGNAL SystemInternal.Unimplemented};

SetDebuggerFiles: PUBLIC PROCEDURE [debugger, debuggee: File.Capability] =
  BEGIN
  bootFiles: Boot.LVBootFiles;
  fileD: FileInternal.Descriptor;
  dTEr, dTEe: Device.Type;
  dOEr, dOEe: CARDINAL;
  linkEr, linkEe: SpecialFile.Link;
  id: Volume.ID = Volume.systemID;
  [deviceType: dTEr, deviceOrdinal: dOEr, link: linkEr] ← GetBootLocation[
    debugger, 0];
  [deviceType: dTEe, deviceOrdinal: dOEe, link: linkEe] ← GetBootLocation[
    debuggee, 0];
  GetFileDescriptorSignals[@debugger, @fileD];
  IF dTEr # dTEe OR dOEr # dOEe OR fileD.volumeID # id THEN
    ERROR InvalidParameters;
  SpecialVolume.GetLogicalVolumeBootFiles[id, @bootFiles];
  bootFiles[debugger] ← [debugger.fID, 0, LOOPHOLE[linkEr]];
  bootFiles[debuggee] ← [debuggee.fID, 0, LOOPHOLE[linkEe]];
  SpecialVolume.SetLogicalVolumeBootFiles[id, @bootFiles];
  END;

SufficientPermissions: PROCEDURE [given, needed: Permissions]
  RETURNS [BOOLEAN] = INLINE {RETURN[Inline.BITAND[given, needed] = needed]};

-- ENTRY PROCEDURES

ChangeAttributes: ENTRY PROCEDURE [
  file: POINTER TO READONLY File.Capability, action: AttributeAction,
  transaction: Transaction.Handle ← nullTransactionHandle] =
  BEGIN
  ENABLE UNWIND => NULL;
  fileD: FileInternal.Descriptor;
  IF ~GetFileDescriptor[file, @fileD] THEN RETURN WITH ERROR Unknown[file↑];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      IF f.immutable AND action # mutable THEN
        RETURN WITH ERROR Error[immutable];
      IF action = permanent AND ~f.temporary THEN RETURN;
      IF transaction # nullTransactionHandle THEN
        BEGIN OPEN TransactionState;
        entry: LogEntry ←
          SELECT action FROM
            immutable => [makeMutable[file: file↑]],
            permanent => [makeTemporary[file: file↑]],
            ENDCASE => ERROR FileImplError[illegalAttributeAction];
        Log[transaction, @entry, txFileProcs];
        END;
      SELECT ChangeAttributesInternal[@f, action] FROM
        ok => NULL;
        volumeUnknown => RETURN WITH ERROR Volume.Unknown[f.volumeID];
        volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[f.volumeID];
        ENDCASE;
      END
    ENDCASE => RETURN WITH ERROR SystemInternal.Unimplemented;
  END;

CloseVolumeAndFlushFiles: PUBLIC ENTRY PROCEDURE [
  v: POINTER TO READONLY Volume.ID] =
  BEGIN -- Someday, flush file caches & make sure nobody is mapped to the bugger
  IF tmpsVolume = v↑ THEN TmpsUnmap[];
  LogicalVolume.CloseLogicalVolume[v ! Volume.Unknown => GOTO unknown];
  EXITS unknown => RETURN WITH ERROR Volume.Unknown[v↑];
  END;

CreateWithIDExternal: PROCEDURE [
  volume: System.VolumeID, initialSize: PageCount, type: Type, id: ID,
  transaction: Transaction.Handle ← nullTransactionHandle] =
  BEGIN
  file: Capability ← [id, FileInternal.maxPermissions];
  fileD: FileInternal.Descriptor;
  CreateWithIDEntry: ENTRY PROCEDURE = --INLINE--
    BEGIN
    ENABLE UNWIND => NULL; -- Should check for existence of file with this ID
    IF type IN PilotFileTypes.PilotVFileType THEN
      ERROR FileImplError[impossibleFileType];
    IF GetFileDescriptor[@file, @fileD, @volume] THEN
      RETURN WITH ERROR ExistingFile;
    IF transaction # nullTransactionHandle THEN
      BEGIN OPEN TransactionState;
      entry: LogEntry ← [delete[file: file]];
      Log[transaction, @entry, txFileProcs];
      END;
    SELECT CreateWithIDInternal[@volume, @initialSize, type, @id] FROM
      ok => NULL;
      insufficientSpace => RETURN WITH ERROR Volume.InsufficientSpace;
      ENDCASE;
    END;
  TmpsEnter[@file, @volume];
  CreateWithIDEntry[];
  END;

DeleteCommon: ENTRY PROCEDURE [
  file: POINTER TO READONLY File.Capability,
  volume: POINTER TO READONLY Volume.ID,
  -- NIL if Delete, volume if DeleteImmutable--
  transaction: Transaction.Handle] =
  BEGIN
  ENABLE UNWIND => NULL; -- we only expect to get this from tx logging
  fileD: FileInternal.Descriptor;
  IF ~GetFileDescriptor[file, @fileD, volume] THEN
    RETURN WITH ERROR Unknown[file↑];
  IF ~SufficientPermissions[file.permissions, File.delete] THEN
    RETURN WITH ERROR Error[insufficientPermissions];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      IF volume = NIL THEN {
        IF f.immutable THEN RETURN WITH ERROR Error[immutable]}
      ELSE IF ~f.immutable THEN RETURN WITH ERROR Error[notImmutable];
      IF transaction # nullTransactionHandle THEN
        BEGIN OPEN TransactionState;
```

```
        entry: LogEntry ←
          [setContents[
            window: [[f.fileID, file.permissions], 0], size: f.size,
            makePermanent: ~f.temporary, makeImmutable: f.immutable]];
        Log[transaction, @entry, txFileProcs];
        entry ←
          [create[
            id: f.fileID, volume: f.volumeID, size: f.size, type: f.type]];
        Log[transaction, @entry, txFileProcs];
        END;
      SELECT DeleteFileOnVolumeInternal[@f] FROM
        ok => NULL;
        volumeUnknown => RETURN WITH ERROR Volume.Unknown[f.volumeID];
        volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[f.volumeID];
        ENDCASE;
      IF f.temporary THEN TmpsRemove[file, @f.volumeID];
      END;
    ENDCASE => RETURN WITH ERROR SystemInternal.Unimplemented;
  END;

  GetFileAttributes: PUBLIC ENTRY PROCEDURE [file: Capability]
    RETURNS [size: PageCount, immutable: BOOLEAN, readOnly: BOOLEAN] =
    BEGIN
    ENABLE UNWIND => NULL;
    f: FileInternal.Descriptor;
    GetFileDescriptorSignals[@file, @f];
    WITH f SELECT FROM
      local =>
        RETURN[size, immutable, ~SufficientPermissions[file.permissions, write]];
      ENDCASE => ERROR SystemInternal.Unimplemented
    END;

  --DeleteTemps: PUBLIC PROCEDURE [volume: Volume.ID] =
  --BEGIN

  --DeleteTempsEntry: ENTRY PROCEDURE =
  --BEGIN
  --SELECT DeleteTempsInternal[
  --@volume] FROM
  --ok => NULL;
  --volumeUnknown => RETURN WITH ERROR Volume.Unknown[volume];
  --volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[volume];
  --ENDCASE;
  --END;

  --DeleteTempsEntry[];
  --END;

  -- Maintained by AltoSize
  -- cacheA, cacheB: RECORD[file: File.ID, size: PageCount] ← [File.nullID, 0];
  -- Access file attributes in label given fileID keeping 2 element MRU cache
  --GetAltoSize: ENTRY PROCEDURE [file: POINTER TO READONLY File.Capability] RETURNS[siz
**e: PageCount] =
  --BEGIN
  --faP: LONG POINTER TO AltoFileDefs.FA = @LOOPHOLE[bufferPointer, LONG POINTER TO A
**ltoFileDefs.LD].eofFA;
  --IF file.fID = cacheA.file THEN RETURN[cacheA.size];
  --IF file.fID = cacheB.file THEN size ← cacheB.size ELSE
```



```
  --BEGIN
  --SimpleSpace.Map[pageBuffer, Space.WindowOrigin[file↑, 0], FALSE];
  --size ← faP.page + MIN[faP.byte, 1];
  --SimpleSpace.Unmap[pageBuffer]
  --END;
  --cacheB ← cacheA; cacheA ← [file.fID, size]
  --END;
  -- Look up file in cache and then in all vfm's, set vID and page group descriptor or signal error.

  GetVIDAndGroup: ENTRY PROCEDURE [
    pVID: POINTER TO Volume.ID, pGroup: POINTER TO FileInternal.PageGroup,
    pFile: POINTER TO READONLY File.Capability, filePage: File.PageNumber] =
    BEGIN
    fileD: FileInternal.Descriptor;
    IF ~GetFileDescriptor[pFile, @fileD] THEN RETURN WITH ERROR Unknown[pFile↑];
    WITH fileD SELECT FROM
      local =>
        BEGIN
        GetVIDAndGroup1: VolProc =
          BEGIN
          success: BOOLEAN;
          updateMarkers ← FALSE;
          [success, pGroup↑] ← FileCache.GetPageGroup[fileD.fileID, filePage];
          IF ~success THEN
            [success, pGroup↑] ← VolFileMap.GetPageGroup[
              volume, @fileD, filePage];
          IF ~(success AND filePage < pGroup.nextFilePage) THEN
            ERROR FileImplError[fileWithHole];
          END;

        pVID↑ ← fileD.volumeID;
        SELECT LogicalVolume.VolumeAccess[pVID, GetVIDAndGroup1] FROM
          ok => NULL;
          volumeUnknown => RETURN WITH ERROR Volume.Unknown[pVID↑];
          volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[pVID↑];
          ENDCASE;
        END;
      remote => RETURN WITH ERROR SystemInternal.Unimplemented;
      ENDCASE;
    END;

  LogContents: PUBLIC ENTRY PROCEDURE [
    transaction: Transaction.Handle, file: Capability, base: PageNumber,
    count: PageCount] =
    BEGIN OPEN TransactionState;
    ENABLE UNWIND => NULL;
    entry: LogEntry ← [setContents[window: [file, base], size: count]];
    Log[transaction, @entry, txFileProcs];
    END;

  MakeBootableOrUnbootable: ENTRY PROCEDURE [
    op: {bootable, unbootable}, file: POINTER TO READONLY File.Capability,
    firstPage: File.PageNumber, count: File.PageCount,
    lastLink: DiskChannel.Address] RETURNS [firstLink: DiskChannel.Address] =
    BEGIN
    fileD: FileInternal.Descriptor;
    group: FileInternal.PageGroup;
    limitPage: File.PageNumber = firstPage + count;
```

```
    nextLink: DiskChannel.Address;
    page: File.PageNumber ← firstPage;
    success: BOOLEAN;
    GetStuff: VolProc =
      BEGIN
      updateMarkers ← FALSE;
      [success, group] ← VolFileMap.GetPageGroup[volume, @fileD, page];
      IF ~success THEN ERROR FileImplError[makeBootablePageGroupNotFound];
      IF firstPage = page THEN
        firstLink ← SubVolume.GetPageAddress[
          fileD.volumeID,
          group.volumePage + (firstPage - group.filePage)].address;
      SELECT TRUE FROM
        op = unbootable => nextLink ← [0, 0, 0];
        group.nextFilePage >= limitPage => {
          group.nextFilePage ← limitPage; nextLink ← lastLink};
        ENDCASE => -- should check success of GetPageGroup in next statement
          nextLink ← SubVolume.GetPageAddress[
            fileD.volumeID, VolFileMap.GetPageGroup[
            volume, @fileD, group.nextFilePage].group.volumePage].address;
      IF group.filePage >= group.nextFilePage THEN
        ERROR FileImplError[makeBootableImpossibleError];
      group.volumePage ←
        group.volumePage + (group.nextFilePage - group.filePage) - 1;
      group.filePage ← group.nextFilePage - 1;
      END;

    TransferLabels: PROCEDURE => INLINE
      BEGIN
      SimpleSpace.Map[
        pageBuffer, [[fileD.fileID, File.read], group.filePage], TRUE, 1];
      LabelTransfer.WriteLabelAndData[
        fileD, group.filePage, group.volumePage, bufferPage, nextLink];
      SimpleSpace.Unmap[pageBuffer];
      END;

    IF ~GetFileDescriptor[file, @fileD] THEN RETURN WITH ERROR Unknown[file↑];
  IF (WITH fD: fileD SELECT FROM local => limitPage > fD.size, ENDCASE => TRUE).
    THEN RETURN WITH ERROR InvalidParameters;
  WHILE page < limitPage DO
    SELECT LogicalVolume.VolumeAccess[@fileD.volumeID, GetStuff] FROM
      ok => NULL;
      volumeUnknown => RETURN WITH ERROR Volume.Unknown[fileD.volumeID];
      volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[fileD.volumeID];
      ENDCASE;
    IF success THEN TransferLabels[];
    page ← group.nextFilePage;
    ENDLOOP;
  END;

OpenVolumeAndDeleteTemps: PUBLIC ENTRY PROCEDURE [
  volume: POINTER TO READONLY Volume.ID] =
BEGIN
SELECT LogicalVolume.OpenLogicalVolume[volume] FROM
  wasOpen => RETURN;
  ok => NULL;
  Unknown => RETURN WITH ERROR Volume.Unknown[volume↑];
  VolumeNeedsScavenging => RETURN WITH ERROR Volume.NeedsScavenging;
```



```
    ENDCASE;
  IF PilotSwitches.switches.u = up THEN
    IF DeleteTempsInternal[volume] # ok THEN
      ERROR FileImplError[volumeWentAwayDuringDeleteTemps];
  END;

Pin: PUBLIC ENTRY PROCEDURE [
  file: File.Capability, page: File.PageNumber, count: File.PageCount] =
  BEGIN
  fileD: FileInternal.Descriptor;
  after: File.PageNumber;
  IF ~GetFileDescriptor[@file, @fileD] THEN RETURN WITH ERROR Unknown[file];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      p: FileInternal.LocalFilePtr ← @f; -- FOR COMPILER BUG
      Pin1: VolProc =
        BEGIN
        success: BOOLEAN;
        group: FileInternal.PageGroup ← [, , page];
        updateMarkers ← FALSE;
        WHILE group.nextFilePage < after DO
          [success, group] ← VolFileMap.GetPageGroup[
            volume, p, group.nextFilePage];
          IF ~success THEN ERROR FileImplError[missingPinnedPageGroup];
          FileCache.SetPageGroup[file.fID, group, TRUE]
          ENDLOOP;
        END;

      after ←
        IF count = KernelFile.defaultPageCount THEN p.size ELSE page + count;
      FileCache.SetFile[f, TRUE];
      SELECT LogicalVolume.VolumeAccess[@f.volumeID, Pin1] FROM
        ok => NULL;
        volumeUnknown => RETURN WITH ERROR Volume.Unknown[f.volumeID];
        volumeNotOpen => RETURN WITH ERROR Volume.NotOpen[f.volumeID];
        ENDCASE;
      END;
    remote => RETURN WITH ERROR SystemInternal.Unimplemented;
    ENDCASE;
  END;

SetSize: PUBLIC ENTRY PROCEDURE [
  file: Capability, size: PageCount, transaction: Transaction.Handle] =
  BEGIN
  ENABLE UNWIND => NULL;
  fileD: FileInternal.Descriptor;
  IF ~GetFileDescriptor[@file, @fileD] THEN RETURN WITH ERROR Unknown[file];
  IF ~SufficientPermissions[file.permissions, File.write] THEN
    RETURN WITH ERROR Error[insufficientPermissions];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      IF f.immutable THEN RETURN WITH ERROR Error[immutable];
      IF transaction # nullTransactionHandle THEN
        IF size # f.size THEN
          BEGIN OPEN TransactionState;
          entry: LogEntry ←
            (SELECT size FROM
```

```
            > f.size => [shrink[file: file, size: f.size]],
            ENDCASE =>
              [setContents[window: [file, size], size: f.size - size]]);
        Log[transaction, @entry, txFileProcs];
        END;
    SELECT SetSizeInternal[@f, @size, file.permissions] FROM
      ok => RETURN;
      insufficientSpace => RETURN WITH ERROR Volume.InsufficientSpace;
      insufficientPermissions =>
        RETURN WITH ERROR Error[insufficientPermissions];
      ENDCASE;
    END;
  remote => RETURN WITH ERROR SystemInternal.Unimplemented;
  ENDCASE;
END;

-- this is its own entry procedure since it can get many nasty errors

TmpsEnter: ENTRY PROCEDURE [
  f: POINTER TO READONLY File.Capability, v: POINTER TO READONLY Volume.ID] =
  BEGIN

  FindIt: INTERNAL PROCEDURE RETURNS [BOOLEAN] = INLINE
    BEGIN
    tOriginal: LONG POINTER TO File.ID ← tmps;
    DO
      IF tmps↑ = File.nullID THEN RETURN[TRUE];
      IF (tmps ← tmps + SIZE[File.ID]) = tmpsLast THEN tmps ← tmpStart;
      IF tmps = tOriginal THEN EXIT;
      ENDLOOP;
    RETURN[FALSE];
    END;

  SELECT TmpsGet[v, last] FROM
    ok => NULL;
    volumeUnknown => RETURN WITH ERROR Volume.Unknown[v↑];
    notOpen => RETURN WITH ERROR Volume.NotOpen[v↑];
    noFile => RETURN; -- next delete tmps will be hard way

    ENDCASE;
  IF ~FindIt[] THEN
    IF ~TmpsGrow[] THEN RETURN WITH ERROR Volume.InsufficientSpace;
  tmps↑ ← f.fID;
  SimpleSpace.ForceOut[tmpsBuffer];
  END;

-- INTERNAL PROCEDURES

ChangeAttributesInternal: INTERNAL PROCEDURE [
  fileD: LocalFilePtr, action: AttributeAction]
  RETURNS [s: LogicalVolume.VolumeAccessStatus] =
  BEGIN
  file: Capability ← [fileD.fileID, maxPermissions];
  link: DiskChannel.Address ← LOOPHOLE[LONG[0]];
  volumePage: LogicalVolume.PageNumber;
  volID: Volume.ID ← fileD.volumeID; -- FOR COMPILER BUG
  GetGroup0: VolProc =
    BEGIN
    group: FileInternal.PageGroup;
```



```
    success: BOOLEAN;
    updateMarkers ← FALSE;
    [success, group] ← VolFileMap.GetPageGroup[volume, fileD, 0];
    IF ~success THEN ERROR FileImplError[missingPage0];
    volumePage ← group.volumePage;
    END;

  SELECT s ← LogicalVolume.VolumeAccess[@volID, GetGroup0] FROM
    ok => NULL;
    volumeUnknown, volumeNotOpen => RETURN;
    ENDCASE;
  IF ~LabelTransfer.VerifyLabels[fileD↑, [0, volumePage, 1]].labelsValid THEN
    SIGNAL LabelError; -- smash time, folks. Watch out for zero size file
  IF action = permanent OR action = temporary THEN
    BEGIN
    IF fileD.type IN PilotFileTypes.PilotRootFileType THEN
      LogicalVolume.PutRootFile[
        @volID, fileD.type, @file ! Volume.Unknown => GOTO VUnknown;
        Volume.NotOpen => GOTO VNotOpen];
    fileD.temporary ← action = temporary;
    EXITS VUnknown => RETURN[volumeUnknown]; VNotOpen => RETURN[volumeNotOpen];
    END
  ELSE fileD.immutable ← action = immutable;

  SimpleSpace.Map[
    pageBuffer, IF fileD.size = 0 THEN Space.defaultWindow ELSE [file, 0],
    TRUE];
  FileCache.FlushFile[file.fID];
  LOOPHOLE[link, PilotDisk.Address] ← LabelTransfer.ReadLabel[
    fileD↑, 0, volumePage].bootChainLink;
  LabelTransfer.WriteLabelAndData[
    fileD↑, 0, volumePage, SimpleSpace.Page[pageBuffer], link];
  SimpleSpace.Unmap[pageBuffer];
  FileCache.SetFile[fileD↑, FALSE]; -- could restore group, but lazy
  IF action = permanent THEN TmpsRemove[@file, @volID];
  END;

CreateWithIDInternal: INTERNAL PROCEDURE [
  volume: POINTER TO READONLY System.VolumeID,
  initialSize: POINTER TO READONLY PageCount, type: File.Type,
  file: POINTER TO READONLY File.ID]
  RETURNS [s: LogicalVolume .FileVolumeStatus[ok..insufficientSpace]] =
  -- Create file given the file identifier (called by CreateWithIDExternal,
  -- OpenVolumeAndDeleteTemps). Expect to see Volume.Unknown,
  -- Volume.NotOpen, Volume.InsufficientSpace
  BEGIN
  fileD: local FileInternal.Descriptor ←
    [file↑, volume↑, local[FALSE, TRUE, 0, type]];
  CreateInner: VolProc =
    BEGIN
    group: FileInternal.PageGroup ← [, 0, 0];
    updateMarkers ← FALSE;
    IF LogicalVolume.FreeVolumePages[volume] <= initialSize↑ THEN
      s ← insufficientSpace
    ELSE
      DO
        group ←
          [group.nextFilePage,
            group.volumePage + group.nextFilePage - group.filePage,
```

```
        initialSize↑];
      VolAllocMap.AllocPageGroup[volume, LONG[@fileD], @group, TRUE];
      -- results in modified group and FilePtr.size
      VolFileMap.InsertPageGroup[volume, LONG[@fileD], @group];
      IF fileD.size = initialSize↑ THEN EXIT;
      ENDLOOP;
    END;

  s ← ok;
  SELECT LogicalVolume.VolumeAccess[@fileD.volumeID, CreateInner, TRUE] FROM
    ok => IF s = ok THEN FileCache.SetFile[fileD, FALSE];
    volumeUnknown => s ← volumeUnknown;
    volumeNotOpen => s ← volumeNotOpen;
    ENDCASE;
  END;

DeleteFileOnVolumeInternal: INTERNAL PROCEDURE [
  fileD: FileInternal.LocalFilePtr]
  RETURNS [s: LogicalVolume.VolumeAccessStatus] =
  BEGIN
  vID: Volume.ID;
  DeleteIt: VolProc =
    BEGIN
    g: FileInternal.PageGroup;
    success: BOOLEAN;
    updateMarkers ← FALSE;
    DO
      [success, g] ← VolFileMap.GetPageGroup[
        volume, fileD, File.lastPageNumber];
      -- group.nextFile page is last page in file
      IF ~success THEN EXIT;
      g ← [0, LogicalVolume.nullVolumePage, g.nextFilePage];
      VolFileMap.DeletePageGroup[volume, fileD, @g];
      -- results in modified group
      VolAllocMap.FreePageGroup[volume, fileD, @g, TRUE];
      -- results in modified fileD

      ENDLOOP;
    END;

  FileCache.FlushFile[fileD.fileID];
  vID ← fileD.volumeID;
  RETURN[LogicalVolume.VolumeAccess[@vID, DeleteIt, TRUE]];
  END;

DeleteTempsInternal: INTERNAL PROCEDURE [v: POINTER TO READONLY Volume.ID]
  RETURNS [s: LogicalVolume.VolumeAccessStatus] =
  BEGIN
  one: File.PageCount ← 1;
  SELECT TmpsGet[v, first] FROM
    noFile => s ← DeleteTmpsInternalOld[v];
    ok => s ← DeleteTmpsInternalNew[v]; -- also deletes tempFile

    volumeUnknown => RETURN[volumeUnknown];
    notOpen => RETURN[volumeNotOpen];
    ENDCASE;
  IF s # ok THEN RETURN; -- Create a tmps file.
  tmpsFile ← [[System.GetUniversalID[]], FileInternal.maxPermissions];
```



```
  SELECT CreateWithIDInternal[
  v, @one, PilotFileTypes.tTempFileList, @tmpsFile.fID] FROM
    insufficientSpace => RETURN; -- there will be no temp file; that's Ok

    volumeUnknown, volumeNotOpen => FileImplError[tmpsVolumeWentAway];
    ENDCASE;
  LogicalVolume.PutRootFile[
    v, PilotFileTypes.tTempFileList, @tmpsFile !
    Volume.Unknown, Volume.NotOpen => FileImplError[tmpsVolumeWentAway]];
  END;

DeleteTmpsInternalNew: INTERNAL PROCEDURE [v: POINTER TO READONLY Volume.ID]
  RETURNS [s: LogicalVolume.VolumeAccessStatus] =
  BEGIN
  cap: File.Capability ← [File.nullID, File.delete];
  fileD: FileInternal.Descriptor;
  p: File.PageNumber;
  t: LONG POINTER TO File.ID;
  FOR p ← 0, p + 1 WHILE p < tmpsFileSize DO
    TmpsMap[p];
    FOR t ← tmpStart, t + SIZE[File.ID] WHILE t # tmpsLast DO
      IF t↑ = File.nullID THEN LOOP;
      cap.fID ← t↑;
      IF GetFileDescriptor[@cap, @fileD, v] THEN
        WITH f: fileD SELECT FROM
          local =>
            IF f.temporary THEN
              IF (s ← DeleteFileOnVolumeInternal[@f]) # ok THEN RETURN
              ELSE LOOP;
          ENDCASE; -- ingore remote files (can't happen), and delete errors

      ENDLOOP;
    ENDLOOP;
  TmpsUnmap[]; -- finally, delete the file itself.
  IF GetFileDescriptor[@tmpsFile, @fileD, v] THEN
    WITH f: fileD SELECT FROM
      local => RETURN[DeleteFileOnVolumeInternal[@f]];
      ENDCASE;
  ERROR FileImplError[disappearedOrRemoteTempFile];
  END;

DeleteTmpsInternalOld: INTERNAL PROCEDURE [v: POINTER TO READONLY Volume.ID]
  RETURNS [s: LogicalVolume.VolumeAccessStatus] =
  BEGIN
  lastCap: File.Capability ← File.nullCapability;
  thisCap: File.Capability;
  fileD: FileInternal.Descriptor;
  DO
    thisCap ← KernelFile.GetNextFile[
      v↑, lastCap ! Volume.Unknown => GOTO unknown;
      Volume.NotOpen => GOTO notOpen];
    IF thisCap = File.nullCapability THEN EXIT;
    IF GetFileDescriptor[@thisCap, @fileD, v] THEN
      WITH f: fileD SELECT FROM
        local =>
          IF f.temporary THEN
            IF (s ← DeleteFileOnVolumeInternal[@f]) # ok THEN RETURN ELSE LOOP;
        ENDCASE; -- ignore remote files (can't happen)
```

```
    lastCap ← thisCap;
    ENDLOOP;
  RETURN[ok];
  EXITS unknown => RETURN[volumeUnknown]; notOpen => RETURN[volumeNotOpen];
  END;

SetSizeInternal: INTERNAL PROCEDURE [
  fileD: FileInternal.LocalFilePtr, size: POINTER TO READONLY File.PageCount,
  permissions: File.Permissions] RETURNS [s: LogicalVolume.FileVolumeStatus] =
  BEGIN
  vs: LogicalVolume.VolumeAccessStatus;
  vID: Volume.ID;
  SizeIt: VolProc =
    BEGIN
    success: BOOLEAN;
    group: FileInternal.PageGroup;
    updateMarkers ← FALSE;
    SELECT fileD.size FROM
      < size↑ =>
        BEGIN
        IF ~SufficientPermissions[permissions, File.write + File.grow] THEN
          s ← insufficientPermissions
        ELSE
          IF LogicalVolume.FreeVolumePages[volume] < size↑ - fileD.size THEN
            s ← insufficientSpace
          ELSE
            BEGIN
            [success, group] ← VolFileMap.GetPageGroup[
              volume, fileD, fileD.size - MIN[fileD.size, 1]];
            IF ~success THEN ERROR FileImplError[missingPageGroupSetSize];
            WHILE fileD.size < size↑ DO
              group ←
                [group.nextFilePage,
                  group.volumePage + (group.nextFilePage - group.filePage),
                  size↑];
              VolAllocMap.AllocPageGroup[volume, fileD, @group, FALSE];
              -- change group + fileD
              VolFileMap.InsertPageGroup[volume, fileD, @group];
              ENDLOOP
            END;
        END;
      > size↑ =>
        BEGIN
        IF ~SufficientPermissions[permissions, File.write + File.shrink] THEN
          s ← insufficientPermissions
        ELSE
          BEGIN
          WHILE fileD.size > size↑ DO
            group ← [size↑, LogicalVolume.nullVolumePage, fileD.size];
            VolFileMap.DeletePageGroup[volume, fileD, @group]; -- changes group
            VolAllocMap.FreePageGroup[volume, fileD, @group, FALSE];
            -- changes FilePtr

            ENDLOOP;
          END;
        END;
      ENDCASE;
    END;

  s ← ok;
```



```
  FileCache.FlushFile[fileD.fileID];
  vID ← fileD.volumeID;
  IF (vs ← LogicalVolume.VolumeAccess[@vID, SizeIt, TRUE]) # ok THEN RETURN[vs];
  FileCache.SetFile[fileD↑, FALSE];
  END;

TmpsGet: INTERNAL PROCEDURE [
  v: POINTER TO READONLY Volume.ID, pg: {first, last}]
  RETURNS [{ok, volumeUnknown, notOpen, noFile}] =
  BEGIN
  IF v↑ = Volume.nullID THEN RETURN[volumeUnknown];
  IF tmpsVolume # v↑ THEN
    BEGIN
    fileD: FileInternal.Descriptor;
    TmpsUnmap[];
    tmpsFile ← KernelFile.GetRootFile[
      PilotFileTypes.tTempFileList, v↑ ! Volume.Unknown => GOTO unknown;
      Volume.NotOpen => GOTO notOpen];
    IF ~GetFileDescriptor[@tmpsFile, @fileD, v] THEN RETURN[noFile];
    WITH fileD SELECT FROM
      local => tmpsFileSize ← size;
      ENDCASE => ERROR FileImplError[remoteTmpsFile];
    TmpsMap[IF pg = first THEN 0 ELSE tmpsFileSize - 1];
    tmpsVolume ← v↑;
    EXITS unknown => RETURN[volumeUnknown]; notOpen => RETURN[notOpen];
    END;
  RETURN[ok];
  END;

TmpsGrow: INTERNAL PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  fileD: FileInternal.Descriptor;
  newSize: File.PageCount ← tmpsFileSize + 1;
  IF ~GetFileDescriptor[@tmpsFile, @fileD, @tmpsVolume] THEN
    ERROR FileImplError[tmpsFileWentAway];
  WITH f: fileD SELECT FROM
    local =>
      SELECT SetSizeInternal[@f, @newSize, File.grow + File.write] FROM
        insufficientSpace => RETURN[FALSE];
        ok => NULL;
        ENDCASE => FileImplError[tmpsFileProblem];
    ENDCASE => ERROR FileImplError[remoteTmpsFile];
  TmpsMap[tmpsFileSize];
  tmpsFileSize ← tmpsFileSize + 1;
  RETURN[TRUE];
  END;

TmpsMap: INTERNAL PROCEDURE [page: File.PageNumber] = INLINE
  BEGIN
  IF tmpsVolume # nullID THEN SimpleSpace.Unmap[tmpsBuffer];
  SimpleSpace.Map[tmpsBuffer, [tmpsFile, page], FALSE];
  tmps ← tmpStart;
  END;

TmpsRemove: INTERNAL PROCEDURE [
  f: POINTER TO READONLY File.Capability, v: POINTER TO READONLY Volume.ID] =
  BEGIN
  t: LONG POINTER TO File.ID ← tmps;
  IF v↑ # Volume.nullID -- impossible, he says-- AND tmpsVolume = v↑ THEN
```

```
    DO
      -- only cheap wins
      IF t↑ = f.fID THEN BEGIN t↑ ← File.nullID; RETURN; END;
      IF t = tmpStart THEN t ← tmpsLast;
      IF (t ← t - SIZE[File.ID]) = tmps THEN EXIT;
      ENDLOOP;
  END;

TmpsUnmap: INTERNAL PROCEDURE = INLINE
  BEGIN
  IF tmpsVolume # Volume.nullID THEN
    BEGIN SimpleSpace.Unmap[tmpsBuffer]; tmpsVolume ← Volume.nullID; END;
  END;

-- Here follows a gross kludge to allow the transaction machinery to sneak
-- past our monitor lock.

txFileProcs: TransactionState.FileOps = [TxCreate, TxMakePerm, TxSetSize];

TxCreate: INTERNAL PROCEDURE [size: PageCount, type: Type] RETURNS [file: ID] =
  BEGIN
  volID: Volume.ID ← Volume.systemID;
  file ← [System.GetUniversalID[]];
  SELECT CreateWithIDInternal[@volID, @size, type, @file] FROM
    ok => NULL;
    insufficientSpace => ERROR Volume.InsufficientSpace;
    ENDCASE;
  END;

TxMakePerm: INTERNAL PROCEDURE [file: ID] =
  BEGIN
  fileC: Capability ← [file, maxPermissions];
  fileD: FileInternal.Descriptor;
  IF ~GetFileDescriptor[@fileC, @fileD] THEN ERROR Unknown[fileC];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      SELECT ChangeAttributesInternal[@f, permanent] FROM
        volumeUnknown => ERROR Volume.Unknown[f.volumeID];
        volumeNotOpen => ERROR Volume.NotOpen[f.volumeID];
        ENDCASE;
      END;
    ENDCASE => ERROR FileImplError[remoteTxLog];
  END;

TxSetSize: INTERNAL PROCEDURE [file: ID, size: PageCount] =
  BEGIN
  fileC: Capability ← [file, maxPermissions];
  fileD: FileInternal.Descriptor;
  IF ~GetFileDescriptor[@fileC, @fileD] THEN ERROR Unknown[fileC];
  WITH f: fileD SELECT FROM
    local =>
      BEGIN
      SELECT SetSizeInternal[@f, @size, maxPermissions] FROM
        insufficientSpace => ERROR Volume.InsufficientSpace;
        volumeUnknown => ERROR Volume.Unknown[f.volumeID];
        volumeNotOpen => ERROR Volume.NotOpen[f.volumeID];
        ENDCASE;
      END;
```



```
    ENDCASE => ERROR FileImplError[remoteTxLog];
  END;

-- Initialization

InitializeFileMgr: PUBLIC PROCEDURE [
  bootFile: LONG POINTER TO disk Boot.Location,
  pLVBootFiles: POINTER TO Boot.LVBootFiles]
  RETURNS [debuggerDeviceType: Device.Type, debuggerDeviceOrdinal: CARDINAL] =
  BEGIN
  IF PilotSwitches.switches.f = down THEN Runtime.CallDebugger["Key Stop F"L];
  Process.Detach[FORK FileHelperProcess[]];
  START FMPrograms.VolAllocMapImpl;
  START FMPrograms.VolFileMapImpl;
  START FMPrograms.MarkerPageImpl;
  [debuggerDeviceType, debuggerDeviceOrdinal] ← START
    FMPrograms.PhysicalVolumeImpl[bootFile, pLVBootFiles];
  -- This will also start VolumeImpl and ScavengeImpl.

  END;


END.
```

(For earlier log entries see Pilot 4.0 archive version.)

April 1, 1980 2:47 PM Gobbel Added transaction handles
April 15, 1980 4:37 PM Gobbel Added nullTransactionHandle
April 17, 1980 10:08 PM Luniewski Added TransactionHandle argument to GetSize and Ge
***tAttributes*
April 18, 1980 2:59 PM Luniewski Temporary exportation of transaction.nullHandle adde
***d*
May 15, 1980 6:11 PM McJones Add page and count parameters to Pin
May 30, 1980 3:07 PM Luniewski PhysicalVolume removed from DIRECTORY. Made co
***mpatible with FileInternal.FileDescriptors being LONG POINTERs. STARTed PhysicalVolumeIm*
***pl. Removed START of VolumeImpl as it is now started by PhysicalVolumeImpl. LogicalVolume*
***.{Open Close}Volume => LogicalVolume.{Open Close}LogicalVolume.*
June 10, 1980 1:20 AM Gobbel Added MakeTemporary and MakeMutable
June 10, 1980 3:10 PM Gobbel Added transaction logging code
July 19, 1980 11:15 AM McJones Deleted nullTransactionHandle and (Transaction.)nullH
***andle; adapted for new PilotDisk.Label; removed dependency in initialization of tmpsLast on SI*
***ZE[File.ID]*
July 25, 1980 3:32 PM Luniewski SpecialVolume.(NotOpen VolumeNeedsScavenging) =
***> Volume.(NotOpen NeedsScavenging)*
August 11, 1980 5:24 PM Gobbel/Knutsen Restructure Create to allow export of Create
***WithID to KernelFile for transaction crash recovery*
August 14, 1980 11:15 AM McJones Delete dependencies on tBootFile; FilePageLabel = >Pil
***otDisk; require File.write in SetSizeInternal*
August 21, 1980 2:10 PM Gobbel Restructure transactional operations to use new interfa
***ce for managing log files*
September 12, 1980 12:07 PM Gobbel Make GetFileDescriptor look at all volume types
September 17, 1980 3:07 PM Luniewski Mods for new priocedure type passed to LogicalVolu
***me.VolumeAccess.*
September 26, 1980 4:29 PM McJones GetBootLocation forgot to add offset to volume page

```
-- MarkerPageImpl.mesa (last edited by: Luniewski on: September 17, 1980 2:44 PM)
-- It is not clear to me how much checking to do when something comes on line. For the time bein
**g, We do no real checking, assuming that marker pages are set up right. This may be the corre
**ct philosophy, since marker pages are copies of the truth for the scavenger.

DIRECTORY
  DiskChannel USING [Drive, GetAttributes, nullDrive],
  File USING [ID, nullID],
  FMPrograms USING [],
  FileInternal USING [maxPermissions],
  LabelTransfer USING [WriteLabels],
  LogicalVolume USING [Handle, LogicalSubvolumeMarker],
  MarkerPage USING [CacheKey, SubVolumeMarkerPage],
  PhysicalVolume USING [Error, ErrorType, ID, nullID],
  PhysicalVolumeFormat USING [Handle, PageNumber, PhysicalSubvolumeMarker],
  PilotFileTypes USING [tSubVolumeMarkerPage],
  SimpleSpace USING [Create, Handle, Map, Page, Unmap],
  SubVolume USING [Find, Handle],
  Utilities USING [LongPointerFromPage];

MarkerPageImpl: MONITOR
  IMPORTS
    DiskChannel, LabelTransfer, PhysicalVolume, SimpleSpace, SubVolume, Utilities
  EXPORTS FMPrograms, MarkerPage
  SHARES File =
  BEGIN
  -- Buffer variables for accessing marker pages
  markerPageBuffer: SimpleSpace.Handle = SimpleSpace.Create[1, hyperspace];
  marker: LONG POINTER TO MarkerPage.SubVolumeMarkerPage =
    Utilities.LongPointerFromPage[SimpleSpace.Page[markerPageBuffer]];
  cacheSize: CARDINAL = 5; -- should be one for each possible disk
  cache: ARRAY [0..cacheSize) OF RECORD [
    occupied: BOOLEAN,
    drive: DiskChannel.Drive,
    physicalID: PhysicalVolume.ID,
    markerID: File.ID] ← ALL[[FALSE, , , ]];
  NotFound: PUBLIC ERROR = CODE;
  MarkerPageError: ERROR [
    {cacheFull, impossibleCacheVarient, subvolumeNotFound}] = CODE;
  -- Create the marker page for the i'th subvolume. The ID is in the cache, and the page groups ha
**ve been set up properly
  CreateMarkerPage: PUBLIC ENTRY PROCEDURE [
    pvHandle: PhysicalVolumeFormat.Handle, lvHandle: LogicalVolume.Handle,
    physicalSubvolumeNumber: CARDINAL] =
    BEGIN OPEN sv: pvHandle.subVolumes[physicalSubvolumeNumber];
    page: LONG CARDINAL ← sv.pvPage + sv.nPages;
    LabelTransfer.WriteLabels[
      file:
      [FindMarkerID[[physicalID[pvHandle.pvID]]], LOOPHOLE[pvHandle.pvID], local[
        FALSE, FALSE, page --Should be driveSize--,
        PilotFileTypes.tSubVolumeMarkerPage]], pageGroup: [page, page, page + 1]];
    MarkerPageMap[FindMarkerID[[physicalID[pvHandle.pvID]]], page];
    marker↑ ←
      [physical:
       [pvID: pvHandle.pvID, label: pvHandle.label,
         bootingInfo: pvHandle.bootingInfo, maxBadPages: pvHandle.maxBadPages,
         labelLength: pvHandle.labelLength, svNumber: physicalSubvolumeNumber,
         descriptor: sv],
```



```
       logical:
       [labelLength: lvHandle.labelLength, type: lvHandle.type,
         label: lvHandle.label, bootingInfo: lvHandle.bootingInfo,
         clientRootFile: lvHandle.clientRootFile]];
    MarkerPageUnmap[];
    END;

  Enter: PUBLIC ENTRY PROCEDURE [
    drive: DiskChannel.Drive, physicalID: PhysicalVolume.ID] =
    BEGIN
    i: CARDINAL;
    FOR i IN [0..LENGTH[cache]) DO
      IF drive = cache[i].drive THEN
        BEGIN cache[i] ← [TRUE, drive, physicalID, File.nullID]; RETURN; END;
      ENDLOOP;
    FOR i IN [0..LENGTH[cache]) DO
      IF ~cache[i].occupied THEN
        BEGIN cache[i] ← [TRUE, drive, physicalID, File.nullID]; RETURN; END;
      ENDLOOP;
    MarkerPageError[cacheFull];
    END;

  EnterMarkerID: PUBLIC ENTRY PROCEDURE [
    c: MarkerPage.CacheKey, markerID: File.ID] =
    BEGIN
    ENABLE UNWIND => NULL;
    i: CARDINAL ← FindInternal[c];
    cache[i].markerID ← markerID;
    END;
  -- Find is exported because some of Special Volume operations use it.

  Find: PUBLIC ENTRY PROCEDURE [c: MarkerPage.CacheKey]
    RETURNS [
      drive: DiskChannel.Drive, physicalID: POINTER TO READONLY PhysicalVolume.ID,
      markerID: POINTER TO READONLY File.ID] =
    BEGIN
    ENABLE UNWIND => NULL;
    i: CARDINAL ← FindInternal[c];
    RETURN[cache[i].drive, @cache[i].physicalID, @cache[i].markerID];
    END;

  Flush: PUBLIC ENTRY PROCEDURE [c: MarkerPage.CacheKey] =
    BEGIN
    ENABLE UNWIND => NULL;
    i: CARDINAL ← FindInternal[c];
    cache[i].occupied ← FALSE;
    END;

  GetNextPhysicalVolume: PUBLIC ENTRY PROCEDURE [thisPvID: PhysicalVolume.ID]
    RETURNS [nextPvID: PhysicalVolume.ID, drive: DiskChannel.Drive] =
    BEGIN
    i: CARDINAL;
    found: BOOLEAN ← (thisPvID = PhysicalVolume.nullID);
    FOR i IN [0..LENGTH[cache]) DO
      IF cache[i].occupied THEN
        IF found THEN RETURN[LOOPHOLE[cache[i].physicalID], cache[i].drive]
        ELSE found ← (thisPvID = LOOPHOLE[cache[i].physicalID]);
      ENDLOOP;
    IF found THEN RETURN[PhysicalVolume.nullID, DiskChannel.nullDrive]
```

```
  ELSE RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  END;

UpdateLogicalMarkerPages: PUBLIC ENTRY PROCEDURE [
  lvHandle: LogicalVolume.Handle] =
  BEGIN
  svH: SubVolume.Handle ← NIL;
  lpage: LONG CARDINAL ← 0;
  found: BOOLEAN;
  WHILE lpage < lvHandle.volumeSize DO
    [found, svH] ← SubVolume.Find[lvHandle.vID, lpage];
    IF ~found THEN ERROR MarkerPageError[subvolumeNotFound];
    MarkerPageMap[
      FindMarkerID[[drive[DiskChannel.GetAttributes[svH.channel].drive]]],
      svH.pvPage + svH.nPages];
    marker.logical.bootingInfo ← lvHandle.bootingInfo;
    marker.logical.clientRootFile ← lvHandle.clientRootFile;
    MarkerPageUnmap[];
    lpage ← svH.lvPage + svH.nPages;
    ENDLOOP;
  END;

UpdatePhysicalMarkerPages: PUBLIC ENTRY PROCEDURE [
  pvHandle: PhysicalVolumeFormat.Handle] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..pvHandle.subVolumeCount) DO
    OPEN sv: pvHandle.subVolumes[i];
    MarkerPageMap[
      FindMarkerID[[physicalID[pvHandle.pvID]]], sv.pvPage + sv.nPages];
    marker.physical.bootingInfo ← pvHandle.bootingInfo;
    MarkerPageUnmap[];
    ENDLOOP;
  END;
-- Internal Procedures

FindInternal: INTERNAL PROCEDURE [c: MarkerPage.CacheKey]
  RETURNS [i: CARDINAL] =
  BEGIN
  FOR i IN [0..LENGTH[cache]) DO
    IF cache[i].occupied THEN
      WITH c SELECT FROM
        drive => IF h = cache[i].drive THEN RETURN;
        physicalID => IF id = cache[i].physicalID THEN RETURN;
        ENDCASE => ERROR MarkerPageError[impossibleCacheVarient];
    ENDLOOP;
  ERROR NotFound;
  END;

FindMarkerID: INTERNAL PROCEDURE [c: MarkerPage.CacheKey] RETURNS [File.ID] =
  INLINE BEGIN RETURN[cache[FindInternal[c]].markerID]; END;

MarkerPageMap: INTERNAL PROCEDURE [ID: File.ID, page: LONG CARDINAL] =
  BEGIN
  SimpleSpace.Map[
    markerPageBuffer, [[ID, FileInternal.maxPermissions], page], FALSE];
  END;
```



```
  MarkerPageUnmap: INTERNAL PROCEDURE =
    BEGIN SimpleSpace.Unmap[markerPageBuffer]; END;

  END.....
LOG
Time: October 1, 1979  12:20 PM      By: Forrest      Action: Created from volumeImpl.
**mesa
Time: October 16, 1979  12:44 PM     By: Forrest      Action: Fix bug in Update SVM pa
**ges
Time: May 20, 1980  4:14 PM    By: Luniewski      Action: PhysicalVolume => PhysicalVolume
**Format
Time: July 24, 1980  9:14 AM   By: Luniewski      Action: Deleted Export to SpecialVolume.  M
**ade GetNextPhysicalVolume export to MarkerPage.
Time: September 17, 1980  2:44 PM    By: Luniewski      Action: Added maxBadpages to
**marker page.
```

```
-- PhysicalVolumeImpl.mesa (last edited by: Luniewski on: October 7, 1980 3:01 PM)

DIRECTORY
  Boot USING [Location, LVBootFiles, PVBootFiles, VolumeType],
  Device USING [nullType, Type],
  DiskChannel USING [
    AwaitStateChange, Create, DiskPageCount, Drive, DriveState, DriveStatus,
    GetAttributes, GetDriveAttributes, GetNextDrive, GetPageNumber, nullDrive,
    nullHandle, PVHandle, SetDriveState, SetDriveTag],
  Environment USING [wordsPerPage],
  File USING [Capability, ID, nullCapability, PageCount, Permissions, read, Type],
  FileInternal USING [maxPermissions],
  FMPrograms USING [ScavengeImpl, VolumeImpl],
  Inline USING [LowHalf],
  LabelTransfer USING [LabelStatus, ReadRootLabel, WriteLabels],
  LogicalVolume USING [Handle, rootPageNumber],
  MarkerPage USING [
    CacheKey, Enter, EnterMarkerID, Find, Flush, GetNextPhysicalVolume, NotFound,
    UpdatePhysicalMarkerPages],
  PhysicalVolume USING [
    CanNotScavenge, Error, ErrorType, ID, Layout, nullBadPage, nullDeviceIndex,
    nullID, PageNumber, VolumeType],
  PhysicalVolumeFormat USING [
    currentVersion, descriptorSize, Handle, IDCheckSum, maxSubVols, nullBadPage,
    nullPVBootFiles, PageNumber, rootPageNumber, Seal, seal],
  PilotDisk USING [GetLabelFilePage, GetLabelType, Label],
  PilotFileTypes USING [tPhysicalVolumeRootPage, tSubVolumeMarkerPage],
  PilotMP USING [cDeleteTemps, cDriveNotReady],
  PilotSwitches USING [switches],
  ProcessorFace USING [SetMP],
  Runtime USING [CallDebugger],
  SimpleSpace USING [Create, ForceOut, Handle, Map, Page, Unmap],
  SpecialVolume USING [nullSubVolume, SubVolume, SubVolumeUnknown],
  SubVolume USING [completion, Find, GetNext, Handle, OffLine, OnLine],
  System USING [GetUniversalID, nullID],
  Utilities USING [LongPointerFromPage],
  VolAllocMap USING [Close],
  VolFileMap USING [Close],
  Volume USING [ID, nullID, PageCount, systemID, Type, Unknown],
  VolumeImplInterface USING [
    BarePvID, CheckLogicalVolume, FindLogicalVolume, GetLVStatus,
    LogicalVolumeCreate, LogicalVolumeErase, OpenInitialVolumes, PinnedFileFlush,
    readOrWrite, RegisterLogicalSubvolume, SubvolumeOffline, SubvolumeOnline,
    VFileEnter],
  VolumeInternal USING [PageNumber];

PhysicalVolumeImpl: MONITOR [
  bootFile: LONG POINTER TO disk Boot.Location,
  pLVBootFiles: POINTER TO Boot.LVBootFiles]
  RETURNS [debuggerDeviceType: Device.Type, debuggerDeviceOrdinal: CARDINAL]
  IMPORTS
    DiskChannel, FMPrograms, Inline, LabelTransfer, MarkerPage, PhysicalVolume,
    PhysicalVolumeFormat, PilotDisk, PilotSwitches, ProcessorFace, Runtime,
    SimpleSpace, SpecialVolume, SubVolume, System, Utilities, VolAllocMap,
    VolFileMap, Volume, VolumeImplInterface
  EXPORTS FMPrograms, PhysicalVolume, SpecialVolume, VolumeImplInterface
  SHARES File, PhysicalVolume =
BEGIN
```



```
-- A note about dependencies. Logically, physical volumes lie below logical volumes.
-- Thus, this module may not call any of the logical volume entries while it has the
-- monitor locked. The volume online/offline machinery is special in that it is logically
-- above logical volumes. Thus, it has need to call into that machinery. The current
-- implementation of these functions imply assumes that no calls back into this module
-- will occur.

PhysicalVolumeImplError: ERROR [ErrorType] = CODE;
ErrorType: TYPE = {
  impossibleDriveStateChangeFailure, impossibleOffLineFailure,
  impossibleSelectError, invalidSubVolumeNumber, physicalVolumeNotFound};
LvHandle: TYPE = LogicalVolume.Handle;
PvHandle: TYPE = PhysicalVolumeFormat.Handle;
debugClass: Boot.VolumeType;
debugSearchState: {tooSoonToLook, looking, done} ← tooSoonToLook;
pvRootPage: PhysicalVolumeFormat.PageNumber =
  PhysicalVolumeFormat.rootPageNumber;

-- Buffer variables for accessing physical volume root page

-- Eventually, set rootPagePages based upon the device type of a volume. When this
-- happens, we will have to arrange to have (arbitrarily?) larger buffers.
rootPagePages: File.PageCount =
  PhysicalVolumeFormat.descriptorSize/Environment.wordsPerPage;
-- the following space MUST be used by PhysicalRootPageAccessInternal as there
-- procedures that do ForceOut's on this space before returning to this procedure
physicalRootPageBuffer: SimpleSpace.Handle =
  -- WE ASSUME that the root page isn't too big
  SimpleSpace.Create[LOOPHOLE[Inline.LowHalf[rootPagePages]], hyperspace, 1];
physicalVolume: PvHandle = Utilities.LongPointerFromPage[
  SimpleSpace.Page[physicalRootPageBuffer]];

-- The following are the variables and procedures exported to PhysicalVolume

maxSubvolumesOnPhysicalVolume: PUBLIC CARDINAL ←
  PhysicalVolumeFormat.maxSubVols;
CanNotScavenge: PUBLIC ERROR = CODE;
-- The following is exported by ScavengeImpl due to a compiler limitation. It should be exported
**here logically.
--Error: PUBLIC ERROR [PhysicalVolume.ErrorType] = CODE;
Handle: PUBLIC TYPE = DiskChannel.PVHandle;

AssertNotAPilotVolume: PUBLIC ENTRY PROCEDURE [instance: Handle] =
  BEGIN
  IF ~ValidateDrive[instance.drive] THEN
    RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
  SELECT DiskChannel.SetDriveState[
  instance.drive, instance.changeCount, channel] FROM
    invalidDrive => RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
    alreadyAsserted => RETURN WITH ERROR PhysicalVolume.Error[alreadyAsserted];
    ENDCASE;
  END;

AssertPilotVolume: PUBLIC ENTRY PROCEDURE [instance: Handle]
  RETURNS [pvID: PhysicalVolume.ID] =
  -- This procedure assumes that this module will not be called by VolumeImpl when this proced
**ure calls VolumeImpl through VolumeImplInterface. If this assumption should ever not be true,
**this procedure must be modified to release the monitor lock before calling into VolumeImpl.
```

```
    BEGIN
    localError: PhysicalVolume.ErrorType;
    resetDriveState: BOOLEAN ← FALSE;
    BEGIN
    IF ~ValidateDrive[instance.drive] THEN
      RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
    IF ~DiskChannel.GetDriveAttributes[instance.drive].ready THEN
      RETURN WITH ERROR PhysicalVolume.Error[notReady];
    SELECT DiskChannel.SetDriveState[instance.drive, instance.changeCount, pilot]
    FROM
      invalidDrive => RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
      alreadyAsserted => RETURN WITH ERROR PhysicalVolume.Error[alreadyAsserted];
      ENDCASE;
    resetDriveState ← TRUE;
    [id: pvID] ← PhysicalVolumeOnLineInternal[
      instance.drive !
      PhysicalVolume.Error => {localError ← error; GO TO pvError};
      PhysicalVolume.CanNotScavenge => GO TO unscavengeable; ];
    EXITS
      pvError => {
        IF resetDriveState THEN -- 0 2'nd arg => guaranteed ok as a status
          [] ← DiskChannel.SetDriveState[instance.drive, 0, inactive];
        RETURN WITH ERROR PhysicalVolume.Error[localError]};
      unscavengeable => {
        IF resetDriveState THEN -- 0 as 2'nd arg => ok as a status
          [] ← DiskChannel.SetDriveState[instance.drive, 0, inactive];
        RETURN WITH ERROR PhysicalVolume.CanNotScavenge};
    END
    END;

  AwaitStateChange: PUBLIC PROCEDURE [
    changeCount: CARDINAL, type: Device.Type, index: CARDINAL]
    RETURNS [currentChangeCount: CARDINAL] = {
    RETURN[DiskChannel.AwaitStateChange[changeCount, type, index]]};

  CreateLogicalVolume: PUBLIC PROCEDURE [
    pvID: PhysicalVolume.ID, size: Volume.PageCount, name: STRING,
    type: Volume.Type, minPVPageNumber: PhysicalVolume.PageNumber]
    RETURNS [Volume.ID] = {
    RETURN[
      VolumeImplInterface.LogicalVolumeCreate[
        pvID, size, name, type, minPVPageNumber]]};

  CreatePhysicalVolume: PUBLIC ENTRY PROCEDURE [instance: Handle, name: STRING]
    RETURNS [PhysicalVolume.ID] =
    BEGIN
    localError: PhysicalVolume.ErrorType;
    BEGIN
    i, labelLength: CARDINAL;
    pvID: VolumeImplInterface.BarePvID;
    ready: BOOLEAN;
    changeCount: CARDINAL;
    state: DiskChannel.DriveState;
    IF name = NIL OR name.length = 0 THEN
      RETURN WITH ERROR PhysicalVolume.Error[nameRequired];
    IF ~ValidateDrive[instance.drive] THEN
      RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
    [ready: ready, changeCount: changeCount, state: state] ←
      DiskChannel.GetDriveAttributes[instance.drive];
```



```
    IF changeCount ~= instance.changeCount THEN
      RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
    IF ~ready THEN RETURN WITH ERROR PhysicalVolume.Error[notReady];
    IF state = pilot THEN
      BEGIN
      PhysicalVolumeOffLineInternal[
        instance.drive !
        PhysicalVolume.Error => {localError ← error; GO TO pvError}];
      IF DiskChannel.SetDriveState[instance.drive, instance.changeCount, inactive]
        ~= ok THEN ERROR;
      END
    ELSE
      IF state = channel THEN
        RETURN WITH ERROR PhysicalVolume.Error[alreadyAsserted];
    IF DiskChannel.SetDriveState[instance.drive, instance.changeCount, pilot] ~=
      ok THEN ERROR;
    pvID ← System.GetUniversalID[];
    RegisterPvInfo[pvID, instance.drive];
    LabelTransfer.WriteLabels[
      -- take care. The label may be more than one page!
      [[pvID], [pvID], local[
        FALSE, FALSE, rootPagePages, PilotFileTypes.tPhysicalVolumeRootPage]],
      [pvRootPage, pvRootPage, pvRootPage + rootPagePages]];
    PhysicalRootPageMap[pvID, readWrite];
    labelLength ← MIN[name.length, LENGTH[physicalVolume.label]];
    -- When the bad page list has a device dependent length, be sure to assign to maxBadPages in
**the following constructor
    physicalVolume↑ ←
      [subVolumeCount: 0, labelLength: labelLength, pvID: [pvID], label: NULL,
        subVolumes: NULL, badPageList: ALL[PhysicalVolumeFormat.nullBadPage]];
    FOR i IN [0..labelLength) DO physicalVolume.label[i] ← name[i]; ENDLOOP;
    RegisterSubvolumeMarker[physicalVolume];
    PhysicalRootPageUnmap[];
    RETURN[[pvID]];
    EXITS pvError => RETURN WITH ERROR PhysicalVolume.Error[localError];
    END
    END;

  EraseLogicalVolume: PUBLIC PROCEDURE [lvID: Volume.ID] = {
    VolumeImplInterface.LogicalVolumeErase[lvID]};

  FinishWithNonPilotVolume: PUBLIC ENTRY PROCEDURE [instance: Handle] =
    BEGIN
    IF ~ValidateDrive[instance.drive] THEN
      RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
    SELECT DiskChannel.GetDriveAttributes[instance.drive].state FROM
      inactive => RETURN;
      pilot => RETURN WITH ERROR PhysicalVolume.Error[hasPilotVolume];
      ENDCASE;
    SELECT DiskChannel.SetDriveState[
    instance.drive, instance.changeCount, inactive] FROM
      invalidDrive => RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
      ok => NULL;
      ENDCASE => ERROR;
    END;

  GetAttributes: PUBLIC ENTRY PROCEDURE [pvID: PhysicalVolume.ID, label: STRING]
    RETURNS [instance: Handle, layout: PhysicalVolume.Layout] =
    BEGIN
```

```
  drive: DiskChannel.Drive = GetPVDrive[pvID];
  IF drive = DiskChannel.nullDrive THEN
    RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  instance ← [drive, DiskChannel.GetDriveAttributes[drive].changeCount];
  IF GetPhysicalVolumeAttributes[pvID, label].subvolumeCount = 1 THEN
    -- Has the side effect of setting the callers label.  Can not return with found = FALSE.
    BEGIN
    lvSize: Volume.PageCount;
    svSize: Volume.PageCount;
    [lvSize: lvSize, subVolumeSize: svSize] ← GetSubVolumeAttributes[pvID, 0];
    -- Can not raise its error due to success of GetPhysicalVolumeAttributes above.
    layout ←
      IF lvSize = svSize THEN singleLogicalVolume ELSE partialLogicalVolume;
    END
  ELSE layout ← multipleLogicalVolumes;
  END;

GetContainingPhysicalVolume: PUBLIC ENTRY PROCEDURE [lvID: Volume.ID]
  RETURNS [pvID: PhysicalVolume.ID] =
  BEGIN
  -- This is an ENTRY procedure so that the volume containing lvID can not go
  -- away between the time that SubVolume is called and MarkerPage is called
  success: BOOLEAN;
  sv: SubVolume.Handle;
  [success, sv] ← SubVolume.Find[lvID, LogicalVolume.rootPageNumber];
  IF ~success THEN RETURN WITH ERROR Volume.Unknown[lvID];
  RETURN[
    MarkerPage.Find[[drive[DiskChannel.GetAttributes[sv.channel]]]].physicalID↑]
  END;

GetHandle: PUBLIC ENTRY PROCEDURE [type: Device.Type, index: CARDINAL]
  RETURNS [Handle] =
  BEGIN
  drive: DiskChannel.Drive;
  IF type = Device.nullType OR index = PhysicalVolume.nullDeviceIndex THEN
    RETURN WITH ERROR PhysicalVolume.Error[noSuchDrive];
  IF (drive ← GetDrive[type, index]) = DiskChannel.nullDrive THEN
    RETURN WITH ERROR PhysicalVolume.Error[noSuchDrive];
  RETURN[[drive, DiskChannel.GetDriveAttributes[drive].changeCount]]
  END;

GetHints: PUBLIC ENTRY PROCEDURE [instance: Handle, label: STRING]
  RETURNS [pvID: PhysicalVolume.ID, volumeType: PhysicalVolume.VolumeType] =
  BEGIN
  -- This should get smarter about returning information when the volume is partially
  -- trashed.  Also, it should not work by onlining/offlining the volume as this has side
  -- effects (in principle) which this operation should not produce.
  CopyLabel: PROCEDURE [pvLabel: PvHandle] =
    BEGIN
    FOR i: CARDINAL IN
      [0..(label.length ← MIN[label.maxlength, pvLabel.labelLength])) DO
      label[i] ← pvLabel.label[i]; ENDLOOP;
    END;
  driveChangeCount: CARDINAL;
  driveState: DiskChannel.DriveState;
  ready: BOOLEAN;
  found: BOOLEAN;
  onlineFound: BOOLEAN;
  IF ~ValidateDrive[instance.drive] THEN
```



```
    RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
  [changeCount: driveChangeCount, state: driveState, ready: ready] ←
    DiskChannel.GetDriveAttributes[instance.drive];
  IF ~ready THEN RETURN WITH ERROR PhysicalVolume.Error[notReady];
  IF instance.changeCount ~= driveChangeCount THEN
    RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
  IF driveState = channel THEN GO TO notPilot;
  found ← TRUE;
  pvID ← MarkerPage.Find[
    [drive[instance.drive]] !
    MarkerPage.NotFound => {found ← FALSE; CONTINUE}].physicalID↑;
  IF ~found THEN
    BEGIN -- temporarily bring the drive online
    -- at this point the drive must be in the inactive state
    SELECT DiskChannel.SetDriveState[
    instance.drive, instance.changeCount, pilot] FROM
      invalidDrive => RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
      ok => NULL;
      ENDCASE => ERROR;
    -- the following should return onlineFound as FALSE since MarkerPage.Find failed!
    [pvID, onlineFound] ← PhysicalVolumeOnLineInternal[
      instance.drive !
      PhysicalVolume.Error, PhysicalVolume.CanNotScavenge =>
        BEGIN
        [] ← DiskChannel.SetDriveState[instance.drive, 0, inactive];
        GO TO notPilot
        END];
    IF onlineFound THEN ERROR;
    END;
  [] ← PhysicalRootPageAccessInternal[pvID, CopyLabel];
  IF ~found THEN -- undo the temporary online that we did above
    BEGIN
    PhysicalVolumeOffLineInternal[instance.drive];
    [] ← DiskChannel.SetDriveState[instance.drive, 0, inactive];
    END;
  RETURN[pvID, isPilot];
  EXITS notPilot => RETURN[PhysicalVolume.nullID, notPilot]
  END;

GetNext: PUBLIC PROCEDURE [pvID: PhysicalVolume.ID]
  RETURNS [PhysicalVolume.ID] =
  BEGIN
  RETURN[
    MarkerPage.GetNextPhysicalVolume[
      pvID ! MarkerPage.NotFound => GO TO notFound].nextPvID];
  EXITS notFound => ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  END;

GetNextBadPage: PUBLIC PROCEDURE [
  pvID: PhysicalVolume.ID, thisBadPageNumber: PhysicalVolume.PageNumber]
  RETURNS [nextBadPageNumber: PhysicalVolume.PageNumber] =
  BEGIN
  found: BOOLEAN ← thisBadPageNumber = PhysicalVolume.nullBadPage;
  Proc: PROCEDURE [p: PvHandle] =
    BEGIN
    i: CARDINAL; -- ASSUME < = MAX[CARDINAL] bad pages
    FOR i IN [0..Inline.LowHalf[p.badPageCount]) DO
      IF found THEN {nextBadPageNumber ← p.badPageList[i]; RETURN};
      found ← p.badPageList[i] = thisBadPageNumber;
```

```
    ENDLOOP;
  nextBadPageNumber ← PhysicalVolume.nullBadPage;
  END;
PhysicalRootPageAccess[pvID, Proc];
END;

GetNextDrive: PUBLIC PROCEDURE [type: Device.Type, index: CARDINAL]
  RETURNS [nextType: Device.Type, nextIndex: CARDINAL] =
BEGIN
drive: DiskChannel.Drive ← GetDrive[type, index];
IF drive = DiskChannel.nullDrive AND
  ~(type = Device.nullType AND index = PhysicalVolume.nullDeviceIndex) THEN
  ERROR PhysicalVolume.Error[noSuchDrive];
drive ← DiskChannel.GetNextDrive[drive];
IF drive = DiskChannel.nullDrive THEN
  RETURN[Device.nullType, PhysicalVolume.nullDeviceIndex];
[deviceType: nextType, deviceOrdinal: nextIndex] ←
  DiskChannel.GetDriveAttributes[drive];
END;

GetNextLogicalVolume: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, lvID: Volume.ID] RETURNS [Volume.ID] =
BEGIN
found: BOOLEAN;
newLVID: Volume.ID;
svCount: CARDINAL;
[found, svCount] ← GetPhysicalVolumeAttributes[pvID, NIL];
IF ~found THEN RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
IF svCount = 0 THEN RETURN[Volume.nullID];
IF lvID = Volume.nullID THEN RETURN[GetSubVolumeAttributes[pvID, 0].lvID];
FOR i: CARDINAL IN [0..svCount) DO
  IF lvID = GetSubVolumeAttributes[pvID, i].lvID THEN
    -- handle special case of a volume with multiple pieces on one phys. vol.
    FOR j: CARDINAL IN [i + 1..svCount) DO
      IF lvID ~= (newLVID ← GetSubVolumeAttributes[pvID, j].lvID) THEN
        RETURN[newLVID];
      REPEAT FINISHED => RETURN[Volume.nullID]
      ENDLOOP;
  ENDLOOP;
RETURN WITH ERROR PhysicalVolume.Error[noSuchLogicalVolume]
END;

InterpretHandle: PUBLIC PROCEDURE [instance: Handle]
  RETURNS [type: Device.Type, index: CARDINAL] =
BEGIN
changeCount: CARDINAL;
IF ~ValidateDrive[instance.drive] THEN
  ERROR PhysicalVolume.Error[invalidHandle];
[deviceType: type, deviceOrdinal: index, changeCount: changeCount] ←
  DiskChannel.GetDriveAttributes[instance.drive];
IF instance.changeCount ~= changeCount THEN
  ERROR PhysicalVolume.Error[invalidHandle]
END;

IsReady: PUBLIC ENTRY PROCEDURE [instance: Handle] RETURNS [ready: BOOLEAN] =
BEGIN
changeCount: CARDINAL;
IF ~ValidateDrive[instance.drive] THEN
```



```
  RETURN WITH ERROR PhysicalVolume.Error[invalidHandle];
[ready: ready, changeCount: changeCount] ← DiskChannel.GetDriveAttributes[
  instance.drive];
IF changeCount = instance.changeCount THEN RETURN[ready]
ELSE RETURN WITH ERROR PhysicalVolume.Error[invalidHandle]
END;

MarkPageBad: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, badPage: PhysicalVolume.PageNumber] =
BEGIN
error: BOOLEAN ← FALSE;
Proc: PROCEDURE [p: PvHandle] =
  BEGIN -- ASSUME < = MAX[CARDINAL] bad pages
  i: CARDINAL;
  tmp: PhysicalVolume.PageNumber;
  FOR i IN [0..Inline.LowHalf[p.badPageCount]) DO
    IF p.badPageList[i] = badPage THEN RETURN; ENDLOOP;
  IF error ← (p.badPageCount > = p.maxBadPages) THEN RETURN;
  FOR i IN [0..Inline.LowHalf[p.badPageCount]) DO
    SELECT p.badPageList[i] FROM
      > badPage =>
        BEGIN
        tmp ← p.badPageList[i];
        p.badPageList[i] ← badPage;
        badPage ← tmp;
        END;
      < badPage => LOOP;
      ENDCASE => ERROR;
    ENDLOOP;
  p.badPageList[Inline.LowHalf[p.badPageCount]] ← badPage;
  p.badPageCount ← p.badPageCount + 1;
  END;
IF ~PhysicalRootPageAccessInternal[pvID, Proc, readWrite] THEN
  RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
IF error THEN RETURN WITH ERROR PhysicalVolume.Error[badSpotTableFull];
END;

Offline: PUBLIC ENTRY PROCEDURE [pvID: PhysicalVolume.ID] =
BEGIN
localError: PhysicalVolume.ErrorType;
BEGIN
drive: DiskChannel.Drive = GetPVDrive[pvID];
IF drive = DiskChannel.nullDrive THEN
  RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
PhysicalVolumeOffLineInternal[
  drive ! PhysicalVolume.Error => {localError ← error; GO TO OfflineError}];
IF DiskChannel.SetDriveState[drive, 0, inactive] ~= ok THEN
  ERROR PhysicalVolumeImplError[impossibleDriveStateChangeFailure];
EXITS OfflineError => RETURN WITH ERROR PhysicalVolume.Error[localError];
END;
END;

-- The following are exported to Special Volume

SubVolumeUnknown: PUBLIC ERROR = CODE;
GetNextSubVolume: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, this: SpecialVolume.SubVolume]
  RETURNS [next: SpecialVolume.SubVolume] =
```

```
  BEGIN
  error: BOOLEAN ← FALSE;
  Proc: PROCEDURE [p: PvHandle] =
    BEGIN
    IF this = SpecialVolume.nullSubVolume THEN
      IF p.subVolumeCount = 0 THEN {next ← SpecialVolume.nullSubVolume; RETURN}
      ELSE
        BEGIN OPEN sv: p.subVolumes[0];
        next ← [sv.lvID, sv.nPages, sv.lvPage, sv.pvPage];
        RETURN;
        END;
    FOR i: [0..PhysicalVolumeFormat.maxSubVols) IN [0..p.subVolumeCount) DO
      OPEN sv: p.subVolumes[i];
      IF sv.lvID = this.lvID AND sv.nPages = this.subVolumeSize AND sv.lvPage =
        this.firstLVPageNumber AND sv.pvPage = this.firstPVPageNumber THEN
        IF i + 1 >= p.subVolumeCount -- Zero orgin count
          THEN {next ← SpecialVolume.nullSubVolume; RETURN}
        ELSE
          BEGIN OPEN sv: p.subVolumes[i + 1];
          next ← [sv.lvID, sv.nPages, sv.lvPage, sv.pvPage];
          RETURN;
          END;
      ENDLOOP;
    error ← TRUE;
    END;
  IF ~PhysicalRootPageAccessInternal[pvID, Proc] THEN
    RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  IF error THEN RETURN WITH ERROR SpecialVolume.SubVolumeUnknown;
  END;

GetPhysicalVolumeBootFiles: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, pBootFiles: LONG POINTER TO Boot.PVBootFiles] =
  BEGIN
  Copy: PROCEDURE [p: PvHandle] = {pBootFiles↑ ← p.bootingInfo};
  IF ~PhysicalRootPageAccessInternal[pvID, Copy, read] THEN
    RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  END;

SetPhysicalVolumeBootFiles: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, pBootFiles: LONG POINTER TO Boot.PVBootFiles] =
  BEGIN
  Copy: PROCEDURE [p: PvHandle] =
    BEGIN
    p.bootingInfo ← pBootFiles↑;
    SimpleSpace.ForceOut[physicalRootPageBuffer];
    MarkerPage.UpdatePhysicalMarkerPages[p];
    END;
  IF ~PhysicalRootPageAccessInternal[pvID, Copy, readWrite] THEN
    RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  END;

-- The following are exported to VolumeImplInterface

AccessPhysicalVolumeRootPage: PUBLIC PROCEDURE [
  id: VolumeImplInterface.BarePvID,
  proc: PROCEDURE [PhysicalVolumeFormat.Handle],
  access: VolumeImplInterface.readOrWrite] = {
  PhysicalRootPageAccess[id, proc, access]};
```



```
-- The following are the internal procedures of this module

DriveSize: PROCEDURE [pvID: PhysicalVolume.ID]
  RETURNS [DiskChannel.DiskPageCount] =
  BEGIN
  RETURN[
    DiskChannel.GetDriveAttributes[
      MarkerPage.Find[[physicalID[pvID]]].drive].nPages];
  END;

CheckPhysicalRootLabel: PROCEDURE [
  label: POINTER TO PilotDisk.Label,
  pvID: POINTER TO VolumeImplInterface.BarePvID] RETURNS [BOOLEAN] =
  BEGIN OPEN label;
  pvID↑ ← fileID; -- could also check pad2...
  RETURN[
    fileID # File.nullCapability.fID AND PilotDisk.GetLabelFilePage[label] =
      pvRootPage AND ~immutable AND ~temporary AND ~zeroSize AND pad1 = 0 AND
      PilotDisk.GetLabelType[label] = PilotFileTypes.tPhysicalVolumeRootPage];
  END;

GetDrive: PROCEDURE [type: Device.Type, index: CARDINAL]
  RETURNS [drive: DiskChannel.Drive] =
  BEGIN
  driveType: Device.Type;
  driveIndex: CARDINAL;
  FOR drive ← DiskChannel.GetNextDrive[DiskChannel.nullDrive],
    DiskChannel.GetNextDrive[drive] UNTIL drive = DiskChannel.nullDrive DO
    [driveType, , driveIndex, ] ← DiskChannel.GetDriveAttributes[drive];
    IF driveType = type AND driveIndex = index THEN RETURN
    ENDLOOP;
  RETURN[DiskChannel.nullDrive];
  END;

GetPhysicalVolumeAttributes: INTERNAL PROCEDURE [
  pvID: PhysicalVolume.ID, name: STRING]
  RETURNS [found: BOOLEAN, subvolumeCount: CARDINAL] =
  BEGIN
  Proc: PROC [p: PvHandle] =
    BEGIN
    i: CARDINAL;
    IF name # NIL THEN
      BEGIN
      name.length ← MIN[name.maxlength, p.labelLength, LENGTH[p.label]];
      FOR i IN [0..name.length) DO name[i] ← p.label[i]; ENDLOOP;
      END;
    subvolumeCount ← p.subVolumeCount;
    END;
  found ← PhysicalRootPageAccessInternal[pvID, Proc]
  END;

GetPVDrive: PROCEDURE [pvID: PhysicalVolume.ID]
  RETURNS [drive: DiskChannel.Drive] =
  BEGIN
  RETURN[
    MarkerPage.Find[
      [physicalID[pvID]] ! MarkerPage.NotFound => GO TO notFound].drive];
  EXITS notFound => RETURN[DiskChannel.nullDrive];
  END;
```

```
GetSubVolumeAttributes: INTERNAL PROCEDURE [
  pvID: PhysicalVolume.ID, subvolumeNumber: CARDINAL]
  RETURNS [
    found: BOOLEAN, lvID: Volume.ID, lvSize: Volume.PageCount,
    subVolumeSize: Volume.PageCount,
    firstLVPageNumber, firstPVPageNumber: PhysicalVolume.PageNumber] =
  BEGIN
  error: BOOLEAN ← TRUE;
  Proc: PROC [p: PvHandle] =
    BEGIN
    IF subvolumeNumber < p.subVolumeCount THEN
      BEGIN OPEN sv: p.subVolumes[subvolumeNumber];
      error ← FALSE;
      lvID ← sv.lvID;
      lvSize ← sv.lvSize;
      subVolumeSize ← sv.nPages;
      firstLVPageNumber ← sv.lvPage;
      firstPVPageNumber ← sv.pvPage;
      END;
    END;
  found ← TRUE;
  IF ~PhysicalRootPageAccessInternal[pvID, Proc] THEN
    ERROR PhysicalVolumeImplError[physicalVolumeNotFound];
  IF error THEN ERROR PhysicalVolumeImplError[invalidSubVolumeNumber];
  END;

IsUtilityPilot: PROCEDURE RETURNS [BOOLEAN] = INLINE
  BEGIN RETURN[PilotSwitches.switches.u = down]; END;

PhysicalRootPageAccess: ENTRY PROCEDURE [
  id: VolumeImplInterface.BarePvID, proc: PROCEDURE [PvHandle],
  access: VolumeImplInterface.readOrWrite ← read] =
  BEGIN
  IF ~PhysicalRootPageAccessInternal[id, proc, access] THEN
    RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
  END;

PhysicalRootPageAccessInternal: INTERNAL PROCEDURE [
  id: VolumeImplInterface.BarePvID, proc: PROCEDURE [PvHandle],
  access: VolumeImplInterface.readOrWrite ← read] RETURNS [found: BOOLEAN] =
  BEGIN
  IF ~SubVolume.Find[[id], pvRootPage].success THEN RETURN[FALSE];
  PhysicalRootPageMap[id, access];
  proc[physicalVolume];
  PhysicalRootPageUnmap[];
  RETURN[TRUE]
  END;

PhysicalRootPageCheck: INTERNAL PROCEDURE [
  pv: PhysicalVolumeFormat.Handle, id: PhysicalVolume.ID] RETURNS [BOOLEAN] =
  BEGIN
  RETURN[
    pv.seal = PhysicalVolumeFormat.seal AND pv.version =
      PhysicalVolumeFormat.currentVersion AND pv.pvID = id];
  END;

PhysicalRootPageMap: INTERNAL PROCEDURE [
  ID: VolumeImplInterface.BarePvID,
  access: VolumeImplInterface.readOrWrite ← read] =
```



```
  BEGIN
  per: File.Permissions =
    IF access = readWrite THEN FileInternal.maxPermissions ELSE File.read;
  SimpleSpace.Map[physicalRootPageBuffer, [[ID], per], pvRootPage], FALSE];
  END;

PhysicalRootPageUnmap: INTERNAL PROCEDURE = {
  SimpleSpace.Unmap[physicalRootPageBuffer]};

PhysicalVolumeOffLineInternal: INTERNAL PROCEDURE [drive: DiskChannel.Drive] =
  BEGIN
  -- This procedure assumes that this module will not be called by VolumeImpl when this proced
**ure calls VolumeImpl through VolumeImplInterface. If this assumption should ever not be true,
**this procedure must be modified to release the monitor lock before calling into VolumeImpl.
  pMarkerID: POINTER TO READONLY File.ID;
  pvid: VolumeImplInterface.BarePvID;
  pPhysicalID: POINTER TO READONLY PhysicalVolume.ID;
  lv: Volume.ID;
  svH: SubVolume.Handle;


    [, pPhysicalID, pMarkerID] ← MarkerPage.Find[
    [drive[drive]] ! MarkerPage.NotFound => GOTO easy];
  pvid ← LOOPHOLE[pPhysicalID↑];
  VolFileMap.Close[TRUE];
  VolAllocMap.Close[TRUE];
  svH ← NIL;
  WHILE (svH ← SubVolume.GetNext[svH]) ~= NIL DO
    -- this loop ensures that there are no open logical volumes on the volume that is to be offline
**d
    IF DiskChannel.GetAttributes[svH.channel].drive ~= drive OR (lv ← svH.lvID)
      = Volume.ID[pvid] THEN LOOP;
    IF VolumeImplInterface.FindLogicalVolume[@lv] THEN
      IF VolumeImplInterface.GetLVStatus[lv].open THEN
        ERROR PhysicalVolume.Error[containsOpenVolumes];
    ENDLOOP;
  svH ← NIL;
  WHILE (svH ← SubVolume.GetNext[svH]) # NIL DO
    -- This loop gets rid Subvolume's knowledge of the logical volumes that use the disappearing
**physical volume as well as flushing the volume from VolumeImpl and the FileCache.
    IF DiskChannel.GetAttributes[svH.channel].drive ~= drive OR svH.lvID =
      Volume.ID[pvid] THEN LOOP;
    VolumeImplInterface.SubvolumeOffline[
      svH.lvID, svH.lvPage = FIRST[VolumeInternal.PageNumber]];
    SubVolume.OffLine[svH.lvID, svH.channel];
    ENDLOOP; -- Now it is finally safe to forget about the physical volume
  VolumeImplInterface.PinnedFileFlush[[pvid]];
  MarkerPage.Flush[[drive[drive]]];
  SubVolume.OffLine[[pvid], DiskChannel.nullHandle];
  EXITS easy => RETURN;
  END;

PhysicalVolumeOnLineInternal: INTERNAL PROCEDURE [drive: DiskChannel.Drive]
  RETURNS [id: PhysicalVolume.ID, alreadyOnline: BOOLEAN] =
  BEGIN
  -- This procedure assumes that this module will not be called by VolumeImpl when this proced
**ure calls VolumeImpl through VolumeImplInterface. If this assumption should ever not be true,
```

```
**this procedure must be modified to release the monitor lock before calling into VolumeImpl
  CleanUpDrive: INTERNAL PROCEDURE =
    BEGIN
    PhysicalRootPageUnmap[];
    VolumeImplInterface.PinnedFileFlush[[pvID]];
    MarkerPage.Flush[[drive[drive]]];
    SubVolume.OffLine[[pvID], DiskChannel.nullHandle];
    END;
  found: BOOLEAN ← TRUE;
  dT: Device.Type;
  dO: CARDINAL;
  i: CARDINAL;
  badPageListPage: PhysicalVolumeFormat.PageNumber;
  numPagesForBadPages: PhysicalVolumeFormat.PageNumber;
  label: PilotDisk.Label;
  labelStatus: LabelTransfer.LabelStatus;
  special: CARDINAL ← LAST[CARDINAL];
  pvID: VolumeImplInterface.BarePvID;
  lvID: Volume.ID;
  pvID ← MarkerPage.Find[
    [drive[drive]] !
    MarkerPage.NotFound => {found ← FALSE; CONTINUE}].physicalID↑;
  IF found THEN RETURN[[pvID], TRUE];
  [label, labelStatus] ← LabelTransfer.ReadRootLabel[drive, pvRootPage];
  SELECT labelStatus FROM
    valid => NULL;
    invalid => ERROR PhysicalVolume.CanNotScavenge;
    diskError => ERROR PhysicalVolume.Error[diskReadError];
    ENDCASE => ERROR PhysicalVolumeImplError[impossibleSelectError];
  IF ~CheckPhysicalRootLabel[@label, @pvID] THEN
    ERROR PhysicalVolume.CanNotScavenge;
  RegisterPvInfo[pvID, drive];
  PhysicalRootPageMap[pvID];
  IF ~PhysicalRootPageCheck[physicalVolume, [pvID]] THEN
    BEGIN CleanUpDrive[]; ERROR PhysicalVolume.CanNotScavenge; END;
  -- Now check to see that the labels on the bad page list page(s) are correct
  numPagesForBadPages ←
    (physicalVolume.maxBadPages + Environment.wordsPerPage -
      1)/Environment.wordsPerPage;
  -- The following statement blows up the compiler
  -- FOR badPageListPage IN [pvRootPage + 1..pvRootPage + numPagesForBadPages] DO
  FOR badPageListPage ← pvRootPage + 1, badPageListPage + 1 WHILE
    badPageListPage <= pvRootPage + numPagesForBadPages DO
    [label, labelStatus] ← LabelTransfer.ReadRootLabel[drive, badPageListPage];
    SELECT labelStatus FROM
      valid => NULL;
      invalid => {CleanUpDrive[]; ERROR PhysicalVolume.CanNotScavenge};
      diskError => {CleanUpDrive[]; ERROR PhysicalVolume.Error[diskReadError]};
      ENDCASE => ERROR PhysicalVolumeImplError[impossibleSelectError];
    ENDLOOP;
  RegisterSubvolumeMarker[physicalVolume];
  -- if we were booted from this volume, specials
  -- First make all of the subvolumes accessible to the VM machinery and discover if this is the b
**ootload volume.
  FOR i IN [0..physicalVolume.subVolumeCount) DO
    OPEN sv: physicalVolume.subVolumes[i];
    VolumeImplInterface.RegisterLogicalSubvolume[sv, [pvID]];
    lvID ← sv.lvID;
```



```
    [deviceType: dT, deviceOrdinal: dO] ← DiskChannel.GetDriveAttributes[drive];
    IF debugSearchState = tooSoonToLook AND bootFile.deviceType = dT AND
      bootFile.deviceOrdinal = dO AND DiskChannel.GetPageNumber[
      drive, LOOPHOLE[bootFile.diskFileID.da]] IN
      [sv.pvPage..sv.pvPage + sv.nPages) THEN special ← i;
    ENDLOOP;
  IF special ~= LAST[CARDINAL] THEN -- Bring the system volume online
    BEGIN OPEN sv: physicalVolume.subVolumes[special];
    VolumeImplInterface.SubvolumeOnline[
      sv.lvID, sv.lvPage = FIRST[VolumeInternal.PageNumber]];
    VolumeImplInterface.CheckLogicalVolume[sv.lvID];
    END;
  FOR i IN [0..physicalVolume.subVolumeCount) DO
    -- Bring all other volumes online
    IF i ~= special THEN
      BEGIN OPEN sv: physicalVolume.subVolumes[i];
      VolumeImplInterface.SubvolumeOnline[
        sv.lvID, sv.lvPage = FIRST[VolumeInternal.PageNumber]];
      VolumeImplInterface.CheckLogicalVolume[sv.lvID];
      END;
    ENDLOOP;
  PhysicalRootPageUnmap[];
  RETURN[[pvID], FALSE]
  END;

RegisterPvInfo: PUBLIC PROCEDURE [
  pvID: VolumeImplInterface.BarePvID, drive: DiskChannel.Drive] =
  BEGIN
  driveSize: LONG CARDINAL ← DiskChannel.GetDriveAttributes[drive].nPages;
  DiskChannel.SetDriveTag[drive, PhysicalVolumeFormat.IDCheckSum[[pvID]]];
  IF ~SubVolume.Find[[pvID], pvRootPage].success THEN
    -- Create a subvolume covering the drive so we can find PVDescriptor and Marker Pages
    SubVolume.OnLine[
      [[pvID], driveSize, pvRootPage, pvRootPage, driveSize],
      DiskChannel.Create[drive, SubVolume.completion]];
  -- Create a VFile for just the root page
  VolumeImplInterface.VFileEnter[
    [pvID], [pvID], pvRootPage, pvRootPage + rootPagePages,
    PilotFileTypes.tPhysicalVolumeRootPage];
  MarkerPage.Enter[drive, [pvID]];
  END;
-- Read the ID off the disk, or if there are no subvolumes (yet) generate one. Someday, the id sh
**ould be in the pvHandle??

RegisterSubvolumeMarker: PUBLIC PROCEDURE [pv: PvHandle] =
  BEGIN
  drive: DiskChannel.Drive = MarkerPage.Find[[physicalID[pv.pvID]]].drive;
  driveSize: LONG CARDINAL = DiskChannel.GetDriveAttributes[drive].nPages;
  id: File.ID;
  page: PhysicalVolumeFormat.PageNumber;
  IF pv.subVolumeCount # 0 THEN
    BEGIN
    page ← pv.subVolumes[0].pvPage + pv.subVolumes[0].nPages;
    -- We should really check the label here...
    id ← LabelTransfer.ReadRootLabel[drive, page].label.fileID;
    END
  ELSE id ← [System.GetUniversalID[]];
  MarkerPage.EnterMarkerID[[drive[drive]], id];
```

```
    -- The marker pages cover the disk
    VolumeImplInterface.VFileEnter[
      LOOPHOLE[pv.pvID], id, 0, driveSize, PilotFileTypes.tSubVolumeMarkerPage];
    END;

  ValidateDrive: PROCEDURE [drive: DiskChannel.Drive] RETURNS [found: BOOLEAN] =
    BEGIN
    FOR existingDrive: DiskChannel.Drive ← DiskChannel.GetNextDrive[
      DiskChannel.nullDrive], DiskChannel.GetNextDrive[existingDrive] UNTIL
      existingDrive = DiskChannel.nullDrive DO
      IF existingDrive = drive THEN RETURN[TRUE]; ENDLOOP;
    RETURN[FALSE];
    END; -- Initialization

  InitDisks: ENTRY PROCEDURE =
    BEGIN OPEN DiskChannel;
    dT: Device.Type;
    dO: CARDINAL;
    drive, systemDrive: Drive ← nullDrive;
    volumeID: Volume.ID ← Volume.nullID;
    WHILE (systemDrive ← GetNextDrive[systemDrive]) # nullDrive DO
      [deviceType: dT, deviceOrdinal: dO] ← GetDriveAttributes[systemDrive];
      IF dT = bootFile.deviceType AND dO = bootFile.deviceOrdinal THEN EXIT;
      ENDLOOP;
    IF systemDrive = nullDrive THEN
      Runtime.CallDebugger["System Drive not found"L];
    IF DiskChannel.SetDriveState[
      systemDrive, DiskChannel.GetDriveAttributes[systemDrive].changeCount, pilot]
      ~= ok THEN
      Runtime.CallDebugger["Could not set SystemDrive state to Pilot"L];
    [] ← PhysicalVolumeOnLineInternal[
      systemDrive !
      PhysicalVolume.Error =>
        IF error = diskReadError THEN
          BEGIN ProcessorFace.SetMP[PilotMP.cDriveNotReady]; RETRY; END
        ELSE CONTINUE];
    IF Volume.systemID = Volume.nullID THEN
      Runtime.CallDebugger["No Logical Volumes on System Drive"L];
    IF PilotSwitches.switches.z = down THEN
      WHILE (drive ← DiskChannel.GetNextDrive[drive]) # nullDrive DO
        [deviceType: dT, deviceOrdinal: dO] ← GetDriveAttributes[drive];
        IF drive # systemDrive THEN
          BEGIN
          IF DiskChannel.SetDriveState[
            drive, DiskChannel.GetDriveAttributes[drive].changeCount, pilot] ~= ok
            THEN
            Runtime.CallDebugger["Could not set SystemDrive state to Pilot"L];
          [] ← PhysicalVolumeOnLineInternal[
            drive !
            PhysicalVolume.Error =>
              IF error = diskReadError THEN
                BEGIN ProcessorFace.SetMP[PilotMP.cDriveNotReady]; RETRY; END
              ELSE CONTINUE];
          END
        ENDLOOP;
    ProcessorFace.SetMP[PilotMP.cDeleteTemps];
    VolumeImplInterface.OpenInitialVolumes[];
    END;
```



```
    START FMPrograms.ScavengeImpl;
    START FMPrograms.VolumeImpl[
      bootFile, pLVBootFiles, @debuggerDeviceType, @debuggerDeviceOrdinal];
    debuggerDeviceType ← Device.nullType;
    -- If we are utilityPilot, the person who installed us smashed in our debugger pointers
    IF IsUtilityPilot[] THEN debugClass ← normal ELSE InitDisks[];
    debugSearchState ← done; -- We will never ever set them from here on....
    RETURN;
    END.
  LOG
  Time: June 5, 1980 6:19 PM By: Luniewski Action: Created file.
  Time: June 11, 1980 10:17 AM By: Luniewski Action: Added export of PhysicalVolume.Handle a
  **nd updated many procedures for the new form of a Handle.
  Time: June 24, 1980 5:29 PM By: McJones Action: New root page format; OISProcessorFace = P
  **rocessorFace.
  Time: July 19, 1980 2:14 PM By: Forrest for Luniewski Action: Fix to CreatePhysicalVolume to h
  **andle RegisterPVInfo error (drive not asserted as Pilot volume); fix to GetNextDrive. Take out ex
  **port of PhysicalVolume.nullID (now Constant).
  Time: July 29, 1980 10:55 AM By: Luniewski Action: Fixes for new Volume/PhysicalVolume/Spe
  **cialVolume.
  Time: September 3, 1980 4:49 PM By: Luniewski Action: Catch some errors raised from Internal
  **procedures and resignal.
  Time: September 18, 1980 5:49 PM. By: Luniewski Action: Changes for new physical volume roo
  **t page format. Changed uses of FilePageLabel to uses of PilotDisk.
  Time: October 3, 1980 4:26 PM By: Luniewski Action: Bug fixes: handle erroneous arguments c
  **orrectly
  Time: October 7, 1980 3:01 PM By: Luniewski Action: Bug fixes: initialize variable in MarkPageB
  **ad
```

```
-- ScavengeImpl.mesa (last edited by: Forrest on: October 9, 1980 4:41 PM)
-- This is a cut at the interim Scavenger

DIRECTORY
  DiskChannel USING [GetAttributes],
  Environment USING [bitsPerWord, wordsPerPage],
  File USING [
    Capability, Create, delete, Delete, DeleteImmutable, Error, GetAttributes, ID,
    MakeImmutable, MakePermanent, nullCapability, nullID, PageCount, PageNumber,
    read, SetSize, Type, Unknown],
  FileInternal USING [Descriptor, FilePtr, PageGroup],
  FMPrograms USING [],
  Inline USING [LowHalf],
  KernelFile USING [GetNextFile, GetRootFile],
  LabelTransfer USING [ReadLabel, WriteLabels, VerifyLabels],
  LogicalVolume USING [
    BeginScavenging, EndScavenging, Free, Handle, PageNumber, PutRootFile,
    rootPageNumber, Vam],
  MarkerPage USING [Find],
  PhysicalVolumeFormat USING [Handle, maxBadPages, PageNumber, rootPageNumber],
  PilotDisk USING [GetLabelFilePage, Label],
  PilotFileTypes USING [
    PilotVFileType, tFreePage, tScavengerLog, tScavengerLogOtherVolume,
    tVolumeAllocationMap, tVolumeFileMap],
  Scavenger USING [Error, ErrorType, FileEntry, Header],
  SimpleSpace USING [Create, Handle, LongPointer, Map, Page, Unmap],
  Space USING [Handle, PageNumber, WindowOrigin],
  SubVolume USING [Find, Handle],
  System USING [GetGreenwichMeanTime],
  SystemInternal USING [Unimplemented],
  Utilities USING [LongPointerFromPage],
  VolAllocMap USING [AccessVAM, Close],
  VolFileMap USING [Close, InitMap, InsertPageGroup],
  Volume USING [
    Close, GetNext, ID, InsufficientSpace, NotOpen, nullID, PageCount, Unknown],
  VolumeInternal USING [PageNumber];

ScavengeImpl: MONITOR
  IMPORTS
    DiskChannel, File, Inline, KernelFile, LabelTransfer, LogicalVolume,
    MarkerPage, PilotDisk, Scavenger, SimpleSpace, System, SystemInternal,
    SubVolume, Utilities, VolAllocMap, VolFileMap, Volume
  EXPORTS LogicalVolume, FMPrograms, Scavenger
  SHARES File =
  BEGIN

  bitsPerPage: CARDINAL = Environment.bitsPerWord*Environment.wordsPerPage;
  BadPageList: TYPE = RECORD [
    p: POINTER TO PageNumber,
    a: ARRAY [0..PhysicalVolumeFormat.maxBadPages] OF PageNumber];
  OrphanHandle: PUBLIC TYPE = VolumeInternal.PageNumber;
  PageNumber: TYPE = LogicalVolume.PageNumber;

  ImpossibleScavengerError: PRIVATE ERROR [
    {cantFindSubvolume, noRoomForVam, pageAlreadyBusy, labelChanged}] = CODE;
  Error: PUBLIC ERROR [Scavenger.ErrorType] = CODE; -- to Scavenge

  -- Unimplemented procs
```



```
  DeleteOrphanPage: PUBLIC PROC [volume: Volume.ID, id: OrphanHandle] = {
    ERROR SystemInternal.Unimplemented};

  ReadBadPage: PUBLIC PROC [
    file: File.ID, page: File.PageNumber, destination: Space.PageNumber] = {
    ERROR SystemInternal.Unimplemented};

  ReadOrphanPage: PUBLIC PROC [
    volume: Volume.ID, id: OrphanHandle, destination: Space.PageNumber]
    RETURNS [
      file: File.ID, type: File.Type, pageNumber: File.PageNumber,
      readErrors: BOOLEAN] = {ERROR SystemInternal.Unimplemented};

  RewritePage: PUBLIC PROC [
    file: File.ID, page: File.PageNumber, source: Space.PageNumber] = {
    ERROR SystemInternal.Unimplemented};

  DeleteLog: PUBLIC PROC [volume: Volume.ID] =
    BEGIN
    logFile: File.Capability ← GetLog[volume];
    IF logFile = File.nullCapability THEN RETURN;
    DeleteLogFile[logFile];
    logFile ← File.nullCapability;
    LogicalVolume.PutRootFile[@volume, PilotFileTypes.tScavengerLog, @logFile];
    END;

  DeleteLogFile: PROC [logFile: File.Capability] =
    BEGIN
    immutable: BOOLEAN;
    volume: Volume.ID ← Volume.nullID;
    [immutable: immutable] ← File.GetAttributes[
      logFile ! File.Unknown => GOTO return];
    IF ~immutable THEN File.Delete[logFile]
    ELSE
      WHILE (volume ← Volume.GetNext[volume]) # Volume.nullID DO
        File.DeleteImmutable[
          logFile, volume !
          Volume.Unknown, Volume.NotOpen, File.Unknown, File.Error => LOOP];
      ENDLOOP
    EXITS return => NULL
    END;

  GetLog: PUBLIC PROC [volume: Volume.ID] RETURNS [logFile: File.Capability] = {
    RETURN[KernelFile.GetRootFile[PilotFileTypes.tScavengerLog, volume]]};

  Scavenge: PUBLIC PROC [volume, logDestination: Volume.ID, repair: BOOLEAN]
    RETURNS [logFile: File.Capability] =
    BEGIN
    oldLog: File.Capability = GetLog[volume];
    context: ScavengeContext;
    IF ~repair THEN ERROR SystemInternal.Unimplemented;
    LogicalVolume.BeginScavenging[@volume];
    DeleteLog[volume];
    context ← OpenScavengeContext[volume];
    IF ~CreateLogFile[logDestination, context] THEN
      BEGIN
      CloseScavengeContext[context];
      LogicalVolume.EndScavenging[@volume];
      ERROR Scavenger.Error[cannotWriteLog]
```

```
    END;
  context.logHeader.repaired ← TRUE;
  OpenLogFile[context];
  [] ← PutWords[context, @context.logHeader, SIZE[Scavenger.Header]];
  EnumerateFiles[context];
  PutHeader[context];
  CloseLogFile[context];
  logFile ← [context.logFile.fID, File.delete + File.read];
  LogicalVolume.PutRootFile[@volume, PilotFileTypes.tScavengerLog, @logFile];
  IF volume = logDestination THEN {
    File.MakePermanent[logFile]; File.MakeImmutable[logFile]};
  CloseScavengeContext[context];
  -- The following two statements are a hack to get around the following problem:
  -- When we created the log file above and deleted the old log file, we caused
  -- FileImpl to fill its tmpsFile cache. The closes are necessary to clear those
  -- caches. It is known to be safe to do because VolumeImpl 1) permits a closed
  -- volume to be closed again without error, and 2) is only called after FileImpl
  -- has cleared its caches (i.e., error checking is done after FileImpl has
  -- done its thing).
  Volume.Close[volume ! Volume.Unknown => CONTINUE];
  LogicalVolume.EndScavenging[@volume];
  END;

EnumerateFiles: PROC [context: ScavengeContext] =
  BEGIN
  file: File.Capability ← File.nullCapability;
  context.logHeader.incomplete ← TRUE;
  DO
    fileEntry: Scavenger.FileEntry;
    file ← KernelFile.GetNextFile[context.volume, file];
    IF file.fID = File.nullID THEN EXIT;
    fileEntry.file ← file.fID;
    fileEntry.numberOfProblems ← 0;
    IF ~PutWords[context, @fileEntry, SIZE[Scavenger.FileEntry]] THEN RETURN;
    context.logHeader.numberOfFiles ← context.logHeader.numberOfFiles + 1;
    ENDLOOP;
  context.logHeader.incomplete ← FALSE;
  END;

-- The following write the log/create the log file
-- This is here when the day comes that multiple scavenges can be going on at once.
-- This will occur when VolumeImpl has been modified and IndexOfContexts is changed
-- appropriately.

ScavengeContext: TYPE = POINTER TO ScavengeRecord;
ScavengeRecord: TYPE = RECORD [
  occupied: BOOLEAN,
  logHeader: Scavenger.Header,
  volume: Volume.ID,
  buffer: Space.Handle, -- Initialize only
  bufferPointer: LONG POINTER, -- Initialize only
  logFile: File.Capability,
  nextWord: CARDINAL, -- words
  fileLengthPages: File.PageCount];

IndexOfContexts: TYPE = [0..1);
Contexts: ARRAY IndexOfContexts OF ScavengeRecord;
freeContext: CONDITION;
```



```
CloseLogFile: PROC [context: ScavengeContext] = {
  SimpleSpace.Unmap[context.buffer]};

CloseScavengeContext: ENTRY PROC [context: ScavengeContext] = {
  context.occupied ← FALSE; NOTIFY freeContext};

CreateLogFile: PROC [logDestination: Volume.ID, context: ScavengeContext]
  RETURNS [success: BOOLEAN] =
  BEGIN
  context.logFile ← File.Create[
    logDestination, 1,
    IF logDestination = context.volume THEN PilotFileTypes.tScavengerLog
    ELSE PilotFileTypes.tScavengerLogOtherVolume ! ANY => GO TO Failed];
  -- well, any does suggest we can't create the log.....
  context.fileLengthPages ← 1;
  RETURN[TRUE]
  EXITS Failed => RETURN[FALSE]
  END;

OpenLogFile: PROC [context: ScavengeContext] =
  BEGIN
  SimpleSpace.Map[context.buffer, [context.logFile, 0], FALSE];
  context.nextWord ← 0
  END;

OpenScavengeContext: ENTRY PROC [volume: Volume.ID]
  RETURNS [context: ScavengeContext] =
  BEGIN
  DO
    -- Until a free context is found
    FOR i: IndexOfContexts IN IndexOfContexts DO
      IF ~Contexts[i].occupied THEN
        BEGIN
        context ← @Contexts[i];
        context.occupied ← TRUE;
        context.logHeader ←
          [volume: volume, date: System.GetGreenwichMeanTime[],
            incomplete: TRUE, repaired: FALSE, numberOfFiles: 0];
        context.volume ← volume;
        RETURN;
        END;
      ENDLOOP;
    WAIT freeContext;
    ENDLOOP
  END;

-- stream interface to ScavengeLog

PutHeader: PROC [context: ScavengeContext] =
  BEGIN
  SimpleSpace.Unmap[context.buffer];
  --(Re)--
  OpenLogFile[context];
  LOOPHOLE[context.bufferPointer, LONG POINTER TO Scavenger.Header]↑ ←
    context.logHeader;
  END;

PutWords: PROC [context: ScavengeContext, words: LONG POINTER, count: CARDINAL]
  RETURNS [success: BOOLEAN] =
```

```
  BEGIN
  THROUGH [0..count) DO
    IF context.nextWord = Environment.wordsPerPage THEN
      BEGIN
      File.SetSize[
        context.logFile, context.fileLengthPages + 1 !
        Volume.InsufficientSpace => GOTO noRoom];
      SimpleSpace.Unmap[context.buffer];
      SimpleSpace.Map[
        context.buffer, [context.logFile, context.fileLengthPages], FALSE];
      context.nextWord ← 0;
      context.fileLengthPages ← context.fileLengthPages + 1;
      END;
    (context.bufferPointer + context.nextWord)↑ ← words↑;
    context.nextWord ← context.nextWord + 1;
    words ← words + 1;
    ENDLOOP;
  RETURN[TRUE]
  EXITS noRoom => RETURN[FALSE]
  END;

-- The following are internal procedures of this module used by the actual scavenger

AdvanceBadPagePointer: PROC [l: POINTER TO BadPageList] = INLINE {
  l.p ← l.p + SIZE[PageNumber]};

CurrentBadPage: PROC [l: POINTER TO BadPageList] RETURNS [PageNumber] = INLINE {
  RETURN[l.p↑]};

GetBadList: PROC [
  svH: SubVolume.Handle, space: SimpleSpace.Handle, l: POINTER TO BadPageList] =
  BEGIN
  RootWindow: PROC RETURNS [Space.WindowOrigin] = INLINE
    BEGIN
    RETURN[
      [[LOOPHOLE[MarkerPage.Find[
          [drive[DiskChannel.GetAttributes[svH.channel]]]].physicalID↑],
          File.read], PhysicalVolumeFormat.rootPageNumber]];
    END;
  lvPage: PageNumber;
  p: POINTER TO PageNumber;
  pv: PhysicalVolumeFormat.Handle = Utilities.LongPointerFromPage[
    SimpleSpace.Page[space]];
  l.p ← @l.a[0];
  SimpleSpace.Map[space, RootWindow[], FALSE];
  FOR i: CARDINAL IN [0..Inline.LowHalf[pv.badPageCount]) DO
    -- ASSUMEs no more than LAST[CARDINAL] bad pages
    IF pv.badPageList[i] NOT IN [svH.pvPage..svH.pvPage + svH.nPages) THEN LOOP;
    lvPage ← (pv.badPageList[i] - svH.pvPage) + svH.lvPage;
    p ← l.p;
    l.p ← l.p + SIZE[PageNumber];
    -- Pages are susposed to be ordered; we'll make sure
    WHILE p # @l.a[0] AND (p - SIZE[PageNumber])↑ > lvPage DO
      p↑ ← (p - SIZE[PageNumber])↑; p ← p - SIZE[PageNumber]; ENDLOOP;
    p↑ ← lvPage;
    ENDLOOP;
  SimpleSpace.Unmap[space];
  l.p↑ ← LAST[LONG CARDINAL];
```



```
  l.p ← @l.a[0];
  END;

MakeFreeDescriptor: PROC [vol: LogicalVolume.Handle]
  RETURNS [FileInternal.Descriptor] = INLINE
  BEGIN
  RETURN[
    [LogicalVolume.Free[vol], vol.vID, local[
      FALSE, FALSE, vol.volumeSize, PilotFileTypes.tFreePage]]];
  END;

ScavengeVolume: PUBLIC PROC [
  vol: LogicalVolume.Handle, space: SimpleSpace.Handle, erase: BOOLEAN] =
  BEGIN
  IF vol.type = nonPilot THEN RETURN; -- Scavenging non-Pilot Volumes is Easy
  vol.changing ← TRUE;
  IF erase THEN VamFind[vol, space]; -- FInd n unblemished pages
  Vam[vol, space, erase];
  Vfm[vol, space, erase];
  vol.changing ← FALSE;
  END;

Vam: PROC [
  vol: LogicalVolume.Handle, space: SimpleSpace.Handle, erase: BOOLEAN] =
  BEGIN
  found: BOOLEAN;
  list: BadPageList;
  next: PageNumber;
  page: PageNumber ← 1;
  svEnd: Volume.PageCount;
  svH: SubVolume.Handle;
  vamEnd: Volume.PageCount ← vol.vamStart + VamSize[vol];
  VolAllocMap.Close[TRUE]; -- Flush any context VamImpl has
  VamInit[vol];
  WHILE page < vol.volumeSize DO
    [found, svH] ← SubVolume.Find[vol.vID, page];
    IF ~found THEN ERROR ImpossibleScavengerError[cantFindSubvolume];
    GetBadList[svH, space, @list];
    svEnd ← svH.lvPage + svH.nPages;
    WHILE page < svEnd DO
      IF page = CurrentBadPage[@list] THEN
        BEGIN
        VamMarkBadPage[vol, page];
        AdvanceBadPagePointer[@list];
        page ← page + 1;
        LOOP;
        END;
      IF page IN [vol.vamStart..vamEnd) THEN BEGIN page ← vamEnd; LOOP; END;
      -- Add First lvPage ov next track to MIN
      next ← MIN[svEnd, CurrentBadPage[@list]];
      IF page < vol.vamStart AND next > vol.vamStart THEN next ← vol.vamStart;
      IF erase THEN VamPieceErase[vol, page, next]
      ELSE VamPieceRebuild[vol, page, next];
      page ← next;
      ENDLOOP;
    ENDLOOP;
  END;
```

```
-- Find VamSize pages of good contiguous disk (for now we don't cross subvolume boundaries)

VamFind: PROC [vol: LogicalVolume.Handle, space: SimpleSpace.Handle] =
  BEGIN
  found: BOOLEAN;
  list: BadPageList;
  page: PageNumber ← 1;
  next: PageNumber;
  svEnd: Volume.PageCount;
  svH: SubVolume.Handle;
  vamSize: Volume.PageCount ← VamSize[vol];
  WHILE page < vol.volumeSize DO
    [found, svH] ← SubVolume.Find[vol.vID, page];
    IF ~found THEN ERROR ImpossibleScavengerError[cantFindSubvolume];
    GetBadList[svH, space, @list];
    svEnd ← svH.lvPage + svH.nPages;
    WHILE page < svEnd DO
      IF page = CurrentBadPage[@list] THEN {
        AdvanceBadPagePointer[@list]; page ← page + 1; LOOP};
      next ← MIN[svEnd, CurrentBadPage[@list]];
      IF (next - page) > vamSize THEN {vol.vamStart ← page; RETURN};
      page ← next;
      ENDLOOP;
    ENDLOOP;
  ERROR ImpossibleScavengerError[noRoomForVam];
  END;

VamInit: PROC [vol: LogicalVolume.Handle] =
  BEGIN
  vamEnd: PageNumber ← vol.vamStart + VamSize[vol];
  page: PageNumber;
  LabelTransfer.WriteLabels[
    FileInternal.Descriptor[
    LogicalVolume.Vam[vol], vol.vID, local[
    FALSE, FALSE, vol.volumeSize, PilotFileTypes.tVolumeAllocationMap]],
    FileInternal.PageGroup[vol.vamStart, vol.vamStart, vamEnd]];
  vol.freePageCount ← vol.volumeSize;
  -- Decremented every time a page is marked busy
  vol.lowerBound ← vol.volumeSize;
  -- we will set this when free pages are found.
  [] ← VolAllocMap.AccessVAM[vol, LogicalVolume.rootPageNumber, TRUE, FALSE];
  FOR page ← vol.vamStart, page + 1 WHILE page < vamEnd DO
    [] ← VolAllocMap.AccessVAM[vol, page, TRUE, FALSE]; ENDLOOP;
  END;

VamMarkBadPage: PROC [vol: LogicalVolume.Handle, page: PageNumber] = {
  [] ← VolAllocMap.AccessVAM[vol, page, TRUE, FALSE]};

VamPieceErase: PROC [vol: LogicalVolume.Handle, this, next: PageNumber] =
  BEGIN
  LabelTransfer.WriteLabels[MakeFreeDescriptor[vol], [this, this, next]];
  -- Pages are marked free in vam; no action needed
  IF vol.lowerBound > this THEN vol.lowerBound ← this;
  END;

VamPieceRebuild: PROC [vol: LogicalVolume.Handle, this, next: PageNumber] =
  BEGIN
  page: PageNumber ← this;
```



```
  pageCount: Volume.PageCount;
  FixUpAndIsFree: PROC [p, rem: PageNumber] RETURNS [isFree: BOOLEAN] = INLINE
    BEGIN
    label: PilotDisk.Label ← LabelTransfer.ReadLabel[
      [, vol.vID, local[, , , ]], 0, p];
    filePage: PageNumber = PilotDisk.GetLabelFilePage[@label];
    IF filePage # 0 OR ~label.zeroSize THEN
      [countValid: pageCount] ← LabelTransfer.VerifyLabels[
        file:
        [label.fileID, vol.vID, local[
          label.immutable, label.temporary, filePage + rem, label.type]],
        pageGroup: [filePage, p, filePage + rem],
        expectErrorAfterFirstPage: TRUE]
    ELSE pageCount ← 1;
    IF pageCount = 0 THEN ImpossibleScavengerError[labelChanged];
    IF
      (label.type = PilotFileTypes.tFreePage AND label.fileID #
        LogicalVolume.Free[vol]) OR label.type = PilotFileTypes.tVolumeFileMap
      THEN
      BEGIN
      -- should also rewrite if Wrong Page???
      LabelTransfer.WriteLabels[MakeFreeDescriptor[vol], [p, p, p + pageCount]];
      RETURN[TRUE];
      END;
    RETURN[label.type = PilotFileTypes.tFreePage]
    END;
  WHILE page < next DO
    IF FixUpAndIsFree[page, next - page] THEN -- pageCount is set as side effect
      {
      IF vol.lowerBound > page THEN vol.lowerBound ← page;
      page ← page + pageCount}
    ELSE
      BEGIN
      i: Volume.PageCount ← 0;
      WHILE (i ← i + 1) <= pageCount --no, a subrange won't work-- DO
        IF VolAllocMap.AccessVAM[vol, page, TRUE, FALSE] THEN
          ERROR ImpossibleScavengerError[pageAlreadyBusy];
        -- can't be already set
        page ← page + 1;
        ENDLOOP;
      END;
    ENDLOOP;
  END;

VamSize: PROC [vol: LogicalVolume.Handle] RETURNS [Volume.PageCount] = {
  RETURN[(vol.volumeSize + bitsPerPage)/bitsPerPage]};

-- Vfm Scavange... Don't have to worry about bad pages, since they were
-- marked "free" in Building the VFM

Vfm: PROC [
  vol: LogicalVolume.Handle, space: SimpleSpace.Handle, erase: BOOLEAN] =
  BEGIN
  found: BOOLEAN;
  list: BadPageList;
  next: PageNumber;
  page: PageNumber ← 1;
  svEnd: Volume.PageCount;
  svH: SubVolume.Handle;
```

```
  VolFileMap.Close[TRUE];
  VolFileMap.InitMap[vol];
  IF ~erase THEN
    WHILE page < vol.volumeSize DO
      [found, svH] ← SubVolume.Find[vol.vID, page];
      IF ~found THEN ERROR ImpossibleScavengerError[cantFindSubvolume];
      GetBadList[svH, space, @list];
      svEnd ← svH.lvPage + svH.nPages;
      WHILE page < svEnd DO
        IF page = CurrentBadPage[@list] THEN {
          AdvanceBadPagePointer[@list]; page ← page + 1; LOOP};
        -- Add  First lvPage or next track to MIN
        next ← MIN[svEnd, CurrentBadPage[@list]];
        -- This next two tests not necessary in VFM since the pages are marked
        -- as Busy in the Vam
        -- IF page IN [vol.vamStart..vamEnd) THEN {page ← vamEnd; LOOP};
        -- IF page<vol.vamStart AND next > vol.vamStart THEN next ← vol.vamStart;
        VfmPieceRebuild[vol, page, next];
        page ← next;
        ENDLOOP;
      ENDLOOP;
  END;

VfmPieceRebuild: PROC [vol: LogicalVolume.Handle, start, next: PageNumber] =
  BEGIN
  fileD: FileInternal.Descriptor ← [, vol.vID, local[, , , ]];
  group: FileInternal.PageGroup;
  label: PilotDisk.Label;
  page: PageNumber ← start;
  pageCount: Volume.PageCount;
  filePage: File.PageNumber;
  WHILE page < next DO
    IF ~VolAllocMap.AccessVAM[vol, page, FALSE, FALSE] THEN {
      page ← page + 1; LOOP};
    label ← LabelTransfer.ReadLabel[fileD, 0, page];
    IF label.type IN PilotFileTypes.PilotVFileType THEN {page ← page + 1; LOOP};
    filePage ← PilotDisk.GetLabelFilePage[@label];
    fileD.fileID ← label.fileID;
    IF filePage = 0 AND label.zeroSize THEN {
      group ← [0, page, 0]; page ← page + 1}
    ELSE
      BEGIN
      rem: Volume.PageCount = next - page;
      [countValid: pageCount] ← LabelTransfer.VerifyLabels[
        file:
        [label.fileID, vol.vID, local[
          label.immutable, label.temporary, filePage + rem, label.type]],
        pageGroup: [filePage, page, filePage + rem],
        expectErrorAfterFirstPage: TRUE];
      IF pageCount = 0 THEN ImpossibleScavengerError[labelChanged];
      group ← [filePage, page, filePage + pageCount];
      page ← page + pageCount;
      END;
    VolFileMap.InsertPageGroup[vol, @fileD, @group];
    ENDLOOP;
  END;

Initialize: PROC =
```



```
  BEGIN
  FOR i: IndexOfContexts IN IndexOfContexts DO
    context: ScavengeContext;
    context ← @Contexts[i];
    context.occupied ← FALSE;
    context.buffer ← SimpleSpace.Create[1, hyperspace];
    context.bufferPointer ← SimpleSpace.LongPointer[context.buffer];
    ENDLOOP;
  END;

Initialize[];
END.....
```

LOG

*Time: October 2, 1979 11:51 AM By: Forrest Action: Created file from Vol*Map*
**Impl.mesa*
*Time: October 16, 1979 1:35 AM By: Forrest Action: Called Vol*map.close afte*
**r scavange*
Time: November 6, 1979 11:34 AM By: Forrest Action: Fixed VFM scavenge (as e
**xamining bogus pages)*
Time: November 16, 1979 4:13 PM By: Forrest Action: Added check for non-pilot
**volumes*
Time: November 19, 1979 11:50 AM By: Forrest Action: Added missing ~ to erase i
**n VFM*
Time: January 10, 1980 6:29 PM By: Forrest Action: Take advantage of new La
**belTransfer interface*
*Time: March 7, 1980 9:16 PM By: Forrest Action: Change calls to Vol*mapClose*
Time: May 29, 1980 10:22 AM By: Luniewski Action: PhysicalVolume = > PhysicalVolume
**Format. Exports PhysicalVolume.Error due to compiler bug vis a vis PhysicalVolumeImpl.*
Time: June 23, 1980 2:11 PM By: Luniewski Action: Interim implementation of the Scave
**nger interface. This just enumerates files after a scavenge completes.*
Time: July 16, 1980 5:57 PM By: McJones Action: Rewrite FOR loop in EnumerateFiles
**as plain DO loop to avoid Mesa AR 4956; FilePageLabel = >PilotDisk*
Time: August 4, 1980 4:33 PM By: Luniewski Action: Delete old log files whenever possibl
**e.*
Time: September 17, 1980 2:39 PM By: Luniewski Action: Use SimpleSpace.LongPo
**inter instead of Space.LongPointer.*
Time: October 9, 1980 4:09 PM By: Forrest Action: Convert log file access to a stream-ty
**pe interface.*
Time: October 11, 1980 10:10 PM By: Forrest Action: Add ExpectErrorAfterFirst
**Page parameter to VerifyLabels.*

```
-- VolAllocMapImpl.mesa (last edited by: Luniewski on: June 17, 1980 10:12 AM)

DIRECTORY
  Environment: FROM "Environment" USING [bitsPerWord, wordsPerPage],
  File: FROM "File" USING [Capability, ID, PageNumber, Type],
  FileInternal: FROM "FileInternal" USING [
    Descriptor, FilePtr, maxPermissions, PageGroup],
  FMPrograms: FROM "FMPrograms",
  Inline: FROM "Inline" USING [BITAND, BITSHIFT, BITXOR],
  LabelTransfer: FROM "LabelTransfer" USING [VerifyLabels, WriteLabels],
  LogicalVolume: FROM "LogicalVolume" USING [Free, Handle, PageNumber, Vam, Vfm],
  PhysicalVolume: FROM "PhysicalVolume" USING [ErrorType],
  PilotFileTypes: FROM "PilotFileTypes" USING [PilotVFileType, tFreePage],
  SimpleSpace: FROM "SimpleSpace" USING [Create, ForceOut, Map, Page, Unmap],
  Space: FROM "Space" USING [Handle, WindowOrigin],
  Utilities: FROM "Utilities" USING [LongPointerFromPage, ShortCARDINAL],
  VolAllocMap: FROM "VolAllocMap",
  Volume: FROM "Volume" USING [ID, nullID];

VolAllocMapImpl: MONITOR
  -- to protect VAM buffer; volume page is protected by callers

  IMPORTS Inline, LabelTransfer, LogicalVolume, SimpleSpace, Utilities
  EXPORTS FMPrograms, PhysicalVolume, VolAllocMap
  SHARES File =
  BEGIN OPEN Inline;
  -- The following is exported to PhysicalVolume as compiler "features" prevent it from being exp
**orted by SpecialVolume.
  Error: PUBLIC ERROR [PhysicalVolume.ErrorType] = CODE;
  FilePtr: TYPE = FileInternal.FilePtr;
  GroupPtr: TYPE = POINTER TO FileInternal.PageGroup;
  LvHandle: TYPE = LogicalVolume.Handle;
  -- common buffer used by procedures to Access VAM; protected by monitor
  bufferHandle: Space.Handle = SimpleSpace.Create[1, hyperspace];
  bufferPointer: LONG POINTER = Utilities.LongPointerFromPage[
    SimpleSpace.Page[bufferHandle]];
  -- Whenever lvID # Volume.nullID, the buffer is (being) mapped.  Therefore, if the ID is wrong an
**d non-null, one can automatically unmap.  See CLose, MapInternal
  currentBufferContents: RECORD [
    lvID: Volume.ID, page: LogicalVolume.PageNumber] ← [Volume.nullID, ];
  bitsPerPage: CARDINAL = (Environment.bitsPerWord*Environment.wordsPerPage);
  VolAllocMapImplError: ERROR [{setSizeToZeroPageGroup}] = CODE;
  LabelError: SIGNAL = CODE;
  -- This code assumes various powers of 2 (because of the truncation).
  AccessVAM: PUBLIC PROCEDURE [
    vol: LvHandle, volumePage: LogicalVolume.PageNumber, set: BOOLEAN,
    clear: BOOLEAN] RETURNS [busy: BOOLEAN] =
    -- set, clear or read one bit of vam, always returns the (previous) value;
    BEGIN
    bit: WORD;
    vamPage: LogicalVolume.PageNumber;
    word: LONG POINTER TO WORD;
    DoIt: ENTRY PROCEDURE = INLINE
      BEGIN
      MapVamPageInternal[vol.vID, vamPage, Vam[vol]];
      busy ← (0 # Inline.BITAND[word↑, bit]);
      IF (busy AND clear) OR (~busy AND set) THEN
        BEGIN
```



```
        word↑ ← Inline.BITXOR[word↑, bit];
        vol.freePageCount ← vol.freePageCount + (IF busy THEN 1 ELSE -1);
        END;
      IF clear THEN vol.lowerBound ← MIN[volumePage, vol.lowerBound];
      -- keep lowerBound to help allocator
      IF set AND volumePage = vol.lowerBound THEN
        vol.lowerBound ← vol.lowerBound + 1;
      END;
    -- The bit and word calculations can correctly be done by truncating first
    bit ← Inline.BITSHIFT[
      1, Utilities.ShortCARDINAL[volumePage] MOD Environment.bitsPerWord];
    word ←
      bufferPointer +
        (Utilities.ShortCARDINAL[volumePage]/Environment.bitsPerWord) MOD
          Environment.wordsPerPage;
    vamPage ← vol.vamStart + volumePage/bitsPerPage;
    DoIt[];
    END;
  -- allocates a group of pages and writes labels (group is a modifiable hint)

  AllocPageGroup: PUBLIC PROCEDURE [
    vol: LvHandle, filePtr: FilePtr, groupPtr: GroupPtr, createFile: BOOLEAN] =
    BEGIN OPEN groupPtr;
    skip: [0..1] = IF ~createFile AND filePage = 0 THEN 1 ELSE 0;
    offset: LONG CARDINAL;
    startPage: LogicalVolume.PageNumber = MAX[volumePage, vol.lowerBound];
    IF createFile OR filePage # 0 THEN
      BEGIN
      volumePage ← startPage;
      WHILE AccessVAM[vol, volumePage, TRUE, FALSE] DO
        --search for unbusy page
        IF (volumePage ← volumePage + 1) >= vol.volumeSize - 1 THEN
          volumePage ← vol.lowerBound;
        IF volumePage = startPage THEN ERROR --Volume.InsufficientSpace--
        ; --avoid rather than back out of

        ENDLOOP;
      END;
    FOR offset ← 0, offset + 1 WHILE offset < nextFilePage - filePage DO
      IF volumePage + offset >= vol.volumeSize THEN EXIT;
      IF offset # 0 THEN
      IF AccessVAM[vol, volumePage + offset, TRUE, FALSE] THEN EXIT;
      --first is set

      ENDLOOP;
    WITH filePtr SELECT FROM
      local =>
        IF type IN PilotFileTypes.PilotVFileType THEN filePage ← volumePage;
      ENDCASE;
    nextFilePage ← filePage + offset; -- side effect
    IF createFile AND nextFilePage = 0 THEN offset ← 1;
    -- creating only the attribute label page
    IF NOT LabelTransfer.VerifyLabels[
      filePtr↑, FileInternal.PageGroup[0, volumePage, skip]].labelsValid THEN
      SIGNAL LabelError;
    IF NOT LabelTransfer.VerifyLabels[
      FreeDescriptor[vol],
      [volumePage + skip, volumePage + skip, volumePage + offset]].labelsValid
```

```
    THEN SIGNAL LabelError;
  -- set new size
  WITH filePtr SELECT FROM local => size ← nextFilePage; ENDCASE => ERROR;
  LabelTransfer.WriteLabels[
    filePtr↑, [filePage, volumePage, filePage + offset]];
  END;

FreeDescriptor: PROCEDURE [v: LvHandle] RETURNS [FileInternal.Descriptor] =
  INLINE
  BEGIN
  RETURN[
    [LogicalVolume.Free[v], v.vID, local[
      FALSE, FALSE, v.volumeSize, PilotFileTypes.tFreePage]]];
  END;
-- free a page group (assumes calls to label operations with zero sizes are noops)
-- This is closely related to SetSize and Delete in FileImpl, as well as VolFileMap.deletePageGrou
**p

FreePageGroup: PUBLIC PROCEDURE [
  vol: LvHandle, filePtr: FilePtr, groupPtr: GroupPtr, deleteFile: BOOLEAN] =
  BEGIN
  Blast: PROCEDURE [g: FileInternal.PageGroup] =
    BEGIN
    p, nextVolumePage: LogicalVolume.PageNumber;
    nextVolumePage ← g.volumePage + g.nextFilePage - g.filePage;
    FOR p ← g.volumePage, p + 1 WHILE p < nextVolumePage DO
      [] ← AccessVAM[vol, p, FALSE, TRUE]; ENDLOOP;
    LabelTransfer.WriteLabels[
      FreeDescriptor[vol], [g.volumePage, g.volumePage, nextVolumePage]];
    END;
  -- Always verify labels; max is for zero size file [0, ?, 0]
  IF ~LabelTransfer.VerifyLabels[
    filePtr↑,
    [groupPtr.filePage, groupPtr.volumePage, MAX[
      groupPtr.nextFilePage, 1]]].labelsValid THEN SIGNAL LabelError;
  WITH filePtr SELECT FROM local => size ← groupPtr.filePage; ENDCASE => ERROR;
  SELECT TRUE FROM
    groupPtr.filePage # 0 => -- if group start # 0, always blast
      Blast[groupPtr↑];
    groupPtr.nextFilePage # 0 -- AND g.filePage = 0-- => -- Set to zero size
      BEGIN
      Blast[
        [groupPtr.filePage + 1, groupPtr.volumePage + 1,
          groupPtr.nextFilePage]];
      -- we could skip the write, but then we'd have to mess around with correcting 0 size files
      LabelTransfer.WriteLabels[filePtr↑, [0, groupPtr.volumePage, 1]];
      END;
    ENDCASE -- g.filePage = g.nextPage = 0-- =>
      IF deleteFile THEN Blast[[0, groupPtr.volumePage, 1]] --delete it--
      ELSE ERROR VolAllocMapImplError[setSizeToZeroPageGroup];
  END;

Vam: PROCEDURE [v: LvHandle] RETURNS [File.ID] = INLINE
  BEGIN RETURN[LogicalVolume.Vam[v]]; END;

Vfm: PROCEDURE [v: LvHandle] RETURNS [File.ID] = INLINE
  BEGIN RETURN[LogicalVolume.Vfm[v]]; END;
-- ENTRY PROCEDURES (Also see local proc in AccessVAM)
-- Force out/ deallocate allocation map
```



```
Close: PUBLIC ENTRY PROCEDURE [final: BOOLEAN] =
  BEGIN
  IF currentBufferContents.lvID # Volume.nullID THEN
    IF final THEN
      BEGIN
      SimpleSpace.Unmap[bufferHandle];
      currentBufferContents.lvID ← Volume.nullID;
      END
    ELSE SimpleSpace.ForceOut[bufferHandle];
  END;
-- INTERNAL PROCEDURE

MapVamPageInternal: PROCEDURE [
  vID: Volume.ID, vamPage: LogicalVolume.PageNumber, vamID: File.ID] = INLINE
  BEGIN
  IF currentBufferContents.lvID # vID THEN
    BEGIN
    IF currentBufferContents.lvID # Volume.nullID THEN
      SimpleSpace.Unmap[bufferHandle];
    currentBufferContents.lvID ← vID;
    -- and fall through for map...

    END
  ELSE
    IF currentBufferContents.page # vamPage THEN SimpleSpace.Unmap[bufferHandle]
      -- and fall through for map...

    ELSE -- vids equal and pages equal, so just....
      RETURN;
  currentBufferContents.page ← vamPage;
  SimpleSpace.Map[
    bufferHandle, Space.WindowOrigin[
    File.Capability[vamID, FileInternal.maxPermissions], vamPage], FALSE];
  END;

END.
```

```
LOG
Time: April 13, 1978 3:32 PM    By: Purcell          Action: Created file
Time: June 23, 1978 12:27 PM  By: Purcell          Action: page 0 special case in alloc/free gro
**up
Time: September 27, 1978 5:15 PM        By: Purcell          Action: don't raise signals and lim
**it bound
Time: October 19, 1978 2:45 PM          By: Purcell          Action: CR 20.103: Use FileTypes
**directly
Time: March 20, 1979 9:30 PM  By: Redell Action: Convert to Mesa 5.0
Time: August 1, 1979 3:10 PM  By: Redell Action: Convert to use FilePageLabel
Time: August 8, 1979 12:05 PM By: Redell Action: Bug fix: wraparound condition in AllocPageGro
**up was off by one: ran off end of volume.
Time: August 29, 1979 10:05 AM          By: Forrest          Action: Changed to match new int
**erface and new FileMgr organization.
Time: September 3, 1979 12:08 PM        By: Forrest          Action: Used LogicalVolume.free,
**vam, vfm vs Opens.
Time: September 4, 1979 11:04 AM        By: Forrest          Action: Fixed up buffer allocation
**/deallocation.
Time: October 1, 1979 5:56 PM By: Forrest          Action: Moved out Open to scavanger.
Time: January 9, 1980 5:43 PM By: Gobbel           Action: Change for new LabelTransfer interf
**ace: VerifyLabels now returns two values.
Time: March 7, 1980 8:49 PM   By: Forrest          Action: Lexically change types for New Logi
**cal Volume; deleted handle from Close.
```

Time: June 17, 1980 10:12 AM By: Luniewski Action: Exprt PhysicalVolume.Error.

```
-- VolFileMapImpl.mesa (last edited by: Luniewski on: September 17, 1980 5:15 PM)

DIRECTORY
  Environment USING [wordsPerPage],
  File USING [Capability, ID, lastPageNumber, nullCapability, PageNumber],
  FileInternal USING [Descriptor, FilePtr, maxPermissions, PageGroup],
  FMPrograms USING [],
  Inline USING [LongCOPY],
  KernelFile USING [],
  LogicalVolume USING [
    Handle, Interval, Key, Level, maxKey, maxID, nullInterval, nullIntervals,
    nullKey, nullVolumePage, PageNumber, Vfm, VolumeAccess, VolumeAccessProc],
  PilotFileTypes USING [tVolumeFileMap],
  SimpleSpace USING [Create, ForceOut, Map, Page, Unmap],
  Space USING [Handle],
  Utilities USING [LongPointerFromPage],
  VolAllocMap USING [AllocPageGroup, FreePageGroup],
  VolFileMap USING [],
  Volume USING [ID, NotOpen, nullID, Unknown];

VolFileMapImpl: MONITOR
  IMPORTS Inline, LogicalVolume, SimpleSpace, Utilities, VolAllocMap, Volume
  EXPORTS FMPrograms, KernelFile, VolFileMap
  SHARES File =
  BEGIN OPEN LogicalVolume;
  -- data types for the vfm
  -- fixed length, uncompressed indexes are stored now now but in the future compressed indexes
**will have variable lengths. Compressed format is defined by VolFileMap.PutNext and VolFileMa
**p.ReadNext
  Buffer: TYPE = RECORD [
    --This is a B-tree page
    data: ARRAY [0..indexInBuffer) OF Index,
    used: BufferPtr];
  BufferPtr: TYPE = INTEGER;
  GroupPtr: TYPE = POINTER TO FileInternal.PageGroup;
  Index: TYPE = RECORD [key: Key, volumePage: PageNumber, ptr: BufferPtr];
  KeyPtr: TYPE = POINTER TO Key;
  FindProc: TYPE = PROCEDURE;
  XtraProc: TYPE = PROCEDURE;
  indexInBuffer: CARDINAL =
    (Environment.wordsPerPage - SIZE[BufferPtr])/SIZE[Index];
  maxReadPtr: CARDINAL = SIZE[Buffer] - SIZE[BufferPtr];
  maxIndex: Index = Index[maxKey, nullVolumePage, 0];
  nullIndex: Index = Index[nullKey, nullVolumePage, 0];
  -- This is the monitor protected data. Someday, we may wish to have multiple sets of these beas
**ts
  currentID: Volume.ID ← Volume.nullID;
  volumeHandle: LogicalVolume.Handle ← NIL;
  -- Buffer used for most access; page# null => buffer is mapped
  bufferPage: PageNumber ← LogicalVolume.nullVolumePage;
  bufferHandle: Space.Handle = SimpleSpace.Create[1, hyperspace];
  buffer: LONG POINTER TO Buffer = Utilities.LongPointerFromPage[
    SimpleSpace.Page[bufferHandle]];
  -- Buffer used for splits/merges
  xtraHandle: Space.Handle = SimpleSpace.Create[1, hyperspace];
  xtraBuffer: LONG POINTER TO Buffer = Utilities.LongPointerFromPage[
    SimpleSpace.Page[xtraHandle]];
  interval: Interval ← nullInterval;
```



```
  old, low, high: Index ← nullIndex;
  VolFileMapImplError: ERROR [{pagesMissingInFile, illegalReturnCode}] = CODE;
  -- External Procedures
  Close: PUBLIC ENTRY PROCEDURE [final: BOOLEAN] = BEGIN ContextFlush[final]; END;
  -- deletes a pageGroup suffix (leaving zeroPageGroup unless explicitly asked) returns group del
**eted into groupPtr

  DeletePageGroup: PUBLIC ENTRY PROCEDURE [
    vol: LogicalVolume.Handle, filePtr: FileInternal.FilePtr,
    groupPtr: GroupPtr] =
    BEGIN
    key: Key ←
      [filePtr.fileID,
        groupPtr.nextFilePage - (IF groupPtr.nextFilePage = 0 THEN 0 ELSE 1)];
    interval: Interval;
    ContextSet[vol];
    interval ← Get[@key, 0];
    IF interval.key.fileID # filePtr.fileID OR interval.volumePage =
      nullVolumePage THEN ERROR VolFileMapImplError[pagesMissingInFile];
    groupPtr.filePage ← key.filePage ← MAX[
      interval.key.filePage, groupPtr.filePage];
    groupPtr.volumePage ←
      interval.volumePage + groupPtr.filePage - interval.key.filePage;
    IF groupPtr.filePage # 0 THEN
      BEGIN
      IF interval.key = key THEN Delete[@key, 0];
      Insert[@key, nullVolumePage, 0];
      -- no check for merging since preceding is assumed present

      END;
    key.filePage ← groupPtr.nextFilePage;
    Delete[@key, 0]; -- no check for merging since following is assumed vacant

    END;
  -- Does this belong here???

  GetNextFile: PUBLIC PROCEDURE [volume: Volume.ID, file: File.Capability]
    RETURNS [nextFile: File.Capability] =
    BEGIN
    fileD: local FileInternal.Descriptor ← [file.fID, volume, local[, , , ]];
    GetNextFileProc: ENTRY LogicalVolume.VolumeAccessProc =
      BEGIN
      key: Key ← Key[fileD.fileID, File.lastPageNumber];
      updateMarkers ← FALSE;
      ContextSet[volume];
      fileD.fileID ← Get[@key, 0].nextKey.fileID;
      IF fileD.fileID = LogicalVolume.maxID THEN
        fileD.fileID ← File.nullCapability.fID; -- end with null

      END;
    SELECT LogicalVolume.VolumeAccess[@volume, GetNextFileProc] FROM
      ok => NULL;
      volumeUnknown => ERROR Volume.Unknown[volume];
      volumeNotOpen => ERROR Volume.NotOpen[volume];
      ENDCASE => ERROR VolFileMapImplError[illegalReturnCode];
    RETURN[
      [fileD.fileID,
        IF fileD.fileID = File.nullCapability.fID THEN 0
```

```
    ELSE FileInternal.maxPermissions]]
  END;
-- finds page group containing key (filePage = nextFilePage = size when off end of file)

GetPageGroup: PUBLIC ENTRY PROCEDURE [
  vol: LogicalVolume.Handle, filePtr: FileInternal.FilePtr,
  filePage: File.PageNumber]
  RETURNS [success: BOOLEAN, group: FileInternal.PageGroup] =
  BEGIN
  key: Key ← Key[filePtr.fileID, filePage];
  interval: Interval;
  ContextSet[vol];
  interval ← Get[@key, 0];
  RETURN[
    interval.key.fileID = filePtr.fileID, FileInternal.PageGroup[
      filePage: interval.key.filePage, volumePage: interval.volumePage,
      nextFilePage:
      (IF interval.nextKey.fileID = filePtr.fileID THEN interval.nextKey
       ELSE interval.key).filePage]]; --covers page zero and size requests

  END;
-- inserts a pageGroup into B-tree (unordered inserts are merged for rebuild)

InsertPageGroup: PUBLIC ENTRY PROCEDURE [
  vol: LogicalVolume.Handle, filePtr: FileInternal.FilePtr,
  groupPtr: GroupPtr] =
  BEGIN
  key: Key ← [filePtr.fileID, groupPtr.filePage];
  interval: Interval;
  ContextSet[vol];
  interval ← Get[@key, 0];
  IF interval.key = key THEN
    BEGIN Delete[@key, 0]; interval ← Get[@key, 0]; END;
  IF key.filePage - interval.key.filePage #
    groupPtr.volumePage - interval.volumePage OR key.fileID #
    interval.key.fileID THEN Insert[@key, groupPtr.volumePage, 0];
  --merge with previous
  key.filePage ← groupPtr.nextFilePage;
  IF interval.nextKey # key AND groupPtr.filePage # groupPtr.nextFilePage THEN
    Insert[@key, nullVolumePage, 0];
  IF interval.nextKey = key AND Get[@key, 0].volumePage =
    interval.volumePage + interval.nextKey.filePage - interval.key.filePage THEN
    Delete[@key, 0]; --merge with following

  END;

InitMap: PUBLIC ENTRY PROCEDURE [vol: LogicalVolume.Handle] =
  BEGIN
  level: LogicalVolume.Level;
  nullKeyInstance: Key ← nullKey;
  vol.interval ← LogicalVolume.nullIntervals;
  ContextSet[vol];
  vol.vfmStart ← CreateVPage[];
  FOR level DECREASING IN [0..vol.treeLevel) DO
    --init the tree
    Insert[
      @nullKeyInstance, (IF level = 0 THEN nullVolumePage ELSE CreateVPage[]),
      level];
```



```
    ENDLOOP;
  END;
-- INTERNAL PROCEDURES.....
-- ~~~~~ These Two Beasties Play with the variables in grand fashion

ContextFlush: INTERNAL PROCEDURE [final: BOOLEAN] = INLINE
  BEGIN
  IF bufferPage # LogicalVolume.nullVolumePage THEN
    IF final THEN
      BEGIN
      SimpleSpace.Unmap[bufferHandle];
      bufferPage ← LogicalVolume.nullVolumePage;
      END
    ELSE SimpleSpace.ForceOut[bufferHandle];
  IF final THEN currentID ← Volume.nullID;
  END;

ContextSet: INTERNAL PROCEDURE [vol: LogicalVolume.Handle] =
  BEGIN
  IF currentID # vol.vID THEN
    BEGIN
    IF bufferPage # LogicalVolume.nullVolumePage THEN
      BEGIN
      SimpleSpace.Unmap[bufferHandle];
      bufferPage ← LogicalVolume.nullVolumePage;
      END;
    currentID ← vol.vID;
    END;
  volumeHandle ← vol;
  END;
-- merely calls VolAllocMap.AllocPageGroup to get a new page for the vfm B-tree

CreateVPage: INTERNAL PROCEDURE RETURNS [LogicalVolume.PageNumber] =
  BEGIN OPEN volumeHandle↑;
  group: FileInternal.PageGroup ← [0, 0, 1];
  vfmFileD: FileInternal.Descriptor ←
    [LogicalVolume.Vfm[volumeHandle], vID, local[
      FALSE, FALSE, volumeSize, PilotFileTypes.tVolumeFileMap]];
  VolAllocMap.AllocPageGroup[volumeHandle, @vfmFileD, @group, TRUE];
  RETURN[group.volumePage];
  END;

DeleteError: SIGNAL = CODE;
-- deletes the index <= the key, error if no index (merge is called from find)
Delete: INTERNAL PROCEDURE [key: KeyPtr, level: Level] =
  BEGIN
  firstFlag, lastFlag: BOOLEAN;
  volumePage: LogicalVolume.PageNumber;
  -- page holding key (delete if firstFlag AND lastFlag)
  nextKey: Key; -- the following key must be slid down over deleted key
  Delete1: INTERNAL FindProc =
    BEGIN
    firstFlag ← (low.ptr = 0);
    lastFlag ← (high.ptr = buffer.used);
    volumePage ← interval.volumePage;
    nextKey ← high.key;
    IF low.key ~= key↑ THEN SIGNAL DeleteError;
    IF firstFlag AND NOT lastFlag THEN
      Inline.LongCOPY[
```

```
      buffer + high.ptr, (buffer.used ← buffer.used - high.ptr), buffer];
    END;
  Delete2: INTERNAL FindProc =
    BEGIN
    tempBuffer: Buffer;
    high ← Index[nextKey, low.volumePage, high.ptr];
    low ← old;
    Inline.LongCOPY[buffer + high.ptr, buffer.used - high.ptr, @tempBuffer];
    PutNext[@high.key, high.volumePage, level];
    Inline.LongCOPY[@tempBuffer, buffer.used - high.ptr, buffer + low.ptr];
    buffer.used ← low.ptr + buffer.used - high.ptr;
    END;
  Find[key, level, Delete1]; -- get the preceeding index
  IF firstFlag THEN
    BEGIN
    Delete[key, level + 1];
    IF lastFlag THEN FreeVPage[volumePage] --free after remaping

    ELSE Insert[@nextKey, volumePage, level + 1];
    END;
  Find[key, level, Delete2]; -- get the preceeding index

  END;
-- executes proc with context (buffer, low, high) surrounding key (merges too)

Find: INTERNAL PROCEDURE [key: KeyPtr, level: Level, proc: FindProc] =
  BEGIN
  interval ← Get[key, level + 1];
  IF interval.volumePage # bufferPage THEN
    BEGIN
    IF bufferPage # LogicalVolume.nullVolumePage THEN
      SimpleSpace.Unmap[bufferHandle];
    SimpleSpace.Map[
      bufferHandle,
      [[LogicalVolume.Vfm[volumeHandle], FileInternal.maxPermissions],
        interval.volumePage], FALSE];
    bufferPage ← interval.volumePage;
    END;
  old ← low ← high ← Index[interval.key, nullVolumePage, 0]; --init reader
  UNTIL Lower[key, @high.key] DO ReadNext[]; ENDLOOP;
  --scan this page till key is passed
  proc[];
  -- test before 'unmap', Merge after; postpone Unmap until later to optimize out
  IF buffer.used <= SIZE[Buffer]/3 AND interval.nextKey # maxKey THEN
    Merge[@old.key, level]; --is interval.key correct? (consider merge)

  END;
-- calls VolAllocMap.FreePageGroup to free a page of the vfm B-tree

FreeVPage: INTERNAL PROCEDURE [volumePage: LogicalVolume.PageNumber] =
  BEGIN
  group: FileInternal.PageGroup ← [volumePage, volumePage, volumePage + 1];
  vfmFileD: FileInternal.Descriptor ←
    [LogicalVolume.Vfm[volumeHandle], volumeHandle.vID, local[
      FALSE, FALSE, volumeHandle.volumeSize, PilotFileTypes.tVolumeFileMap]];
  VolAllocMap.FreePageGroup[volumeHandle, @vfmFileD, @group, TRUE];
  END;

Get: INTERNAL PROCEDURE [key: KeyPtr, level: Level]
```



```
  RETURNS [interval: Interval] =
  BEGIN -- returns value without reading a page when possible
  Get1: INTERNAL FindProc =
    BEGIN
    volumeHandle.interval[level] ← Interval[low.key, high.volumePage, high.key];
    END;
  IF level > volumeHandle.treeLevel OR level > LAST[Level] THEN ERROR;
  IF level = volumeHandle.treeLevel THEN
    RETURN[Interval[nullKey, volumeHandle.vfmStart, maxKey]];
  interval ← volumeHandle.interval[level]; -- try the cached intry
  IF Lower[key, @interval.key] OR ~Lower[key, @interval.nextKey] THEN
    Find[key, level, Get1];
  RETURN[volumeHandle.interval[level]];
  END;
-- inserts (key, volumePage (including duplicates) calling split if necessary

Insert: INTERNAL PROCEDURE [key: KeyPtr, volumePage: PageNumber, level: Level] =
  BEGIN
  splitFlag: BOOLEAN;
  Insert1: INTERNAL FindProc =
    BEGIN
    tempBuffer: Buffer;
    IF (splitFlag ← (buffer.used > maxReadPtr - SIZE[Index])) THEN RETURN;
    Inline.LongCOPY[buffer + high.ptr, buffer.used - high.ptr, @tempBuffer];
    IF low.ptr < buffer.used THEN PutNext[key, high.volumePage, level];
    PutNext[@high.key, volumePage, level];
    Inline.LongCOPY[@tempBuffer, buffer.used - high.ptr, buffer + low.ptr];
    buffer.used ← low.ptr + buffer.used - high.ptr;
    END;
  Find[key, level, Insert1];
  IF splitFlag THEN BEGIN Split[key, level]; Find[key, level, Insert1]; END;
  END;
-- compares two keys for ordering

Lower: INTERNAL PROCEDURE [a, b: KeyPtr] RETURNS [BOOLEAN] =
  BEGIN
  x: POINTER TO ARRAY [0..SIZE[File.ID]) OF CARDINAL = LOOPHOLE[@a.fileID];
  y: POINTER TO ARRAY [0..SIZE[File.ID]) OF CARDINAL = LOOPHOLE[@b.fileID];
  i: [0..SIZE[File.ID]);
  FOR i IN [0..SIZE[File.ID]) DO
    SELECT TRUE FROM
      (x[i] < y[i]) => RETURN[TRUE];
      (x[i] > y[i]) => RETURN[FALSE];
      ENDCASE; -- if equal keep checking

    ENDLOOP;
  RETURN[a.filePage < b.filePage OR b↑ = maxKey];
  -- maxKey < maxKey, to close key space

  END;
-- tries to merge page of oldInterval with next page at same level or with root
-- cannot merge last page of any level except rootLevel

Merge: INTERNAL PROCEDURE [key: KeyPtr, level: Level] =
  BEGIN
  mergeFlag: BOOLEAN;
  leftInterval, rightInterval: Interval;
  Merge1: INTERNAL FindProc =
    BEGIN
```

```
    Merge2: INTERNAL XtraProc =
      BEGIN
      xtraBufferUsed: INTEGER ← xtraBuffer.used;
      -- used to solve stack modeling error
      IF level = volumeHandle.treeLevel - 1 THEN xtraBuffer.used ← 0;
      --clear now instead of deleteing later
      IF (mergeFlag ← (buffer.used + xtraBuffer.used < SIZE[Buffer])) THEN
        --is merging possible
        BEGIN --merge pages
        Inline.LongCOPY[buffer, buffer.used, xtraBuffer + xtraBufferUsed];
        --merge buffer with xtra
        xtraBuffer.used ← xtraBuffer.used + buffer.used;
        -- buffer.used remains to prevent Find from attempting a merge
        --interval.volumePage ← ??Interval.volumePage;--
        --update cache with new page for deleted contents

        END
      ELSE
        BEGIN
        --balance pages simply to provide hysteresis against futil merge attempts
        WHILE low.ptr < (buffer.used - xtraBuffer.used)/2 DO
          ReadNext[]; ENDLOOP; --find middle
        Inline.LongCOPY[buffer, low.ptr, xtraBuffer + xtraBufferUsed];
        --move first of buffer to xtra
        Inline.LongCOPY[buffer + low.ptr, buffer.used - low.ptr, buffer];
        --slide down the rest of buffer
        xtraBuffer.used ← xtraBuffer.used + low.ptr;
        buffer.used ← buffer.used - low.ptr;
        --use low to insert while it is still valid
        rightInterval.key ← low.key;
        END;
      END;
    rightInterval ← interval;
    Xtra[leftInterval.volumePage, Merge2];
    END;
  leftInterval ← Get[key, level + 1]; -- so as to get a valid volumePage
  Find[@leftInterval.nextKey, level, Merge1]; --beware the merging
  Delete[@leftInterval.nextKey, level + 1];
  IF mergeFlag THEN FreeVPage[rightInterval.volumePage]
  ELSE Insert[@rightInterval.key, rightInterval.volumePage, level + 1];
  -- insert new index;

  END;
-- compresses item in the context of low
-- Note the side effect on low but not on high ! !
-- no compression is implemented in this version, but useful one would include:
-- front compression (especially to shrink page groups back to 2 fields)

PutNext: INTERNAL PROCEDURE [
  key: KeyPtr, volumePage: PageNumber, level: Level] =
  BEGIN -- merge page groups when possible
  old ← low;
  low ←
    (LOOPHOLE[buffer + low.ptr, LONG POINTER TO Index]↑ ← Index[
      key↑, volumePage, SIZE[Index]]); --.ptr is just an increment
  low.ptr ← low.ptr + old.ptr; --low.ptr was just an increment
  volumeHandle.interval[level] ← Interval[old.key, low.volumePage, low.key];
  -- keep cache up to date in the face of changes
```



```
  END;
-- decompresses item at high to become low & bumps high
-- Note the side effect on low and high ! !
-- no compression is implemented in this version

ReadNext: INTERNAL PROCEDURE =
  BEGIN
  IF high.ptr > buffer.used THEN ERROR;
  old ← low;
  low ← high;
  high ←
    (IF high.ptr < buffer.used THEN LOOPHOLE[buffer + high.ptr, LONG POINTER TO
      Index]↑ ELSE Index[maxKey, nullVolumePage, 0]);
  high.ptr ← high.ptr + low.ptr; --high.ptr was just an increment

  END;
-- moves half of buffer (or root) to xtraBuffer, creating new page of tree

Split: INTERNAL PROCEDURE [key: KeyPtr, level: Level] =
  BEGIN
  keyStone: Key; -- half way mark
  page: PageNumber ← CreateVPage[];
  Split1: INTERNAL FindProc =
    BEGIN
    Split2: INTERNAL XtraProc =
      BEGIN
      high ← Index[interval.key, nullVolumePage, 0]; -- quick readReset
      UNTIL high.ptr > buffer.used/2 DO ReadNext[]; ENDLOOP;
      Inline.LongCOPY[
        buffer + low.ptr, (xtraBuffer.used ← buffer.used - low.ptr),
        xtraBuffer]; --split
      buffer.used ← low.ptr;
      --volume.interval[level + 1].nextKey ← low.key;--
      -- interval of buffer is smaller (must keep high key above right)????
      keyStone ← low.key;
      END;
    Xtra[page, Split2];
    END;
  Find[key, level, Split1];
  Insert[@keyStone, page, level + 1];
  END;

Xtra: INTERNAL PROCEDURE [page: LogicalVolume.PageNumber, proc: XtraProc] =
  -- maps, operates, unmaps xtraBuffer
  BEGIN
  SimpleSpace.Map[
    xtraHandle,
    [[LogicalVolume.Vfm[volumeHandle], FileInternal.maxPermissions], page],
    FALSE];
  proc[];
  SimpleSpace.Unmap[xtraHandle];
  END;

END.
LOG
Log trimmed to Teak
Time: March 7, 1980 7:53 PM  By: Forrest  Action: Moved in GetNextFile; deleted Handl
**e from Close.  Named the error in Delete.
Time: July 25, 1980 2:42 PM  By: Luniewski  Action: Deleted use of SpecialVolume.
```

Time: September 17, 1980 5:15 PM By: Luniewski Action: Modified GetNextFile.Get
***NextFileProc to return a BOOLEAN value as required by VolumeAccess.*

```
-- VolumeImpl.mesa (last edited by: Jose on: October 21, 1980 1:54 PM

DIRECTORY
  Boot USING [Location, LVBootFiles, nullDiskFileID, VolumeType],
  Device USING [Type],
  DiskChannel USING [DiskPageCount, Drive, GetAttributes, GetDriveAttributes],
  Environment USING [wordsPerPage],
  File USING [
    Capability, ID, nullCapability, nullID, PageCount, PageNumber, Permissions,
    Type],
  FileCache USING [
    FlushFile, FlushFilesOnVolume, GetFilePtrs, PinnedAction, SetFile,
    SetPageGroup],
  FileInternal USING [Descriptor, maxPermissions, PageGroup],
  FMPrograms USING [],
  Inline USING [LowHalf],
  KernelFile USING [],
  LabelTransfer USING [ReadLabel, WriteLabels],
  LogicalVolume USING [
    CloseVolumeAndFlushFiles, currentVersion, Handle, nullBoot, OpenStatus,
    OpenVolumeAndDeleteTemps, PageNumber, ReadOnlyVolume, rootPageNumber,
    ScavengeVolume, seal, SetFree, SetVam, SetVfm, VolumeAccess, VolumeAccessProc,
    VolumeAccessStatus],
  MarkerPage USING [CreateMarkerPage, Find, UpdateLogicalMarkerPages],
  PhysicalVolume USING [Error, ErrorType, ID, PageNumber],
  PhysicalVolumeFormat USING [
    descriptorSize, Handle, maxSubVols, PageCount, PageNumber, rootPageNumber,
    SubVolumeDesc],
  PilotFileTypes USING [
    PilotRootFileType, tFreePage, tLogicalVolumeRootPage, tVolumeAllocationMap,
    tVolumeFileMap],
  PilotDisk USING [GetLabelFilePage, GetLabelType, Label],
  PilotSwitches USING [switches --.u,.z--],
  Process USING [InitializeCondition, Ticks],
  Scavenger USING [Error, ErrorType, Scavenge],
  SimpleSpace USING [Create, ForceOut, Handle, Map, Page, Unmap],
  SpecialVolume USING [],
  SubVolume USING [Find, Handle, OnLine],
  System USING [GetUniversalID, NetworkAddress, UniversalID],
  Utilities USING [LongPointerFromPage],
  VolAllocMap USING [Close],
  VolFileMap USING [Close],
  Volume USING [
    ID, NeedsScavenging, NotOpen, nullID, onlyEnumerateCurrentType, Open,
    PageCount, Status, systemID, Type, TypeSet, Unknown],
  VolumeExtras USING [],
  VolumeImplInterface USING [AccessPhysicalVolumeRootPage],
  VolumeInternal USING [PageNumber];

VolumeImpl: MONITOR [
  -- to protect activeVolume and LVT
  bootFile: LONG POINTER TO disk Boot.Location,
  pLVBootFiles: POINTER TO Boot.LVBootFiles,
  debuggerDeviceType: POINTER TO Device.Type,
  debuggerDeviceOrdinal: POINTER TO CARDINAL]
IMPORTS
  DiskChannel, FileCache, PilotDisk, Inline, LabelTransfer, LogicalVolume,
  MarkerPage, PhysicalVolume, PilotSwitches, Process, Scavenger, SimpleSpace,
```



```
  SubVolume, System, Utilities, VolAllocMap, VolFileMap, Volume,
  VolumeImplInterface
EXPORTS
  FMPrograms, KernelFile, LogicalVolume, SpecialVolume, Volume, VolumeExtras,
  VolumeImplInterface
SHARES File =
BEGIN
LvHandle: TYPE = LogicalVolume.Handle;
PvHandle: TYPE = PhysicalVolumeFormat.Handle;
scavengingComplete: CONDITION;
debugClass: Boot.VolumeType;
debugSearchState: {tooSoonToLook, looking, done} ← tooSoonToLook;
lvRootPage: LogicalVolume.PageNumber = LogicalVolume.rootPageNumber;
pvRootPage: PhysicalVolumeFormat.PageNumber =
  PhysicalVolumeFormat.rootPageNumber;
-- The following is used as a buffer for the scavenger.  It must be large enough to hold
-- the largest possible physical volume root page + bad page list
extraBuffer: SimpleSpace.Handle = SimpleSpace.Create[
  PhysicalVolumeFormat.descriptorSize/Environment.wordsPerPage, hyperspace, 1];
-- The "active volume" whose LogicalVolume.Descriptor (root page) is buffered in memory
logicalRootPageBuffer: SimpleSpace.Handle = SimpleSpace.Create[1, hyperspace];
activeVolume: LvHandle = Utilities.LongPointerFromPage[
  SimpleSpace.Page[logicalRootPageBuffer]];
activeVID: Volume.ID ← Volume.nullID;
-- active volume (nullID => no active volume)
-- Logical Volume Table (lists logical volumes which are on-line)
LVTHandle: TYPE = --LONG--POINTER TO Volume.ID;
VolumeStatus: TYPE = RECORD [
  lvID: Volume.ID,
  beingScavenged, onLine, open, readOnly: BOOLEAN,
  -- readOnly is a user settable property of the volume.  A volume is read-only if readOnly is TRU
**E OR if the volume is of a "higher" type than the system volume.
  type: Volume.Type,
  pieceCount: CARDINAL [0..777B)];
nullVolumeStatus: VolumeStatus =
  [Volume.nullID, FALSE, FALSE, FALSE, FALSE, nonPilot, 0];
LVT: ARRAY LVTIndex OF VolumeStatus ← ALL[nullVolumeStatus];
LVTIndex: TYPE = [0..maxLVs);
maxLVs: CARDINAL = 12; -- Maximum logical volumes allowed on-line at once
VolImplError: ERROR [
  {
  impossibleSelectError, impossibleSubvolumeNotFound, notFoundInLVT,
  lvTableFull, illegalReturnCode}] = CODE;

-- Volume

systemID: PUBLIC Volume.ID ← Volume.nullID;
InsufficientSpace: PUBLIC ERROR = CODE;
NeedsScavenging: PUBLIC ERROR = CODE;
NotOpen: PUBLIC ERROR [volume: Volume.ID] = CODE;
Unknown: PUBLIC ERROR [volume: Volume.ID] = CODE;

Close: PUBLIC PROCEDURE [volume: Volume.ID] = {
  LogicalVolume.CloseVolumeAndFlushFiles[@volume]};

GetAttributes: PUBLIC PROCEDURE [volume: Volume.ID]
  RETURNS [
    volumeSize, freePageCount: Volume.PageCount, rootFile: File.Capability] =
```

```
    BEGIN
    GetAttributes1: PROCEDURE [vol: LvHandle] =
      BEGIN
      volumeSize ← vol.volumeSize;
      freePageCount ← vol.freePageCount;
      rootFile ← vol.clientRootFile;
      END;
    GetLogicalRootPage[@volume, GetAttributes1];
    END;

  GetLabelString: PUBLIC PROCEDURE [volume: Volume.ID, s: STRING] =
    BEGIN
    badVersion: BOOLEAN;
    GetLabelString1: PROCEDURE [vol: LvHandle] =
      BEGIN
      i: CARDINAL;
      s.length ← 0;
      IF badVersion ← (vol.version # LogicalVolume.currentVersion) THEN RETURN;
      s.length ← MIN[s.maxlength, vol.labelLength, LENGTH[vol.label]];
      FOR i IN [0..s.length) DO s[i] ← vol.label[i]; ENDLOOP;
      END;
    IF s # NIL THEN
      BEGIN
      GetLogicalRootPage[@volume, GetLabelString1];
      IF badVersion THEN ERROR Volume.NeedsScavenging;
      END;
    END;

  GetNext: PUBLIC ENTRY PROCEDURE [
    volume: Volume.ID, includeWhichVolumes: Volume.TypeSet ← []]
    RETURNS [Volume.ID] =
    BEGIN
    volumeFound: BOOLEAN ← volume = Volume.nullID;
    lv: LVTIndex;
    systemVolumeType: Volume.Type = LVGetStatus[Volume.systemID].type;
    IF IsUtilityPilot[] THEN
      includeWhichVolumes ← [normal: TRUE, debugger: TRUE, debuggerDebugger: TRUE]
    ELSE
      IF includeWhichVolumes = Volume.onlyEnumerateCurrentType THEN
        includeWhichVolumes[systemVolumeType] ← TRUE;
    -- all others are already FALSE
    FOR lv IN LVTIndex DO
      IF volumeFound AND LVT[lv].lvID # Volume.nullID AND LVT[lv].onLine AND
        includeWhichVolumes[LVT[lv].type] THEN RETURN[LVT[lv].lvID];
      IF LVT[lv].lvID = volume THEN volumeFound ← TRUE;
      ENDLOOP;
    IF volumeFound THEN RETURN[Volume.nullID]
    ELSE RETURN WITH ERROR Volume.Unknown[volume];
    END;

  GetStatus: PUBLIC ENTRY PROCEDURE [vID: Volume.ID]
    RETURNS [status: Volume.Status] =
    BEGIN
    onLine, open: BOOLEAN;
    IF vID = Volume.nullID OR ~LogicalVolumeFind[@vID] THEN RETURN[unknown];
    [onLine: onLine, open: open] ← LVGetStatus[vID];
    IF ~onLine THEN status ← partiallyOnLine
    ELSE
      IF open THEN status ← open
      ELSE
```



```
        BEGIN -- only activate when absolutely necessary!
        Activate[vID];
        IF activeVolume.changing THEN status ← closedAndInconsistent
        ELSE status ← closedAndConsistent;
        END;
    END;

  GetType: PUBLIC PROCEDURE [lvID: Volume.ID] RETURNS [type: Volume.Type] =
    BEGIN
    GetTypeInner: PROCEDURE [vol: LvHandle] = {type ← vol.type};
    GetLogicalRootPage[@lvID, GetTypeInner];
    END;

  IsOnServer: PUBLIC PROCEDURE [
    volume: Volume.ID, netAddress: System.NetworkAddress] = {
    --not implemented--};

  Open: PUBLIC PROCEDURE [volume: Volume.ID] =
    BEGIN
    DeleteTemps: ENTRY PROCEDURE RETURNS [doDelete: BOOLEAN] = INLINE
      BEGIN
      IF volume = Volume.nullID OR ~LogicalVolumeFind[@volume] THEN
        RETURN WITH ERROR Volume.Unknown[volume];
      IF LVGetStatus[volume].type <= debugClass -- type of system volume -- THEN
        RETURN[TRUE]
      ELSE RETURN[FALSE];
      END;

    IF ~DeleteTemps[] THEN
      SELECT OpenLogicalVolume[@volume] FROM
        ok, wasOpen => NULL;
        Unknown => ERROR Volume.Unknown[volume];
        VolumeNeedsScavenging => ERROR Volume.NeedsScavenging;
        ENDCASE => ERROR
    ELSE LogicalVolume.OpenVolumeAndDeleteTemps[@volume];
    END;

  SetRootFile: PUBLIC PROCEDURE [volume: Volume.ID, file: File.Capability] =
    BEGIN
    SetRootFile1: LogicalVolume.VolumeAccessProc =
      BEGIN volume.clientRootFile ← file; updateMarkers ← TRUE; END;
    SignalVolumeAccess[@volume, SetRootFile1];
    END;

  --
  -- VolumeExtras

  OpenVolume: PUBLIC PROCEDURE [volume: Volume.ID, readOnly: BOOLEAN] =
    BEGIN
    DeleteTemps: ENTRY PROCEDURE RETURNS [doDelete: BOOLEAN] = INLINE
      BEGIN
      IF volume = Volume.nullID OR ~LogicalVolumeFind[@volume] THEN
        RETURN WITH ERROR Volume.Unknown[volume];
      IF readOnly THEN RETURN[FALSE];
      IF LVGetStatus[volume].type <= debugClass -- type of system volume -- THEN
        RETURN[TRUE]
      ELSE RETURN[FALSE];
      END;
```

```
    IF ~DeleteTemps[] THEN
    SELECT OpenLogicalVolumeInternal[@volume, readOnly] FROM
      ok, wasOpen => NULL;
      Unknown => ERROR Volume.Unknown[volume];
      VolumeNeedsScavenging => ERROR Volume.NeedsScavenging;
      ENDCASE => ERROR
  ELSE LogicalVolume.OpenVolumeAndDeleteTemps[@volume];
  END;

--
-- SpecialVolume

GetLogicalVolumeBootFiles: PUBLIC PROCEDURE [
  lvID: Volume.ID, pBootFiles: LONG POINTER TO Boot.LVBootFiles] =
  BEGIN
  GetLVBootFiles1: PROCEDURE [vol: LvHandle] = {pBootFiles↑ ← vol.bootingInfo};
  IF pBootFiles # NIL THEN GetLogicalRootPage[@lvID, GetLVBootFiles1];
  END;

SetLogicalVolumeBootFiles: PUBLIC PROCEDURE [
  lvID: Volume.ID, pBootFiles: LONG POINTER TO Boot.LVBootFiles] =
  BEGIN
  SetLVBootFiles1: LogicalVolume.VolumeAccessProc =
    BEGIN volume.bootingInfo ← pBootFiles↑; updateMarkers ← TRUE; END;
  IF pBootFiles # NIL THEN SignalVolumeAccess[@lvID, SetLVBootFiles1];
  END;

--
-- KernelFile

GetRootFile: PUBLIC PROCEDURE [type: File.Type, volume: Volume.ID]
  RETURNS [file: File.Capability] =
  BEGIN
  GetRootFile1: PROCEDURE [vol: LvHandle] = {
    file ← [vol.rootFileID[type], FileInternal.maxPermissions]};
  IF type NOT IN PilotFileTypes.PilotRootFileType THEN ERROR;
  GetLogicalRootPage[@volume, GetRootFile1];
  IF file.fID = File.nullCapability.fID THEN file ← File.nullCapability;
  END;

PutRootFile: PUBLIC PROCEDURE [
  pVID: POINTER TO READONLY Volume.ID, type: File.Type,
  file: POINTER TO READONLY File.Capability] =
  BEGIN
  PutProc: LogicalVolume.VolumeAccessProc = {
    volume.rootFileID[type] ← file.fID; updateMarkers ← TRUE};
  IF type IN PilotFileTypes.PilotRootFileType THEN
    SignalVolumeAccess[pVID, PutProc];
  END;

--
-- LogicalVolume

ReadOnlyVolume: PUBLIC ERROR = CODE;
BeginScavenging: PUBLIC ENTRY PROCEDURE [pVID: POINTER TO READONLY Volume.ID] =
  BEGIN
  IF pVID↑ = Volume.nullID OR ~LogicalVolumeFind[pVID] THEN
    RETURN WITH ERROR Unknown[pVID↑];
  IF LVGetStatus[pVID↑].open THEN RETURN WITH ERROR Scavenger.Error[volumeOpen];
```



```
  IF IsReadOnly[pVID↑] THEN RETURN WITH ERROR LogicalVolume.ReadOnlyVolume;
  Activate[pVID↑];
  activeVolume.changing ← TRUE;
  SimpleSpace.ForceOut[logicalRootPageBuffer];
  LVSetScavenged[pVID↑, TRUE];
  LogicalVolume.ScavengeVolume[activeVolume, extraBuffer, FALSE];
  VolFileMap.Close[TRUE];
  VolAllocMap.Close[TRUE];
  activeVolume.changing ← FALSE;
  Deactivate[];
  END;

CloseLogicalVolume: PUBLIC ENTRY PROCEDURE [
  pVID: POINTER TO READONLY Volume.ID] =
  BEGIN
  IF pVID↑ = Volume.nullID OR ~LogicalVolumeFind[pVID] THEN
    RETURN WITH ERROR Unknown[pVID↑];
  Activate[pVID↑];
  FileCache.FlushFilesOnVolume[pVID↑, keep];
  -- Let all I/O on the volume quiesce
  VolFileMap.Close[TRUE];
  VolAllocMap.Close[TRUE];
  FileCache.FlushFilesOnVolume[pVID↑, keep];
  -- Wait for any I/O that slipped in through the cracks to finish
  LVSetOpen[pVID↑, FALSE];
  Deactivate[];
  END;

EndScavenging: PUBLIC ENTRY PROCEDURE [pVID: POINTER TO READONLY Volume.ID] = {
  LVSetScavenged[pVID↑, FALSE]; BROADCAST scavengingComplete};

GetContainingPhysicalVolume: PUBLIC PROCEDURE [lvID: Volume.ID]
  RETURNS [pvID: PhysicalVolume.ID] =
  BEGIN
  RETURN[
    MarkerPage.Find[
      [drive[
        DiskChannel.GetAttributes[
        SubVolume.Find[lvID, lvRootPage].subVolume.channel]]]].physicalID↑]
  END;

LogicalVolumeCreate: PUBLIC ENTRY PROCEDURE [
  pvID: PhysicalVolume.ID, size: Volume.PageCount, name: STRING,
  type: Volume.Type, minPVPageNumber: PhysicalVolume.PageNumber]
  RETURNS [Volume.ID] =
  BEGIN
  i: CARDINAL;
  lvID: Volume.ID ← [System.GetUniversalID[]];
  minVolumeSize: PhysicalVolumeFormat.PageCount = 50;
  bareVolumeSize: PhysicalVolumeFormat.PageCount = 1 + 1 + 6;
  thisSv: CARDINAL;
  thisSvStart: PhysicalVolumeFormat.PageNumber;
  error: PhysicalVolume.ErrorType;
  isError: BOOLEAN ← FALSE;
  proc1: PROCEDURE [p: PvHandle] =
    BEGIN
    goodPages: PhysicalVolumeFormat.PageCount ← 0;
    pg: PhysicalVolumeFormat.PageCount;
```

```
  WordsToPages: PROCEDURE [words: PhysicalVolumeFormat.PageNumber]
    RETURNS [PhysicalVolumeFormat.PageNumber] = INLINE {
    RETURN[(words + Environment.wordsPerPage - 1)/Environment.wordsPerPage]};
  IsBadPage: PROCEDURE [pg: PhysicalVolumeFormat.PageNumber]
    RETURNS [BOOLEAN] =
    BEGIN
    i: CARDINAL;
    -- ASSUME that there are < = MAX[CARDINAL] bad pages.
    FOR i IN [0..Inline.LowHalf[p.badPageCount]) DO
      IF p.badPageList[i] = pg THEN RETURN[TRUE]; ENDLOOP;
    RETURN[FALSE];
    END;
  thisSv ← p.subVolumeCount;
  IF (isError ← thisSv >= PhysicalVolumeFormat.maxSubVols) THEN {
    error ← tooManySubvolumes; RETURN; };
  IF thisSv # 0 THEN
    thisSvStart ←
      p.subVolumes[thisSv - 1].pvPage + p.subVolumes[thisSv - 1].nPages + 1
  ELSE
    thisSvStart ←
      pvRootPage + WordsToPages[PhysicalVolumeFormat.descriptorSize];
  thisSvStart ← MAX[thisSvStart, minPVPageNumber];
  WHILE IsBadPage[thisSvStart] DO
    thisSvStart ← thisSvStart + 1;
    IF (size ← size - 1) = 0 THEN EXIT;
    ENDLOOP;
  WHILE IsBadPage[thisSvStart + size] DO
    IF (size ← size - 1) = 0 THEN EXIT; ENDLOOP;
  FOR pg ← thisSvStart, pg + 1 WHILE pg < thisSvStart + size DO
    IF ~IsBadPage[pg] THEN -- should really look for contig VAM
      IF (goodPages ← goodPages + 1) >= bareVolumeSize THEN RETURN;
    ENDLOOP;
  error ← subvolumeHasTooManyBadPages;
  isError ← TRUE;
  END;
proc2: PROCEDURE [p: PvHandle] =
  BEGIN
  p.subVolumes[thisSv] ← [lvID, size, 0, thisSvStart, size];
  p.subVolumeCount ← thisSv + 1;
  MarkerPage.CreateMarkerPage[p, activeVolume, thisSv];
  END;
IF name = NIL OR name.length = 0 THEN
  RETURN WITH ERROR PhysicalVolume.Error[nameRequired];
IF size < minVolumeSize THEN
  RETURN WITH ERROR PhysicalVolume.Error[pageCountTooSmallForVolume];
IF ~SubVolume.Find[LOOPHOLE[pvID], pvRootPage].success THEN
  RETURN WITH ERROR PhysicalVolume.Error[physicalVolumeUnknown];
VolumeImplInterface.AccessPhysicalVolumeRootPage[pvID, proc1];
-- Can not raise any errors due to check in previous statement
IF isError THEN RETURN WITH ERROR PhysicalVolume.Error[error];
IF thisSvStart + size >= DriveSize[pvID] THEN
  RETURN WITH ERROR PhysicalVolume.Error[insufficientSpace];
RegisterLogicalSubvolume[[lvID, size, 0, thisSvStart, size], pvID];
LabelTransfer.WriteLabels[
  [LOOPHOLE[lvID], lvID, local[
    FALSE, FALSE, 1, PilotFileTypes.tLogicalVolumeRootPage]],
  [lvRootPage, lvRootPage, lvRootPage + 1]];
Activate[lvID];
```



```
  activeVolume↑ ← [vID: lvID, type: type, volumeSize: size];
  LogicalVolume.SetFree[activeVolume, [System.GetUniversalID[]]];
  LogicalVolume.SetVam[activeVolume, [System.GetUniversalID[]]];
  LogicalVolume.SetVfm[activeVolume, [System.GetUniversalID[]]];
  activeVolume.labelLength ← MIN[name.length, LENGTH[activeVolume.label]];
  FOR i IN [0..activeVolume.labelLength) DO
    activeVolume.label[i] ← name[i]; ENDLOOP;
  SimpleSpace.ForceOut[logicalRootPageBuffer];
  EnterLV[lvID];
  LogicalVolumeCheck[lvID];
  IF type # nonPilot THEN
    LogicalVolume.ScavengeVolume[activeVolume, extraBuffer, TRUE];
  SimpleSpace.ForceOut[logicalRootPageBuffer];
  VolumeImplInterface.AccessPhysicalVolumeRootPage[pvID, proc2, readWrite];
  -- Can not raise any errors due to check in previous statement
  VolFileMap.Close[TRUE];
  VolAllocMap.Close[TRUE];
  activeVolume.changing ← FALSE;
  Deactivate[];
  RETURN[lvID];
  END;

LogicalVolumeErase: PUBLIC ENTRY PROCEDURE [lvID: Volume.ID] =
  BEGIN
  -- May be performed on an Open volume! (in which case, the volume is left Open)
  t: PilotFileTypes.PilotRootFileType;
  IF lvID = Volume.nullID OR ~LogicalVolumeFind[@lvID] THEN
    RETURN WITH ERROR Unknown[lvID];
  IF IsReadOnly[lvID] THEN RETURN WITH ERROR LogicalVolume.ReadOnlyVolume;
  Activate[lvID];
  activeVolume.changing ← TRUE;
  FOR t IN
    [FIRST[PilotFileTypes.PilotRootFileType]..LAST[
      PilotFileTypes.PilotRootFileType]] DO
    SELECT t FROM
      PilotFileTypes.tVolumeAllocationMap, PilotFileTypes.tVolumeFileMap,
        PilotFileTypes.tFreePage => LOOP;
      ENDCASE => activeVolume.rootFileID[t] ← File.nullID;
    ENDLOOP;
  activeVolume.clientRootFile ← File.nullCapability;
  activeVolume.bootingInfo ← LogicalVolume.nullBoot;
  SimpleSpace.ForceOut[logicalRootPageBuffer];
  LogicalVolume.ScavengeVolume[activeVolume, extraBuffer, TRUE];
  VolFileMap.Close[TRUE];
  VolAllocMap.Close[TRUE];
  activeVolume.changing ← FALSE;
  Deactivate[];
  END;

OpenLogicalVolume: PUBLIC PROCEDURE [pVID: POINTER TO READONLY Volume.ID]
  RETURNS [LogicalVolume.OpenStatus] =
  -- Use VolumeExtras.OpenVolume for read-only volume opening
  {RETURN[OpenLogicalVolumeInternal[pVID, FALSE]]; };

VolumeAccess: PUBLIC ENTRY PROCEDURE [
  pVID: POINTER TO READONLY Volume.ID, proc: LogicalVolume.VolumeAccessProc,
  modify: BOOLEAN ← FALSE] RETURNS [LogicalVolume.VolumeAccessStatus] =
  BEGIN
```

```
    open, beingScavenged, updateMarkers: BOOLEAN;
    IF pVID↑ = Volume.nullID OR ~LogicalVolumeFind[pVID] THEN
      RETURN[volumeUnknown];
    IF modify AND IsReadOnly[pVID↑] THEN RETURN[volumeReadOnly];
    Activate[pVID↑];
    -- Only allow access if the volume is open or being scavenged.
    -- The being scavenged option should only be exercised by the Scavenger.
    [open: open, beingScavenged: beingScavenged] ← LVGetStatus[pVID↑];
    IF ~(open OR beingScavenged) THEN {Deactivate[]; RETURN[volumeNotOpen]; };
    IF modify THEN
      BEGIN
      activeVolume.changing ← TRUE;
      SimpleSpace.ForceOut[logicalRootPageBuffer];
      END;
    updateMarkers ← proc[activeVolume];
    IF ~modify THEN RETURN[ok];
    IF updateMarkers THEN MarkerPage.UpdateLogicalMarkerPages[activeVolume];
    activeVolume.changing ← FALSE;
    Deactivate[];
    RETURN[ok];
    END;

--
-- VolumeImplInterface

CheckLogicalVolume: PUBLIC ENTRY PROCEDURE [vID: Volume.ID] = {
  LogicalVolumeCheck[vID]; Deactivate[]};

FindLogicalVolume: PUBLIC ENTRY PROCEDURE [vID: POINTER TO READONLY Volume.ID]
  RETURNS [BOOLEAN] = {RETURN[LogicalVolumeFind[vID]]};

GetLVStatus: PUBLIC ENTRY PROCEDURE [vID: Volume.ID]
  RETURNS [onLine, open: BOOLEAN] = {[open, onLine] ← LVGetStatus[vID]};

OpenInitialVolumes: PUBLIC PROCEDURE =
  BEGIN
  volumeID: Volume.ID;
  GetNextOnLineVolume: ENTRY PROCEDURE [volume: Volume.ID] RETURNS [Volume.ID] =
    BEGIN
    volumeFound: BOOLEAN ← volume = Volume.nullID;
    lv: LVTIndex;
    FOR lv IN LVTIndex DO
      IF volumeFound AND LVT[lv].lvID # Volume.nullID AND LVT[lv].onLine AND
        LVT[lv].type = debugClass THEN RETURN[LVT[lv].lvID];
      IF LVT[lv].lvID = volume THEN volumeFound ← TRUE;
      ENDLOOP;
    IF volumeFound THEN RETURN[Volume.nullID]
    ELSE RETURN WITH ERROR Volume.Unknown[volume];
    END;
  IF IsUtilityPilot[] THEN RETURN;
  FOR volumeID ← GetNextOnLineVolume[Volume.nullID], GetNextOnLineVolume[
    volumeID] UNTIL volumeID = Volume.nullID DO
    Volume.Open[
      volumeID !
      Volume.NeedsScavenging => {
        [] ← Scavenger.Scavenge[volumeID, volumeID, TRUE]; RETRY}];
    ENDLOOP;
  END;
```



```
  PinnedFileEnter: PUBLIC PROCEDURE [
    fd: FileInternal.Descriptor, grp: FileInternal.PageGroup] =
    BEGIN
    IF ~FileCache.GetFilePtrs[0, fd.fileID].success THEN {
      FileCache.SetFile[fd, TRUE]; FileCache.SetPageGroup[fd.fileID, grp, TRUE]};
    END;

  PinnedFileFlush: PUBLIC PROCEDURE [file: File.ID] = {FileCache.FlushFile[file]};

  RegisterLogicalSubvolume: PUBLIC PROCEDURE [
    sv: PhysicalVolumeFormat.SubVolumeDesc, pvID: PhysicalVolume.ID] =
    BEGIN
    IF ~SubVolume.Find[sv.lvID, sv.lvPage].success THEN
      SubVolume.OnLine[
        sv, SubVolume.Find[[LOOPHOLE[pvID]], pvRootPage].subVolume.channel];
    IF sv.lvPage = lvRootPage THEN
      -- Set the volume file to cover just the root page;
      VFileEnter[
        sv.lvID, LOOPHOLE[sv.lvID], lvRootPage, lvRootPage + 1,
        PilotFileTypes.tLogicalVolumeRootPage];
    END;

  RegisterVFiles: PUBLIC PROCEDURE [v: LvHandle] =
    BEGIN
    RegisterVFile: PROCEDURE [t: File.Type] = INLINE {
      VFileEnter[v.vID, v.rootFileID[t], 1, v.volumeSize, t]};
    RegisterVFile[PilotFileTypes.tFreePage];
    RegisterVFile[PilotFileTypes.tVolumeAllocationMap];
    RegisterVFile[PilotFileTypes.tVolumeFileMap];
    END;

  -- perhaps this should use GetLogicalRootPage???

  SignalVolumeAccess: PUBLIC PROCEDURE [
    pVID: POINTER TO READONLY Volume.ID, proc: LogicalVolume.VolumeAccessProc] =
    BEGIN
    SELECT LogicalVolume.VolumeAccess[pVID, proc, TRUE] FROM
      ok => RETURN;
      volumeUnknown => ERROR Volume.Unknown[pVID↑];
      volumeNotOpen => ERROR Volume.NotOpen[pVID↑];
      volumeReadOnly => ERROR LogicalVolume.ReadOnlyVolume;
      ENDCASE => ERROR VolImplError[illegalReturnCode];
    END;

  SubvolumeOffline: PUBLIC ENTRY PROCEDURE [lvID: Volume.ID, root: BOOLEAN] =
    BEGIN
    IF root THEN
      BEGIN
      Activate[lvID];
      UnregisterVFiles[activeVolume];
      Deactivate[flush];
      END;
    LVDecrementPieceCount[lvID];
    END;

  SubvolumeOnline: PUBLIC ENTRY PROCEDURE [lvID: Volume.ID, root: BOOLEAN] =
    BEGIN -- we ignore the root argument. It is provided for consistency with
    -- SubvolumeOffline where it is currently needed. Eventually we may
    -- use it here.
```

```
    IF LogicalVolumeFind[@lvID] THEN LVIncrementPieceCount[lvID]
    ELSE EnterLV[lvID];
    RETURN;
    END;

  UnregisterVFiles: PUBLIC PROCEDURE [v: LvHandle] =
    BEGIN
    UnregisterVFile: PROCEDURE [t: File.Type] = INLINE {
      PinnedFileFlush[v.rootFileID[t]]};
    UnregisterVFile[PilotFileTypes.tFreePage];
    UnregisterVFile[PilotFileTypes.tVolumeAllocationMap];
    UnregisterVFile[PilotFileTypes.tVolumeFileMap];
    END;

  VFileEnter: PUBLIC PROCEDURE [
    volume: Volume.ID, file: File.ID, fileAndVolPage: File.PageNumber,
    nextPage: VolumeInternal.PageNumber, type: File.Type] =
    BEGIN
    PinnedFileEnter[
      [file, volume, local[FALSE, FALSE, nextPage, type]],
      [fileAndVolPage, fileAndVolPage, nextPage]];
    END;

  --
  -- Logical volume table management (private)

  EnterLV: INTERNAL PROCEDURE [vID: Volume.ID] =
    BEGIN
    FOR lv: LVTIndex IN LVTIndex DO
      IF LVT[lv].lvID = Volume.nullID THEN
        BEGIN
        -- readOnly must be FALSE in the following so that scavenging a volume
        -- at Open time works correctly.  This may have to be changed if Open
        -- ever takes an access mode argument.
        LVT[lv] ←
          [lvID: vID, pieceCount: 1, beingScavenged: FALSE, open: FALSE,
            onLine: FALSE, readOnly: FALSE, type: nonPilot];
        RETURN
        END;
      ENDLOOP;
    ERROR VolImplError[lvTableFull];
    END;

  LogicalVolumeFind: INTERNAL PROCEDURE [vID: POINTER TO READONLY Volume.ID]
    RETURNS [found: BOOLEAN] =
    BEGIN
    FOR lv: LVTIndex IN LVTIndex DO
      IF LVT[lv].lvID = vID↑ THEN RETURN[LVT[lv].onLine]; ENDLOOP;
    RETURN[FALSE]
    END;

  LogicalVolumeOffLine: INTERNAL PROCEDURE [vID: Volume.ID] =
    BEGIN
    FOR lv: LVTIndex IN LVTIndex DO
      IF LVT[lv].lvID = vID THEN
        BEGIN -- Move remaining entries to front so enumeration works correctly
        FOR lvMove: LVTIndex IN [lv + 1..LAST[LVTIndex]) DO
          LVT[lvMove - 1] ← LVT[lvMove]; ENDLOOP;
        LVT[LAST[LVTIndex]] ← nullVolumeStatus;
```



```
        RETURN;
        END;
      ENDLOOP;
    ERROR VolImplError[notFoundInLVT];
    END;

  LVDecrementPieceCount: INTERNAL PROCEDURE [vID: Volume.ID] =
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    p.pieceCount ← p.pieceCount - 1;
    IF p.pieceCount = 0 THEN LogicalVolumeOffLine[vID];
    END;

  LVGetEntryPointer: INTERNAL PROCEDURE [vID: Volume.ID]
    RETURNS [POINTER TO VolumeStatus] =
    BEGIN
    lv: LVTIndex;
    FOR lv IN LVTIndex DO IF LVT[lv].lvID = vID THEN RETURN[@LVT[lv]] ENDLOOP;
    ERROR VolImplError[notFoundInLVT];
    END;

  LVGetStatus: INTERNAL PROCEDURE [vID: Volume.ID]
    RETURNS [beingScavenged, open, onLine, readOnly: BOOLEAN, type: Volume.Type] =
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    RETURN[p.beingScavenged, p.open, p.onLine, p.readOnly, p.type]
    END;

  LVIncrementPieceCount: INTERNAL PROCEDURE [vID: Volume.ID] = INLINE
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    p.pieceCount ← p.pieceCount + 1;
    END;

  LVSetOnLine: INTERNAL PROCEDURE [vID: Volume.ID, onLine: BOOLEAN] = INLINE
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    p.onLine ← onLine;
    END;

  LVSetOpen: INTERNAL PROCEDURE [vID: Volume.ID, open: BOOLEAN] = INLINE
    BEGIN p: POINTER TO VolumeStatus = LVGetEntryPointer[vID]; p.open ← open; END;

  LVSetReadOnly: INTERNAL PROCEDURE [vID: Volume.ID, readOnly: BOOLEAN] = INLINE
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    p.readOnly ← readOnly;
    END;

  LVSetScavenged: INTERNAL PROCEDURE [vID: Volume.ID, beingScavenged: BOOLEAN] =
    INLINE
    BEGIN
    p: POINTER TO VolumeStatus = LVGetEntryPointer[vID];
    p.beingScavenged ← beingScavenged;
    END;

  LVSetType: INTERNAL PROCEDURE [vID: Volume.ID, type: Volume.Type] = INLINE
    BEGIN p: POINTER TO VolumeStatus = LVGetEntryPointer[vID]; p.type ← type; END;
```

```
--
-- Various utility procedures

DriveSize: PROCEDURE [pvID: PhysicalVolume.ID]
  RETURNS [DiskChannel.DiskPageCount] =
  BEGIN
  RETURN[
    DiskChannel.GetDriveAttributes[
      MarkerPage.Find[[physicalID[pvID]]].drive].nPages];
  END;

IsReadOnly: INTERNAL PROCEDURE [vID: Volume.ID] RETURNS [BOOLEAN] = INLINE {
  RETURN[
    (~IsUtilityPilot[]) AND
      ((LVGetStatus[vID].type > debugClass) OR LVGetStatus[vID].readOnly)]};

IsUtilityPilot: PROCEDURE RETURNS [BOOLEAN] = INLINE {
  RETURN[PilotSwitches.switches.u = down]};

-- Similar to VolumeAccess, but only requires the Root Page Subvolume to be around.
-- Intended for use by read-only procedures.

GetLogicalRootPage: ENTRY PROCEDURE [
  pVID: POINTER TO READONLY Volume.ID, proc: PROCEDURE [LvHandle]] =
  BEGIN
  IF pVID↑ = Volume.nullID OR ~SubVolume.Find[pVID↑, lvRootPage].success THEN
    RETURN WITH ERROR Unknown[pVID↑];
  Activate[pVID↑];
  IF activeVolume.seal # LogicalVolume.seal THEN {
    Deactivate[flush]; RETURN WITH ERROR Unknown[pVID↑]};
  proc[activeVolume];
  END;

-- Provides access to volume files of one volume at a time

Activate: INTERNAL PROCEDURE [vID: Volume.ID] =
  BEGIN
  IF activeVID # vID THEN
    BEGIN
    Deactivate[flush];
    SimpleSpace.Map[
      logicalRootPageBuffer,
      [[LOOPHOLE[vID], FileInternal.maxPermissions], lvRootPage], FALSE];
    activeVID ← vID;
    END;
  END;

-- Deactivates the active volume.  If the volume is Open, clear the changing bit.

Deactivate: INTERNAL PROCEDURE [mode: {forceout, flush} ← forceout] =
  BEGIN
  IF activeVID = Volume.nullID THEN RETURN;
  VolFileMap.Close[mode = flush];
  VolAllocMap.Close[mode = flush];
  IF mode = flush THEN
    BEGIN
    SimpleSpace.Unmap[logicalRootPageBuffer];
    activeVID ← Volume.nullID;
    END
```



```
  ELSE SimpleSpace.ForceOut[logicalRootPageBuffer];
  END;

-- When LogicalVolumeCheck is called, the Subvolume is Online; if the LvRoot subvolume
-- is online, the LvRoot page file cache entries are set.  If the Root SV is accessible
-- and of the right type, Vam, free and Vfm files are registered.
-- Called from CreateLogicalVolume, CheckLogicalVolume

LogicalVolumeCheck: INTERNAL PROCEDURE [vID: Volume.ID] =
  BEGIN
  good: BOOLEAN;
  size: Volume.PageCount;
  svH: SubVolume.Handle;
  [good, svH] ← SubVolume.Find[vID, lvRootPage];
  IF good THEN good ← ReadAndCheckLogicalRootLabel[vID];
  IF good THEN
    BEGIN
    Activate[vID];
    good ← activeVolume.seal = LogicalVolume.seal AND activeVolume.version =
      LogicalVolume.currentVersion AND vID = activeVolume.vID;
    IF good THEN
      BEGIN
      good ← LogicalVolumeLike[vID];
      LVSetType[vID, activeVolume.type];
      END;
    size ← activeVolume.volumeSize;
    -- ~good volumes can be debuggers
    IF LogicalVolumeDebuggerCheck[activeVolume] THEN
      -- Set the debugger boot pointers back in Pilot Control
      BEGIN
      IF pLVBootFiles ~= NIL THEN pLVBootFiles↑ ← activeVolume.bootingInfo;
      [deviceType: debuggerDeviceType↑, deviceOrdinal: debuggerDeviceOrdinal↑] ←
        DiskChannel.GetDriveAttributes[
        DiskChannel.GetAttributes[svH.channel].drive];
      debugSearchState ← done;
      END;
    END;
  WHILE good AND size # svH.lvPage + svH.nPages DO
    [good, svH] ← SubVolume.Find[vID, svH.lvPage + svH.nPages]; ENDLOOP;
  IF good THEN
    BEGIN
    -- The theory is that we don't have to check to see if the volume is already on line
    -- since this subvolume is new
    RegisterVFiles[activeVolume];
    LVSetOnLine[vID, TRUE];
    END
  -- Deactivate is done in caller

  END;

-- This may have to get smarter when we have boot messages

LogicalVolumeDebuggerCheck: PROCEDURE [vol: LvHandle]
  RETURNS [isMyDebugger: BOOLEAN] =
  BEGIN
  IF debugSearchState # looking OR vol.seal # LogicalVolume.seal OR vol.version
    <= 1 OR vol.bootingInfo[debugger] = Boot.nullDiskFileID OR vol.bootingInfo[
    debuggee] = Boot.nullDiskFileID THEN RETURN[FALSE];
  SELECT debugClass FROM
```

```
    nonPilot, debuggerDebugger => RETURN[FALSE]; -- these guys have no debugger

    normal => RETURN[vol.type = debugger];
    debugger => RETURN[vol.type = debuggerDebugger];
    ENDCASE => ERROR;
  END;

LogicalVolumeLike: INTERNAL PROCEDURE [vID: Volume.ID] RETURNS [good: BOOLEAN] =
  BEGIN
  SELECT TRUE FROM
    IsUtilityPilot[] => good ← activeVolume.type # nonPilot;
    -- don't put non-Pilot volumes online

    debugSearchState = tooSoonToLook =>
      -- PhysicalVolumeOnLine has arranged that the first subvolume we find will be
      -- the system volume
      BEGIN
      systemID ← vID;
      debugClass ← activeVolume.type;
      debugSearchState ← looking;
      good ← TRUE;
      END;
    ENDCASE => good ← activeVolume.type # nonPilot;
  END;

OpenLogicalVolumeInternal: ENTRY PROCEDURE [
  pVID: POINTER TO READONLY Volume.ID, readOnly: BOOLEAN]
  RETURNS [LogicalVolume.OpenStatus] =
  BEGIN
  IF pVID↑ = Volume.nullID OR ~LogicalVolumeFind[pVID] THEN RETURN[Unknown];
  WHILE LVGetStatus[pVID↑].beingScavenged DO WAIT scavengingComplete; ENDLOOP;
  IF LVGetStatus[pVID↑].open THEN RETURN[wasOpen];
  Activate[pVID↑];
  LVSetOpen[pVID↑, TRUE];
  LVSetReadOnly[pVID↑, readOnly];
  RegisterVFiles[activeVolume];
  IF activeVolume.changing THEN
    BEGIN
    LVSetOpen[pVID↑, FALSE];
    Deactivate[];
    RETURN[VolumeNeedsScavenging]
    END;
  Deactivate[];
  RETURN[ok];
  END;

ReadAndCheckLogicalRootLabel: PROCEDURE [lvID: Volume.ID] RETURNS [BOOLEAN] =
  BEGIN
  label: PilotDisk.Label ← LabelTransfer.ReadLabel[
    [LOOPHOLE[lvID], lvID, local[
      FALSE, FALSE, 1, PilotFileTypes.tLogicalVolumeRootPage]], lvRootPage,
    lvRootPage];
  BEGIN OPEN label;
  -- could also check pad2...
  RETURN[
    fileID = LOOPHOLE[lvID] AND PilotDisk.GetLabelFilePage[@label] = lvRootPage
      AND ~immutable AND ~temporary AND ~zeroSize AND pad1 = 0 AND
      PilotDisk.GetLabelType[@label] = PilotFileTypes.tLogicalVolumeRootPage];
  END;
```



```
  END;

Process.InitializeCondition[@scavengingComplete, LAST[Process.Ticks]];
END.
```

(For earlier log entries see Pilot 3.0 archive version.)
January 25, 1980 2:54 PM McJones Delete InstallBootVolume; add Get/SetPhysicalVolumeBoo
***tFiles, GetContainingPhysicalVolume; add minPVPageNumber to CreateLogicalVolume*
February 3, 1980 3:57 PM McJones Add device types to module parameters, results
March 7, 1980 11:52 PM Forrest re-arranged opens to call up into FileImpl and from there to her
***e (see comments in LogicalVolume). Made Openning smarter about always setting Open bit. C*
***hanged FileVolumeAccess to not pass on parameters deleted from FileAccess Procs.*
March 11, 1980 10:37 AM Forrest Added display of cDeleteTemps.
June 5, 1980 3:13 PM Luniewski PhysicalVolume => PhysicalVolumeFormat. Added removable
***volume support including a new logical volume table format, a GetHints operation for PhysicalV*
***olumeImpl to use, the Volume.GetStatus operation and recognition of the ability to unpin file ca*
***che entries. Modified to move all physical volume implementation stuff into PhysicalVolumeImp*
***l.*
June 19, 1980 2:47 PM Luniewski Added BeginScavenging and EndScavenging.
July 17, 1980 5:13 PM Forrest Added second arg to GetNext procedure; added GetLabelType..
August 1, 1980 10:00 AM Luniewski Made less conservative user of Deactivate (performance im
***provement). Mods to permit read-up and read-write down of volumes. Hooks to permit some o*
***perations to explicitly specify read-only/read-write access.*
August 15, 1980 2:54 PM McJones VolumeSet =>TypeSet
September 3, 1980 4:40 PM Luniewski Fix EnterLV to permit scavenging at Open time. Clear bo
***otingInfo on erasing a logical volume.*
September 18, 1980 7:22 PM Luniewski Changes for new logical volume format. Open volumes
***of higher type than system volume without deleting temps.*
September 22, 1980 12:02 PM Luniewski Do delete temps on volumes of type <= system volum
***e.*
October 9, 1980 11:15 AM Luniewski Implement VolumeExtras.OpenVolume
October 21, 1980 1:54 PM Jose Increase maxLVs from 8 to 12

```
-- FileCacheImpl.mesa (last edited by: Yokota on: September 17, 1980 4:11 PM)

DIRECTORY
  Environment USING [Base, wordsPerPage],
  File USING [ID, PageNumber],
  FileCache USING [PinnedAction],
  FileInternal USING [Descriptor, FilePtr, PageGroup],
  FilerPrograms USING [],
  Process USING [DisableAborts],
  ResidentHeap USING [first64K, MakeNode],
  System USING [UniversalID],
  Volume USING [ID],
  Zone USING [nil];

FileCacheImpl: MONITOR
  IMPORTS Process, ResidentHeap, Zone EXPORTS FileCache, FilerPrograms =
  BEGIN OPEN FileInternal;
  -- General cache mechanism --
  Base: TYPE = Environment.Base;
  base: Base = ResidentHeap.first64K; -- base pointer for CEptr's
  Cache: TYPE = POINTER TO CacheRecord;
  CacheRecord: TYPE = RECORD [
    cacheType: CacheType,
    mru: CEptr, -- most recently used entry (head of lru chain)
    free: CEptr, -- head of free chain
    CleanUp: CacheCleanUpProc]; -- proc to cleanup when an entry is replaced
  CacheType: TYPE = {file, pageGroup};
  CacheEntry: TYPE = RECORD [
    -- carefully arranged to not waste bits
    next: CEptr, -- next entry in lru chain
    pinned: BOOLEAN, -- TRUE if this cache entry is pinned
    refCnt: [0..37777B), -- number of readlocks set by FindCacheEntry
    body:
      SELECT COMPUTED CacheType FROM
        file => [fd: Descriptor, pgList: PageGroupCEptr],
        pageGroup => [group: PageGroup, fileCacheEntry: FileCEptr],
        ENDCASE];
  CEptr: TYPE = Base RELATIVE POINTER TO CacheEntry;
  FileCEptr: TYPE = Base RELATIVE POINTER TO file CacheEntry;
  PageGroupCEptr: TYPE = Base RELATIVE POINTER TO pageGroup CacheEntry;
  nilCEptr: CEptr = Zone.nil;
  nilFileCEptr: FileCEptr = LOOPHOLE[nilCEptr];
  nilPageGroupCEptr: PageGroupCEptr = LOOPHOLE[nilCEptr];
  CacheCleanUpProc: TYPE = PROCEDURE [cePtr: CEptr];
  NoCleanUp: CacheCleanUpProc; -- null case indicator (unbound control link)
  returned: CONDITION;
  -- Key for cache searches --
  CacheKey: TYPE = RECORD [
    SELECT OVERLAID CacheType FROM
      file => [fileID: File.ID],
      pageGroup => [fileCacheEntry: FileCEptr, filePage: File.PageNumber],
      ENDCASE];
  -- The following code calculates the number of file cache entries to initially create. As the initial
  **cache is allocated as one chunk out of the resident heap, it is best if the initial file cache use as
  **close to an integral number of pages as possible.
  PageSize: CARDINAL = Environment.wordsPerPage; -- for conciseness below
  FCESize: CARDINAL = SIZE[file CacheEntry];
  PGESize: CARDINAL = SIZE[pageGroup CacheEntry];
```



```
  ResidentHeapImplOverhead: CARDINAL = 1;
  -- Must be at least as large as the actual overhead imposed by ResidentHeapImpl
  NumPages: CARDINAL =
    (FCacheMinimumSize*FCESize + PGCacheMinimumSize*PGESize +
      ResidentHeapImplOverhead + PageSize - 1)/PageSize;
  -- number of pages to allocate initially
  Slop: CARDINAL =
    NumPages*PageSize - ResidentHeapImplOverhead - FCacheMinimumSize*FCESize -
      PGCacheMinimumSize*PGESize;
  -- extra space to be allocated as equally as possible to file and page group cache entries
  FirstExtraFCEs: CARDINAL = (Slop/2)/FCESize;
  FirstExtraPGEs: CARDINAL = (Slop/2)/PGESize;
  LastSlop: CARDINAL = Slop - FirstExtraFCEs*FCESize - FirstExtraPGEs*PGESize;
  ExtraFCEs: CARDINAL =
    IF FCESize > PGESize THEN FirstExtraFCEs
    ELSE FirstExtraFCEs + LastSlop/FCESize;
  ExtraPGEs: CARDINAL =
    IF PGESize >= FCESize THEN FirstExtraPGEs
    ELSE FirstExtraPGEs + LastSlop/PGESize;
  -- File descriptor cache --
  FCache: Cache;
  FCacheRecord: CacheRecord;
  FCacheIndex: TYPE = [0..FCacheSize);
  -- There are 4 pinned entries for each logical volume of the correct type, 1 for physical volumes,
  **and 1 for logical volumes of the "wrong" type.
  FCacheMinimumSize: CARDINAL = 40; -- must be 2 or greater
  FCacheSize: CARDINAL = FCacheMinimumSize + ExtraFCEs;
  FCacheCleanUp: INTERNAL CacheCleanUpProc =
    BEGIN -- flush corresponding entries from page group cache
    WHILE RemoveCacheEntry[PGCache, cePtr] DO NULL ENDLOOP
    END;
  -- Page Group cache --

  PGCache: Cache;
  PGCacheRecord: CacheRecord;
  PGCacheIndex: TYPE = [0..PGCacheSize);
  PGCacheMinimumSize: CARDINAL = 40; -- must be 2 or greater
  PGCacheSize: CARDINAL = PGCacheMinimumSize + ExtraPGEs;
  GetFilePtrs: PUBLIC ENTRY PROCEDURE [count: CARDINAL, fileID: File.ID]
    RETURNS [success: BOOLEAN, fD: FileInternal.FilePtr] =
    BEGIN
    fEntry: CEptr;
    [success, fEntry] ← FindCacheEntry[
      get, count, FCache, CacheKey[file[fileID]]];
    IF success THEN WITH base[fEntry] SELECT file FROM file => fD ← @fd; ENDCASE;
    END;

  ReturnFilePtrs: PUBLIC ENTRY PROCEDURE [
    count: CARDINAL, fD: FileInternal.FilePtr] =
    BEGIN
    [] ← FindCacheEntry[return, count, FCache, CacheKey[file[fD.fileID]]];
    END;

  SetFile: PUBLIC ENTRY PROCEDURE [fd: Descriptor, pinned: BOOLEAN] =
    BEGIN
    SetCacheEntry[
      FCache, CacheEntry[nilCEptr, pinned, 0, file[fd, nilPageGroupCEptr]]]
    END;
```

```
FlushFile: PUBLIC ENTRY PROCEDURE [fileID: File.ID] =
  BEGIN
  found: BOOLEAN;
  fEntry: CEptr;
  [found, fEntry] ← FindCacheEntry[locate, 0, FCache, CacheKey[file[fileID]]];
  IF found THEN [] ← RemoveCacheEntry[FCache, fEntry];
  END;

GetPageGroup: PUBLIC ENTRY PROCEDURE [
  fileID: File.ID, filePage: File.PageNumber]
  RETURNS [success: BOOLEAN, pg: PageGroup] =
  BEGIN
  fEntry, pgEntry: CEptr;
  [success, fEntry] ← FindCacheEntry[locate, 0, FCache, CacheKey[file[fileID]]];
  IF ~success THEN RETURN;
  [success, pgEntry] ← FindCacheEntry[
    locate, 0, PGCache, CacheKey[pageGroup[LOOPHOLE[fEntry], filePage]]];
  IF success THEN
    WITH base[pgEntry] SELECT pageGroup FROM pageGroup => pg ← group; ENDCASE;
  END;

SetPageGroup: PUBLIC ENTRY PROCEDURE [
  fileID: File.ID, group: PageGroup, pinned: BOOLEAN] =
  BEGIN
  success: BOOLEAN;
  fEntry: CEptr;
  [success, fEntry] ← FindCacheEntry[locate, 0, FCache, CacheKey[file[fileID]]];
  IF ~success THEN ERROR;
  SetCacheEntry[
    PGCache, CacheEntry[
    nilCEptr, pinned, 0, pageGroup[group, LOOPHOLE[fEntry]]]];
  END;

FlushFilesOnVolume: PUBLIC ENTRY PROCEDURE [
  lvID: Volume.ID, pin: FileCache.PinnedAction] =
  BEGIN
  foundAny: BOOLEAN ← TRUE;
  WHILE foundAny DO
    foundAny ← FALSE;
    FOR current: CEptr ← FCache.mru, base[current].next WHILE current ~=
      nilCEptr DO
      WITH fileEntry: base[current] SELECT FCache.cacheType FROM
        file =>
          IF UIDEqual[@fileEntry.fd.volumeID, LONG[@lvID]] THEN
            IF fileEntry.pinned AND pin = error THEN ERROR
            ELSE
              IF NOT (fileEntry.pinned AND pin = keep) THEN
                BEGIN
                foundAny ← TRUE;
                [] ← RemoveCacheEntry[FCache, current]
                END;
        ENDCASE => ERROR;
      ENDLOOP;
    ENDLOOP;
  END;
-- Utility routines

IsMatch: INTERNAL PROCEDURE [cacheType: CacheType, key: CacheKey, entry: CEptr]
  RETURNS [BOOLEAN] = INLINE
```



```
  BEGIN
  RETURN[
    WITH base[entry] SELECT cacheType FROM
      file => UIDEqual[LONG[@key.fileID], @fd.fileID],
      pageGroup => key.fileCacheEntry = fileCacheEntry AND key.filePage IN
        [group.filePage..group.nextFilePage),
      ENDCASE => -- never happens--FALSE];
  END;

UIDEqual: INTERNAL PROCEDURE [a, b: LONG POINTER] RETURNS [BOOLEAN] =
  -- This procedure actually only deals with UniversalID's but is declared in this manner because
**this procedure is invoked with UID's packaged into a record that are inconvenient to unpack.
  INLINE
  BEGIN
  p1: LONG POINTER TO WORD = LOOPHOLE[a];
  p2: LONG POINTER TO WORD = LOOPHOLE[b];
  FOR i: CARDINAL IN [0..SIZE[System.UniversalID]) DO
    IF (p1 + i)↑ ~= (p2 + i)↑ THEN RETURN[FALSE]; ENDLOOP;
  RETURN[TRUE];
  END;

GetMru: PROCEDURE [cacheType: CacheType, key: CacheKey]
  RETURNS [LONG POINTER TO CEptr] = INLINE
  BEGIN
  RETURN[
    SELECT cacheType FROM
      file => @FCache.mru,
      pageGroup => LOOPHOLE[@(base[key.fileCacheEntry]).pgList]
      ENDCASE => ERROR]
  END;
  -- FindCacheEntry searches for an entry in the indicated cache having the indicated key, and ret
**urns the specified number of pointers to it (i.e. a pointer that may be copied that many times). If
**the pointers will be passed outside the monitor, op = get, and a later matching call with op = re
**turn must occur when the client discards the pointers; these adjust the refCnt of the entry appro
**priately. If only the contents of the entry are passed outside the monitor, op = locate, and the re
**fCnt is not incremented.

FindCacheEntry: INTERNAL PROCEDURE [
  op: {get, return, locate}, count: CARDINAL, cache: Cache, key: CacheKey]
  RETURNS [success: BOOLEAN, ceptr: CEptr] =
  BEGIN OPEN cache;
  current: CEptr; -- start search with most-recently-used
  mru: LONG POINTER TO CEptr;
  previous: CEptr ← nilCEptr;
  mru ← GetMru[cacheType, key];
  current ← mru↑;
  WHILE current ~= nilCEptr DO
    OPEN base[current];
    IF IsMatch[cacheType, key, current] THEN -- found the matching cache entry
      BEGIN
      IF previous ~= nilCEptr THEN -- promote it to mru
        {base[previous].next ← next; next ← mru↑; mru↑ ← current};
      SELECT op FROM
        get => refCnt ← refCnt + count;
        return => IF (refCnt ← refCnt - count) = 0 THEN BROADCAST returned;
        ENDCASE;
      RETURN[TRUE, current]
      END;
    previous ← current;
```

```
    current ← next; -- advance down LRU chain

    ENDLOOP;
  RETURN[FALSE, nilCEptr]; -- search failed

  END;

SetCacheEntry: INTERNAL PROCEDURE [cache: Cache, newEntry: CacheEntry] =
  BEGIN OPEN cache;
  entry, lastPtr, previous: CEptr;
  key: CacheKey;
  spliceOutMru, spliceInMru: LONG POINTER TO CEptr;
  WITH newEntry SELECT cacheType FROM
    file => key ← CacheKey[file[fd.fileID]];
    pageGroup => key ← CacheKey[pageGroup[fileCacheEntry, group.filePage]];
    ENDCASE;
  previous ← nilCEptr;
  spliceInMru ← GetMru[cacheType, key];
  spliceOutMru ← NIL;
  FOR current: CEptr ← spliceInMru↑, base[current].next WHILE current ~=
    nilCEptr DO
    IF IsMatch[cacheType, key, current] THEN
      BEGIN
      base[current].refCnt ← base[current].refCnt + newEntry.refCnt;
      base[current].pinned ← base[current].pinned OR newEntry.pinned;
      IF previous ~= nilCEptr THEN
        BEGIN
        -- must move current to mru (ELSE clause would be to move mru (= current) to mru - a n
**o-op).
        base[previous].next ← base[current].next;
        base[current].next ← spliceInMru↑;
        spliceInMru↑ ← current;
        END;
      RETURN;
      END;
    previous ← current;
    ENDLOOP;
  IF free ~= nilCEptr THEN -- Steal a free entry if they exist
    BEGIN entry ← free; free ← base[free].next; END
  ELSE
    BEGIN
    -- Must find the LRU replacable entry
    entry ← lastPtr ← previous ← nilCEptr;
    SELECT cacheType FROM
      file =>
        FOR fPtr: FileCEptr ← LOOPHOLE[(spliceOutMru ← spliceInMru)↑],
          LOOPHOLE[base[fPtr].next] WHILE fPtr ~= nilCEptr DO
          IF base[fPtr].refCnt = 0 AND ~base[fPtr].pinned THEN {
            previous ← lastPtr; entry ← fPtr};
          lastPtr ← fPtr;
          ENDLOOP;
      pageGroup =>
        FOR fPtr: FileCEptr ← LOOPHOLE[FCache.mru], LOOPHOLE[base[fPtr].next]
          WHILE fPtr ~= nilCEptr DO
          lastPtr ← nilCEptr;
          FOR pgPtr: PageGroupCEptr ← base[fPtr].pgList, LOOPHOLE[base[
            pgPtr].next] WHILE pgPtr ~= nilCEptr DO
            IF base[pgPtr].refCnt = 0 AND ~base[pgPtr].pinned THEN {
```



```
              spliceOutMru ← LOOPHOLE[@(base[fPtr]).pgList];
              previous ← lastPtr;
              entry ← pgPtr};
            lastPtr ← pgPtr;
            ENDLOOP;
          ENDLOOP;
      ENDCASE => ERROR;
    IF entry = nilCEptr THEN
      SELECT cacheType FROM -- make a new cache entry
        file => entry ← ResidentHeap.MakeNode[SIZE[file CacheEntry]].node;
        pageGroup =>
          entry ← ResidentHeap.MakeNode[SIZE[pageGroup CacheEntry]].node;
        ENDCASE => ERROR
    ELSE
      BEGIN -- remove the LRU entry (= entry)
      IF CleanUp # NoCleanUp THEN CleanUp[entry]; -- clean up for removal
      IF previous ~= nilCEptr THEN base[previous].next ← base[entry].next
        -- remove last eligible entry from mru chain

      ELSE spliceOutMru↑ ← base[entry].next;
      -- There is no previous so entry must currently be MRU, splice it out of list.

      END;
    END;
  WITH newEntry SELECT cacheType FROM
    file =>
      base[entry] ←
        [next: spliceInMru↑, pinned: newEntry.pinned, refCnt: newEntry.refCnt,
          body: file[fd: fd, pgList: nilPageGroupCEptr]];
    pageGroup =>
      base[entry] ←
        [next: spliceInMru↑, pinned: newEntry.pinned, refCnt: newEntry.refCnt,
          body: pageGroup[group: group, fileCacheEntry: fileCacheEntry]];
    ENDCASE;
  spliceInMru↑ ← entry; -- Complete splicing in entry as MRU

  END;

RemoveCacheEntry: INTERNAL PROCEDURE [cache: Cache, fce: CEptr]
  RETURNS [success: BOOLEAN] =
  BEGIN OPEN cache;
  current, previous: CEptr;
  mru: LONG POINTER TO CEptr ←
    IF cacheType = file THEN @FCache.mru
    ELSE LOOPHOLE[@(base[LOOPHOLE[fce, FileCEptr]]).pgList];
  DO
    current ← mru↑; -- start search with most-recently-used
    WHILE current ~= nilCEptr DO
      IF
        WITH base[current] SELECT cacheType FROM
          file => current = fce,
          pageGroup => fileCacheEntry = fce,
          ENDCASE => FALSE -- (never happens)
        THEN EXIT;
      previous ← current;
      current ← base[current].next; -- advance down LRU chain
```

```
      REPEAT FINISHED => RETURN[FALSE]; -- search failed

      ENDLOOP;
    IF base[current].refCnt = 0 THEN EXIT;
    WAIT returned; -- entry busy: try again when activity subsides

    ENDLOOP;
  IF CleanUp ~= NoCleanUp THEN CleanUp[current]; -- clean up entry for removal
  IF current = mru THEN -- remove current from lru chain
    THEN mru ← base[current].next
  ELSE base[previous].next ← base[current].next;
  base[current].next ← free;
  free ← current; -- put on free chain
  RETURN[TRUE];
  END;

Initialize: PROCEDURE =
  -- Initialization of file caches --
  BEGIN
  FCacheArray: LONG DESCRIPTOR FOR ARRAY FCacheIndex OF file CacheEntry;
  PGCacheArray: LONG DESCRIPTOR FOR ARRAY PGCacheIndex OF pageGroup CacheEntry;
  node: CEptr;
  i: CARDINAL;
  Process.DisableAborts[@returned];
  node ← ResidentHeap.MakeNode[NumPages*PageSize, a1].node;
  -- storage for the initial caches
  FCache ← @FCacheRecord;
  FCache.cacheType ← file;
  FCache.mru ← nilCEptr;
  FCache.CleanUp ← FCacheCleanUp;
  FCache.free ← node;
  FCacheArray ← DESCRIPTOR[@base[FCache.free], FCacheSize];
  FOR i IN FCacheIndex DO
    -- set up free chain
    FCacheArray[i].next ←
      IF i = LAST[FCacheIndex] THEN nilCEptr ELSE (node ← node + FCESize)
    ENDLOOP;
  node ← node + FCESize;
  -- start the page group cache after the last file cache entry
  PGCache ← @PGCacheRecord;
  PGCache.cacheType ← pageGroup;
  PGCache.mru ← nilCEptr;
  PGCache.CleanUp ← NoCleanUp;
  PGCache.free ← node;
  PGCacheArray ← DESCRIPTOR[@base[PGCache.free], PGCacheSize];
  FOR i IN PGCacheIndex DO
    -- set up free chain
    PGCacheArray[i].next ←
      IF i = LAST[PGCacheIndex] THEN nilCEptr ELSE (node ← node + PGESize)
    ENDLOOP;
  END;

  Initialize[];
  END.
```

LOG

Time: April 20, 1978 3:09 PM By: Redell Action: Created file
Time: May 4, 1978 1:32 PM By: Redell Action: bug: get free entry => refCnt ← 0
Time: July 25, 1978 1:31 PM By: Redell Action: clean crash if cache fills up with pinned entries.

Time: September 11, 1978 5:42 PM By: Redell Action: Fixed flush to wait if I/O in progress.
***Re-did lost edits to expand cache size to 32 entries(!)*
Time: August 29, 1979 4:49 PM By: Ladner Action: installed cache instrumentation
Time: February 14, 1980 7:39 PM By: Gobbel Action: Fixed bug in UtilityPilot m
***ode operation: forgot to zero refcnt when getting a new entry*
Time: May 16, 1980 9:39 AM By: Luniewski Action: FrameOps -> Frame.
Time: May 23, 1980 12:08 PM By: Luniewski Action: Added FlushFilesOnVolume and Cac
***heEntry pinned field. Modified to use ResidentHeap for cache storage and relative pointers for i*
***nternal cache pointers.*
Time: September 3, 1980 2:07 PM By: Luniewski Action: Speeded up algorithm by
***adding a fast pre-search to SetCacheEntry, chaining page group entries off of the correspondin*
***g file cache entry and adding an internal, INLINE, UID comparer program. Deleted performance*
***monitoring code. Diabled aborts on condition variable.*
Time: September 17, 1980 4:10 PM By: Yokota Action: mru is converted to spliceInMru and splic
***eOutMru.*

```
-- Filer>FilerControl.mesa (last edited by Knutsen on April 9, 1980 3:35 PM)

DIRECTORY
  FilerPrograms: FROM "FilerPrograms",
  StoragePrograms: FROM "StoragePrograms";

FilerControl: PROGRAM IMPORTS FilerPrograms EXPORTS StoragePrograms =
  BEGIN
  InitializeFiler: PUBLIC PROCEDURE =
    BEGIN OPEN FilerPrograms;
    START FileCacheImpl;
    START FilerExceptionImpl;
    START FilerTransferImpl;
    START FileTaskImpl;
    --START RemotePageTransferImpl;
    START SubVolumeImpl;
    END;

  END.
LOG
Time: February 28, 1979 11:39 AM      By: Redell Action: Created file from old InitializeFPT.me
**sa
Time: January 30, 1980 4:52 PM        By: Gobbel      Action: Deactivate RemotePageTr
**ansferImpl
Time: January 30, 1980 4:52 PM        By: Knutsen     Action: Convert FilerControl: PRO
**GRAM to InitializeFiler: PROCEDURE.
```

```
-- FilerExceptionImpl.mesa (last edited by: Forrest on: April 28, 1980 9:29 AM) --
DIRECTORY
  FilePageTransfer: FROM "FilePageTransfer" USING [Request],
  FilerException: FROM "FilerException",
  FilerPrograms: FROM "FilerPrograms",
  Frame: FROM "Frame" USING [Alloc, Free],
  RuntimeInternal: FROM "RuntimeInternal" USING [MakeFsi];

--Process: FROM "Process" USING [DisableAborts];
FilerExceptionImpl: MONITOR
  IMPORTS Frame, RuntimeInternal --, Process

  EXPORTS FilerException, FilerPrograms =
  BEGIN
  FE: TYPE = RECORD [pFEPrev: PFE, req: FilePageTransfer.Request];
  fsiFE: CARDINAL = RuntimeInternal.MakeFsi[SIZE[FE]];
  PFE: TYPE = POINTER TO FE;
  pFELast: PFE ← NIL; -- points to last of fifo queue of exception reports
  report: CONDITION;
  -- Waits for a Filer exception to occur
  Await: PUBLIC ENTRY PROCEDURE RETURNS [rwReq: FilePageTransfer.Request] =
    BEGIN
    pFE, pFENext: PFE;
    WHILE pFELast = NIL DO WAIT report ENDLOOP;
    -- Remove first report from queue and return it
    FOR pFE ← pFELast, pFE.pFEPrev WHILE pFE.pFEPrev ~= NIL DO
      pFENext ← pFE; ENDLOOP;
    rwReq ← pFE.req;
    IF pFE = pFELast THEN pFELast ← NIL ELSE pFENext.pFEPrev ← NIL;
    Frame.Free[pFE]
    END;
  -- Add report to end of queue

  Report: PUBLIC ENTRY PROCEDURE [rwReq: FilePageTransfer.Request] =
    BEGIN
    pFE: PFE = LOOPHOLE[Frame.Alloc[fsiFE], PFE];
    pFE↑ ← [pFELast, rwReq];
    pFELast ← pFE;
    NOTIFY report
    END;
  --Process.DisableAborts[@report];

  END.
LOG
Time: February 27, 1979 9:51 PM          By: Redell Action: Created file from old CacheMissImpl.
**mesa
Time: November 16, 1979 2:24 PM          By: Forrest          Action: Changed to allocate Fram
**es for storage
Time: April 28, 1980 9:26 AM    By: Forrest          Action: FrameOps = > Frame.
Time: timeStamp      By: yourName          Action: shortDescription
```

```
-- FilerTransferImpl.mesa (last edited by: Forrest on: October 11, 1980 11:16 PM)

-- Things to consider:
-- 1) Large XWire request (>Transfer.maxConcurrency) will hang forever; should split up as with la
**bel requests (must deal with multiple processes however...simply making this module a monitor
**would seem to introduce deadlocks ...?)
-- 2) Main routines for data and label transfers should be combined. Note that combined routine wi
**ll be an external procedure and will execute within monitor for label transfers but outside it for d
**ata...
-- 3) Current code in Initiate will fail if a page group crosses a physical volume boundary.

DIRECTORY
  DiskChannel USING [
    Address, CompletionHandle, Create, CreateCompletionObject, Delete, Drive,
    Handle, InitiateIO, IORequest, Label, PLabel, WaitAny],
  Environment USING [PageCount, PageNumber, wordsPerPage],
  File USING [PageCount, PageNumber],
  FileCache USING [GetFilePtrs, GetPageGroup, ReturnFilePtrs],
  FileInternal USING [Descriptor, FilePtr, PageGroup],
  FilePageTransfer USING [Request],
  FilerException USING [Report],
  FilerPrograms USING [],
  FileTask USING [LabelWait --,XWireStart--],
  Inline USING [LongNumber, LowHalf],
  LabelTransfer USING [LabelStatus, Operation],
  PilotDisk USING [Label],
  --RemotePageTransfer USING [Initiate, Suggest],
  ResidentMemory USING [Allocate],
  SubVolume USING [Find, Handle, PageNumber, StartIO, IO],
  --System USING [nullNetworkAddress],
  --SystemInternal USING [altoFPSeries, UniversalID],
  Utilities USING [PageFromLongPointer],
  VolumeInternal USING [ --altoVolume,--PageNumber];

FilerTransferImpl: MONITOR
  IMPORTS
    DiskChannel, FileCache, FilerException, FileTask, Inline, ResidentMemory,
    SubVolume, Utilities
  EXPORTS FilePageTransfer, FilerPrograms, LabelTransfer
  SHARES File =
  BEGIN

  -- Temporary histogram to record statistics on runs of pages transferred
  logging: BOOLEAN ← FALSE;
  hist: ARRAY [0..16] OF CARDINAL ← ALL[0];
  Log: PRIVATE PROC [size: CARDINAL] = INLINE
    BEGIN
    FOR i: CARDINAL IN [0..16] DO
      IF size = 0 THEN {hist[i] ← hist[i] + 1; EXIT} ELSE size ← size/2; ENDLOOP
    END;
  -- hist[j] counts requests of size [2↑(j-1)..(2↑j)-1]

  Initiate: PUBLIC PROC [req: FilePageTransfer.Request]
    RETURNS [countInitiated: Environment.PageCount] =
    BEGIN OPEN FilePageTransfer, req;
    groupSize: Environment.PageCount;
    fileFound, groupFound: BOOLEAN;
    fileP: FileInternal.FilePtr;
    --fileD: FileInternal.Descriptor;
```



```
    rReq: Request = req;
    group: FileInternal.PageGroup;
    countInitiated ← 0;
    IF count = 0 THEN RETURN;
    --IF LOOPHOLE[file.fID, SystemInternal.UniversalID].series =
    --  SystemInternal.altoFPSeries THEN + + Alto files are special remote files
    -- BEGIN
    -- fileP ← @fileD;
    -- fileD ← FileInternal.Descriptor[file.fID, VolumeInternal.altoVolume, remote[]];
    -- END
    --ELSE
    -- BEGIN
    [fileFound, fileP] ← FileCache.GetFilePtrs[1, file.fID];
    IF ~fileFound THEN {FilerException.Report[req]; RETURN};
    -- END;
    WHILE count > 0 DO
      -- break request into page groups and start transfers
      WITH fileP SELECT FROM
        local =>
          BEGIN
          svH: SubVolume.Handle;
          svFound: BOOLEAN;
          svPage: SubVolume.PageNumber;
          [groupFound, group] ← FileCache.GetPageGroup[file.fID, filePage];
          IF ~groupFound THEN {
            FileCache.ReturnFilePtrs[1, fileP]; FilerException.Report[req]; EXIT};
          groupSize ← MIN[count, Inline.LowHalf[group.nextFilePage - filePage]];
          IF logging THEN Log[groupSize];
          [svFound, svH] ← SubVolume.Find[fileP.volumeID, group.volumePage];
          svPage ← group.volumePage - svH.lvPage + (filePage - group.filePage);
          BEGIN
          io: SubVolume.IO ←
            [op: operation, subVolume: svH, subVolumePage: svPage, filePtr: fileP,
              memPage: memoryPage, filePage: filePage, pageCount: groupSize];
          SubVolume.StartIO[@io];
          END;
          END;
        remote =>
          BEGIN
          --groupSize ← 1; + + yuck! This should be in runs, too
          --FileTask.XWireStart[memoryPage, fileP];
          --RemotePageTransfer.Suggest[[operation, file.fID, filePage, memoryPage]];
          -- Alto file hack: remove later

          END;
        ENDCASE;
      filePage ← filePage + groupSize;
      memoryPage ← memoryPage + groupSize;
      count ← count - groupSize;
      countInitiated ← countInitiated + groupSize;
      IF count > 0 THEN [] ← FileCache.GetFilePtrs[1, file.fID];
      ENDLOOP;
    --WITH fileP SELECT FROM
    -- remote => RemotePageTransfer.Initiate[rReq, System.nullNetworkAddress]; ENDCASE;

    END;

  -- Start of monitor implementing LabelTransfer interface
```

```
-- Empty page is source of zeros (sink for garbage) when writing (reading) labels.

emptyPagePtr: LONG POINTER = ResidentMemory.Allocate[hyperspace, 1];
emptyPage: Environment.PageNumber = Utilities.PageFromLongPointer[emptyPagePtr];
completion: DiskChannel.CompletionHandle ← DiskChannel.CreateCompletionObject[];

ReadLabel: PUBLIC ENTRY PROC [
  file: FileInternal.Descriptor, filePage: File.PageNumber,
  volumePage: VolumeInternal.PageNumber] RETURNS [label: PilotDisk.Label] =
  BEGIN
  pageGroup: FileInternal.PageGroup ← [filePage, volumePage, filePage + 1];
  [label, ] ← Perform[readLabel, @file, @pageGroup, emptyPage];
  END;

ReadRootLabel: PUBLIC ENTRY PROC [
  drive: DiskChannel.Drive, rootPage: VolumeInternal.PageNumber]
  RETURNS [label: PilotDisk.Label, labelValid: LabelTransfer.LabelStatus] =
  BEGIN OPEN DiskChannel;
  rootChannel: Handle ← Create[drive, completion];
  -- temporary channel for root page
  request: IORequest;
  pLabel: PLabel;
  labelStorage: ARRAY [0..SIZE[Label] + 3) OF WORD;
  -- space for label plus extra for quad-alignment
  pLabel ← LOOPHOLE[((LOOPHOLE[LONG[@labelStorage[0]], LONG INTEGER] + 3)/4)*4];
  --quad-align label
  BEGIN OPEN request;
  -- Can't use constructor because of blasted PRIVATE fields...
  channel ← rootChannel;
  diskPage ← rootPage;
  memoryPage ← emptyPage;
  count ← 1;
  command ← vrr;
  label ← pLabel;
  dontIncrement ← TRUE;
  END;
  InitiateIO[@request];
  labelValid ←
    SELECT WaitAny[completion].status FROM
      goodCompletion => valid,
      hardwareError, notReady => diskError,
      ENDCASE => invalid;
  Delete[rootChannel];
  label ← LOOPHOLE[pLabel↑];
  END;

WriteLabels: PUBLIC ENTRY PROC [
  file: FileInternal.Descriptor, pageGroup: FileInternal.PageGroup] =
  BEGIN OPEN Environment, pageGroup;
  LOOPHOLE[emptyPagePtr, LONG POINTER TO ARRAY [0..wordsPerPage) OF CARDINAL]↑ ←
    ALL[0];
  [] ← Perform[writeLabel, @file, @pageGroup, emptyPage];
  END;

-- This is an imperfect simulation for passing expectErrorAfterFirstPage
-- down the old disk channel (lacks recalibrates at the least)

VerifyLabels: PUBLIC ENTRY PROC [
```



```
  file: FileInternal.Descriptor, pageGroup: FileInternal.PageGroup,
  expectErrorAfterFirstPage: BOOLEAN ← FALSE]
  RETURNS [labelsValid: BOOLEAN, countValid: File.PageCount] =
  BEGIN
  THROUGH [0..10) DO
    [, labelsValid, countValid] ← Perform[
      verifyLabel, @file, @pageGroup, emptyPage];
    IF labelsValid OR (expectErrorAfterFirstPage AND countValid # 0) THEN EXIT;
    ENDLOOP;
  END;

ReadLabelAndData: PUBLIC ENTRY PROC [
  file: FileInternal.Descriptor, filePage: File.PageNumber,
  volumePage: VolumeInternal.PageNumber, memoryPage: Environment.PageNumber]
  RETURNS [PilotDisk.Label] =
  BEGIN
  pageGroup: FileInternal.PageGroup ← [filePage, volumePage, filePage + 1];
  RETURN[Perform[readLabelAndData, @file, @pageGroup, memoryPage].label]
  END;

WriteLabelAndData: PUBLIC ENTRY PROC [
  file: FileInternal.Descriptor, filePage: File.PageNumber,
  volumePage: VolumeInternal.PageNumber, memoryPage: Environment.PageNumber,
  bootChainLink: DiskChannel.Address] =
  BEGIN
  pageGroup: FileInternal.PageGroup ←
    [filePage: filePage, volumePage: volumePage, nextFilePage: filePage + 1];
  [] ← Perform[
    writeLabelsAndData, @file, @pageGroup, memoryPage, chained, bootChainLink];
  END;

Perform: INTERNAL PROC [
  -- Should be combined with Initiate (above)
  operation: LabelTransfer.Operation,
  fileP: POINTER TO READONLY FileInternal.Descriptor,
  pageGroupPtr: POINTER TO READONLY FileInternal.PageGroup,
  memoryPage: Environment.PageNumber, labelType: {normal, chained} ← normal,
  bootChainLink: DiskChannel.Address ← NULL]
  RETURNS [
    label: PilotDisk.Label, labelsValid: BOOLEAN, countValid: File.PageCount] =
  BEGIN OPEN pageGroupPtr;
  count: File.PageCount ← nextFilePage - filePage;
  -- LONG to handle giant verifies, etc??
  subVol: SubVolume.Handle;
  subVolumeFound: BOOLEAN;
  IF logging THEN Log[Inline.LowHalf[count]];
  [subVolumeFound, subVol] ← SubVolume.Find[fileP.volumeID, volumePage];
  IF ~subVolumeFound THEN ERROR; -- local volume must be found
  WITH fileP SELECT FROM
    remote => ERROR; -- can't transfer remote labels

    local =>
      BEGIN
      labelsValid ← TRUE;
      IF count > 0 THEN
        BEGIN
        io: SubVolume.IO ←
          [op: operation, subVolume: subVol, subVolumePage: volumePage,
            memPage: memoryPage, filePtr: fileP,
```

```
            filePage: pageGroupPtr.filePage,
            -- Following causes StackModelingError in 6.0u compiler:
            -- pageCount: Inline.LowHalf[count], + + should be long
            pageCount: LOOPHOLE[count, Inline.LongNumber].lowbits,
            -- should be long
            fixedMemPage: operation IN [readLabel..verifyLabel],
            chained: labelType = chained, link: bootChainLink];
          SubVolume.StartIO[@io];
          [label, labelsValid, countValid] ← FileTask.LabelWait[];
        END;
      END;
    ENDCASE;
  END;

END....
```

February 28, 1979 9:28 AM Redell Create from FilePageTransferImpl and LabelTransferIm
***pl*
March 21, 1979 12:01 PM Redell Convert to Mesa 5.0
August 13, 1979 3:59 PM Redell Minor mods for boot-chain machinery
September 5, 1979 3:13 PM McJones AR1593: Add countInitiated result to Initiate
September 17, 1979 4:42 PM Forrest Change Read Root Label
October 17, 1979 8:41 PM Redell Add histogram
November 26, 1979 3:49 PM Gobbel Initial changes for runs of pages: SubVolume.StartIO ta
***kes count arg, always 1 for now*
December 5, 1979 4:24 PM Gobbel Runs of pages for real
January 9, 1980 2:29 PM Gobbel Add countValid to VerifyLabels and Perform
January 30, 1980 3:16 PM Gobbel Remove Alto file stuff
August 14, 1980 9:54 AM McJones WriteLabelsAndData = >WriteLabelAndData; FilePageL
***abel = >PilotDisk*
October 11, 1980 9:30 PM Forrest Add expectLabelCheck to VerifyLabels (and hack imple
***mentation until we come up with correct solution)*

```
-- FileTaskImpl.mesa (last edited by: Gobbel on: January 30, 1980 11:16 PM) --
-- Things to consider:
-- 1) The task table is currently determined from the constant FileTask.maxConcurrency. Probably
**an even number of pages should be filled with tasks and the resulting table size be exported as f
**ileTask.maxConcurrency. (Does any other piece of code depend on this being a compile-time c
**onstant?). On the other hand, allocating an even number of pages may be too heavy-handed...

DIRECTORY
  DiskChannel: FROM "DiskChannel" USING [IORequest, IORequestHandle, Label],
  Environment: FROM "Environment" USING [
    Base, PageCount, PageNumber, wordsPerPage],
  File: FROM "File" USING [PageCount],
  FilePageTransfer: FROM "FilePageTransfer",
  FilePageLabel: FROM "FilePageLabel" USING [Label],
  FileInternal: FROM "FileInternal" USING [FilePtr, Operation],
  FileCache: FROM "FileCache" USING [ReturnFilePtrs],
  FilerPrograms: FROM "FilerPrograms",
  FileTask: FROM "FileTask",
  Process: FROM "Process" USING [DisableAborts],
  ResidentMemory: FROM "ResidentMemory" USING [Allocate],
  RuntimeInternal: FROM "RuntimeInternal" USING [WorryCallDebugger],
  Utilities: FROM "Utilities" USING [ShortCARDINAL],
  VM: FROM "VM" USING [Interval];

FileTaskImpl: MONITOR
  IMPORTS FileCache, Process, ResidentMemory, RuntimeInternal, Utilities
  EXPORTS FileTask, FilePageTransfer, FilerPrograms =
  BEGIN
  ErrorHalt: PROCEDURE =
    BEGIN RuntimeInternal.WorryCallDebugger["Error in FileTaskImpl"L]; END;

  nTasks: CARDINAL = FileTask.maxConcurrency;
  taskCount, taskMax: CARDINAL ← 0; -- temp measurement counters
  taskStorage: CARDINAL = nTasks*SIZE[Task];
  taskPages: Environment.PageCount =
    taskStorage/Environment.wordsPerPage +
      (IF taskStorage MOD Environment.wordsPerPage = 0 THEN 0 ELSE 1);
  table: Environment.Base ← ResidentMemory.Allocate[hyperspace, taskPages];
  -- File Task Table
  TaskNo: TYPE = [0..nTasks);
  TaskPtr: TYPE = LONG POINTER TO Task;
  TaskHandle: TYPE = Environment.Base RELATIVE POINTER TO Task;
  nil: TaskHandle = LOOPHOLE[177777B];
  Task: TYPE = RECORD [
    next: TaskHandle,
    operationClass: OperationClass,
    label: DiskChannel.Label,
    labelError: BOOLEAN,
    memPage: Environment.PageNumber,
    pageCount: File.PageCount,
    countOK: File.PageCount, -- number of pages transferred successfully
    filePtr: FileInternal.FilePtr,
    var:
      SELECT OVERLAID * FROM
        disk => [request: DiskChannel.IORequest],
        xwire => NULL, -- ??? --
        ENDCASE];
  OperationClass: TYPE = {dataOperation, labelOperation};
```



```
  doneD, doneL, free: CONDITION;
  doneDQ, doneLQ, freeQ: TaskHandle;
  --FileTask.--
  DiskStart: PUBLIC ENTRY PROCEDURE [
    mPage: Environment.PageNumber, count: File.PageCount,
    fPtr: FileInternal.FilePtr, operation: FileInternal.Operation]
    RETURNS [DiskChannel.IORequestHandle] =
    BEGIN
    taskH: TaskHandle;
    taskP: TaskPtr;
    WHILE freeQ = nil DO WAIT free ENDLOOP;
    taskH ← freeQ;
    taskP ← @table[taskH];
    freeQ ← taskP.next;
    taskMax ← MAX[taskMax, taskCount ← taskCount + 1];
    BEGIN OPEN taskP;
    operationClass ←
      IF operation IN FilePageTransfer.Operation THEN dataOperation
      ELSE labelOperation;
    memPage ← mPage;
    pageCount ← count;
    filePtr ← fPtr;
    request.label ← @label;
    request.tag ← LOOPHOLE[taskH]; -- store handle in request for DiskFinish
    RETURN[@request]
    END;
    END;
  --FileTask.--

  DiskFinish: PUBLIC ENTRY PROCEDURE [
    reqH: DiskChannel.IORequestHandle, labelValid: BOOLEAN] =
    BEGIN
    taskH: TaskHandle = LOOPHOLE[reqH.tag]; -- go directly to task using Handle
    taskP: TaskPtr = @table[taskH];
    BEGIN OPEN taskP;
    labelError ← ~labelValid;
    countOK ← reqH.countDone;
    IF operationClass = dataOperation THEN
      BEGIN
      next ← doneDQ;
      doneDQ ← taskH;
      NOTIFY doneD;
      FileCache.ReturnFilePtrs[1, filePtr];
      IF labelError THEN RuntimeInternal.WorryCallDebugger["Disk label check"];
      -- Should be non-fatal somehow

      END
    ELSE BEGIN next ← doneLQ; doneLQ ← taskH; NOTIFY doneL END;
    END;
    END;
  -- EtherStart: PUBLIC ENTRY PROCEDURE [mPage: Environment.PageNumber, fPtr: FileInternal
**.FilePtr] =
  -- BEGIN
  -- Real version must handle duplicates, retransmissions, etc. ...will probably need a fancier key...
  -- END;
  -- EtherFinish: PUBLIC ENTRY PROCEDURE [mPage: Environment.PageNumber, other args...]
** =
  -- BEGIN
```

```
-- Real version must handle duplicates, retransmissions, etc. ...will probably need a fancier key...
-- END;
--FilePageTransfer.--

Wait: PUBLIC ENTRY PROCEDURE RETURNS [VM.Interval] =
  BEGIN
  taskH: TaskHandle;
  taskP: TaskPtr;
  WHILE doneDQ = nil DO WAIT doneD ENDLOOP;
  taskH ← doneDQ;
  taskP ← @table[taskH];
  doneDQ ← taskP.next;
  BEGIN OPEN taskP;
  next ← freeQ;
  freeQ ← taskH;
  NOTIFY free;
  taskCount ← taskCount - 1;
  RETURN[VM.Interval[memPage, Utilities.ShortCARDINAL[countOK]]]
  -- kludge! remove this when Environment.PageCount is 32 bits

  END;
  END;
--FileTask.--

LabelWait: PUBLIC ENTRY PROCEDURE
  RETURNS [
    label: FilePageLabel.Label, labelValid: BOOLEAN,
    countValid: File.PageCount] =
  BEGIN
  taskH: TaskHandle;
  taskP: TaskPtr;
  WHILE doneLQ = nil DO WAIT doneL ENDLOOP;
  taskH ← doneLQ;
  taskP ← @table[taskH];
  doneLQ ← taskP.next;
  BEGIN OPEN taskP;
  next ← freeQ;
  freeQ ← taskH;
  NOTIFY free;
  taskCount ← taskCount - 1;
  RETURN[LOOPHOLE[label], ~labelError, countOK]
  END;
  END;
-- Initialize File Task Table --

BEGIN
lastTask: TaskHandle = LOOPHOLE[(nTasks - 1)*SIZE[Task]];
taskH: TaskHandle ← LOOPHOLE[0];
freeQ ← taskH;
UNTIL taskH = lastTask DO
  taskH ← table[taskH].next ← taskH + SIZE[Task] ENDLOOP;
table[lastTask].next ← nil;
doneDQ ← doneLQ ← nil;
END;
Process.DisableAborts[@doneD];
Process.DisableAborts[@doneL];
Process.DisableAborts[@free];
END.
```



```
LOG
Time: February 28, 1979  11:33 AM          By: Redell Action: Created file from old TransferImpl.m
**esa
Time: March 7, 1979  6:04 PM    By: Redell Action: Converted to Mesa 5.0; removed reference to ol
**d FilePageTransferInternal.
Time: March 26, 1979  4:59 PM  By: Redell Action: Added check for label error on data transfer (Fo
**r now, fatal error = > Call Debugger).
Time: August 20, 1979  12:19 PM            By: Redell Action: Removed LOOPHOLEs to DiskChan
**nel types.
Time: November 26, 1979  12:31 PM          By: Gobbel          Action: Added count to FTTEntry
**and Wait.
Time: December 16, 1979  3:32 PM           By: Redell Action: Moved task tabel to hyperspace. cha
**nged linear searches to LIFO queues, removed Alto machinery, general cleanup.
Time: January 29, 1980  1:29 PM            By: Gobbel          Action: Merge Redell's changes w
**ith changes for new LabelTransfer (countValid added to LabelWait), change VM.Interval to FileI
**nternal.Interval.
```

```
-- SubVolumeImpl.mesa (last edited by: Luniewski on: May 20, 1980 3:04 PM)
--Things to consider:
--1) Taking subvolumes offline needs more synchronization. Can probably cover subvolumes with
**filecache reference counts since holder of a SubVolume.Handle usually holds a FilePtr for a file
**on the subvolume. Offline must then flush file cache before removing subvolume (allows pendin
**g I/O to complete).

DIRECTORY
  --AltoDiskController: FROM "AltoDiskController",
  DiskChannel: FROM "DiskChannel" USING [
    Address, Command, CompletionHandle, CreateCompletionObject, GetPageAddress,
    Handle, --Idle,--InitiateIO, IORequestHandle, nullHandle, --Restart,--
    WaitAny],
  Environment: FROM "Environment" USING [PageNumber],
  File: FROM "File" USING [ID, PageCount, PageNumber],
  FileInternal: FROM "FileInternal" USING [FilePtr, Operation],
  FilePageLabel: FROM "FilePageLabel" USING [
    Label, nullLabel, SetFilePage, SetType],
  FilerPrograms: FROM "FilerPrograms",
  FileTask: FROM "FileTask" USING [DiskFinish, DiskStart],
  Inline: FROM "Inline" USING [LowHalf, DIVMOD],
  PhysicalVolumeFormat: FROM "PhysicalVolumeFormat" USING [SubVolumeDesc],
  Process: FROM "Process" USING [Detach],
  RuntimeInternal: FROM "RuntimeInternal" USING [WorryCallDebugger],
  SubVolume: FROM "SubVolume" USING [Descriptor, Handle, IOptr],
  Volume: FROM "Volume" USING [ID],
  VolumeInternal: FROM "VolumeInternal" USING [PageNumber];

SubVolumeImpl: MONITOR
  IMPORTS DiskChannel, FilePageLabel, FileTask, Inline, Process, RuntimeInternal
  EXPORTS --AltoDiskController,--SubVolume, FilerPrograms =
  BEGIN OPEN DiskChannel;
  -- Preliminary version of subvolume cache
  cacheSize: CARDINAL = 12;
  -- gasp; each opened physical Volume takes one plus 1 for each real subvolume.
  CacheIndex: TYPE = [1..cacheSize];
  cache: ARRAY CacheIndex OF CacheEntry ← ALL[[FALSE, ]];
  CacheEntry: TYPE = RECORD [occupied: BOOLEAN, svDesc: SubVolume.Descriptor];
  CePtr: TYPE = ORDERED POINTER TO CacheEntry;
  ceFirst: CePtr = LOOPHOLE[@cache[FIRST[CacheIndex]]]; -- loophole for Ordered
  ceLast: CePtr = LOOPHOLE[@cache[LAST[CacheIndex]]];
  -- Signals down here are bad
  ErrorHalt: PROCEDURE [s: STRING] =
    BEGIN RuntimeInternal.WorryCallDebugger[s]; END;

  Find: PUBLIC ENTRY PROCEDURE [vID: Volume.ID, page: VolumeInternal.PageNumber]
    RETURNS [success: BOOLEAN, subVolume: SubVolume.Handle] =
    BEGIN [success, subVolume] ← FindSV[vID, page]; END;

  FindSV: INTERNAL PROCEDURE [vID: Volume.ID, page: VolumeInternal.PageNumber]
    RETURNS [BOOLEAN, SubVolume.Handle] = INLINE
    BEGIN
    cePtr: CePtr;
    FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr <= ceLast DO
      IF cePtr.occupied AND vID = cePtr.svDesc.lvID AND page IN
        [cePtr.svDesc.lvPage..cePtr.svDesc.lvPage + cePtr.svDesc.nPages) THEN
        RETURN[TRUE, @cePtr.svDesc];
      ENDLOOP;
```



```
    RETURN[FALSE, NIL]
    END;

  GetNext: PUBLIC ENTRY PROCEDURE [inSubVolume: SubVolume.Handle]
    RETURNS [outSubVolume: SubVolume.Handle] =
    BEGIN
    cePtr: CePtr;
    found: BOOLEAN ← inSubVolume = NIL;
    FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr <= ceLast DO
      IF found THEN BEGIN IF cePtr.occupied THEN RETURN[@cePtr.svDesc]; END
      ELSE found ← (inSubVolume = @cePtr.svDesc);
      ENDLOOP;
    IF ~found THEN ErrorHalt["illegalSubvolumeHandle"L];
    RETURN[NIL]
    END;

  GetPageAddress: PUBLIC ENTRY PROCEDURE [
    vID: Volume.ID, page: VolumeInternal.PageNumber]
    RETURNS [channel: DiskChannel.Handle, address: DiskChannel.Address] =
    BEGIN
    found: BOOLEAN;
    svH: SubVolume.Handle;
    [found, svH] ← FindSV[vID, page];
    IF NOT found THEN ErrorHalt["getPageAddressFailure"L];
    RETURN[
      svH.channel, DiskChannel.GetPageAddress[
        svH.channel, svH.pvPage + (page - svH.lvPage)]]
    END;

  OnLine: PUBLIC ENTRY PROCEDURE [
    subVolume: PhysicalVolumeFormat.SubVolumeDesc, channel: DiskChannel.Handle] =
    BEGIN OPEN subVolume;
    cePtr: CePtr;
    FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr <= ceLast DO
      IF ~cePtr.occupied THEN
        BEGIN
        cePtr↑ ←
          [occupied: TRUE,
            svDesc:
            [lvID: lvID, lvPage: lvPage, pvPage: pvPage, nPages: nPages,
              channel: channel]];
        RETURN;
        END;
      ENDLOOP;
    ErrorHalt["subvolume cache overflow"L];
    END;
  -- Flush Subvolume; (null Handle flushes all subvolumes of a given logical volume)

  OffLine: PUBLIC ENTRY PROCEDURE [vID: Volume.ID, channel: DiskChannel.Handle] =
    BEGIN
    cePtr: CePtr;
    FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr <= ceLast DO
      OPEN cePtr;
      IF svDesc.lvID = vID AND
        (channel = DiskChannel.nullHandle OR channel = svDesc.channel) THEN
        occupied ← FALSE;
      ENDLOOP;
    END;
```

```
--Acquire: PUBLIC ENTRY PROCEDURE =
--BEGIN + + Idles all Pilot subvolumes to allow Alto file I/O
--cePtr: CePtr;
--FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr < = ceLast
--DO IF cePtr.occupied THEN Idle[cePtr.svDesc.channel]; ENDLOOP;
--END;
--Release: PUBLIC ENTRY PROCEDURE =
--BEGIN + + Restarts all Pilot subvolumes when Alto file I/O is completed
--cePtr: CePtr;
--FOR cePtr ← ceFirst, cePtr + SIZE[CacheEntry] WHILE cePtr < = ceLast
--DO IF cePtr.occupied THEN Restart[cePtr.svDesc.channel]; ENDLOOP;
--END;

StartIO: PUBLIC --ENTRY--PROCEDURE [io: SubVolume.IOptr] =
  BEGIN
  labelSize: CARDINAL = SIZE[FilePageLabel.Label];
  ErrorHalt1: PROCEDURE RETURNS [DiskChannel.Command] = LOOPHOLE[ErrorHalt];
  requestHandle: IORequestHandle;
  properLabel: FilePageLabel.Label ← FilePageLabel.nullLabel;
  properLabel.fileID ← io.filePtr.fileID;
  FilePageLabel.SetFilePage[@properLabel, io.filePage];
  IF io.chained THEN properLabel.bootChainLink ← LOOPHOLE[io.link];
  WITH io.filePtr SELECT FROM -- couldn't his be done at a higher level???

    local =>
      BEGIN
      FilePageLabel.SetType[@properLabel, type];
      IF io.filePage = 0 THEN
        BEGIN
        properLabel.immutable ← immutable;
        properLabel.temporary ← temporary;
        properLabel.zeroSize ← (size = 0);
        END;
      END;
    ENDCASE => ErrorHalt["can't do label operations on remote files"L];
  requestHandle ← FileTask.DiskStart[
    io.memPage, io.pageCount, io.filePtr, io.op];
  BEGIN OPEN requestHandle;
  channel ← io.subVolume.channel;
  diskPage ← io.subVolume.pvPage + io.subVolumePage;
  label↑ ← LOOPHOLE[properLabel];
  memoryPage ← io.memPage;
  count ← io.pageCount;
  dontIncrement ← io.fixedMemPage;
  command ←
    SELECT io.op FROM
      read => vvr,
      write => vvw,
      readLabel => vrr,
      writeLabel => vww,
      verifyLabel => vvr,
      readLabelAndData => vrr,
      writeLabelsAndData => vww,
      ENDCASE => ErrorHalt1[];
  END;
  InitiateIO[requestHandle];
  END;
```



```
FinishIO: PROCEDURE = -- root of Finisher process
  BEGIN
  requestHandle: IORequestHandle;
  labelValid: BOOLEAN ← TRUE;
  hardErrorString: STRING ← "Unrecoverable disk error page XXXXX";
  DO
    requestHandle ← WaitAny[completion];
    SELECT requestHandle.status FROM -- must clearly get smarter

      goodCompletion => labelValid ← TRUE;
      labelDoesNotMatch => labelValid ← FALSE;
      ENDCASE =>
        BEGIN
        i: CARDINAL (0..5];
        num: CARDINAL ← Inline.LowHalf[requestHandle.diskPage];
        dig: CARDINAL;
        DO
          FOR i IN (0..5] DO
            [num, dig] ← Inline.DIVMOD[num, 10];
            hardErrorString[hardErrorString.length - i] ←
              IF dig = 0 AND num = 0 AND i # 1 THEN ' ELSE '0 + dig;
            ENDLOOP;
          RuntimeInternal.WorryCallDebugger[hardErrorString]
          ENDLOOP;
        END;
    FileTask.DiskFinish[requestHandle, labelValid];
    ENDLOOP;
  END;
--Initialization

  completion: PUBLIC CompletionHandle ← CreateCompletionObject[];
  Process.Detach[FORK FinishIO[]];
  END.
```

LOG

Time: February 22, 1979 4:33 PM By: Redell Action: Created file from old DiskImpl and VolumeCa
***cheImpl*
Time: March 21, 1979 6:00 PM By: Redell Action: Converted to Mesa 5.0
Time: March 26, 1979 4:50 PM By: Redell Action: Fixed typo: write = > put in StartIO.
Time: July 30, 1979 10:44 AM By: Forrest Action: Made compile with new subvolume. Upped cac
***he to 8.*
Time: August 13, 1979 6:32 PM By: Redell Action: Added bootchain machinery; will be modified w
***hen disk driver is changed to accept runs of pages.*
Time: September 10, 1979 6:29 PM By: Forrest Action: Use Pilot file types vs fileTypes.
Time: September 19, 1979 8:54 AM By: Forrest Action: Add GetNext; Change Offline to also take
***channel Handle.*
Time: September 21, 1979 11:43 AM By: Forrest Action: Add Set/GetMarker ID.
Time: September 22, 1979 2:44 PM By: Forrest Action: took out Set/GetMarker ID.
Time: November 26, 1979 3:45 PM By: Gobbel Action: Changes for new DiskChannel (runs of pa
***ges).*
Time: December 13, 1979 7:20 PM By: Gobbel Action: Added fixedMemPage option, make param
***eter to StartIO be handle on a record.*
Time: January 22, 1980 1:38 PM By: Gobbel Action: Tell disk page number when we go to the de
***bugger for a hard error.*
Time: January 30, 1980 3:30 PM By: Gobbel Action: Remove Alto file stuff.
Time: May 20, 1980 3:04 PM By: Luniewski Action: PhysicalVolume = > PhysicalVolumeFormat.

```
-- Swapper>CachedRegionImplA.mesa (last edited by Forrest/Richard on October 13, 1980 10:00
**AM)
-- Watchout: This module was updated without protection. Paul actually has it checked out, and
**will merge in my change at his checkin
-- Implements the region cache and access to it.
-- Exclusive access to the region cache is provided by the monitor lock "regionCacheLock". Excl
**usive access to individual region descriptors is provided by the checkedOut field in the descript
**ors in the cache.
-- Note: No frame heap ALLOC's may be executed within any ENTRY procedures in this monitor, s
**o that they may be called from the allocation trap handler. If this rule were not followed, we cou
**ld get an allocation trap when we held the monitor lock, thus making DeallocateClean inaccessi
**ble to the allocation trap handler.

DIRECTORY
  CachedRegion,
  CachedRegionInternal,
  Environment USING [wordsPerPage],
  Frame USING [GetReturnLink, SetReturnLink],
  Inline USING [LongCOPY],
  MStore USING [Allocate, AllocateIfFree, Deallocate],
  PageMap USING [Flags, flagsVacant, GetF],
  PrincOps USING [Port],
  Process USING [DisableAborts, InitializeMonitor],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  RuntimeInternal USING [WorryCallDebugger],
  SimpleSpace USING [AllocateVM],
  Space USING [defaultWindow],
  SpecialSpace USING [realMemorySize],
  StoragePrograms USING [countVM, DescribeSpace, LongPointerFromPage, outlaw],
  VM USING [Interval, PageCount, PageNumber, PageOffset];

CachedRegionImplA: MONITOR LOCKS regionCacheLock
  IMPORTS
    CachedRegionInternal, Frame, Inline, MStore, PageMap, Process,
    ProcessInternal, RuntimeInternal, SimpleSpace, SpecialSpace, StoragePrograms
  EXPORTS CachedRegion, CachedRegionInternal, StoragePrograms
  SHARES CachedRegionInternal, PageMap =
  BEGIN OPEN CachedRegion, CachedRegionInternal;
  -- public data:
  checkIn: PUBLIC CONDITION;
  -- broadcast whenever a region descriptor is checked in.
  pageTop: PUBLIC VM.PageNumber;
  -- private data:
  regionCacheLock: PUBLIC MONITORLOCK; -- (PRIVATE in interface)
  pageRegionCache: VM.PageNumber;
  countRegionCache: VM.PageCount; -- amount of VM allocated.
  countCacheMapped: VM.PageCount;
  -- number of pages of region cache currently mapped.
  --nDescMax: CARDINAL; + + current max number of Desc's in cache.
  IDesc: TYPE = [0.. --nDescMax--0);
  -- index of region descriptor cache (grows dynamically).
  RegionDesc: TYPE = ARRAY IDesc OF Desc;
  BaseDesc: TYPE = LONG BASE POINTER TO RegionDesc; -- allocated externally.
  DebuggerBaseDesc: TYPE = LONG POINTER TO RegionDesc; -- (for debugger use only.)
  PDesc: TYPE = BaseDesc RELATIVE ORDERED POINTER [0..177777B] TO Desc;
  nPad: CARDINAL = 1;
  -- For the convenience of Find, one extra dummy Desc is prepended to the cache to allow a PDe
**sc to be less than pDescFirst.
```



```
  baseDesc: BaseDesc; -- base of cache
  pDescFirst: PDesc = FIRST[PDesc] + nPad*SIZE[Desc];
  -- offset of first entry of cache
  pDescLast: PDesc; -- offset of current last entry of cache
  pDescMax: PDesc; -- offset of last possible entry of cache
  pDescCommutator: PDesc;
  -- offset of entry which replacement algorithm will consider next
  ----------------------------------------
  -- Initialization
  ----------------------------------------
  --StoragePrograms.--
  InitializeRegionCacheA: PUBLIC PROCEDURE =
    BEGIN
    Process.InitializeMonitor[@regionCacheLock];
    -- must precede call to InitializeInternal
    InitializeInternal[];
    END;

  InitializeInternal: ENTRY --so Initialize* can be called--PROCEDURE =
    BEGIN
    nDescMax: CARDINAL; -- current max number of Desc's in cache.
    Process.DisableAborts[@checkIn];
    countRegionCache ←
      (SpecialSpace.realMemorySize*SIZE[Desc] + Environment.wordsPerPage -
        1)/Environment.wordsPerPage; -- enough VM for one region per real page.
    pageRegionCache ← SimpleSpace.AllocateVM[countRegionCache, hyperspace];
    countCacheMapped ← 1; -- one mapped page initialiy.
    MStore.Allocate[interval: [page: pageRegionCache, count: countCacheMapped]];
    baseDesc ← StoragePrograms.LongPointerFromPage[pageRegionCache];
    nDescMax ← (countCacheMapped*Environment.wordsPerPage)/SIZE[Desc] - nPad;
    pDescMax ← pDescFirst + SIZE[Desc]*(nDescMax - 1);
    pDescLast ← pDescCommutator ← pDescFirst;
    pageTop ← FIRST[VM.PageNumber] + StoragePrograms.countVM;
    baseDesc[pDescLast] ←
      [ -- initial entry in the cache.
        interval: [page: pageTop, count: 0],
        -- page + count = end of implemented VM.
        level: 0, -- don't care
        levelMapped: 0, -- don't care
        hasSwapUnits: FALSE, -- don't care
        dTemperature: FIRST[DTemperature], -- don't care
        dPinned: TRUE, -- we pin the last entry, which contains pageTop.
        dDirty: FALSE, -- don't care
        state: missing, -- don't care
        writeProtected: FALSE, -- don't care
        needsLogging: FALSE, -- don't care
        beingFlushed: FALSE]; -- don't care
    InitializeAllocateMStoreRuthlessly[]; -- allocate frame; initialize PORT.
    InitializeDeallocateClean[]; -- allocate frame; initialize PORT.
    [] ← InitializeFind[]; -- allocate frame; initialize PORT.
    [] ← InitializeInsertIfRoom[]; -- allocate frame; initialize PORT.

    END;
  --StoragePrograms.--

  InitializeRegionCacheB: PUBLIC PROCEDURE =
    BEGIN
    StoragePrograms.DescribeSpace[
```

```
    StoragePrograms.outlaw, pageRegionCache, countRegionCache,
    Space.defaultWindow]; -- tells Swapper about region cache space.

  END;
-------------------------------------------
-- monitor entries
-------------------------------------------

AwaitNotCheckedOut: PUBLIC ENTRY PROCEDURE [pageMember: VM.PageNumber] =
  BEGIN
  found: BOOLEAN;
  pDesc: PDesc;
  DO
    [found, pDesc] ← Find[pageMember];
    IF ~found OR baseDesc[pDesc].state ~= checkedOut THEN EXIT;
    WAIT checkIn
    ENDLOOP
  END;

CheckIn: PUBLIC ENTRY PROCEDURE [desc: Desc] =
  BEGIN
  found: BOOLEAN;
  pDesc: PDesc;
  [found, pDesc] ← Find[desc.interval.page];
  --assert--
  IF ~found OR baseDesc[pDesc].state ~= checkedOut THEN ERROR;
  IF desc.state = missing THEN
    -- Delete the entry
    BEGIN
    --assert--
    IF desc.dPinned THEN ERROR;
    -- Move descriptors following pDesc one place towards first
    Inline.LongCOPY[
      from: @baseDesc[pDesc + SIZE[Desc]], nwords: pDescLast - pDesc,
      to: @baseDesc[pDesc]];
    pDescLast ← pDescLast - SIZE[Desc];
    IF pDesc < pDescCommutator THEN
      pDescCommutator ← pDescCommutator - SIZE[Desc];
    END
  ELSE --desc.state = present--
    -- Update the entry
    BEGIN desc.dTemperature ← LAST[DTemperature]; baseDesc[pDesc] ← desc; END;
  BROADCAST checkIn;
  END;

CheckOut: PUBLIC ENTRY PROCEDURE [
  pageMember: VM.PageNumber, ifCheckedOut: ReturnWait] RETURNS [desc: Desc] =
  BEGIN
  found: BOOLEAN;
  pDesc: PDesc;
  DO
    [found, pDesc] ← Find[pageMember];
    desc ← baseDesc[pDesc];
    IF found THEN
      BEGIN OPEN baseDesc[pDesc];
      IF state ~= checkedOut THEN BEGIN state ← checkedOut; EXIT END
      ELSE IF ifCheckedOut = return THEN EXIT ELSE WAIT checkIn
      END
    ELSE BEGIN desc.state ← missing; desc.interval.count ← 0; EXIT END;
    ENDLOOP;
```



```
  END;

Insert: PUBLIC ENTRY PROCEDURE [desc: Desc] RETURNS [descVictim: Desc] =
  BEGIN
  success: BOOLEAN;
  --REPEAT ... UNTIL region cache big enough--
  DO
    [success, descVictim] ← InsertIfRoom[desc];
    IF success THEN EXIT
    ELSE
      BEGIN
      IF countCacheMapped >= countRegionCache THEN ERROR;
      -- insufficient VM allocated for region cache.
      allocateMStoreRuthlesslyInternal[
        interval: [pageRegionCache + countCacheMapped, 1]];
      countCacheMapped ← countCacheMapped + 1;
      pDescMax ← pDescMax + (Environment.wordsPerPage/SIZE[Desc])*SIZE[Desc];
      END;
    ENDLOOP;
  END;
-------------------------------------------------------------
-- monitor internals
-------------------------------------------------------------

allocateMStoreRuthlesslyInternal: PUBLIC --INTERNAL--
  AllocateMStoreRuthlesslyInternal ←
  -- (PRIVATE in interface) ("←" due to compiler glitch)
  [LOOPHOLE[@AwaitAllocateMStoreRuthlesslyRequest]];
-- an indirect control link to the PORT.
AwaitAllocateMStoreRuthlesslyRequest: --RESPONDING--PORT
  RETURNS [interval: VM.Interval];
-- args/results match allocateMStoreRuthlesslyInternal (but swapped).
InitializeAllocateMStoreRuthlessly: INTERNAL PROCEDURE =
  BEGIN
  countAllocated: VM.PageCount;
  intervalRemaining: VM.Interval;
  LOOPHOLE[AwaitAllocateMStoreRuthlesslyRequest, PrincOps.Port].dest ←
    Frame.GetReturnLink[];
  -- set my PORT call to return to my caller on call below.
  DO
    --FOREVER--
    -- Return; Await new request; Process it;
    intervalRemaining ← AwaitAllocateMStoreRuthlesslyRequest[];
    Frame.SetReturnLink[
      LOOPHOLE[AwaitAllocateMStoreRuthlesslyRequest, PrincOps.Port].dest];
    -- for debugger
    --UNTIL whole interval allocated--
    DO
      countAllocated ← MStore.AllocateIfFree[intervalRemaining];
      intervalRemaining ←
        [intervalRemaining.page + countAllocated,
          intervalRemaining.count - countAllocated];
      IF intervalRemaining.count = 0 THEN EXIT ELSE DeallocateClean[];
      ENDLOOP;
    ENDLOOP;
  END;

DeallocateClean: --INTERNAL--PROCEDURE = LOOPHOLE[@AwaitDeallocateCleanRequest];
-- an indirect control link to the PORT.
```

```
AwaitDeallocateCleanRequest: --RESPONDING--PORT;
InitializeDeallocateClean: INTERNAL PROCEDURE =
  BEGIN
  victim: PDesc ← pDescFirst;
  startingPoint: PDesc;
  LOOPHOLE[AwaitDeallocateCleanRequest, PrincOps.Port].dest ←
    Frame.GetReturnLink[];
  -- set my PORT call to return to my caller on call below.
  DO
    --FOREVER--
    -- Return; Await new request; Process it;
    AwaitDeallocateCleanRequest[];
    Frame.SetReturnLink[
      LOOPHOLE[AwaitDeallocateCleanRequest, PrincOps.Port].dest];
    -- for debugger
    IF victim >= pDescLast THEN victim ← pDescFirst;
    -- (assures victim is sensible.)
    startingPoint ← victim;
    --UNTIL checked-in, In, and clean region found--
    DO
      -- scope of baseDesc[victim]:
      BEGIN OPEN baseDesc[victim];
      IF state ~= checkedOut AND
        ((~hasSwapUnits AND state IN InSwappable) OR
          (hasSwapUnits AND state IN AliveSwappable)) THEN
        BEGIN --see if all-clean region--
        allVacant: BOOLEAN ← TRUE;
        ProcessInternal.DisableInterrupts[];
        -- prevents other processes from dirtying pages.
        FOR offset: VM.PageOffset IN [0..interval.count) DO
          --look for any dirty pages--
          flags: PageMap.Flags = PageMap.GetF[
            interval.page + offset].valueOld.flags;
          IF flags # PageMap.flagsVacant THEN {
            IF flags.dirty THEN EXIT; allVacant ← FALSE};
          REPEAT
            FINISHED => -- whole region is clean.
              IF ~allVacant THEN
                BEGIN
                MStore.Deallocate[interval: interval, promised: FALSE];
                -- interrupts are disabled!
                state ← outAlive;
                ProcessInternal.EnableInterrupts[];
                GO TO Deallocated;
                END;
          ENDLOOP;
        ProcessInternal.EnableInterrupts[];
        END --seeing if all-clean region--
      ;
      END --USING desc--
      ;
      victim ← victim + SIZE[Desc];
      IF victim >= pDescLast THEN victim ← pDescFirst;
      IF victim = startingPoint THEN
        RuntimeInternal.WorryCallDebugger["All memory pinned, busy, or dirty"];
      REPEAT --looking for clean region-- Deallocated => NULL;
      ENDLOOP;
    ENDLOOP;
  END;
```



```
  Find: --INTERNAL--PROCEDURE [page: VM.PageNumber]
    RETURNS [found: BOOLEAN, pDesc: PDesc] =
    -- Find region which includes page. If none, return first region in cache following page.
    LOOPHOLE[@AwaitFindRequest]; -- an indirect control link to the PORT.
  AwaitFindRequest: --RESPONDING--PORT [found: BOOLEAN, pDesc: PDesc]
    RETURNS [page: VM.PageNumber]; -- args/results match Find (but swapped).
  InitializeFind: INTERNAL PROCEDURE RETURNS [found: BOOLEAN, pDesc: PDesc]
    --to match PORT args-- =
    BEGIN
    --pDesc: PDesc; + + "returned result", AND, offset of entry last returned by Find. (Conceptua
**lly, this is a global.)
    page: VM.PageNumber;
    iDescOrigin: CARDINAL = (pDescFirst - FIRST[PDesc])/SIZE[Desc];
    iDesc, iDescL, iDescU: CARDINAL;
    pageComp: VM.PageNumber;
    LOOPHOLE[AwaitFindRequest, PrincOps.Port].dest ← Frame.GetReturnLink[];
    -- set my PORT call to return to my caller on call below.
    pDesc ← pDescFirst;
    DO
      --FOREVER--
      -- Return result; Await new request; Process it;
      page ← AwaitFindRequest[found, pDesc];
      Frame.SetReturnLink[LOOPHOLE[AwaitFindRequest, PrincOps.Port].dest];
      -- for debugger
      --assert--
      IF page = pageTop THEN ERROR;
      IF pDesc >= pDescLast THEN pDesc ← pDescFirst;
      -- assures pDesc is still reasonable.
      -- scope of SameAsLastTime--
      BEGIN
      IF page IN
        [baseDesc[
          pDesc].interval.page..baseDesc[pDesc].interval.page + baseDesc[
                                  pDesc].interval.count) THEN
        GO TO SameAsLastTime;
      iDescL ← iDescOrigin;
      iDescU ← (pDescLast - FIRST[PDesc])/SIZE[Desc];
      --UNTIL search terminates--
      DO
        iDesc ← (iDescL + iDescU)/2;
        pageComp ← baseDesc[FIRST[PDesc] + iDesc*SIZE[Desc]].interval.page;
        IF page < pageComp THEN iDescU ← iDesc - 1
          -- note that iDescU, a CARDINAL, might be iDescL-1 here.

        ELSE IF page > pageComp THEN iDescL ← iDesc + 1 ELSE GO TO Exact;
        IF iDescU < iDescL THEN GO TO NotExact;
        ENDLOOP;
      EXITS
        Exact => BEGIN pDesc ← FIRST[PDesc] + iDesc*SIZE[Desc]; found ← TRUE END;
        NotExact =>
          -- Assert: page>"iDescU".page AND page<"iDescU + 1".page AND iDescU + 1 = iDescL.
          IF iDescL = iDescOrigin THEN BEGIN pDesc ← pDescFirst; found ← FALSE END
          ELSE
            BEGIN
            pDesc ← FIRST[PDesc] + iDescU*SIZE[Desc];
            IF page <
              baseDesc[pDesc].interval.page + baseDesc[pDesc].interval.count THEN
              found ← TRUE
            ELSE BEGIN pDesc ← pDesc + SIZE[Desc]; found ← FALSE END;
```

```
        END;
      SameAsLastTime => found ← TRUE;
    END;
    ENDLOOP;
  END;

  InsertIfRoom: --INTERNAL--PROCEDURE [desc: Desc]
    RETURNS [success: BOOLEAN, descVictim: Desc] =
    LOOPHOLE[@AwaitInsertIfRoomRequest]; -- an indirect control link to the PORT.
  AwaitInsertIfRoomRequest: --RESPONDING--PORT [
    success: BOOLEAN, descVictim: Desc] RETURNS [desc: Desc];
  -- args/results match InsertIfRoom (but swapped).
  InitializeInsertIfRoom: INTERNAL PROCEDURE
    RETURNS [success: BOOLEAN, descVictim: Desc] --to match PORT args-- =
    BEGIN
    desc: Desc;
    found: BOOLEAN;
    pDesc: PDesc;
    LOOPHOLE[AwaitInsertIfRoomRequest, PrincOps.Port].dest ← Frame.GetReturnLink[
      ]; -- set my PORT call to return to my caller on call below.
    DO
      --FOREVER--
      -- Return result; Await new request; Process it;
      desc ← AwaitInsertIfRoomRequest[success, descVictim];
      Frame.SetReturnLink[LOOPHOLE[AwaitInsertIfRoomRequest, PrincOps.Port].dest];
      -- for debugger
      -- scope of Done --
      BEGIN
      descVictim.dDirty ← FALSE; -- or set descVictim ← "descTop" ?
      IF pDescLast >= pDescMax THEN
        -- Find coldest unpinned descriptor whose swap units are all out, and kick him upstairs:
        BEGIN
        pDescCommOld: PDesc = pDescCommutator;
        DO
          OPEN baseDesc[pDescCommutator];
          IF state ~= checkedOut AND
            ((~hasSwapUnits AND state ~IN In) OR
              (hasSwapUnits AND state ~IN Mapped)) THEN
            BEGIN
            IF dTemperature = FIRST[DTemperature] THEN {
              IF ~dPinned THEN EXIT ELSE --dPinned: skip this desc--NULL}
            ELSE dTemperature ← dTemperature - 1;
            END;
          pDescCommutator ←
            IF pDescCommutator = pDescLast THEN pDescFirst
            ELSE pDescCommutator + SIZE[Desc];
          IF pDescCommutator = pDescCommOld THEN
            BEGIN success ← FALSE; GO TO Done END; -- no available slot in cache.

          ENDLOOP;
        descVictim ← baseDesc[pDescCommutator];
        -- Move descriptors following pDescCommutator one place towards first
        Inline.LongCOPY[
          from: @baseDesc[pDescCommutator + SIZE[Desc]],
          nwords: pDescLast - pDescCommutator, to: @baseDesc[pDescCommutator]];
        pDescLast ← pDescLast - SIZE[Desc];
        END;
      [found, pDesc] ← Find[desc.interval.page];
      --assert--
```



```
      IF found THEN ERROR;
      -- Move descriptors at and following pDesc one place towards last
      pDescLast ← pDescLast + SIZE[Desc];
      LongMoveUp[
        pSource: @baseDesc[pDesc], size: pDescLast - pDesc,
        pSink: @baseDesc[pDesc + SIZE[Desc]]];
      IF pDesc <= pDescCommutator THEN
        pDescCommutator ← pDescCommutator + SIZE[Desc];
      -- Insert desc
      desc.dTemperature ← FIRST[DTemperature];
      baseDesc[pDesc] ← desc;
      success ← TRUE;
      EXITS Done => NULL;
      END;
      ENDLOOP;
    END;

  END.
LOG
-- For previous log entries, see Pilot 3.0 archived version.
Time: February 20, 1980 10:42 AM  By: Knutsen  Action: VM for Region cache pass
**ed in as a parameter. Real mem for region cache allocated proportional to real memory size. Fi
**x Find to not search past beginning of cache.
Time: April 15, 1980 1:09 PM  By: Knutsen  Action: Added AllocateMStoreRuthlessly. De
**allocateClean, InitializeRegionCacheA/B. Rename FirstNotBefore to Find. Make InsertIfFree a
**nd Find a coroutine.
Time: April 25, 1980 2:20 PM  By: Forrest  Action: ControlDefs and PrincOps.
Time: June 16, 1980 5:24 PM  By: Gobbel  Action: Make compatible with new version of
**Desc.
Time: July 30, 1980 1:37 PM  By: Knutsen  Action: Fix Insert to actually kick descs out o
**f the cache. Make compatible with new region desc.
Time: October 3, 1980 10:21 PM  By: Forrest  Action: Change to use PageMap.
**GetF vs MStore.GetState. (Paul actually has module checked out).
Time: October 13, 1980 9:58 AM  By: Forrest/Richard  Action: Change deallocateClean t
**o notice when a region is allVacant, and skip to next
```

```
-- CachedRegionImplB.mesa (last edited by: Gobbel on: October 9, 1980 2:58 PM)

-- Things to consider:

-- 1) Analyze dynamic usage of MStore.Relocate for possible inline cases
-- This program implements operations on regions and swap units and provides higher-level acces
**s to region descriptors.

-- Implementation notes:

-- Exclusive access to the region cache is provided by the monitor lock of CachedRegionImplA. E
**xclusive access to individual region descriptors is provided by the checkedOut field in the descr
**iptors in the cache. Exclusive access to a region is provided by temporarily removing the regio
**n from its normal place in the client's address space.
-- If a region is being remapped, there are two possible windows in which a swap unit may reside.
-- Note that a swap unit which is In may nevertheless have zero real pages assigned, if it is past the
**end of the mapping file.
-- Basically, there are two groups of operations: 1: activate and age; 2: everything else. The first g
**roup may happen at any time, due to page faults and the ReplacementProcess. The second gro
**up are assumed to be used in a sequential fashion by the clients of the Swapper.

-- Caveats:

-- The proper operation of flush, remap, unmap is dependent on there being only one client operat
**ing on a region at a time (exclusive of page faults), and that the client applies the particular oper
**ation (successfully) once for each swap unit in the region. Furthermore, the current implement
**ation demands that there be only one space being remapped at any given time!
-- If a swap unit is in, and its region is inPinned, we promise that it will be continuously present at it
**s assigned address. The way ForkWrite is currently implemented, we may not write the swap un
**it for this reason.

DIRECTORY
  CachedRegion,
  CachedRegionInternal USING [AwaitNotCheckedOut, CheckIn, CheckOut],
  CachedSpace USING [DataOrFile, Desc, Get, Handle, SizeSwapUnit],
  Environment USING [bitsPerWord, Word, wordsPerPage],
  File USING [lastPageNumber, PageNumber],
  FilePageTransfer USING [Initiate, Wait],
  Inline USING [LongCOPY],
  MStore USING [Allocate, Deallocate, Promise, Relocate],
  PageFault USING [Restart],
  PageMap USING [
    Flags, flagsClean, flagsDirty, --flagsDirtyReferenced,--flagsNone,
    flagsNotReferenced, flagsReferenced, flagsVacant, flagsWriteProtected, GetF],
  Process USING [Detach],
  SwapBuffer USING [Allocate, Deallocate],
  SwapperPrograms,
  SwapTask USING [Advance, Fork, State],
  Utilities USING [Bit, BitGet, BitPut, LongPointerFromPage, PageFromLongPointer],
  VM USING [Interval, PageCount, PageNumber, PageOffset],
  VMMapLog USING [
    Descriptor, PatchTable, PatchTableEntry, PatchTableEntryBasePointer,
    PatchTableEntryPointer];

CachedRegionImplB: PROGRAM [pMapLogDesc: LONG POINTER]


  IMPORTS
```



```
    CachedRegionInternal, CachedSpace, FilePageTransfer, Inline, MStore,
    PageFault, PageMap, Process, SwapBuffer, SwapTask, Utilities
  EXPORTS CachedRegion, SwapperPrograms
  SHARES File, PageMap =

  BEGIN OPEN CachedRegion;
  flagsDirtyReferenced: PageMap.Flags =
    [writeProtected: FALSE, dirty: TRUE, referenced: TRUE];
  -- should be in PageMap! (AR 5157)
  Bug: PRIVATE ERROR [BugType] = CODE;
  BugType: TYPE = {
    badSwapUnitSize, copyInButNotSwappable, copyInButReadOnly,
    copyOutButNotSwappable, copyOutToProtected, deadButSwapUnits,
    deadReadOnlyRegion, flushButPinned, funnyAction, funnyOutcome, funnyState,
    makeWritableButNotSwappable, mapButNotUnmapped, mapProtectedDataSpace,
    pinUnmapped, readButAlreadyIn, regionSpaceInconsistent, remapButNotSwappable,
    remapButReadOnly, remapButUnmapped, remapSpaceTooBig, remapToReadOnly,
    swapUnitsOnRealSpace, writeProtectAndNeedsLogging,
    writeProtectedAndNeedsLogging, writeProtectButPinned};

  PatchTableIndex: TYPE = [0..50);
  patchTable: MACHINE DEPENDENT RECORD [
    header: VMMapLog.PatchTable,
    body: ARRAY PatchTableIndex OF VMMapLog.PatchTableEntry];

  inSwappableCold: InSwappable ← inSwappableWarmest;
  -- for age Operation. If region temperature is inSwappableCold and it has not been referenced, i
**t will be swapped out. (A variable so we can experiment with it using the debugger.)

  pageLocationInSpace: PUBLIC PageLocationInSpace ← DESCRIPTOR[NIL, 0];
  -- array supplied by caller of remap. If a swap unit is Out during a remap operation, this specifie
**s which window it is in (the old or the new). Current implementation is: an array of bits, one for
**each page in the space being remapped. We use the bit corresponding to the first page of each
**swap unit to specify the swap unit's location.

  Initialize: PROCEDURE =
    BEGIN
    patchTable.header ←
      [limit: FIRST[VMMapLog.PatchTableEntryPointer],
        maxLimit:
        LOOPHOLE[SIZE[VMMapLog.PatchTableEntry]*LENGTH[patchTable.body]],
        entries:];
    LOOPHOLE[pMapLogDesc, LONG POINTER TO VMMapLog.Descriptor].patchTable ←
      @patchTable.header;
    Process.Detach[FORK PageTransferProcess[]];
    -- region cache must have been initialized previously.

    END;

  Apply: PUBLIC PROCEDURE [pageMember: VM.PageNumber, operation: Operation]
    RETURNS [outcome: Outcome, pageNext: VM.PageNumber] =
    -- The operation acts on the swap unit containing pageMember, except that get, map, and unpi
**n always act on the entire region.
    -- Sets pageNext to the start of the next swap unit or region if a descriptor for the region contai
**ning pageMember is in the region descriptor cache; else sets pageNext to start of the next regi
**on in the cache (possibly having state = missing); if no next region, sets pageNext to pageTop.
    BEGIN
    -- (this procedure body is not indented since it runs for 10 more pages!)
```

```
    region: Desc;
    -- When the region desc is gotten from the region cache, if the region ~hasSwapUnits, region.s
**tate is the truth. If the region hasSwapUnits and region.state IN AliveSwappable, the specific va
**lue of state is meaningless. In this case, region.state is supplied by Apply from [inSwappableCo
**ldest..outAlive] using the page flags. If the region hasSwapUnits and region.state = inPinned, o
**utDead, or unMapped, region.state shows the target state for all of the swap units.
    swapInterval: VM.Interval;
    -- the interval of the swap unit. (If the region does not have swap units, the swap interval is the
**whole region.)
    inOutState: {in, out};
    -- in means that there are pages in this swap unit that are mapped and they are now in memory
**. out means either that there are no mapped pages in this swap unit, or that they are not current
**ly in memory. Note that region.state could be IN In, and yet have inOutState = out, in the case t
**hat there are no mapped pages in the swap unit. Therefore, if an operation demands that a swa
**p unit be in, it must check to see if there are any mapped pages in the swap unit.
    gotMappingSpaceDesc, gotParentSpaceDesc: BOOLEAN;
    mappingSpace, parentSpace: CachedSpace.Desc;
    -- Parent and mapping space may or may not be same.
    ioStarted: BOOLEAN;
    -- Note: as soon as ioStarted is set to TRUE (by ForkRead or ForkWrite), the region descriptor
**must be considered read-only by this procedure (i.e. no further state transitions).

    -- Variables declared before here are referenced globally by Apply's local procedures.
    OldNew: TYPE = {oldWindow, newWindow};

    InWindow: PROCEDURE [pageInSpace: VM.PageNumber] RETURNS [which: OldNew] =
      INLINE
      BEGIN -- Which window is this page in?
      RETURN[
        LOOPHOLE[Utilities.BitGet[
          BASE[pageLocationInSpace], pageInSpace - mappingSpace.interval.page]]]
      END;

    MarkInWindow: PROCEDURE [which: OldNew, pageInSpace: VM.PageNumber] = INLINE
      -- Mark that this page is in which window.
      BEGIN
      Utilities.BitPut[
        LOOPHOLE[which, Utilities.Bit], BASE[pageLocationInSpace],
        pageInSpace - mappingSpace.interval.page]
      END;

    ReadDead: PROCEDURE [
      bufferPage: VM.PageNumber, offsetInSwapUnit: VM.PageOffset,
      count: VM.PageCount] =
      -- Zeroes out given area in buffer and relocates back into the swap unit, marking the pages di
**rty in the process.
      -- (The swap unit must be out and dead, and not writeProtected.)
      BEGIN
      pFirst: LONG POINTER TO Environment.Word;
      offset: VM.PageOffset;
      IF region.writeProtected OR region.needsLogging THEN
        Bug[deadReadOnlyRegion]; -- can't write into readOnly region.
      region.dDirty ← TRUE; -- changing from dead to alive
      FOR offset IN [0..count) DO
        pFirst ← Utilities.LongPointerFromPage[
          bufferPage + offsetInSwapUnit + offset];
        pFirst↑ ← 0;
        Inline.LongCOPY[
```



```
          from: pFirst, nwords: Environment.wordsPerPage - 1, to: pFirst + 1];
        ENDLOOP;
      [] ← MStore.Relocate[
        [bufferPage + offsetInSwapUnit, count],
        swapInterval.page + offsetInSwapUnit, PageMap.flagsNone,
        PageMap.flagsDirty];
      END;

    GetMappingSpaceDesc: PROCEDURE RETURNS [gotIt: BOOLEAN] =
      BEGIN
      IF ~gotMappingSpaceDesc THEN
        BEGIN OPEN region --USING[level.levelMapped,interval]--;
        IF gotParentSpaceDesc AND levelMapped = level THEN
          mappingSpace ← parentSpace
        ELSE
          CachedSpace.Get[
            @mappingSpace, CachedSpace.Handle[levelMapped, interval.page]];
        gotMappingSpaceDesc ← TRUE;
        END;
      IF mappingSpace.state = missing THEN {
        outcome ← [spaceDMissing[region.levelMapped]]; RETURN[gotIt: FALSE]}
      ELSE RETURN[gotIt: TRUE];
      END;

    ForkRead: PROCEDURE [stateNext: State] =
      -- Implicit input parameters: operation.action, region, swapInterval, inOutState, pageMember
**, parentSpace, gotParentSpaceDesc. Implicit output parameters: region.state, region.dDirty, io
**Started, outcome.
      -- Normal operation: The swap unit must be out. Starts input of the swap unit from the mapp
**ing space's window. If the swap unit is dead, no reading is done, but instead real memory is allo
**cated, initialized to zero, and marked dirty (and region.dDirty is set, reflecting the transition fro
**m dead to alive). At completion of input, flags are set on all pages using region.writeProtected a
**nd region.needsLogging. On return, region.state is set to stateNext.
      -- Operation for copyIn[from]: The swap unit may be in or out. If it is in, it is data from the cur
**rent window. Reads leading pages from the other window "from" and makes them dirty, then r
**eads (as required) further pages from the current window. If no pages are required from the oth
**er window, ForkRead is a no-op.
      -- Operation for remapB[from]: mappingSpace will have changed to the new window. "from
**" describes the old window. The swap unit may be in or out. If it is In, it is data from the old win
**dow. Reads in (as required) leading pages from the old window, then reads (as required) furthe
**r pages from the current window, then deallocates (as required) any further pages not mapped i
**n the current window. Any pages that came from the old window are marked dirty.
      -- Operation notes:
      -- If the mapping space desc is not available, outcome will bet set to [spaceDMissing[]], and
**no other action taken.
      -- If any reading is necessary, it is initiated and ioStarted set to TRUE. At this point, the regio
**n descriptor must be read-only to Apply, since a copy of it is bundled with the read request ("Sw
**apTask"). The PageTransferProcess (see below) completes the input, and when complete, che
**cks in the (copy of the) region descriptor.
      -- Real memory is allocated and freed as necessary.
      BEGIN
      buffer: VM.Interval;
      -- buffer for swapping. Size of mapped part of swap unit.
      offsetInSpace: VM.PageOffset; -- offset of swap unit in mapping space.
      countmapped: VM.PageCount; -- number of mapped pages in this swap unit.
      currentState: State ← region.state;
      -- the state of the region at entry to ForkRead.
      IF ~GetMappingSpaceDesc[].gotIt THEN RETURN;
```

```
    -- (outcome = [spaceDMissing] has been set.)
    region.state ← stateNext;
    -- sets the state of the region when IO is completed (so we're committed at this point).
    IF region.writeProtected AND ~mappingSpace.writeProtected THEN
      ERROR Bug[regionSpaceInconsistent];
    -- region/space unallowably inconsistent.
    offsetInSpace ← swapInterval.page - mappingSpace.interval.page;
    countmapped ←
      IF mappingSpace.countMapped <= offsetInSpace THEN 0
      ELSE MIN[
        swapInterval.count, CARDINAL[mappingSpace.countMapped - offsetInSpace]];
    buffer ← SwapBuffer.Allocate[countmapped];
    WITH operation SELECT FROM
      copyIn --[from]-- =>
        BEGIN
        offsetInOtherSpace: VM.PageOffset =
          swapInterval.page - from.interval.page;
        -- offset of swap unit in other space.
        countmappedOther: VM.PageCount
          -- number of mapped pages in this swap unit mapped in "from" (never more than count
**mapped).
          =
          IF from.countMapped <= offsetInOtherSpace THEN 0
          ELSE MIN[
            countmapped, CARDINAL[from.countMapped - offsetInOtherSpace]];
        notMappedInOther: VM.PageCount
          -- number of mapped pages in this swap unit not mapped in "from".
          =
          IF countmapped <= countmappedOther THEN 0
          ELSE CARDINAL[countmapped - countmappedOther];
        IF countmappedOther > 0 THEN -- there's something to do..
          BEGIN
          countToTransfer: VM.PageCount =
            countmappedOther +
              (IF inOutState = out AND currentState ~= outDead THEN
                notMappedInOther ELSE 0);
          IF region.writeProtected OR region.needsLogging THEN
            Bug[copyInButReadOnly]; -- can't dirty it and writeProtect it.
          [] ← SwapTask.Fork[
            [swapInterval.page, buffer.count], PageMap.flagsDirty, buffer.page,
            --countRemaining:--countToTransfer, region];
          ioStarted ← TRUE; -- since we know countToTransfer>0.
          IF inOutState = out THEN
            BEGIN
            MStore.Allocate[buffer];
            IF currentState = outDead THEN
              -- zero out trailing pages only mapped in current window.
              ReadDead[
                bufferPage: buffer.page, offsetInSwapUnit: countmappedOther,
                count: notMappedInOther]
            ELSE --currentState = outAlive--
              [] ← FilePageTransfer.Initiate[
                [ -- read in trailing pages only mapped in current window.
                  file: mappingSpace.window.file,
                  filePage:
                    mappingSpace.window.base + offsetInSpace + countmappedOther,
                  memoryPage: buffer.page + countmappedOther,
                  count: notMappedInOther,
```



```
                  opSpecific: read[priorityPage: File.lastPageNumber]]];
            END
          ELSE --inOutState = in--
            [] ← MStore.Relocate[
              [swapInterval.page, countmappedOther], buffer.page,
              PageMap.flagsNone, PageMap.flagsClean];
            -- move the real mem under pages mapped in other window up to swap buffer.
            [] ← FilePageTransfer.Initiate[
              [ -- always read in pages mapped in other window.
                file: from.window.file,
                filePage: from.window.base + offsetInOtherSpace,
                memoryPage: buffer.page, count: countmappedOther,
                opSpecific: read[priorityPage: File.lastPageNumber]]]
            END;
          END;
      remapB --[from]-- =>
        -- Note: there's a lot of code in common here with copyIn.  It is kept separate so that we c
**an conveniently throw it away when Remap is retired.
        BEGIN
        countmappedOther: VM.PageCount
          -- number of mapped pages in this swap unit coming from "from" (never more than cou
**ntmapped).
          =
          IF from.countMapped <= offsetInSpace THEN 0
          ELSE MIN[countmapped, CARDINAL[from.countMapped - offsetInSpace]];
        notMappedInOther: VM.PageCount
          -- number of mapped pages in this swap unit not mapped in "from".
          =
          IF countmapped <= countmappedOther THEN 0
          ELSE CARDINAL[countmapped - countmappedOther];
        countToTransfer: VM.PageCount =
          notMappedInOther +
            (IF inOutState = out AND currentState ~= outDead THEN
              countmappedOther ELSE 0);
        IF region.writeProtected OR region.needsLogging THEN
          Bug[remapButReadOnly]; -- can't dirty it and writeProtect it.
        [] ← SwapTask.Fork[
          [swapInterval.page, buffer.count], PageMap.flagsDirty, buffer.page,
          --countRemaining:--countToTransfer, region];
        ioStarted ← (countToTransfer > 0);
        IF inOutState = out THEN
          BEGIN
          MStore.Allocate[buffer];
          IF currentState = outDead THEN
            -- zero out leading pages mapped in other (old) window.
            ReadDead[
              bufferPage: buffer.page, offsetInSwapUnit: 0,
              count: countmappedOther]
          ELSE --currentState = outAlive--
            [] ← FilePageTransfer.Initiate[
              [ -- read in leading pages mapped in other (old) window.
                file: from.window.file,
                filePage: from.window.base + offsetInSpace,
                memoryPage: buffer.page, count: countmappedOther,
                opSpecific: read[priorityPage: File.lastPageNumber]]];
          END
        ELSE --inOutState = in--
          -- (note that the stuff that is in is from the old window.)
```

```
        {
        [] ← MStore.Relocate[
          [swapInterval.page, countmappedOther], swapInterval.page,
          PageMap.flagsReferenced, PageMap.flagsDirty];
        -- make the pages from the old window that are in be dirty.
        MStore.Deallocate[
          [page: swapInterval.page + countmapped,
            count: swapInterval.count - countmapped], --promised:--FALSE];
        -- free any pages no longer mapped by current (new) window.
        MStore.Allocate[
          [page: buffer.page + countmappedOther, count: notMappedInOther]]};
      -- allocate any pages that weren't mapped by old window.
      [] ← FilePageTransfer.Initiate[
        [ -- always read in any pages only mapped in current (new) window.
          file: mappingSpace.window.file,
          filePage:
          mappingSpace.window.base + offsetInSpace + countmappedOther,
          memoryPage: buffer.page + countmappedOther, count: notMappedInOther,
          opSpecific: read[priorityPage: File.lastPageNumber]]];
    END;
  ENDCASE --all normal operations-- =>
    IF countmapped > 0 THEN -- there's something to do..
      BEGIN
      IF inOutState = in THEN ERROR Bug[readButAlreadyIn];
      MStore.Allocate[buffer];
      IF currentState = outDead THEN
        -- zero out buffer area and relocate back into region
        ReadDead[
          bufferPage: buffer.page, offsetInSwapUnit: 0, count: countmapped]
      ELSE --currentState = outAlive--{
        [] ← SwapTask.Fork[
          [swapInterval.page, buffer.count],
          IF region.writeProtected OR region.needsLogging THEN
          PageMap.flagsWriteProtected ELSE PageMap.flagsClean, buffer.page,
          --countRemaining:--countmapped, region];
        ioStarted ← TRUE; -- since we know countmapped>0.
        [] ← FilePageTransfer.Initiate[
          [file: mappingSpace.window.file,
            filePage: mappingSpace.window.base + offsetInSpace,
            memoryPage: buffer.page, count: countmapped,
            opSpecific: read[
            priorityPage:
            mappingSpace.window.base + offsetInSpace +
              (pageMember - swapInterval.page)]]]};
      END;
    IF ~ioStarted THEN SwapBuffer.Deallocate[buffer];
    -- (If ioStarted, PageTransferProcess will deallocate the buffer.)

  END;

ForkWrite: PROCEDURE [stateNext: State] =
  -- Implicit input parameters: operation.action, region, swapInterval, inOutState, pageMember
**, parentSpace, gotParentSpaceDesc. Implicit output parameters: region.state, region.dDirty, io
**Started, outcome.
  -- Normal operation: The swap unit must be in. Starts output of the swap unit to the mapping
**space's window. If the swap unit is clean, no writing is done, but instead the real memory is just
**freed. At competion of input, flags are set on all pages using region.writeProtected and region.
**needsLogging. On return, region.state is set to stateNext.
```



```
  -- Operation for copyOut[to]: The swap unit must be in. Writes leading pages to the other wi
**ndow "to" - even if they are clean. stateNext must be equal to the current region.state.
  -- Operation notes:
  -- If the mapping space desc is not available, outcome will bet set to [spaceDMissing[]], and
**no other action taken.
  -- If any writing is necessary, it is initiated and ioStarted set to TRUE. At this point, the region
**descriptor must be read-only to Apply, since a copy of it is bundled with the write request ("Swa
**pTask"). The PageTransferProcess (see below) completes the input, and when complete, chec
**ks in the (copy of the) region descriptor.
  -- Real memory is freed as appropriate.
  BEGIN
  region.state ← stateNext; -- valid after I/O, if any, is competed
  WITH operation SELECT FROM
    copyOut --[to]-- => -- move this code to main Apply copyOut arm?
      IF GetMappingSpaceDesc[].gotIt THEN
        BEGIN
        offsetInSpace: VM.PageOffset =
          swapInterval.page - mappingSpace.interval.page;
        -- offset of swap unit in mapping space.
        countmapped: VM.PageCount =
          IF mappingSpace.countMapped <= offsetInSpace THEN 0
          ELSE MIN[
            swapInterval.count, CARDINAL[
              mappingSpace.countMapped - offsetInSpace]];
        -- number of mapped pages in this swap unit.
        offsetInOtherSpace: VM.PageOffset =
          swapInterval.page - to.interval.page;
        -- offset of swap unit in other space.
        countmappedOther: VM.PageCount =
          IF to.countMapped <= offsetInOtherSpace THEN 0
          ELSE MIN[countmapped, to.countMapped - offsetInOtherSpace];
        -- number of mapped pages in this swap unit going to "to" (never more than countmap
**ped).
        IF region.writeProtected AND ~mappingSpace.writeProtected THEN
          ERROR Bug[regionSpaceInconsistent];
        -- region/space unallowably inconsistent.
        IF countmappedOther > 0 THEN -- there's something to do..
          BEGIN
          buffer: VM.Interval = SwapBuffer.Allocate[countmappedOther];
          flags: PageMap.Flags = MStore.Relocate[
            [swapInterval.page, countmappedOther], buffer.page,
            PageMap.flagsNone, PageMap.flagsWriteProtected].flags;
          -- relocated and protected during write to delay client references in order to allow us t
**o maintain the referenced and dirty status bits, and to assure that an I/O device can compute a
**consistent checksum. Protecting it is also a flag to PageTransferProcess that these pages are b
**eing written, not read, so don't apply patches.
          [] ← SwapTask.Fork[
            [swapInterval.page, countmappedOther], flags, buffer.page,
            --countRemaining:--countmappedOther, region];
          ioStarted ← TRUE;
          [] ← FilePageTransfer.Initiate[
            [ -- always write pages mapped in other window.
              file: to.window.file,
              filePage: to.window.base + offsetInOtherSpace,
              memoryPage: buffer.page, count: countmappedOther,
              promise: FALSE,
              -- since we are leaving the stuff in memory, we don't need to promise it.
              opSpecific: write[]]];
```

```
        END;
      END;
    ENDCASE --all normal operations-- =>
      BEGIN
      flagsFirst: PageMap.Flags = PageMap.GetF[
        swapInterval.page].valueOld.flags;
      IF flagsFirst ~= PageMap.flagsVacant AND ~flagsFirst.writeProtected THEN
        -- there may be something to do
        BEGIN
        buffer: VM.Interval = SwapBuffer.Allocate[swapInterval.count];
        -- could actually scan nonvacant prefix of swapInterval
        flags: PageMap.Flags ← MStore.Relocate[
          swapInterval, buffer.page, PageMap.flagsNone,
          PageMap.flagsWriteProtected].flags;
        -- promise the real mem if it will be freed.
        flags.writeProtected ← region.writeProtected OR region.needsLogging;
        IF ~flags.dirty -- matches backing file
          THEN
        BEGIN -- no writing required..
        IF stateNext IN In THEN -- put it back where it was
          [] ← MStore.Relocate[
            buffer, swapInterval.page, PageMap.flagsNone, flags]
        ELSE MStore.Deallocate[interval: buffer, promised: FALSE];
        -- throw it away
        SwapBuffer.Deallocate[buffer];
        END
      ELSE -- writing required..
        IF GetMappingSpaceDesc[].gotIt THEN -- need mapping space descriptor
          BEGIN
          offsetInSpace: VM.PageOffset =
            swapInterval.page - mappingSpace.interval.page;
          countmapped: VM.PageCount =
            IF mappingSpace.countMapped <= offsetInSpace THEN 0
            ELSE MIN[
              swapInterval.count, CARDINAL[
                mappingSpace.countMapped - offsetInSpace]];
          promise: BOOLEAN = stateNext ~IN In;
          countInitiated: VM.PageCount;
          IF region.writeProtected AND ~mappingSpace.writeProtected THEN
            ERROR Bug[regionSpaceInconsistent];
          -- region/space unallowably inconsistent.
          flags.dirty ← FALSE;
          [] ← SwapTask.Fork[
            [swapInterval.page, buffer.count], flags, buffer.page,
            countmapped, region];
          ioStarted ← TRUE; -- since we know countmapped>0.
          countInitiated ← FilePageTransfer.Initiate[
            [file: mappingSpace.window.file,
              filePage: mappingSpace.window.base + offsetInSpace,
              memoryPage: buffer.page, count: countmapped, promise: promise,
              opSpecific: write[]]];
          IF promise THEN MStore.Promise[countInitiated];
          -- promise the first batch of pages being written, FilePageTransfer will promise the r
**est.

          END
        ELSE -- couldn't find mappingSpace--
          BEGIN
          [] ← MStore.Relocate[
```



```
            buffer, swapInterval.page, PageMap.flagsNone, flags];
          SwapBuffer.Deallocate[buffer];
          END;
        END
      ELSE --was vacant or writeProtected--
        IF stateNext ~IN In THEN
          MStore.Deallocate[interval: swapInterval, promised: FALSE];
      END;
    END;

  -- Main body of Apply
  endRegion: VM.PageNumber;
  gotMappingSpaceDesc ← FALSE;
  gotParentSpaceDesc ← FALSE;
  ioStarted ← FALSE;
  region ← CachedRegionInternal.CheckOut[pageMember, operation.ifCheckedOut];
  pageNext ← endRegion ← region.interval.page + region.interval.count;
  -- assume ~hasSwapUnits
  outcome ← [ok[]];
  SELECT region.state FROM
    = checkedOut => --Operation.ifCheckedOut = skip--NULL;
    = missing =>
      -- in this case, the value returned for pageNext is the start of the next region known in the r
**egion cache (see CheckOut).
      IF operation.ifMissing = report THEN outcome ← [regionDMissing[]]
      ELSE --operation.ifMissing = skip--NULL;
    IN DescAvailable =>
      BEGIN OPEN region; --scope of Exit, etc.--
      BEGIN
      IF beingFlushed AND operation.action ~= flush THEN
        -- we pretend the desc has already been flushed.
        {
        IF operation.ifMissing = report THEN outcome ← [regionDMissing[]];
        --ELSE -operation.ifMissing = skip- NULL;
        GO TO Exit};
      IF operation.action ~= get THEN
        -- (swapInterval, etc., not needed for get)
        BEGIN --non-get action--
        IF ~hasSwapUnits THEN
          BEGIN
          swapInterval ← interval;
          inOutState ←
            IF PageMap.GetF[swapInterval.page].valueOld.flags =
            PageMap.flagsVacant THEN out ELSE in;
          END
        ELSE --hasSwapUnits--
          BEGIN -- set interval and state of the swap unit:
          CachedSpace.Get[
            @parentSpace, CachedSpace.Handle[
            level: level, page: interval.page]];
          -- get desc of immediate parent to get sizeSwapUnit.
          IF parentSpace.state = missing THEN {
            outcome ← [spaceDMissing[level]]; GO TO Exit}
          ELSE --parent space desc available--
            BEGIN
            gotParentSpaceDesc ← TRUE;
            IF parentSpace.sizeSwapUnit ~IN CachedSpace.SizeSwapUnit OR
              ~parentSpace.hasSwapUnits THEN ERROR Bug[badSwapUnitSize];
            -- improper use or region/space inconsistent.
```

```
        swapInterval.page ←
          pageMember -
            ((pageMember - region.interval.page) MOD
              parentSpace.sizeSwapUnit);
        pageNext ← MIN[
          endRegion, swapInterval.page + parentSpace.sizeSwapUnit];
        -- the start of the next swap unit or region (the last swap unit may be short).
        swapInterval.count ← pageNext - swapInterval.page;
        inOutState ←
          IF PageMap.GetF[swapInterval.page].valueOld.flags =
          PageMap.flagsVacant THEN out ELSE in;
        -- look at page flags to find if this swap unit is in or out:
        IF region.state IN Swappable THEN
          -- (leave unmapped and inPinned alone)
          BEGIN
          IF region.state = outDead THEN Bug[deadButSwapUnits];
          region.state ←
            IF inOutState = out THEN outAlive
            -- no way to tell whether dead or alive!
            ELSE inSwappableCold;
          -- must be inSwappableCold if swap units are ever going to be aged out.

          END;
        END;
      END;
    END; --non-get action--
  WITH operation SELECT FROM
    activate --[why]-- =>
      -- (activate[why:activate] is a hint and is legal on unmapped regions.)
      IF state = unmapped THEN {
        IF why = pagefault THEN GO TO AddressFault
        ELSE --why = activate--NULL} -- ignore the hint

      ELSE --state IN Mapped--
        BEGIN
        IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit;
        -- (outcome = [spaceDMissing] has been set.)
        IF why = pagefault AND pageMember >=
          mappingSpace.interval.page + mappingSpace.countMapped THEN
          GO TO AddressFault;
        IF inOutState = out THEN
          BEGIN
          IF mappingSpace.state = beingRemapped THEN
            -- the space desc has changed to the new window
            IF InWindow[swapInterval.page] = oldWindow THEN GO TO Exit;
          -- the swap unit needs to be read from the old window (but the space desc has chan
**ged to the new window and we don't know the old window right now). We ignore the request for
**now, and this (pagefault) request will be satisfied when the region is read in by remapB.
          ForkRead[
            stateNext:
            IF state = inPinned THEN inPinned ELSE inSwappableWarmest]
          -- inSwappableWarmest is ok here for swap units. ForkRead will take care of any de
**ad-to-alive transition.

          END -- ELSE + + inOutState = in + +
        -- IF why = pagefault THEN NULL + + the page is now in (it has been brought in since
**the pagefault)
        -- ELSE + + why = activate + + NULL; + + it was already in! That's fine.
```



```
        END;
      age =>
        IF state IN InSwappable THEN
          BEGIN
          IF MStore.Relocate[
            swapInterval, swapInterval.page, PageMap.flagsNotReferenced,
            PageMap.flagsNone].flags.referenced THEN
            -- (gets referenced bit and resets it)
            state ← inSwappableWarmest
            -- inSwappableWarmest is ok here for swap units.

          ELSE
            IF state > inSwappableCold THEN state ← PRED[state]
            ELSE --state = inSwappableCold--
              BEGIN
              IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit;
              IF mappingSpace.state = beingRemapped THEN
                -- the space desc has changed to the new window
                MarkInWindow[newWindow, swapInterval.page];
              -- client has dirtied this swap unit when it needs to be in and dirtied, now the regio
**n is ageing out. We write the region into the new window, and mark that it is there. May attempt
**to write some vacant pages! See AR2382.
              ForkWrite[stateNext: outAlive];
              END;
          END;
      clean --[andWriteProtect, andNeedsLogging]-- =>
        BEGIN
        IF state IN Swappable THEN
          BEGIN
          IF state = outDead THEN {
            ForkRead[stateNext: inSwappableWarmest];
            -- get the stuff in and dirty to assure initialized to zeroes.
            GO TO Retry} -- better luck next time.

          ELSE --state IN AliveSwappable--
            BEGIN
            writeProtected ← writeProtected OR andWriteProtect;
            needsLogging ← needsLogging OR andNeedsLogging;
            IF needsLogging THEN {
              writeProtected ← FALSE;
              --IF andWriteProtect THEN Bug[writeProtectAndNeedsLogging]--
              };
            dDirty ← TRUE; -- change in writeProtect/needsLogging status.
            IF inOutState = in THEN ForkWrite[stateNext: state]
            -- makes it clean, and sets the flags.

            END;
          END
        ELSE NULL; -- a no-op on pinned and unmapped regions.

        END;
      copyIn --[from]-- => -- input from the specified window.
        {
        IF state ~IN Swappable THEN Bug[copyInButNotSwappable];
        needsLogging ← FALSE;
        dDirty ← TRUE;
        ForkRead[stateNext: inSwappableWarmest]};
      -- (additional copyIn logic is in ForkRead.)
```

```
        copyOut --[to]-- => -- output to the specified window.
          BEGIN
          IF state ~IN Swappable THEN Bug[copyOutButNotSwappable];
          IF inOutState = out THEN -- get the swapUnit in..
            BEGIN
            IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit;
            -- better luck next time.
            IF swapInterval.page >=
              mappingSpace.interval.page + mappingSpace.countMapped THEN {
              outcome ← [ok[]]; GO TO Exit};
            -- Done: no mapped pages in swap unit.
            ForkRead[stateNext: inSwappableWarmest];
            GO TO Retry; -- better luck later when it's in.

            END
          ELSE --inOutState = in--{
            IF to.writeProtected THEN Bug[copyOutToProtected];
            ForkWrite[stateNext: state]};
          -- (additional copyOut logic is in ForkWrite.)

          END;
        createSwapUnits => {
          IF state = outDead THEN state ← outAlive;
          -- we don't allow swap units to be outDead.
          hasSwapUnits ← TRUE;
          dDirty ← TRUE;
          pageNext ← endRegion}; -- acts on whole region.

        deactivate =>
          IF state IN InSwappable THEN ForkWrite[stateNext: outAlive];
        flush =>
          -- flush all swap units of the region from memory and the region descriptor from the cac
**he.
          -- the caller must, in ascending order, call flush once successfully for each swap unit of
**the region!
          BEGIN
          IF state = inPinned THEN Bug[flushButPinned];
          IF dDirty THEN outcome ← [regionDDirty[]]
          ELSE --desc is clean, region not pinned--
            BEGIN
            beingFlushed ← TRUE;
            -- desc will pretend it's missing until it really is. (desc does not become dirty from this
**statement)
            IF pageNext < endRegion THEN
              --the caller will flush more swap units later--{
              IF inOutState = in THEN ForkWrite[stateNext: outAlive]
              ELSE --inOutState = out--NULL}
            ELSE --this is last swap unit of the region--{
              IF inOutState = in THEN ForkWrite[stateNext: missing]
              ELSE -- inOutState = out --state ← missing};
            -- the beingFlushed state will terminate when the desc vanishes (becomes missing).

            END;
          END;
        get --[andResetDDirty, pDescResult]-- =>
          -- "get (whole) region descriptor"
          {
          pageNext ← endRegion; -- acts on whole region.
          pDescResult↑ ← region;
```



```
          IF andResetDDirty THEN dDirty ← FALSE};
        kill --[andDeallocate]-- =>
          -- this is a hint. Ignored for unmapped or pinned regions. andDeallocate is not a hint. it
**is always done. andDeallocate is not used in the current implementation, but may be useful in t
**he future.
          IF state IN AliveSwappable THEN -- there may be something to do..
            {
            IF inOutState = in THEN
              MStore.Deallocate[interval: swapInterval, promised: FALSE];
            -- (if it's writeProtected or needsLogging, it must be clean, so we can throw it away.)
            state ←
              IF writeProtected OR needsLogging OR hasSwapUnits THEN outAlive
              ELSE outDead;
            dDirty ← TRUE};
        makeWritable => {
          IF state ~IN Swappable THEN Bug[makeWritableButNotSwappable];
          writeProtected ← FALSE;
          IF ~hasSwapUnits THEN needsLogging ← FALSE;
          [] ← MStore.Relocate[
            swapInterval, swapInterval.page, --PageMap.--flagsDirtyReferenced,
            PageMap.flagsNone]};
        map --[level, backFileType, andWriteProtect, andNeedsLogging]-- =>
          BEGIN
          IF state ~= unmapped THEN Bug[mapButNotUnmapped];
          pageNext ← endRegion; -- acts on whole region.
          writeProtected ← andWriteProtect;
          needsLogging ← andNeedsLogging;
          IF writeProtected AND needsLogging THEN
            Bug[writeProtectedAndNeedsLogging];
          levelMapped ← level;
          dDirty ← TRUE;
          SELECT backFileType FROM
            file => state ← outAlive;
            data => {
              IF writeProtected OR needsLogging THEN Bug[mapProtectedDataSpace];
              state ← IF hasSwapUnits THEN outAlive ELSE outDead};
            -- it's not convenient (or very useful) to have "all swap units are outDead".

            none => {
              IF hasSwapUnits THEN Bug[swapUnitsOnRealSpace];
              -- can't repeat for each swap unit.
              state ← outDead;
              ForkRead[stateNext: inPinned]};
            -- allocates real memory and pins it; no actual reading since dead.

            ENDCASE;
          END;
        noOp => NULL;
        pin =>
          IF state = unmapped THEN Bug[pinUnmapped]
          ELSE
            IF inOutState = out THEN ForkRead[stateNext: inPinned]
            ELSE --inOutState = in--state ← inPinned;
        -- ok to pin a (swap unit of a) pinned region

        remapA --[firstClean]-- =>
          -- (implicit parameter: pageLocationInSpace.) There will soon be two windows associat
**ed with this swap unit. We must keep track of which window the most recent copy of each swap
```

```
**unit is in.
        BEGIN
        IF state ~IN Swappable THEN Bug[remapButNotSwappable];
        IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit;
        -- better luck next time.
        IF
          (mappingSpace.interval.count + Environment.bitsPerWord -
            1)/Environment.bitsPerWord > LENGTH[pageLocationInSpace] THEN
          Bug[remapSpaceTooBig];
        MarkInWindow[oldWindow, swapInterval.page];
        IF inOutState = in AND firstClean THEN ForkWrite[stateNext: state];
        END;
      remapB --[from]-- =>
        -- the new window has arrived. Make sure that the swap unit is in the new window, or is
**In and dirty.
        BEGIN
        IF state = unmapped THEN Bug[remapButUnmapped];
        IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit;
        -- better luck next time.
        IF mappingSpace.writeProtected THEN Bug[remapToReadOnly];
        region.writeProtected ← FALSE;
        region.needsLogging ← FALSE;
        dDirty ← TRUE;
        IF InWindow[swapInterval.page] = oldWindow THEN
          ForkRead[stateNext: inSwappableWarmest];
        -- even if it's currently In, to get growth from new window (additional remapB logic in Fo
**rkRead).
        -- MarkInWindow[newWindow, swapInterval.page]; + + when the region is next written
**, it will be written to the new window, and marked as such by ForkWrite.

        END;
      unmap =>
        BEGIN
        wasPinned: BOOLEAN = (state = inPinned);
        dDirty ← TRUE; -- (mapped to unmapped transition)
        IF inOutState = out THEN state ← unmapped
        ELSE --inOutState = in--{
          IF ~GetMappingSpaceDesc[].gotIt THEN GO TO Exit; -- try again later.
          IF mappingSpace.dataOrFile = CachedSpace.DataOrFile[data] THEN {
            MStore.Deallocate[interval: swapInterval, promised: FALSE];
            -- no need to swap out a vanishing data space.
            state ← unmapped}
          ELSE ForkWrite[stateNext: unmapped]};
        IF wasPinned AND outcome = [ok[]] THEN
          outcome ← [notePinned[levelMax: level]];
        END;
      unpin => {
        pageNext ← endRegion; -- acts on whole region.
        IF state = inPinned THEN state ← inSwappableWarmest};
      -- ok for swap units.

      ENDCASE --operation-- => ERROR Bug[funnyAction];
    EXITS
      Exit => NULL; -- outcome has been set already.

      AddressFault => outcome ← [error[region.state]];
      Retry => outcome ← [retry[]];
    END --scope of Exit, etc.--
    ;
```



```
      IF ~ioStarted THEN {
        CachedRegionInternal.CheckIn[desc: region];
        WITH outcome SELECT FROM
          ok, notePinned => PageFault.Restart[region.interval];
          error, regionDDirty, regionDMissing, retry, spaceDMissing => NULL;
          ENDCASE => Bug[funnyOutcome]}
      ELSE --ioStarted--
        IF operation.afterForking = wait THEN
          CachedRegionInternal.AwaitNotCheckedOut[pageMember];
      END --DescAvailable, USING region--
    ;
    ENDCASE --region.state-- => ERROR Bug[funnyState];
  END --Apply--
; -- (this procedure body is not indented since it runs for 10 more pages!)

PageTransferProcess: PROCEDURE =
  -- Completes the I/O started by ForkRead and ForkWrite (q.v.).
  BEGIN
  resartOnLeadingPages: BOOLEAN ← TRUE;
  -- (a variable so we can experiment using CoPilot.)
  pageRunInterval, bufferInterval: VM.Interval;
  task: SwapTask.State;
  DO
    --FOREVER--
    bufferInterval ← FilePageTransfer.Wait[];
    task ← SwapTask.Advance[bufferInterval];
    pageRunInterval ← VM.Interval[
      task.swapInterval.page + (bufferInterval.page - task.bufferPage),
      bufferInterval.count];
    SELECT task.region.state FROM
      IN In =>
        BEGIN
        -- If this interval is write protected, it is because it is being written (not being read). If it is
**being written, we should not apply patches. If it is being read, we should apply patches.
        IF ~PageMap.GetF[bufferInterval.page].flags.writeProtected THEN
          ApplyPatches[pageRunInterval, bufferInterval.page];
        [] ← MStore.Relocate[
          bufferInterval, pageRunInterval.page, PageMap.flagsNone, task.flags];
        -- relocate pageRunInterval back to region.
        IF task.countRemaining ~= 0 AND resartOnLeadingPages THEN
          PageFault.Restart[pageRunInterval];
        -- restart client for leading pages of swap unit.

        END;
      ENDCASE -- unmapped, missing, IN Out -- =>
        MStore.Deallocate[interval: bufferInterval, promised: TRUE];
    -- (all pages being written were promised.)
    IF task.countRemaining = 0 THEN
      BEGIN
      SwapBuffer.Deallocate[
        VM.Interval[task.bufferPage, task.swapInterval.count]];
      -- free any real memory left in the buffer.
      CachedRegionInternal.CheckIn[desc: task.region];
      PageFault.Restart[task.region.interval];
      -- restart this client for last page, and all clients faulting anywhere in the entire region.

      END;
    ENDLOOP;
```

```
    END;

  ApplyPatches: PROCEDURE [interval: VM.Interval, pageBuffer: VM.PageNumber] =
    INLINE
    BEGIN
    OPEN
      pT: LOOPHOLE[pMapLogDesc, LONG POINTER TO VMMapLog.Descriptor].patchTable;
    entryPtr: VMMapLog.PatchTableEntryPointer;
    entrySize: CARDINAL = SIZE[VMMapLog.PatchTableEntry];
    firstEntry: VMMapLog.PatchTableEntryPointer = FIRST[
      VMMapLog.PatchTableEntryPointer];
    FOR entryPtr ← firstEntry, entryPtr + entrySize WHILE entryPtr # pT.limit DO
      OPEN
        e: LOOPHOLE[@pT.entries[0], VMMapLog.PatchTableEntryBasePointer][
          entryPtr];
      IF Utilities.PageFromLongPointer[e.address] IN
        [interval.page..interval.page + interval.count) THEN
        BEGIN
        offset: LONG INTEGER =
          e.address - Utilities.LongPointerFromPage[interval.page];
        (Utilities.LongPointerFromPage[pageBuffer] + offset)↑ ← e.value;
        END;
      ENDLOOP;
    END;

  Initialize[];
  END.
```

(For earlier log entries see Pilot 4.0 archive version.)

```
April 24, 1980  11:54 AM        Knutsen   PageTransferProcess must RestartFaulted on entire re
**gion.
April 30, 1980  5:24 PM         Gobbel    Make outputSpecial start a read and return with outco
**me = retry if region not in.
August 5, 1980  3:16 PM         Knutsen   Implement makeWritable, redo copyIn, copyOut. Imple
**ment needsLogging versus writeProtect.  inOutState now has only two states.
September 4, 1980  10:54 PM     Gobbel    Fix copyIn bug, add some error checking.
September 16, 1980  1:47 PM     Knutsen   Make CopyIn/Out handle short-mapped windows.  Re
**work ForkRead/Write.  ApplyPatches using GetF instruction.  Fix another Remap bug.
September 19, 1980  6:53 PM     McJones   AR 5889; ForkWrite[copyOut] must put back real memo
**ry if no I/O started
September 24, 1980  1:00 PM     McJones   Change ForkWrite once again to get mappingSpace de
**scriptor only if region is dirty
September 26, 1980  4:26 PM     McJones   ForkWrite forgot to relocate memory if spaceDmissing
October 9, 1980  2:58 PM        Gobbel    Don't turn off needsLogging for region with uniform sw
**ap units.
```

```
-- CachedSpaceImpl.mesa (last edited by: McJones on: September 24, 1980 2:47 PM)

-- Things to consider:
-- 1) To allow multi-MDS, move global data to monitored record

DIRECTORY
  CachedSpace,
  Frame USING [Alloc],
  PilotSwitches USING [switches --.u--],
  RuntimeInternal USING [MakeFsi],
  SwapperPrograms;

CachedSpaceImpl: MONITOR
  IMPORTS Frame, PilotSwitches, RuntimeInternal
  EXPORTS CachedSpace, SwapperPrograms =
  BEGIN OPEN CachedSpace;

  CacheEntry: TYPE = RECORD [pCENext: PCE, desc: Desc];
  PCE: TYPE = POINTER TO CacheEntry;
  ICE: TYPE = [0..60); -- algorithm assumes at least two entries

  -- Monitor data:
  rgCE: ARRAY ICE OF CacheEntry;
  pCEMru: PCE;
  pCEFree: PCE;

  -- Interpretation of handles is not "strict": page may be any page of space
  FindCacheEntry: INTERNAL PROC [space: Handle]
    RETURNS [found: BOOLEAN, pCE: PCE] =
    BEGIN
    pCECur, pCEPrev: PCE;
    FOR pCECur ← pCEMru, pCECur.pCENext WHILE pCECur # NIL DO
      OPEN pCECur;
      IF space.level = desc.level AND space.page IN
        [desc.interval.page..desc.interval.page + desc.interval.count) THEN
        GO TO Found;
      pCEPrev ← pCECur;
      REPEAT
        Found =>
          BEGIN
          IF pCECur # pCEMru THEN
            BEGIN
            pCEPrev.pCENext ← pCENext;
            pCENext ← pCEMru;
            pCEMru ← pCECur
            END;
          RETURN[TRUE, pCECur]
          END;
        FINISHED => RETURN[FALSE, NIL];
      ENDLOOP
    END;

  Delete: PUBLIC ENTRY PROC [space: Handle] =
    BEGIN
    found: BOOLEAN;
    pCE: PCE;
    [found, pCE] ← FindCacheEntry[space]; -- Assume pCEMru = pCE
    IF found THEN
      BEGIN --assert--
```



```
      IF pCE.desc.dPinned THEN ERROR;
      pCEMru ← pCE.pCENext;
      pCE.pCENext ← pCEFree;
      pCEFree ← pCE
      END
    END;

  Get: PUBLIC ENTRY PROC [pDescResult: PDesc, space: Handle] =
    BEGIN
    found: BOOLEAN;
    pCE: PCE;
    [found, pCE] ← FindCacheEntry[space];
    IF found THEN pDescResult↑ ← pCE.desc ELSE pDescResult.state ← missing
    END;

  Insert: PUBLIC ENTRY PROC [pDescVictim: PDesc, pDesc: PDesc] =
    BEGIN
    pCE, pCECur, pCEPrev: PCE;
    IF FindCacheEntry[Handle[level: pDesc.level, page: pDesc.interval.page]].found
      THEN ERROR; -- Assertion --
    pDescVictim.dDirty ← FALSE;
    IF pCEFree ~= NIL THEN -- Free cache entry is available; use it
      {pCE ← pCEFree; pCEFree ← pCEFree.pCENext}
    ELSE
      IF PilotSwitches.switches.u = down THEN
        -- if utilityPilot, don't mess with other entries.
        pCE ← LOOPHOLE[Frame.Alloc[RuntimeInternal.MakeFsi[SIZE[CacheEntry]]]]
      ELSE
        BEGIN
        -- Replace lru element, which can't be same as mru since free chain is empty
        pCE ← NIL;
        FOR pCECur ← pCEMru, pCECur.pCENext WHILE pCECur.pCENext ~= NIL DO
          IF ~pCECur.pCENext.desc.dPinned THEN
            BEGIN pCE ← pCECur.pCENext; pCEPrev ← pCECur END
          ENDLOOP;
        IF pCE = NIL THEN ERROR; -- no unpinned entries => death
        pCEPrev.pCENext ← pCE.pCENext;
        IF pCE.desc.dDirty THEN pDescVictim↑ ← pCE.desc;
        END;
    pCE.pCENext ← pCEMru;
    pCEMru ← pCE;
    pCE.desc ← pDesc↑
    END;

  Update: PUBLIC ENTRY PROC [pDesc: PDesc] RETURNS [found: BOOLEAN] =
    BEGIN
    pCE: PCE;
    [found, pCE] ← FindCacheEntry[
      Handle[level: pDesc.level, page: pDesc.interval.page]];
    IF found THEN {pCE.desc ← pDesc↑; pCE.desc.dDirty ← TRUE}
    END;

  BEGIN
  pCE: PCE;
  pCE ← pCEFree ← @rgCE[FIRST[ICE]];
  THROUGH [FIRST[ICE]..LAST[ICE]) DO
    pCE.pCENext ← pCE + SIZE[CacheEntry]; pCE ← pCE.pCENext ENDLOOP;
  rgCE[LAST[ICE]].pCENext ← pCEMru ← NIL
  END
```

END.

June 1, 1978 3:00 PM	McJones	Create file
June 8, 1978 10:32 AM	McJones	Update didn't mark cache entry dirty
June 27, 1978 11:19 AM	McJones	Add consistency check for no replaceable cache entrie
**s		
September 9, 1978 2:55 PM	McJones	Suppress clearing of dDirty in Insert
August 29, 1979 4:51 PM	Ladner	Install instrumentation
March 16, 1980 6:39 PM	Forrest	Try expand cache hack again
May 7, 1980 1:30 PM Forrest	FrameOps = >Frame	
July 11, 1980 4:45 PM	Gobbel	Increase size from 32 to 42 (?)
August 29, 1980 5:10 PM	Forrest	Remove instrumentation
September 24, 1980 2:47 PM	McJones	Increase size from 42 to 60

```
-- MStoreImpl.mesa (last edited by Forrest on October 3, 1980 10:12 PM)

-- The monitor lock serializes calls on Allocate and Deallocate, but in addition interrupts are disabl
**ed whenever allocationMap and the hardware page map are inconsistent. The reason for this is
**that RecoverMStore must be called when interrupts are disabled and hence can't wait on a mon
**itor lock.

-- Note: No frame heap ALLOC's may be executed within any ENTRY procedures in this monitor, s
**o that they may be called from the allocation trap handler. If this rule were not followed, we cou
**ld get an allocation trap when we held the monitor lock, thus making Deallocate inaccessible to
**the allocation trap handler. Please see comments in MStore.mesa.

-- Things to consider:
--  1) Divide allocationMap into "segments"

DIRECTORY
  Environment USING [bitsPerWord],
  Frame USING [GetReturnLink, SetReturnLink],
  Inline USING [BITAND, BITNOT, BITOR, BITSHIFT, COPY],
  MStore,
  PageMap,
  PilotSwitches USING [switches --.t--],
  PrincOps USING [Port],
  Process USING [DisableAborts, InitializeMonitor],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  ProcessorFace USING [dedicatedRealMemory],
  RuntimeInternal USING [WorryCallDebugger],
  SpecialSpace,
  StoragePrograms USING [countVM],
  SwapperPrograms,
  VM USING [Interval, PageCount, PageNumber, PageOffset];

MStoreImpl: MONITOR LOCKS mStoreLock
  IMPORTS
    Frame, Inline, PageMap, PilotSwitches, Process, ProcessInternal,
    ProcessorFace, RuntimeInternal, StoragePrograms
  EXPORTS MStore, SpecialSpace, StoragePrograms
  SHARES MStore, PageMap =
  BEGIN OPEN Environment, MStore, PageMap, VM;

  --
  -- SpecialSpace

  realMemorySize: PUBLIC PageCount; -- see definition in SpecialSpace

  --
  -- StoragePrograms

  InitializeMStore: PUBLIC PROC =
    BEGIN
    Process.InitializeMonitor[@mStoreLock];
    -- must precede call to Initialize below
    Initialize[];
    END;

  --
  -- MStore
```



```
  dandelionMemSize: PageCount =
    768 --two boards--
      --Dandelion:-- - 64 --map-- - 256 --display + germ--
      --Dolphin:-- + 202 --display-- + 16 --germ--
  ;

  -- Monitor data:
  mStoreLock: PUBLIC MONITORLOCK; -- (PRIVATE in interface)
  -- Bit (rp MOD bitsPerWord) of word rp/bitsPerWord of allocationMap is 1 iff
  -- real page rp is allocated.
  -- Initially all real pages are allocated.
  allocationMapSpace: ARRAY [0..(maxRealPages + bitsPerWord - 1)/bitsPerWord) OF
    WORD;
  allocationMap: POINTER TO ARRAY [0..LENGTH[allocationMapSpace]) OF WORD =
    @allocationMapSpace;
  wMin, wMax: CARDINAL; -- bounds on words of allocationMap containing free pages
  countFree: PageCount ← 0;
  countPromised: PageCount ← 0;
  countThreshold: PageCount ← 0;
  countHeldBack: PageCount ← 0; -- for simulation of Dandelion memory size.
  allocation, deallocation: CONDITION; -- (aborts disabled)

  Initialize: ENTRY --so Initialize* can be called--PROC =
    BEGIN
    page: PageNumber;
    realPageMin: RealPageNumber ← LAST[RealPageNumber];
    realPageMax: RealPageNumber ← FIRST[RealPageNumber];
    allocationMap[0] ← 177777B;
    Inline.COPY[
      from: @allocationMap[0], to: @allocationMap[1],
      nwords: LENGTH[allocationMap↑] - 1];
    Process.DisableAborts[@allocation];
    Process.DisableAborts[@deallocation];
    realMemorySize ← ProcessorFace.dedicatedRealMemory;
    FOR page IN [FIRST[PageNumber]..FIRST[PageNumber] + StoragePrograms.countVM)
      DO
      value: Value = SetF[page + FIRST[PageNumber], valueClean];
      -- SetF of vacant is no-op
      IF value.flags ~= flagsVacant THEN
        BEGIN
        realMemorySize ← realMemorySize - 1;
        realPageMin ← MIN[realPageMin, value.realPage];
        realPageMax ← MAX[realPageMax, value.realPage];
        END
      ENDLOOP;
    IF PilotSwitches.switches.t = down THEN {
      countHeldBack ← realMemorySize - dandelionMemSize;
      realMemorySize ← dandelionMemSize;
      }
    ELSE countHeldBack ← 0;
    wMin ← realPageMin/bitsPerWord;
    wMax ← realPageMax/bitsPerWord;
    [] ← InitializeAllocateIfFree[]; -- allocate frame; initialize PORT.
    InitializeDeallocate[]; -- allocate frame; initialize PORT.

    END;

  Allocate: PUBLIC ENTRY PROC [interval: Interval] =
```

```
  BEGIN
  countBatch: PageCount;
  WHILE interval.count > 0 DO
    WHILE countFree <= countHeldBack DO WAIT deallocation ENDLOOP;
    countBatch ← MIN[interval.count, countFree - countHeldBack];
    [] ← allocateIfFreeInternal[interval: [interval.page, countBatch]];
    interval.page ← interval.page + countBatch;
    interval.count ← interval.count - countBatch;
    ENDLOOP;
  END;

allocateIfFreeInternal: PUBLIC --INTERNAL--AllocateIfFreeInternal ←
  -- (PRIVATE in interface) ("←" due to compiler glitch)
  -- Allocates and map real memory to as much of the specified interval as possible
  -- Guaranteed not to do an ALLOC from the frame heap.
  [LOOPHOLE[@AwaitAllocateIfFreeRequest]];
-- an indirect control link to the PORT.

AwaitAllocateIfFreeRequest: --RESPONDING--PORT [countAllocated: PageCount]
  RETURNS [interval: Interval];
-- args/results match allocateIfFreeInternal (but swapped).

InitializeAllocateIfFree: INTERNAL PROC RETURNS [countAllocated: PageCount]
  --to match PORT args-- =
  BEGIN
  interval: Interval;
  w, b: CARDINAL; -- allocationMap rover: word and bit number
  word: WORD; -- temporary, used to hold allocationMap[w]
  LOOPHOLE[AwaitAllocateIfFreeRequest, PrincOps.Port].dest ←
    Frame.GetReturnLink[];
  -- set my PORT call to return to my caller on call below.
  w ← wMin;
  b ← 0; -- ok to check of a couple unnecessary pages at first
  DO
    --FOREVER--
    -- Return result; Await new request; Process it;
    interval ← AwaitAllocateIfFreeRequest[countAllocated];
    Frame.SetReturnLink[
      LOOPHOLE[AwaitAllocateIfFreeRequest, PrincOps.Port].dest];
    -- for debugger.
    interval.count ← countAllocated ← MIN[
      interval.count, countFree - countHeldBack];
    WHILE interval.count > 0 DO
      -- allocate page at interval.page
      word ← allocationMap[w];
      DO
        -- Advance to next possible bit position
        IF (b ← (b + 1) MOD bitsPerWord) = 0 THEN
          DO
            w ← IF w = wMax THEN wMin ELSE w + 1;
            IF (word ← allocationMap[w]) ~= 177777B THEN EXIT;
            ENDLOOP;
        -- See if it is free
        IF Inline.BITAND[word, Inline.BITSHIFT[1, b]] = 0 THEN EXIT;
        ENDLOOP;
      ProcessInternal.DisableInterrupts[];
      -- maintain consistency for RecoverMStore
      allocationMap[w] ← Inline.BITOR[word, Inline.BITSHIFT[1, b]];
      -- Verify that page is not already mapped:
```



```
      IF GetF[interval.page].valueOld.flags ~= flagsVacant THEN
        RuntimeInternal.WorryCallDebugger[
          "MStore.Allocate given already-mapped vm"L];
      Assoc[interval.page, Value[FALSE, flagsClean, w*bitsPerWord + b]];
      countFree ← countFree - 1;
      ProcessInternal.EnableInterrupts[];
      interval.page ← interval.page + 1;
      interval.count ← interval.count - 1;
      ENDLOOP;
    IF countFree + countPromised < countThreshold + countHeldBack THEN
      BROADCAST allocation;
    ENDLOOP;
  END;

AwaitBelowThreshold: PUBLIC ENTRY PROC RETURNS [newCycle: BOOLEAN] =
  BEGIN
  IF countFree + countPromised < countThreshold + countHeldBack THEN
    RETURN[FALSE];
  UNTIL countFree + countPromised < countThreshold + countHeldBack DO
    WAIT allocation ENDLOOP;
  RETURN[TRUE];
  END;

deallocateInternal: PUBLIC DeallocateInternal ←
  -- (PRIVATE in interface) ("←" due to compiler glitch)
  [LOOPHOLE[@AwaitDeallocateRequest]]; -- an indirect control link to the PORT.

AwaitDeallocateRequest: --RESPONDING--PORT
  RETURNS [interval: Interval, promised: BOOLEAN];
-- args/results match deallocateInternal (but swapped).

InitializeDeallocate: INTERNAL PROC =
  BEGIN
  interval: Interval;
  promised: BOOLEAN;
  count: PageCount;
  LOOPHOLE[AwaitDeallocateRequest, PrincOps.Port].dest ← Frame.GetReturnLink[];
  -- set my PORT call to return to my caller on call below.
  DO
    --FOREVER--
    -- Await new request; Process it;
    [interval, promised] ← AwaitDeallocateRequest[];
    Frame.SetReturnLink[LOOPHOLE[AwaitDeallocateRequest, PrincOps.Port].dest];
    -- for debugger
    count ← 0;
    ProcessInternal.DisableInterrupts[];
    -- maintain consistency for RecoverMStore
    FOR offset: PageOffset IN [0..interval.count) DO
      value: Value = SetF[interval.page + offset, valueVacant];
      IF value.flags ~= flagsVacant THEN
        BEGIN
        w: CARDINAL = value.realPage/bitsPerWord;
        bit: WORD = Inline.BITSHIFT[1, value.realPage MOD bitsPerWord];
        allocationMap[w] ← Inline.BITAND[allocationMap[w], Inline.BITNOT[bit]];
        count ← count + 1;
        END;
      ENDLOOP;
    countFree ← countFree + count;
```

```
    ProcessInternal.EnableInterrupts[];
    IF promised THEN countPromised ← countPromised - count
    ELSE BROADCAST deallocation;
    ENDLOOP;
  END;

Promise: PUBLIC ENTRY PROC [count: PageCount] = {
  countPromised ← countPromised + count};

--StoragePrograms.--

RecoverMStore: PUBLIC PROC =
  BEGIN
  page: PageNumber ← 0;
  FOR realPage: RealPageNumber IN RealPageNumber DO
    w: CARDINAL = realPage/bitsPerWord;
    bit: WORD = Inline.BITSHIFT[1, realPage MOD bitsPerWord];
    IF Inline.BITAND[allocationMap[w], bit] = 0 THEN
      DO
        -- find next vacant virtual page
        IF GetF[page ← page + 1].valueOld.flags = flagsVacant THEN {
          Assoc[page, Value[FALSE, flagsClean, realPage]]; EXIT}
        ENDLOOP;
    ENDLOOP;
  END;

Relocate: PUBLIC ENTRY PROC [
  interval: Interval, pageDest: PageNumber, flagsKeep, flagsAdd: Flags]
  RETURNS [flags: Flags, anyVacant: BOOLEAN] =
  -- See note in MStore
  BEGIN
  ValueAnd: PROC [mv1, mv2: Value] RETURNS [Value] = LOOPHOLE[Inline.BITAND];
  ValueOr: PROC [mv1, mv2: Value] RETURNS [Value] = LOOPHOLE[Inline.BITOR];
  valueKeep: Value = [FALSE, flagsKeep, 7777B];
  -- mask to extract flags and realPage
  valueAdd: Value = [FALSE, flagsAdd, 0]; -- bit mask to OR in new flags
  value1: Value;
  value2: Value =
    IF flagsKeep = flagsNone AND interval.page = pageDest THEN valueAdd
    ELSE valueVacant;
  valueMax: Value ← [logSingleError:, flags: flagsNone, realPage:];
  anyVacant ← FALSE;
  FOR offset: PageOffset IN [0..interval.count) DO
    ProcessInternal.DisableInterrupts[];
    -- maintain consistency for RecoverMStore
    value1 ← SetF[interval.page + offset, value2];
    IF value1.flags = flagsVacant THEN anyVacant ← TRUE
    ELSE
      BEGIN
      IF value2 = valueVacant THEN
        Assoc[
          pageDest + offset, ValueOr[ValueAnd[value1, valueKeep], valueAdd]];
      valueMax ← ValueOr[valueMax, value1];
      END;
    ProcessInternal.EnableInterrupts[];
    ENDLOOP;
  flags ← valueMax.flags
  END;
```



```
  SetThreshold: PUBLIC ENTRY PROC [count: PageCount] = {
    countThreshold ← count; BROADCAST allocation};

  DonateDedicatedRealMemory: PUBLIC ENTRY PROC [
    page: PageNumber, size: PageCount] =
    BEGIN
    minDonate: RealPageNumber ← LAST[RealPageNumber];
    maxDonate: RealPageNumber ← FIRST[RealPageNumber];
    FOR offset: PageOffset IN [0..size) DO
      value: Value = GetF[page + offset];
      IF value.flags = flagsVacant THEN LOOP;
      minDonate ← MIN[minDonate, value.realPage];
      maxDonate ← MAX[maxDonate, value.realPage];
      ENDLOOP;
    deallocateInternal[interval: [page: page, count: size], promised: FALSE];
    wMin ← MIN[minDonate/bitsPerWord, wMin];
    wMax ← MAX[maxDonate/bitsPerWord, wMax];
    END;

  END.
```

(For earlier entries see Pilot 4.0 archive version.)

April 16, 1980 9:35 AM Knutsen Make Deallocate, GetState coroutines; add InitializeMS
***tore[]; recover = >RecoverMStore; add "t" key switch*
April 28, 1980 9:35 AM Forrest FrameOps = >Frame, ControlDefs = >PrincOps
May 30, 1980 9:21 AM Knutsen Make "t" key switch work right; also, amount of useabl
***e real mem settable dynamically*
August 12, 1980 11:04 AM McJones Add ProcessorFace.dedicatedRealMemory to realMem
***orySize; delete INLINE from Initialize*
September 23, 1980 11:03 AM McJones Add check that virtual page to be mapped is not alread
***y mapped*
October 3, 1980 9:04 PM Forrest Add donateDedicatedMemory. Change some ValueAN
***D[] = valueVacant to .flags = flagsVacant. Gunned GetState (the devil made me do it...)*

```
-- PageFaultImpl.mesa (last edited by: McJones on: September 5, 1980  11:36 AM)

DIRECTORY
  Frame USING [GetReturnLink, MyLocalFrame],
  PageFault USING [],
  PrincOps USING [BytePC, Frame, GlobalFrameHandle, NullFrame, StateVector],
  ProcessInternal USING [DisableInterrupts],
  ProcessOperations USING [EnableAndRequeue, IndexToHandle, ReadPSB, Requeue],
  PSB USING [Condition, Empty, Handle, nullHandle, PDA, Queue],
  RuntimeInternal USING [WorryCallDebugger],
  SDDefs USING [SD, sPageFault],
  SwapperPrograms USING [],
  Trap USING [ReadOTP],
  VM USING [Interval, PageNumber];

PageFaultImpl: MONITOR
  IMPORTS Frame, ProcessInternal, ProcessOperations, RuntimeInternal, Trap
  EXPORTS PageFault, SwapperPrograms =
  BEGIN

  queueFaulted: PSB.Queue ← [tail: PSB.Empty];
  queueNoticed: PSB.Queue ← [tail: PSB.Empty];
  faultOccurred: CONDITION;


  PageFaultTrap: INTERNAL PROCEDURE = -- called from outside monitor!
    -- Simulate fault notification which will eventually be done by processor
    -- (Given the intermediate PrincOps traps format, the stashing of the Trap parameter
    -- in the frame.local.unused is unnecessary but may remind us to do the right thing later.)
    BEGIN
    page: VM.PageNumber;
    state: RECORD [dontcare: UNSPECIFIED, v: PrincOps.StateVector];
    -- capture state
    state.v ← STATE;
    state.v.dest ← Frame.GetReturnLink[];
    state.v.source ← PrincOps.NullFrame;
    page ← LOOPHOLE[Trap.ReadOTP[]]; -- trap parameter
    LOOPHOLE[Frame.MyLocalFrame[], POINTER TO local PrincOps.Frame].unused ← page;
    -- put it where Await can find it (it was already there, though)
    IF page = LAST[VM.PageNumber] THEN
      DO RuntimeInternal.WorryCallDebugger["Map out of bounds"] ENDLOOP;
    ProcessInternal.DisableInterrupts[]; -- protect log and requeue
    Log[page, state.v.dest]; -- remove this someday...
    -- Simulate naked notify (recall interrupts are disabled):
    BEGIN OPEN cond: LOOPHOLE[faultOccurred, PSB.Condition];
    IF cond.tail = PSB.Empty THEN cond.wakeup ← TRUE ELSE NOTIFY faultOccurred;
    END;
    -- Add this process to the fault queue and reschedule:
    ProcessOperations.EnableAndRequeue[
      @PSB.PDA.ready, @queueFaulted, ProcessOperations.ReadPSB[]];
    RETURN WITH state.v
    END;

  Await: PUBLIC ENTRY PROCEDURE RETURNS [VM.PageNumber] =
    BEGIN
    p: PSB.Handle;
    DO
      IF queueFaulted ~= [tail: PSB.Empty] THEN
        BEGIN
```



```
        p ← ProcessOperations.IndexToHandle[
          PSB.PDA[ProcessOperations.IndexToHandle[queueFaulted.tail]].link.next];
        ProcessOperations.Requeue[@queueFaulted, @queueNoticed, p];
        WITH PSB.PDA[p].state SELECT local FROM local => RETURN[unused]; ENDCASE;
        END;
      WAIT faultOccurred;
      ENDLOOP
    END;

  Restart: PUBLIC ENTRY PROCEDURE [interval: VM.Interval] =
    BEGIN
    pLast: PSB.Handle = ProcessOperations.IndexToHandle[queueNoticed.tail];
    p, pLink: PSB.Handle;
    IF queueNoticed ~= [tail: PSB.Empty] THEN
      FOR p ← ProcessOperations.IndexToHandle[
        PSB.PDA[ProcessOperations.IndexToHandle[queueNoticed.tail]].link.next],
        pLink DO
        pLink ← ProcessOperations.IndexToHandle[PSB.PDA[p].link.next];
        WITH PSB.PDA[p].state SELECT local FROM
          local =>
            IF unused IN [interval.page..interval.page + interval.count) THEN
              ProcessOperations.Requeue[@queueNoticed, @PSB.PDA.ready, p];
          ENDCASE;
        IF p = pLast THEN EXIT;
        ENDLOOP;
    END;

  -- Event log (for debugging)

  Event: TYPE = RECORD [
    page: VM.PageNumber,
    process: PSB.Handle,
    local: POINTER TO local PrincOps.Frame,
    global: PrincOps.GlobalFrameHandle,
    pc: PrincOps.BytePC];
  lEvent: TYPE = [0..20);
  rgEvent: ARRAY lEvent OF Event ← ALL[[0, PSB.nullHandle, NIL, NIL, [0]]];
  pEvent: POINTER TO Event ← @rgEvent[FIRST[lEvent]];
  Log: INTERNAL PROC [
    page: VM.PageNumber, trapee: POINTER TO local PrincOps.Frame] = INLINE
    BEGIN
    pEvent↑ ←
      [page, ProcessOperations.ReadPSB[], trapee, trapee.accesslink, trapee.pc];
    pEvent ←
      IF pEvent = @rgEvent[LAST[lEvent]] THEN @rgEvent[FIRST[lEvent]]
      ELSE pEvent + SIZE[Event];
    END;

  -- Initialization

  SDDefs.SD[SDDefs.sPageFault] ← PageFaultTrap;

  END.
```

```
June 12, 1978  9:46 AM      McJones  Create file
June 20, 1978  10:53 AM     McJones  Replace "mark bit" with two queues; add correct simul
**ation of naked notify
June 23, 1978  1:30 PM      McJones  Remove RestartQuantifier
```

July 25, 1978 5:53 PM McJones Trap didn't set state.dest (or .source); Restart traversed
***queue incorrectly*
September 19, 1978 12:03 PM McJones Add event log
March 28, 1979 10:10 AM McJones Initialization of rgEvent clobbered following words
April 28, 1980 10:59 AM Forrest FrameOps = >Frame, ControlDefs = >PrincOps, TrapOp
***s = >Trap. PSBDefs = >PSB, ProcessOps = >ProcessOperations; rearrange logging code a bit usi*
***ng an INLINE; use ALL construct to initialize log table*
June 10, 1980 8:27 AM Forrest Detect -1 parameter to PageFaultTrap
August 26, 1980 7:07 PM McJones New PSB, ProcessOperation

```
-- Swapper>ResidentMemoryImpl.mesa (last edited by Forrest on April 28, 1980 9:57 AM)
-- Note: No frame heap ALLOC's may be executed within any ENTRY procedures in this monitor, s
**o that they may be called from the allocation trap handler. If this rule were not followed, we cou
**ld get an allocation trap when we held the monitor lock, thus making AllocateMDS inaccessible
**to the allocation trap handler. Please see comments in ResidentMemory.mesa.

DIRECTORY
  CachedRegionInternal: FROM "CachedRegionInternal" USING [
    AllocateMStoreRuthlessly],
  Environment: FROM "Environment" USING [bitsPerWord, PageCount, PageNumber],
  Frame: FROM "Frame" USING [GetReturnLink, SetReturnLink],
  Inline: FROM "Inline" USING [LowHalf],
  MStore: FROM "MStore" USING [Deallocate],
  PrincOps: FROM "PrincOps" USING [Port],
  Process: FROM "Process" USING [InitializeMonitor],
  ResidentMemory: FROM "ResidentMemory",
  RuntimeInternal: FROM "RuntimeInternal" USING [WorryCallDebugger],
  SimpleSpace: FROM "SimpleSpace" USING [AllocateVM],
  Space: FROM "Space" USING [defaultWindow],
  StoragePrograms: FROM "StoragePrograms" USING [
    DescribeSpace, LongPointerFromPage, lpMDS, outlaw, PageFromLongPointer,
    pageMDS],
  Utilities: FROM "Utilities" USING [Bit, BitIndex, BitGet, BitPut],
  VM: FROM "VM" USING [Interval];

ResidentMemoryImpl: MONITOR LOCKS residentMemoryLock
  IMPORTS
    CachedRegionInternal, Frame, Inline, MStore, Process, RuntimeInternal,
    SimpleSpace, StoragePrograms, Utilities
  EXPORTS ResidentMemory, StoragePrograms
  SHARES ResidentMemory =
  BEGIN OPEN Environment, ResidentMemory;
  -- Parameters:
  countMax: PageCount = 40; -- amount of VM allocated for each location.
  -- Variables:
  residentMemoryLock: PUBLIC MONITORLOCK; -- (PRIVATE in interface)
  locationData: TYPE = RECORD [
    page: PageNumber, -- starting page of VM area.
    allocationMap: ARRAY [0..(countMax + bitsPerWord - 1)/bitsPerWord) OF
      CARDINAL]; -- PACKED ARRAY OF BOOLEAN in Mesa 6.
  first64K: locationData;
  mds: locationData;
  hyperspace: locationData;
  locations: ARRAY Location OF POINTER TO locationData ←
    [@first64K,
      IF StoragePrograms.pageMDS = PageNumber[0] THEN @first64K ELSE @mds,
      @hyperspace];
  free: Utilities.Bit = Utilities.Bit[0];
  -- for allocationMap. A free page is unmapped.
  busy: Utilities.Bit = Utilities.Bit[1];
  j: Utilities.BitIndex;
  --StoragePrograms.--
  InitializeResidentMemoryA: PUBLIC ENTRY PROCEDURE =
    BEGIN
    Process.InitializeMonitor[@residentMemoryLock];
    InitializeInternal[];
    END;
```



```
  InitializeInternal: ENTRY --so InitializeAllocate can be called--PROCEDURE =
    INLINE
    BEGIN
    location: Location;
    FOR location IN Location DO
      OPEN loc: locations[location];
      FOR j IN [0..countMax) DO
        Utilities.BitPut[free, @locations[location].allocationMap, j]; ENDLOOP;
      IF ~(location = mds AND locations[mds] = locations[first64K])
        --(if MDS = first64K, we don't allocate twice)-- THEN
        loc.page ← SimpleSpace.AllocateVM[countMax, location];
      ENDLOOP;
    [] ← InitializeAllocate[]; -- allocate frame; initialize PORT.
    [] ← InitializeAllocateMDS[]; -- allocate frame; initialize PORT.

    END;
  --StoragePrograms.--

  InitializeResidentMemoryB: PUBLIC ENTRY PROCEDURE =
    -- tells Swapper about resident memory spaces.
    BEGIN
    location: Location;
    FOR location IN Location DO
      IF ~(location = mds AND locations[mds] = locations[first64K]) THEN
        StoragePrograms.DescribeSpace[
          StoragePrograms.outlaw, locations[location].page, countMax,
          Space.defaultWindow];
      ENDLOOP;
    END;

  allocateInternal: PUBLIC --INTERNAL--AllocateInternal ←
    -- (PRIVATE in interface) ("←" due to compiler glitch)
    -- Guaranteed not to do an ALLOC from the frame heap.
    [LOOPHOLE[@AwaitAllocateRequest]]; -- an indirect control link to the PORT.
  AwaitAllocateRequest: --RESPONDING--PORT [lp: LONG POINTER TO UNSPECIFIED]
    RETURNS [where: Location, pages: PageCount];
  -- args/results match allocateInternal (but swapped).
  InitializeAllocate: INTERNAL PROCEDURE RETURNS [lp: LONG POINTER TO UNSPECIFIED]
    --to match PORT args-- =
    BEGIN
    where: Location;
    pages: PageCount;
    LOOPHOLE[AwaitAllocateRequest, PrincOps.Port].dest ← Frame.GetReturnLink[];
    -- set my PORT call to return to my caller on call below.
    DO
      --FOREVER--
      -- Return result; Await new request; Process it;
      [where, pages] ← AwaitAllocateRequest[lp];
      Frame.SetReturnLink[LOOPHOLE[AwaitAllocateRequest, PrincOps.Port].dest];
      -- for debugger
      --scope of loc: --
      BEGIN OPEN loc: locations[where];
      start, pageInRun: CARDINAL;
      FOR start ← 0, pageInRun + 1 WHILE start + pages < countMax DO
        -- look for good starting page..
        FOR pageInRun IN [start..start + pages) DO
          -- look for pages contiguous pages..
          IF Utilities.BitGet[@loc.allocationMap, pageInRun] = busy THEN EXIT;
          -- pageInRun must survive loop exit.
```

```
        REPEAT
          FINISHED => -- found pages contiguous free pages.
            BEGIN
            CachedRegionInternal.AllocateMStoreRuthlessly[
              VM.Interval[loc.page + start, pages]];
            FOR pageInRun IN [start..start + pages) DO
              Utilities.BitPut[busy, @loc.allocationMap, pageInRun]; ENDLOOP;
            lp ← StoragePrograms.LongPointerFromPage[loc.page + start];
            GO TO Done;
            END;
        ENDLOOP;
      REPEAT -- (not enough contiguous pages beginning at start)

        FINISHED =>
          DO
            RuntimeInternal.WorryCallDebugger["Out of VM for resident memory"L]
            ENDLOOP; -- (so just increase countMax.)

      ENDLOOP;
    EXITS Done => NULL;
    END --scope of loc--
    ;
    ENDLOOP;
  END;

Free: PUBLIC ENTRY PROCEDURE [
  where: Location, pages: CARDINAL, lp: LONG POINTER] =
  BEGIN OPEN loc: locations[where];
  start: CARDINAL = StoragePrograms.PageFromLongPointer[lp] - loc.page;
  pageInRun: CARDINAL;
  MStore.Deallocate[
    interval: VM.Interval[loc.page + start, pages], promised: FALSE];
  FOR pageInRun IN [start..start + pages) DO
    Utilities.BitPut[free, @loc.allocationMap, pageInRun]; ENDLOOP;
  END;

allocateMDSInternal: PUBLIC --INTERNAL--AllocateMDSInternal ←
  -- (PRIVATE in interface) ("←" due to compiler glitch)
  -- Guaranteed not to do an ALLOC from the frame heap.
  [LOOPHOLE[@AwaitAllocateMDSRequest]]; -- an indirect control link to the PORT.
AwaitAllocateMDSRequest: --RESPONDING--PORT [p: POINTER TO UNSPECIFIED]
  RETURNS [pages: PageCount];
-- args/results match allocateMDSInternal (but swapped).
InitializeAllocateMDS: INTERNAL PROCEDURE RETURNS [p: POINTER TO UNSPECIFIED]
  --to match PORT args-- =
  BEGIN
  pages: PageCount;
  LOOPHOLE[AwaitAllocateMDSRequest, PrincOps.Port].dest ← Frame.GetReturnLink[];
  -- set my PORT call to return to my caller on call below.
  DO
    --FOREVER--
    -- Return result; Await new request; Process it;
    pages ← AwaitAllocateMDSRequest[p];
    Frame.SetReturnLink[LOOPHOLE[AwaitAllocateMDSRequest, PrincOps.Port].dest];
    -- for debugger
    p ← Inline.LowHalf[allocateInternal[mds, pages].lp - StoragePrograms.lpMDS];
    ENDLOOP;
  END;
```



```
END.
```

LOG

Time: April 16, 1980 8:52 AM By: Knutsen Action: Created file from PilotControl of Mar
***ch 11, 1980. Added InitializeResidentMemoryA/B. Names changed to AllocateMDS, FreeMDS.*
***Implement Free. Make Allocate alloc in first64K and hyperspace, as well as MDS. Made Allocat*
***e, AlocateMDS coroutines.*
Time: April 28, 1980 9:56 AM By: Forrest Action: ControlDefs = >PrincOps, FrameOps
*** = > Frame*

```
-- Swapper>SimpleSpaceImpl.mesa (last edited by Knutsen on September 2, 1980 2:52 PM)
-- This module is structurally at a level above the Swapper, and below the FileMgr. It is of type "C
**ached Descriptor", but the items exported to StoragePrograms must be Initially Resident.

DIRECTORY
  CachedRegion USING [
    Apply, BackFileType, Desc, forceOut, Insert, kill, Operation, Outcome, pin,
    unmap],
  CachedSpace USING [
    Desc, Get, Handle, handleNull, Insert, Level, SizeSwapUnit, Update],
  Environment USING [maxPagesInMDS],
  File USING [nullID, write],
  Inline USING [BITAND],
  MStore USING [Deallocate, Relocate],
  PageMap USING [flagsNone, flagsWriteProtected],
  PilotSwitches USING [switches --.u--],
  SimpleSpace,
  Space USING [defaultWindow, Handle, PageCount, PageNumber, WindowOrigin],
  StoragePrograms USING [
    countHyperspace, createSimpleSpace, empty, EnumerateStartList, pageHyperspace,
    pageMDS, SpaceOptions, StartListProc, tableBase],
  VM USING [Interval];

SimpleSpaceImpl: PROGRAM
  IMPORTS
    CachedRegion, CachedSpace, Inline, MStore, PilotSwitches, StoragePrograms
  EXPORTS SimpleSpace, StoragePrograms
  SHARES File --USING [Capability]--; =
  BEGIN OPEN SimpleSpace;
  -- + + Space. + + Handle: PUBLIC TYPE = CachedSpace.Handle; + + can't do this because of
**a name confict between Space.Map and SimpleSpace.Map, etc.
  -- Handle: TYPE = CachedSpace.Handle; + + can't do this because of a name confict between
**Space.Handle "warning: Handle is private but matches an export"
  SimpleSpaceHandle: TYPE = CachedSpace.Handle;
  locationData: TYPE = RECORD [total: VM.Interval, largestHole: VM.Interval];
  locations: ARRAY Location OF POINTER TO locationData;
  first64K, mds, hyperspace: locationData;
  initializationDisabled: BOOLEAN ← FALSE;
  Bug: PRIVATE ERROR [BugType] = CODE;
  BugType: TYPE = {
    badOutcome, initializationDisabled, memoryOverflow, notEndRegion,
    regionCacheOverflow, spaceCacheOverflow, spaceDMissing, notSimpleSpace};
  -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  -- Exported to StoragePrograms:
  -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  nullSpaceHandle: PUBLIC Space.Handle ← LOOPHOLE[CachedSpace.handleNull];
  -- for use before Space.nullHandle is initialized.
  InitializeSimpleSpace: PUBLIC PROCEDURE[] =
    BEGIN
    InitializeLocations: StoragePrograms.StartListProc --[index]-- =
      -- Initializes allocator data from empty spaces in StartPilot-allocated memory.
      BEGIN
      WITH e: StoragePrograms.tableBase[index] SELECT FROM
        space =>
          IF e.type.class = empty THEN
            DescribeSpace[
              StoragePrograms.empty, e.vmpage, e.pages, Space.defaultWindow];
        ENDCASE;
```



```
      END;
    --      Total VM Interval      Largest Hole
    first64K ← [[Space.PageNumber[0], Environment.maxPagesInMDS], [0, 0]];
    mds ←
      [[StoragePrograms.pageMDS, Environment.maxPagesInMDS],
        [StoragePrograms.pageMDS, 0]];
    hyperspace ←
      [[StoragePrograms.pageHyperspace, StoragePrograms.countHyperspace],
        [StoragePrograms.pageHyperspace, 0]];
    locations ←
      [first64K: @first64K,
        mds:
        IF StoragePrograms.pageMDS = first64K.total.page THEN @first64K ELSE @mds,
        hyperspace: @hyperspace];
    StoragePrograms.EnumerateStartList[InitializeLocations];
    END;

  AllocateVM: PUBLIC PROCEDURE [count: Space.PageCount, location: Location]
    RETURNS [page: Space.PageNumber] =
    BEGIN OPEN loc: locations[location];
    IF count > loc.largestHole.count THEN ERROR Bug[memoryOverflow];
    -- free space exhausted.
    page ← loc.largestHole.page;
    loc.largestHole.page ← loc.largestHole.page + count;
    loc.largestHole.count ← loc.largestHole.count - count;
    END;

  DescribeSpace: PUBLIC PROCEDURE [
    options: StoragePrograms.SpaceOptions, page: Space.PageNumber,
    count: Space.PageCount, window: Space.WindowOrigin] =
    BEGIN
    DescribeSpaceInternal[options, page, count, window, SimpleSpace.noSwapUnits];
    END;

  DescribeSpaceInternal: PROCEDURE [
    options: StoragePrograms.SpaceOptions, page: Space.PageNumber,
    count: Space.PageCount, window: Space.WindowOrigin,
    sizeSwapUnit: SimpleSpace.SizeSwapUnit] =
    BEGIN OPEN StoragePrograms;
    location: Location;
    regionVictim: CachedRegion.Desc;
    space, spaceVictim: CachedSpace.Desc;
    levelPrimarySpace: CachedSpace.Level =
      -- (VM is at level 0, MDS is at level 1.)
      IF page IN
      [StoragePrograms.pageMDS..StoragePrograms.pageMDS +
                                Environment.maxPagesInMDS) THEN 2 ELSE 1;
    writeProtected: BOOLEAN = Inline.BITAND[window.file.permissions, File.write] =
      0;
    IF initializationDisabled THEN ERROR Bug[initializationDisabled];
    IF count = 0 THEN RETURN;
    IF options.mStoreDeallocate THEN
      MStore.Deallocate[interval: [page, count], promised: FALSE];
    IF options.emptyInterval THEN --update allocator data--
      FOR location IN Location DO
        OPEN loc: locations[location];
        IF count > loc.largestHole.count AND page >= loc.total.page AND
          page + count <= loc.total.page + loc.total.count THEN
```

```
        loc.largestHole ← [page: page, count: count];
      ENDLOOP;
    IF options.createRegion AND
      (PilotSwitches.switches.u = up OR options.pinRegionD) THEN
      BEGIN
      levelRegion: CachedSpace.Level =
        levelPrimarySpace + (IF options.subspace THEN 1 ELSE 0);
      regionVictim ← CachedRegion.Insert[
        [interval: [page, count], level: levelRegion,
          levelMapped: levelPrimarySpace,
          hasSwapUnits: sizeSwapUnit ~= SimpleSpace.noSwapUnits, dTemperature:,
          -- don't care
          dPinned: options.pinRegionD, dDirty: TRUE,
          state:
          SELECT TRUE FROM
            options.pinned => inPinned,
            options.initiallyResident => inSwappableWarmest,
            options.mapped => outAlive,
            ENDCASE => unmapped, writeProtected: writeProtected,
          needsLogging: FALSE, beingFlushed: FALSE]];
      IF regionVictim.dDirty THEN ERROR Bug[regionCacheOverflow];
      IF options.initiallyResident AND writeProtected THEN
        -- The following depends on Relocate NOT temporarily vacating the pages
        [] ← MStore.Relocate[
          interval: [page, count], pageDest: page, flagsKeep: PageMap.flagsNone,
          flagsAdd: PageMap.flagsWriteProtected];
      -- write protect the pages now in memory.

      END;
    IF options.createSpace AND
      (PilotSwitches.switches.u = up OR options.pinRegionD) THEN
      BEGIN
      space ←
        [interval: [page, count], level: levelPrimarySpace,
          dPinned: options.pinSpaceD, dDirty: TRUE, pinned: options.pinned,
          state: IF options.mapped THEN mapped ELSE unmapped,
          writeProtected: writeProtected, -- (ok if state = unmapped.)
          hasSwapUnits: sizeSwapUnit ~= SimpleSpace.noSwapUnits,
          sizeSwapUnit: MAX[1, sizeSwapUnit],
          -- (avoid illegal swap unit size of 0)
          dataOrFile: file, pageRover: page,
          vp: long[window: window, countMapped: count, transaction: LOOPHOLE[0]]];
      -- transaction is don't care, and Transaction.nullHandle is not initialized yet.
      CachedSpace.Insert[@spaceVictim, @space];
      IF spaceVictim.dDirty THEN ERROR Bug[spaceCacheOverflow];
      END;
    END;

  HandleFromPage: PUBLIC PROCEDURE [page: Space.PageNumber]
    RETURNS [handle: Space.Handle] = {
    RETURN[
      LOOPHOLE[SimpleSpaceHandle[
        level:
        IF page IN
        [StoragePrograms.pageMDS..StoragePrograms.pageMDS +
                                  Environment.maxPagesInMDS) THEN 2 ELSE 1,
        page: page]]]};
```



```
  SuperFromPage: PUBLIC PROCEDURE [page: Space.PageNumber]
    RETURNS [handle: Space.Handle, superPage: Space.PageNumber] = {
    IF page IN
      [StoragePrograms.pageMDS..StoragePrograms.pageMDS +
                                  Environment.maxPagesInMDS) THEN
      RETURN[
        handle: LOOPHOLE[SimpleSpaceHandle[
        level: 1, page: StoragePrograms.pageMDS]],
          superPage: StoragePrograms.pageMDS]
    ELSE
      RETURN[
        handle: LOOPHOLE[SimpleSpaceHandle[
        level: 0, page: FIRST[Space.PageNumber]]],
          superPage: FIRST[Space.PageNumber]]};
  ----------------------------------------------------
  -- SimpleSpace implementation:
  ----------------------------------------------------

  ApplyToSpace: PROCEDURE [
    spaceD: POINTER TO CachedSpace.Desc, operation: CachedRegion.Operation] =
    -- Performs, in ascending order, operation (successfully) for each swap unit of the space (exac
**tly as required by CachedRegion.Apply).
    BEGIN
    page, pageNext: Space.PageNumber;
    outcome: CachedRegion.Outcome;
    FOR page ← spaceD.interval.page, pageNext WHILE page <
      spaceD.interval.page + spaceD.interval.count DO
      -- for all swapUnits..
      DO
        --REPEAT operation UNTIL outcome = ok or notePinned--
        [outcome, pageNext] ← CachedRegion.Apply[page, operation];
        WITH outcome SELECT FROM
          ok, notePinned => EXIT;
          retry => NULL; -- go around again

          ENDCASE => ERROR Bug[badOutcome];
        ENDLOOP;
      ENDLOOP;
    IF pageNext ~= spaceD.interval.page + spaceD.interval.count THEN
      ERROR Bug[notEndRegion];
    END;

  CopyIn: PUBLIC PROCEDURE [
    handle: Space.Handle, window: Space.WindowOrigin,
    countMapped: Space.PageCount] =
    BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
    spaceD, fromSpace: CachedSpace.Desc;
    CachedSpace.Get[@spaceD, hnd];
    IF spaceD.state = missing OR ~spaceD.dPinned THEN ERROR Bug[notSimpleSpace];
    fromSpace ← spaceD;
    -- sets fromSpace.interval, .mapped, and .writeProtected (to keep swapper happy).
    fromSpace.window ← window;
    fromSpace.countMapped ← MIN[countMapped, spaceD.countMapped];
    ApplyToSpace[
      @spaceD,
      [ifMissing: report, ifCheckedOut: wait, afterForking: wait,
        vp: copyIn[from: @fromSpace]]];
    END;
```

```
CopyOut: PUBLIC PROCEDURE [
  handle: Space.Handle, window: Space.WindowOrigin,
  countMapped: Space.PageCount] =
  BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
  spaceD, toSpace: CachedSpace.Desc;
  CachedSpace.Get[@spaceD, hnd];
  IF spaceD.state = missing OR ~spaceD.dPinned THEN ERROR Bug[notSimpleSpace];
  toSpace ← spaceD;
  -- sets fromSpace interval, .mapped, and .writeProtected (to keep swapper happy).
  toSpace.window ← window;
  toSpace.countMapped ← MIN[countMapped, spaceD.countMapped];
  ApplyToSpace[
    @spaceD,
    [ifMissing: report, ifCheckedOut: wait, afterForking: wait,
      vp: copyOut[to: @toSpace]]];
  END;

Create: PUBLIC PROCEDURE [
  count: Space.PageCount, location: Location,
  sizeSwapUnit: SimpleSpace.SizeSwapUnit] RETURNS [handle: Space.Handle] =
  BEGIN OPEN locations[location];
  hnd: SimpleSpaceHandle;
  IF count > largestHole.count THEN ERROR Bug[memoryOverflow];
  -- free space exhausted
  hnd.page ← largestHole.page;
  hnd.level ←
    IF hnd.page IN [mds.total.page..mds.total.page + mds.total.count) THEN 2
    ELSE 1;
  largestHole.page ← largestHole.page + count;
  largestHole.count ← largestHole.count - count;
  DescribeSpaceInternal[
    StoragePrograms.createSimpleSpace, hnd.page, count, Space.defaultWindow,
    sizeSwapUnit];
  RETURN[LOOPHOLE[hnd]];
  END;

DisableInitialization: PUBLIC PROCEDURE = {initializationDisabled ← TRUE};

ForceOut: PUBLIC PROCEDURE [handle: Space.Handle] =
  BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
  spaceD: CachedSpace.Desc;
  CachedSpace.Get[@spaceD, hnd];
  IF spaceD.state = missing OR ~spaceD.dPinned THEN ERROR Bug[notSimpleSpace];
  ApplyToSpace[@spaceD, CachedRegion.forceOut];
  END;

Kill: PUBLIC PROCEDURE [handle: Space.Handle] =
  BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
  spaceD: CachedSpace.Desc;
  CachedSpace.Get[@spaceD, hnd];
  IF spaceD.state = missing OR ~spaceD.dPinned THEN ERROR Bug[notSimpleSpace];
  ApplyToSpace[@spaceD, CachedRegion.kill];
  END;

Map: PUBLIC PROCEDURE [
  handle: Space.Handle, window: Space.WindowOrigin, andPin: BOOLEAN,
  countMapped: Space.PageCount] =
  -- If window.file.fID = nullID, then we allocate real memory and pin it (no backing file); in this ca
```



```
**se, andPin and countMapped are ignored.
  -- Else (backing file) assumes window.file is writable and file doesn't end before space.
  BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
  spaceD: CachedSpace.Desc;
  backFileType: CachedRegion.BackFileType;
  CachedSpace.Get[@spaceD, hnd];
  IF --spaceD.state = missing OR--spaceD.state ~= unmapped OR ~spaceD.dPinned THEN
    ERROR Bug[notSimpleSpace];
  -- we check for unmapped here because we will update the SpaceD before doing Apply, which
**would make debugging harder.
  spaceD.window ← window; -- either a real window or Space.defaultWindow
  IF window.file.fID = File.nullID THEN
    BEGIN --create space with no backing file--
    spaceD.pinned ← TRUE;
    spaceD.state ← mapped; -- this is a convenient lie.
    spaceD.writeProtected ← FALSE;
    -- spaceD.hasSwapUnits is set when the simpleSpace is created.
    spaceD.dataOrFile ← data;
    spaceD.countMapped ← spaceD.interval.count; -- this is a convenient lie.
    backFileType ← none;
    END
  ELSE
    BEGIN --create space with backing file--
    spaceD.pinned ← andPin;
    spaceD.state ← mapped;
    spaceD.writeProtected ←
      (Inline.BITAND[window.file.permissions, File.write] = 0);
    -- spaceD.hasSwapUnits is set when the simpleSpace is created.
    spaceD.dataOrFile ← file;
    spaceD.countMapped ←
      IF countMapped = defaultCount THEN spaceD.interval.count ELSE countMapped;
    backFileType ← file;
    END;
  IF ~CachedSpace.Update[@spaceD].found THEN ERROR Bug[spaceDMissing];
  ApplyToSpace[
    @spaceD,
    [ifMissing: report, ifCheckedOut: wait, afterForking:,
      vp: map[
      spaceD.level, backFileType, spaceD.writeProtected, --andNeedsLogging:--
      FALSE]]];
  IF backFileType = file AND andPin THEN
    ApplyToSpace[@spaceD, CachedRegion.pin];
  END;

Unmap: PUBLIC PROCEDURE [handle: Space.Handle] =
  BEGIN OPEN hnd: LOOPHOLE[handle, SimpleSpaceHandle];
  spaceD: CachedSpace.Desc;
  CachedSpace.Get[@spaceD, hnd];
  IF --spaceD.state = missing OR--spaceD.state ~= mapped OR ~spaceD.dPinned THEN
    ERROR Bug[notSimpleSpace];
  -- we check for unmapped here because the Swapper can't.
  ApplyToSpace[@spaceD, CachedRegion.unmap];
  spaceD.pinned ← FALSE;
  spaceD.state ← unmapped;
  IF ~CachedSpace.Update[@spaceD].found THEN ERROR Bug[spaceDMissing];
  END;

END.
```

LOG
(For earlier log entries, see Pilot 4.0 archive version.)
April 16, 1980 12:27 PM Knutsen Added InitializeSimpleSpace, AllocateVM; fix up for no
***nzero MDS; implement simpleSpaces with swap units.*
July 1, 1980 5:32 PM Knutsen New StartPilot/StartList; different SpaceOptions; move handling o
***f empty spaces to InitializeSimpleSpace; fix bug unmapping spaces w/ swapUnits; export nullS*
***paceHandle.*
July 14, 1980 5:22 PM Gobbel Made compatible with new CachedRegion.Desc, make
***Unmap do equivalent of ApplyToInterval so we get all swap units.*
July 16, 1980 2:43 PM Gobbel Made DescribeSpaceInternal look at U switch and pinn
***ed boolean before making new cache entries.*
July 19, 1980 1:03 PM McJones Added missing " = CODE" to definition of Error.
July 1, 1980 5:32 PM Knutsen Made compatible with new CachedRegion.Desc.
September 2, 1980 2:52 PM Knutsen Implemented Kill, CopyIn, CopyOut. Handle swap units
***in region. Page[] now an inline. Tried to convert to exported type Handle.. so sorry, Compiler w*
***on't allow!*

```
-- SwapBufferImpl.mesa (last edited by: McJones on: September 23, 1980 4:01 PM)

DIRECTORY
  SwapBuffer USING [],
  Environment USING [Base, bitsPerWord, first64K, Word],
  Inline USING [BITAND, BITNOT, BITOR, BITSHIFT],
  PageMap USING [flagsVacant, GetF],
  ResidentHeap USING [MakeNode],
  Runtime USING [CallDebugger],
  RuntimeInternal USING [WorryCallDebugger],
  SimpleSpace USING [AllocateVM],
  Space USING [defaultWindow],
  SpecialSpace USING [realMemorySize],
  StoragePrograms USING [DescribeSpace, outlaw],
  SwapperPrograms USING [],
  VM USING [Interval, PageCount, PageNumber, PageOffset],
  Zone USING [Status];

SwapBufferImpl: MONITOR
  IMPORTS
    Inline, PageMap, ResidentHeap, Runtime, RuntimeInternal, SimpleSpace,
    SpecialSpace, StoragePrograms
  EXPORTS SwapBuffer, SwapperPrograms
  SHARES PageMap =
  BEGIN

  Status: TYPE = {free, busy}; -- i.e. 0 is free

  pageBuffer: VM.PageNumber;
  countBuffer: VM.PageCount;
  pAllocationMap: LONG POINTER TO ARRAY [0..0) OF Environment.Word;
  deallocation: CONDITION;

  Error: PROCEDURE =
    BEGIN
    DO
      RuntimeInternal.WorryCallDebugger["SwapBufferImpl consistency check"L]
      ENDLOOP
    END;

  InitializeSwapBuffer: PUBLIC PROCEDURE =
    BEGIN
    allocMapSize: CARDINAL;
    node: Environment.Base RELATIVE POINTER;
    s: Zone.Status;
    w: CARDINAL;
    countBuffer ← SpecialSpace.realMemorySize;
    -- big enough to hold all of real memory.
    pageBuffer ← SimpleSpace.AllocateVM[countBuffer, hyperspace];
    StoragePrograms.DescribeSpace[
      StoragePrograms.outlaw, pageBuffer, countBuffer, Space.defaultWindow];
    -- tells Swapper about swap buffer space.
    allocMapSize ←
      (countBuffer + Environment.bitsPerWord - 1)/Environment.bitsPerWord;
    [node, s] ← ResidentHeap.MakeNode[allocMapSize];
    IF s ~= okay THEN Runtime.CallDebugger["Resident heap full"L];
    pAllocationMap ← @Environment.first64K[node];
    FOR w IN [0..allocMapSize) DO pAllocationMap[w] ← 0 ENDLOOP;
    -- SetBlock would be handy
```



```
    END;

  Allocate: PUBLIC ENTRY PROCEDURE [count: VM.PageCount]
    RETURNS [interval: VM.Interval] =
    BEGIN
    offsetPage: VM.PageOffset;
    offsetOffset: VM.PageOffset;
    DO
      -- until enough space is available
      FOR offsetPage ← 0, offsetPage + offsetOffset + 1 WHILE offsetPage + count
        <= countBuffer DO
        BEGIN
        FOR offsetOffset IN [0..count) DO
          IF GetStatus[offsetPage + offsetOffset] ~= free THEN GO TO HoleTooSmall
          ENDLOOP;
        FOR offsetOffset IN [0..count) DO
          SetStatus[offsetPage + offsetOffset, busy] ENDLOOP;
        RETURN[[pageBuffer + offsetPage, count]]
        EXITS HoleTooSmall => NULL
        END
        ENDLOOP;
      WAIT deallocation
      ENDLOOP
    END;

  Deallocate: PUBLIC ENTRY PROCEDURE [interval: VM.Interval] =
    BEGIN
    offset: VM.PageOffset;
    --assert--
    IF interval.page ~IN [pageBuffer..pageBuffer + countBuffer) OR
      interval.page + interval.count ~IN [pageBuffer..pageBuffer + countBuffer]
      THEN Error[];
    FOR offset IN [0..interval.count) DO
      --assert--
      IF PageMap.GetF[interval.page].valueOld.flags ~= PageMap.flagsVacant THEN
        Error[];
      SetStatus[interval.page - pageBuffer + offset, free];
      ENDLOOP;
    BROADCAST deallocation;
    END;

  GetStatus: PROCEDURE [offset: VM.PageOffset] RETURNS [Status] =
    BEGIN
    w: CARDINAL = offset/Environment.bitsPerWord;
    bit: Environment.Word = Inline.BITSHIFT[
      1, offset MOD Environment.bitsPerWord];
    RETURN[IF Inline.BITAND[pAllocationMap[w], bit] = 0 THEN free ELSE busy]
    END;

  SetStatus: PROCEDURE [offset: VM.PageOffset, status: Status] =
    BEGIN
    w: CARDINAL = offset/Environment.bitsPerWord;
    bit: Environment.Word = Inline.BITSHIFT[
      1, offset MOD Environment.bitsPerWord];
    pAllocationMap[w] ← Inline.BITOR[
      Inline.BITAND[pAllocationMap[w], Inline.BITNOT[bit]],
      IF status = free THEN 0 ELSE bit]
    END;
```

```
END.
```

May 22, 1978 10:57 AM	*McJones*	*Create file*
August 4, 1978 9:42 AM	*McJones*	*Allocate allocationMap locally; add initialization, condit*
***ion variable*		
August 18, 1978 11:53 AM	*McJones*	*Initialization followed module END!*
September 7, 1978 10:42 AM	*Redell*	*Allow allocation/deallocation of zero-length buffers*
September 8, 1978 2:00 PM	*McJones*	*Fix check for end of intervalBuffer in Allocate*
October 5, 1979 9:03 AM	*McJones*	*Allocation map in heap*
April 2, 1980 1:54 PM	*Knutsen*	*Add InitializeSwapBuffer*
September 23, 1980 12:15 PM	*McJones*	*Check that buffer is not mapped in Deallocate*

```
-- Swapper>SwapperExceptionImpl.mesa (last edited by: Forrest on: April 28, 1980  10:19 AM)

DIRECTORY
  CachedRegion: FROM "CachedRegion" USING [Operation, Outcome],
  Frame: FROM "Frame" USING [Alloc, Free],
  RuntimeInternal: FROM "RuntimeInternal" USING [MakeFsi],
  SwapperException: FROM "SwapperException",
  SwapperPrograms: FROM "SwapperPrograms",
  VM: FROM "VM" USING [PageNumber];

SwapperExceptionImpl: MONITOR
  IMPORTS Frame, RuntimeInternal EXPORTS SwapperException, SwapperPrograms =
  BEGIN OPEN SwapperException;
  QE: TYPE = RECORD [
    pQEPrev: POINTER TO QE,
    page: VM.PageNumber,
    operation: CachedRegion.Operation,
    outcome: CachedRegion.Outcome];
  PQE: TYPE = POINTER TO QE;
  fsiQE: CARDINAL = RuntimeInternal.MakeFsi[SIZE[QE]];
  pQELast: PQE ← NIL; -- points to last of fifo queue of exception reports
  report: CONDITION;
  Await: PUBLIC ENTRY PROCEDURE
    RETURNS [
      page: VM.PageNumber, operation: CachedRegion.Operation,
      outcome: CachedRegion.Outcome] =
    BEGIN
    pQE, pQENext: PQE;
    -- Wait for queue nonempty
    WHILE pQELast = NIL DO WAIT report ENDLOOP;
    -- Remove first report from queue and return it
    FOR pQE ← pQELast, pQE.pQEPrev WHILE pQE.pQEPrev ~= NIL DO
      pQENext ← pQE ENDLOOP;
    [, page, operation, outcome] ← pQE↑;
    IF pQE = pQELast THEN pQELast ← NIL ELSE pQENext.pQEPrev ← NIL;
    Frame.Free[pQE]
    END;

  Report: PUBLIC ENTRY PROCEDURE [
    page: VM.PageNumber, operation: CachedRegion.Operation,
    outcome: CachedRegion.Outcome] =
    BEGIN
    -- Add report to end of queue
    pQE: PQE = LOOPHOLE[Frame.Alloc[fsiQE], PQE];
    pQE↑ ← [pQELast, page, operation, outcome];
    pQELast ← pQE;
    NOTIFY report
    END;

  END.
LOG
Time: June 6, 1978  3:31 PM    By: McJones      Action: Created file (stub only)
Time: July 31, 1978  10:53 AM  By: McJones      Action: Replace stubs with real implementati
**on
Time: April 28, 1980  10:18 AM  By: Forrest     Action: FrameOps => Frame.
```

```
-- Swapper>SwapperControl.mesa (last edited by Knutsen on April 15, 1980 12:05 PM)

DIRECTORY
  CachedRegion: FROM "CachedRegion" USING [
    age, Apply, Outcome, pagefault, pageTop],
  Environment: FROM "Environment" USING [PageCount, PageNumber],
  MStore: FROM "MStore" USING [AwaitBelowThreshold, SetThreshold],
  PageFault: FROM "PageFault" USING [Await],
  Process: FROM "Process" USING [Detach, Yield],
  Runtime: FROM "Runtime" USING [CallDebugger],
  StoragePrograms: FROM "StoragePrograms",
  SwapperException: FROM "SwapperException" USING [Report],
  SwapperPrograms: FROM "SwapperPrograms" USING [
    CachedRegionImplB, CachedSpaceImpl, PageFaultImpl, InitializeSwapBuffer,
    SwapperExceptionImpl, SwapTaskImpl];

SwapperControl: PROGRAM
  IMPORTS
    CachedRegion, MStore, PageFault, Process, Runtime, SwapperException,
    SwapperPrograms
  EXPORTS StoragePrograms =
  BEGIN
  PageFaultProcess: PROCEDURE =
    BEGIN
    page: Environment.PageNumber;
    outcome: CachedRegion.Outcome;
    DO
      page ← PageFault.Await[];
      outcome ← CachedRegion.Apply[page, CachedRegion.pagefault].outcome;
      -- starts region coming in, faulting process will be woken up when it is in.
      WITH outcome SELECT FROM
        ok => NULL;
        regionDMissing, spaceDMissing, error =>
          SwapperException.Report[page, CachedRegion.pagefault, outcome];
        ENDCASE => ERROR
      ENDLOOP
    END;

  ReplacementProcess: PUBLIC PROCEDURE [threshold: Environment.PageCount] =
    BEGIN
    deadlockPasses: CARDINAL = 1000; -- meant to be about 60 sec of trying.
    passes: CARDINAL;
    newCycle: BOOLEAN;
    page, pageNext: Environment.PageNumber;
    outcome: CachedRegion.Outcome;
    MStore.SetThreshold[threshold];
    page ← FIRST[Environment.PageNumber];
    passes ← 0;
    DO
      --FOREVER--
      newCycle ← MStore.AwaitBelowThreshold[];
      IF newCycle THEN passes ← 0;
      [outcome, pageNext] ← CachedRegion.Apply[page, CachedRegion.age];
      -- ages this region. If it is old enough, it will be swapped out.
      SELECT outcome.kind FROM
        ok => NULL;
        spaceDMissing => SwapperException.Report[page, CachedRegion.age, outcome];
        ENDCASE => ERROR;
      IF pageNext < CachedRegion.pageTop THEN page ← pageNext
```



```
      ELSE
        BEGIN
        page ← FIRST[Environment.PageNumber];
        IF (passes ← passes + 1) >= deadlockPasses THEN
          Runtime.CallDebugger["No regions to swap out!"];
        Process.Yield[]; -- give others a chance to finish..

        END;
      ENDLOOP
    END;

  InitializeSwapper: PUBLIC PROCEDURE [pMapLogDesc: LONG POINTER TO UNSPECIFIED] =
    BEGIN
    -- CachedRegionImplA is started by PilotControl (see StoragePrograms).
    -- MStoreImpl is started by PilotControl (see StoragePrograms).
    -- SimpleSpaceImpl is started by PilotControl (see StoragePrograms).
    -- ResidentMemoryImpl is started by PilotControl (see StoragePrograms).
    -- ResidentHeapImpl is started by PilotControl (see StoragePrograms).
    START SwapperPrograms.CachedSpaceImpl;
    START SwapperPrograms.CachedRegionImplB[pMapLogDesc];
    SwapperPrograms.InitializeSwapBuffer[];
    -- (CachedSpaceImpl must be started first.)
    START SwapperPrograms.PageFaultImpl;
    START SwapperPrograms.SwapTaskImpl;
    START SwapperPrograms.SwapperExceptionImpl;
    Process.Detach[FORK PageFaultProcess[]];
    -- (ReplacementProcess is forked elsewhere.)

    END;

  END.
LOG
For earlier log entries see Teak archive version.
Time: January 31, 1980 1:03 PM By: Knutsen/McJones Action: Add Region Ca
**che parameters/switch to InitializeSwapper starting interface.
Time: February 21, 1980 6:53 PM By: Knutsen Action: Implement Region Cache
**parameters. Don't START MStoreImpl (now started by PilotControl).
Time: March 20, 1980 12:50 PM By: Knutsen Action: Made ReplacementProcess detect d
**eadlocks.
Time: April 15, 1980 12:05 PM By: Knutsen Action: Various components started early by
**PilotControl.
```


SwapperControl.mesa. 15-Apr-80 12:06:05 2

```
-- Swapper>SwapTaskImpl.mesa (last edited by Forrest on April 28, 1980 11:31 AM)
-- Things to consider:
-- 1) When converting to multi-MDS, PState must be long pointer; may only free frame from same
**MDS as allocated it!
-- 2) Change Advance to return region, action, bufferPage, and count (or lastFlag) separately

DIRECTORY
  CachedRegion: FROM "CachedRegion" USING [Desc],
  Frame: FROM "Frame" USING [Alloc, Free],
  PageMap: FROM "PageMap" USING [Flags],
  RuntimeInternal: FROM "RuntimeInternal" USING [MakeFsi],
  SwapperPrograms: FROM "SwapperPrograms",
  SwapTask: FROM "SwapTask",
  VM: FROM "VM" USING [Interval, PageNumber, PageCount];

SwapTaskImpl: MONITOR
  IMPORTS Frame, RuntimeInternal EXPORTS SwapperPrograms, SwapTask SHARES SwapTask
  =
  BEGIN OPEN SwapTask;
  fsiState: CARDINAL = RuntimeInternal.MakeFsi[SIZE[State]];
  swapTaskList: PState ← NIL;
  Fork: PUBLIC ENTRY PROCEDURE [
    swapInterval: VM.Interval, flags: PageMap.Flags, bufferPage: VM.PageNumber,
    swapCount: VM.PageCount, region: CachedRegion.Desc]
    RETURNS [forked: BOOLEAN] =
    BEGIN
    pState: PState;
    IF ~(forked ← swapCount > 0) THEN RETURN;
    pState ← LOOPHOLE[Frame.Alloc[fsiState], PState];
    pState↑ ← [swapTaskList, swapInterval, flags, bufferPage, swapCount, region];
    swapTaskList ← pState;
    END;

  Advance: PUBLIC ENTRY PROCEDURE [interval: VM.Interval] RETURNS [state: State] =
    BEGIN
    pState, pStatePrev: PState;
    pState ← swapTaskList;
    DO
      OPEN pState;
      --assert--
      IF pState = NIL THEN ERROR;
      IF interval.page IN [bufferPage..bufferPage + swapInterval.count) THEN EXIT;
      pStatePrev ← pState;
      pState ← next;
      ENDLOOP;
    --assert--
    IF pState.countRemaining < interval.count THEN ERROR;
    pState.countRemaining ← pState.countRemaining - interval.count;
    state ← pState↑;
    IF pState.countRemaining = 0 THEN
      BEGIN
      IF pState = swapTaskList THEN swapTaskList ← pState.next
      ELSE pStatePrev.next ← pState.next;
      Frame.Free[pState];
      END;
    END;

  END.
LOG
```



```
Time: Sometime in March 1978 By: McJones Action: Created file
Time: September 5, 1978 4:39 PM By: Redell Action: Removed 'Action' records. Added 's
**wapCount' argument and 'forked' result to Fork.
Time: October 13, 1979 12:26 PM By: Knutsen Action: Swap task is now a swap
**unit, not a region.
Time: November 26, 1979 12:20 PM By: Gobbel Action: Advance now uses count i
**nstead of doing one page at a time.
Time: April 28, 1980 11:31 AM By: Forrest Action: FrameOps = > Frame.
```

```
-- Transactions>TransactionLogImpl.mesa (last edited by Gobbel on September 26, 1980 4:33 P
**M)

DIRECTORY
  File --USING [xxx]--
  ,
  FileInternal --USING [xxx]--
  ,
  Inline --USING [xxx]--
  ,
  PilotFileTypes --USING [xxx]--
  ,
  SimpleSpace --USING [xxx]--
  ,
  Space --USING [xxx]--
  ,
  Transaction --USING [xxx]--
  ,
  TransactionInternal --USING [xxx]--
  ,
  TransactionState --USING [xxx]--
  ,
  Utilities --USING [xxx]--
  ;

TransactionLogImpl: PROGRAM
  -- not a MONITOR. The procedures in this module are only called from inside the monitor of Tra
**nsactionStateImpl.

  IMPORTS Inline, SimpleSpace, TransactionInternal, Utilities
  EXPORTS TransactionInternal
  SHARES File --USING[Capability]-- =
  BEGIN OPEN TransactionInternal;
  bufferSpace: SimpleSpace.Handle;
  countBuffer: Space.PageCount = 40;
  BugType: TYPE = {illegalLogOperation, logForInactiveTransaction};
  Bug: PRIVATE --PROGRAMMING--ERROR [BugType] = CODE;
  -- Initialization
  InitializeLogsA: PUBLIC PROCEDURE =
    -- Called once at system startup - create simple spaces for later use.
    BEGIN
    IF disabled THEN RETURN;
    bufferSpace ← SimpleSpace.Create[countBuffer, hyperspace];
    END;

  InitializeLogsB: PUBLIC PROCEDURE[] =
    BEGIN -- may be something here someday-- END;
  ----------------------------------------

  AssureLogRoom: PROCEDURE [
    pTrans: LONG POINTER TO active TxDescriptor, countOp: FilePageCount,
    fileOps: TransactionState.FileOps] =
    -- Assures that there is enough room in the log file for this transaction to hold countOp addition
**al pages.
    -- If no log file is allocated yet, allocates a log file for transaction. Uses a preallocated log file if
**one available, else creates one.
    -- May raise Volume.InsufficientSpace.
    BEGIN
```



```
    countGrowExtra: FilePageCount = 10;
    -- add this much extra when we have to grow a log file.
    trUnused: TxDescPtr;
    IF ~pTrans.fileInUse THEN -- allocate a log file..
      BEGIN
      FOR txUnused: ValidTxIndex IN ValidTxIndex DO
        -- look for an unused, preallocated file..
        trUnused ← @state[txUnused];
        IF ~trUnused.fileInUse AND trUnused.logFile ~= File.nullID THEN
          -- found a preallocated file. Swap file info into current transaction.
          {
          tempID: File.ID;
          tempPage: FilePageNumber;
          tempID ← pTrans.logFile;
          tempPage ← pTrans.logFileSize;
          pTrans.logFile ← trUnused.logFile;
          pTrans.logFileSize ← trUnused.logFileSize;
          trUnused.logFile ← tempID;
          trUnused.logFileSize ← tempPage;
          TransactionInternal.UpdateStateFile[];
          EXIT};
        REPEAT
          FINISHED => -- have to create a file..
            {
            pTrans.logFile ← fileOps.CreateFile[
              (pTrans.logFileSize ← countOp + countGrowExtra),
              PilotFileTypes.tTransactionLog];
            -- may raise Volume.InsufficientSpace.
            TransactionInternal.UpdateStateFile[];
            fileOps.MakeFilePermanent[pTrans.logFile]};
        -- (now that we've remembered its ID)

        ENDLOOP;
      pTrans.fileInUse ← TRUE;
      pTrans.pageFree ← FIRST[FilePageNumber];
      pTrans.countPrevLogEntry ← 0;
      END;
    -- grow the log file as necessary..
    IF pTrans.pageFree + countOp > pTrans.logFileSize THEN -- grow the log file..
      {
      tempSize: FilePageCount = countOp + pTrans.pageFree + countGrowExtra;
      -- that needed + extra.
      fileOps.SetFileSize[pTrans.logFile, tempSize];
      -- may raise Volume.InsufficientSpace.
      pTrans.logFileSize ← tempSize}; -- now that we have sufficient space.

    END;

  LogInternal: PUBLIC PROCEDURE [
    txi: ValidTxIndex, op: TransactionState.LogEntryPtr,
    fileOps: TransactionState.FileOps] =
    -- May raise Volume.InsufficientSpace.
    BEGIN
    entryData: LONG POINTER TO LogFileEntryData;
    pTrans: TxDescPtr = @state[txi];
    countOp: FilePageCount = -- total number of pages to log for this operation.
      countLogEntry +
        (WITH oper: op SELECT FROM
```

```
        setContents => ShortenPageCount[oper.size],
        ENDCASE => 0);
    WITH trans: pTrans SELECT FROM
      active =>
        BEGIN
        AssureLogRoom[@trans, countOp, fileOps];
        -- may raise Volume.InsufficientSpace.
        WITH oper: op SELECT FROM
          setContents =>
            -- First log the data, then go back and log the operation info.  That way, if the operation i
**nfo is in the log file, we know that the following data is also in the log file.
            BEGIN
            count: FilePageCount;
            baseLog: File.PageNumber ← trans.pageFree + countLogEntry;
            -- (skip over header page)
            baseData: FilePageNumber ← ShortenPageNumber[oper.window.base];
            countRemaining: FilePageCount ← ShortenPageCount[oper.size];
            IF countRemaining > 0 --UNTIL count exhausted-- THEN
              DO
                count ← MIN[countBuffer, countRemaining];
                SimpleSpace.Map[
                  handle: bufferSpace,
                  window: [[trans.logFile, permissions], baseLog], andPin: FALSE,
                  countMapped: count];
                SimpleSpace.CopyIn[
                  handle: bufferSpace, window: [oper.window.file, baseData],
                  countMapped: count];
                SimpleSpace.Unmap[bufferSpace];
                baseLog ← baseLog + countBuffer;
                baseData ← baseData + countBuffer;
                IF countRemaining < countBuffer THEN EXIT;
                countRemaining ← countRemaining - countBuffer;
                ENDLOOP;
              END;
            create, delete, move, makeMutable, makeTemporary, shrink => NULL;
            ENDCASE => ERROR Bug[illegalLogOperation];
          -- Log the operation info:
          SimpleSpace.Map[
            handle: bufferSpace,
            window: [[trans.logFile, permissions], trans.pageFree], andPin: FALSE,
            countMapped: countLogEntry];
          SimpleSpace.Kill[bufferSpace];
          entryData ← LOOPHOLE[SimpleSpace.LongPointer[bufferSpace]];
          BEGIN OPEN e: entryData.logFileEntry;
          e.seal ← logFileFormatVersion;
          e.transactionHandle ← e.transactionHandle2 ← Handle[trans.txEdition, txi];
          e.countPrevEntry ← trans.countPrevLogEntry;
          e.op ← op↑;
          END;
          entryData.checksumData ← Checksum[
            0, SIZE[LogFileEntry], @entryData.logFileEntry];
          SimpleSpace.Unmap[bufferSpace];
          trans.pageFree ← trans.pageFree + countOp;
          trans.countPrevLogEntry ← countOp;
          -- remember count of this entry for next time.

          END;
      ENDCASE => ERROR Bug[logForInactiveTransaction];
```



```
    END;

  END.
```

LOG

June 2, 1980 2:05 PM	*Gobbel*	*Created file.*
July 21, 1980 2:23 PM	*McJones*	*Converted to new UNCOUNTED ZONE representation.*
July 22, 1980 2:59 PM	*Gobbel*	*Now to make it work....*
August 27, 1980 1:04 PM	*Knutsen*	*Merged Log Table into State Table. Major cleanup.*

```
-- Transactions>TransactionImpl.mesa (last edited by Fay on October 16, 1980 4:16 PM)
-- This module implements high-level transaction operations: Commit, Abort, Crash Recovery.
-- Serial access to individual transactions is accomplished by checking-out a transaction from Tra
**nsactionStateImpl.
-- This module runs at a level within Pilot above the VMMgr.

DIRECTORY
  Environment USING [wordsPerPage],
  File USING [
    Capability, Create, Delete, Error, GetSize, MakeImmutable, MakePermanent,
    Move, nullCapability, nullID, PageCount, SetSize, Unknown],
  Inline USING [LongCOPY],
  KernelFile USING [
    CreateWithID, ExistingFile, GetRootFile, MakeMutable, MakeTemporary],
  KernelSpace USING [ReleaseFromTransaction],
  PilotFileTypes USING [tTransactionStateFile],
  PilotMP USING [cTransactionCrashRecovery],
  PilotSwitches USING [switches --x--],
  ProcessorFace USING [SetMP],
  Runtime USING [CallDebugger, IsBound],
  SimpleSpace USING [Kill, Map, Unmap],
  Space USING [
    Create, Delete, defaultBase, defaultWindow, Error, ForceOut, GetWindow,
    Handle, Kill, LongPointer, LongPointerFromPage, Map, PageCount, PageNumber,
    PageOffset, Unmap, virtualMemory, VMPageNumber, WindowOrigin],
  SpecialTransaction USING [ClientStart],
  StoragePrograms USING [],
  TemporaryTransaction USING [],
  Transaction USING [Handle],
  TransactionInternal USING [
    Checksum, countLogEntry, disabled, FilePageCount, FilePageNumber, Handle,
    LogFileEntry, LogFileEntryData, LogFileEntryPtr, logFileFormatVersion,
    MaybeCrash, permissions, ShortenPageCount, standardLogFileCount, state, State,
    state1Window, state2Window, stateCount, StateData, StateEdition,
    stateFormatVersion, stateSpace, TxDescPtr, TxIndex, ValidTxEdition,
    ValidTxIndex],
  TransactionState USING [
    InitializeStateA, InitializeStateB, LogOp, NoMoreOperations, RecordCommit,
    ReleaseTransaction, SpaceNodePtr],
  Utilities USING [LongPointerFromPage],
  Volume USING [systemID, Unknown];

TransactionImpl: MONITOR
  -- This module's monitor is only used to control access to the two utility spaces used by Abort an
**d crash recovery. Access to individual transactions is controlled by the monitor of Transaction
**StateImpl.

  IMPORTS
    File, Inline, KernelFile, KernelSpace, PilotSwitches, ProcessorFace, Runtime,
    SimpleSpace, Space, SpecialTransaction, TransactionInternal, TransactionState,
    Utilities, Volume
  EXPORTS StoragePrograms, TemporaryTransaction, Transaction
  SHARES File --USING [permissions]-- =
  BEGIN OPEN TransactionInternal;
  --Transaction.--
  Handle: PUBLIC TYPE = TransactionInternal.Handle;
  logSpaceSize: Space.PageCount ← 500;
  -- arbitrary number, big enough to avoid doing lots of Maps and Unmaps on logSpace
```



```
  entrySpace, logSpace, clientSpace: Space.Handle; -- for Abort processing.
  BugType: TYPE = {
    xxx, abortFileError, abortSpaceError, illegalLogOperation,
    inactiveTransaction, invalidLogEntry, stateDataNotEqualToState,
    unexpectedErrorRestoringSpaces, unexpectedErrorValidateLogFile};
  -- BugType: TYPE = {commitSpaceError, crashRecoveryFileError, crashRecoverySpaceError, ill
**egalLogOperation, impossibleStateInconsistency, invalidLogHandle, invalidVersionSeal, logAlr
**eadyAllocated, logFileDisappeared, logFileVolumeError, missingPinnedPageGroup, remoteLog
**File, spaceAlreadyInTransaction, tooManyTransactions};
  Bug: PRIVATE --PROGRAMMING--ERROR [BugType] = CODE;
  ------------------------------------------
  -- Monitor External Procedures
  ------------------------------------------
  --TemporaryTransaction.--
  DisableTransactions: PUBLIC PROCEDURE = {disabled ← TRUE};

  Abort: PUBLIC PROCEDURE [transaction: TransactionInternal.Handle] =
    BEGIN
    -- Save the windows that the spaces are currently mapped to, and unmap them:
    spaceList: TransactionState.SpaceNodePtr;
    IF disabled THEN RETURN;
    spaceList ← TransactionState.NoMoreOperations[transaction];
    -- may raise InvalidHandle
    FOR space: TransactionState.SpaceNodePtr ← spaceList, space.nextSpace UNTIL
      space = NIL DO
      space.window ← Space.GetWindow[
        space.handle !
        -- (note that GetWindow will return defaultWindow if the space is mapped as a data space.
**We will not do anything to that space during Abort processing.)

        Space.Error, File.Unknown, Volume.Unknown => {
          space.window ← Space.defaultWindow; CONTINUE}];
      -- we assume client deleted the space or the file.
      IF space.window # Space.defaultWindow THEN
        Space.Unmap[
          space.handle !
          Space.Error => {space.window ← Space.defaultWindow; CONTINUE}];
      ENDLOOP;
    -- Back out the changes that have been made to the client's files:
    RestoreFilesInTransaction[transaction.index];
    -- Map the spaces back to their windows and deallocate the space list elements:
    FOR space: TransactionState.SpaceNodePtr ← spaceList, space.nextSpace UNTIL
      space = NIL DO
      IF space.window # Space.defaultWindow THEN
        Space.Map[
          space.handle, space.window !
          Space.Error, File.Unknown, Volume.Unknown => CONTINUE;
          -- we assume client deleted the space or the file was deleted in the course of Abort proce
**ssing.
          ANY => ERROR Bug[unexpectedErrorRestoringSpaces]];
      -- client messing with files during transaction!
      KernelSpace.ReleaseFromTransaction[
        space.handle, transaction, --andInvalidate:--FALSE
        -- (if we were going to invalidate, we would have to do it earlier.)
        ! Space.Error => CONTINUE];
      ENDLOOP;
    ReleaseLog[transaction.index];
    TransactionState.ReleaseTransaction[transaction];
```

```
    END;
  -- Note: Begin[] is implemented in TransactionStateImpl.

  Commit: PUBLIC PROCEDURE [transaction: TransactionInternal.Handle] =
    BEGIN
    -- At present, all transaction space and file operations use an "undo log", so Commit is very si
**mple: force out all mapped spaces, record the commit in the Transaction State file, then clear th
**e log file, then free the tx entry in the State.
    spaceList: TransactionState.SpaceNodePtr;
    IF disabled THEN RETURN;
    spaceList ← TransactionState.NoMoreOperations[transaction];
    -- may raise InvalidHandle
    FOR space: TransactionState.SpaceNodePtr ← spaceList, space.nextSpace UNTIL
      space = NIL DO
      Space.ForceOut[space.handle ! Space.Error => CONTINUE];
      -- If error, we assume client deleted the space.
      KernelSpace.ReleaseFromTransaction[
        space.handle, transaction ! Space.Error => CONTINUE];
      ENDLOOP;
    TransactionState.RecordCommit[transaction];
    -- now the log file will be ignored if we crash.
    ReleaseLog[transaction.index];
    TransactionState.ReleaseTransaction[transaction];
    END;
  ----------------------------------------
  -- Initialization and Crash Recovery
  ----------------------------------------
  --StoragePrograms.--

  InitializeTransactionData: PUBLIC PROCEDURE[] =
    -- Initializes nullHandle exported variable, creates SimpleSpaces for logging.
    BEGIN
    TransactionState.InitializeStateA[]; -- creates stateSpace.

    END;
  --StoragePrograms.--

  RecoverTransactions: PUBLIC --EXTERNAL--PROCEDURE[] =
    -- Initializes Transaction modules, performs Crash Recovery.
    -- This procedure is called during Pilot initialization after the FileMgr and VMMgr are started, bu
**t before anyone is using transactions, so (1) we can use high-level File and Space operations, a
**nd (2) no one will call Log from inside FileImpl, which would trigger a deadlock since we are usi
**ng File operations ourself.
    BEGIN
    ProcessorFace.SetMP[PilotMP.cTransactionCrashRecovery];
    IF PilotSwitches.switches.x = down THEN Runtime.CallDebugger["Key Stop X"L];
    IF Runtime.IsBound[SpecialTransaction.ClientStart] THEN
      SpecialTransaction.ClientStart[];
    logSpace ← Space.Create[logSpaceSize, Space.virtualMemory];
    clientSpace ← Space.Create[logSpaceSize, Space.virtualMemory];
    entrySpace ← Space.Create[countLogEntry, Space.virtualMemory];
    -- Create real swap units to avoid UniformSwapUnit deadlock bug:
    FOR base: Space.PageOffset ← 0, base + 20 WHILE base < logSpaceSize DO
      [] ← Space.Create[MIN[20, logSpaceSize - base], logSpace, base];
      [] ← Space.Create[MIN[20, logSpaceSize - base], clientSpace, base];
      ENDLOOP;
    RecoverState[]; -- performs Crash Recovery on the State Table.
    TransactionState.InitializeStateB[];
```



```
    FOR txi: ValidTxIndex IN ValidTxIndex DO
      ValidateLogFile[txi];
      WITH trans: state[txi] SELECT FROM
        active =>
          BEGIN
          trans.noMoreOperations ← TRUE; -- (keeps ReleaseTransaction happy.)
          trans.spaceList ← NIL;
          TransactionInternal.MaybeCrash[1];
          IF ~trans.committed THEN RestoreFilesInTransaction[txi, TRUE];
          TransactionInternal.MaybeCrash[5];
          ReleaseLog[txi];
          TransactionState.ReleaseTransaction[
            TransactionInternal.Handle[trans.txEdition, txi]]
          END;
        free => NULL;
        ENDCASE;
      ENDLOOP;
    END;
  ----------------------------------------
  -- Support routines
  ----------------------------------------

  RestoreFilesInTransaction: ENTRY PROCEDURE [
    txi: ValidTxIndex, crashRecovery: BOOLEAN ← FALSE] =
    -- Restores client files involved in the transaction.
    -- Notice that this routine may be (possibly partially) executed an arbitrary number of times if w
**e crash during crash recovery.
    BEGIN
    pTrans: TxDescPtr = @state[txi];
    entry: LogFileEntry;
    logPage: Space.PageNumber ← Space.VMPageNumber[logSpace];
    clientPage: Space.PageNumber ← Space.VMPageNumber[clientSpace];
    pageBase, pageEntry: FilePageNumber;
    WITH trans: pTrans SELECT FROM
      active =>
        IF trans.pageFree > 0 THEN -- there's something in the log..
          BEGIN
          pageEntry ← trans.pageFree - trans.countPrevLogEntry;
          -- start with the last entry, process file back to front
          entry.countPrevEntry ← trans.countPrevLogEntry;
          -- so maybe we have to do some adjusting for the first one, but this saves keeping this nu
**mber in the state file...
          DO
            -- UNTIL all entries processed
            pageBase ←
              IF pageEntry < logSpaceSize THEN 0
              ELSE (pageEntry + countLogEntry) - logSpaceSize;
            -- first find the minimum possible pageBase value that would include the header page f
**or the next entry to be processed...
            IF pageEntry + entry.countPrevEntry > pageBase + logSpaceSize THEN
              pageBase ← MIN[
                pageEntry + (entry.countPrevEntry - countLogEntry), pageEntry];
            -- then, if not all of the entry about to be mapped would be in the space with that pageB
**ase, add to it to get as much as possible of that entry (all of it, if it will fit) into the space, rememb
**ering not to make pageBase greater than pageEntry.
            Space.Map[logSpace, [[trans.logFile, permissions], pageBase]];
            WHILE pageEntry >= pageBase DO
              -- scope of IgnorableError --
```

```
        BEGIN
        ENABLE
          BEGIN
          File.Unknown, Volume.Unknown => GOTO IgnorableError;
          File.Error =>
            SELECT type FROM
              immutable, insufficientPermissions, notImmutable =>
                GOTO IgnorableError;
              ENDCASE => ERROR Bug[abortFileError];
          END;
        entry ← LOOPHOLE[Utilities.LongPointerFromPage[
          logPage + (pageEntry - pageBase)], LogFileEntryPtr]↑;
        IF entry.seal # logFileFormatVersion OR
          entry.transactionHandle.index # txi OR
          entry.transactionHandle.txEdition # trans.txEdition OR
          entry.transactionHandle2.index # txi OR
          entry.transactionHandle2.txEdition # trans.txEdition THEN
          ERROR Bug[invalidLogEntry];
        WITH entry.op SELECT FROM
          create =>
            KernelFile.CreateWithID[
              volume, size, type, id ! KernelFile.ExistingFile => CONTINUE];
          -- (we already created it during previous incomplete crash Recovery.)

          delete => File.Delete[file];
          makeMutable => KernelFile.MakeMutable[file];
          makeTemporary =>
            IF crashRecovery THEN File.Delete[file]
            ELSE KernelFile.MakeTemporary[file];
          move => File.Move[file, volume];
          setContents =>
            BEGIN
            -- Note: must do things in the proper order below! (FIRST set contents of file, THE
**N MakePermanent, THEN MakeImmutable)
            fileSize: File.PageCount = File.GetSize[window.file];
            mapSize: Space.PageCount ← MIN[
              ShortenPageCount[size], logSpaceSize];
            logData: LONG POINTER TO UNSPECIFIED ←
              LOOPHOLE[Utilities.LongPointerFromPage[
              logPage + (pageEntry - pageBase) + countLogEntry]];
            clientData: LONG POINTER TO UNSPECIFIED ←
              LOOPHOLE[Utilities.LongPointerFromPage[clientPage]];
            clientWindow: Space.WindowOrigin ← window;
            IF fileSize < window.base + size THEN
              File.SetSize[window.file, window.base + size];
            IF pageEntry + countLogEntry + size > pageBase + logSpaceSize
              THEN
              BEGIN -- get logSpace and clientWindow in sync
              pageBase ← pageEntry + countLogEntry;
              Space.Unmap[logSpace];
              Space.Map[logSpace, [[trans.logFile, permissions], pageBase]];
              logData ← LOOPHOLE[Utilities.LongPointerFromPage[logPage]];
              END;
            DO
              Space.Map[clientSpace, clientWindow];
              Inline.LongCOPY[
                from: logData, to: clientData,
                nwords:
                MIN[
```



```
                  mapSize, ShortenPageCount[
                    (window.base + size) -
                      clientWindow.base]]*Environment.wordsPerPage];
                IF clientWindow.base + mapSize >= window.base + size THEN
                  EXIT;
                clientWindow.base ← clientWindow.base + mapSize;
                pageBase ← pageBase + mapSize;
                Space.Unmap[clientSpace];
                Space.Unmap[logSpace];
                Space.Map[logSpace, [[trans.logFile, permissions], pageBase]];
                ENDLOOP;
              Space.Unmap[clientSpace];
              IF makePermanent THEN File.MakePermanent[window.file];
              IF makeImmutable THEN File.MakeImmutable[window.file];
              TransactionInternal.MaybeCrash[6];
              END;
            shrink => File.SetSize[file, size];
            ENDCASE => ERROR Bug[illegalLogOperation];
          EXITS IgnorableError => NULL;
          -- come here on all errors that may result from trying to execute a log entry that has al
**ready been executed (e.g., trying to delete a file that already been deleted). This can occur if th
**ere was a crash during a previous attempt to process the log for this transaction.

            END;
            IF pageEntry = 0 THEN GOTO Done;
            pageEntry ← pageEntry - entry.countPrevEntry;
            ENDLOOP;
          Space.Unmap[logSpace];
          IF pageBase = 0 THEN EXIT;
          REPEAT Done => Space.Unmap[logSpace];
          ENDLOOP;
        END;
      ENDCASE => ERROR Bug[inactiveTransaction];
    -- the transaction should have been active.

    END;

  ReleaseLog: ENTRY PROCEDURE [txi: ValidTxIndex] = {ReleaseLogInternal[txi]};

  ReleaseLogInternal: INTERNAL PROCEDURE [txi: ValidTxIndex] =
    -- Resets the transaction's log file to the standard size, zeroes its contents, then releases it for
**general use.
    BEGIN
    logData: LONG POINTER TO UNSPECIFIED ← Space.LongPointer[logSpace];
    entryData: LONG POINTER TO UNSPECIFIED ← Space.LongPointer[entrySpace];
    trans: TxDescPtr = @state[txi];
    IF trans.fileInUse THEN
      BEGIN
      Space.Map[entrySpace, [[trans.logFile, permissions], 0]];
      Space.Kill[entrySpace];
      entryData↑ ← 0;
      -- first clear page 0 to avoid confusing crash recovery if we crash while releasing the file...
      Inline.LongCOPY[
        from: entryData, to: entryData + 1,
        nwords: (countLogEntry*Environment.wordsPerPage) - 1];
      Space.Unmap[entrySpace];
      IF trans.logFileSize # standardLogFileCount THEN
        File.SetSize[
```

```
        [trans.logFile, permissions], trans.logFileSize ← standardLogFileCount];
      Space.Map[logSpace, [[trans.logFile, permissions], countLogEntry]];
      Space.Kill[logSpace];
      logData↑ ← 0;
      -- clear all of the pages so that we will not re-execute any entries of this log file due to any fut
**ure crashes..
      Inline.LongCOPY[
        from: logData, to: logData + 1,
        nwords:
        ((standardLogFileCount - countLogEntry)*Environment.wordsPerPage) - 1];
      Space.Unmap[logSpace];
      END;
    trans.fileInUse ← FALSE;
    END;

  RecoverState: PROCEDURE[] =
    BEGIN
    -- Finds the valid data in the old Transaction State File (creates a new State Table if none), map
**s stateSpace, and puts the State Table into it.
    -- First look for the old transaction state file. If it doesn't exist, we're just starting up with a new
**system volume, so create the file and put it into the logical volume root page. If it does exist, do
**crash recovery on it: discover which copy is the most-recent correct one, and copy it to stateSp
**ace.
    -- The present algorithm should happily handle a file containing an unreadable page if the Sca
**venger were to substitute another page for it (filled with zeroes).
    createStateTable: BOOLEAN;
    state1Edition, state2Edition: StateEdition;
    state1Valid, state2Valid: BOOLEAN;
    minStateFileCount: File.PageCount = 3*stateCount;
    -- two stable-storage copies + scratch storage.
    dataFile: File.Capability;
    workingStateWindow: Space.WindowOrigin;
    state1Space, state2Space: Space.Handle;
    state1, state2: LONG POINTER TO State;
    stateData: LONG POINTER TO StateData;
    -- (stateSpace is created by InitializeStateA.)
    stateData ← LOOPHOLE[Space.LongPointer[stateSpace]];
    state ← @stateData.state;
    IF LOOPHOLE[stateData, LONG POINTER] # LOOPHOLE[state, LONG POINTER] THEN
      ERROR Bug[stateDataNotEqualToState];
    -- we use state also as a pointer to stateData (to avoid having to keep two copies of an equivale
**nt pointer in our global frame), so, check that they are indeed equal.
    state1Space ← Space.Create[
      stateCount, Space.virtualMemory, Space.defaultBase];
    state2Space ← Space.Create[
      stateCount, Space.virtualMemory, Space.defaultBase];
    state1 ← LOOPHOLE[Space.LongPointer[state1Space]];
    state2 ← LOOPHOLE[Space.LongPointer[state2Space]];
    state1Window.base ← 0;
    state2Window.base ← state1Window.base + stateCount;
    workingStateWindow.base ← state2Window.base + stateCount;
    createStateTable ← FALSE; -- assume State file is well-formed..
    -- UNTIL the State is well-formed --
    DO
      -- scope of CreateStateFile --
      BEGIN
      dataFile ← KernelFile.GetRootFile[
        PilotFileTypes.tTransactionStateFile, Volume.systemID];
```



```
      state1Window.file ← state2Window.file ← workingStateWindow.file ← dataFile;
      IF File.GetSize[dataFile ! File.Unknown => GO TO CreateStateFile] <
        minStateFileCount THEN File.SetSize[dataFile, minStateFileCount];
      -- (fills with zeros)
      [state1Valid, state1Edition] ← CheckState[
        @state1Window, state1Space, state1];
      [state2Valid, state2Edition] ← CheckState[
        @state2Window, state2Space, state2];
      SimpleSpace.Map[stateSpace, workingStateWindow, --andPin:--FALSE];
      -- the working copy of the State.
      SimpleSpace.Kill[stateSpace];
      -- Find the latest, correct copy of the State Table, and copy it into state:
      SELECT TRUE FROM
        state1Valid AND state2Valid =>
          SELECT CompareStateEdition[value: state1Edition, to: state2Edition] FROM
            equal, greater =>
              Inline.LongCOPY[from: state1, nwords: SIZE[StateData], to: state];
            less =>
              Inline.LongCOPY[from: state2, nwords: SIZE[StateData], to: state];
            ENDCASE;
        state1Valid AND ~state2Valid =>
          Inline.LongCOPY[from: state1, nwords: SIZE[StateData], to: state];
        ~state1Valid AND state2Valid =>
          Inline.LongCOPY[from: state2, nwords: SIZE[StateData], to: state];
        ~state1Valid AND ~state2Valid => -- no valid State.
          IF ~createStateTable THEN {
            SimpleSpace.Unmap[stateSpace]; GO TO CreateStateFile}
          ELSE -- create the initial State Table..
            FOR txi: TxIndex IN TxIndex DO
              -- (TxIndex, not ValidTxIndex)
              state[txi] ←
                [stateEdition: FIRST[StateEdition], seal: stateFormatVersion,
                  txEdition: FIRST[ValidTxEdition], fileInUse: FALSE,
                  logFile: File.nullID, logFileSize: 0, vp: free[]];
            ENDLOOP;
        ENDCASE;
      EXIT; -- state is now well-formed.

      EXITS
        CreateStateFile =>
          BEGIN
          Space.Unmap[state1Space ! Space.Error => CONTINUE];
          -- ("Error[noWindow]")
          Space.Unmap[state2Space ! Space.Error => CONTINUE];
          -- ("Error[noWindow]")
          IF dataFile # File.nullCapability THEN {
            Runtime.CallDebugger[
              "Crash recovery data malformed. P(roceed) will delete it."L];
            File.Delete[dataFile ! File.Unknown => CONTINUE]};
          dataFile ← File.Create[
            Volume.systemID, minStateFileCount,
            PilotFileTypes.tTransactionStateFile];
          File.MakePermanent[dataFile];
          -- enters file in volume root page as a side effect. (yes, I'm disgusted too).
          createStateTable ← TRUE;
          -- (we will create it next time through the loop.)

          END;
```

```
    END -- scope of CreateStateFile --
    ;
    ENDLOOP; -- loop until the State is well-formed.
  -- At this point, state contains a well-formed State Table and is the latest available version.
  Space.Delete[state1Space];
  Space.Delete[state2Space];
  END;

CheckState: PROCEDURE [
  window: POINTER TO Space.WindowOrigin, space: Space.Handle,
  testState: LONG POINTER TO State]
  RETURNS [valid: BOOLEAN, testStateEdition: StateEdition] =
  -- As a side effect, maps space to window.
  BEGIN
  pStateData: LONG POINTER TO StateData = LOOPHOLE[testState];
  Space.Map[space, window↑];
  IF pStateData.checksumData # Checksum[0, SIZE[State], @pStateData.state] THEN
    {valid ← FALSE; RETURN};
  testStateEdition ← testState[FIRST[TxIndex]].stateEdition;
  FOR txi: TxIndex IN TxIndex DO
    -- check for consistency (TxIndex, not ValidTxIndex)
    IF testState[txi].seal # stateFormatVersion OR testState[txi].txEdition ~IN
      ValidTxEdition OR testState[txi].stateEdition # testStateEdition THEN {
      valid ← FALSE; RETURN};
    ENDLOOP;
  valid ← TRUE;
  RETURN;
  END;

CompareStateEdition: PROCEDURE [value: StateEdition, to: StateEdition]
  RETURNS [relationship: {less, equal, greater}] =
  -- compares "value" to "to", taking possible wraparound into account.
  BEGIN
  wrapLimit: CARDINAL = (LAST[StateEdition] - FIRST[StateEdition])/2;
  -- if more than half of the range behind, we assume it wrapped around.
  IF value = to THEN RETURN[equal]
  ELSE
    IF value < to THEN {
      IF to - value < wrapLimit THEN RETURN[less]
        -- "value" is less than, and close to "to".

      ELSE RETURN[greater]}
    ELSE --value>to--{
      IF value - to < wrapLimit THEN RETURN[greater]
        -- "value" is greater than, and close to "to".

      ELSE RETURN[less]};
  END;

ValidateLogFile: ENTRY PROCEDURE [txi: ValidTxIndex] =
  -- If transaction is active, locates end of valid log entries. If inactive, sets to standard size and c
**lears it.
  -- Sets logFileSize, fileInUse, pageFree, countPrevLogEntry, as appropriate.
  BEGIN
  pTrans: TxDescPtr = @state[txi];
  IF pTrans.logFile = File.nullID THEN pTrans.fileInUse ← FALSE
  ELSE
    -- scope of FileError --
    BEGIN
```



```
    ENABLE {
      File.Unknown => GO TO FileError;
      ANY => ERROR Bug[unexpectedErrorValidateLogFile]};
    pTrans.logFileSize ← ShortenPageCount[
      File.GetSize[File.Capability[pTrans.logFile, permissions]]];
    -- may raise File.Unknown.
    WITH trans: pTrans SELECT FROM
      active =>
        BEGIN
        countOp, countOpPrev: FilePageCount ← 0;
        -- total number of pages logged for this, previous operation.
        pageBase, pageEntry: FilePageNumber ← 0;
        entryData: LONG POINTER TO LogFileEntryData;
        -- scope of PartialLogFileEntry, etc. --
        BEGIN
        pageLog: Space.PageNumber ← Space.VMPageNumber[logSpace];
        entry: LogFileEntryPtr;
        DO
          -- UNTIL pageEntry = trans.logFileSize
          Space.Map[logSpace, [[trans.logFile, permissions], pageBase]];
          -- may raise File.Unknown.
          WHILE pageEntry < pageBase + logSpaceSize DO
            entryData ← LOOPHOLE[Space.LongPointerFromPage[
              pageLog + pageEntry - pageBase]];
            entry ← @entryData.logFileEntry;
            IF Checksum[0, SIZE[LogFileEntry], entry] # entryData.checksumData
              OR entry.seal # logFileFormatVersion OR
              entry.transactionHandle.index # txi OR
              entry.transactionHandle.txEdition # trans.txEdition OR
              entry.countPrevEntry # countOp OR entry.op.operation ~IN
              TransactionState.LogOp OR entry.transactionHandle2.index # txi OR
              entry.transactionHandle2.txEdition # trans.txEdition THEN
              GO TO InvalidLogEntry;
            countOpPrev ← countOp;
            countOp ←
              countLogEntry +
                (WITH oper: entry.op SELECT FROM
                  setContents => ShortenPageCount[oper.size],
                  ENDCASE => 0);
            IF (pageEntry ← pageEntry + countOp) > trans.logFileSize THEN
              GO TO PartialLogFileEntry; -- (equal is OK)
            IF pageEntry = trans.logFileSize THEN GO TO Done;
            ENDLOOP;
          Space.Unmap[logSpace];
          pageBase ← pageEntry;
          REPEAT Done => NULL;
          ENDLOOP;
        trans.countPrevLogEntry ← countOp;
        trans.pageFree ← pageEntry;
        EXITS
          PartialLogFileEntry => {
            Runtime.CallDebugger[
              "Incomplete Transaction Log File entry. (P)roceed will ignore it."L];
            trans.pageFree ← pageEntry - countOp;
            -- back up to last valid entry.
            trans.countPrevLogEntry ← countOpPrev};
          InvalidLogEntry => {
            IF LOOPHOLE[entryData, LONG POINTER TO LONG CARDINAL]↑ # 0 THEN
```

```
            Runtime.CallDebugger[
              "Garbage Transaction Log File entry.  (P)roceed will ignore it."L];
            trans.pageFree ← pageEntry;
            trans.countPrevLogEntry ← countOp};
          END;
          Space.Unmap[logSpace];
          IF trans.fileInUse AND trans.pageFree = 0 THEN ReleaseLogInternal[txi];
          trans.fileInUse ← (trans.pageFree > 0);
          END;
        free => {
          trans.fileInUse ← TRUE;
          -- (assures that file will be filled with zeroes by ReleaseLog.)
          ReleaseLogInternal[txi]};
        ENDCASE;
      EXITS
        FileError => {
          Runtime.CallDebugger[
            "Missing Transaction Log File.  (P)roceed will ignore it."L];
          pTrans.logFile ← File.nullID;
          pTrans.fileInUse ← FALSE};
      END;
    END;

  END.
LOG
May 8, 1980  1:21 PM Gobbel    Create file.
July 19, 1980  1:43 PM         McJones  Space.Copy = >CopyOut.  export Transaction.Handle.
July 22, 1980  4:13 PM         Gobbel   Make it real.
September 2, 1980  8:18 PM     Knutsen  Major cleanup.  Added State and LogFile crash recover
**y code.  Split code out into TransactionStateImpl.
September 8, 1980  6:53 PM     Gobbel   Reset spaceList in crash recovery.
September 15, 1980  10:22 AM   Knutsen  Made compatible with new ReleaseFromTransaction.
September 15, 1980  5:47 PM    Gobbel   Made RecoverFilesInTransaction, ValidateLogFile, and
**ReleaseLog able to deal with arbitrarily large log files.
October 16, 1980  4:16 PM      Fay      Added Volume.Unknown to catch phrases for Space.G
**etWindow and Space.Map calls in Abort; made Abort do nothing if transactions disabled.
```

```
-- Transactions>TransactionStateImpl.mesa (last edited by Gobbel on September 25, 1980 1:39 P
**M)
-- This module implements access to the transaction data base and operations on it.
-- This module runs at a level within Pilot below the FileManager and above SimpleSpace. In fact,
**Log is only called from inside the FileImpl monitor. (However, during initialization this module u
**ses the facilities of the VMMgr and FileMgr.)

DIRECTORY
  Environment USING [Base, first64K],
  Inline USING [LowHalf],
  ResidentHeap USING [FreeNode, MakeNode],
  SimpleSpace USING [CopyOut, Create, Handle],
  Space USING [Handle, WindowOrigin],
  SpecialTransaction USING [OptionalCrash],
  Transaction USING [],
  TransactionInternal USING [
    Checksum, FilePageNumber, Handle, InitializeLogsA, InitializeLogsB,
    invalidTxEdition, LogInternal, nullTxIndex, State, StateData, StateEdition,
    stateCount, TxIndex, ValidTxEdition, ValidTxIndex],
  TransactionState USING [FileOps, LogEntryPtr, SpaceNode, SpaceNodePtr],
  Volume USING [InsufficientSpace],
  Zone USING [Status];

TransactionStateImpl: MONITOR
  IMPORTS Inline, ResidentHeap, SimpleSpace, TransactionInternal, Volume
  EXPORTS SpecialTransaction, Transaction, TransactionInternal, TransactionState =
  BEGIN OPEN TransactionInternal;
  --Transaction.--
  Handle: PUBLIC TYPE = TransactionInternal.Handle;
  crashProc: SpecialTransaction.OptionalCrash ← NullProc;
  --TransactionInternal.--
  disabled: PUBLIC BOOLEAN ← FALSE;
  -- TRUE causes all (public) transaction operations to no-op.
  --Transaction.--
  InvalidHandle: PUBLIC ERROR = CODE;
  --Transaction.--
  nullHandle: PUBLIC Handle ← [txEdition: invalidTxEdition, index: nullTxIndex];
  --TransactionInternal.--
  state: PUBLIC LONG POINTER TO State;
  -- (LOOPHOLE[state] is also a pointer to stateData)
  txAllocRover: ValidTxIndex ← FIRST[ValidTxIndex];
  -- wanders along looking for a free transaction.
  --TransactionInternal.--
  state1Window, state2Window: PUBLIC Space.WindowOrigin;
  -- the two representatives for the stable storage containing the state table.
  --TransactionInternal.--
  stateSpace: PUBLIC SimpleSpace.Handle; -- the working copy of the state in VM.
  Bug: PRIVATE --PROGRAMMING--ERROR [BugType] = CODE;
  BugType: TYPE = {
    xxx, spaceAlreadyInTransaction, spaceNotInTransaction, tooManyTransactions};
  -- BugType: TYPE = {commitSpaceError, crashRecoveryFileError, crashRecoverySpaceError, ill
**egalLogOperation, impossibleStateInconsistency, invalidLogHandle, invalidVersionSeal, logAlr
**eadyAllocated, logFileDisappeared, logFileVolumeError, missingPinnedPageGroup, remoteLog
**File, spaceAlreadyInTransaction, tooManyTransactions};
  ----------------------------------
  -- Initialization
  ----------------------------------
  --TransactionState.--
```



```
  InitializeStateA: PUBLIC ENTRY PROCEDURE[] =
    -- (stateSpace must be a SimpleSpace because Log can be called from within the VMMgr moni
**tor.)
    {
    stateSpace ← SimpleSpace.Create[stateCount, hyperspace];
    -- (may want swap units someday.)
    TransactionInternal.InitializeLogsA[]};
  --TransactionState.--

  InitializeStateB: PUBLIC ENTRY PROCEDURE[] = {
    TransactionInternal.InitializeLogsB[]; UpdateStateFile[]};
  -- (now that all initialization and crash recovery has been done to State.)
  ----------------------------------
  -- Monitor External Procedures
  ----------------------------------
  --TransactionInternal.--

  MaybeCrash: PUBLIC SpecialTransaction.OptionalCrash = {crashProc[unimportance]};
  --SpecialTransaction.--

  NullProc: PUBLIC SpecialTransaction.OptionalCrash = {};
  --SpecialTransaction.--

  SetCrashProcedure: PUBLIC PROCEDURE [
    optionalCrash: SpecialTransaction.OptionalCrash] = {
    crashProc ← optionalCrash};
  ----------------------------------
  -- Monitor Entry Procedures
  ----------------------------------
  --TransactionState.--

  AddToTransaction: PUBLIC ENTRY PROCEDURE [
    transaction: TransactionInternal.Handle, space: Space.Handle] =
    BEGIN
    txi: TxIndex;
    sNode: TransactionState.SpaceNodePtr;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition ~=
      state[txi].txEdition THEN RETURN WITH ERROR InvalidHandle;
    WITH trans: state[txi] SELECT FROM
      active =>
        BEGIN
        IF trans.noMoreOperations THEN RETURN WITH ERROR InvalidHandle;
        FOR sp: TransactionState.SpaceNodePtr ← trans.spaceList, sp.nextSpace
          UNTIL sp = NIL DO
          -- (for debugging only)
          IF sp.handle = space THEN ERROR Bug[spaceAlreadyInTransaction];
          ENDLOOP;
        sNode ← first64K.NEW[TransactionState.SpaceNode];
        sNode.handle ← space;
        sNode.nextSpace ← trans.spaceList;
        -- (sNode.window is just scratch storage)
        trans.spaceList ← sNode;
        END;
      ENDCASE => RETURN WITH ERROR InvalidHandle;
    END;
  --Transaction.--

  Begin: PUBLIC ENTRY PROCEDURE RETURNS [Handle] =
```

```
    -- Finds a free transaction, makes it active, updates State.
    -- No log file is allocated at this time because client may just be allocating a handle to reserve f
**or special uses.  Star does this.
    BEGIN
    examLimit: CARDINAL = LAST[ValidTxIndex] - FIRST[ValidTxIndex] + 1;
    examined: CARDINAL ← 0;
    IF disabled THEN RETURN[nullHandle];
    -- (this will cause Space and File ops to no-op.)
    UNTIL state[txAllocRover].status = free DO
      txAllocRover ←
        IF txAllocRover < LAST[ValidTxIndex] THEN SUCC[txAllocRover]
        ELSE FIRST[ValidTxIndex];
      IF (examined ← SUCC[examined]) >= examLimit THEN
        ERROR Bug[tooManyTransactions];
      ENDLOOP;
    state[txAllocRover].vp ← active[
      noMoreOperations: FALSE, committed: FALSE, spaceList: NIL,
      pageFree: FIRST[FilePageNumber], countPrevLogEntry: 0];
    IF state[txAllocRover].fileInUse THEN ERROR Bug[xxx];
    UpdateStateFile[]; -- now the transaction has officially begun.
    RETURN[
      TransactionInternal.Handle[
        txEdition: state[txAllocRover].txEdition, index: txAllocRover]];
    END;
  --TransactionState.--

  RecordCommit: PUBLIC ENTRY PROCEDURE [transaction: TransactionInternal.Handle] =
    -- All updates to client files must have been forced out to the disk before this routine is called!
    BEGIN
    txi: TxIndex;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition ~=
      state[txi].txEdition THEN ERROR Bug[xxx];
    WITH trans: state[txi] SELECT FROM
      active => {
        IF ~trans.noMoreOperations THEN ERROR Bug[xxx];
        trans.committed ← TRUE;
        UpdateStateFile[]};
      -- if we crash after this, the undo log will be ignored.

      ENDCASE => ERROR Bug[xxx];
    END;

  Log: PUBLIC ENTRY PROCEDURE [
    transaction: TransactionInternal.Handle, op: TransactionState.LogEntryPtr,
    fileOps: TransactionState.FileOps] =
    -- May raise Volume.InsufficientSpace.
    BEGIN
    txi: TxIndex;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition ~=
      state[txi].txEdition THEN RETURN WITH ERROR InvalidHandle;
    WITH trans: state[txi] SELECT FROM
      active => {
        IF trans.noMoreOperations THEN RETURN WITH ERROR InvalidHandle;
        TransactionInternal.LogInternal[
          txi, op, fileOps ! Volume.InsufficientSpace => GO TO VolInsuffSpace]};
      ENDCASE => RETURN WITH ERROR InvalidHandle;
    EXITS VolInsuffSpace => RETURN WITH ERROR Volume.InsufficientSpace;
    END;
```



```
  --TransactionState.--

  NoMoreOperations: PUBLIC ENTRY PROCEDURE [
    transaction: TransactionInternal.Handle]
    RETURNS [spaceList: TransactionState.SpaceNodePtr] =
    -- Checks out this transaction to Abort or Commit.  Further client operations will raise InvalidHa
**ndle.
    BEGIN
    txi: TxIndex;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition ~=
      state[txi].txEdition THEN RETURN WITH ERROR InvalidHandle;
    WITH trans: state[txi] SELECT FROM
      active => {
        IF trans.noMoreOperations THEN RETURN WITH ERROR InvalidHandle;
        trans.noMoreOperations ← TRUE;
        RETURN[trans.spaceList]};
      ENDCASE => RETURN WITH ERROR InvalidHandle;
    END;
  --TransactionState.--

  ReleaseTransaction: PUBLIC ENTRY PROCEDURE [
    transaction: TransactionInternal.Handle] =
    BEGIN
    txi: TxIndex;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition ~=
      state[txi].txEdition THEN ERROR Bug[xxx];
    WITH trans: state[txi] SELECT FROM
      active =>
        BEGIN
        space, spaceNext: TransactionState.SpaceNodePtr;
        IF ~trans.noMoreOperations THEN ERROR Bug[xxx];
        FOR space ← trans.spaceList, spaceNext UNTIL space = NIL DO
          spaceNext ← space.nextSpace; first64K.FREE[@space] ENDLOOP;
        trans.txEdition ←
          IF trans.txEdition < LAST[ValidTxEdition] THEN SUCC[trans.txEdition]
          ELSE FIRST[ValidTxEdition]; -- makes the previous tx handle obsolete.
        state[txi].vp ← free[];
        -- UpdateStateFile[];  + + updating here would cost extra disk writes now, but save extra w
**ork during Crash Recovery.

        END;
      ENDCASE => ERROR Bug[xxx];
    END;
  --TransactionState.--

  WithdrawFromTransaction: PUBLIC ENTRY PROCEDURE [
    transaction: TransactionInternal.Handle, space: Space.Handle] =
    BEGIN
    txi: TxIndex;
    sNodePrevP: LONG POINTER TO TransactionState.SpaceNodePtr;
    sNode: TransactionState.SpaceNodePtr;
    IF (txi ← transaction.index) ~IN ValidTxIndex OR transaction.txEdition #
      state[txi].txEdition THEN RETURN WITH ERROR InvalidHandle;
    WITH trans: state[txi] SELECT FROM
      active =>
        BEGIN
        IF trans.noMoreOperations THEN RETURN WITH ERROR InvalidHandle;
        sNodePrevP ← @trans.spaceList;
        FOR sNode ← sNodePrevP↑, sNode.nextSpace UNTIL sNode = NIL DO
```

```
        IF sNode.handle = space THEN EXIT;
        sNodePrevP ← @sNode.nextSpace;
        REPEAT FINISHED => ERROR Bug[spaceNotInTransaction];
        ENDLOOP;
      sNodePrevP↑ ← sNode.nextSpace;
      first64K.FREE[@sNode];
      END;
    ENDCASE => RETURN WITH ERROR InvalidHandle;
  END;
---------------------------------------------
-- Monitor Internal Procedures
---------------------------------------------
--TransactionInternal.--

UpdateStateFile: PUBLIC INTERNAL PROCEDURE[] =
  -- Marks all transactions with the stamp of the (incremented) current edition of State, then writ
**es two copies of the state data on the disk. In the event of a crash while writing the two copies, t
**he edition info is used by Crash Recovery to determine which copy is the most-recent and corre
**ct copy.
  BEGIN
  pStateData: LONG POINTER TO StateData = LOOPHOLE[state];
  currentStateEdition: StateEdition ← state[FIRST[ValidTxIndex]].stateEdition;
  -- (any one will do)
  currentStateEdition ←
    IF currentStateEdition < LAST[StateEdition] THEN SUCC[currentStateEdition]
    ELSE FIRST[StateEdition];
  FOR txi: TxIndex IN TxIndex DO
    state[txi].stateEdition ← currentStateEdition; ENDLOOP;
  pStateData.checksumData ← Checksum[0, SIZE[State], @pStateData.state];
  MaybeCrash[3];
  SimpleSpace.CopyOut[stateSpace, state1Window];
  MaybeCrash[2];
  SimpleSpace.CopyOut[stateSpace, state2Window];
  MaybeCrash[4];
  END;
-------------------------------------------------
-- Allocator for SpaceNodes
-------------------------------------------------
-- The procedures here have no particular relation to Monitors; They may be called from inside o
**r outside any monitor.

Procs: TYPE = MACHINE DEPENDENT RECORD [
  Make(0): PROCEDURE [UNCOUNTED ZONE, CARDINAL] RETURNS [LONG POINTER],
  Free(1): PROCEDURE [UNCOUNTED ZONE, LONG POINTER]];
uncountedZoneObject: MACHINE DEPENDENT RECORD [
  procs(0:0..31): LONG POINTER TO Procs,
  data(2:0..31): LONG POINTER --to instance state, if any--
  ] ← [procs: @procs, data: NIL --never used--];
procs: Procs ← [Make: MakeNode, Free: FreeNode];
first64K: UNCOUNTED ZONE ← LOOPHOLE[LONG[@uncountedZoneObject]];
ResidentHeapProblem: ERROR = CODE;
MakeNode: PROCEDURE [z: UNCOUNTED ZONE, n: CARDINAL] RETURNS [LONG POINTER] =
  BEGIN
  r: Environment.Base RELATIVE POINTER;
  status: Zone.Status;
  [node: r, s: status] ← ResidentHeap.MakeNode[n];
  IF status ~= okay THEN ERROR ResidentHeapProblem;
  RETURN[@Environment.first64K[r]]
```



```
  END;

FreeNode: PROCEDURE [z: UNCOUNTED ZONE, p: LONG POINTER] =
  BEGIN
  status: Zone.Status ← ResidentHeap.FreeNode[LOOPHOLE[Inline.LowHalf[p]]];
  IF status ~= okay THEN ERROR ResidentHeapProblem;
  END;

END.
```

LOG

May 8, 1980 1:21 PM	*Gobbel*	*Create file.*
July 19, 1980 1:43 PM	*McJones*	*Space.Copy = >CopyOut. export Transaction.Handle.*
July 22, 1980 4:13 PM	*Gobbel*	*Make it real.*
September 2, 1980 3:50 PM	*Knutsen*	*Major cleanup. Split code out from TransactionImpl.*
September 8, 1980 6:35 PM	*Gobbel*	*Added WithdrawFromTransaction.*
September 25, 1980 12:40 PM	*Gobbel*	*Added SpecialTransaction ops.*

```
-- VMMgr>HierarchyImpl.mesa (last edited by Knutsen on July 30, 1980 4:35 PM)
-- If there is an entry in the space cache, it is the truth. Information in the Hierarchy is 100% valid o
**nly if there is no entry in the cache.

DIRECTORY
  CachedSpace USING [
    Delete, Desc, Get, Handle, handleNull, Insert, Level, PDesc, State, Update],
  Hierarchy,
  STree USING [Delete, Desc, Get, Insert, Key, Update],
  Space USING [defaultWindow],
  Transaction USING [nullHandle],
  VM USING [Interval, PageCount, PageNumber],
  VMMPrograms USING [];

HierarchyImpl: PROGRAM
  -- logically, this should be a MONITOR, but it is protected by the monitor lock of SpaceImpl, whic
**h is the only caller of this program.
  IMPORTS CachedSpace, STree, Transaction EXPORTS Hierarchy, VMMPrograms =
  BEGIN OPEN Hierarchy;
  NotFound: PUBLIC ERROR = CODE;
  pDescBuff: CachedSpace.PDesc
    -- must point to pinned storage because of CachedSpace calls
    --residentFrames-- = @desc;
  desc: CachedSpace.Desc; -- must be in pinned storage (see above).
  LoadDesc: PROCEDURE [handle: CachedSpace.Handle]
    RETURNS [isSpaceHandle, isSwapUnitHandle: BOOLEAN] =
    -- Loads matching space descriptor into global variable pDescBuff↑. handle may be a space h
**andle or a swap unit handle. Returns [FALSE, FALSE] (and pDescBuff↑.state = missing) if not fo
**und. isSpaceHandle and isSwapUnitHandle are mutually exclusive. Handles are not required t
**o fall exactly on the appropriate boundaries.
    BEGIN
    descVictim: CachedSpace.Desc;
    -- Must try the cache first
    IF handle.level <= LAST[CachedSpace.Level] THEN
      -- (don't look for space if this must be a swap unit handle)
      BEGIN
      CachedSpace.Get[pDescBuff, handle];
      -- look for a space handle in cache (must try cache before full Hierarchy).
      IF pDescBuff.state ~= missing THEN RETURN[TRUE, FALSE];
      END;
    CachedSpace.Get[pDescBuff, CachedSpace.Handle[handle.level - 1, handle.page]];
    -- look for a swap unit handle in cache.
    IF pDescBuff.state ~= missing AND pDescBuff.hasSwapUnits THEN
      RETURN[FALSE, TRUE];
    -- Try full Hierarchy second
    [] ← STree.Get[@LOOPHOLE[pDescBuff↑, STree.Desc], [hierarchy[handle]]];
    -- look for a space handle in Hierarchy.
    IF pDescBuff.state ~= missing THEN
      BEGIN isSpaceHandle ← TRUE; isSwapUnitHandle ← FALSE; END
    ELSE
      BEGIN
      [] ← STree.Get[
        @LOOPHOLE[pDescBuff↑, STree.Desc],
        [hierarchy[CachedSpace.Handle[handle.level - 1, handle.page]]]];
      -- look for a swap unit handle in Hierarchy.
      IF pDescBuff.state = missing THEN RETURN[FALSE, FALSE]
      ELSE
        BEGIN
        isSpaceHandle ← FALSE;
```



```
        isSwapUnitHandle ← pDescBuff.hasSwapUnits;
        IF ~isSwapUnitHandle THEN RETURN[FALSE, FALSE]
        -- (don't cache unwanted desc)

        END;
      END;
    -- Cache this (clean) descriptor, possibly displacing a dirty descriptor from the cache
    pDescBuff.dPinned ← FALSE;
    pDescBuff.dDirty ← FALSE;
    CachedSpace.Insert[@descVictim, pDescBuff];
    IF descVictim.dDirty THEN STree.Update[@LOOPHOLE[descVictim, STree.Desc]];
    RETURN[isSpaceHandle, isSwapUnitHandle];
    END;

  Delete: PUBLIC PROCEDURE [handle: CachedSpace.Handle] =
    BEGIN CachedSpace.Delete[handle]; STree.Delete[[hierarchy[handle]]] END;

  FindFirstWithin: PUBLIC PROCEDURE [
    handle: CachedSpace.Handle, count: VM.PageCount]
    RETURNS [CachedSpace.Handle] =
    -- Bypasses cache
    BEGIN
    desc: STree.Desc;
    keyNext: STree.Key = STree.Get[@desc, [hierarchy[handle]]];
    IF count = 0 THEN RETURN[CachedSpace.handleNull]
    ELSE
      IF desc.descH.state ~= missing THEN
        RETURN[[handle.level, desc.descH.interval.page]]
      ELSE
        IF keyNext.handleH.level = handle.level AND keyNext.handleH.page <
          handle.page + count THEN RETURN[keyNext.handleH]
        ELSE RETURN[CachedSpace.handleNull]
    END;

  GetDescriptor: PUBLIC PROCEDURE [
    pDescResult: POINTER TO CachedSpace.Desc, handle: CachedSpace.Handle]
    RETURNS [validSpace, validSwapUnit: BOOLEAN] =
    BEGIN
    [validSpace, validSwapUnit] ← LoadDesc[handle];
    pDescResult↑ ← pDescBuff↑;
    -- return copy of descriptor to caller (state = missing if not found).
    IF validSpace THEN RETURN[handle.page = pDescBuff.interval.page, FALSE]
    ELSE
      IF validSwapUnit THEN
        RETURN[
          FALSE,
          (handle.page - pDescBuff.interval.page) MOD pDescBuff.sizeSwapUnit =
          0]
      ELSE RETURN[validSpace, validSwapUnit];
    -- = [FALSE, FALSE] (but space desc may have been found).

    END;

  GetInterval: PUBLIC PROCEDURE [handle: CachedSpace.Handle]
    RETURNS [validSpace, validSwapUnit: BOOLEAN, interval: VM.Interval] =
    BEGIN
    [validSpace, validSwapUnit] ← LoadDesc[handle];
    IF validSpace THEN
      RETURN[handle.page = pDescBuff.interval.page, FALSE, pDescBuff.interval]
```

```
    ELSE
      IF validSwapUnit THEN
        BEGIN
        remainder: CARDINAL =
          (handle.page - pDescBuff.interval.page) MOD pDescBuff.sizeSwapUnit;
        startPage: VM.PageNumber = handle.page - remainder;
        RETURN[
          FALSE, remainder = 0,
            [startPage, MIN[
              pDescBuff.sizeSwapUnit,
                (pDescBuff.interval.page + pDescBuff.interval.count) -
                  startPage]]];
        END
      ELSE RETURN[validSpace, validSwapUnit, [0, 0]]; -- = [FALSE, FALSE, ...]

    END;

  ValidSpaceOrSwapUnit: PUBLIC PROCEDURE [handle: CachedSpace.Handle]
    RETURNS [BOOLEAN] =
    BEGIN
    validSpaceHandle, validSwapUnitHandle: BOOLEAN;
    [validSpaceHandle, validSwapUnitHandle] ← LoadDesc[handle];
    IF (validSpaceHandle AND handle.page = pDescBuff.interval.page) OR
      (validSwapUnitHandle AND
        (handle.page - pDescBuff.interval.page) MOD pDescBuff.sizeSwapUnit = 0)
      THEN RETURN[TRUE]
    ELSE RETURN[FALSE];
    END;

  Insert: PUBLIC PROCEDURE [handle: CachedSpace.Handle, count: VM.PageCount] =
    BEGIN
    desc: STree.Desc ←
      [hierarchy[
        CachedSpace.Desc[
        interval: [handle.page, count], level: handle.level, dPinned: FALSE,
        -- don't care
        dDirty: FALSE, -- don't care
        pinned: FALSE, state: unmapped, writeProtected: FALSE, -- don't care
        hasSwapUnits: FALSE, sizeSwapUnit: 1, -- don't care
        dataOrFile: data, -- don't care
        pageRover: handle.page,
        vp: long[
        window: Space.defaultWindow, -- don't care
        countMapped: 0, -- don't care
        transaction: Transaction.nullHandle]]]]; -- not part of a transaction yet.
    STree.Insert[@desc];
    END;

  Touch: PUBLIC PROCEDURE [handle: CachedSpace.Handle] =
    BEGIN
    [] ← LoadDesc[handle];
    IF pDescBuff.state = missing THEN ERROR NotFound;
    END;

  Update: PUBLIC PROCEDURE [pDesc: POINTER TO CachedSpace.Desc] =
    BEGIN
    pDescBuff↑ ← pDesc↑;
    IF ~CachedSpace.Update[pDescBuff].found THEN
      STree.Update[@LOOPHOLE[pDescBuff↑, STree.Desc]];
```



```
    END;

  END.
```

LOG

Time: May 23, 1978 9:45 AM By: McJones Action: Created file

Time: June 27, 1978 6:35 PM By: McJones Action: Didn't set dPinned to FALSE before c
***alling CachedSpace.Insert, .Update*

Time: August 29, 1978 6:04 PM By: McJones Action: Clear dDirty before calling CachedS
***pace.Insert*

Time: September 3, 1978 1:25 PM By: McJones Action: FindFirstWithin ignored ke
***yNext.levelH*

Time: September 10, 1978 1:38 PM By: McJones Action: Added pinned to construc
***tor in Insert, etc.*

Time: February 1, 1979 2:00 PM By: McJones CR20.169: Added pageRover to C
***achedSpace.Desc*

Time: February 20, 1979 10:51 AM By: McJones CR20.177: Changed Update to pr
***eserve dPinned (!)*

Time: September 7, 1979 10:24 AM By: McJones Action: FindFirstWithin ignored co
***unt = 0*

Time: October 1, 1979 9:42 AM By: Knutsen Action: Made compatible with new CachedS
***pace.Desc.*

Time: November 8, 1979 11:36 AM By: Knutsen Action: Use Swap Unit Handles as.
***well as Space Handles. Procedures return validSpace and validSwapUnit instead of validHandl*
***e. GetState replaced by ValidSpaceOrSwapUnit.*

Time: November 9, 1979 11:47 AM By: Knutsen Action: Make Touch not require a
***strict handle.*

Time: November 14, 1979 2:08 PM By: Knutsen AR2832: Make LoadDesc not cac
***he an unrequested descriptor.*

Time: July 30, 1980 4:34 PM By: Knutsen Action: Insert must initialize desc with nullTr
***ansaction.*

```
-- VMMgr>MapLogImpl.mesa (last edited by Knutsen on June 3, 1980 12:19 PM)

DIRECTORY
  CachedRegion USING [activate, Apply, deactivate, Outcome],
  CachedSpace USING [Desc],
  Environment USING [wordsPerPage],
  File USING [Capability, PageCount],
  Inline USING [LowHalf],
  KernelFile USING [GetFilePoint],
  MapLog USING [],
  PilotSwitches USING [switches --.m--],
  RuntimeInternal USING [CleanMapLog],
  SimpleSpace USING [Create, Handle, Map, Page],
  Space USING [WindowOrigin],
  Utilities USING [LongPointerFromPage],
  VM USING [Interval, PageCount, PageNumber, PageOffset],
  VMMapLog USING [Descriptor, Entry, EntryBasePointer, EntryPointer],
  VMMgrStore USING [AllocateMapLogFile],
  VMMPrograms USING [];

MapLogImpl: PROGRAM [pMapLogDesc: LONG POINTER]
  -- logically, this should be a monitor, but it is only called from SpaceImpl, and is protected by its
**monitor lock.

  IMPORTS
    CachedRegion, Inline, KernelFile, PilotSwitches, RuntimeInternal, SimpleSpace,
    Utilities, VMMgrStore
  EXPORTS MapLog, VMMPrograms =
  BEGIN OPEN VMMapLog;
  threshold: CARDINAL = Environment.wordsPerPage - 6*SIZE[VMMapLog.Entry];
  -- we will swap in the next map log page whenever we are past this point in the current page.
  maxEntryCount: CARDINAL = 4096; -- should be in VMMapLog!
  EntryCount: TYPE = CARDINAL [1..maxEntryCount]; -- should be in VMMapLog!
  Bug: PRIVATE --PROGRAMMING--ERROR [type: BugType] = CODE;
  BugType: TYPE = {bug0, bug1, bug2, bug3, bug4};
  pageLog: VM.PageNumber;
  countLog: VM.PageNumber;
  bLog: VMMapLog.EntryBasePointer;
  mapLogging: BOOLEAN = PilotSwitches.switches.m = up;
  WriteLog: PUBLIC PROCEDURE [
    interval: VM.Interval, pSpaceD: POINTER TO CachedSpace.Desc] =
    -- Writes one or more log entries; If pSpaceD = NIL then interval is now unmapped.
    BEGIN
    IF mapLogging THEN
      BEGIN OPEN LOOPHOLE[pMapLogDesc, LONG POINTER TO Descriptor];
      writerNext: EntryPointer;
      pEntry: LONG POINTER TO Entry;
      count: VM.PageCount;
      offsetWriter, offsetWriterNext: VM.PageOffset;
      fileOffset: File.PageCount ← 0;
      offsetWriter ← LOOPHOLE[writer, CARDINAL]/Environment.wordsPerPage;
      WHILE interval.count > 0 DO
        -- write another log entry--
        IF (writerNext ← writer + SIZE[Entry]) = limit THEN
          writerNext ← FIRST[EntryPointer];
        offsetWriterNext ←
          LOOPHOLE[writerNext, CARDINAL]/Environment.wordsPerPage;
        WHILE writerNext = reader DO RuntimeInternal.CleanMapLog[] ENDLOOP;
```



```
        pEntry ← @bLog[writer];
        count ← MIN[interval.count, maxEntryCount];
        IF pSpaceD = NIL THEN
          pEntry↑ ←
            [page: interval.page, count: count, writeProtected:, fill:,
              filePoint: nil[]]
        ELSE
          BEGIN
          KernelFile.GetFilePoint[
            pEntry, @pSpaceD.window.file, pSpaceD.window.base + fileOffset];
          pEntry.page ← interval.page;
          count ← pEntry.count ← MIN[pEntry.count, count];
          pEntry.writeProtected ← pSpaceD.writeProtected;
          fileOffset ← fileOffset + count;
          END;
        IF offsetWriterNext ~= offsetWriter THEN
          --swap out the page containing writer--
          IF CachedRegion.Apply[
            pageLog + offsetWriter, CachedRegion.deactivate].outcome ~= [ok[]]
            THEN ERROR Bug[bug0];
        writer ← writerNext;
        offsetWriter ← offsetWriterNext;
        interval ← [interval.page + count, interval.count - count];
        ENDLOOP;
      IF LOOPHOLE[writer, CARDINAL] MOD Environment.wordsPerPage > threshold THEN
        --start the next page needed coming in--
        IF CachedRegion.Apply[
          pageLog + (offsetWriter + 1) MOD countLog,
          CachedRegion.activate].outcome ~= [ok[]] THEN ERROR Bug[bug1];
      END;
    END;

  Initialize: PROCEDURE =
    BEGIN OPEN pM: LOOPHOLE[pMapLogDesc, LONG POINTER TO VMMapLog.Descriptor];
    handle: SimpleSpace.Handle;
    window: Space.WindowOrigin;
    -- We make sure the backing file exists even if we aren't logging this time. This will (tend to) en
**sure that the file is contiguous in case it is used at a later time.
    countLog ← Inline.LowHalf[
      VMMgrStore.AllocateMapLogFile[pWindowResult: @window].count];
    IF countLog ~IN [2..maxEntryCount] THEN ERROR Bug[bug2];
    KernelFile.GetFilePoint[@pM.self, @window.file, window.base];
    IF pM.self.count < countLog THEN ERROR Bug[bug3];
    -- log window not contiguous.
    IF mapLogging THEN
      BEGIN
      handle ← SimpleSpace.Create[
        count: countLog, location: hyperspace, sizeSwapUnit: 1];
      SimpleSpace.Map[handle: handle, window: window, andPin: FALSE];
      pageLog ← SimpleSpace.Page[handle];
      pM.self.page ← pageLog;
      pM.self.count ← countLog;
      pM.writer ← pM.reader ← FIRST[EntryPointer];
      pM.limit ←
        LOOPHOLE[countLog*Environment.wordsPerPage/SIZE[Entry]*SIZE[Entry],
        EntryPointer];
      bLog ← LOOPHOLE[Utilities.LongPointerFromPage[pM.self.page]];
      IF CachedRegion.Apply[pageLog + 0, CachedRegion.activate].outcome ~= [ok[]]
```

```
        THEN ERROR Bug[bug4]; -- start the page containing writer coming in.

      END;
    END;

  Initialize[];
  END.
```

LOG

Time: August 1, 1978 10:11 AM By: McJones Action: Create file
Time: August 7, 1978 4:51 PM By: McJones Action: pDesc.self.page wasn't initialized
Time: August 8, 1978 9:10 AM By: McJones Action: WriteLog didn't set entry page field i
**n case of non-nil pWindow
Time: August 8, 1978 3:25 PM By: McJones Action: limit initialization didn't convert page
**s to words
Time: August 29, 1978 4:44 PM By: McJones Action: Add VMMode
Time: September 5, 1978 6:48 PM By: McJones Action: Replace signal with Clean
**MapLog[], GetFilePoint moved to SpecialFile
Time: September 15, 1978 4:47 PM By: McJones Action: Getting ready for "unifor
**m" swap unit management
Time: September 29, 1978 11:05 AM By: McJones CR20.42: Replace PutRootFile wit
**h MakePermanent
Time: July 31, 1979 1:12 PM By: McJones Action: Prepared to add writeProtected to m
**ap log entry
Time: August 16, 1979 8:56 PM By: McJones Action: Add writeProtected to map log entry;
**SpecialFile => KernelFile; VMMode => PilotSwitches
Time: September 4, 1979 10:00 AM By: Forrest Action: Change to use PilotFileTy
**pes
Time: November 7, 1979 2:39 PM By: McJones AR2744: Create backing file even
**if map log disabled
Time: November 21, 1979 9:18 AM By: Knutsen Action: Use backing file now provi
**ded by STLeafImpl.
Time: June 3, 1980 12:19 PM By: Knutsen Action: Use Uniform Swap Units. Activate/d
**eactivate as appropriate. Named the errors.

```
-- VMMgr>ProjectionImpl.mesa (last edited by Knutsen on September 16, 1980 9:48 AM)
-- The routines in this module are only called from within the SpaceImpl monitor. Thus, the databa
**se accessed by this module need not be protected by a local monitor because it is already prote
**cted by the SpaceImpl monitor.

DIRECTORY
  CachedRegion USING [Apply, Desc, Insert, State, Outcome],
  CachedSpace USING [Level],
  Projection USING [],
  STLeaf USING [alive, ProjectionState],
  STree USING [Delete, Desc, Get, Insert, Key, PDesc, Update],
  VM USING [PageNumber, PageCount],
  VMMPrograms USING [];

ProjectionImpl: PROGRAM [countVM: VM.PageCount] -- (not a monitor)
  IMPORTS CachedRegion, STree EXPORTS Projection, VMMPrograms =
BEGIN
  Bug: ERROR [type: BugType] = CODE; -- not to be caught by client.
  BugType: TYPE = {funnyLevel, funnyOutcome, mergeAtZero};
  DeleteSwapUnits: PUBLIC PROCEDURE [pageMember: VM.PageNumber] =
    BEGIN
    projDesc: projection STree.Desc;
    [] ← STree.Get[
      LOOPHOLE[LONG[@projDesc], STree.PDesc], [projection[pageMember]]];
    IF projDesc.hasSwapUnits THEN {
      projDesc.hasSwapUnits ← FALSE;
      STree.Update[LOOPHOLE[LONG[@projDesc], STree.PDesc]]};
    END;

  ForceOut: PUBLIC PROCEDURE [pageMember: VM.PageNumber] =
    BEGIN
    desc: CachedRegion.Desc;
    outcome: CachedRegion.Outcome = CachedRegion.Apply[
      pageMember,
      [ifMissing: report, ifCheckedOut: wait, afterForking:,
        vp: get[andResetDDirty: TRUE, pDescResult: @desc]]].outcome;
    WITH outcome SELECT FROM
      ok =>
        IF desc.dDirty THEN
          BEGIN OPEN desc;
          projDesc: projection STree.Desc ←
            [projection[
              -- transform a CachedRegion.Desc into an projection STLeaf.Desc:
              page: interval.page, level: level,
              state:
              IF state IN STLeaf.ProjectionState THEN state
              ELSE --state IN CachedRegion.Alive--STLeaf.alive,
              -- (in a projection desc, we only know that the region is some flavor of alive.)
              levelMapped: levelMapped, writeProtected: writeProtected,
              hasSwapUnits: hasSwapUnits, needsLogging: needsLogging]];
          STree.Update[LOOPHOLE[LONG[@projDesc], STree.PDesc]]
          END;
      regionDMissing => NULL;
      ENDCASE => ERROR Bug[funnyOutcome];
    END;

  Get: PUBLIC PROCEDURE [pageMember: VM.PageNumber]
    RETURNS [desc: CachedRegion.Desc] =
```



```
    BEGIN
    projDesc: projection STree.Desc;
    keyNext: STree.Key;
    descVictim: CachedRegion.Desc;
    outcome: CachedRegion.Outcome = CachedRegion.Apply[
      pageMember,
      [ifMissing: report, ifCheckedOut: wait, afterForking:,
        vp: get[andResetDDirty: FALSE, pDescResult: @desc]]].outcome;
    WITH outcome SELECT FROM
      ok => NULL; -- descriptor is in the cache, just return .it

      regionDMissing =>
        -- Otherwise, fetch descriptor from hierarchy and cache it..
        BEGIN
        keyNext ← STree.Get[
          LOOPHOLE[LONG[@projDesc], STree.PDesc], [projection[pageMember]]];
        BEGIN OPEN projDesc;
        desc ←
          [ -- transform an projection STLeaf.Desc into a CachedRegion.Desc:
            interval: [page, keyNext.pageP - page], level: level, dPinned: FALSE,
            dTemperature:, -- don't care
            dDirty: FALSE, state: state, -- i.e. outAlive.
            levelMapped: levelMapped, beingFlushed: FALSE,
            writeProtected: writeProtected, hasSwapUnits: hasSwapUnits,
            needsLogging: needsLogging];
        END;
        descVictim ← CachedRegion.Insert[desc];
        IF descVictim.dDirty THEN
          BEGIN OPEN descVictim;
          projDesc ←
            [projection[
              -- transform a CachedRegion.Desc into an projection STLeaf.Desc:
              page: interval.page, level: level,
              state:
              IF state IN STLeaf.ProjectionState THEN state
              ELSE --state IN CachedRegion.Alive--STLeaf.alive,
              -- (in a projection desc, we only know that the region is some flavor of alive.)
              levelMapped: levelMapped, writeProtected: writeProtected,
              hasSwapUnits: hasSwapUnits, needsLogging: needsLogging]];
          STree.Update[LOOPHOLE[LONG[@projDesc], STree.PDesc]]
          END
        END;
      ENDCASE => ERROR Bug[funnyOutcome];
    END;

  Merge: PUBLIC PROCEDURE [pageNext: VM.PageNumber] =
    BEGIN
    projDesc: projection STree.Desc;
    IF pageNext = 0 THEN ERROR Bug[mergeAtZero];
    -- Unconditionally "resurrect" a dead region: conservative approach to avoid propagating dea
**dness
    [] ← STree.Get[
      LOOPHOLE[LONG[@projDesc], STree.PDesc], [projection[pageNext - 1]]];
    IF projDesc.state = outDead THEN {
      projDesc.state ← outAlive;
      STree.Update[LOOPHOLE[LONG[@projDesc], STree.PDesc]]};
    -- Do the merge: (depends on the fact that the size of a projection STree.Desc is implicit)
    STree.Delete[[projection[pageNext]]];
```

```
  END;

Split: PUBLIC PROCEDURE [pageNext: VM.PageNumber] =
  BEGIN
  projDesc: projection STree.Desc;
  IF pageNext ~= countVM THEN
    BEGIN
    -- Depends on the fact that the size of a projection STree.Desc is implicit
    [] ← STree.Get[
      LOOPHOLE[LONG[@projDesc], STree.PDesc], [projection[pageNext]]];
    IF projDesc.page ~= pageNext THEN {
      projDesc.page ← pageNext;
      STree.Insert[LOOPHOLE[LONG[@projDesc], STree.PDesc]]};
    END;
  END;

Touch: PUBLIC PROCEDURE [pageMember: VM.PageNumber] = {[] ← Get[pageMember]};

TranslateLevel: PUBLIC PROCEDURE [pageMember: VM.PageNumber, delta: INTEGER] =
  BEGIN
  projDesc: projection STree.Desc;
  [] ← STree.Get[
    LOOPHOLE[LONG[@projDesc], STree.PDesc], [projection[pageMember]]];
  IF projDesc.level + delta ~IN CachedSpace.Level THEN ERROR Bug[funnyLevel];
  projDesc.level ← projDesc.level + delta;
  STree.Update[LOOPHOLE[LONG[@projDesc], STree.PDesc]]
  END;

END.
```

LOG

May 24, 1978 10:13 AM	*McJones*	*Created file.*
June 23, 1978 10:57 AM	*McJones*	*Split didn't check for pageNext = countVM.*
August 2, 1978 9:27 AM	*McJones*	*Updated to new CachedRegion interface.*
August 4, 1978 2:19 PM	*McJones*	*Added beingRemapped.*
August 29, 1978 7:32 PM	*McJones*	*Cleared dDirty before calling CachedRegion.Insert.*
August 21, 1979 4:33 PM	*Knutsen*	*Made compatible with new CachedRegion.Desc.*
September 28, 1979 3:57 PM	*Knutsen*	*Made compatible with new CachedRegion.Desc.*
February 25, 1980 6:01 PM	*Knutsen*	*Added DeleteSwapUnits. Named the ERRORs.*
June 16, 1980 4:57 PM	*Gobbel*	*Made compatible with new CachedRegion.Desc.*
July 30, 1980 2:52 PM	*Knutsen*	*Made compatible with new CachedRegion.Desc.*
September 16, 1980 9:48 AM	*Knutsen*	*DeleteSwapUnits failed if no existing swap units.*

```
-- VMMgr>SpaceImplA.mesa (last edited by Fay on October 10, 1980 9:09 PM)
-- Note: There is another version of SpaceImpl for UtilityPilot which should track changes to this m
**odule.
-- Implementation Notes:
-- The routines herein must have this organization: first check for all client errors and RETURN
**WITH ERROR if any; then, make changes to the VM databases, etc.
-- The MONITORLOCK of SpaceImplA/B protects the Hierarchy and Projection data base and the
**MapLog. That is, those facilities are only accessed from within the Space monitor, and so are n
**ot themselves MONITORs, but rely on the serialization provided by the SpaceImpl MONITORLO
**CK. This makes the calls to those facilities faster.
-- Things to Consider:
-- GetAttributes could be augmented by operations for accessing individual space attributes, avoi
**ding unnecessary disk accesses, if performance requirements so dictate.

DIRECTORY
  CachedRegion USING [
    activate, Apply, createSwapUnits, deactivate, Desc, flush, startForceOut,
    kill, Mapped, Operation, Outcome, pageTop, pin, unpin, wait],
  CachedSpace USING [Desc, Handle, handleNull, handleVM, Level, SizeSwapUnit],
  Environment USING [Long, maxPagesInMDS, wordsPerPage],
  File USING [Capability, Permissions, read, write],
  Hierarchy USING [
    Delete, FindFirstWithin, GetDescriptor, GetInterval, Insert, NotFound, Touch,
    Update, ValidSpaceOrSwapUnit],
  Inline USING [BITAND, LongMult],
  PrincOps USING [ControlLink, PrefixHandle],
  PrincOpsRuntime USING [GFT],
  Process USING [Detach],
  Projection USING [
    DeleteSwapUnits, ForceOut, Get, Merge, Split, Touch, TranslateLevel],
  Runtime USING [CallDebugger, GlobalFrame],
  RuntimeInternal USING [Codebase],
  Space,
  SpaceImplInternal USING [
    InitializeSpaceImplB, Interval, Level, RegionD, SpaceD, spaceLock,
    UnmapInternal],
  SpecialSpace USING [],
  SwapperException USING [Await],
  Transaction USING [Handle, InvalidHandle, nullHandle],
  TransactionState USING [WithdrawFromTransaction],
  VM USING [Interval, PageCount],
  VMMPrograms USING [];

SpaceImplA: MONITOR LOCKS SpaceImplInternal.spaceLock
  IMPORTS
    CachedRegion, Hierarchy, Inline, Process, Projection, Runtime,
    RuntimeInternal, SpaceImplInternal, SwapperException, Transaction,
    TransactionState
  EXPORTS Space, SpaceImplInternal, SpecialSpace, VMMPrograms
  SHARES File --USING [Capability.permissions]-- =
  BEGIN OPEN Space, SpaceImplInternal;
  Handle: PUBLIC TYPE = CachedSpace.Handle;
  Alignment: TYPE = {code, powerOf2, none};
  ChildrenSelf: TYPE = {children, self};
  ----------------------------------------------------------------------
  -- Space implementation data:
  ----------------------------------------------------------------------
  -- Public Data:
```



```
  Error: PUBLIC ERROR [type: ErrorType] = CODE;
  InsufficientSpace: PUBLIC ERROR [available: PageCount] = CODE;
  mds: PUBLIC Handle;
  nullHandle: PUBLIC Handle ← LOOPHOLE[CachedSpace.handleNull];
  nullTransactionHandle: PUBLIC Transaction.Handle ← Transaction.nullHandle;
  virtualMemory: PUBLIC Handle ← LOOPHOLE[CachedSpace.handleVM];
  -- Private Data:
  Bug: PRIVATE --PROGRAMMING--ERROR [type: BugType] = CODE;
  -- not to be caught by client;
  BugType: TYPE = {bug0, bug1, bug2, bug3, bug4};
  countVM: VM.PageCount;
  handleMDS: CachedSpace.Handle;
  intervalMDS: Interval;
  maxPagesInCodeBlock: PageCount = Environment.maxPagesInMDS;
  codeBdyMask: WORD = LOOPHOLE[-(maxPagesInCodeBlock - 1) - 1, WORD];
  -- = BITNOT[maxPagesInCodeBlock-1]
  --SpaceImplInternal.--
  spaceLock: PUBLIC MONITORLOCK; -- (private to SpaceImpl#)
  ----------------------
  -- Initialization
  ----------------------
  InitializeSpace: PUBLIC PROCEDURE [
    countVM: VM.PageCount, handleMDS: CachedSpace.Handle] = {
    InitializeInternal[countVM, handleMDS]};
  -- (just to change the parameter names.)

  InitializeInternal: PROCEDURE [
    ctVM: VM.PageCount, handMDS: CachedSpace.Handle] =
    BEGIN
    countVM ← ctVM; -- copy params into globals..
    mds ← LOOPHOLE[handleMDS ← handMDS];
    intervalMDS ← Hierarchy.GetInterval[handleMDS].interval;
    InitializeSpaceImplB[];
    Process.Detach[FORK VMMHelperProcess[]];
    END;
  ----------------------------------------------------------------------
  -- SpecialSpace implementation
  ----------------------------------------------------------------------
  ----------------------
  -- Monitor externals
  ----------------------

  CreateAligned: PUBLIC --EXTERNAL--PROCEDURE [
    size: Space.PageCount, parent: Space.Handle]
    RETURNS [newSpace: Space.Handle] =
    -- Create a space which begins at a page within virtual memory which is an integral multiple of
**the smallest power of two not less than the given size. (Thus if size is already a power of two, th
**e beginning page number will have at least as many low-order zero bits as does size.)
    BEGIN
    {
    RETURN[CreateInternal[size, parent, defaultBase, --alignment:--powerOf2]]};
    END;

  CreateForCode: PUBLIC --EXTERNAL--PROCEDURE [
    size: PageCount, parent: Handle, base: PageOffset]
    RETURNS [newSpace: Space.Handle] =
    -- Creates a space guaranteed not to cross a 64KW boundary.
    {RETURN[CreateInternal[size, parent, base, --alignment:--code]]};
```

```
  MakeCodeResident: PUBLIC --EXTERNAL--PROCEDURE [frame: PROGRAM] =
    LOOPHOLE[MakeProcedureResident];
  MakeCodeSwappable: PUBLIC --EXTERNAL--PROCEDURE [frame: PROGRAM] =
    LOOPHOLE[MakeProcedureSwappable];
  MakeGlobalFrameResident: PUBLIC --EXTERNAL--PROCEDURE [frame: PROGRAM] = {
    MakeResident[GetHandle[PageFromLongPointer[LOOPHOLE[frame, POINTER]]]]};

  MakeGlobalFrameSwappable: PUBLIC --EXTERNAL--PROCEDURE [frame: PROGRAM] = {
    MakeSwappable[GetHandle[PageFromLongPointer[LOOPHOLE[frame, POINTER]]]]};

  MakeProcedureResident: PUBLIC --EXTERNAL--PROCEDURE [
    proc: --procedure PrincOps.ControlLink--UNSPECIFIED] = {
    MakeResident[SpaceForCode[proc]]};

  MakeProcedureSwappable: PUBLIC --EXTERNAL--PROCEDURE [
    proc: --procedure PrincOps.ControlLink--UNSPECIFIED] = {
    MakeSwappable[SpaceForCode[proc]]};

  SpaceForCode: --EXTERNAL--PROCEDURE [link: --PrincOps.ControlLink--UNSPECIFIED]
    RETURNS [space: Handle] =
    -- link must be either a procedure ControlLink or a PROGRAM (a GlobalFrameHandle).
    BEGIN OPEN lk: LOOPHOLE[link, PrincOps.ControlLink];
    codeBase: LONG PrincOps.PrefixHandle = RuntimeInternal.Codebase[
      Runtime.GlobalFrame[link]];
    offset: CARDINAL;
    --UNTIL non-indirect link found--
    DO
      IF lk.proc THEN {
        offset ← LOOPHOLE[codeBase.entry[
          lk.ep + PrincOpsRuntime.GFT[lk.gfi].epbias].initialpc];
        EXIT}
      ELSE
        IF lk.indirect THEN {lk ← lk.link↑; LOOP}
        ELSE --global frame handle--{offset ← 0; EXIT};
      ENDLOOP;
    RETURN[GetHandle[PageFromLongPointer[codeBase + offset]]];
    END;
  ----------------------
  -- Monitor entries
  ----------------------

  MakeResident: PUBLIC ENTRY PROCEDURE [space: Handle] =
    -- OK to make an already-resident space resident.
    BEGIN
    Pinnable: PROCEDURE [regionD: RegionD] RETURNS [BOOLEAN] = {
      RETURN[
        regionD.state IN CachedRegion.Mapped AND regionD.levelMapped <=
          space.level]};
    spaceD: SpaceD;
    validSpace, validSwapUnit: BOOLEAN;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[@spaceD, space];
    IF validSwapUnit THEN RETURN WITH ERROR Error[notApplicableToSwapUnit]
    ELSE IF ~validSpace THEN RETURN WITH ERROR Error[invalidHandle];
    IF ~ForAllRegions[spaceD.interval, Pinnable] THEN
      RETURN WITH ERROR Error[invalidMappingOperation];
    spaceD.pinned ← TRUE;
    Hierarchy.Update[@spaceD];
    ApplyToInterval[spaceD.interval, CachedRegion.pin]
```



```
    END;

  MakeSwappable: PUBLIC ENTRY PROCEDURE [space: Handle] =
    BEGIN
    spaceD: SpaceD;
    validSpace, validSwapUnit: BOOLEAN;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[@spaceD, space];
    IF validSwapUnit THEN RETURN WITH ERROR Error[notApplicableToSwapUnit]
    ELSE IF ~validSpace THEN RETURN WITH ERROR Error[invalidHandle];
    IF ~spaceD.pinned THEN RETURN WITH ERROR Error[invalidMappingOperation];
    spaceD.pinned ← FALSE;
    Hierarchy.Update[@spaceD];
    ApplyToInterval[spaceD.interval, CachedRegion.unpin]
    END;
  ----------------------------------------------------------------------
  -- Space implementation:
  ----------------------------------------------------------------------
  ----------------------
  -- Monitor externals
  ----------------------

  Create: PUBLIC --EXTERNAL--PROCEDURE [
    size: PageCount, parent: Handle, base: PageOffset] RETURNS [Handle] = {
    RETURN[CreateInternal[size, parent, base, --alignment:--none]]};

  Delete: PUBLIC --EXTERNAL--PROCEDURE [space: Handle] = {
    DeleteInternal[space, self]};

  DeleteSwapUnits: PUBLIC --EXTERNAL--PROCEDURE [space: Handle] = {
    DeleteInternal[space, children]};

  LongPointerFromPage: PUBLIC --EXTERNAL--PROCEDURE [page: PageNumber]
    RETURNS [LONG POINTER] = {
    RETURN[LOOPHOLE[Inline.LongMult[page, Environment.wordsPerPage]]]};

  PageFromLongPointer: PUBLIC --EXTERNAL--PROCEDURE [lp: LONG POINTER]
    RETURNS [page: PageNumber] = {
    OPEN LOOPHOLE[lp, num Environment.Long];
    IF lp = NIL THEN ERROR Error[invalidParameters]
    ELSE RETURN[highbits*256 + lowbits/256]};
  ----------------------
  -- Monitor entries:
  ----------------------

  Activate: PUBLIC ENTRY PROCEDURE [space: Handle] = {
    IF ~ApplyToSpace[space, CachedRegion.activate].validHandle THEN
      RETURN WITH ERROR Error[invalidHandle]};

  CreateInternal: ENTRY PROCEDURE [
    size: PageCount, parent: Handle, base: PageOffset, alignment: Alignment]
    RETURNS [Handle] =
    BEGIN OPEN handleParent: LOOPHOLE[parent, CachedSpace.Handle];
    largestPowerOf2Cardinal: CARDINAL = (LAST[CARDINAL]/2) + 1;
    -- assumes binary representation for CARDINALs.
    validSpace, validSwapUnit: BOOLEAN;
    handleNew: CachedSpace.Handle;
    page: PageNumber; -- the start of the new space.
    regionD: RegionD;
```

```
  spaceDParent: SpaceD;
  [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[
    @spaceDParent, handleParent];
  IF validSwapUnit THEN RETURN WITH ERROR Error[notApplicableToSwapUnit]
  ELSE IF ~validSpace THEN RETURN WITH ERROR Error[invalidHandle];
  IF handleParent.level >= LAST[Level] THEN
    RETURN WITH ERROR Error[spaceTreeTooDeep];
  IF spaceDParent.hasSwapUnits THEN
    RETURN WITH ERROR Error[notApplicableToSwapUnit];
  IF size = 0 OR (base ~= defaultBase AND base >= spaceDParent.interval.count)
    OR (alignment = powerOf2 AND size > largestPowerOf2Cardinal) THEN
    RETURN WITH ERROR Error[invalidParameters];
  IF base ~= defaultBase THEN --explicit base given--
    BEGIN
    regionD ← Projection.Get[page ← handleParent.page + base];
    -- so page IN regionD.interval
    IF size > (regionD.interval.page + regionD.interval.count) - page OR
      regionD.level ~= handleParent.level OR
      (alignment = code AND page/maxPagesInCodeBlock ~=
        (page + size - 1)/maxPagesInCodeBlock) THEN
      RETURN WITH ERROR Error[invalidParameters];
    END
  ELSE --create anywhere--
    BEGIN OPEN spaceDParent; -- USING [interval, pageRover]
    countMax: PageCount ← 0; -- largest suitable interval in space.
    HoleNotFound: INTERNAL PROCEDURE [regionD: RegionD]
      RETURNS [notFound: BOOLEAN] =
      -- Looks for a suitable hole of a suitable size. Sets page, countmax.
      BEGIN
      codeInterval: VM.Interval;
      IF regionD.level = handleParent.level THEN -- (can not have swap units)
        BEGIN
        SELECT alignment FROM
          code => {
            [codeInterval] ← MaxIntervalForCode[regionD];
            IF codeInterval.count > countMax THEN {
              page ← codeInterval.page; countMax ← codeInterval.count}};
          powerOf2 =>
            BEGIN
            pwrOfTwo: CARDINAL;
            FOR pwrOfTwo ← 1, pwrOfTwo*2 UNTIL pwrOfTwo >= size DO ENDLOOP;
            ERROR; -- UNIMPLEMENTED

            END;
          none =>
            IF regionD.interval.count > countMax THEN {
              page ← regionD.interval.page; countMax ← regionD.interval.count};
          ENDCASE => ERROR Bug[bug0];
        IF countMax >= size THEN RETURN[FALSE];
        --the current region is big enough--
        END;
      RETURN[TRUE]; -- "keep going"

      END;
    IF ForAllRegions[
      [pageRover, interval.count - (pageRover - interval.page)], HoleNotFound]
      THEN
      IF ForAllRegions[[interval.page, pageRover - interval.page], HoleNotFound]
        THEN RETURN WITH ERROR InsufficientSpace[available: countMax];
```



```
      pageRover ← page + size; -- pageRover = interval.page + interval.count is ok
      Hierarchy.Update[@spaceDParent]; -- (update pageRover.)

      END;
    regionD ← Projection.Get[page]; -- so page IN regionD.interval
    ApplyToInterval[regionD.interval, CachedRegion.flush];
    -- assure that the Projection is current (and disable client refs while changing).
    handleNew ← CachedSpace.Handle[1 + handleParent.level, page];
    Hierarchy.Insert[handleNew, size];
    Projection.Split[page];
    Projection.Split[page + size];
    Projection.TranslateLevel[pageMember: page, delta: 1];
    RETURN[LOOPHOLE[handleNew]]
    END;

  CreateUniformSwapUnits: PUBLIC ENTRY PROCEDURE [
    size: PageCount, parent: Handle] =
    BEGIN OPEN handleParent: LOOPHOLE[parent, CachedSpace.Handle];
    spaceDParent: SpaceD;
    validSpace, validSwapUnit: BOOLEAN;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[
      @spaceDParent, handleParent];
    IF validSwapUnit THEN RETURN WITH ERROR Error[notApplicableToSwapUnit]
    ELSE IF ~validSpace THEN RETURN WITH ERROR Error[invalidHandle];
    IF handleParent.level < LAST[Level] THEN
      IF Hierarchy.FindFirstWithin[
        CachedSpace.Handle[handleParent.level + 1, handleParent.page],
        spaceDParent.interval.count] ~= CachedSpace.handleNull THEN
        RETURN WITH ERROR Error[spaceNotUnitary]; -- no children allowed.
    IF spaceDParent.hasSwapUnits THEN RETURN WITH ERROR Error[spaceNotUnitary];
    IF size = 0 OR size ~IN CachedSpace.SizeSwapUnit THEN
      RETURN WITH ERROR Error[invalidParameters];
    spaceDParent.hasSwapUnits ← TRUE;
    spaceDParent.sizeSwapUnit ← size;
    Hierarchy.Update[@spaceDParent];
    ApplyToInterval[spaceDParent.interval, CachedRegion.createSwapUnits];
    END;

  Deactivate: PUBLIC ENTRY PROCEDURE [space: Handle] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    IF ~ApplyToSpace[handle, CachedRegion.deactivate].validHandle THEN
      RETURN WITH ERROR Error[invalidHandle];
    END;

  DeleteInternal: ENTRY PROCEDURE [space: Handle, which: ChildrenSelf] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    level, topLevel: Level;
    DeleteIfLeaf: INTERNAL PROCEDURE [regionD: RegionD] RETURNS [BOOLEAN] =
      -- Implicit parameter: level.
      -- Assume level>0 (i.e. space~= virtualMemory)
      BEGIN
      IF regionD.level = level THEN
        -- Region described by regionD corresponds to a leaf space.
        BEGIN
        leafSpaceD: CachedSpace.Desc;
        mergeLeft, mergeRight: BOOLEAN;
        intervalParent: Interval = Hierarchy.GetInterval[
          Handle[level - 1, regionD.interval.page]].interval;
```

```
    pageNext: PageNumber = regionD.interval.page + regionD.interval.count;
    flushInterval: Interval ← regionD.interval; -- assume no merging
    IF ~Hierarchy.GetDescriptor[
      @leafSpaceD, Handle[level, regionD.interval.page]].validSpace THEN
      ERROR Bug[bug1];
    IF leafSpaceD.transaction ~= nullTransactionHandle THEN
      TransactionState.WithdrawFromTransaction[
        leafSpaceD.transaction, Handle[level, regionD.interval.page] !
        Transaction.InvalidHandle => CONTINUE];
    IF regionD.interval.page > FIRST[PageNumber] THEN
      BEGIN
      leftNeighbor: RegionD = Projection.Get[regionD.interval.page - 1];
      mergeLeft ←
        (regionD.interval.page ~= intervalParent.page AND leftNeighbor.level <
          regionD.level);
      IF mergeLeft THEN
        flushInterval ←
          [flushInterval.page - leftNeighbor.interval.count,
            flushInterval.count + leftNeighbor.interval.count];
      END
    ELSE mergeLeft ← FALSE; -- no left neighbor.
    IF pageNext < CachedRegion.pageTop THEN
      BEGIN
      rightNeighbor: RegionD = Projection.Get[pageNext];
      mergeRight ←
        (pageNext ~= intervalParent.page + intervalParent.count AND
          rightNeighbor.level < regionD.level);
      IF mergeRight THEN
        flushInterval.count ←
          flushInterval.count + rightNeighbor.interval.count;
      END
    ELSE mergeRight ← FALSE; -- no right neighbor.
    IF regionD.state IN CachedRegion.Mapped AND regionD.levelMapped =
      regionD.level THEN
      BEGIN
      spaceD: SpaceD;
      [] ← Hierarchy.GetDescriptor[
        @spaceD, CachedSpace.Handle[regionD.level, regionD.interval.page]];
      UnmapInternal[@spaceD];
      END;
    ApplyToInterval[flushInterval, CachedRegion.flush];
    -- make sure Projection is current (and prevent client refs while changing). (Flush this regi
**on and any required neighbors.)
    IF regionD.hasSwapUnits THEN
      Projection.DeleteSwapUnits[regionD.interval.page];
    Hierarchy.Delete[
      CachedSpace.Handle[regionD.level, regionD.interval.page]];
    -- At this point, the properties of the current region must match the properties of any adjace
**nt regions with which it will coalesce.
    Projection.TranslateLevel[pageMember: regionD.interval.page, delta: -1];
    IF mergeLeft THEN Projection.Merge[regionD.interval.page];
    IF mergeRight THEN Projection.Merge[pageNext];
    END;
  RETURN[TRUE]; -- "keep going"

  END;
levelLeaves: Level ← handle.level;
MaxLevel: PROCEDURE [regionD: RegionD] RETURNS [BOOLEAN] = {
  levelLeaves ← MAX[levelLeaves, regionD.level]; RETURN[TRUE]};
```



```
  -- "keep going"
  validSpace, validSwapUnit: BOOLEAN;
  spaceD: CachedSpace.Desc;
  interval: Interval;
  [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[@spaceD, handle];
  IF validSwapUnit THEN RETURN WITH ERROR Error[notApplicableToSwapUnit]
  ELSE IF ~validSpace THEN RETURN WITH ERROR Error[invalidHandle];
  interval ← spaceD.interval;
  IF handle.level <= handleMDS.level AND handleMDS.page IN
    [interval.page..interval.page + interval.count) THEN
    RETURN WITH ERROR Error[invalidParameters]; -- don't delete the MDS!
  [] ← ForAllRegions[interval, MaxLevel];
  -- levelLeaves ← max region level, region in interval
  topLevel ←
    IF which = children THEN handle.level + 1 --delete only children--
    ELSE handle.level --delete children and self--
  ;
  FOR level DECREASING IN [topLevel..levelLeaves] DO
    [] ← ForAllRegions[interval, DeleteIfLeaf]; ENDLOOP;
  IF which = children THEN -- delete uniform swap units of self:
    BEGIN
    ApplyToInterval[interval, CachedRegion.flush];
    Projection.DeleteSwapUnits[interval.page];
    spaceD.hasSwapUnits ← FALSE;
    Hierarchy.Update[@spaceD];
    END;
  END;

MaxIntervalForCode: INTERNAL PROCEDURE [regionD: RegionD] RETURNS [Interval] =
  BEGIN OPEN regionD.interval; -- page, count
  endRegion: PageNumber;
  firstBdy: PageNumber = Inline.BITAND[page + maxPagesInCodeBlock, codeBdyMask];
  --overflow is ok, yields 0--
  IF (endRegion ← page + count) <= firstBdy OR firstBdy = 0 THEN
    RETURN[[page, count]] -- region ends before first boundary

  ELSE
    IF Inline.BITAND[endRegion, codeBdyMask] ~= firstBdy THEN
      RETURN[[firstBdy, maxPagesInCodeBlock]]
      -- region contains a whole CodeBlock

    ELSE --region contains exactly one code boundary--
      IF firstBdy - page >= endRegion - firstBdy THEN
        RETURN[[page, firstBdy - page]]
      ELSE RETURN[[firstBdy, endRegion - firstBdy]];
  END;

ForceOut: PUBLIC ENTRY PROCEDURE [space: Handle] =
  BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
  IF ~ApplyToSpace[handle, CachedRegion.startForceOut].validHandle THEN
    RETURN WITH ERROR Error[invalidHandle];
  [] ← ApplyToSpace[handle, CachedRegion.wait];
  END;

GetAttributes: PUBLIC ENTRY PROCEDURE [space: Handle]
  RETURNS [
    parent, lowestChild, nextSibling: Handle, base: PageOffset, size: PageCount,
    mapped: BOOLEAN] =
```

```
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    spaceD: CachedSpace.Desc;
    validSpace, validSwapUnit: BOOLEAN;
    interval: Interval;
    [validSpace, validSwapUnit, interval] ← Hierarchy.GetInterval[handle];
    IF ~(validSpace OR validSwapUnit) THEN RETURN WITH ERROR Error[invalidHandle];
    IF handle.level = 0 THEN {parent ← nextSibling ← nullHandle; base ← 0}
    ELSE
      BEGIN
      intervalParent: Interval = Hierarchy.GetInterval[
        CachedSpace.Handle[handle.level - 1, interval.page]].interval;
      pageNext: PageNumber = interval.page + interval.count;
      countNext: PageCount =
        intervalParent.page + intervalParent.count - pageNext;
      parent ← LOOPHOLE[CachedSpace.Handle[
        handle.level - 1, intervalParent.page]];
      nextSibling ←
        IF validSwapUnit THEN
        IF pageNext < intervalParent.page + intervalParent.count THEN
        LOOPHOLE[CachedSpace.Handle[handle.level, pageNext]] -- next swap unit
         ELSE nullHandle
        ELSE --validSpace--LOOPHOLE[Hierarchy.FindFirstWithin[
          CachedSpace.Handle[handle.level, pageNext], countNext]];
      base ← interval.page - intervalParent.page;
      END;
    [] ← Hierarchy.GetDescriptor[@spaceD, handle];
    -- (gets desc of parent space if this is a swap unit handle)
    lowestChild ←
      IF validSwapUnit THEN nullHandle
      ELSE --validSpace--
        IF spaceD.hasSwapUnits THEN LOOPHOLE[CachedSpace.Handle[
        handle.level + 1, handle.page]] -- first swap unit

        ELSE ---hasSwapUnits--
          IF handle.level < LAST[Level] THEN LOOPHOLE[Hierarchy.FindFirstWithin[
          CachedSpace.Handle[handle.level + 1, handle.page], interval.count]]
          ELSE --handle.level = LAST[Level]--nullHandle;
    size ← interval.count;
    mapped ← IF validSwapUnit THEN FALSE ELSE spaceD.state = mapped;
    END;

  GetHandle: PUBLIC ENTRY PROCEDURE [page: PageNumber] RETURNS [Handle] =
    BEGIN
    regionD: CachedRegion.Desc;
    handleLevel: CARDINAL;
    -- (not a CachedSpace.Level, since a swap unit may have handle.level = LAST[Level] + 1)
    IF page >= countVM THEN RETURN WITH ERROR Error[invalidParameters];
    regionD ← Projection.Get[page]; -- (gets level of parent space if swap units)
    handleLevel ← regionD.level + (IF regionD.hasSwapUnits THEN 1 ELSE 0);
    RETURN[
      LOOPHOLE[CachedSpace.Handle[
        handleLevel, Hierarchy.GetInterval[
        CachedSpace.Handle[handleLevel, page]].interval.page]]]
    END;

  GetWindow: PUBLIC ENTRY PROCEDURE [space: Handle] RETURNS [WindowOrigin] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    spaceD: SpaceD;
```



```
    validSpace, validSwapUnit: BOOLEAN;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[@spaceD, handle];
    IF ~validSpace AND ~validSwapUnit THEN RETURN WITH ERROR Error[invalidHandle];
    IF spaceD.state ~= mapped THEN RETURN WITH ERROR Error[noWindow];
    RETURN[
      IF spaceD.dataOrFile = data THEN defaultWindow
      ELSE
        [[spaceD.window.file.fID,
          IF spaceD.writeProtected THEN File.read ELSE File.read + File.write],
        spaceD.window.base]];
    END;

  Kill: PUBLIC ENTRY PROCEDURE [space: Handle] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    validSpace, validSwapUnit: BOOLEAN;
    interval: Interval;
    [validSpace, validSwapUnit, interval] ← Hierarchy.GetInterval[handle];
    IF ~validSpace AND ~validSwapUnit THEN RETURN WITH ERROR Error[invalidHandle];
    ApplyToInterval[interval, CachedRegion.kill];
    END;

  LongPointer: PUBLIC ENTRY PROCEDURE [space: Handle] RETURNS [LONG POINTER] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    IF ~Hierarchy.ValidSpaceOrSwapUnit[handle] THEN
      RETURN WITH ERROR Error[invalidHandle];
    RETURN[LongPointerFromPage[handle.page]];
    END;

  Pointer: PUBLIC ENTRY PROCEDURE [space: Handle] RETURNS [POINTER] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    IF ~Hierarchy.ValidSpaceOrSwapUnit[handle] THEN
      RETURN WITH ERROR Error[invalidHandle];
    IF handle.level <= handleMDS.level OR handle.page ~IN
      [intervalMDS.page..intervalMDS.page + intervalMDS.count) THEN
      RETURN WITH ERROR Error[invalidParameters];
    RETURN[LOOPHOLE[(handle.page - intervalMDS.page)*wordsPerPage]];
    END;

  VMPageNumber: PUBLIC ENTRY PROCEDURE [space: Handle] RETURNS [PageNumber] =
    BEGIN OPEN handle: LOOPHOLE[space, CachedSpace.Handle];
    IF ~Hierarchy.ValidSpaceOrSwapUnit[handle] THEN
      RETURN WITH ERROR Error[invalidHandle];
    RETURN[handle.page];
    END;
  ----------------------
  -- Monitor internal procedures:
  ----------------------
  --SpaceImplInternal.--

  NotePinned: PUBLIC SIGNAL [levelMax: Level, page: PageNumber] = CODE;
  --SpaceImplInternal.--
  ApplyToInterval: PUBLIC INTERNAL PROCEDURE [
    interval: Interval, operation: CachedRegion.Operation] =
    -- Performs, in ascending order, operation (successfully) for each swap unit of each region of i
**nterval (exactly as required by CachedRegion.Apply).
    -- May raise NotePinned (only if operation.action = unmap).
    -- for operation = writeProtect, flush, remap, and unmap, interval must start and end on a regio
**n boundary! (restriction of CachedRegion.Apply)
```

```
    BEGIN
    page, pageNext: PageNumber;
    outcome: CachedRegion.Outcome;
    FOR page ← interval.page, pageNext WHILE page < interval.page + interval.count
      DO
      -- for each swap unit in interval..
      DO
        --REPEAT operation UNTIL outcome = ok or notePinned--
        [outcome, pageNext] ← CachedRegion.Apply[page, operation];
        WITH outcome SELECT FROM
          ok => EXIT;
          notePinned -- [levelMax] -- =>
            BEGIN SIGNAL NotePinned[levelMax, page]; EXIT END;
          regionDMissing => Projection.Touch[page];
          regionDDirty => Projection.ForceOut[page];
          retry => NULL; -- go around again

          spaceDMissing -- [level] -- =>
            Hierarchy.Touch[CachedSpace.Handle[level, page]];
          -- error --
          ENDCASE => ERROR Bug[bug2];
        ENDLOOP;
      ENDLOOP;
    END;
  --SpaceImplInternal.--

  ApplyToSpace: PUBLIC INTERNAL PROCEDURE [
    handle: CachedSpace.Handle, operation: CachedRegion.Operation]
    RETURNS [validHandle: BOOLEAN] =
    BEGIN
    validSpace, validSwapUnit: BOOLEAN;
    interval: Interval;
    [validSpace, validSwapUnit, interval] ← Hierarchy.GetInterval[handle];
    validHandle ← validSpace OR validSwapUnit;
    IF validHandle THEN ApplyToInterval[interval, operation];
    END;
  --SpaceImplInternal.--

  ForAllRegions: PUBLIC INTERNAL PROCEDURE [
    interval: Interval, Predicate: PROCEDURE [RegionD] RETURNS [true: BOOLEAN]]
    RETURNS [BOOLEAN] =
    -- Returns as soon as Predicate returns FALSE.
    BEGIN
    page: PageNumber;
    regionD: RegionD;
    FOR page ← interval.page, regionD.interval.page + regionD.interval.count WHILE
      page < interval.page + interval.count DO
      regionD ← Projection.Get[page];
      IF ~Predicate[regionD] THEN RETURN[FALSE]
      ENDLOOP;
    RETURN[TRUE]
    END;
  ------------------------------------------------------------------------
  -- Virtual Memory Management support functions
  ------------------------------------------------------------------------

  VMMHelperProcess: PROCEDURE = -- root of VMM helper process, a monitor external
    -- provides access to the Hierarchy and Projection for page fault handling and the Replacemen
  **t Process.
```



```
    BEGIN
    page: PageNumber;
    operation: CachedRegion.Operation;
    outcome: CachedRegion.Outcome;
    HandleException: ENTRY PROCEDURE = INLINE
      -- Since this is a monitor entry, new entries may not be added asynchronously to the space a
  **nd region caches due to page faults. Thus, if a region or space descriptor is flushed from its ca
  **che, the higher-level databases can be temporarily inconsistent (not satisfying the monitor invar
  **iant).
      BEGIN
      DO
        --UNTIL ok--
        WITH outcome SELECT FROM
          ok => RETURN;
          regionDMissing => Projection.Touch[page];
          spaceDMissing --[level] -- =>
            Hierarchy.Touch[
              CachedSpace.Handle[level, page] ! Hierarchy.NotFound => CONTINUE];
          -- If a pagefault or an age happens, and then the space is deleted before this process can
  **service its needs, the space desc will be missing. In this case, we just retry the operation in the
  **current context (and the right thing will happen).

          error --[state]-- =>
            IF operation.action = activate THEN
              Runtime.CallDebugger["AddressFault"]
            ELSE ERROR Bug[bug3];
          ENDCASE => ERROR Bug[bug4];
        outcome ← CachedRegion.Apply[page, operation].outcome;
        ENDLOOP;
      END;
    DO
      --FOREVER--
      [page, operation, outcome] ← SwapperException.Await[];
      HandleException[];
      ENDLOOP;
    END;

  END.
```

LOG

(For earlier log entries see Pilot 3.0 archive version.)

Time: January 25, 1980 1:50 PM By: Knutsen Action: Use new ErrorType's.

Time: January 28, 1980 10:54 AM By: Forrest Action: Made SpaceImpl take starting para
***meters and eliminated InitSpace; copied in (most) of LongPtr< = >Page from UtilitiesImpl.*

Time: February 25, 1980 6:04 PM By: Knutsen AR3594: CreateUSU[spaceWithS
***U] raised wrong signal. AR1935: use Projection.DeleteSwapUnits. Renumbered the Internal err*
***ors.*

Time: April 16, 1980 10:09 AM By: Knutsen Action: Remap must always wait for operatio
***n complete to avoid having the file deleted out from under the swapping operation. DeleteIfLeaf*
***must avoid walking off the ends of VM. Converted SpaceImpl[] into InitializeSpace[] AGAIN!*

Time: April 17, 1980 1:03 PM By: Gobbel Action: Added transaction handles.

Time: April 21, 1980 6:01 PM By: Gobbel Action: Implemented Space.Copy.

Time: May 19, 1980 5:42 PM By: Gobbel Action: FrameOps = >Frame, ControlDefs =
***> PrincOps.*

Time: May 21, 1980 2:31 PM By: Gobbel Action: Mesa 6 change: f.code.longbase repl
***aced by RuntimeInternal.CodeBase[frame] in SpaceForCode.*

Time: June 16, 1980 5:12 PM By: Gobbel Action: Transaction operations added.

Time: June 19, 1980 2:27 PM By: Gobbel Action: Created SpaceImplA from parts of ol
***d SpaceImpl.*

Time: August 7, 1980 1:06 PM By: Knutsen Action: Moved initialization relevant to Spac
***eImplB to there. Moved ApplyToInterval, etc. here. GetWindow returns current writeProtected-*
***ness. Implement MakeGlobalFrame*, MakeProcedure*, etc.*
Time: September 8, 1980 4:14 PM By: Knutsen Action: Delete now does Withdra
***wFromTransaction.*
Time: September 11, 1980 1:15 PM By: Gobbel Action: Imported TransactionStat
***e.*
Time: October 10, 1980 9:09 PM By: Fay Action: Fixed DeleteIfLeaf to correctly recog
***nize a leaf space with no right neighbor.*

```
-- VMMgr>SpaceImplB.mesa (last edited by Knutsen on October 16, 1980 11:02 AM)
-- Note: There is another version of SpaceImpl for UtilityPilot which should track changes to this m
**odule.
-- Handling of transactions: In the beginning, a space is created, and it is not part of a transaction.
**A space becomes associated with a transaction whenever a non-null transaction handle is pass
**ed to a Space operation (except CopyIn/Out), and the transaction handle is stored in the space
**descriptor at that time. If later a different transaction handle comes along for an operation on th
**at space, it is an error. A space ceases to be part of a transaction when: (1) ReleaseFromTrans
**action is called; (2) the space is unmapped or deleted. Remap is handled as as if it were an Un
**map followed by a Map.
-- Implementation Notes:
-- The routines herein have this organization: first, check for all client errors, and RETURN WITH E
**RROR if any; then, make changes to the VM databases.
-- Future Improvements:
-- If we would remember when a whole mapping space had been logged, we could avoid doing it a
**gain later.

DIRECTORY
  CachedRegion USING [
    BackFileType, Desc, forceOut, invalidate, makeWritable, Mapped,
    maxRemapSpaceSize, needsLogging, Operation, pageLocationInSpace, startUnmap,
    wait, writeProtect],
  CachedSpace USING [DataOrFile, Desc, Handle, Level],
  Environment USING [bitsPerWord, wordsPerPage],
  File USING [
    Capability, Error, ErrorType, firstPageNumber, lastPageNumber, nullID,
    PageCount, read, Unknown, write],
  Frame USING [GetReturnLink],
  Hierarchy USING [GetDescriptor, GetInterval, Update],
  Inline USING [BITAND, LowHalf],
  KernelFile USING [GetFileAttributes, LogContents],
  KernelSpace USING [],
  MapLog USING [WriteLog],
  PrincOps USING [StateVector, TrapLink],
  Projection USING [Get],
  ResidentHeap USING [first64K, FreeNode, MakeNode],
  Runtime USING [CallDebugger],
  SDDefs USING [SD, sWriteProtect],
  Space USING [
    defaultWindow, Error, ErrorType, InsufficientSpace, PageCount, PageNumber,
    WindowOrigin],
  SpaceImplInternal USING [
    ApplyToInterval, Interval, ForAllRegions, Level, NotePinned, RegionD, SpaceD,
    spaceLock],
  SystemInternal USING [Unimplemented],
  SpecialSpace USING [],
  Transaction USING [Handle, InvalidHandle, nullHandle],
  TransactionState USING [AddToTransaction, WithdrawFromTransaction],
  Trap USING [ReadOTP],
  VM USING [Interval],
  VMMPrograms USING [],
  VMMgrStore USING [AllocateWindow, DeallocateWindow],
  Volume USING [ID, InsufficientSpace, Unknown],
  Zone USING [Base, Status];

SpaceImplB: MONITOR LOCKS SpaceImplInternal.spaceLock
  IMPORTS
    CachedRegion, File, Frame, Hierarchy, Inline, KernelFile, MapLog, Projection,
```



```
    ResidentHeap, Runtime, Space, SpaceImplInternal, SystemInternal, Transaction,
    TransactionState, Trap, VMMgrStore, Volume
  EXPORTS KernelSpace, Space, SpaceImplInternal
  SHARES File -- USING [fID, permissions] -- =
  BEGIN OPEN Space, SpaceImplInternal;
  Handle: PUBLIC TYPE = CachedSpace.Handle;
  Bug: PRIVATE ERROR [type: BugType] = CODE;
  BugType: TYPE = {
    badTxButNeedsLogging, fileDisappeared, initHeapErr, nullTxButNeedsLogging,
    noSpaceForLogging, remapHeapAlloc, remapHeapFree, spaceDAbsent,
    unmappedSpaceToUnmapInternal, volDisappeared,
    yourSpaceIsAlreadyinOtherTransaction};
  pageSize: CARDINAL = Environment.wordsPerPage;
  nullTransaction: Transaction.Handle = Transaction.nullHandle;
  initRemapSize: PageCount = 250;
  -- initial guess at page count of largest space to be remapped.
  currentRemapSize: PageCount ← initRemapSize;
  ----------
  -- Initialization:
  ----------
  --SpaceImplInternal.--
  InitializeSpaceImplB: PUBLIC --EXTERNAL--PROCEDURE[] =
    BEGIN
    node: Zone.Base RELATIVE POINTER TO UNSPECIFIED;
    size: CARDINAL;
    status: Zone.Status;
    size ← (initRemapSize + Environment.bitsPerWord - 1)/Environment.bitsPerWord;
    [node, status] ← ResidentHeap.MakeNode[n: size, alignment:];
    -- for remapping operations.
    IF status ~= okay THEN ERROR Bug[initHeapErr];
    CachedRegion.pageLocationInSpace ← DESCRIPTOR[
      @ResidentHeap.first64K[node], size]; -- for remapping operations.
    SDDefs.SD[SDDefs.sWriteProtect] ← WriteProtectTrap;
    END;
  ----------
  -- Monitor External (and Nested Internal) Procedures:
  ----------

  CopyIn: PUBLIC --EXTERNAL--PROCEDURE [
    space: Handle, window: WindowOrigin, transaction: Transaction.Handle] =
    -- The transaction parameter is only present for possible future implementation of locking. It is
**not relevant to the current implementation.
    BEGIN
    CopyInInternal: INTERNAL PROCEDURE[] =
      BEGIN
      spaceD, mappingSpaceD, fromSpaceD: SpaceD;
      GetCopyInfo[@spaceD, @mappingSpaceD, space];
      IF mappingSpaceD.writeProtected THEN {
        IF KernelFile.GetFileAttributes[mappingSpaceD.window.file].immutable THEN
          File.Error[immutable]
        ELSE File.Error[insufficientPermissions]};
      fromSpaceD ← mappingSpaceD; -- generate a desc for the new window..
      fromSpaceD.interval ← spaceD.interval;
      ProcessWindow[@fromSpaceD, @window, forReadingOrWriting, fileSpaceOnly];
      -- may raise File.Unknown or Volume.Unknown.
      IF mappingSpaceD.transaction ~= nullTransaction THEN {
        ApplyToInterval[spaceD.interval, CachedRegion.forceOut];
        -- make sure backing file is up-to-date.
```

```
      KernelFile.LogContents[
        -- may raise Volume.InsufficientSpace.
        transaction: mappingSpaceD.transaction,
        file: [mappingSpaceD.window.file.fID, File.read + File.write],
        base:
        mappingSpaceD.window.base +
          (spaceD.interval.page - mappingSpaceD.interval.page),
        count: spaceD.countMapped]};
    ApplyToInterval[
      spaceD.interval,
      [ifMissing: report, ifCheckedOut: wait, afterForking: return,
        vp: copyIn[from: @fromSpaceD]]];
    ApplyToInterval[spaceD.interval, CachedRegion.wait];
    -- must wait for complete, since client could delete the source window.

    END --CopyInInternal--
  ;
  EnterSpace[CopyInInternal];
  END;

CopyOut: PUBLIC --EXTERNAL--PROCEDURE [
  space: Handle, window: WindowOrigin, transaction: Transaction.Handle] =
  BEGIN
  CopyOutInternal: INTERNAL PROCEDURE[] =
    BEGIN
    spaceD, mappingSpaceD, toSpaceD: SpaceD;
    GetCopyInfo[@spaceD, @mappingSpaceD, space];
    toSpaceD ← mappingSpaceD; -- generate a desc for the new window..
    toSpaceD.interval ← spaceD.interval;
    ProcessWindow[@toSpaceD, @window, forWriting, fileSpaceOnly];
    -- may raise File.Unknown or Volume.Unknown.
    IF transaction ~= nullTransaction THEN
      KernelFile.LogContents[
        transaction, toSpaceD.window.file, toSpaceD.window.base,
        toSpaceD.countMapped]; -- may raise Transaction.InvalidHandle.
    -- We don't need to update the space desc with the possibly-new transaction since the transa
**ction is associated with the window copied to, not the current space.
    ApplyToInterval[
      spaceD.interval,
      [ifMissing: report, ifCheckedOut: wait, afterForking: return,
        vp: copyOut[to: @toSpaceD]]];
    ApplyToInterval[spaceD.interval, CachedRegion.wait];
    -- must wait for complete, since client could delete the destination window.

    END --CopyOutInternal--
  ;
  EnterSpace[CopyOutInternal];
  END;

MakeReadOnly: PUBLIC --EXTERNAL--PROCEDURE [
  space: Handle, transaction: Transaction.Handle] =
  BEGIN
  MakeReadOnlyInternal: INTERNAL PROCEDURE[] =
    BEGIN
    spaceD: SpaceD;
    GetSpaceDesc[@spaceD, space, --requiredState:--mapped];
    JoinTransaction[space, @spaceD, transaction];
    -- may raise Transaction.InvalidHandle.
```



```
    spaceD.writeProtected ← TRUE;
    Hierarchy.Update[@spaceD];
    -- note that the Projection and Hierarchy disagree until the following statement is completed.
    ApplyToInterval[spaceD.interval, CachedRegion.writeProtect];
    END;
  EnterSpace[MakeReadOnlyInternal];
  END;

MakeWritable: PUBLIC --EXTERNAL--PROCEDURE [
  space: Handle, file: File.Capability, transaction: Transaction.Handle] =
  BEGIN
  MakeWritableInternal: INTERNAL PROCEDURE[] =
    BEGIN
    spaceD: SpaceD;
    immutable: BOOLEAN;
    readOnly: BOOLEAN;
    GetSpaceDesc[@spaceD, space, --requiredState:--mapped];
    IF file.fID ~= spaceD.window.file.fID THEN Error[invalidMappingOperation];
    [immutable: immutable, readOnly: readOnly] ← KernelFile.GetFileAttributes[
      file];
    IF immutable THEN File.Error[immutable];
    IF readOnly THEN File.Error[insufficientPermissions];
    JoinTransaction[space, @spaceD, transaction];
    -- may raise Transaction.InvalidHandle.
    spaceD.writeProtected ← FALSE;
    Hierarchy.Update[@spaceD];
    -- note that the Projection and Hierarchy disagree until the following statement is completed.
    ApplyToInterval[
      spaceD.interval,
      IF spaceD.transaction = nullTransaction THEN CachedRegion.makeWritable
      ELSE CachedRegion.needsLogging];
    END;
  EnterSpace[MakeWritableInternal];
  END;

Map: PUBLIC --EXTERNAL--PROCEDURE [
  space: Handle, window: WindowOrigin, transaction: Transaction.Handle] =
  BEGIN
  MapInternal: INTERNAL PROCEDURE[] =
    BEGIN
    spaceD: SpaceD;
    GetSpaceDesc[@spaceD, space, --requiredState:--unmapped];
    ProcessWindow[@spaceD, @window, forReadingOrWriting, dataSpaceOK];
    -- may raise File.Unknown, Volume.InsufficientSpace, or Volume.Unknown.
    JoinTransaction[
      space, @spaceD, transaction -- may raise Transaction.InvalidHandle.
        ! UNWIND => ReleaseDefaultWindow[@spaceD]; ];
    -- No more signals should be raised past this point!
    spaceD.state ← mapped;
    Hierarchy.Update[@spaceD];
    ApplyToInterval[
      spaceD.interval,
      [ifMissing: report, ifCheckedOut: wait, afterForking: --don't care--,
        vp: map[
        level: space.level,
        backFileType:
        IF spaceD.dataOrFile = CachedSpace.DataOrFile[file] THEN file ELSE data,
        andWriteProtect: spaceD.writeProtected,
```

```
        andNeedsLogging: ~spaceD.writeProtected AND spaceD.transaction ~ =
          nullTransaction]]];
      MapLog.WriteLog[Interval[spaceD.interval.page, spaceD.countMapped], @spaceD]
      END --MapInternal--
    ;
    EnterSpace[MapInternal];
    END;
  --KernelSpace.--

  ReleaseFromTransaction: PUBLIC --EXTERNAL--PROCEDURE [
    space: Handle, transaction: Transaction.Handle, andInvalidate: BOOLEAN] =
    BEGIN
    ReleaseFromTransactionInternal: INTERNAL PROCEDURE[] =
      BEGIN
      spaceD: SpaceD;
      GetSpaceDesc[@spaceD, space, --requiredState:--dontCare];
      IF spaceD.transaction ~ = transaction THEN RETURN
        -- the space in the given transaction has been deleted.  This is a different space (but with th
**e same handle).

      ELSE
        BEGIN
        IF spaceD.state = mapped THEN {
          IF andInvalidate THEN
            ApplyToInterval[spaceD.interval, CachedRegion.invalidate];
          IF ~spaceD.writeProtected THEN
            ApplyToInterval[spaceD.interval, CachedRegion.makeWritable]};
        spaceD.transaction ← nullTransaction;
        Hierarchy.Update[@spaceD];
        END;
      END;
    EnterSpace[ReleaseFromTransactionInternal];
    END;

  Remap: PUBLIC --EXTERNAL--PROCEDURE [
    space: Handle, window: WindowOrigin, transaction: Transaction.Handle] =
    BEGIN
    RemapInternal: INTERNAL PROCEDURE[] =
      BEGIN
      spaceDOld, spaceDNew: SpaceD;
      GetSpaceDesc[@spaceDOld, space, --requiredState:--mapped];
      spaceDNew ← spaceDOld; -- generate a desc for the new space/window.
      IF spaceDOld.transaction ~ = nullTransaction THEN
        TransactionState.WithdrawFromTransaction[
          spaceDOld.transaction, space ! Transaction.InvalidHandle => CONTINUE];
      -- (tx had already ended)
      spaceDNew.transaction ← nullTransaction;
      -- (will join if caller passed new transacton)
      ProcessWindow[@spaceDNew, @window, forWriting, dataSpaceOK];
      -- may raise File.Unknown, Volume.InsufficientSpace, or Volume.Unknown.
      -- scope of UNWIND --
      BEGIN
      ENABLE UNWIND => ReleaseDefaultWindow[@spaceDNew];
      JoinTransaction[space, @spaceDNew, transaction];
      -- may raise Transaction.InvalidHandle.
      IF spaceDNew.transaction ~ = nullTransaction THEN
        -- log the contents of the destination window..
        KernelFile.LogContents[
          spaceDNew.transaction, spaceDNew.window.file, spaceDNew.window.base,
```



```
          spaceDNew.countMapped]; -- may raise Volume.InsufficientSpace.

      END --scope of UNWIND--
      ;
      -- No more signals should be raised past this point!
      IF spaceDOld.interval.count > currentRemapSize THEN
        -- get a bigger remap array
        BEGIN
        node: Zone.Base RELATIVE POINTER TO UNSPECIFIED;
        size: CARDINAL;
        status: Zone.Status;
        IF spaceDOld.interval.count > CachedRegion.maxRemapSpaceSize THEN
          ERROR SystemInternal.Unimplemented;
        IF ResidentHeap.FreeNode[
          Inline.LowHalf[
          BASE[CachedRegion.pageLocationInSpace] - ResidentHeap.first64K]] ~ = okay
          THEN ERROR Bug[remapHeapFree];
        currentRemapSize ← spaceDOld.interval.count;
        size ←
          (currentRemapSize + Environment.bitsPerWord -
            1)/Environment.bitsPerWord;
        [node, status] ← ResidentHeap.MakeNode[n: size, alignment:];
        IF status ~ = okay THEN ERROR Bug[remapHeapAlloc];
        CachedRegion.pageLocationInSpace ← DESCRIPTOR[
          @ResidentHeap.first64K[node], size];
        END;
      -- At this point, swap units may soon reside in either of two windows.  Mark all swap units "in
**old window", and force out dirty ones unless data space or write protected:
      ApplyToInterval[
        spaceDOld.interval,
        [ifMissing: report, ifCheckedOut: wait, afterForking: return,
          vp: remapA[
          firstClean: spaceDOld.dataOrFile = file AND
          ~spaceDOld.writeProtected]]];
      -- (assume writeProtected implies clean, i.e. no magic stores by debugger, etc.)
      -- implicit parameter: CachedRegion.pageLocationInSpace.
      spaceDNew.state ← beingRemapped;
      -- "the space desc has changed to the new window".
      Hierarchy.Update[@spaceDNew];
      -- Ensure each region is in the new window, or is in and dirty:
      ApplyToInterval[
        spaceDNew.interval,
        [ifMissing: report, ifCheckedOut: wait, afterForking: return,
          vp: remapB[from: @spaceDOld]]];
      -- implicit parameter: CachedRegion.pageLocationInSpace.
      spaceDNew.state ← mapped;
      -- "once again, there is only one window for this space"
      Hierarchy.Update[@spaceDNew];
      -- (end of scope of pageLocationInSpace)
      -- (For remote swapping, must unpin old file (and containing volume) from FilePageTransferr
**er cache)
      ApplyToInterval[spaceDNew.interval, CachedRegion.wait];
      -- must wait for complete, since we or client may delete the old window.
      ReleaseDefaultWindow[@spaceDOld];
      MapLog.WriteLog[
        Interval[spaceDNew.interval.page, spaceDNew.countMapped], @spaceDNew];
      IF spaceDNew.countMapped < spaceDNew.interval.count THEN
        MapLog.WriteLog[
```

```
        Interval[
        spaceDNew.countMapped,
        spaceDNew.interval.count - spaceDNew.countMapped], NIL]
    END --RemapInternal--
  ;
  EnterSpace[RemapInternal];
  END;

Unmap: PUBLIC --EXTERNAL--PROCEDURE [space: Handle] =
  BEGIN
  UnmapLocalInternal: INTERNAL PROCEDURE[] = {
    spaceD: SpaceD;
    GetSpaceDesc[@spaceD, space, --requiredState:--mapped];
    IF spaceD.transaction ~= nullTransaction THEN {
      TransactionState.WithdrawFromTransaction[
        spaceD.transaction, space ! Transaction.InvalidHandle => CONTINUE];
      -- (tx had already ended)
      spaceD.transaction ← nullTransaction};
    UnmapInternal[@spaceD]};
  EnterSpace[UnmapLocalInternal];
  END;

WriteProtectTrap: --EXTERNAL--PROCEDURE[] =
  -- Handler for processor-generated trap.
  BEGIN
  record: RECORD [
    -- so compiler won't grouse about not referencing dummy.
    faultingPage: PageNumber,
    dummy: CARDINAL];
  -- to force state vector to not overlap local 0 and local 1.
  state: PrincOps.StateVector;
  state ← STATE; -- must be first instruction.
  record.faultingPage ← LOOPHOLE[Trap.ReadOTP[], PageNumber];
  -- should be second instruction, as trap parameter is put in local zero.
  HandleWriteProtectTrap[record.faultingPage];
  state.dest ← Frame.GetReturnLink[];
  state.source ← PrincOps.TrapLink;
  RETURN WITH state; -- restart the trapee, he should work this time.

  END;
-------------------------------
-- Monitor Entry Procedures:
-------------------------------

EnterSpace: PUBLIC ENTRY PROCEDURE [internalProc: PROCEDURE[]] =
  -- Enters the Space monitor and calls internalProc. internalProc may freely signal or not catch
**signals from below (but should catch UNWIND from here if appropriate).
  BEGIN
  unknownFile: File.Capability;
  fileErrorType: File.ErrorType;
  spaceAvailable: Space.PageCount;
  spaceErrorType: Space.ErrorType;
  unknownVolume: Volume.ID;
  BEGIN
  internalProc[
    ! File.Error --[type]-- => {fileErrorType ← type; GO TO FileError};
    File.Unknown --[file]-- => {unknownFile ← file; GO TO FileUnknown};
    Space.Error --[type]-- => {spaceErrorType ← type; GO TO SpaceError};
```



```
    Space.InsufficientSpace --[available]-- => {
      spaceAvailable ← available; GO TO SpaceInsufficientSpace};
    Transaction.InvalidHandle => GO TO TransactionInvalidHandle;
    Volume.Unknown --[volume]-- => {
      unknownVolume ← volume; GO TO VolumeUnknown};
    Volume.InsufficientSpace => GO TO VolumeInsufficientSpace];
  EXITS
    FileError => RETURN WITH ERROR File.Error[fileErrorType];
    FileUnknown => RETURN WITH ERROR File.Unknown[unknownFile];
    SpaceError => RETURN WITH ERROR Space.Error[spaceErrorType];
    SpaceInsufficientSpace =>
      RETURN WITH ERROR Space.InsufficientSpace[spaceAvailable];
    TransactionInvalidHandle => RETURN WITH ERROR Transaction.InvalidHandle;
    VolumeUnknown => RETURN WITH ERROR Volume.Unknown[unknownVolume];
    VolumeInsufficientSpace => RETURN WITH ERROR Volume.InsufficientSpace;
  END;
  END;

HandleWriteProtectTrap: ENTRY PROCEDURE [faultingPage: PageNumber] =
  BEGIN
  -- Either this region is in a transaction and needs to be logged, or this is a real write protect faul
**t.
  mappingSpace: SpaceD;
  swapInterval: VM.Interval;
  region: RegionD = Projection.Get[faultingPage];
  IF ~region.needsLogging THEN Runtime.CallDebugger["WriteProtectFault"L];
  [] ← Hierarchy.GetDescriptor[
    @mappingSpace, [region.levelMapped, region.interval.page]];
  IF mappingSpace.state = missing THEN ERROR Bug[spaceDAbsent];
  IF mappingSpace.transaction = nullTransaction THEN
    ERROR Bug[nullTxButNeedsLogging];
  swapInterval ← Hierarchy.GetInterval[
    CachedSpace.Handle[
    region.level + (IF region.hasSwapUnits THEN 1 ELSE 0),
    faultingPage]].interval;
  KernelFile.LogContents[
    transaction: mappingSpace.transaction,
    file: [mappingSpace.window.file.fID, File.read + File.write],
    base:
    (swapInterval.page - mappingSpace.interval.page) + mappingSpace.window.base,
    count:
    IF swapInterval.page + swapInterval.count <=
    mappingSpace.interval.page + mappingSpace.countMapped THEN
    swapInterval.count
    ELSE
      (mappingSpace.interval.page + mappingSpace.countMapped) -
        swapInterval.page ! File.Unknown => ERROR Bug[fileDisappeared];
    Transaction.InvalidHandle => ERROR Bug[badTxButNeedsLogging];
    Volume.InsufficientSpace => ERROR Bug[noSpaceForLogging];
    Volume.Unknown => ERROR Bug[volDisappeared]];
  ApplyToInterval[swapInterval, CachedRegion.makeWritable];
  END;
-------------------------------
--Monitor Internal Procedures:
-------------------------------

GetCopyInfo: INTERNAL PROCEDURE [
  pSpaceD, pMappingSpaceD: POINTER TO SpaceD, space: Handle] =
```

```
    -- Validates space, assures that it is mapped or a descendent of a mapped space, returns spac
**e descriptor for space (synthesizes one if space is a swap unit) and mapping space.
    BEGIN
    validSpace, validSwapUnit: BOOLEAN;
    region: RegionD;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[pSpaceD, space];
    IF ~validSpace AND ~validSwapUnit THEN Error[invalidHandle];
    region ← Projection.Get[space.page];
    IF region.state ~IN CachedRegion.Mapped THEN Error[noWindow];
    IF space.level = pSpaceD.level AND pSpaceD.level = region.levelMapped THEN
      pMappingSpaceD↑ ← pSpaceD↑
    ELSE
      [] ← Hierarchy.GetDescriptor[
        pMappingSpaceD, Handle[space.page, region.levelMapped]];
    IF validSwapUnit THEN
      pSpaceD.interval ← Hierarchy.GetInterval[space].interval;
    END;

  GetSpaceDesc: INTERNAL PROCEDURE [
    pSpaceD: POINTER TO SpaceD, space: Handle,
    requiredState: {mapped, unmapped, dontCare}] =
    -- Validates that space is a real space (not a swap unit), gets space descriptor, assures desired
**mapping state.
    BEGIN
    Unmapped: INTERNAL PROCEDURE [regionD: RegionD] RETURNS [BOOLEAN] = {
      RETURN[regionD.state = unmapped]};
    validSpace, validSwapUnit: BOOLEAN;
    [validSpace, validSwapUnit] ← Hierarchy.GetDescriptor[pSpaceD, space];
    IF validSwapUnit THEN Error[notApplicableToSwapUnit]
    ELSE IF ~validSpace THEN Error[invalidHandle];
    SELECT requiredState FROM
      mapped => IF pSpaceD.state ~= mapped THEN Error[noWindow];
      unmapped =>
        IF ~ForAllRegions[pSpaceD.interval, Unmapped] THEN
          Error[invalidMappingOperation];
      dontCare => NULL;
      ENDCASE;
    END;

  JoinTransaction: INTERNAL PROCEDURE [
    space: Handle, pSpaceD: POINTER TO SpaceD, transaction: Transaction.Handle] =
    -- It is best if this procedure is called after all other possible errors have been detected, since t
**here is no way to back out of the AddToTransaction.
    -- May raise Transaction.InvalidHandle.
    {
    IF transaction ~= nullTransaction AND pSpaceD.transaction ~= transaction THEN
      {
      IF pSpaceD.transaction ~= nullTransaction THEN
        Bug[yourSpaceIsAlreadyInOtherTransaction];
      pSpaceD.transaction ← transaction;
      TransactionState.AddToTransaction[transaction, space]}};

  ProcessWindow: INTERNAL PROCEDURE [
    pSpaceD: POINTER TO SpaceD, pWindow: POINTER TO Space.WindowOrigin,
    usage: {forWriting, forReadingOrWriting},
    kind: {dataSpaceOK, fileSpaceOnly}] =
    -- Fills in pSpaceD.dataOrFile, .writeProtected, .countMapped, and .window. Allocates window
**if data space.
```



```
    -- May raise File Unknown, Volume InsufficientSpace, or Volume Unknown (as well as Space E
**rror, etc. etc.).
    BEGIN
    IF pWindow↑ = defaultWindow THEN --data space--
      BEGIN
      IF kind = fileSpaceOnly THEN Error[invalidMappingOperation];
      pSpaceD.dataOrFile ← data;
      pSpaceD.writeProtected ← FALSE;
      pSpaceD.countMapped ← pSpaceD.interval.count;
      pSpaceD.window.file.fID ← File.nullID; -- "backing store not allocated yet"
      VMMgrStore.AllocateWindow[@(pSpaceD.window), pSpaceD.interval.count];
      -- may raise Volume.InsufficientSpace.

      END
    ELSE --file space--
      BEGIN
      countFile: File.PageCount;
      immutable: BOOLEAN;
      countMappedFile: File.PageCount;
      pSpaceD.dataOrFile ← file;
      [size: countFile, immutable: immutable, readOnly: pSpaceD.writeProtected] ←
        KernelFile.GetFileAttributes[pWindow.file];
      IF pWindow.base ~IN [File.firstPageNumber..File.lastPageNumber] THEN
        Error[invalidWindow];
      IF Inline.BITAND[pWindow.file.permissions, File.read] = 0 THEN
        File.Error[insufficientPermissions];
      IF pSpaceD.writeProtected AND usage = forWriting THEN {
        IF immutable THEN File.Error[immutable]
        ELSE File.Error[insufficientPermissions]};
      IF ~pSpaceD.writeProtected AND immutable THEN File.Error[immutable];
      countMappedFile ←
        IF countFile <= pWindow.base THEN 0 ELSE countFile - pWindow.base;
      -- the mapped amount of this space (no danger of underflow)
      pSpaceD.countMapped ←
        IF pSpaceD.interval.count <= countMappedFile THEN pSpaceD.interval.count
        ELSE Inline.LowHalf[countMappedFile]; -- (no danger of truncation)
      pSpaceD.window ← pWindow↑;
      END;
    -- (For remote swapping, must pin file (and containing volume) in FilePageTransferrer caches)

    END;

  ReleaseDefaultWindow: INTERNAL PROCEDURE [pSpaceD: POINTER TO SpaceD] = {
    IF pSpaceD.dataOrFile = data AND pSpaceD.window.file.fID ~= File.nullID THEN
      VMMgrStore.DeallocateWindow[@(pSpaceD↑.window), pSpaceD.interval.count]};
  --SpaceImplInternal.--

  UnmapInternal: PUBLIC INTERNAL PROCEDURE [pSpaceD: POINTER TO SpaceD] =
    BEGIN OPEN pSpaceD;
    IF state ~= mapped THEN ERROR Bug[unmappedSpaceToUnmapInternal];
    ApplyToInterval[
      interval, CachedRegion.startUnmap !
      NotePinned --[levelMax, page]-- =>
        BEGIN
        levelSub: Level;
        spaceDSub: SpaceD;
        FOR levelSub IN [level + 1..levelMax] DO
          [] ← Hierarchy.GetDescriptor[
            @spaceDSub, CachedSpace.Handle[levelSub, page]];
```

```
        IF spaceDSub.pinned THEN {
          spaceDSub.pinned ← FALSE; Hierarchy.Update[@spaceDSub]; EXIT};
        ENDLOOP;
      RESUME
      ;
      END; ];
  ApplyToInterval[interval, CachedRegion.wait];
  pinned ← FALSE;
  state ← unmapped;
  pSpaceD.transaction ← nullTransaction;
  -- since there's no longer any assosciated file.
  Hierarchy.Update[pSpaceD];
  -- (For remote swapping, must unpin old file (and containing volume) from FilePageTransferrer
**cache)
  IF dataOrFile = data THEN
    VMMgrStore.DeallocateWindow[@window, interval.count];
  MapLog.WriteLog[Interval[interval.page, countMapped], NIL];
  END;

END.
LOG
(For earlier log entries see Pilot 3.0 archive version.)
January 25, 1980 1:50 PM      Knutsen   Use new ErrorType's.
January 28, 1980 10:54 AM     Forrest   Made SpaceImpl take starting parameters and eliminat
**ed InitSpace; copied in (most) of LongPtr< = >Page from UtilitiesImpl.
February 25, 1980 6:04 PM     Knutsen   AR3594: CreateUSU[spaceWithSU] raised wrong signal
**. AR1935: use Projection.DeleteSwapUnits. Renumbered the Internal errors.
April 16, 1980 10:09 AM       Knutsen   Remap must always wait for operation complete to avoi
**d having the file deleted out from under the swapping operation. DeleteIfLeaf must avoid walkin
**g off the ends of VM. Converted SpaceImpl[] into InitializeSpace[] AGAIN!
April 17, 1980 1:03 PM        Gobbel    Added transaction handles.
April 21, 1980 6:01 PM        Gobbel    Implemented Space.Copy.
May 19, 1980 5:42 PM          Gobbel    FrameOps = >Frame, ControlDefs = > PrincOps.
May 21, 1980 2:31 PM          Gobbel    Mesa 6 change: f.code.longbase replaced by RuntimeI
**nternal.CodeBase[frame] in SpaceForCode.
June 16, 1980 5:12 PM         Gobbel    Transaction operations added.
July 19, 1980 12:00 PM        McJones   Split module into SpaceImplA/B. Define nullTransacti
**on locally; don't disable interrupts in WriteProtectTrap; add new parameters to MakeReadonly,
**MakeWritable, and Remap
August 11, 1980 5:11 PM       Knutsen   Make spaces enter transactions anytime. Implement co
**py-on-write via writeProtect trap. Use new Apply operations for write protection. Add ReleaseFr
**omTransaction. Moved ApplyToInterval, etc. to SpaceImplA. Revise error handling.
August 12, 1980 7:11 PM       Knutsen   CheckCopyArgs didn't copy space desc.
August 29, 1980 4:46 PM       Knutsen   SpaceInternal renamed to KernelSpace. Map didn't ret
**urn invalidWindow for bad base. If file immutable and write permission, Error[immutable]. Han
**dle some cases of AllocateWindow signalling Vol.InsuffSpace.
September 15, 1980 9:35 AM    Knutsen   Restructured signal handling. Fix CopyIn/Out, MakeR
**eadOnly, MakeWritable. Fix ReleaseFromTransaction deadlock. CopyIn/Out do not join transa
**ction. Leave transaction on Unmap and possibly on Remap.
October 10, 1980 10:44 PM     Fay       Fixed CopyIn to raise File.Error[immutable] for space w
**ith immutable backing file and [insufficientPermissions] for any other writeProtected spaces; re
**ordered checks in MakeWritable so that File.Error[immutable] is raised for a space with immuta
**ble window.
October 16, 1980 11:02 AM     Knutsen   Fix HandleWriteProtectTrap and CopyIn to supply write
**permission in Capability used for logging.
```

```
-- VMMgr>STLeafImpl.mesa (last edited by Knutsen on September 16, 1980 5:56 PM)

DIRECTORY
  Environment USING [Base, bitsPerWord, first64K],
  File USING [
    Capability, Create, Delete, GetSize, ID, LimitPermissions, MakePermanent,
    PageCount, PageNumber, read, SetSize, Unknown, write],
  Inline USING [LowHalf],
  KernelFile USING [GetRootFile],
  PilotFileTypes USING [tAnonymousFile, tVMBackingFile],
  PilotSwitches USING [switches --.v--],
  ResidentHeap USING [MakeNode],
  Runtime USING [CallDebugger],
  SimpleSpace USING [Create, Page, Map],
  Space USING [Handle, PageNumber, WindowOrigin],
  StoragePrograms USING [LongPointerFromPage],
  STLeaf,
  Utilities USING [BitIndex],
  Volume USING [ID, GetAttributes, SystemID],
  VM USING [PageCount, PageNumber, PageOffset],
  VMMgrStore,
  VMMPrograms,
  Zone USING [Status];

STLeafImpl: PROGRAM
  -- logically, this should be a MONITOR, but at present all procedures (in STLeaf and VMMgrStore
**) in this module are only called from within the SpaceImpl monitor.


  IMPORTS
    File, Inline, KernelFile, PilotSwitches, ResidentHeap, Runtime, SimpleSpace,
    StoragePrograms, Volume
  EXPORTS STLeaf, VMMgrStore, VMMPrograms
  SHARES File --USING[fID]-- =
  BEGIN
  Bug: PRIVATE --PROGRAMMING--ERROR [BugType] = CODE;
  BugType: TYPE = {bug0, bug1, bug2, bug3, bug4, bug5, bug6};

  --
  -- VMMgrStore

  --                                  Layout of VM backing file:
  -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  -- | MapLog | Space B-trees | backing pool for Data Spaces |
  -- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  countLog: VM.PageCount = 40;
  countSpaceBTreesMax: VM.PageCount = 200; -- allows increasing countSpaceBTrees
  -- from the debugger, if necessary
  CountSpaceBTreesRange: TYPE = CARDINAL [0..countSpaceBTreesMax);
  countSpaceBTrees: CountSpaceBTreesRange ← 150; -- must be big enough to satisfy
  -- any Pilot client.
  minCountDataPool: VM.PageCount = 250; -- enough space for a (Dolphin) bitmap.
  maxCountDataPool: VM.PageCount = 1000; -- (arbitrary)
  countDataPool: VM.PageCount; -- depends on space available on volume.
  minBackFileSize: VM.PageCount = countLog + countSpaceBTrees + minCountDataPool;
  -- minimum size for backing file.
  backFileSize: File.PageCount;
```



```
  pageLog: VM.PageNumber = 0;
  pageSpaceBTrees: VM.PageNumber = pageLog + countLog;
  pageDataPool: VM.PageNumber = pageSpaceBTrees + countSpaceBTrees;
  systemVolume: Volume.ID;
  backingFile: File.Capability;
  -- data for Data Pool:
  fractionThreshold: CARDINAL = 10; -- windows larger than countDataPool /
  -- fractionThreshold will be allocated as separate files.
  countThreshold: VM.PageCount;
  pageDataPoolAvailable: LONG POINTER TO PACKED ARRAY [0..0) OF BOOLEAN;
  nSatisfiedFromDataPool, nCreated, currentUtilization: CARDINAL ← 0; -- usage
  -- statistics.

  InitVMMgrStore: PROCEDURE =
    BEGIN
    volSize, freePages, desiredBackFileSize: File.PageCount;
    systemVolume ← Volume.SystemID[];
    [volSize, freePages, ] ← Volume.GetAttributes[volume: systemVolume];
    desiredBackFileSize ← MAX[
      minBackFileSize, MIN[maxCountDataPool, volSize/16, freePages/10]];
    backingFile ← KernelFile.GetRootFile[
      PilotFileTypes.tVMBackingFile, systemVolume];
    BEGIN
    backFileSize ← File.GetSize[backingFile ! File.Unknown => GO TO Create];
    IF backFileSize < minBackFileSize THEN
      File.SetSize[backingFile, backFileSize ← minBackFileSize];
    EXITS
      Create => {
        backingFile ← File.Create[
          systemVolume, backFileSize ← desiredBackFileSize,
          PilotFileTypes.tVMBackingFile];
        File.MakePermanent[backingFile];
        }
    END;
    IF backFileSize > LAST[VM.PageCount] THEN ERROR Bug[bug0];
    countDataPool ← Inline.LowHalf[backFileSize] - countLog - countSpaceBTrees;
    -- initialize Data Pool:
    BEGIN
    size: CARDINAL =
      (countDataPool + Environment.bitsPerWord - 1)/Environment.bitsPerWord;
    node: Environment.Base RELATIVE POINTER;
    s: Zone.Status;
    k: CARDINAL;
    [node, s] ← ResidentHeap.MakeNode[size];
    IF s ~= okay THEN ERROR Bug[bug1];
    pageDataPoolAvailable ← @Environment.first64K[node];
    IF maxCountDataPool > LAST[Utilities.BitIndex] THEN ERROR Bug[bug2];
    -- restriction of Mesa 6.
    FOR k IN [0..countDataPool) DO pageDataPoolAvailable[k] ← TRUE ENDLOOP;
    countThreshold ← countDataPool/fractionThreshold;
    END;
    END;

  AllocateWindow: PUBLIC PROCEDURE [
    pWindowResult: LONG POINTER TO Space.WindowOrigin, count: VM.PageCount] =
    BEGIN
    offsetPage: VM.PageCount; -- offset of first free page of current run.
    offsetOffset: VM.PageCount; -- offset of current free page from offsetPage.
    IF count <= countThreshold THEN
```

```
    FOR offsetPage ← 0, offsetOffset + 1 WHILE offsetPage + count <=
      countDataPool DO
      -- scope of HoleTooSmall --
      BEGIN
      FOR offsetOffset IN [offsetPage..offsetPage + count) DO
        -- search for big enough run of free pages..
        IF ~pageDataPoolAvailable[offsetOffset] THEN GO TO HoleTooSmall;
        -- (value of offsetOffset must survive loop exit.)

        ENDLOOP;
      FOR offsetOffset IN [offsetPage..offsetPage + count) DO
        -- mark run busy..
        pageDataPoolAvailable[offsetOffset] ← FALSE;
        ENDLOOP;
      nSatisfiedFromDataPool ← nSatisfiedFromDataPool + 1;
      -- tally usage statistics.
      currentUtilization ← currentUtilization + count;
      pWindowResult↑ ←
        [File.LimitPermissions[backingFile, File.read + File.write], LONG[
          pageDataPool + offsetPage]];
      RETURN;
      EXITS HoleTooSmall => NULL;
      END;
      ENDLOOP;
  -- contiguous space not available from Data Pool. Allocate a new file:
  nCreated ← nCreated + 1; -- tally usage statistics.
  pWindowResult↑ ←
    [File.Create[systemVolume, count, PilotFileTypes.tAnonymousFile], 0];
  END;

DeallocateWindow: PUBLIC PROCEDURE [
  pWindow: LONG POINTER TO Space.WindowOrigin, count: VM.PageCount] =
  BEGIN OPEN pWindow --USING[file, base]--;
  IF file.fID = backingFile.fID THEN
    BEGIN
    offsetPage, offsetOffset: VM.PageOffset;
    offsetPage ← Inline.LowHalf[base] - pageDataPool;
    IF base > LAST[VM.PageOffset] OR offsetPage ~IN [0..countDataPool) OR
      offsetPage + count ~IN [0..countDataPool] THEN ERROR Bug[bug3];
    FOR offsetOffset IN [offsetPage..offsetPage + count) DO
      IF pageDataPoolAvailable[offsetOffset] THEN ERROR Bug[bug4];
      pageDataPoolAvailable[offsetOffset] ← TRUE;
      ENDLOOP;
    currentUtilization ← currentUtilization - count;
    END
  ELSE File.Delete[file];
  END;

AllocateMapLogFile: PUBLIC PROCEDURE [
  pWindowResult: LONG POINTER TO Space.WindowOrigin]
  RETURNS [size: VM.PageCount] = {
  pWindowResult↑ ← Space.WindowOrigin[backingFile, pageLog]; RETURN[countLog]};

--
-- STLeaf

SpaceBTreeOverflow: PRIVATE --PROGRAMMING--ERROR = CODE;
spaceBTree: Space.Handle;
```



```
  pageBTree: Space.PageNumber; -- the starting page of the BTree area.
  leafAvailable: PACKED ARRAY CountSpaceBTreesRange OF BOOLEAN;
  InitSTLeaf: PROCEDURE =
    BEGIN
    IF countSpaceBTrees > countSpaceBTreesMax THEN ERROR;
    -- in case runtime checks are off
    leafAvailable ← ALL[TRUE];
    spaceBTree ← SimpleSpace.Create[
      countSpaceBTrees, hyperspace, --sizeSwapUnit:--1];
    SimpleSpace.Map[
      handle: spaceBTree, window: [backingFile, pageSpaceBTrees], andPin: FALSE];
    pageBTree ← SimpleSpace.Page[spaceBTree];
    END;

  Close: PUBLIC PROCEDURE [iLeaf: STLeaf.ILeaf] = BEGIN END;
  -- this procedure is now functionless, and therefore obsolete.

  Create: PUBLIC PROCEDURE RETURNS [iLeaf: STLeaf.ILeaf] =
    BEGIN
    k: CARDINAL;
    FOR k IN [0..countSpaceBTrees) DO
      IF leafAvailable[k] THEN {
        leafAvailable[k] ← FALSE; RETURN[STLeaf.ILeaf[k]]};
      ENDLOOP;
    ERROR SpaceBTreeOverflow;
    END;

  Delete: PUBLIC PROCEDURE [iLeaf: STLeaf.ILeaf] =
    BEGIN
    IF leafAvailable[iLeaf] THEN ERROR Bug[bug5];
    leafAvailable[iLeaf] ← TRUE;
    END;

  Open: PUBLIC PROCEDURE [iLeaf: STLeaf.ILeaf] RETURNS [STLeaf.PLeaf] =
    BEGIN
    IF leafAvailable[iLeaf] THEN ERROR Bug[bug6];
    RETURN[StoragePrograms.LongPointerFromPage[pageBTree + iLeaf]];
    END;

  --
  -- Initialization

  IF PilotSwitches.switches.v = down THEN Runtime.CallDebugger["Key Stop V"L];
  InitVMMgrStore[]; -- 1 of 2.
  InitSTLeaf[]; -- 2 of 2.

  END.
```

May 16, 1978 11:29 AM	*McJones*	*Created file*
June 20, 1978 2:26 PM	*McJones*	*Added SimpleSpace calls*
June 21, 1978 10:31 AM	*McJones*	*Open didn't set rgBuff.open*
June 21, 1978 10:46 AM	*McJones*	*Close didn't stop searching when buffer found*
June 26, 1978 5:43 PM	*McJones*	*Added iBuffMru hack*
August 2, 1978 10:09 AM	*McJones*	*Enabled File.SetSize calls*
August 9, 1978 1:42 AM	*Purcell*	*Use new SimpleSpace*
September 6, 1978 9:51 AM	*McJones*	*Use GetRootFile*
September 29, 1978 10:52 AM	*McJones*	*CR20.42: Replaced MakePermanent with PutRootFile.*

August 8, 1979 4:30 PM	McJones	SpecialFile => KernelFile
September 4, 1979 10:02 AM	Forrest	Changed to use PilotFileTYpes
September 6, 1979 3:07 PM	Ladner	Installed instrumentation in leaf cache
November 26, 1979 5:50 PM	Knutsen	Added VMMgrStore implementation.
November 27, 1979 12:41 PM	Knutsen	InitVMMgrStore forgot to set backFileSize sometimes.
Approx May 15, 1980	Forrest	Converted to Mesa 6.
May 29, 1980 3:20 PM	Knutsen	Use VM Backing File for Hierarchy and Projection. Use packed booleans for bit operations. Named the ERRORs. Moved Key Stop V here from VMMControl.
June 3, 1980 5:36 PM	Knutsen	Don't use VM Backing File for large spaces.
July 30, 1980 2:04 PM	Knutsen	Change MapLog from 20 to 40 pages.
August 15, 1980 5:39 PM	McJones	Add " = CODE" to Bug
August 16, 1980 4:26 PM	McJones	Change countSpaceBTrees from 40 to 60
September 15, 1980 2:46 PM	Fay	Change countSpaceBTrees from 60 to 150, and make variable.
September 16, 1980 5:57 PM	Knutsen	Add forgotten = CODE to ERROR.

```
-- VMMgr>STreeImpl.mesa (last edited by Knutsen on July 30, 1980 4:05 PM)
-- Things to consider:
-- 1) Use LongMove for root insert/delete
-- 2) Collapse merge/balance logic
-- 3) For multi-mds, remove variables from global frame
-- 4) Move cDesc from leaves to root; improve merge/balance logic
-- 5) Some calls to LongMove could be LongCOPY

DIRECTORY
  CachedSpace USING [handleVM, Level],
  Environment USING [Base, first64K],
  Inline USING [LongCOPY, LowHalf],
  ResidentHeap USING [FreeNode, MakeNode],
  Space USING [defaultWindow],
  STLeaf,
  STree,
  Transaction USING [nullHandle],
  Utilities USING [LongMove],
  VM USING [PageCount],
  Zone USING [Status];

STreeImpl: PROGRAM [countVM: VM.PageCount, kind: STLeaf.Kind]
  -- logically, this program should be a monitor, but each instance of it has only one client (Hierarc
**hyImpl or ProjectionImpl) and is protected by their monitor locks.
  IMPORTS Inline, ResidentHeap, STLeaf, Transaction, Utilities EXPORTS STree =
  BEGIN OPEN STLeaf, STree;
  sizeDesc: CARDINAL; -- SIZE[kind Desc]
  cDesc: INTEGER;
  -- total number of descriptors in the STree (for measurements only)
  --
  -- Keys
  keyTop: Key; -- upper bound for kind Key
  KeyFromPDesc: PROCEDURE [pDesc: PDesc] RETURNS [Key] =
    BEGIN
    WITH pDesc SELECT kind FROM
      hierarchy =>
        RETURN[[hierarchy[[level: descH.level, page: descH.interval.page]]]];
      ENDCASE -- projection -- => RETURN[[projection[pDesc.page]]]
    END;
  --
  -- The root

  -- If cKey = 0, then mpIlLeafKey has no entries; rgILeaf[0] specifies the only leaf page.
  -- If cKey>0, then the values mpIlLeafKey[i], for i IN [1..cKey], are ascending:
  --       leaf rgILeaf[0] contains descriptors matching keys<mpIlLeafKey[1];
  --       leaf rgILeaf[i] contains descriptors matching keys IN [mpIlLeafKey[i]..mpIlLeafKey[i + 1
**]), i IN [1..cKey);
  --       leaf rgILeaf[cKey] contains descriptors matching keys> = mpIlLeafKey[cKey].
  -- The two arrays making up the root are stored in a single ResidentHeap (really only needs to be
**"ResidentDescriptorHeap") node, with mpIlLeafKey preceding rgILeaf.

  IlLeaf: TYPE = CARDINAL; -- Index of ILeaf entry in root
  MpIlLeafKey: TYPE = LONG POINTER TO ARRAY IlLeaf (0..0] OF Key;
  -- maxlength is cKeyMax
  RgILeaf: TYPE = LONG POINTER TO ARRAY IlLeaf [0..0] OF ILeaf;
  -- maxlength is cKeyMax + 1
  cKeyMax: CARDINAL;
  cKey: CARDINAL;
```



```
  mpIlLeafKey: MpIlLeafKey;
  rgILeaf: RgILeaf;
  rootGrowthTenths: CARDINAL ← 5; -- see ExpandRoot
  Bug: PRIVATE --PROGRAMMING--ERROR [type: BugType] = CODE;
  BugType: TYPE = {bug0, bug1};
  AllocateRoot: PROCEDURE [cKeyMax: CARDINAL]
    RETURNS [mpIlLeafKey: MpIlLeafKey, rgILeaf: RgILeaf] =
    BEGIN
    sizeMpIlLeafKey: CARDINAL = cKeyMax*SIZE[Key];
    node: Environment.Base RELATIVE POINTER;
    s: Zone.Status;
    [node, s] ← ResidentHeap.MakeNode[
      sizeMpIlLeafKey + (cKeyMax + 1)*SIZE[ILeaf]];
    IF s ~= okay THEN ERROR Bug[bug0];
    mpIlLeafKey ← @Environment.first64K[node];
    rgILeaf ← @Environment.first64K[node + sizeMpIlLeafKey];
    END;

  ExpandRoot: PROCEDURE =
    BEGIN
    s: Zone.Status;
    cKeyMaxNew: CARDINAL =
      cKeyMax + (cKeyMax*rootGrowthTenths + 9 --round up--)/10;
    mpIlLeafKeyNew: MpIlLeafKey;
    rgILeafNew: RgILeaf;
    [mpIlLeafKeyNew, rgILeafNew] ← AllocateRoot[cKeyMaxNew];
    Inline.LongCOPY[
      from: mpIlLeafKey, nwords: cKeyMax*SIZE[Key], to: mpIlLeafKeyNew];
    Inline.LongCOPY[
      from: rgILeaf, nwords: (cKeyMax + 1)*SIZE[ILeaf], to: rgILeafNew];
    s ← ResidentHeap.FreeNode[Inline.LowHalf[mpIlLeafKey]];
    IF s ~= okay THEN ERROR;
    mpIlLeafKey ← mpIlLeafKeyNew;
    rgILeaf ← rgILeafNew;
    cKeyMax ← cKeyMaxNew
    END;

  SearchRoot: PROCEDURE [key: Key] RETURNS [ilLeaf: IlLeaf] =
    BEGIN
    FOR ilLeaf ← 0, ilLeaf + 1 DO
      IF ilLeaf = cKey OR
        (WITH mpIlLeafKey[ilLeaf + 1] SELECT kind FROM
          hierarchy => key.handleH.level < handleH.level OR key.handleH.level =
            handleH.level AND key.handleH.page < handleH.page,
          ENDCASE -- projection -- => key.pageP < pageP) THEN EXIT
      ENDLOOP
    END;

  UpdateRoot: PROCEDURE [ilLeaf: IlLeaf, pLeaf: PLeaf] =
    BEGIN
    IF ilLeaf ~= 0 AND pLeaf.cDesc ~= 0 THEN
      mpIlLeafKey[ilLeaf] ← KeyFromPDesc[@pLeaf.descFirst]
    END;
  --
  -- Leaves

  cDescMax: CDesc; -- maximum number of descriptors per leaf
  cDescMin: CDesc; -- threshold for balancing
  SearchLeaf: PROCEDURE [key: Key, pLeaf: PLeaf]
```

```
  RETURNS [found: BOOLEAN, pDesc: PDesc, last: BOOLEAN] =
  -- Returns last = TRUE if the descriptor coming after given key is not on this leaf
  BEGIN
  iDesc: CARDINAL;
  pDescNext: PDesc;
  --assert--
  IF pLeaf.cDesc = 0 THEN ERROR;
  found ← TRUE;
  last ← FALSE;
  pDesc ← @pLeaf.descFirst;
  FOR iDesc IN [0..pLeaf.cDesc - (IF kind = projection THEN 1 ELSE 0)) DO
    pDescNext ← pDesc + sizeDesc;
    WITH pDesc SELECT kind FROM
      hierarchy =>
        BEGIN OPEN descH;
        IF key.handleH.level = level AND key.handleH.page <
          interval.page + interval.count OR key.handleH.level < level THEN
          BEGIN
          IF key.handleH.level ~= level OR key.handleH.page < interval.page THEN
            found ← FALSE
          ELSE IF iDesc + 1 = pLeaf.cDesc THEN last ← TRUE;
          EXIT
          END
        END;
      projection => IF key.pageP < pDescNext.page THEN EXIT;
      ENDCASE;
    pDesc ← pDescNext
    REPEAT
      FINISHED => BEGIN last ← TRUE; IF kind = hierarchy THEN found ← FALSE END
    ENDLOOP
  END;
--
-- STree's

Delete: PUBLIC PROCEDURE [key: Key] =
  BEGIN
  ilLeaf: IlLeaf = SearchRoot[key];
  pLeaf: PLeaf = STLeaf.Open[rglLeaf[ilLeaf]];
  matching: BOOLEAN;
  pDesc: PDesc;
  [matching, pDesc] ← SearchLeaf[key, pLeaf];
  --assert--
  IF ~matching THEN ERROR;
  -- Move descriptors following pDesc up one place
  pLeaf.cDesc ← pLeaf.cDesc - 1;
  Utilities.LongMove[
    pSource: pDesc + sizeDesc,
    size: Inline.LowHalf[@pLeaf.descFirst + sizeDesc*pLeaf.cDesc - pDesc],
    pSink: pDesc];
  -- Update root entry in case deleted descriptor was first on leaf
  UpdateRoot[ilLeaf, pLeaf];
  -- If number of descriptors remaining on leaf is small enough, perform merge or balance
  IF pLeaf.cDesc = 0 AND ilLeaf = cKey THEN
    BEGIN
    -- Delete last leaf
    STLeaf.Close[rglLeaf[ilLeaf]];
    STLeaf.Delete[rglLeaf[ilLeaf]];
    cKey ← cKey - 1;
    RETURN
    END;
```



```
  IF pLeaf.cDesc < cDescMin AND ilLeaf < cKey THEN
    BEGIN
    ilLeafNext: IlLeaf = ilLeaf + 1;
    pLeafNext: PLeaf = STLeaf.Open[rglLeaf[ilLeafNext]];
    IF pLeaf.cDesc + pLeafNext.cDesc <= cDescMax THEN
      -- Merge higher neighbor into this leaf
      BEGIN
      ilLeafAfter: IlLeaf;
      -- Append contents of merged leaf
      Utilities.LongMove[
        pSource: @pLeafNext.descFirst, size: sizeDesc*pLeafNext.cDesc,
        pSink: @pLeaf.descFirst + sizeDesc*pLeaf.cDesc];
      pLeaf.cDesc ← pLeaf.cDesc + pLeafNext.cDesc;
      -- Delete merged leaf
      STLeaf.Close[rglLeaf[ilLeafNext]];
      STLeaf.Delete[rglLeaf[ilLeafNext]];
      -- Delete its root entry
      cKey ← cKey - 1;
      FOR ilLeafAfter IN [ilLeafNext..cKey] DO
        mpIlLeafKey[ilLeafAfter] ← mpIlLeafKey[ilLeafAfter + 1];
        rglLeaf[ilLeafAfter] ← rglLeaf[ilLeafAfter + 1]
        ENDLOOP
      END
    ELSE
      -- Balance with next leaf
      BEGIN
      cDescDelta: CDesc = (pLeafNext.cDesc - pLeaf.cDesc)/2;
      -- >0, or we would have merged
      -- Append first cDescDelta descriptors from next leaf to this one
      Utilities.LongMove[
        pSource: @pLeafNext.descFirst, size: sizeDesc*cDescDelta,
        pSink: @pLeaf.descFirst + sizeDesc*pLeaf.cDesc];
      pLeaf.cDesc ← pLeaf.cDesc + cDescDelta;
      -- Delete first cDescDelta descriptors of next leaf
      pLeafNext.cDesc ← pLeafNext.cDesc - cDescDelta;
      Utilities.LongMove[
        pSource: @pLeafNext.descFirst + sizeDesc*cDescDelta,
        size: sizeDesc*pLeafNext.cDesc, pSink: @pLeafNext.descFirst];
      -- Update root entry since identity of first descriptor changed
      UpdateRoot[ilLeafNext, pLeafNext];
      STLeaf.Close[rglLeaf[ilLeafNext]]
      END
    END;
  STLeaf.Close[rglLeaf[ilLeaf]];
  cDesc ← cDesc - 1
  END;

Get: PUBLIC PROCEDURE [pDescResult: PDesc, key: Key] RETURNS [keyNext: Key] =
  BEGIN
  ilLeaf: IlLeaf = SearchRoot[key];
  pLeaf: PLeaf = STLeaf.Open[rglLeaf[ilLeaf]];
  found, last: BOOLEAN;
  pDesc: PDesc;
  [found, pDesc, last] ← SearchLeaf[key, pLeaf];
  IF ~found THEN
    WITH pDescResult SELECT kind FROM
      hierarchy => descH.state ← missing;
      projection => ERROR;
      ENDCASE
  ELSE Utilities.LongMove[pSource: pDesc, size: sizeDesc, pSink: pDescResult];
```

```
    SELECT TRUE FROM
      ~last => keyNext ← KeyFromPDesc[pDesc + (IF found THEN sizeDesc ELSE 0)];
      -- last AND --
      ilLeaf < cKey => keyNext ← mpIlLeafKey[ilLeaf + 1];
      -- last AND ilLeaf = cKey --
      ENDCASE => keyNext ← keyTop;
    STLeaf.Close[rgILeaf[ilLeaf]]
    END;

  Insert: PUBLIC PROCEDURE [pDesc: PDesc] =
    BEGIN
    key: Key = KeyFromPDesc[pDesc];
    ilLeaf: IlLeaf;
    pLeaf: PLeaf;
    inserted: BOOLEAN;
    DO
      ilLeaf ← SearchRoot[key];
      pLeaf ← STLeaf.Open[rgILeaf[ilLeaf]];
      IF pLeaf.cDesc = cDescMax THEN
        -- Leaf is full; split it and try again
        BEGIN
        ilLeafNew: IlLeaf = ilLeaf + 1;
        ilLeafAfter: IlLeaf;
        -- Create new leaf to follow current one
        iLeafNew: ILeaf = STLeaf.Create[];
        pLeafNew: PLeaf = STLeaf.Open[iLeafNew];
        cDescNew: CDesc = pLeaf.cDesc/2;
        -- Move last cDescNew descriptors of this leaf to new one
        pLeaf.cDesc ← pLeaf.cDesc - cDescNew;
        pLeafNew.cDesc ← cDescNew;
        Utilities.LongMove[
          pSource: @pLeaf.descFirst + sizeDesc*pLeaf.cDesc,
          size: sizeDesc*cDescNew, pSink: @pLeafNew.descFirst];
        -- Insert new root entry
        IF cKey = cKeyMax THEN ExpandRoot[];
        FOR ilLeafAfter DECREASING IN [ilLeafNew..cKey] DO
          mpIlLeafKey[ilLeafAfter + 1] ← mpIlLeafKey[ilLeafAfter];
          rgILeaf[ilLeafAfter + 1] ← rgILeaf[ilLeafAfter]
          ENDLOOP;
        cKey ← cKey + 1;
        rgILeaf[ilLeafNew] ← iLeafNew;
        UpdateRoot[ilLeafNew, pLeafNew];
        STLeaf.Close[iLeafNew];
        -- Restart
        inserted ← FALSE
        END
      ELSE
        -- Leaf is not full; perform insert and exit
        BEGIN
        matching: BOOLEAN;
        pDescTarget: PDesc;
        [matching, pDescTarget] ← SearchLeaf[key, pLeaf];
        SELECT kind FROM
          hierarchy => IF matching THEN ERROR;
          projection => pDescTarget ← pDescTarget + sizeDesc;
          -- insert after current matching entry

          ENDCASE;
        -- Move all descriptors not before pDescTarget down one place
```



```
        Utilities.LongMove[
          pSource: pDescTarget,
          size: Inline.LowHalf[
          @pLeaf.descFirst + sizeDesc*pLeaf.cDesc - pDescTarget],
          pSink: pDescTarget + sizeDesc];
        -- Insert descriptor
        Utilities.LongMove[pSource: pDesc, size: sizeDesc, pSink: pDescTarget];
        pLeaf.cDesc ← pLeaf.cDesc + 1;
        -- Update root entry in case inserted descriptor is first on leaf
        UpdateRoot[ilLeaf, pLeaf];
        -- Indicate finished
        inserted ← TRUE
        END;
      STLeaf.Close[rgILeaf[ilLeaf]];
      IF inserted THEN EXIT
      ENDLOOP;
    cDesc ← cDesc + 1
    END;

  Update: PUBLIC PROCEDURE [pDesc: PDesc] =
    BEGIN
    key: Key = KeyFromPDesc[pDesc];
    ilLeaf: IlLeaf = SearchRoot[key];
    pLeaf: PLeaf = STLeaf.Open[rgILeaf[ilLeaf]];
    matching: BOOLEAN;
    pDescTarget: PDesc;
    [matching, pDescTarget] ← SearchLeaf[key, pLeaf];
    --assert--
    IF ~matching THEN ERROR;
    -- Update descriptor
    Utilities.LongMove[pSource: pDesc, size: sizeDesc, pSink: pDescTarget];
    -- Update root entry in case updated descriptor is first on leaf
    UpdateRoot[ilLeaf, pLeaf];
    STLeaf.Close[rgILeaf[ilLeaf]];
    END;

  Initialize: PROCEDURE =
    BEGIN
    iLeaf: ILeaf = STLeaf.Create[];
    pLeaf: PLeaf = STLeaf.Open[iLeaf];
    cDesc ← 1;
    pLeaf.cDesc ← 1;
    SELECT kind FROM
      hierarchy =>
        BEGIN
        sizeDesc ← SIZE[hierarchy Desc];
        keyTop ←
          [hierarchy[
            [level: LAST[CachedSpace.Level],
              page: CachedSpace.handleVM.page + countVM]]];
        cKeyMax ← 25;
        pLeaf.descFirst ←
          [hierarchy[
            [level: CachedSpace.handleVM.level,
              interval: [CachedSpace.handleVM.page, countVM], dPinned: FALSE,
              -- don't care
              dDirty: FALSE, -- don't care
              pinned: FALSE, state: unmapped, writeProtected:, -- don't care
```

```
            hasSwapUnits: FALSE, sizeSwapUnit: 1, -- don't care
            dataOrFile:, -- don't care
            pageRover: CachedSpace.handleVM.page,
            vp: long[
            window: Space.defaultWindow, -- don't care
            countMapped: 0, -- don't care
            transaction: Transaction.nullHandle]]];
        -- not part of a transaction yet.

        END;
      projection =>
        BEGIN
        sizeDesc ← SIZE[projection Desc];
        keyTop ← [projection[CachedSpace.handleVM.page + countVM]];
        cKeyMax ← 2;
        pLeaf.descFirst ←
          [projection[
            page: CachedSpace.handleVM.page, level: CachedSpace.handleVM.level,
            levelMapped: FIRST[CachedSpace.Level], -- don't care
            hasSwapUnits: FALSE, state: unmapped, writeProtected: FALSE,
            -- don't care
            needsLogging: FALSE]] -- don't care

        END;
      ENDCASE => ERROR Bug[bug1];
    STLeaf.Close[iLeaf];
    [mpilLeafKey, rgiLeaf] ← AllocateRoot[cKeyMax];
    cKey ← 0;
    rgiLeaf[0] ← iLeaf;
    cDescMax ← sizeLeafBody/sizeDesc;
    cDescMin ← cDescMax/3; -- must be < cDescMax/2 for merge/balance logic

    END;

  Initialize[];
  END.
LOG
Time: May 5, 1978 2:55 PM By: McJones Action: Create file
Time: June 22, 1978 4:06 PM By: McJones Action: keyTop.page was one too large (bot
**h kinds)
Time: June 26, 1978 1:06 PM By: McJones Action: projection Insert put new entry befor
**e old matching entry
Time: July 17, 1978 12:06 PM By: McJones Action: Upper bound for split root update loo
**p was ) instead of ]; Delete allowed last page to become empty; Delete wouldn't merge second-
**to-last page; Upper bound for merge root update loop was ) instead of ];
        Balance updated wrong root entry
Time: August 31, 1978 11:59 AM By: McJones Action: Bugs in Get's keyNext co
**mputation
Time: September 9, 1978 5:02 PM By: McJones Action: Added pinned to CachedS
**pace constructor
Time: September 22, 1978 6:33 PM By: McJones Action: SearchLeaf didn't set last
**when found = TRUE
Time: February 1, 1979 2:24 PM By: McJones Action: pageRover in CachedSpa
**ce.Desc
Time: September 6, 1979 11:10 AM By: McJones Action: Dynamic root expansion
Time: October 1, 1979 9:56 AM By: Knutsen Action: Made compatible with new CachedR
**egion.Desc and CachedSpace.Desc.
Time: July 7, 1980 4:32 PM By: Gobbel Action: Made compatible with new CachedR
```



```
**egion.Desc and CachedSpace.Desc.
Time: July 30, 1980 4:05 PM By: Gobbel Action: Made compatible with new CachedR
**egion.Desc. Initial Hierarchy entry for VM must have Transaction.nullHandle.
```

```
-- VMMgr>VMMControl.mesa (last edited by Knutsen on July 1, 1980 10:44 AM)
-- Packing Considerations: IsUtilityPilot[], IsDiagnosticPilot must be initially Swapped In.

DIRECTORY
  CachedRegion USING [Apply, Desc, Outcome, pageTop],
  CachedSpace USING [Desc, Get, Handle, Level],
  Environment USING [maxPagesInMDS, PageCount, PageNumber],
  File USING [nullID],
  Hierarchy USING [Insert],
  MapLog USING [WriteLog],
  Projection USING [Split, TranslateLevel],
  SimpleSpace USING [DisableInitialization],
  StoragePrograms USING [pageMDS],
  STree USING [STreeImpl],
  VM USING [Interval],
  VMMPrograms;

VMMControl: PROGRAM
  IMPORTS
    CachedRegion, CachedSpace, Hierarchy, MapLog, Projection, SimpleSpace,
    StoragePrograms, STreeHier: STree, STreeProj: STree, VMMPrograms
  EXPORTS StoragePrograms
  SHARES File =
  BEGIN
  mds: CachedSpace.Handle =
    [level: FIRST[CachedSpace.Level] + 1, page: StoragePrograms.pageMDS];
  levelVM: CachedSpace.Level ← FIRST[CachedSpace.Level];
  levelMDS: CachedSpace.Level ← levelVM + 1; -- MDS is child of VM.
  Bug: PRIVATE --PROGRAMMING--ERROR [type: BugType] = CODE;
  BugType: TYPE = {bug0, bug1};
  InitializeVMMgr: PUBLIC PROCEDURE [
    countVM: Environment.PageCount, pMapLogDesc: LONG POINTER] =
    BEGIN
    START VMMPrograms.STLeafImpl[];
    START VMMPrograms.MapLogImpl[pMapLogDesc: pMapLogDesc];
    START STreeHier.STreeImpl[countVM: countVM, kind: hierarchy];
    START STreeProj.STreeImpl[countVM: countVM, kind: projection];
    START VMMPrograms.HierarchyImpl;
    START VMMPrograms.ProjectionImpl[countVM: countVM];
    FillHierarchyAndProjection[];
    VMMPrograms.InitializeSpace[countVM: countVM, handleMDS: mds];
    END;

  IsDiagnosticPilot, IsUtilityPilot: PUBLIC PROCEDURE RETURNS [BOOLEAN] =
    BEGIN RETURN[FALSE] END;
  -- See comments with DescribeSpace in StoragePrograms.mesa!
  -- This procedure takes the space and region cache entries created by PilotControl.DescribeIniti
**allyResidentSpaces and SimpleSpaceImpl.DescribeSpace, and makes corresponding entries in
**the Hierarchy and Projection databases.
  -- When the Hierarchy and Projection are created, they each have a single entry representing all
**of VM as a single unmapped space. As new spaces and regions are created below, they inherit
**the properties of their parents i.e. they are unmapped. In the VMMgr however, the algorithm for
**accessing a Hierarchy or Projection entry is: look in the cache; if the entry is there, it is the truth
**; if it is not there, the entry in the Hierarchy or Projection is the truth. The code below demands
**that there already be an entry in the space cache for every unitary or family space (i.e. the ones
**that are (themselves) mapped) and an entry in the region cache for every swapUnit of every unit
**ary or family space. Therefore, we do not need to make the entries in the Hierarchy and Projecti
**on contain the proper attributes.
```



```
  FillHierarchyAndProjection: PROCEDURE =
    BEGIN
    pageRegion: Environment.PageNumber;
    outcome: CachedRegion.Outcome;
    region: CachedRegion.Desc;
    space: CachedSpace.Desc; -- the desc of the current unitary or family space.
    SimpleSpace.DisableInitialization[]; -- no more simpleSpaces may be created.
    CreateSpace[mds.level, VM.Interval[mds.page, Environment.maxPagesInMDS], NIL];
    -- create MDS space.
    space.interval.count ← 0;
    -- there is no current space. (No page is contained in our current space.)
    pageRegion ← FIRST[Environment.PageNumber]; -- From the beginning of VM..
    WHILE pageRegion < CachedRegion.pageTop DO
      --until all regions in cache processed--
      [outcome, pageRegion] ← CachedRegion.Apply[
        -- What's next in the region cache?
        pageRegion,
        [ifMissing: report, ifCheckedOut: wait, afterForking:,
          vp: get[andResetDDirty: FALSE, pDescResult: @region]]];
      SELECT outcome.kind FROM
        ok => -- we got the descriptor of the next region in the cache --
          BEGIN
          IF region.interval.page ~IN
            [space.interval.page..space.interval.page + space.interval.count) THEN
            BEGIN -- this region is in a new primary space --
            CachedSpace.Get[
              @space, [level: region.levelMapped, page: region.interval.page]];
            -- get the desc of the unitary or family space which contins this region.
            IF region.interval.page ~= space.interval.page THEN ERROR Bug[bug0];
            -- primary space not completely tiled with regions.
            CreateSpace[
              space.level, space.interval,
              IF space.state = mapped THEN @space ELSE NIL];
            -- put unitary or family space into VMMgr database.

            END;
          IF region.interval ~= space.interval THEN
            CreateSpace[region.level, region.interval, NIL];
          -- put swap unit of family space into VMMgr database.

          END;
        regionDMissing => NULL;
        -- this region is unused. Proceed to the next region boundary.

        ENDCASE => ERROR Bug[bug1];
      ENDLOOP;
    END;

  CreateSpace: PROCEDURE [
    level: CachedSpace.Level, interval: VM.Interval,
    pSpaceD: POINTER TO CachedSpace.Desc] =
    -- Creates Hierarchy & Projection entries for one space; dirty cache entries should already exi
**st if the space or region is mapped.
    BEGIN OPEN i: interval;
    Hierarchy.Insert[CachedSpace.Handle[level, i.page], i.count];
    Projection.Split[i.page];
    Projection.Split[i.page + i.count];
    Projection.TranslateLevel[pageMember: i.page, delta: 1];
```

```
    IF pSpaceD ~= NIL AND pSpaceD.window.file.fID ~= File.nullID THEN
      MapLog.WriteLog[interval, pSpaceD];
    END;

  END.
```

LOG

Time: June 20, 1978 5:58 PM By: McJones Action: Created file
Time: June 22, 1978 6:12 PM By: McJones Action: Added projection(/hierarchy?) init
Time: June 23, 1978 1:23 PM By: McJones Action: Added countVM parameter to Projec
***tionImpl, SpaceImpl*
Time: July 10, 1978 10:18 AM By: McJones Action: Added creation of first64K
Time: July 18, 1978 10:59 AM By: McJones Action: Added countReal, ...
Time: August 2, 1978 10:18 AM By: McJones Action: Added MapLogImpl
Time: August 7, 1978 8:11 PM By: Purcell Action: Call SimpleSpace.DisableInitializatio
***n; new Hierarchy/Projection filling logic*
Time: August 29, 1978 4:14 PM By: McJones Action: (De)typed pMapLogDesc
Time: September 6, 1978 10:18 AM By: McJones Action: Fixed test for first subspac
***e of parent*
Time: September 15, 1978 5:20 PM By: McJones Action: Added map logging
Time: July 11, 1979 3:28 PM By: McJones Action: Nonparent mapped space handled i
***ncorrectly*
Time: July 31, 1979 2:03 PM By: McJones Action: Changed MapLog to include writePr
***otected*
Time: August 29, 1979 3:40 PM By: Knutsen Action: Changed VMMControl: PROGRAM to
***InitVMMgr: PROCEDURE.*
Time: September 7, 1979 8:50 AM By: McJones Action: Added V key stop
Time: November 26, 1979 3:53 PM By: Knutsen Action: Deleted volumeData para
***meter of InitSpace.*
Time: April 10, 1980 4:35 PM By: Knutsen Action: Changes for nonzero MDS. Changed
***VMMControl: PROGRAM to InitVMMgr: PROCEDURE AGAIN!*
Time: May 29, 1980 3:18 PM By: Knutsen Action: Moved Key Stop V to STLeafImpl.
Time: July 1, 1980 10:44 AM By: Knutsen Action: Minor cleanups.

```
-- BootSwapCross.mesa (last edited by: Forrest on: May 16, 1980 2:19 PM)

DIRECTORY
  Boot: FROM "Boot" USING [LP, MDSIndex, pRequest, ReadMDS],
  BootSwap: FROM "BootSwap",
  Frame: FROM "Frame" USING [Free, MyLocalFrame, MyGlobalFrame],
  Inline: FROM "Inline" USING [COPY, LongCOPY],
  Mopcodes: FROM "Mopcodes" USING [zWR],
  PrincOps: FROM "PrincOps" USING [
    FrameHandle, GlobalFrameHandle, Port, TrapLink];

BootSwapCross: PROGRAM
  IMPORTS Boot, Frame, Inline EXPORTS BootSwap SHARES BootSwap =
  BEGIN OPEN BootSwap, PrincOps;

  -- Inter-MDS germ linkage

  Initialize: PUBLIC PROCEDURE [mdsiOther: Boot.MDSIndex] =
    BEGIN OPEN port: LOOPHOLE[Port, PrincOps.Port], Frame;
    InitializeMDS[]; -- set pMon
    -- Set up frame for Procedure and make it pending on Port
    port.dest ← [frame[MyLocalFrame[]]]; -- temporarily connect Port to self
    Procedure[];
    -- Now set up local and global frames with copy in other MDS
    lCross.accesslink ← gCross;
    lCross.pc ← port.frame.pc;
    lCross.returnlink ← TrapLink;
    Free[port.frame];
    port.frame ← lCross;
    Inline.COPY[from: MyGlobalFrame[], nwords: nGCross, to: gCross];
    -- now mustn't set global variables
    Inline.LongCOPY[
      from: lCross, nwords: nLCross + nGCross,
      to: Boot.LP[highbits: mdsiOther, lowbits: lCross]];
    pMon.CrossMDSCall ←
      LOOPHOLE[gCross +
               (@port - LOOPHOLE[MyGlobalFrame[], POINTER TO PrincOps.Port])];
    -- connect CrossMDSCall to Port
    pMon.pcCross ← LOOPHOLE[lCross.pc]
    END;

  InitializeMDS: PUBLIC PROCEDURE =
    BEGIN
    pMon ← LOOPHOLE[Boot.LP[highbits: mdsiGerm, lowbits: lCross - SIZE[Mon]], LONG
      POINTER TO Mon]
    END;

  -- The numbers nGCross and nLCross must be divisible by 4 and not too small!

  nGCross: CARDINAL = 12;
  limit: CARDINAL = LOOPHOLE[Boot.pRequest];
  gCross: GlobalFrameHandle ← LOOPHOLE[limit - (limit MOD 4) - nGCross];
  -- variable to avoid compiler bug
  nLCross: CARDINAL = 8;
  lCross: FrameHandle ← LOOPHOLE[gCross - nLCross];
  -- variable to avoid compiler bug

  pMon: PUBLIC LONG POINTER TO Mon;
```



```
  Port: PORT RETURNS [mdsiOther: Boot.MDSIndex, Dest: PROCEDURE];

  Procedure: PROCEDURE =
    BEGIN
    -- address of this frame must be same in both MDS's; global frames must be "similar"
    mdsiOther: Boot.MDSIndex;
    Dest: PROCEDURE;
    mdsiBack: Boot.MDSIndex;
    DO
      [mdsiOther, Dest] ← Port[];
      [Dest, mdsiBack] ← Save2AndWriteMDS[Dest, Boot.ReadMDS[], mdsiOther];
      Dest[];
      WriteMDS[mdsiBack];
    ENDLOOP
    END;

  Save2AndWriteMDS: PROCEDURE [s1, s2: UNSPECIFIED, mdsi: Boot.MDSIndex]
    RETURNS [t1, t2: UNSPECIFIED] = LOOPHOLE[WriteMDS]; -- t1 = s1 AND t2 = s2
  WriteMDS: PROCEDURE [mdsi: Boot.MDSIndex] = MACHINE CODE
    BEGIN Mopcodes.zWR, 3 END;

  END.
LOG
Time: December 18, 1978 9:30 AM          By: McJones     Action: Created file
Time: May 3, 1980 10:32 AM               By: Forrest     Action: FrameOps = > Frame, Con
**trolDefs = > PrincOps
Time: May 16, 1980 2:20 PM               By: Forrest     Action: Delete Import of BootSwa
**p
```


BootSwapCross.mesa. 16-May-80 14:20:46 2

```
-- FrameImpl.mesa (last edited by: McJones on: August 26, 1980 12:27 PM)

DIRECTORY
  BcdOps USING [BcdBase],
  Frame USING [Alloc, Free, GetReturnFrame, SetReturnLink],
  Inline USING [BITAND, COPY],
  PilotLoadStateFormat USING [ConfigIndex, LoadState],
  PrincOps USING [
    ControlLink, CSegPrefix, EPRange, FrameHandle, GFTIndex, GlobalFrameHandle,
    LastAVSlot, MainBodyIndex, NullFrame, NullLink, NullGlobalFrame, UnboundLink],
  PrincOpsRuntime USING [EmptyGFTItem, FreedGFTItem, GetFrame, GFT, GFTItem],
  Runtime USING [NullProgram, UnboundProcedure, UnNew],
  RuntimeInternal USING [Codebase, FrameSize, loadStatePage],
  RuntimePrograms,
  SDDefs USING [sCopy, SD, sGFTLength, sUnNew],
  System USING [gmtEpoch, GreenwichMeanTime],
  Utilities USING [LongPointerFromPage];

FrameImpl: MONITOR
  IMPORTS Frame, Inline, PrincOpsRuntime, Runtime, RuntimeInternal, Utilities
  EXPORTS Runtime, RuntimeInternal, RuntimePrograms =

  BEGIN OPEN PrincOps, PrincOpsRuntime;

  -- Public SIGNALs and ERRORs:
  InvalidFrame: PUBLIC ERROR [frame: FrameHandle] = CODE;
  InvalidGlobalFrame: PUBLIC ERROR [frame: GlobalFrameHandle] = CODE;
  NoGlobalFrameSlots: PUBLIC SIGNAL [CARDINAL] = CODE;

  gftrover: CARDINAL ← 0; -- okay to start at 0 because incremented before used
  NullProgram: PROGRAM = Runtime.NullProgram;

  InitializeFrames: PUBLIC PROCEDURE =
    -- Module initialization:
    BEGIN SDDefs.SD[SDDefs.sCopy] ← Copy; SDDefs.SD[SDDefs.sUnNew] ← zUnNew; END;
  -- Procedures: (ordered by kind {external, entry, internal}; within a kind, alphabetically)


  -- External Procedures (public and private):

  DeletedFrame: PUBLIC PROC [gfi: GFTIndex] RETURNS [BOOLEAN] = {
    RETURN[GFT[gfi] = FreedGFTItem]; };

  GetCaller: PUBLIC PROC RETURNS [PROGRAM] = {
    RETURN[LOOPHOLE[Frame.GetReturnFrame[].returnlink.frame.accesslink]]; };

  GlobalFrame: PUBLIC PROC [link: UNSPECIFIED] RETURNS [PROGRAM] =
    BEGIN OPEN l: LOOPHOLE[link, ControlLink];
    DO
      IF l = UnboundLink THEN link ← SIGNAL Runtime.UnboundProcedure[link]
      ELSE
        IF l.proc THEN
          RETURN[
            IF l.gfi IN [1..SDDefs.SD[SDDefs.sGFTLength]) THEN LOOPHOLE[GetFrame[
            GFT[l.gfi]], PROGRAM] ELSE NullProgram]
        ELSE
          IF l.indirect THEN link ← l.link↑
          ELSE -- Frame
```



```
          BEGIN
          IF link = 0 THEN RETURN[NullProgram];
          IF ValidGlobalFrame[link] THEN RETURN[link];
          RETURN[
            IF ValidGlobalFrame[l.frame.accesslink] THEN
            LOOPHOLE[l.frame.accesslink, PROGRAM] ELSE NullProgram];
          END;
      ENDLOOP;
  END;

  IsBound: PUBLIC PROC [link: UNSPECIFIED] RETURNS [BOOLEAN] = {
    RETURN[link # UnboundLink AND link # NullLink]; };

  MakeFsi: PUBLIC PROCEDURE [words: CARDINAL] RETURNS [fsi: CARDINAL] =
    BEGIN
    FOR fsi IN [0..LastAVSlot) DO
      IF RuntimeInternal.FrameSize[fsi] >= words THEN RETURN; ENDLOOP;
    RETURN[words]
    END;

  GetBcdTime: PUBLIC PROC RETURNS [System.GreenwichMeanTime] =
    BEGIN
    loadstate: LONG PilotLoadStateFormat.LoadState =
      Utilities.LongPointerFromPage[RuntimeInternal.loadStatePage];
    frame: PrincOps.FrameHandle = Frame.GetReturnFrame[];
    globalFrame: PrincOps.GlobalFrameHandle = frame.accesslink;
    IF globalFrame.copied THEN RETURN[System.gmtEpoch]
    ELSE
      BEGIN
      bcd: BcdOps.BcdBase = loadstate.bcds[
        loadstate.gft[globalFrame.gfi].config].base;
      RETURN[[bcd.version.time]]
      END;
    END;

  GetBuildTime: PUBLIC PROC RETURNS [System.GreenwichMeanTime] =
    BEGIN
    loadstate: LONG PilotLoadStateFormat.LoadState =
      Utilities.LongPointerFromPage[RuntimeInternal.loadStatePage];
    bcd: BcdOps.BcdBase = loadstate.bcds[
      FIRST[PilotLoadStateFormat.ConfigIndex]].base;
    RETURN[[bcd.version.time]];
    END;


  SelfDestruct: PUBLIC PROCEDURE =
    BEGIN
    destructee: PrincOps.FrameHandle = Frame.GetReturnFrame[];
    Frame.SetReturnLink[destructee.returnlink];
    Runtime.UnNew[LOOPHOLE[GlobalFrame[destructee], PROGRAM]];
    Frame.Free[destructee];
    RETURN
    END;

  -- Entry Procedures (public and private):

  -- conceptually, Copy is PUBLIC, but it is accessed via the System Dispatch table.

  Copy: --PUBLIC--ENTRY PROC [old: GlobalFrameHandle]
```

```
  RETURNS [new: GlobalFrameHandle] =
  BEGIN
  linkspace: CARDINAL;
  ok: BOOLEAN;
  codebase: LONG POINTER TO PrincOps.CSegPrefix;
  IF ~ValGlobalFrame[old] THEN RETURN WITH ERROR InvalidGlobalFrame[old];
  codebase ← RuntimeInternal.Codebase[LOOPHOLE[old, PROGRAM]];
  [new, linkspace] ← AllocGlobalFrame[old, codebase];
  new ← new + linkspace;
  new↑ ←
    [gfi:, alloced: TRUE, shared: TRUE, copied: TRUE, started: FALSE,
      trapxfers: FALSE, codelinks: old.codelinks, code: old.code, global:];
  [new.gfi, ok] ← EntGlobalFrame[new, codebase.header.info.ngfi];
  IF ~ok THEN RETURN WITH ERROR NoGlobalFrameSlots[codebase.header.info.ngfi];
  new.code.out ← TRUE; -- cause trap
  new.global[0] ← NullGlobalFrame;
  old.shared ← TRUE;
  IF linkspace # 0 THEN
    Inline.COPY[from: old - linkspace, to: new - linkspace, nwords: linkspace];
  END;

EnterGlobalFrame: PUBLIC ENTRY PROCEDURE [
  frame: GlobalFrameHandle, nslots: CARDINAL] RETURNS [entryindex: GFTIndex] =
  BEGIN
  ok: BOOLEAN;
  [entryindex, ok] ← EntGlobalFrame[frame, nslots];
  IF ~ok THEN RETURN WITH ERROR NoGlobalFrameSlots[nslots];
  END;

GetNextGlobalFrame: PUBLIC ENTRY PROCEDURE [frame: GlobalFrameHandle]
  RETURNS [GlobalFrameHandle] =
  BEGIN
  IF frame # NullGlobalFrame THEN
    IF ~ValGlobalFrame[frame] THEN RETURN WITH ERROR InvalidGlobalFrame[frame];
  RETURN[GetNxGlobalFrame[frame]];
  END;

RemoveGlobalFrame: PUBLIC ENTRY PROCEDURE [frame: GlobalFrameHandle] =
  BEGIN RemvGlobalFrame[frame]; END;

-- conceptually, (z)UnNew is PUBLIC, but it is accessed via the System Dispatch table.

zUnNew: --PUBLIC--ENTRY PROCEDURE [frame: GlobalFrameHandle] =
  BEGIN
  sharer: GlobalFrameHandle ← NullGlobalFrame;
  original: GlobalFrameHandle ← NullGlobalFrame;
  copy, f: GlobalFrameHandle ← NullGlobalFrame;
  codebase: LONG POINTER TO PrincOps.CSegPrefix;
  nothers: CARDINAL ← 0;
  nlinks: CARDINAL;
  IF ~ValGlobalFrame[frame] THEN RETURN WITH ERROR InvalidGlobalFrame[frame];
  codebase ← RuntimeInternal.Codebase[LOOPHOLE[frame, PROGRAM]];
  nlinks ← codebase.header.info.nlinks;
  FOR f ← GetNxGlobalFrame[NullGlobalFrame], GetNxGlobalFrame[f] UNTIL f =
    NullGlobalFrame DO
    IF f # frame THEN
      BEGIN
      IF f.global[0] = frame AND ~f.started THEN f.global[0] ← NullFrame;
      IF RuntimeInternal.Codebase[LOOPHOLE[f, PROGRAM]] = codebase THEN
```



```
        IF f.copied THEN copy ← f ELSE original ← f;
      END;
    ENDLOOP;
  -- to aid debugging, for now we don't delete the original copy because it has the original links.
  IF original = NullGlobalFrame AND ~frame.copied AND copy # NullGlobalFrame
    THEN RETURN WITH ERROR InvalidGlobalFrame[frame];
  -- BEGIN OPEN LoadStateDefs;
  -- config: ConfigIndex;
  -- cgfi: GFTIndex;
  -- copy.copied ← FALSE;
  -- [] ← InputLoadState[];
  -- [cgfi: cgfi, config: config] ← MapRealToConfig[frame.gfi];
  -- EnterGfi[cgfi: 0, rgfi: frame.gfi, config: ConfigNull];
  -- EnterGfi[cgfi: cgfi, rgfi: copy.gfi, config: config];
  -- ReleaseLoadState[];
  -- END;
  RemvGlobalFrame[frame];
  IF frame.alloced THEN
    BEGIN
    Align: PROCEDURE [POINTER, WORD] RETURNS [POINTER] =
      LOOPHOLE[Inline.BITAND];
    IF frame.codelinks THEN Frame.Free[frame]
    ELSE Frame.Free[Align[frame - nlinks, 177774B]];
    END;
  END;

ValidateFrame: PUBLIC ENTRY PROCEDURE [frame: FrameHandle] =
  BEGIN OPEN LOOPHOLE[frame, rep ControlLink];
  IF proc OR indirect OR ~ValGlobalFrame[frame.accesslink] THEN
    RETURN WITH ERROR InvalidFrame[frame];
  END;

ValidateGlobalFrame: PUBLIC ENTRY PROC [g: GlobalFrameHandle] =
  BEGIN IF ~ValGlobalFrame[g] THEN RETURN WITH ERROR InvalidGlobalFrame[g]; END;

ValidGlobalFrame: PRIVATE ENTRY PROCEDURE [g: GlobalFrameHandle]
  RETURNS [BOOLEAN] = INLINE BEGIN RETURN[ValGlobalFrame[g]] END;
-- Internal Procedures:

AllocGlobalFrame: INTERNAL PROCEDURE [
  old: GlobalFrameHandle, cp: LONG POINTER TO PrincOps.CSegPrefix]
  RETURNS [frame: GlobalFrameHandle, linkspace: CARDINAL] =
  BEGIN
  pbody: LONG POINTER = cp + CARDINAL[cp.entry[MainBodyIndex].initialpc];
  nlinks: CARDINAL = cp.header.info.nlinks;
  linkspace ←
    IF ~old.codelinks THEN
    nlinks + Inline.BITAND[-LOOPHOLE[nlinks, INTEGER], 3B] ELSE 0;
  frame ← Frame.Alloc[MakeFsi[(pbody - 1)↑ + linkspace]];
  RETURN
  END;

EntGlobalFrame: INTERNAL PROCEDURE [frame: GlobalFrameHandle, nslots: CARDINAL]
  RETURNS [entryindex: GFTIndex, ok: BOOLEAN] =
  BEGIN
  i, imax, n, epoffset: CARDINAL;
  i ← gftrover;
  imax ← SDDefs.SD[SDDefs.sGFTLength] - nslots;
```

```
    n ← 0;
    DO
      IF (i ← IF i >= imax THEN 1 ELSE i + 1) = gftrover THEN
        RETURN[entryindex: 0, ok: FALSE];
      IF GFT[i] # EmptyGFTItem THEN n ← 0 ELSE IF (n ← n + 1) = nslots THEN EXIT;
      ENDLOOP;
    entryindex ← (gftrover ← i) - nslots + 1;
    epoffset ← 0;
    FOR i IN [entryindex..gftrover] DO
      GFT[i].framePtr ← frame;
      GFT[i].epbias ← epoffset;
      epoffset ←
        epoffset +
          -- 1 with new format, 32 with old format
          (IF SIZE[GFTItem] = 2 THEN EPRange ELSE 1);
      ENDLOOP;
    RETURN[entryindex: entryindex, ok: TRUE];
    END;

  GetNxGlobalFrame: INTERNAL PROCEDURE [frame: GlobalFrameHandle]
    RETURNS [GlobalFrameHandle] =
    BEGIN
    gfi: GFTIndex;
    IF frame = NullGlobalFrame THEN gfi ← 1 ELSE gfi ← frame.gfi + 1;
    WHILE gfi < SDDefs.SD[SDDefs.sGFTLength] DO
      frame ← GetFrame[GFT[gfi]];
      IF frame # NullGlobalFrame AND GFT[gfi].epbias = 0 THEN RETURN[frame];
      gfi ← gfi + 1;
      ENDLOOP;
    RETURN[NullGlobalFrame]
    END;

  InGFT: INTERNAL PROC [g: GlobalFrameHandle] RETURNS [BOOLEAN] =
    BEGIN
    i: CARDINAL;
    FOR i IN [0..SDDefs.SD[SDDefs.sGFTLength]) DO
      IF GetFrame[GFT[i]] = g AND g.gfi = i THEN RETURN[TRUE]; ENDLOOP;
    RETURN[FALSE];
    END;

  -- InGFT: INTERNAL PROCEDURE [g: GlobalFrameHandle] RETURNS [BOOLEAN] =
  -- BEGIN
  -- IF (LOOPHOLE[g, CARDINAL] MOD 4) ~= 0  + + (assure whole thing is on one page)
  -- THEN RETURN[FALSE]
  -- ELSE IF GFT[g.gfi].frame = g
  -- THEN RETURN[TRUE]
  -- ELSE RETURN[FALSE];
  -- END;

  RemvGlobalFrame: INTERNAL PROCEDURE [frame: GlobalFrameHandle] =
    BEGIN
    FOR i: CARDINAL ← frame.gfi, i + 1 WHILE i < SDDefs.SD[SDDefs.sGFTLength] AND
      GetFrame[GFT[i]] = frame DO
      GFT[i] ← IF frame.copied THEN EmptyGFTItem ELSE FreedGFTItem; ENDLOOP;
    END;

  ValGlobalFrame: INTERNAL PROC [g: GlobalFrameHandle] RETURNS [BOOLEAN] = INLINE
    BEGIN OPEN LOOPHOLE[g, rep ControlLink];
```



```
    RETURN[~proc AND ~indirect AND InGFT[g]];
    END;

  END.
```

LOG

(For earlier log entries see Pilot 4.0 archive version.)

Time: April 14, 1980 10:45 AM By: Knutsen Action: Now STARTed by InitializeFrames[].
Time: April 29, 1980 9:04 PM By: Forrest Action: Put back old InGFT
Time: May 3, 1980 11:44 AM By: Forrest Action: Initial Mesa 6.0 conversion
Time: July 20, 1980 9:03 PM By: Forrest Action: PrincOpsRuntime
Time: August 1, 1980 2:04 PM By: Luniewski Action: Rename to FrameImpl
Time: August 26, 1980 12:27 PM By: McJones Action: Add GetBcdTime

```
-- Instructions.Mesa (last edited by: Forrest on: May 3, 1980 12:39 PM)

DIRECTORY
  Environment: FROM "Environment" USING [Byte],
  Frame: FROM "Frame" USING [GetReturnFrame, GetReturnLink],
  Inline: FROM "Inline" USING [
    BITSHIFT, BITXOR, DIVMOD, LDIVMOD, LongNumber, LongDiv, LongDivMod, LongMult],
  Mopcodes: FROM "Mopcodes" USING [zKFCB],
  PrincOps: FROM "PrincOps" USING [GlobalFrameHandle, StateVector],
  ProcessInternal: FROM "ProcessInternal",
  RuntimeInternal: FROM "RuntimeInternal" USING [Codebase],
  RuntimePrograms: FROM "RuntimePrograms",
  SDDefs: FROM "SDDefs" USING [
    sBLTE, sBLTEC, sBLTECL, sBLTEL, sBoundsFault, sBYTBLTE, sBYTBLTEC, sBYTBLTECL
**,
    sBYTBLTEL, SD, sLongDiv, sLongDivMod, sLongMod, sLongMul, sLongStringCheck,
    sPointerFault, sSignedDiv, sStringInit, sULongDiv, sULongDivMod, sULongMod];

Instructions: PROGRAM
  IMPORTS Frame, Inline, RuntimeInternal EXPORTS RuntimePrograms =
  BEGIN

  -- Unimplemented instructions

  BlockEqual: PROCEDURE [p1: POINTER, n: CARDINAL, p2: POINTER]
    RETURNS [BOOLEAN] = -- BLTE
    BEGIN
    i: CARDINAL;
    FOR i IN [0..n) DO IF (p1 + i)↑ # (p2 + i)↑ THEN RETURN[FALSE]; ENDLOOP;
    RETURN[TRUE]
    END;

  BlockEqualCodeLong: PROCEDURE [p1: LONG POINTER, n, offset: CARDINAL]
    RETURNS [BOOLEAN] = -- BLTECL
    BEGIN
    p2: LONG POINTER =
      RuntimeInternal.Codebase[
        LOOPHOLE[Frame.GetReturnFrame[].accesslink, PROGRAM]] + offset;
    i: CARDINAL;
    FOR i IN [0..n) DO IF (p1 + i)↑ # (p2 + i)↑ THEN RETURN[FALSE]; ENDLOOP;
    RETURN[TRUE]
    END;

  BlockEqualCode: PROCEDURE [p1: POINTER, n, offset: CARDINAL] RETURNS [BOOLEAN] =
    -- BLTEC
    BEGIN
    p2: LONG POINTER =
      RuntimeInternal.Codebase[
        LOOPHOLE[Frame.GetReturnFrame[].accesslink, PROGRAM]] + offset;
    i: CARDINAL;
    FOR i IN [0..n) DO IF (p1 + i)↑ # (p2 + i)↑ THEN RETURN[FALSE]; ENDLOOP;
    RETURN[TRUE]
    END;

  BlockEqualLong: PROCEDURE [p1: LONG POINTER, n: CARDINAL, p2: LONG POINTER]
    RETURNS [BOOLEAN] = -- BLTEL
    BEGIN
    i: CARDINAL;
    FOR i IN [0..n) DO IF (p1 + i)↑ # (p2 + i)↑ THEN RETURN[FALSE]; ENDLOOP;
```



```
    RETURN[TRUE]
    END;

  ByteBlockEqual: PROCEDURE [p1: PPA, n: CARDINAL, p2: PPA] RETURNS [BOOLEAN] =
    -- BYTBLTE
    {RETURN[BlockEqual[p1: p1, p2: p2, n: n/2] AND p1[n - 1] = p2[n - 1]]; };

  ByteBlockEqualCode: PROCEDURE [p1: POINTER, n, offset: CARDINAL]
    RETURNS [BOOLEAN] = -- BYTBLTEC
    BEGIN
    codebase: LONG POINTER = RuntimeInternal.Codebase[
      LOOPHOLE[Frame.GetReturnFrame[].accesslink, PROGRAM]];
    RETURN[ByteBlockEqualLong[p2: codebase + offset, p1: p1, n: n]]
    END;

  ByteBlockEqualCodeLong: PROCEDURE [p1: LONG POINTER, n, offset: CARDINAL]
    RETURNS [result: BOOLEAN] = -- BYTBLTECL
    BEGIN
    codebase: LONG POINTER = RuntimeInternal.Codebase[
      LOOPHOLE[Frame.GetReturnFrame[].accesslink, PROGRAM]];
    RETURN[ByteBlockEqualLong[p2: codebase + offset, p1: p1, n: n]]
    END;

  PPA: TYPE = POINTER TO PACKED ARRAY [0..0) OF Environment.Byte;

  ByteBlockEqualLong: PROC [p1: LONG PPA, n: CARDINAL, p2: LONG PPA]
    RETURNS [BOOLEAN] = -- BYTBLTEL
    {RETURN[BlockEqualLong[p1: p1, p2: p2, n: n/2] AND p1[n - 1] = p2[n - 1]]; };


  -- Data shuffling

  StringInit: PROCEDURE [coffset, n: CARDINAL, reloc, dest: POINTER] =
    BEGIN
    g: PrincOps.GlobalFrameHandle = Frame.GetReturnFrame[].accesslink;
    i: CARDINAL;
    codebase: LONG POINTER =
      RuntimeInternal.Codebase[LOOPHOLE[g, PROGRAM]] + coffset;
    FOR i IN [0..n) DO (dest + i)↑ ← (codebase + i)↑ + reloc; ENDLOOP;
    RETURN
    END;

  -- Long, signed and mixed mode arithmetic

  Number: TYPE = Inline.LongNumber;
  DIVMOD: PROCEDURE [n, d: CARDINAL] RETURNS [QR] = LOOPHOLE[Inline.DIVMOD];
  LDIVMOD: PROCEDURE [nlow, nhigh, d: CARDINAL] RETURNS [QR] =
    LOOPHOLE[Inline.LDIVMOD];
  QR: TYPE = RECORD [q, r: INTEGER];
  PQR: TYPE = POINTER TO QR;

  SignDivide: PROCEDURE =
    BEGIN
    state: PrincOps.StateVector;
    p: PQR;
    t: CARDINAL;
    negnum, negden: BOOLEAN;
    state ← STATE;
    state.stkptr ← t ← state.stkptr - 1;
```

```
  state.dest ← Frame.GetReturnLink[];
  p ← @state.stk[t - 1];
  IF negden ← (p.r < 0) THEN p.r ← -p.r;
  IF negnum ← (p.q < 0) THEN p.q ← -p.q;
  p↑ ← DIVMOD[n: p.q, d: p.r];
  IF Inline.BITXOR[negnum, negden] # 0 THEN p.q ← -p.q;
  IF negnum THEN p.r ← -p.r;
  RETURN WITH state
  END;

DDivMod: PROCEDURE [num, den: Number] RETURNS [quotient, remainder: Number] =
  BEGIN
  negNum, negDen: BOOLEAN ← FALSE;
  IF LOOPHOLE[num.highbits, INTEGER] < 0 THEN
    BEGIN negNum ← TRUE; num.li ← -num.li; END;
  IF LOOPHOLE[den.highbits, INTEGER] < 0 THEN
    BEGIN negDen ← TRUE; den.li ← -den.li; END;
  [quotient: quotient, remainder: remainder] ← DUnsignedDivMod[
    num: num, den: den];
  IF Inline.BITXOR[negNum, negDen] # 0 THEN quotient.li ← -quotient.li;
  IF negNum THEN remainder.li ← -remainder.li;
  RETURN
  END;

DDiv: PROC [a, b: Number] RETURNS [Number] = {RETURN[DDivMod[a, b].quotient]; };

DMod: PROCEDURE [a, b: Number] RETURNS [r: Number] = {
  [remainder: r] ← DDivMod[a, b]; RETURN; };

DMultiply: PROCEDURE [a, b: Number] RETURNS [product: Number] =
  BEGIN
  product.lc ← Inline.LongMult[a.lowbits, b.lowbits];
  product.highbits ←
    product.highbits + a.lowbits*b.highbits + a.highbits*b.lowbits;
  RETURN
  END;

DUnsignedDivMod: PROCEDURE [num, den: Number]
  RETURNS [quotient, remainder: Number] =
  BEGIN OPEN Inline;
  qq: CARDINAL;
  count: [0..31);
  lTemp: Number;
  IF den.highbits = 0 THEN
    BEGIN
    [quotient.highbits, qq] ← LongDivMod[
      LOOPHOLE[Number[num[lowbits: num.highbits, highbits: 0]]], den.lowbits];
    [quotient.lowbits, remainder.lowbits] ← LongDivMod[
      LOOPHOLE[Number[num[lowbits: num.lowbits, highbits: qq]]], den.lowbits];
    remainder.highbits ← 0;
    END
  ELSE
    BEGIN
    count ← 0;
    quotient.highbits ← 0;
    lTemp ← den;
    WHILE lTemp.highbits # 0 DO
      -- normalize
      lTemp.lowbits ←
```



```
        BITSHIFT[lTemp.lowbits, -1] + BITSHIFT[lTemp.highbits, 15];
      lTemp.highbits ← BITSHIFT[lTemp.highbits, -1];
      count ← count + 1;
      ENDLOOP;
    IF num.highbits >= lTemp.lowbits THEN
      BEGIN -- subtract off 2↑16*divisor and fix up count
      div: Number ← Number[num[lowbits: 0, highbits: lTemp.lowbits]];
      qq ← LongDiv[num.lc - div.lc, lTemp.lowbits]/2 + 100000B;
      count ← count - 1;
      END
    ELSE qq ← LongDiv[num.lc, lTemp.lowbits]; -- trial quotient
    qq ← BITSHIFT[qq, -count];
    lTemp.lc ← LongMult[den.lowbits, qq]; -- multiply by trial quotient
    lTemp.highbits ← lTemp.highbits + den.highbits*qq;
    UNTIL lTemp.lc <= num.lc DO
      -- decrease quotient until product is small enough
      lTemp.lc ← lTemp.lc - den.lc;
      qq ← qq - 1;
      ENDLOOP;
    quotient.lowbits ← qq;
    remainder.lc ← num.lc - lTemp.lc;
    END;
  RETURN
  END;

DUnsignedDiv: PROCEDURE [a, b: Number] RETURNS [Number] = {
  RETURN[DUnsignedDivMod[a, b].quotient]; };

DUnsignedMod: PROCEDURE [a, b: Number] RETURNS [r: Number] = {
  [remainder: r] ← DUnsignedDivMod[a, b]; RETURN; };

-- Other

LongStringCheck: PROCEDURE =
  BEGIN
  state: PrincOps.StateVector;
  tos, index: CARDINAL;
  p: POINTER TO LONG STRING;
  BoundsFault: PROCEDURE = MACHINE CODE
    BEGIN Mopcodes.zKFCB, SDDefs.sBoundsFault END;
  PointerFault: PROCEDURE = MACHINE CODE
    BEGIN Mopcodes.zKFCB, SDDefs.sPointerFault END;
  state ← STATE;
  tos ← state.stkptr;
  index ← state.stk[tos];
  p ← @state.stk[tos - 2];
  IF p↑ = NIL THEN PointerFault[];
  IF index >= p↑.maxlength THEN BoundsFault[];
  -- This statement is new for the dandelion. It should be compatible with the Dolphin's pre-princ
**ops stuff (he says, crossing fingers)
  state.dest ← Frame.GetReturnLink[];
  state.source ← 0;
  RETURN WITH state;
  END;

Initialize: PROCEDURE =
  BEGIN OPEN SDDefs;
  pSD: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
```

```
    pSD[sBLTE] ← BlockEqual;
    pSD[sBYTBLTE] ← ByteBlockEqual;
    pSD[sBLTEC] ← BlockEqualCode;
    pSD[sBYTBLTEC] ← ByteBlockEqualCode;
    pSD[sBLTECL] ← BlockEqualCodeLong;
    pSD[sBYTBLTECL] ← ByteBlockEqualCodeLong;
    pSD[sBLTEL] ← BlockEqualLong;
    pSD[sBYTBLTEL] ← ByteBlockEqualLong;
    pSD[sLongDiv] ← DDiv;
    pSD[sLongDivMod] ← DDivMod;
    pSD[sLongMod] ← DMod;
    pSD[sLongMul] ← DMultiply;
    pSD[sLongStringCheck] ← LongStringCheck;
    pSD[sULongDivMod] ← DUnsignedDivMod;
    pSD[sULongMod] ← DUnsignedMod;
    pSD[sULongDiv] ← DUnsignedDiv;
    pSD[sSignedDiv] ← SignDivide;
    pSD[sStringInit] ← StringInit;
    END;

  -- Main Body;

  Initialize[];

  END.
```

LOG

Time: July 20, 1978 9:04 AM By: Sandman Action: Bug in StringInit
Time: August 7, 1978 8:51 AM By: Sandman Action: Got Codebase from RuntimeInternal
***instead of CodebaseDefs*
Time: March 14, 1979 11:27 AM By: McJones Action: Mesa 5
Time: July 11, 1979 1:33 PM By: Jose Action: ???
Time: October 3, 1979 3:43 PM By: McJones Action: Bug in DUnsignedDivMod
Time: December 5, 1979 8:20 AM By: Sandman Action: AR 3056 BlockEqual spee
***dup*
Time: May 3, 1980 12:28 PM By: Forrest Action: Mesa 6

```
-- PilotNub.mesa (last edited by: Forrest on: October 4, 1980 2:53 PM)

DIRECTORY
  Boot USING [ReadMDS],
  BootSwap USING [InLoad, OutLoad, Teledebug],
  CPSwapDefs USING [
    BBArray, BBHandle, callDP, DebugParameter, ExternalStateVector, SwapInfo,
    startDP, SwapReason, UBBPointer, UserBreakBlock, VersionID],
  DebuggerSwap USING [canSwap, parameters],
  DeviceCleanup USING [Item, Linkage, Perform, Reason],
  Environment USING [Byte, maxPagesInMDS, PageCount, PageNumber],
  Frame USING [
    GetReturnFrame, GetReturnLink, MyLocalFrame, SetReturnFrame, SetReturnLink],
  Inline USING [HighHalf, LowHalf],
  KeyboardFace USING [keyboard],
  Keys USING [DownUp, KeyBits],
  Mopcodes USING [zDUP, zRFS],
  PilotMP USING [cCantSwap, cCleanup, cClient, cHang, Code],
  PilotSwitches USING [switches --.h, .i, .r--],
  PrincOps USING [
    ControlLink, FieldDescriptor, FrameHandle, GlobalFrameHandle, NullFrame, Port,
    StateVector, SVPointer],
  Process USING [Detach, GetPriority, Priority, SetPriority, SetTimeout, Ticks],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  ProcessOperations USING [
    HandleToIndex, IndexToHandle, ReadPSB, ReadPTC, ReadWDC, WritePSB, WritePTC,
    WriteWDC],
  ProcessorFace USING [
    BootButton, GetClockPulses, microsecondsPerHundredPulses, SetMP],
  PSB USING [PDA, StateVector],
  Runtime USING [Interrupt],
  RuntimeInternal USING [],
  RuntimePrograms USING [],
  SDDefs USING [
    sAlternateBreak, sBreak, sBreakBlock, sBreakBlockSize, sCallDebugger,
    sCoreSwap, SD, sInterrupt, sProcessBreakpoint, sUncaughtSignal,
    sWorryCallDebugger, sXferTrap],
  XferTrap USING [ReadXTS, Status, WriteXTS];

PilotNub: MONITOR -- for CheckInterrupt, DeviceCleanup.Install

  IMPORTS
    Boot, BootSwap, DebuggerSwap, DeviceCleanup, Frame, Inline, KeyboardFace,
    PilotSwitches, Process, ProcessInternal, ProcessOperations, ProcessorFace,
    Runtime, XferTrap
  EXPORTS DeviceCleanup, RuntimeInternal, RuntimePrograms
  SHARES DeviceCleanup =
  BEGIN

  -- This module is the debugger's representative in the client world.

  loadStatePage: PUBLIC Environment.PageNumber; -- exported to RuntimeInternal

  CAbort: PUBLIC SIGNAL = CODE;
  CantSwap: PUBLIC SIGNAL = CODE;
  KillThisTurkey: SIGNAL = CODE;
  Quit: SIGNAL = CODE;
```



```
  --
  -- Breakpoints

  numberBlocks: CARDINAL = 5; -- number of break blocks
  BreakBlocks: TYPE = MACHINE DEPENDENT RECORD [
    header(0): CPSwapDefs.BBArray, -- length plus (zero-length) array of blocks
    body(1): ARRAY [0..numberBlocks) OF CPSwapDefs.UserBreakBlock];
  breakBlocks: BreakBlocks;

  InitBreakBlocks: PROC =
    BEGIN OPEN SDDefs;
    SD[sBreakBlock] ← Inline.LowHalf[LONG[@breakBlocks]];
    -- mds relative (should really be long pointer)
    SD[sBreakBlockSize] ← SIZE[BreakBlocks];
    breakBlocks.header.length ← 0;
    END;

  Break: PROC = -- executed by (non-worry) BRK instruction
    BEGIN
    xferTrapStatus: XferTrap.Status;
    state: RECORD [a: UNSPECIFIED, v: PrincOps.StateVector];
    state.v ← STATE;
    state.v.dest ← Frame.GetReturnFrame[];
    state.v.source ← 0;
    xferTrapStatus ← XferTrap.ReadXTS[];
    ProcessBreakpoint[@state.v];
    -- isn't this supposed to go through SD[sProcessBreakpoint]?
    IF xferTrapStatus = on THEN XferTrap.WriteXTS[skip1];
    RETURN WITH state.v
    END;

  ProcessBreakpoint: PROC [s: PrincOps.SVPointer] =
    BEGIN
    inst: Environment.Byte;
    swap: BOOLEAN;
    [inst, swap] ← DoBreakpoint[s];
    IF swap THEN CoreSwap[breakpoint, s] ELSE s.instbyte ← inst
    -- replant the instruction and go on

    END;

  -- make this INLINE someday??

  DoBreakpoint: PROC [s: PrincOps.SVPointer] RETURNS [Environment.Byte, BOOLEAN] =
    BEGIN
    bba: CPSwapDefs.BBHandle = SDDefs.SD[SDDefs.sBreakBlock];
    l: PrincOps.FrameHandle ← s.dest;
    FOR i: CARDINAL IN [0..bba.length) DO
      ubb: CPSwapDefs.UBBPointer = @bba.blocks[i];
      IF ubb.frame = l.accesslink AND ubb.pc = CARDINAL[l.pc] THEN
        IF TrueCondition[ubb, l, s] THEN EXIT ELSE RETURN[ubb.inst, FALSE]
      ENDLOOP;
    RETURN[0, TRUE]
    END;

  -- decide whether to take the breakpoint

  TrueCondition: PROC [
```

```
  ubb: CPSwapDefs.UBBPointer, base: POINTER, s: PrincOps.SVPointer]
  RETURNS [BOOLEAN] = INLINE
  BEGIN
  ReadField: PROC [POINTER, PrincOps.FieldDescriptor] RETURNS [UNSPECIFIED] =
    MACHINE CODE BEGIN Mopcodes.zRFS END;
  fd: PrincOps.FieldDescriptor;
  locL, locR: POINTER;
  left, right: UNSPECIFIED;
  IF ubb.counterL THEN RETURN[(ubb.ptrL ← ubb.ptrL + 1) = ubb.ptrR];
  locL ←
    SELECT TRUE FROM
      ubb.localL => base + LOOPHOLE[ubb.ptrL, CARDINAL],
      ubb.stackRelative => @s.stk[LOOPHOLE[ubb.ptrL, CARDINAL]],
      ENDCASE => ubb.ptrL;
  fd ← [offset: 0, posn: ubb.posnL, size: ubb.sizeL];
  left ← ReadField[locL, fd];
  IF ~ubb.immediateR THEN
    BEGIN
    fd ← [offset: 0, posn: ubb.posnR, size: ubb.sizeR];
    locR ← IF ubb.localR THEN base + LOOPHOLE[ubb.ptrR, CARDINAL] ELSE ubb.ptrR;
    right ← ReadField[locR, fd]
    END
  ELSE right ← ubb.ptrR;
  RETURN[
    SELECT ubb.relation FROM
      lt => left < right,
      gt => left > right,
      eq => left = right,
      ne => left # right,
      le => left <= right,
      ge => left >= right,
      ENDCASE => FALSE]
  END;

-- executed by worry-mode BRK instruction

WorryBreaker: PROC RETURNS [PrincOps.FrameHandle] =
  BEGIN OPEN PrincOps;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  xferTrapStatus: XferTrap.Status;
  state.v.instbyte ← 0;
  state.v.stkptr ← 1;
  state.v.stk[0] ← Frame.MyLocalFrame[];
  state.v.dest ← Frame.GetReturnLink[];
  ProcessInternal.DisableInterrupts[];
  DO
    xferTrapStatus ← XferTrap.ReadXTS[];
    IF xferTrapStatus = on THEN XferTrap.WriteXTS[skip1];
    ProcessInternal.EnableInterrupts[];
    TRANSFER WITH state.v;
    ProcessInternal.DisableInterrupts[];
    state.v ← STATE;
    state.v.dest ← Frame.GetReturnFrame[];
    state.v.source ← 0;
    swapInfo.state ← @state.v;
    swapInfo.reason ← worrybreak; -- set mds too
    ToDebugger[@swapInfo];
    ENDLOOP
  END;
```



```
--
-- Uncaught signals

Catcher: PROC [msg, signal: UNSPECIFIED, frame: PrincOps.FrameHandle] =
  BEGIN
  Punt: PROC [c: PilotMP.Code] = INLINE {ProcessorFace.SetMP[c]; DO ENDLOOP};
  SignallerGF: PrincOps.GlobalFrameHandle;
  state: PrincOps.StateVector;
  f: PrincOps.FrameHandle;
  state.stk[0] ← msg;
  state.stk[1] ← signal;
  state.stkptr ← 0;
  -- The call stack below here is: Signaller, [Signaller,] offender
  f ← Frame.GetReturnFrame[];
  SignallerGF ← f.accesslink;
  state.dest ← f ← f.returnlink.frame;
  IF f.accesslink = SignallerGF THEN state.dest ← f.returnlink;
  IF signal = CantSwap THEN Punt[PilotMP.cCantSwap];
  BEGIN
  CoreSwap[uncaughtsignal, @state ! CAbort => GOTO abort];
  EXITS
    abort =>
      IF signal = ABORTED THEN {BackStop[frame]; ERROR KillThisTurkey}
      ELSE ERROR ABORTED
  END
  END;

BackStop: PROC [root: PrincOps.FrameHandle] =
  BEGIN
  endProcess: PrincOps.ControlLink = root.returnlink;
  Caller: PROC = LOOPHOLE[Frame.GetReturnLink[]];
  root.returnlink ← [frame[Frame.MyLocalFrame[]]];
  Frame.SetReturnFrame[PrincOps.NullFrame];
  Caller[ ! KillThisTurkey => CONTINUE];
  Frame.SetReturnLink[endProcess];
  END;

--
-- Interrupts (e.g. CTRL-SWAT)

wakeup: CONDITION;
ticksPerWakeup: Process.Ticks = 1;
priority: Process.Priority = 6;

CheckInterrupt: ENTRY PROC = -- default CTRL-SWAT watcher
  BEGIN
  interruptState: Keys.DownUp ← up;
  pKeys: LONG POINTER TO Keys.KeyBits = LOOPHOLE[KeyboardFace.keyboard];
  DO
    ENABLE ABORTED => CONTINUE;
    WAIT wakeup;
    IF pKeys[Spare3] = down AND pKeys[Ctrl] = down AND pKeys[LeftShift] = up
      THEN
      BEGIN
      IF interruptState = up AND PSB.PDA.externalStateVector # NIL THEN {
        interruptState ← down; Runtime.Interrupt[]}
      END
    ELSE interruptState ← up
    ENDLOOP
  END;
```

```
Interrupt: PROC = -- implementation of Runtime.Interrupt
  BEGIN
  state: RECORD [a, b: UNSPECIFIED, v: PrincOps.StateVector];
  state.v ← STATE;
  state.v.dest ← Frame.MyLocalFrame[];
  CoreSwap[breakpoint, @state.v]
  END;

InitializeInterrupt: PUBLIC PROC =
  -- Initialize the default CTRL-SWAT watcher.
  -- Must not be invoked until KeyboardFace has been initialized.
  BEGIN
  IF PilotSwitches.switches.i = down THEN
    BEGIN OPEN Process;
    save: Priority = GetPriority[];
    SetTimeout[@wakeup, ticksPerWakeup];
    SetPriority[priority];
    Detach[FORK CheckInterrupt];
    SetPriority[save];
    END;
  END;

--
-- Miscellaneous Runtime, RuntimeInternal items

CallDebugger: PROC [s: STRING] =
  -- Runtime.CallDebugger is KFCB[sCallDebugger]
  BEGIN
  state: RECORD [a, b: UNSPECIFIED, v: PrincOps.StateVector];
  state.v ← STATE;
  state.v.stk[0] ← s;
  state.v.stkptr ← 1;
  state.v.dest ← Frame.GetReturnLink[];
  CoreSwap[explicitcall, @state.v]
  END;

WorryCallDebugger: PROC RETURNS [PrincOps.FrameHandle] =
  BEGIN OPEN PrincOps;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  xferTrapStatus: XferTrap.Status;
  state.v.instbyte ← 0;
  state.v.stkptr ← 1;
  state.v.stk[0] ← Frame.MyLocalFrame[];
  state.v.dest ← Frame.GetReturnLink[];
  ProcessInternal.DisableInterrupts[];
  DO
    xferTrapStatus ← XferTrap.ReadXTS[];
    IF xferTrapStatus = on THEN XferTrap.WriteXTS[skip1];
    ProcessInternal.EnableInterrupts[];
    TRANSFER WITH state.v;
    ProcessInternal.DisableInterrupts[];
    state.v ← STATE;
    state.v.dest ← state.v.stk[state.v.stkptr + 1];
    Frame.SetReturnLink[state.v.dest];
    state.v.source ← 0;
    swapInfo.state ← @state.v;
    swapInfo.reason ← worrycall;
    ToDebugger[@swapInfo];
```



```
    ENDLOOP
  END;

CleanMapLog: PUBLIC PROC =
  BEGIN
  state: RECORD [a, b: UNSPECIFIED, v: PrincOps.StateVector];
  state.v ← STATE;
  state.v.stkptr ← 0;
  state.v.dest ← Frame.GetReturnLink[];
  CoreSwap[cleanmaplog, @state.v]
  END;

-- Procedures that cause swap to debugger

swapInfo: CPSwapDefs.ExternalStateVector; -- nub-debugger communication area

parmstring: STRING = [40];

GetMDS: PROC RETURNS [Environment.PageNumber] =
  BEGIN RETURN[Boot.ReadMDS[]*Environment.maxPagesInMDS] END;

CoreSwap: PROC [why: CPSwapDefs.SwapReason, sp: PrincOps.SVPointer] =
  BEGIN
  DP: CPSwapDefs.DebugParameter;
  decode: PROC RETURNS [BOOLEAN] = -- decode the SwapReason
    BEGIN
    f: PrincOps.GlobalFrameHandle;
    lsv: PrincOps.StateVector;
    SELECT swapInfo.reason FROM
      proceed, resume => RETURN[TRUE];
      call =>
        BEGIN
        lsv ← LOOPHOLE[swapInfo.parameter, CPSwapDefs.callDP].sv;
        lsv.source ← Frame.MyLocalFrame[];
        TRANSFER WITH lsv;
        lsv ← STATE;
        LOOPHOLE[swapInfo.parameter, CPSwapDefs.callDP].sv ← lsv;
        why ← return
        END;
      start =>
        BEGIN
        f ← LOOPHOLE[swapInfo.parameter, CPSwapDefs.startDP].frame;
        IF ~f.started THEN START LOOPHOLE[f, PROGRAM] ELSE RESTART f;
        why ← return
        END;
      quit => SIGNAL Quit;
      ENDCASE => RETURN[TRUE];
    RETURN[FALSE]
    END;

  -- Body of CoreSwap

  -- IF ~DebuggerSwap.canSwap THEN SIGNAL CantSwap;
  swapInfo.state ← sp;
  DP.string ← parmstring;
  swapInfo.parameter ← @DP;
  -- versionident, debuggee, lspages, fill, mapLog set by Initialize
  -- level, loadstatepage set by MemorySwap
  swapInfo.mds ← GetMDS[];
```

```
    DO
      swapInfo.reason ← why;
      ProcessInternal.DisableInterrupts[];
      ToDebugger[@swapInfo];
      ProcessInternal.EnableInterrupts[];
      BEGIN
      IF decode[
        ! CAbort => IF swapInfo.level > 0 THEN {why ← return; CONTINUE};
        Quit => GOTO abort] THEN EXIT
      EXITS abort => SIGNAL CAbort
      END
      ENDLOOP
    END;

  -- Serious swapper

  level: INTEGER ← -1;

  ToDebugger: PORT [POINTER TO CPSwapDefs.ExternalStateVector]; -- formerly WBPort
  FromPilot: PORT RETURNS [POINTER TO CPSwapDefs.ExternalStateVector];
  -- formerly CSPort

  pulsesPerTwentySeconds: LONG CARDINAL;

  MemorySwap: PROC [pESV: POINTER TO CPSwapDefs.ExternalStateVector] =
    BEGIN
    PKeys: PROC RETURNS [LONG POINTER TO Keys.KeyBits] = INLINE {
      RETURN[LOOPHOLE[KeyboardFace.keyboard]]};
    SwapIt: PROC = INLINE
      BEGIN
      savewdc, saveptc: UNSPECIFIED;
      xferTrapStatus: XferTrap.Status = XferTrap.ReadXTS[];
      xferTrapHandler: UNSPECIFIED = SDDefs.SD[SDDefs.sXferTrap];
      pESV.level ← level;
      pESV.loadstatepage ← loadStatePage; -- possibly updated by loader
      pESV.mds ← Inline.HighHalf[LONG[LOOPHOLE[1, POINTER]]];
      -- ("1" since 0 collides with NIL.)
      SDDefs.SD[SDDefs.sXferTrap] ← Frame.MyLocalFrame[];
      -- in case we are restarted in trap mode
      XferTrap.WriteXTS[off];
      -- Save process state not captured in PDA:
      -- Change pda => psb in ExternalStateVector:
      pESV.pda ← LOOPHOLE[ProcessOperations.HandleToIndex[
        ProcessOperations.ReadPSB[]]];
      saveptc ← ProcessOperations.ReadPTC[];
      savewdc ← ProcessOperations.ReadWDC[];
      -- Ensure we will be visible after a Set Process context command in the debugger:
      PSB.PDA[PSB.PDA.currentState].frame ← [frame[Frame.MyLocalFrame[]]];
      DeviceCleanup.Perform[turnOff]; -- turn all devices off
      IF PilotSwitches.switches.h = down THEN
        BEGIN
        AddToStack: PROC [BOOLEAN] = MACHINE CODE BEGIN END;
        GetTOS: PROC RETURNS [BOOLEAN] = MACHINE CODE BEGIN Mopcodes.zDUP; END;
        RemoveFromStack: PROC RETURNS [BOOLEAN] = MACHINE CODE BEGIN END;
        AddToStack[TRUE];
        ProcessorFace.SetMP[PilotMP.cHang];
        WHILE GetTOS[] DO ENDLOOP;
        [] ← RemoveFromStack[];
        END
```



```
      ELSE
        IF ~DebuggerSwap.canSwap OR PilotSwitches.switches.r = down THEN
          BEGIN
          ProcessorFace.SetMP[PilotMP.cCantSwap];
          BootSwap.Teledebug[@DebuggerSwap.parameters.locDebugger];
          END
          -- OutLoad onto swatee, then boot Debugger.

        ELSE
          IF BootSwap.OutLoad[@DebuggerSwap.parameters.locDebuggee, restore] =
            outLoaded THEN
            BEGIN OPEN DebuggerSwap, parameters;
            -- The next line should be in BootSwap.InLoad but blows up the compiler
            IF pMicrocodeCopy ~= NIL THEN DeviceCleanup.Perform[kill];
            BootSwap.InLoad[pMicrocodeCopy, pGermCopy, nGerm, @locDebugger]
            -- never returns

            END;
      -- Restore process state not captured in PDA:
      ProcessOperations.WriteWDC[savewdc];
      ProcessOperations.WritePTC[saveptc];
      -- Change pda => psb in ExternalStateVector:
      ProcessOperations.WritePSB[
        ProcessOperations.IndexToHandle[LOOPHOLE[pESV.pda]]];
      DeviceCleanup.Perform[turnOn]; -- turn devices back on
      level ← pESV.level;
      XferTrap.WriteXTS[xferTrapStatus];
      SDDefs.SD[SDDefs.sXferTrap] ← xferTrapHandler;
      ProcessorFace.SetMP[PilotMP.cClient]; -- announce our return

      END;
    DO
      pESV ← FromPilot[];
      -- Set our return link so Display Stack will work:
      Frame.SetReturnLink[LOOPHOLE[FromPilot, PrincOps.Port].dest];
      DO
        SwapIt[];
        SELECT pESV.reason FROM
          kill => ProcessorFace.BootButton[];
          showscreen =>
            BEGIN
            pulsesThen: LONG CARDINAL = ProcessorFace.GetClockPulses[];
            prevSpare3: Keys.DownUp ← PKeys[][Spare3];
            DO
              IF (ProcessorFace.GetClockPulses[] - pulsesThen) >
                pulsesPerTwentySeconds THEN EXIT;
              IF prevSpare3 = down THEN prevSpare3 ← PKeys[][Spare3]
              ELSE
                IF PKeys[][Spare3] = down THEN {
                  WHILE PKeys[][Spare3] = down DO --snoore-- ENDLOOP; EXIT}
                ELSE NULL -- both up

              ENDLOOP;
            END;
          ENDCASE => EXIT;
        pESV.reason ← return;
        ENDLOOP
      ENDLOOP
    END;
```

```
--
-- DeviceCleanup implementation

linkage: PUBLIC DeviceCleanup.Linkage;
InitializeDeviceCleanup: PROC =
  BEGIN
  reason: DeviceCleanup.Reason;
  pItem: POINTER TO Item;
  LOOPHOLE[AwaitPerform, PrincOps.Port].dest ← Frame.GetReturnLink[];
  linkage.Perform ← LOOPHOLE[@AwaitPerform];
  DO
    linkage.Await ← LOOPHOLE[Install];
    reason ← AwaitPerform[];
    ProcessorFace.SetMP[PilotMP.cCleanup];
    linkage.Await ← LOOPHOLE[Frame.MyLocalFrame[]];
    FOR pItem ← pItemFirst, pItem.pItemNext WHILE pItem ~= NIL DO
      [] ← pItem.Procedure[reason] -- value should be pItem

    ENDLOOP
  ENDLOOP
END;

AwaitPerform: PORT RETURNS [reason: DeviceCleanup.Reason];

Install: ENTRY PROC [pItem: POINTER TO Item] =
  BEGIN
  fCaller: PrincOps.FrameHandle = Frame.GetReturnFrame[]; -- cleanup procedure
  pItem↑ ← [pItemNext: pItemFirst, Procedure: LOOPHOLE[fCaller]];
  pItemFirst ← pItem;
  Frame.SetReturnLink[fCaller.returnlink]
  END;

Item: PUBLIC TYPE = RECORD [
  pItemNext: POINTER TO Item,
  Procedure: PROC [DeviceCleanup.Reason] RETURNS [POINTER TO Item]];
pItemFirst: POINTER TO Item ← NIL; -- list of waiting cleanup procedures

--
-- Initialization

InitializePilotNub: PUBLIC PROC [
  pageLoadState: Environment.PageNumber, countLoadState: Environment.PageCount,
  pVMMapLog: LONG POINTER --TO VMMMapLog.Descriptor--] =
  BEGIN
  InitializeDeviceCleanup[];

  loadStatePage ← pageLoadState;
  -- swapInfo.state set in caller of MemorySwap
  --  (e.g. CoreSwap, WorryBreaker, WorryCallDebugger)
  -- swapInfo.reason set in caller of MemorySwap
  --  (e.g. CoreSwap, WorryBreaker, WorryCallDebugger)
  -- swapInfo.level set in MemorySwap
  -- swapInfo.parameter set by debugger and obeyed by CoreSwap
  swapInfo.versionident ← CPSwapDefs.VersionID;
  -- swapInfo.loadstatepage set in MemorySwap
  swapInfo.lspages ← countLoadState;
  swapInfo.mapLog ← pVMMapLog;
  swapInfo.mds ← GetMDS[]; -- delete this when WorryBreak, etc. set the field
  swapInfo.fill ← ALL[0];
```



```
  PSB.PDA.externalStateVector ← CPSwapDefs.SwapInfo↑ ← @swapInfo;
  InitBreakBlocks[];
  BEGIN OPEN SDDefs;
  pSD: POINTER TO ARRAY [0..0) OF UNSPECIFIED = SD;
  pSD[sProcessBreakpoint] ← ProcessBreakpoint;
  pSD[sUncaughtSignal] ← Catcher;
  pSD[sInterrupt] ← Interrupt;
  pSD[sCallDebugger] ← CallDebugger;
  pSD[sBreak] ← Break;
  pSD[sAlternateBreak] ← WorryBreaker[];
  pSD[sWorryCallDebugger] ← WorryCallDebugger[];
  pulsesPerTwentySeconds ←
    LONG[20]*1000000*100/ProcessorFace.microsecondsPerHundredPulses;
  LOOPHOLE[ToDebugger, PrincOps.Port].out ← @FromPilot;
  -- connect ToDebugger to FromPilot
  LOOPHOLE[FromPilot, PrincOps.Port].out ← @ToDebugger;
  -- connect FromPilot to ToDebugger
  pSD[sCoreSwap] ← @FromPilot;
  LOOPHOLE[FromPilot, PrincOps.Port].in ← MemorySwap;
  ToDebugger[NIL]; -- allocate frame for MemorySwap

  END;
  END;

END.
```

```
(For earlier log entries see Pilot 4.0 archive version.)
April 29, 1980  9:50 PM          Forrest    Move in  Memory swap stuff from Traps
May 3, 1980  2:29 PM Forrest     Mesa 6.0
June 23, 1980  6:26 PM           McJones    Fix WorryCallDebugger bug; OISProcessorFace = >Pro
**cessorFace; make TrueCondition INLINE; allocate break blocks in global frame
July 28, 1980  6:46 PM           McJones    New KeyboardFace, Keys
July 31, 1980  7:21 PM           Forrest    Implement Hang switch
August 28, 1980  2:59 PM         McJones    Merge InterruptKey with PilotNub; timeout userscreen
**after twenty seconds; SetMP[cCleanup]
August 27, 1980  4:00 PM         McJones    New PSB, ProcessOperations, XferTrap
October 3, 1980  1:30 PM         Forrest    Reverse sense of i switch; add code to dally during use
**rscreen as long as swat held down; Jim Sandman fixed conditional Break logic
```

```
-- Processes.mesa (last edited by: Johnsson on: October 10, 1980 6:31 PM)

DIRECTORY
  Environment USING [Long, PageCount, PageNumber, wordsPerPage],
  Frame USING [Free, GetReturnFrame, MyLocalFrame, SetReturnFrame],
  Inline USING [BITAND, BITNOT, BITOR, BITSHIFT, LowHalf],
  PrincOps USING [Frame, FrameHandle, NullFrame, StateVector, TrapLink],
  Process USING [Milliseconds, Priority, Ticks],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  ProcessOperations USING [
    Broadcast, EnableAndRequeue, Enter, Exit, HandleToIndex, IndexToHandle,
    Notify, ReadPSB, ReadPTC, ReEnter, Wait, WritePSB, WriteWDC],
  ProcessorFace USING [millisecondsPerTick, reservedNakedNotifyMask],
  PSB USING [
    Condition, Empty, MonitorLock, PDA, PDABase, ProcessDataArea, Handle, Index,
    PSB, Queue, StartPsb, UnlockedEmpty],
  RuntimePrograms USING [],
  SDDefs USING [SD, sFork, sJoin, sProcessTrap];

Processes: MONITOR [
  pagePDA: Environment.PageNumber, countPDA: Environment.PageCount]LOCKS
  processLock
  IMPORTS Frame, Inline, ProcessInternal, ProcessorFace, ProcessOperations
  EXPORTS Process, ProcessInternal, RuntimePrograms =
  BEGIN OPEN Process, ProcessInternal, ProcessOperations, PSB;

  DyingFrameHandle: TYPE = POINTER TO dying PrincOps.Frame;
  LongConditionPtr: TYPE = LONG POINTER TO Condition;
  LongCONDITIONPtr: TYPE = LONG POINTER TO CONDITION;

  processLock: MONITORLOCK;
  busyLevels: WORD;
  -- bit-mask of busy naked-notify levels (numbered right-to-left)
  dead, frameReady, frameTaken, rebirth: CONDITION;
  deadFrame: DyingFrameHandle ← NIL; -- only for detached processes
  defaultPriority: Priority = 1;
  -- this should be in a Pilot definitions somewhere!
  pda: PDABase;

  InvalidProcess: PUBLIC ERROR [process: Index] = CODE;
  TooManyProcesses: PUBLIC ERROR = CODE;

  -- Non-ENTRY procedures

  DisableAborts: PUBLIC PROCEDURE [condition: LongCONDITIONPtr] = {
    LOOPHOLE[condition, LongConditionPtr].enableAborts ← FALSE};

  DisableTimeout: PUBLIC PROCEDURE [condition: LongCONDITIONPtr] = {
    LOOPHOLE[condition, LongConditionPtr].timeout ← 0};

  EnableAborts: PUBLIC PROCEDURE [condition: LongCONDITIONPtr] = {
    LOOPHOLE[condition, LongConditionPtr].enableAborts ← TRUE};

  GetCurrent: PUBLIC PROCEDURE RETURNS [p: UNSPECIFIED] = {
    RETURN[HandleToIndex[ReadPSB[]]]};

  GetPriority: PUBLIC PROCEDURE RETURNS [p: Priority] = {
    RETURN[pda[ReadPSB[]].link.priority]};
```



```
  InitializeCondition: PUBLIC PROCEDURE [
    condition: LongCONDITIONPtr, ticks: Ticks] = {
    LOOPHOLE[condition, LongConditionPtr]↑ ←
      [tail: Empty, enableAborts: FALSE, wakeup: FALSE, timeout: MAX[1, ticks]]};

  InitializeMonitor: PUBLIC PROCEDURE [monitor: LONG POINTER TO MONITORLOCK] = {
    LOOPHOLE[monitor, LONG POINTER TO MonitorLock]↑ ← UnlockedEmpty; };

  MsecToTicks: PUBLIC PROCEDURE [ms: Milliseconds] RETURNS [Ticks] = {
    RETURN[
      (ms + ProcessorFace.millisecondsPerTick -
        1)/ProcessorFace.millisecondsPerTick]};

  ProcessTrap: PROCEDURE RETURNS [BOOLEAN] =
    BEGIN
    abortee: PrincOps.FrameHandle;
    state: RECORD [a: UNSPECIFIED, v: PrincOps.StateVector];
    Enter: PROCEDURE [a: UNSPECIFIED] RETURNS [BOOLEAN] =
      LOOPHOLE[ProcessOperations.Enter];
    LongEnter: PROCEDURE [a, b: UNSPECIFIED] RETURNS [BOOLEAN] = LOOPHOLE[Enter];
    state.v ← STATE;
    IF state.v.stkptr = 4 THEN
      UNTIL LongEnter[state.v.stk[0], state.v.stk[1]] DO ENDLOOP
    ELSE UNTIL Enter[state.v.stk[0]] DO ENDLOOP;
    abortee ← Frame.GetReturnFrame[];
    abortee.pc ← [abortee.pc + 1];
    pda[ReadPSB[]].flags.abortPending ← FALSE;
    ERROR ABORTED;
    -- if ABORTED is made resumable RETURN[TRUE]

    END;

  SetPriority: PUBLIC PROCEDURE [p: Priority] =
    BEGIN
    h: Handle;
    DisableInterrupts[];
    h ← ReadPSB[];
    pda[h].link.priority ← p;
    EnableAndRequeue[@pda.ready, @pda.ready, h]
    END;

  SecondsToTicks: PUBLIC PROCEDURE [sec: CARDINAL] RETURNS [Ticks] =
    BEGIN
    MaxTicks: Ticks = LAST[Ticks];
    ticks: Environment.Long =
      [lc[
        (LONG[sec]*LONG[1000] + ProcessorFace.millisecondsPerTick -
          1)/ProcessorFace.millisecondsPerTick]];
    RETURN[IF ticks.highbits # 0 THEN MaxTicks ELSE ticks.lowbits]
    END;

  -- A timeout of zero ticks is reserved for "timeout disabled".

  SetTimeout: PUBLIC PROCEDURE [condition: LongCONDITIONPtr, ticks: Ticks] = {
    LOOPHOLE[condition, LongConditionPtr].timeout ← MAX[1, ticks]; };

  TicksToMsec: PUBLIC PROCEDURE [ticks: Ticks] RETURNS [Milliseconds] = {
    RETURN[ticks*ProcessorFace.millisecondsPerTick]};
```

```
ValidateProcess: PUBLIC PROCEDURE [p: Index] = {[] ← CheckProcess[p]};

CheckProcess: PROCEDURE [p: Index] RETURNS [h: Handle] =
  BEGIN
  h ← IndexToHandle[p];
  IF p < StartPsb OR p > pda.count + StartPsb OR pda[h].flags.state = frameTaken
    OR pda[h].flags.state = dead THEN ERROR InvalidProcess[p]
  END;

Yield: PUBLIC PROCEDURE =
  BEGIN
  DisableInterrupts[];
  EnableAndRequeue[@pda.ready, @pda.ready, ReadPSB[]]
  END;

-- ENTRY Procedures

Abort: PUBLIC ENTRY PROCEDURE [process: UNSPECIFIED] =
  BEGIN
  h: Handle ← CheckProcess[process ! UNWIND => NULL];
  ProcessInternal.DisableInterrupts[];
  pda[h].flags.abortPending ← TRUE;
  IF pda[h].flags.state = alive AND ~pda[h].link.enterFailed THEN {
    waiting: BOOLEAN ← FALSE;
    IF pda[h].timeout # 0 THEN waiting ← TRUE
    ELSE {
      p, tail: Index ← pda.ready.tail;
      DO
        IF p = HandleToIndex[h] THEN EXIT;
        IF (p ← pda.block[p].link.next) = tail THEN {waiting ← TRUE; EXIT};
        ENDLOOP};
    IF waiting THEN pda[h].timeout ← MAX[CARDINAL[ReadPTC[] + 1], 1]};
  ProcessInternal.EnableInterrupts[];
  END;

Fork: PROCEDURE [root: UNSPECIFIED] RETURNS [Index] =
  BEGIN
  sv: PrincOps.StateVector;
  self: PrincOps.FrameHandle;
  Forker: PROCEDURE [Index];
  newPSB: Index;
  sv ← STATE;
  WHILE ~Enter[@processLock] DO NULL ENDLOOP;
  self ← Frame.MyLocalFrame[];
  Forker ← LOOPHOLE[self.returnlink];
  IF LOOPHOLE[rebirth, Condition].tail = Empty THEN {
    Exit[@processLock]; ERROR TooManyProcesses; };
  newPSB ← pda.block[LOOPHOLE[rebirth, Condition].tail].link.next;
  pda.block[newPSB] ← PSB[
    link:
    [enterFailed: FALSE, priority: pda[ReadPSB[]].link.priority,
      next: pda.block[newPSB].link.next, stateVector: FALSE],
    flags: [state: dead, cleanup: Empty, detached: FALSE, abortPending: FALSE],
    state: self, timeout: 0];
  Notify[@rebirth]; -- wake up newPSB, and set alive; DEPENDS
  pda.block[newPSB].flags.state ← alive;
  -- on new process not preempting parent...
  Frame.SetReturnFrame[PrincOps.NullFrame];
  Forker[newPSB]; -- "returns" handle to site of FORK
```



```
  -- Note that the lines above are executed by the forking process, while
  -- the lines below are executed by the forked process. Note also that the
  -- monitor remains LOCKED during this fancy footwork...!
  sv.dest ← root;
  sv.source ← End;
  Exit[@processLock];
  RETURN WITH sv
  END;

Detach: PUBLIC ENTRY PROCEDURE [process: UNSPECIFIED] =
  BEGIN
  h: Handle = CheckProcess[process];
  pda[h].flags.detached ← TRUE;
  BROADCAST frameTaken
  END;

End: PROCEDURE =
  BEGIN
  sv: RECORD [a: UNSPECIFIED, vec: PrincOps.StateVector];
  frame: DyingFrameHandle;
  h: Handle;
  sv.vec ← STATE;
  WHILE ~Enter[@processLock] DO NULL ENDLOOP;
  frame ← LOOPHOLE[Frame.MyLocalFrame[]];
  frame.state ← alive;
  h ← ReadPSB[];
  pda[h].flags.state ← frameReady;
  pda[h].flags.abortPending ← FALSE; -- too late for Aborts: they no-op
  Broadcast[@frameReady];
  UNTIL pda[h].flags.state = frameTaken OR pda[h].flags.detached DO
    Wait[@processLock, @frameTaken, LOOPHOLE[frameTaken, Condition].timeout];
    WHILE ~ReEnter[@processLock, @frameTaken] DO NULL ENDLOOP;
    ENDLOOP;
  IF deadFrame # NIL THEN {Frame.Free[deadFrame]; deadFrame ← NIL};
  IF pda[h].flags.detached THEN deadFrame ← frame;
  -- leave our frame for freeing
  frame.state ← dead;
  pda[h].flags.state ← dead;
  Broadcast[@dead];
  Wait[@processLock, @rebirth, LOOPHOLE[rebirth, Condition].timeout];
  WHILE ~ReEnter[@processLock, @rebirth] DO NULL ENDLOOP;
  -- dying process exits here; JOINing process does below.
  sv.vec.dest ← frame.returnlink; -- set to site of JOIN by Join
  sv.vec.source ← 0;
  Exit[@processLock];
  RETURN WITH sv.vec
  END;

Join: ENTRY PROCEDURE [process: UNSPECIFIED] RETURNS [PrincOps.FrameHandle] =
  BEGIN
  h: Handle = CheckProcess[process];
  frame: DyingFrameHandle;
  self: PrincOps.FrameHandle = Frame.MyLocalFrame[];
  WHILE pda[h].flags.state # frameReady DO WAIT frameReady ENDLOOP;
  -- Guaranteed to be a dying frame by the time we get here
  frame ← LOOPHOLE[pda[h].state];
  pda[h].flags.state ← frameTaken;
  BROADCAST frameTaken;
```

```
  WHILE frame.state # dead DO WAIT dead ENDLOOP;
  frame.returnlink ← self.returnlink; -- site of JOIN
  RETURN[frame]
  END;

-- Naked notify stuff

NakedNotifyLevel: TYPE = [0..16);

AllocateNakedCondition: PUBLIC PROC
  RETURNS [cv: LONG POINTER TO CONDITION, mask: WORD] =
  BEGIN
  level: NakedNotifyLevel;
  FOR level IN NakedNotifyLevel DO
    mask ← Inline.BITSHIFT[1, LAST[NakedNotifyLevel] - level];
    IF Inline.BITAND[mask, busyLevels] = 0 THEN
      BEGIN
      busyLevels ← Inline.BITOR[busyLevels, mask];
      cv ← LOOPHOLE[@pda.interrupt[level]];
      RETURN
      END;
    ENDLOOP;
  ERROR -- no more levels available (What should this really do?)

  END;

DeallocateNakedCondition: PUBLIC PROC [cv: LONG POINTER TO CONDITION] =
  BEGIN
  level: NakedNotifyLevel;
  FOR level IN NakedNotifyLevel DO
    mask: WORD ← Inline.BITSHIFT[1, LAST[NakedNotifyLevel] - level];
    IF cv = LOOPHOLE[@pda.interrupt[level], LONG POINTER TO CONDITION] THEN
      BEGIN
      busyLevels ← Inline.BITAND[busyLevels, Inline.BITNOT[mask]];
      pda.interrupt[level].tail ← Empty;
      RETURN
      END;
    ENDLOOP;
  ERROR -- handed bogus pointer (What should this really do?)

  END;

Pause: PUBLIC ENTRY PROC [ticks: Process.Ticks] =
  BEGIN c: CONDITION; SetTimeout[@c, ticks]; WAIT c END;

Initialize: PROCEDURE =
  BEGIN
  psb: Index;
  root: PrincOps.FrameHandle;
  DisableTimeout[@dead];
  DisableTimeout[@frameReady];
  DisableTimeout[@frameTaken];
  DisableTimeout[@rebirth];
  SDDefs.SD[SDDefs.sProcessTrap] ← ProcessTrap;
  SDDefs.SD[SDDefs.sFork] ← Fork;
  SDDefs.SD[SDDefs.sJoin] ← Join;

  pda ← PDA;
  pda.currentState ← Inline.LowHalf[@pda.states[defaultPriority]];
```



```
  pda.count ←
    (countPDA*Environment.wordsPerPage - SIZE[ProcessDataArea])/SIZE[PSB];

  -- Fabricate PSB for self
  psb ← StartPsb;
  pda.block[psb] ← PSB[
    link:
    [enterFailed: FALSE, priority: defaultPriority, next: psb,
      stateVector: FALSE],
    flags: [state: alive, cleanup: Empty, detached: TRUE, abortPending: FALSE],
    state: Frame.MyLocalFrame[], timeout: 0];
  pda.ready ← Queue[tail: psb];
  WritePSB[IndexToHandle[psb]];

  -- Complete the illusion that self has been detached
  FOR root ← Frame.MyLocalFrame[], LOOPHOLE[root.returnlink] DO
    IF root.returnlink = PrincOps.TrapLink THEN EXIT ENDLOOP;
  root.returnlink ← LOOPHOLE[End];

  -- Set up free PSB pool ("rebirth" condition)
  THROUGH [1..pda.count) DO
    psb ← psb + 1;
    pda.block[psb] ← PSB[
      link:
      [enterFailed: FALSE, priority: defaultPriority, next: psb - 1,
        stateVector: FALSE],
      flags:
      [state: dead, cleanup: Empty, detached: FALSE, abortPending: FALSE],
      state: PrincOps.NullFrame, timeout: 0];
    ENDLOOP;
  pda.block[StartPsb + 1].link.next ← psb;
  LOOPHOLE[rebirth, Condition].tail ← psb;

  -- Initialize naked-notify allocator
  busyLevels ← ProcessorFace.reservedNakedNotifyMask;

  -- Start interrupts
  ProcessOperations.WriteWDC[0];
  END;

Initialize[];

END.
```

(For earlier log entries see Pilot 4.0 archive version.)

April 29, 1980 5:53 PM	*Forrest*	*Drop PSB's crossing pages on floor; move Initialize Tim*
***eout machinery to ProcessorHead*		
May 3, 1980 10:46 AM	*Forrest*	*Mesa 6.0 Conversion*
May 14, 1980 6:19 PM	*McJones*	*Use OISProcessorFace.reservedNakedNotifyMask; star*
***t interrupts at end of initialization (as before)*		
June 23, 1980 5:34 PM	*McJones*	*OISProcessorFace = >ProcessorFace*
August 5, 1980 5:54 PM	*Sandman*	*New PSB format*
September 29, 1980 10:30 AM	*Johnsson*	*New ProcessTrap*

```
-- MesaRuntime>Signals.Mesa  (last edited by: Forrest on: May 3, 1980  11:08 AM)

DIRECTORY
  Environment: FROM "Environment" USING [Byte],
  Frame: FROM "Frame" USING [
    Alloc, Free, GetReturnFrame, GetReturnLink, MyGlobalFrame, MyLocalFrame,
    SetReturnLink],
  Mopcodes: FROM "Mopcodes" USING [
    zCATCH, zJ2, zJ9, zJB, zJW, zKFCB, zPORTI, zSLB],
  PrincOps: FROM "PrincOps" USING [
    ControlLink, Frame, FrameHandle, localbase, NullFrame, StateVector, BytePC],
  RuntimeInternal: FROM "RuntimeInternal" USING [Codebase],
  RuntimePrograms: FROM "RuntimePrograms",
  SDDefs: FROM "sddefs" USING [
    SD, sError, sErrorList, sReturnError, sReturnErrorList, sSignal, sSignalList,
    sUncaughtSignal, sUnnamedError];

Signals: PROGRAM
  IMPORTS Frame, RuntimeInternal EXPORTS RuntimeInternal, RuntimePrograms =

  BEGIN

  BYTE: TYPE = Environment.Byte;

  CatchPointer: TYPE = POINTER TO catch PrincOps.Frame;
  CatchCall: TYPE = PROCEDURE [SIGNAL] RETURNS [ActionCode];
  CatchContinue: TYPE = PROCEDURE;

  ActionCode: TYPE = INTEGER;
  reject: ActionCode = 0;
  resume: ActionCode = 1;
  exit: ActionCode = -1;

  SendMsgSignal: PUBLIC SIGNAL RETURNS [UNSPECIFIED, UNSPECIFIED] = CODE;

  signalling: CARDINAL = 177777B;
  notSignalling: CARDINAL = 0;

  MarkSignalFrame: PROCEDURE [value: CARDINAL] = MACHINE CODE
    BEGIN Mopcodes.zSLB, 3 --OFFSET[mark]-- END;

  SignalHandler: PROCEDURE [signal: SIGNAL, message: UNSPECIFIED] =
    BEGIN

    SignalFrame: TYPE = POINTER TO FRAME[SignalHandler];

    frame, nextFrame: PrincOps.FrameHandle;
    target, nextTarget: PrincOps.FrameHandle;
    self: PrincOps.FrameHandle = Frame.MyLocalFrame[];
    start: PrincOps.FrameHandle;
    catchFrame: CatchPointer;
    action: ActionCode;
    unwinding: BOOLEAN;
    catchPhrase: BOOLEAN;
    catchFSIndex: BYTE;
    catchPC, exitPC: PrincOps.BytePC;
    catchState: PrincOps.StateVector;

    MarkSignalFrame[signalling];
```



```
    unwinding ← FALSE;
    start ← GetFrame[self.returnlink];
    target ← PrincOps.NullFrame;
    DO
      nextFrame ← start;
      UNTIL nextFrame = target DO
        frame ← nextFrame;
        IF frame.accesslink = Frame.MyGlobalFrame[] AND frame.mark THEN
          BEGIN OPEN thisSignaller: LOOPHOLE[frame, SignalFrame];
          IF unwinding THEN
            BEGIN
            IF signal = thisSignaller.signal THEN
              nextTarget ←
                IF thisSignaller.unwinding THEN thisSignaller.nextTarget
                ELSE thisSignaller.nextFrame;
            IF thisSignaller.unwinding THEN
              BEGIN
              IF thisSignaller.frame = LOOPHOLE[frame.returnlink] THEN
                frame.returnlink ← [frame[thisSignaller.nextFrame]];
              Frame.Free[thisSignaller.frame];
              END;
            nextFrame ← GetFrame[frame.returnlink];
            END
          ELSE
            nextFrame ←
              IF signal # thisSignaller.signal THEN
              IF thisSignaller.unwinding THEN thisSignaller.nextFrame
              ELSE GetFrame[frame.returnlink]
              ELSE
                IF thisSignaller.unwinding THEN thisSignaller.nextTarget
                ELSE thisSignaller.nextFrame;
          END
        ELSE nextFrame ← GetFrame[frame.returnlink];
        IF unwinding AND nextTarget = frame THEN nextTarget ← nextFrame;
        [catchPhrase, catchFSIndex, catchPC] ← CheckCatch[frame];
        IF catchPhrase THEN
          BEGIN
          catchFrame ← Frame.Alloc[catchFSIndex];
          catchFrame↑ ← PrincOps.Frame[
            accesslink: frame.accesslink, pc: catchPC, returnlink: [frame[self]],
            extensions: catch[
            unused:, staticlink: frame + PrincOps.localbase,
            messageval: message]];
          action ← LOOPHOLE[catchFrame, CatchCall][
            IF unwinding THEN LOOPHOLE[UNWIND] ELSE signal !
            SendMsgSignal => RESUME [message, signal]];
          catchState ← STATE;
          SELECT action FROM
            reject => NULL;
            resume =>
              IF unwinding THEN ERROR ResumeError
              ELSE
                BEGIN
                catchState.dest ← Frame.GetReturnLink[];
                catchState.source ← 0;
                RETURN WITH catchState;
                END;
            exit =>
              BEGIN
              -- catchFrame is waiting to execute its exit jump
```

```
            exitPC ← catchFrame.pc;
            Frame.Free[catchFrame];
            target ← LOOPHOLE[catchState.stk[0] - PrincOps.localbase];
            nextTarget ← nextFrame;
            unwinding ← TRUE;
            message ← NIL;
            GO TO StartUnwind;
            END;
          ENDCASE;
        END;
      IF unwinding THEN
        BEGIN
        IF frame = start THEN start ← nextFrame;
        IF frame = LOOPHOLE[self.returnlink] THEN
          self.returnlink ← [frame[nextFrame]];
        Frame.Free[frame];
        END;
      REPEAT StartUnwind => NULL; FINISHED => EXIT
      ENDLOOP;
    ENDLOOP;
  IF unwinding THEN target.pc ← exitPC
  ELSE UncaughtSignal[message, signal, frame];
  -- formerly Punted if no uncaught signal catcher
  RETURN
  END;

CheckCatch: PROCEDURE [frame: PrincOps.FrameHandle]
  RETURNS [catchPhrase: BOOLEAN, fsIndex: BYTE, pc: PrincOps.BytePC] =
  BEGIN OPEN Mopcodes;
  code: LONG POINTER TO PACKED ARRAY [0..0) OF BYTE;
  MarkSignalFrame[notSignalling];
  code ← RuntimeInternal.Codebase[LOOPHOLE[frame.accesslink, PROGRAM]];
  pc ← [frame.pc];
  DO
    SELECT code[pc] FROM
      zCATCH => BEGIN catchPhrase ← TRUE; EXIT END;
      zPORTI => pc ← [pc + 1];
      ENDCASE => BEGIN catchPhrase ← FALSE; EXIT END
    ENDLOOP;
  IF catchPhrase THEN
    BEGIN -- code[pc] points at zCatch
    fsIndex ← code[pc + 1];
    SELECT code[pc ← [pc + 2]] FROM
      zJB => pc ← [pc + 2];
      IN [zJ2..zJ9] => pc ← [pc + 1];
      zJW => pc ← [pc + 3];
      ENDCASE
    END;
  RETURN
  END;

GetFrame: PROCEDURE [link: PrincOps.ControlLink]
  RETURNS [PrincOps.FrameHandle] =
  BEGIN
  -- MarkSignalFrame[notSignalling];
  DO
    IF ~link.proc THEN
      IF link.indirect THEN link ← link.link↑
      ELSE --frame link--RETURN[link.frame]
```



```
    ELSE RETURN[PrincOps.NullFrame];
    ENDLOOP;
  END;


Signal: PROCEDURE [signal: SIGNAL, message: UNSPECIFIED] = SignalHandler;

SignalList: PROCEDURE [signal: SIGNAL, message: POINTER TO UNSPECIFIED] =
  BEGIN
  MarkSignalFrame[notSignalling];
  SignalHandler[signal, message ! UNWIND => Frame.Free[message]];
  Frame.Free[message];
  RETURN
  END;


ResumeError: PUBLIC SIGNAL = CODE;

Error: PROCEDURE [signal: SIGNAL, message: UNSPECIFIED] =
  BEGIN
  MarkSignalFrame[notSignalling];
  SignalHandler[signal, message];
  ERROR ResumeError
  END;

ErrorList: PROCEDURE [signal: SIGNAL, message: POINTER TO UNSPECIFIED] =
  BEGIN
  MarkSignalFrame[notSignalling];
  SignalHandler[signal, message ! UNWIND => Frame.Free[message]];
  Frame.Free[message];
  ERROR ResumeError
  END;

ReturnError: PROCEDURE [signal: SIGNAL, message: UNSPECIFIED] =
  BEGIN
  caller: PrincOps.FrameHandle = Frame.GetReturnFrame[];
  Frame.SetReturnLink[caller.returnlink];
  MarkSignalFrame[notSignalling];
  SignalHandler[signal, message ! UNWIND => Frame.Free[caller]];
  Frame.Free[caller];
  ERROR ResumeError
  END;

ReturnErrorList: PROCEDURE [signal: SIGNAL, message: POINTER TO UNSPECIFIED] =
  BEGIN
  caller: PrincOps.FrameHandle = Frame.GetReturnFrame[];
  Frame.SetReturnLink[caller.returnlink];
  MarkSignalFrame[notSignalling];
  SignalHandler[
    signal, message ! UNWIND => {Frame.Free[caller]; Frame.Free[message]; }];
  Frame.Free[caller];
  Frame.Free[message];
  ERROR ResumeError
  END;

UnnamedError: PROCEDURE =
  BEGIN
  MarkSignalFrame[notSignalling];
  SignalHandler[LOOPHOLE[-1], -1];
```

```
    ERROR ResumeError
    END;

  UncaughtSignal: PROCEDURE [
    msg, signal: UNSPECIFIED, frame: PrincOps.FrameHandle] = MACHINE CODE
    BEGIN Mopcodes.zKFCB, SDDefs.sUncaughtSignal END;


  Initialize: PROCEDURE =
    BEGIN OPEN SDDefs;
    pSD: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
    pSD[sSignalList] ← SignalList;
    pSD[sSignal] ← Signal;
    pSD[sErrorList] ← ErrorList;
    pSD[sError] ← Error;
    pSD[sReturnErrorList] ← ReturnErrorList;
    pSD[sReturnError] ← ReturnError;
    pSD[sUnnamedError] ← UnnamedError;
    END;

  Initialize[];

  END.
```

LOG

Time: August 7, 1978 8:57 AM By: Sandman Action: Got Codebase from RuntimeInternal
***instead of CodebaseDefs*
Time: August 30, 1978 2:06 PM By: Sandman Action: Removed TrapDefs and PuntMesa
Time: March 13, 1979 3:15 PM By: McJones Action: Mesa 5
Time: April 4, 1979 11:02 AM By: McJones Action: Added (conditional) BytePC logic
Time: September 21, 1979 2:37 PM By: McJones Action: Removed (conditional) By
***tePC logic*
Time: May 3, 1980 11:07 AM By: Forrest Action: Mesa 6.0 Conversion

```
-- SnapshotImpl.mesa (last edited by: McJones on: September 18, 1980 9:45 AM)

DIRECTORY
  Boot USING [Location, LVBootFiles, PVBootFiles],
  BootFile USING [currentVersion, Header, MemorySizeToFileSize],
  BootSwap USING [InLoad, OutLoad],
  Device USING [Type],
  DeviceCleanup USING [Perform],
  Environment USING [PageCount, wordsPerPage],
  File USING [Capability, GetAttributes, PageNumber, Type, Unknown],
  FileTypes USING [tUntypedFile],
  KernelFile USING [GetBootLocation],
  PhysicalVolume USING [
    GetAttributes, GetContainingPhysicalVolume, ID, InterpretHandle],
  PilotMP USING [cClient],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  ProcessOperations USING [
    ReadPSB, ReadPTC, ReadWDC, WritePSB, WritePTC, WriteWDC],
  ProcessorFace USING [BootButton, SetMP],
  PSB USING [Handle],
  RuntimePrograms USING [],
  Snapshot USING [],
  Space USING [
    Create, Handle, LongPointer, Map, nullHandle, Unmap, virtualMemory],
  SpecialFile USING [InvalidParameters, Link, MakeBootable, MakeUnbootable],
  SpecialVolume USING [
    GetLogicalVolumeBootFiles, GetPhysicalVolumeBootFiles,
    SetLogicalVolumeBootFiles, SetPhysicalVolumeBootFiles],
  StoragePrograms USING [RecoverMStore],
  TemporaryBooting USING [Switches],
  TemporarySetGMT USING [SetGMT],
  Volume USING [ID];

SnapshotImpl: MONITOR -- just to protect space used by MakeBootable


  IMPORTS
    Boot, BootFile, BootSwap, DeviceCleanup, File, KernelFile, PhysicalVolume,
    ProcessInternal, ProcessOperations, ProcessorFace, Space, SpecialFile,
    SpecialVolume, StoragePrograms, TemporarySetGMT
  EXPORTS RuntimePrograms, Snapshot, TemporaryBooting
  SHARES File =

  BEGIN

  Switches: TYPE = TemporaryBooting.Switches;

  --
  -- RuntimePrograms

  InitializeSnapshot: PUBLIC PROCEDURE = BEGIN END; -- start module

  --
  -- Snapshot

  OutLoad: PUBLIC PROCEDURE [file: File.Capability, firstPage: File.PageNumber]
    RETURNS [inLoaded: BOOLEAN] =
    BEGIN
    location: disk Boot.Location;
```



```
    psb: PSB.Handle;
    ptc: CARDINAL;
    wdc: CARDINAL;
    location.diskFileID ← [fID: file.fID, firstPage: firstPage, da:];
    [deviceType: location.deviceType, deviceOrdinal: location.deviceOrdinal,
      link: LOOPHOLE[location.diskFileID.da, SpecialFile.Link]] ←
      KernelFile.GetBootLocation[file, firstPage];
    -- Save process state not captured in PDA:
    ProcessInternal.DisableInterrupts[]; -- make it hold still first
    psb ← ProcessOperations.ReadPSB[];
    ptc ← ProcessOperations.ReadPTC[];
    wdc ← ProcessOperations.ReadWDC[];
    DeviceCleanup.Perform[turnOff]; -- turn all devices off
    inLoaded ← BootSwap.OutLoad[@location, restore] ~= outLoaded;
    IF inLoaded THEN
      BEGIN
      -- Restore process state not captured in PDA.
      ProcessOperations.WriteWDC[wdc];
      ProcessOperations.WritePTC[ptc];
      ProcessOperations.WritePSB[psb];
      -- The following is a temporary substitute for a clock chip.
      -- We must do it with interrupts off or Communication will be using the Ethernet--
      -- if we get an allocation trap, all is lost.
      TemporarySetGMT.SetGMT[];
      END;
    DeviceCleanup.Perform[turnOn]; -- turn devices back on
    ProcessorFace.SetMP[PilotMP.cClient]; -- announce our return
    ProcessInternal.EnableInterrupts[];
    END;

  InLoadFromBootLocation: PUBLIC PROCEDURE [
    pMicrocode, pGerm: LONG POINTER, countGerm: Environment.PageCount,
    location: POINTER TO Boot.Location, switches: Switches] =
    BEGIN
    ProcessInternal.DisableInterrupts[]; -- stop other processes
    -- Ensure all real memory is mapped somewhere in virtual address space:
    StoragePrograms.RecoverMStore[];
    DeviceCleanup.Perform[turnOff]; -- turn all devices off
    -- Disconnect all devices (e.g. release funny memory):
    DeviceCleanup.Perform[disconnect];
    -- The next line should be in BootSwap.InLoad but blows up the compiler
    IF pMicrocode ~= NIL THEN DeviceCleanup.Perform[kill];
    -- Copy switches to communication area (in germ's SD) (must do in BootSwap.InLoad if pGer
**m~ = NIL)
    BootSwap.InLoad[
      pMicrocode, pGerm, countGerm*Environment.wordsPerPage, location, switches]
    -- Can't get here...

    END;

  InLoad: PUBLIC PROCEDURE [
    pMicrocode, pGerm: LONG POINTER, countGerm: Environment.PageCount,
    file: File.Capability, firstPage: File.PageNumber, switches: Switches] =
    BEGIN
    location: disk Boot.Location;
    location.diskFileID ← [fID: file.fID, firstPage: firstPage, da: NULL];
    [deviceType: location.deviceType, deviceOrdinal: location.deviceOrdinal,
      link: LOOPHOLE[location.diskFileID.da, SpecialFile.Link]] ←
```

```
    KernelFile.GetBootLocation[file, firstPage];
  InLoadFromBootLocation[pMicrocode, pGerm, countGerm, @location, switches]
  -- Can't get here...

  END;

  --
  -- TemporaryBooting

  tBootFile: PUBLIC File.Type ← FileTypes.tUntypedFile; -- delete this soon...

  MakeBootable: PUBLIC PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber] =
    BEGIN
    count: Environment.PageCount = GetBootFileSize[@file, firstPage];
    [] ← SpecialFile.MakeBootable[
      file: file, firstPage: firstPage, count: count, lastLink: LOOPHOLE[LONG[0]]
      ! SpecialFile.InvalidParameters => ERROR InvalidParameters];
    END;

  sh: Space.Handle ← Space.nullHandle;
  -- this would be local if UtilityPilot could delete spaces

  GetBootFileSize: ENTRY PROCEDURE [
    pFile: POINTER TO File.Capability, firstPage: File.PageNumber]
    RETURNS [count: Environment.PageCount] =
    BEGIN
    pHeader: LONG POINTER TO BootFile.Header;
    error: BOOLEAN ← FALSE;
    IF sh = Space.nullHandle THEN
      sh ← Space.Create[size: 1, parent: Space.virtualMemory];
    pHeader ← Space.LongPointer[sh];
    BEGIN
    Space.Map[
      sh, [pFile↑, firstPage] !
      File.Unknown => BEGIN error ← TRUE; GO TO Unmapped END];
    IF pHeader.version ~= BootFile.currentVersion THEN error ← TRUE
    ELSE count ← BootFile.MemorySizeToFileSize[pHeader.countData];
    Space.Unmap[sh];
    EXITS Unmapped => NULL
    END;
    IF error THEN RETURN WITH ERROR InvalidParameters;
    END;

  MakeUnbootable: PUBLIC PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber] =
    BEGIN
    SpecialFile.MakeUnbootable[
      file: file, firstPage: firstPage, count: GetBootFileSize[@file, firstPage] !
      SpecialFile.InvalidParameters => ERROR InvalidParameters];
    END;

  InstallVolumeBootFile: PUBLIC ENTRY PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber] =
    BEGIN
    bootFiles: Boot.LVBootFiles;
    volume: Volume.ID = File.GetAttributes[file].volume;
    link: SpecialFile.Link = KernelFile.GetBootLocation[file, firstPage].link;
    SpecialVolume.GetLogicalVolumeBootFiles[volume, @bootFiles];
```



```
    bootFiles[pilot] ← [file.fID, firstPage, LOOPHOLE[link]];
    SpecialVolume.SetLogicalVolumeBootFiles[volume, @bootFiles];
    END;

  InstallPhysicalVolumeBootFile: PUBLIC ENTRY PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber] =
    BEGIN
    bootFiles: Boot.PVBootFiles;
    link: SpecialFile.Link = KernelFile.GetBootLocation[file, firstPage].link;
    pvID: PhysicalVolume.ID = PhysicalVolume.GetContainingPhysicalVolume[
      File.GetAttributes[file].volume];
    SpecialVolume.GetPhysicalVolumeBootFiles[pvID, @bootFiles];
    bootFiles[pilot] ← [file.fID, firstPage, LOOPHOLE[link]];
    SpecialVolume.SetPhysicalVolumeBootFiles[pvID, @bootFiles];
    END;

  BootFromFile: PUBLIC PROCEDURE [
    file: File.Capability, firstPage: File.PageNumber, switches: Switches] =
    BEGIN
    InLoad[
      pMicrocode: NIL, pGerm: NIL, countGerm: 0, file: file, firstPage: firstPage,
      switches: switches]
    END;

  BootFromVolume: PUBLIC PROCEDURE [volume: Volume.ID, switches: Switches] =
    BEGIN
    bootFiles: Boot.LVBootFiles;
    location: Boot.Location;
    SpecialVolume.GetLogicalVolumeBootFiles[volume, @bootFiles];
    location.diskFileID ← bootFiles[pilot];
    [deviceType: location.deviceType, deviceOrdinal: location.deviceOrdinal] ←
      GetDevice[PhysicalVolume.GetContainingPhysicalVolume[volume]];
    InLoadFromBootLocation[
      pMicrocode: NIL, pGerm: NIL, countGerm: 0, location: @location,
      switches: switches]
    END;

  GetDevice: PROCEDURE [pvID: PhysicalVolume.ID]
    RETURNS [deviceType: Device.Type, deviceOrdinal: CARDINAL] =
    BEGIN OPEN PhysicalVolume;
    [type: deviceType, index: deviceOrdinal] ← InterpretHandle[
      GetAttributes[pvID].instance];
    END;

  -- This procedure should be changed to take a PhysicalVolume.ID:

  BootFromPhysicalVolume: PUBLIC PROCEDURE [
    volume: Volume.ID, switches: Switches] =
    BEGIN
    pvID: PhysicalVolume.ID = PhysicalVolume.GetContainingPhysicalVolume[volume];
    bootFiles: Boot.PVBootFiles;
    location: Boot.Location;
    SpecialVolume.GetPhysicalVolumeBootFiles[pvID, @bootFiles];
    location.diskFileID ← bootFiles[pilot];
    [deviceType: location.deviceType, deviceOrdinal: location.deviceOrdinal] ←
      GetDevice[pvID];
    InLoadFromBootLocation[
      pMicrocode: NIL, pGerm: NIL, countGerm: 0, location: @location,
```

```
    switches: switches]
  END;

BootButton: PUBLIC PROCEDURE [switches: Switches] =
  BEGIN
  -- WHAT SHOULD WE DO WITH THE SWITCHES?
  ProcessorFace.BootButton[] -- never returns

  END;

InvalidParameters: PUBLIC ERROR = CODE;

END.
```

(For earlier log entries see Pilot 4.0 archive version.)
April 14, 1980 11:04 AM Knutsen MStore.Recover = >StoragePrograms.RecoverMStore;
***module now STARTed by InitializeSnapshot[]*
April 16, 1980 5:39 PM McJones ControlPrograms.InitializeGMT = >TemporarySetGMT.
***SetGMT*
April 28, 1980 10:06 AM Forrest FrameOps = >Frame
June 25, 1980 10:27 AM McJones OISProcessorFace = >ProcessorFace; disconnect devi
***ces in InLoad*
July 25, 1980 4:57 PM Luniewski Ops and defs moved to PhysicalVolume from SpecialV
***olume*
August 15, 1980 9:53 AM McJones tBootFile = >tUntypedFile; save wdc in OutLoad
September 9, 1980 2:04 PM McJones OutLoad shouldn't SetMP[cClient] until after turning de
***vices back on*
September 9, 1980 2:10 PM McJones OutLoad must save/restore state no longer part of PDA
September 17, 1980 3:14 PM McJones Add InLoadFromBootLocation, and call it from BootFro
***m[Physical]Volume*

```
-- Traps.mesa (last edited by: Forrest on: October 3, 1980 2:00 PM)

DIRECTORY
  Environment USING [Byte, wordsPerPage],
  Frame USING [Free, GetReturnFrame, GetReturnLink, MyLocalFrame, SetReturnFrame],
  Inline USING [BITAND, BITOR],
  Mopcodes USING [zDESCBS, zLDIV, zPORTI, zPORTO, zSFC],
  ProcessorFace USING [SetMP],
  PilotMP USING [cEarlyTrap, Code],
  PrincOps USING [
    AV, AVItem, ControlLink, ControlModule, FrameCodeBase, FrameHandle, FrameVec,
    GlobalFrameHandle, LargeReturnSlot, MainBodyIndex, NullControl, NullFrame,
    NullGlobalFrame, Port, PortHandle, PrefixHandle, SpecialReturnSlot,
    StateVector, SVPointer],
  PrincOpsRuntime USING [GetFrame, GFT],
  ProcessInternal USING [DisableInterrupts, EnableInterrupts],
  ResidentMemory USING [AllocateMDS, FreeMDS],
  Runtime USING [ValidateGlobalFrame],
  RuntimeInternal USING [WorryCallDebugger],
  RuntimePrograms USING [],
  SDDefs USING [
    sAllocTrap, sBoundsFault, sControlFault, SD, sDivideCheck, sError,
    sHardwareError, sPointerFault, sRestart, sSignal, sStackError, sStart,
    sSwapTrap, sUnbound, sWakeupError, sXferTrap, sZeroDivisor],
  Trap USING [ReadATP, ReadOTP],
  XferTrap USING [WriteXTS];

Traps: PROGRAM
  IMPORTS
    Frame, Inline, PrincOpsRuntime, ProcessInternal, ProcessorFace,
    ResidentMemory, Trap, Runtime, RuntimeInternal, XferTrap
  EXPORTS Runtime, RuntimeInternal, RuntimePrograms =
  BEGIN OPEN PrincOps;

  BoundsFault: PUBLIC SIGNAL = CODE;
  ControlFault: PUBLIC SIGNAL [source: FrameHandle] RETURNS [ControlLink] = CODE;
  DivideCheck: PUBLIC SIGNAL = CODE;
  HardwareError: ERROR = CODE;
  LinkageFault: PUBLIC ERROR = CODE;
  PointerFault: PUBLIC SIGNAL = CODE;
  PortFault: PUBLIC ERROR = CODE;
  StartFault: PUBLIC SIGNAL [dest: PROGRAM] = CODE;
  StackError: PUBLIC ERROR = CODE;
  UnboundProcedure: PUBLIC SIGNAL [dest: ControlLink] RETURNS [ControlLink] =
    CODE;
  WakeupError: ERROR = CODE;
  ZeroDivisor: PUBLIC SIGNAL = CODE;

  CodeBytesPtr: TYPE = LONG POINTER TO PACKED ARRAY [0..0) OF Environment.Byte;

  -- Non-trap type procedures
  Codebase, GetTableBase: PUBLIC PROC [frame: PROGRAM] RETURNS [l: LONG POINTER] =
    BEGIN OPEN c: LOOPHOLE[l, FrameCodeBase];
    c ← LOOPHOLE[frame, GlobalFrameHandle].code;
    c.out ← FALSE;
    END;

  -- assumes code is swapped in (out bit off)
```



```
  BumpPC: PROCEDURE [f: FrameHandle, i: INTEGER] = INLINE {f.pc ← [f.pc + i]};

  GetCodeBytes: PROC [frame: FrameHandle] RETURNS [CodeBytesPtr] = INLINE {
    RETURN[LOOPHOLE[frame.accesslink.code.longbase]]};

  FrameSize: PUBLIC PROC [fsi: CARDINAL] RETURNS [CARDINAL] = {
    RETURN[FrameVec[fsi]]};

  MainBody: PROC [GlobalFrameHandle] RETURNS [ControlLink] = MACHINE CODE
    BEGIN Mopcodes.zDESCBS, PrincOps.MainBodyIndex END;

  Restart: PUBLIC PROC [dest: GlobalFrameHandle] =
    BEGIN
    stops: BOOLEAN;
    frame: FrameHandle;
    IF dest = NullGlobalFrame THEN ERROR StartFault[LOOPHOLE[dest, PROGRAM]];
    IF ~dest.started THEN Start[[frame[dest]]];
    stops ← LOOPHOLE[Codebase[LOOPHOLE[dest, PROGRAM]], LONG
      PrefixHandle].header.info.stops;
    IF ~stops THEN ERROR StartFault[LOOPHOLE[dest, PROGRAM]];
    IF (frame ← dest.global[0]) # NullFrame THEN
      BEGIN
      frame.returnlink ← Frame.GetReturnLink[];
      Frame.SetReturnFrame[frame]
      END
    END;

  Start: PUBLIC PROCEDURE [cm: ControlModule] =
    BEGIN
    CM: PROGRAM;
    FramesStarted: PROCEDURE RETURNS [BOOLEAN] = INLINE
      BEGIN OPEN PrincOpsRuntime;
      dest: ControlLink = LOOPHOLE[Runtime.ValidateGlobalFrame];
      RETURN[GetFrame[GFT[dest.gfi]].started];
      END;
    state: StateVector;
    state ← STATE;
    CM ← LOOPHOLE[cm];
    IF ~cm.multiple THEN
      BEGIN OPEN f: cm.frame;
      IF @f = NullGlobalFrame OR f.started THEN ERROR StartFault[CM];
      -- don't start trap module Frames early!!
      IF FramesStarted[] THEN Runtime.ValidateGlobalFrame[@f];
      StartCM[f.global[0], @f, @state];
      IF ~f.started THEN {f.started ← TRUE; StartWithState[@f, @state]; }
      ELSE IF state.stkptr # 0 THEN SIGNAL StartFault[CM];
      END
    ELSE
      BEGIN
      StartCM[cm, NIL, NIL];
      IF state.stkptr # 0 THEN SIGNAL StartFault[CM];
      END;
    RETURN
    END;

  StartCM: PROCEDURE [
    cm: ControlModule, frame: GlobalFrameHandle, state: SVPointer] =
    BEGIN
    Call: PROCEDURE [ControlLink] = MACHINE CODE BEGIN Mopcodes.zSFC END;
    SELECT TRUE FROM
```

```
    cm = NullControl => RETURN;
    cm.multiple =>
      BEGIN
      length: CARDINAL;
      cm.multiple ← FALSE;
      IF (length ← cm.list.nModules) = 0 THEN RETURN;
      cm.list.nModules ← 0;
      FOR i: CARDINAL IN [0..length) DO
        StartCM[[frame[cm.list.frames[i]]], frame, state]; ENDLOOP;
      Frame.Free[cm.list];
      END;
    cm.frame.started => RETURN;
    ENDCASE =>
      BEGIN
      control: ControlModule ← cm.frame.global[0];
      IF control # cm THEN StartCM[control, frame, state];
      IF ~cm.frame.started THEN
        BEGIN
        cm.frame.started ← TRUE;
        IF frame # cm.frame THEN Call[MainBody[cm.frame]]
        ELSE StartWithState[frame, state];
        END;
      END;
  RETURN
  END;

StartWithState: PROC [frame: GlobalFrameHandle, state: SVPointer] =
  BEGIN
  s: StateVector ← state↑;
  retFrame: FrameHandle ← Frame.GetReturnLink[].frame;
  s.dest ← MainBody[frame];
  s.source ← retFrame.returnlink;
  Frame.Free[retFrame];
  RETURN WITH s;
  END;

-- Frame Trap and Allocation

FrameSegment: TYPE = POINTER TO FrameSegmentHeader;
FrameSegmentHeader: TYPE = MACHINE DEPENDENT RECORD [
  link: FrameSegment, pages: CARDINAL, size, fsi: CARDINAL];

frameListHead: FrameSegment ← NIL;
-- maintain a list of all new "permanent" frame segments
LargeFrameSlot: CARDINAL = 12;
PageSize: CARDINAL = Environment.wordsPerPage;
ExtraSpaceSize: CARDINAL = PageSize;
ExtraSpace: ARRAY [0..ExtraSpaceSize) OF WORD;
InitNewSpace: POINTER = LOOPHOLE[Inline.BITOR[BASE[ExtraSpace], 3]];
InitWordsLeft: CARDINAL = BASE[ExtraSpace] + ExtraSpaceSize - InitNewSpace;

AllocTrap: PROC [otherframe: FrameHandle] RETURNS [myframe: FrameHandle] =
  BEGIN
  ATFrame: TYPE = POINTER TO FRAME[AllocTrap];
  state: StateVector;
  newframe: FrameHandle;
  fsize, findex: CARDINAL;
  p: POINTER;
  newspace: FrameSegment;
```



```
  NewSpacePtr: POINTER;
  NULLPtr: FrameHandle = LOOPHOLE[0];
  WordsLeft: CARDINAL ← 0;
  recurring: BOOLEAN ← otherframe = NULLPtr;

  myframe ← Frame.MyLocalFrame[];
  state.dest ← myframe.returnlink;
  state.source ← 0;
  state.instbyte ← 0;
  state.stk[0] ← myframe;
  state.stkptr ← 1;

  ProcessInternal.DisableInterrupts[]; -- so that undo below works

  DO


    IF ~recurring THEN
    BEGIN
    LOOPHOLE[otherframe, ATFrame].NewSpacePtr ← InitNewSpace;
    LOOPHOLE[otherframe, ATFrame].WordsLeft ← InitWordsLeft;
    AV[SpecialReturnSlot] ← [data[0, empty]]
    END;

  ProcessInternal.EnableInterrupts[]; -- guarantees one more instruction
  TRANSFER WITH state;
  ProcessInternal.DisableInterrupts[];

  state ← STATE;
  findex ← LOOPHOLE[Trap.ReadATP[]];
  SDDefs.SD[SDDefs.sAllocTrap] ← otherframe;
  IF ~recurring THEN FlushLargeFrames[]
  ELSE
    IF (p ← AV[SpecialReturnSlot].link) # LOOPHOLE[AVItem[data[0, empty]]]
      THEN
      BEGIN
      WordsLeft ← WordsLeft + (NewSpacePtr - p + 1);
      NewSpacePtr ← p - 1;
      AV[SpecialReturnSlot] ← [data[0, empty]]
      END;
  IF findex < LargeFrameSlot THEN
    BEGIN
    fsize ← FrameVec[findex] + 1; -- includes overhead word
    THROUGH [0..1] DO
      p ← NewSpacePtr + 1;
      IF fsize <= WordsLeft THEN
        BEGIN
        newframe ← p;
        (p - 1)↑ ← IF recurring THEN SpecialReturnSlot ELSE findex;
        WordsLeft ← WordsLeft - fsize;
        NewSpacePtr ← NewSpacePtr + fsize;
        EXIT
        END
      ELSE
        BEGIN
        FOR i: CARDINAL DECREASING IN [0..findex) DO
          IF FrameVec[i] < WordsLeft THEN {
            (p - 1)↑ ← i; p↑ ← AV[i].link; AV[i].link ← p; EXIT}
          ENDLOOP;
        NewSpacePtr ← (newspace ← ResidentMemory.AllocateMDS[1]) + 3;
```

```
        newspace↑ ← [link: frameListHead, pages: 1, size:, fsi:];
        frameListHead ← newspace;
        WordsLeft ← PageSize - 3
        END
      ENDLOOP
    END
  ELSE
    BEGIN
    pages: CARDINAL;
    fsize ← FrameVec[findex];
    newspace ← ResidentMemory.AllocateMDS[
      pages ← (fsize + PageSize + 3)/PageSize];
    newspace↑ ← [link: NIL, pages: pages, size: fsize, fsi: LargeReturnSlot];
    newframe ← LOOPHOLE[newspace + 4]
    END;
  newframe.accesslink ← LOOPHOLE[AV[findex].link];
  AV[findex].frame ← newframe;
  state.source ← 0;
  state.dest ← myframe.returnlink;
  SDDefs.SD[SDDefs.sAllocTrap] ← myframe
  ENDLOOP
END;

FlushLargeFrames: PUBLIC PROC = INLINE
  BEGIN
  ItemPointer: TYPE = POINTER TO AVItem;
  p: POINTER;
  frameSpace: FrameSegment;
  item: ItemPointer ← @AV[LargeReturnSlot];
  WHILE item.tag = frame DO
    p ← item.frame;
    item.frame ← p↑;
    frameSpace ← p - 4;
    ResidentMemory.FreeMDS[frameSpace, frameSpace.pages]
    ENDLOOP
  END;

-- Other traps

BoundsFaultTrap: PROC =
  BEGIN
  frame: PrincOps.FrameHandle;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  state.v.dest ← frame ← Frame.GetReturnFrame[];
  BumpPC[frame, 1];
  state.v.stkptr ← state.v.stkptr - 1;
  SIGNAL BoundsFault; -- pc is advanced on this trap
  RETURN WITH state.v
  END;

CodeTrap: PROC =
  BEGIN OPEN PrincOpsRuntime;
  dest: ControlLink;
  state: StateVector;
  frame: GlobalFrameHandle;
  state ← STATE;
  dest ← Trap.ReadOTP[];
  state.dest ← Frame.GetReturnLink[];
  DO
```



```
    IF dest.proc THEN {frame ← GetFrame[GFT[dest.gfi]]; EXIT}
    ELSE
      IF dest.indirect THEN dest ← dest.link↑
      ELSE {frame ← dest.frame.accesslink; EXIT}; -- frame

    ENDLOOP;
  IF ~frame.started THEN Start[[frame[frame]]];
  frame.code.out ← FALSE;
  RETURN WITH state
  END;

DivideCheckTrap: PROC =
  BEGIN
  frame: PrincOps.FrameHandle;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  state.v.dest ← frame ← Frame.GetReturnFrame[];
  BumpPC[frame, 1];
  state.v.stkptr ← state.v.stkptr - 2;
  SIGNAL DivideCheck; -- pc is advanced on this trap
  RETURN WITH state.v
  END;

HardwareErrorTrap: PROC =
  BEGIN
  state: RECORD [a, b: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  ERROR HardwareError
  END;

PointerFaultTrap: PROC =
  BEGIN
  frame: PrincOps.FrameHandle;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  state.v.dest ← frame ← Frame.GetReturnFrame[];
  BumpPC[frame, 1];
  SIGNAL PointerFault; -- pc is advanced on this trap
  RETURN WITH state.v
  END;

StackErrorTrap: PROC =
  BEGIN
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  foo: BOOLEAN;
  state.v ← STATE;
  foo ← TRUE; -- The garbage with foo is so all the code for the next statement
  -- looks like it is in this procedure
  -- to the debugger (when it looks in the fine grain table)
  IF foo THEN ERROR StackError
  END;

OpcodeLengths: PACKED ARRAY [0..255] OF [0..3] =
  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1,
    2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 2, 3, 2, 2, 2, 3, 0, 0, 0, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 3, 3,
    2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 2, 3, 2, 3, 1, 1, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 1, 1, 1, 2, 2, 1, 1, 1, 2, 1; 1, 1, 1, 0, 0,
    0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
```

```
    1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
    1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1, 2, 2, 2,
    0, 2, 2, 1, 0];

UnboundProcedureTrap: PROC =
  BEGIN
  sourceOp: Environment.Byte;
  frame: PrincOps.FrameHandle;
  dest: ControlLink;
  state: StateVector;
  state ← STATE;
  dest ← Trap.ReadOTP[];
  state.dest ← frame ← Frame.GetReturnFrame[];
  BumpPC[frame, OpcodeLengths[sourceOp ← GetCodeBytes[frame][frame.pc]]];
  IF sourceOp = Mopcodes.zSFC OR sourceOp = Mopcodes.zPORTO THEN
    state.stkptr ← state.stkptr - 1;
  [] ← SIGNAL UnboundProcedure[dest];
  RETURN WITH state
  END;

WakeupErrorTrap: PROC =
  BEGIN
  state: RECORD [a, b: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  ERROR WakeupError
  END;

ZeroDivisorTrap: PROC =
  BEGIN
  frame: PrincOps.FrameHandle;
  state: RECORD [a: UNSPECIFIED, v: StateVector];
  state.v ← STATE;
  state.v.dest ← frame ← Frame.GetReturnFrame[];
  state.v.stkptr ←
    state.v.stkptr -
      (IF GetCodeBytes[frame][frame.pc] = Mopcodes.zLDIV THEN 2 ELSE 1);
  BumpPC[frame, 1];
  SIGNAL ZeroDivisor; -- pc is advanced on this trap
  RETURN WITH state.v
  END;

-- Control Fault Trap

ControlFaultTrap: PROC =
  BEGIN
  errorStart, savedState: StateVector;
  sourceOp: Environment.Byte;
  NullPort: PortHandle = LOOPHOLE[0];
  PORTI: PROC = MACHINE CODE BEGIN Mopcodes.zPORTI END;
  p, q: PortHandle;
  sourceFrame, self: FrameHandle;
  source: ControlLink;

  savedState ← STATE;
  self ← Frame.MyLocalFrame[];
  sourceFrame ← self.returnlink.frame;
```



```
  source ← Trap.ReadOTP[];
  sourceOp ← GetCodeBytes[sourceFrame][sourceFrame.pc];
  BumpPC[sourceFrame, OpcodeLengths[sourceOp]];
  IF sourceOp = Mopcodes.zPORTO THEN
    BEGIN
    savedState.stkptr ← savedState.stkptr - 1;
    p ← source.port;
    q ← p.dest.port;
    IF q = NullPort THEN errorStart.stk[0] ← LinkageFault
    ELSE
      BEGIN
      q↑ ← Port[links[NullFrame, [indirect[port[p]]]]];
      errorStart.stk[0] ← PortFault
      END;
    errorStart.stk[1] ← 0; -- message
    errorStart.instbyte ← 0;
    errorStart.stkptr ← 2;
    errorStart.source ← sourceFrame.returnlink;
    -- lets UNWIND skip trapping frame
    errorStart.dest ← SDDefs.SD[SDDefs.sError];
    IF savedState.stkptr = 0 THEN RETURN WITH errorStart -- RESPONDING port

    ELSE
      BEGIN
      p.frame ← self;
      TRANSFER WITH errorStart;
      PORTI[];
      p.frame ← sourceFrame;
      savedState.stk[savedState.stkptr + 1] ← savedState.source ← p;
      savedState.stk[savedState.stkptr] ← savedState.dest ← p.dest;
      RETURN WITH savedState
      END
    END
  ELSE -- not a port call
    BEGIN
    Punt: PROC [c: PilotMP.Code] = INLINE
      BEGIN ProcessorFace.SetMP[c]; DO ENDLOOP END;
    IF sourceOp = Mopcodes.zSFC THEN savedState.stkptr ← savedState.stkptr - 1;
    IF SDDefs.SD[SDDefs.sSignal] = 0 THEN Punt[PilotMP.cEarlyTrap];
    -- avoid loop
    savedState.dest ← sourceFrame;
    [] ← SIGNAL ControlFault[sourceFrame];
    RETURN WITH savedState -- to press on

    END
  END;

-- XFER break tracing implementation

continueTracing: BOOLEAN;

StartTrace: PUBLIC PROC [
  loc: POINTER, val: UNSPECIFIED, mask: WORD, equal: BOOLEAN] =
  BEGIN
  state: PrincOps.StateVector;
  lval: UNSPECIFIED;
  continueTracing ← TRUE;
  state ← STATE;
  state.dest ← Frame.GetReturnLink[];
  SDDefs.SD[SDDefs.sXferTrap] ← state.source ← Frame.MyLocalFrame[];
```

```
    ProcessInternal.DisableInterrupts[];
    DO
      lval ← Inline.BITAND[loc↑, mask];
      IF (IF equal THEN val = lval ELSE val # lval) THEN
        RuntimeInternal.WorryCallDebugger["TraceTrap"L];
      IF ~continueTracing THEN {
        SDDefs.SD[SDDefs.sXferTrap] ← 0;
        ProcessInternal.EnableInterrupts[];
        RETURN WITH state}
      ELSE {
        XferTrap.WriteXTS[skip1];
        ProcessInternal.EnableInterrupts[];
        TRANSFER WITH state};
      ProcessInternal.DisableInterrupts[];
      state ← STATE;
      XferTrap.WriteXTS[off];
      state.dest ← Frame.GetReturnLink[];
      state.source ← 0;
      ENDLOOP
    END;

  StopTrace: PUBLIC PROC = {continueTracing ← FALSE};

  Initialize: PROC = -- this code need only be initially resident
    BEGIN OPEN SDDefs;
    pSD: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
    pSD[sStackError] ← StackErrorTrap;
    pSD[sWakeupError] ← WakeupErrorTrap;
    pSD[sAllocTrap] ← AllocTrap[AllocTrap[NullFrame]];
    pSD[sControlFault] ← ControlFaultTrap;
    pSD[sSwapTrap] ← CodeTrap;
    pSD[sUnbound] ← UnboundProcedureTrap;
    pSD[sZeroDivisor] ← ZeroDivisorTrap;
    pSD[sDivideCheck] ← DivideCheckTrap;
    pSD[sHardwareError] ← HardwareErrorTrap;
    pSD[sBoundsFault] ← BoundsFaultTrap;
    pSD[sPointerFault] ← PointerFaultTrap;
    pSD[sStart] ← Start;
    pSD[sRestart] ← Restart;

    END;

  -- Main body

    Initialize[];
  END.
LOG
(For earlier log entries see Amargosa archive version.)
Time: April 10, 1980 5:16 PM  By: Knutsen  Action: Make compatible with changed Resi
**dentMemory procedure names; set mds in MemorySwap
Time: April 29, 1980 9:38 PM  By: Forrest  Action: Move MemorySwap and callers to Tr
**aps. MiscDeviceCleanups to OISProcessorHeadD0. Rearrange and eliminate MONITOR
Time: May 3, 1980 1:28 PM  By: Forrest  Action: Mesa 6.0 (non-Dandelion) conversio
**n
Time: June 23, 1980 5:46 PM  By: McJones  Action: OISProcessorFace = >ProcessorFac
**e; no longer disable interrupts when getting PrincOps trap parameters
Time: September 2, 1980 1:56 PM  By: Johnsson  Action: New XferTrap semantics
Time: September 29, 1980 9:30 AM  By: Johnsson  Action: all princops traps occur w
```



```
**ith state restored to beginning of instruction.
Time: October 3, 1980 1:54 PM By: Forrest  Action: Move Unimplemented Trap to Proce
**ssor Head.
```

```
-- BcdOperations.mesa
-- Last Modified by Sandman, July 17, 1980 11:55 AM
-- Copyright Xerox Corporation 1979, 1980

DIRECTORY
  BcdDefs USING [
    Base, CTIndex, CTNull, CTRecord, EVIndex, EVNull, EVRecord, EXPIndex, EXPNull,
    EXPRecord, FTIndex, FTNull, FTRecord, IMPIndex, IMPNull, IMPRecord, MTIndex,
    MTNull, MTRecord, Namee, NameRecord, NTIndex, NTNull, NTRecord, NullName,
    SGIndex, SGNull, SGRecord, SpaceID, SPIndex, SPNull, SPRecord, VersionStamp],
  BcdOps USING [
    BcdBase, CTHandle, EVHandle, EXPHandle, FTHandle, IMPHandle, MTHandle,
    NTHandle, SGHandle, SPHandle];

BcdOperations: PROGRAM EXPORTS BcdOps = PUBLIC

  BEGIN OPEN BcdOps, BcdDefs;

  ProcessConfigs: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [CTHandle, CTIndex] RETURNS [BOOLEAN]]
    RETURNS [cth: CTHandle, cti: CTIndex] =
    BEGIN
    ctb: Base = LOOPHOLE[bcd + bcd.ctOffset];
    i: CARDINAL;
    cti ← FIRST[CTIndex];
    FOR i IN [0..bcd.nConfigs) DO
      cth ← @ctb[cti];
      IF proc[cth, cti] THEN RETURN;
      cti ← cti + SIZE[CTRecord] + cth.nControls;
      ENDLOOP;
    RETURN[NIL, CTNull];
    END;

  ProcessExternals: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [EVHandle, EVIndex] RETURNS [BOOLEAN]]
    RETURNS [evh: EVHandle, evi: EVIndex] =
    BEGIN
    evb: Base = LOOPHOLE[bcd + bcd.evOffset];
    FOR evi ← FIRST[EVIndex], evi + SIZE[EVRecord] + evh.length UNTIL evi =
      bcd.evLimit DO evh ← @evb[evi]; IF proc[evh, evi] THEN RETURN; ENDLOOP;
    RETURN[NIL, EVNull];
    END;

  ProcessExports: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [EXPHandle, EXPIndex] RETURNS [BOOLEAN]]
    RETURNS [eth: EXPHandle, eti: EXPIndex] =
    BEGIN
    etb: Base = LOOPHOLE[bcd + bcd.expOffset];
    i: CARDINAL;
    eti ← FIRST[EXPIndex];
    FOR i IN [0..bcd.nExports) DO
      eth ← @etb[eti];
      IF proc[eth, eti] THEN RETURN;
      eti ← eti + SIZE[EXPRecord] + eth.size;
      ENDLOOP;
    RETURN[NIL, EXPNull];
    END;
```



```
  ProcessFiles: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [FTHandle, FTIndex] RETURNS [BOOLEAN]]
    RETURNS [fth: FTHandle, fti: FTIndex] =
    BEGIN
    ftb: Base = LOOPHOLE[bcd + bcd.ftOffset];
    FOR fti ← FIRST[FTIndex], fti + SIZE[FTRecord] UNTIL fti = bcd.ftLimit DO
      fth ← @ftb[fti]; IF proc[fth, fti] THEN RETURN; ENDLOOP;
    RETURN[NIL, FTNull];
    END;

  ProcessImports: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [IMPHandle, IMPIndex] RETURNS [BOOLEAN]]
    RETURNS [ith: IMPHandle, iti: IMPIndex] =
    BEGIN
    itb: Base = LOOPHOLE[bcd + bcd.impOffset];
    i: CARDINAL;
    iti ← FIRST[IMPIndex];
    FOR i IN [0..bcd.nImports) DO
      ith ← @itb[iti];
      IF proc[ith, iti] THEN RETURN;
      iti ← iti + SIZE[IMPRecord];
      ENDLOOP;
    RETURN[NIL, IMPNull];
    END;

  ProcessModules: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [MTHandle, MTIndex] RETURNS [BOOLEAN]]
    RETURNS [mth: MTHandle, mti: MTIndex] =
    BEGIN
    mtb: Base = LOOPHOLE[bcd + bcd.mtOffset];
    i: CARDINAL;
    mti ← FIRST[MTIndex];
    FOR i IN [0..bcd.nModules) DO
      mth ← @mtb[mti];
      IF proc[mth, mti] THEN RETURN;
      mti ← mti + SIZE[MTRecord] + mth.frame.length;
      ENDLOOP;
    RETURN[NIL, MTNull];
    END;

  ProcessNames: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [NTHandle, NTIndex] RETURNS [BOOLEAN]]
    RETURNS [nth: NTHandle, nti: NTIndex] =
    BEGIN
    ntb: Base = LOOPHOLE[bcd + bcd.ntOffset];
    FOR nti ← FIRST[NTIndex], nti + SIZE[NTRecord] UNTIL nti = bcd.ntLimit DO
      nth ← @ntb[nti]; IF proc[nth, nti] THEN RETURN; ENDLOOP;
    RETURN[NIL, NTNull];
    END;

  ProcessSegs: PROCEDURE [
    bcd: BcdBase, proc: PROCEDURE [SGHandle, SGIndex] RETURNS [BOOLEAN]]
    RETURNS [sgh: SGHandle, sgi: SGIndex] =
    BEGIN
    sgb: Base = LOOPHOLE[bcd + bcd.sgOffset];
    FOR sgi ← FIRST[SGIndex], sgi + SIZE[SGRecord] UNTIL sgi = bcd.sgLimit DO
      sgh ← @sgb[sgi]; IF proc[sgh, sgi] THEN RETURN; ENDLOOP;
    RETURN[NIL, SGNull];
    END;
```

```
ProcessSpaces: PROCEDURE [
  bcd: BcdBase, proc: PROCEDURE [SPHandle, SPIndex] RETURNS [BOOLEAN]]
  RETURNS [sph: SPHandle, spi: SPIndex] =
  BEGIN
  spb: Base = LOOPHOLE[bcd + bcd.spOffset];
  FOR spi ← FIRST[SPIndex], spi + SIZE[SPRecord] + sph.length*SIZE[SpaceID]
    UNTIL spi = bcd.spLimit DO
    sph ← @spb[spi]; IF proc[sph, spi] THEN RETURN; ENDLOOP;
  RETURN[NIL, SPNull];
  END;

FindName: PROCEDURE [bcd: BcdBase, owner: Namee] RETURNS [name: NameRecord] =
  BEGIN
  n: NTHandle;

  FindOwner: PROCEDURE [nth: NTHandle, nti: NTIndex] RETURNS [BOOLEAN] =
    BEGIN RETURN[owner = nth.item]; END;

  RETURN[
    IF (n ← ProcessNames[bcd, FindOwner].nth) = NIL THEN NullName ELSE n.name]
  END;

ModuleVersion: PROCEDURE [bcd: BcdBase, mti: MTIndex]
  RETURNS [version: VersionStamp] =
  BEGIN
  mtb: Base = LOOPHOLE[bcd + bcd.mtOffset];
  ftb: Base = LOOPHOLE[bcd + bcd.ftOffset];
  RETURN[ftb[mtb[mti].file].version];
  END;


END....
```

```
-- PilotLoaderCore.mesa Edited by Sandman on October 21, 1980 10:57 AM
-- Edited by Forrest on October 25, 1980 7:45 PM

DIRECTORY
  BcdDefs USING [
    Base, CTIndex, CTNull, EPIndex, EPLimit, EVIndex, EVNull, EXPIndex, EXPNull,
    FPIndex, FPNull, FPRecord, FTIndex, FTSelf, IMPIndex, Link, MTIndex, MTNull,
    NameRecord, PackedString, TMIndex, TMNull, TMRecord, UnboundLink, VarLimit,
    VersionStamp],
  BcdOps USING [
    BcdBase, CTHandle, EXPHandle, FPHandle, FTHandle, IMPHandle, MTHandle,
    NameString, ProcessConfigs, ProcessExports, ProcessImports, ProcessModules,
    TMHandle],
  Frame USING [Alloc],
  Inline USING [BITAND],
  PilotLoaderOps USING [
    AllocateFrames, AppendName, Binding, BindLink, CloseLinkSpace, DestroyMap,
    FindCode, FrameList, FreeSpace, GetSpace, InitBinding, InitializeMap,
    OpenLinkSpace, ReadLink, ReleaseBinding, ReleaseFrames, WriteLink],
  PilotLoadStateFormat USING [ModuleInfo],
  PilotLoadStateOps USING [
    AcquireBcd, BcdExports, BcdExportsTypes, BcdUnresolved, ConfigIndex,
    EnterModule, GetMap, GetModule, InputLoadState, Map, MapConfigToReal,
    ReleaseBcd, ReleaseLoadState, ReleaseMap, UpdateLoadState],
  PrincOps USING [
    ControlLink, ControlModule, GFTIndex, GFTNull, GlobalFrameHandle, NullControl,
    NullGlobalFrame, NullLink, UnboundLink],
  PrincOpsRuntime USING [GetFrame, GFT],
  RuntimeInternal USING [EnterGlobalFrame, MakeFsi];

PilotLoaderCore: PROGRAM
  IMPORTS
    BcdOps, Frame, Inline, PilotLoaderOps, PilotLoadStateOps, PrincOpsRuntime,
    RuntimeInternal
  EXPORTS PilotLoaderOps =

  BEGIN OPEN BcdDefs, BcdOps;

  Binding: TYPE = PilotLoaderOps.Binding;
  ConfigIndex: TYPE = PilotLoadStateOps.ConfigIndex;
  Map: TYPE = PilotLoadStateOps.Map;
  GlobalFrameHandle: TYPE = PrincOps.GlobalFrameHandle;
  ControlModule: TYPE = PrincOps.ControlModule;

  VersionMismatch: PUBLIC SIGNAL [name: STRING] = CODE;

  New: PUBLIC PROCEDURE [bcd: BcdBase, framelinks: BOOLEAN]
    RETURNS [cm: PrincOps.ControlModule] =
    BEGIN OPEN PilotLoadStateOps, PilotLoaderOps;
    system: BcdBase ← NIL;
    map: Map ← DESCRIPTOR[NIL, 0];
    sMap: Map ← DESCRIPTOR[NIL, 0];
    binding: Binding ← DESCRIPTOR[NIL, 0];
    nbcds, i: CARDINAL;
    resolved, canBind: BOOLEAN;
    fl: PilotLoaderOps.FrameList ← NIL;
    CleanUpNew: PROCEDURE =
      BEGIN
```



```
      frames: FrameList;
      DestroyMap[map];
      IF BASE[binding] # NIL THEN [] ← ReleaseBinding[bcd, binding];
      UNTIL fl = NIL DO frames ← fl; fl ← fl.next; FreeSpace[frames]; ENDLOOP;
      ReleaseBcd[bcd];
      ReleaseLoadState[];
      RETURN
      END;
    BEGIN
    ENABLE UNWIND => {IF fl # NIL THEN ReleaseFrames[bcd, fl, map]; CleanUpNew[]};
    resolved ← bcd.nImports = 0;
    map ← InitializeMap[bcd];
    nbcds ← InputLoadState[];
    [cm: cm, fl: fl] ← LoadModules[bcd, map, nbcds, framelinks];
    [] ← BindImports[bcd, bcd, binding ← InitBinding[bcd]];
    resolved ← ProcessLinks[bcd, bcd, map, binding, nbcds, TRUE];
    binding ← ReleaseBinding[bcd, binding];
    FOR i DECREASING IN [0..nbcds) DO
      IF ~resolved AND BcdExports[i] THEN
        BEGIN
        ENABLE UNWIND => ReleaseBcd[system];
        system ← AcquireBcd[i];
        binding ← InitBinding[bcd];
        canBind ← BindImports[bcd, system, binding];
        resolved ←
          IF canBind THEN ProcessLinks[bcd, system, map, binding, i, FALSE]
          ELSE FALSE;
        binding ← ReleaseBinding[bcd, binding];
        END;
      IF BcdUnresolved[i] AND (bcd.nExports # 0 OR bcd.nModules = 1) THEN
        BEGIN
        ENABLE
          UNWIND => {
            IF BASE[binding] # NIL THEN binding ← ReleaseBinding[system, binding];
            ReleaseMap[sMap];
            ReleaseBcd[system]};
        IF system = NIL THEN system ← AcquireBcd[i];
        sMap ← GetMap[i];
        binding ← InitBinding[system];
        canBind ← BindImports[system, bcd, binding];
        IF canBind THEN
          [] ← ProcessLinks[system, bcd, sMap, binding, nbcds, FALSE];
        ReleaseMap[sMap];
        sMap ← DESCRIPTOR[NIL, 0];
        binding ← ReleaseBinding[system, binding];
        END;
      IF bcd.typeExported AND BcdExportsTypes[i] THEN
        BEGIN
        ENABLE UNWIND => ReleaseBcd[system];
        IF system = NIL THEN system ← AcquireBcd[i];
        CheckTypes[bcd, system];
        END;
      IF system # NIL THEN ReleaseBcd[system];
      system ← NIL;
      ENDLOOP;
    UpdateLoadState[nbcds, bcd];
    CleanUpNew[];
    END;
  END;
```

```
LoadModules: PROCEDURE [
  bcd: BcdBase, map: Map, config: ConfigIndex, allframelinks: BOOLEAN]
  RETURNS [cm: PrincOps.ControlModule, fl: PilotLoaderOps.FrameList] =
  BEGIN
  f: PilotLoaderOps.FrameList;
  frames: POINTER;
  space: CARDINAL;
  single: BOOLEAN ← bcd.nModules = 1;
  resident: BOOLEAN;
  GetFrameSizes: PROCEDURE [mth: MTHandle] = {
    IF allframelinks OR mth.links = frame OR ~mth.code.linkspace THEN
      space ← space + mth.frame.length;
    space ← NextMultipleOfFour[space] + mth.framesize;
    resident ← resident OR mth.residentFrame};
  FrameInit: PROCEDURE [mth: MTHandle] = {
    frame: GlobalFrameHandle;
    framelinks: BOOLEAN;
    gfi: PrincOps.GFTIndex;
    framelinks ← allframelinks OR mth.links = frame OR ~mth.code.linkspace;
    IF framelinks THEN frames ← frames + mth.frame.length;
    frame ← NextMultipleOfFour[frames];
    frames ← frame + mth.framesize;
    gfi ← RuntimeInternal.EnterGlobalFrame[frame, mth.ngfi];
    FOR i: PrincOps.GFTIndex IN [0..mth.ngfi) DO
      map[i + mth.gfi] ← gfi + i;
      PilotLoadStateOps.EnterModule[
        gfi + i,
        [gfi: mth.gfi + i, config: config, resolved: mth.frame.length = 0]];
      ENDLOOP;
    frame↑ ←
      [gfi: gfi, copied: FALSE, alloced: single, shared: FALSE, started: FALSE,
        trapxfers: FALSE, codelinks: ~framelinks, global:, code:]};
  DoFramePack: PROC [fph: FPHandle, fpi: FPIndex] RETURNS [BOOLEAN] = {
    mtb: Base = LOOPHOLE[bcd + bcd.mtOffset];
    space ← 0;
    resident ← FALSE;
    FOR i: CARDINAL IN [0..fph.length) DO
      GetFrameSizes[@mtb[fph.modules[i]]]; ENDLOOP;
    f ← PilotLoaderOps.AllocateFrames[
      size: NextMultipleOfFour[space], single: single, resident: resident];
    f.next ← fl;
    fl ← f;
    frames ← f.frame;
    FOR i: CARDINAL IN [0..fph.length) DO
      FrameInit[@mtb[fph.modules[i]]]; ENDLOOP;
    RETURN[FALSE]};
  OtherFrameSizes: PROC [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] = {
    resident ← FALSE;
    IF map[mth.gfi] = PrincOps.GFTNull THEN GetFrameSizes[mth];
    RETURN[FALSE]};
  OtherFrameInit: PROC [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] = {
    IF map[mth.gfi] = PrincOps.GFTNull THEN FrameInit[mth]; RETURN[FALSE]};
  fl ← NIL;
  BEGIN
  ENABLE UNWIND => PilotLoaderOps.ReleaseFrames[bcd, fl, map];
  [] ← ProcessFramePacks[bcd, DoFramePack];
  space ← 0;
  [] ← BcdOps.ProcessModules[bcd, OtherFrameSizes];
```



```
  IF space # 0 THEN {
    f ← PilotLoaderOps.AllocateFrames[
      size: NextMultipleOfFour[space], single: single, resident: resident];
    f.next ← fl;
    fl ← f;
    frames ← f.frame;
    [] ← BcdOps.ProcessModules[bcd, OtherFrameInit]};
  PilotLoaderOps.FindCode[bcd, map];
  cm ← AssignControlModules[bcd, map];
  END;
  RETURN[cm, fl];
  END;

NextMultipleOfFour: PROCEDURE [n: UNSPECIFIED] RETURNS [UNSPECIFIED] =
  BEGIN RETURN[n + Inline.BITAND[-LOOPHOLE[n, INTEGER], 3B]]; END;

ProcessFramePacks: PROCEDURE [
  bcd: BcdBase, proc: PROC [FPHandle, FPIndex] RETURNS [BOOLEAN]]
  RETURNS [fph: FPHandle, fpi: FPIndex] =
  BEGIN
  fpb: Base = LOOPHOLE[bcd + bcd.fpOffset];
  FOR fpi ← FIRST[FPIndex], fpi + SIZE[FPRecord] + fph.length*SIZE[MTIndex]
    UNTIL fpi = bcd.fpLimit DO
    IF proc[fph ← @fpb[fpi], fpi] THEN RETURN; ENDLOOP;
  RETURN[NIL, FPNull];
  END;

BindImports: PROCEDURE [bcd, system: BcdBase, binding: Binding]
  RETURNS [canBind: BOOLEAN] =
  BEGIN
  ForEachImport: PROCEDURE [ith: IMPHandle, iti: IMPIndex] RETURNS [BOOLEAN] =
    BEGIN
    i: CARDINAL;
    issb, sysssb: BcdOps.NameString;
    module: MTIndex;
    export: EXPIndex;
    ExpMatch: PROCEDURE [eth: EXPHandle, eti: EXPIndex] RETURNS [BOOLEAN] =
      BEGIN
      RETURN[
        eth.port = ith.port AND EqualNames[issb, sysssb, ith.name, eth.name] AND
          EqualVersions[bcd, system, ith.file, eth.file]]
      END;
    ModuleMatch: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN
      RETURN[
        EqualNames[issb, sysssb, ith.name, mth.name] AND EqualVersions[
          bcd, system, ith.file, mth.file]]
      END;
    issb ← LOOPHOLE[bcd + bcd.ssOffset];
    sysssb ← LOOPHOLE[system + system.ssOffset];
    IF ith.port = interface THEN
      BEGIN
      export ← BcdOps.ProcessExports[system, ExpMatch].eti;
      FOR i IN [0..ith.ngfi) DO
        IF export = EXPNull THEN
          binding[ith.gfi + i] ← [whichgfi: i, body: notbound[]]
        ELSE {
          canBind ← TRUE;
          binding[ith.gfi + i] ← [whichgfi: i, body: interface[export]]};
```

```
      ENDLOOP
    END
  ELSE
    BEGIN
    module ← BcdOps.ProcessModules[system, ModuleMatch].mti;
    FOR i IN [0..ith.ngfi) DO
      IF module = MTNull THEN
        binding[ith.gfi + i] ← [whichgfi: i, body: notbound[]]
      ELSE {
        canBind ← TRUE;
        binding[ith.gfi + i] ← [whichgfi: i, body: module[module]]};
      ENDLOOP;
    END;
  RETURN[FALSE];
  END;
canBind ← FALSE;
[] ← BcdOps.ProcessImports[bcd, ForEachImport];
END;

EqualNames: PROCEDURE [
  ps1, ps2: BcdOps.NameString, name1, name2: BcdDefs.NameRecord]
  RETURNS [BOOLEAN] =
  BEGIN
  i: CARDINAL;
  IF ps1.size[name1] # ps2.size[name2] THEN RETURN[FALSE];
  FOR i IN [0..ps1.size[name1]) DO
    IF ps1.string.text[name1 + i] # ps2.string.text[name2 + i] THEN
      RETURN[FALSE];
    ENDLOOP;
  RETURN[TRUE];
  END;

EqualVersions: PROCEDURE [bcd1, bcd2: BcdBase, fti1, fti2: BcdDefs.FTIndex]
  RETURNS [BOOLEAN] =
  BEGIN
  v1, v2: LONG POINTER TO BcdDefs.VersionStamp;
  f1: FTHandle ← @LOOPHOLE[bcd1 + bcd1.ftOffset, Base][fti1];
  f2: FTHandle ← @LOOPHOLE[bcd2 + bcd2.ftOffset, Base][fti2];
  v1 ← IF fti1 = FTSelf THEN @bcd1.version ELSE @f1.version;
  v2 ← IF fti2 = FTSelf THEN @bcd2.version ELSE @f2.version;
  IF v1↑ = v2↑ THEN RETURN[TRUE];
  v1 ← IF fti1 = FTSelf THEN @bcd1.version ELSE @f1.version;
  BadVersion[
    ssb: LOOPHOLE[bcd1 + bcd1.ssOffset],
    name: IF fti1 = FTSelf THEN bcd1.source ELSE f1.name];
  RETURN[FALSE];
  END;

BadVersion: PROCEDURE [ssb: BcdOps.NameString, name: BcdDefs.NameRecord] =
  BEGIN
  filename: STRING ← [40];
  PilotLoaderOps.AppendName[s: filename, ssb: ssb, name: name];
  SIGNAL VersionMismatch[filename];
  END;

ProcessLinks: PROCEDURE [
  bcd, system: BcdBase, map: Map, binding: Binding, config: ConfigIndex,
  initial: BOOLEAN] RETURNS [BOOLEAN] =
  BEGIN OPEN PrincOps;
```



```
  smtb: Base = LOOPHOLE[system + system.mtOffset];
  setb: Base = LOOPHOLE[system + system.expOffset];
  unresolved: BOOLEAN ← FALSE;
  NewLink: PROCEDURE [old: ControlLink, link: Link]
    RETURNS [new: ControlLink, resolved: BOOLEAN] =
    BEGIN
    gfi: GFTIndex ← 0;
    FindLink: PROCEDURE [link: Link]
      RETURNS [new: ControlLink, resolved: BOOLEAN] =
      BEGIN
      ep: EPIndex;
      inside: BOOLEAN;
      rgfi: GFTIndex ← GFTNull;
      new ← PrincOps.UnboundLink;
      IF (inside ← link.gfi < bcd.firstdummy) THEN
        BEGIN new ← ConvertLink[link]; rgfi ← map[link.gfi] END
      ELSE {
        bindLink: PilotLoaderOps.BindLink = binding[link.gfi];
        WITH b: bindLink SELECT FROM
          interface =>
            BEGIN
            e: EXPHandle = @setb[b.eti];
            SELECT e.port FROM
              interface =>
                BEGIN
                ep ← link.ep + (b.whichgfi*EPLimit);
                link ← e.links[ep];
                rgfi ← PilotLoadStateOps.MapConfigToReal[link.gfi, config];
                END;
              ENDCASE;
            END;
          module =>
            BEGIN
            m: MTHandle = @smtb[b.mti];
            link ← [variable[vgfi: m.gfi, var: 0, vtag: var]];
            rgfi ← PilotLoadStateOps.MapConfigToReal[m.gfi, config];
            END;
          ENDCASE};
        SELECT link.vtag FROM
          var => new ← FindVariableLink[inside, link, rgfi];
          proc0, proc1 => BEGIN new ← ConvertLink[link]; new.gfi ← rgfi END;
          ENDCASE;
        RETURN[new: new, resolved: rgfi # GFTNull]
      END;
    FindVariableLink: PROCEDURE [inside: BOOLEAN, el: Link, rgfi: GFTIndex]
      RETURNS [link: ControlLink] =
      BEGIN
      ep: CARDINAL;
      evi: EVIndex;
      evb: Base;
      gfi: GFTIndex ← el.vgfi;
      mth: MTHandle;
      frame: GlobalFrameHandle;
      FindModule: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
        BEGIN
        mgfi: GFTIndex ← mth.gfi;
        IF gfi IN [mth.gfi..mgfi + mth.ngfi) THEN
          BEGIN ep ← VarLimit*(gfi - mgfi); RETURN[TRUE] END;
        RETURN[FALSE]
```

```
    END;
  mth ← BcdOps.ProcessModules[
    IF inside THEN bcd ELSE system, FindModule].mth;
  IF mth = NIL THEN RETURN[PrincOps.NullLink];
  evb ←
    IF ~inside THEN LOOPHOLE[system + system.evOffset, Base]
    ELSE LOOPHOLE[bcd + bcd.evOffset, Base];
  frame ← PrincOpsRuntime.GetFrame[PrincOpsRuntime.GFT[rgfi]];
  IF (ep ← ep + el.var) = 0 THEN RETURN[LOOPHOLE[frame]];
  IF (evi ← mth.variables) = EVNull THEN RETURN[PrincOps.NullLink];
  RETURN[LOOPHOLE[frame + evb[evi].offsets[ep]]];
  END;

new ← old;
resolved ← FALSE; -- was true; this is last minute change in Pilot 5.0
SELECT link.vtag FROM
  proc0, proc1 =>
    IF old = PrincOps.UnboundLink THEN
      [new: new, resolved: resolved] ← FindLink[link];
  var =>
    IF old = PrincOps.NullLink THEN
      [new: new, resolved: resolved] ← FindLink[link];
  ENDCASE => new ← LOOPHOLE[link.typeID];
RETURN
END;

ModuleSearch: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
  BEGIN OPEN PrincOps, PrincOpsRuntime;
  i: CARDINAL;
  gfi: GFTIndex = map[mth.gfi];
  frame: GlobalFrameHandle ← GetFrame[GFT[gfi]];
  resolved, bound: BOOLEAN;
  old, new: ControlLink;
  info: PilotLoadStateFormat.ModuleInfo ← PilotLoadStateOps.GetModule[gfi];
  IF frame = PrincOps.NullGlobalFrame OR info.resolved THEN RETURN[FALSE];
  PilotLoaderOps.OpenLinkSpace[frame, mth];
  IF initial THEN
    FOR i IN [0..mth.frame.length) DO
      PilotLoaderOps.WriteLink[
        offset: i,
        link:
        SELECT mth.frame.frag[i].vtag FROM
          var, type => NullLink,
          ENDCASE => UnboundLink];
      ENDLOOP;
  resolved ← TRUE;
  FOR i IN [0..mth.frame.length) DO
    old ← PilotLoaderOps.ReadLink[i];
    [new: new, resolved: bound] ← NewLink[link: mth.frame.frag[i], old: old];
    IF bound THEN PilotLoaderOps.WriteLink[offset: i, link: new]
    ELSE resolved ← FALSE;
    ENDLOOP;
  FOR i IN [gfi..gfi + mth.ngfi) DO
    info ← PilotLoadStateOps.GetModule[i];
    info.resolved ← resolved;
    PilotLoadStateOps.EnterModule[i, info];
    ENDLOOP;
  PilotLoaderOps.CloseLinkSpace[frame];
```



```
    RETURN[FALSE];
    END;

  [] ← BcdOps.ProcessModules[bcd, ModuleSearch];
  RETURN[unresolved];
  END;

ConvertLink: PROCEDURE [bl: Link] RETURNS [cl: PrincOps.ControlLink] =
  BEGIN
  IF bl = UnboundLink THEN RETURN[PrincOps.UnboundLink];
  SELECT bl.vtag FROM
    var => cl ← [procedure[gfi: bl.vgfi, ep: bl.var, tag: FALSE]];
    proc0, proc1 => cl ← [procedure[gfi: bl.gfi, ep: bl.ep, tag: TRUE]];
    type => cl ← LOOPHOLE[bl.typeID];
    ENDCASE;
  RETURN
  END;

ProcessTypeMap: PROCEDURE [
  bcd: BcdBase, proc: PROC [TMHandle, TMIndex] RETURNS [BOOLEAN]]
  RETURNS [tmh: TMHandle, tmi: TMIndex] =
  BEGIN
  tmb: Base = LOOPHOLE[bcd + bcd.tmOffset];
  FOR tmi ← FIRST[TMIndex], tmi + SIZE[TMRecord] UNTIL tmi = bcd.tmLimit DO
    IF proc[tmh ← @tmb[tmi], tmi] THEN RETURN; ENDLOOP;
  RETURN[NIL, TMNull];
  END;

CheckTypes: PROCEDURE [bcd1, bcd2: BcdBase] =
  BEGIN
  typeError: STRING = "Exported Type Clash"L;
  typb1: Base = LOOPHOLE[bcd1 + bcd1.typOffset];
  typb2: Base = LOOPHOLE[bcd2 + bcd2.typOffset];
  TypeMap1: PROCEDURE [tmh1: TMHandle, tmi1: TMIndex] RETURNS [BOOLEAN] =
    BEGIN
    TypeMap2: PROCEDURE [tmh2: TMHandle, tmi2: TMIndex] RETURNS [BOOLEAN] =
      BEGIN
      IF tmh2.offset = tmh1.offset AND tmh2.version = tmh1.version THEN
        BEGIN
        IF typb1[tmh1.map] # typb2[tmh2.map] THEN
          ERROR VersionMismatch[typeError];
        RETURN[TRUE];
        END
      ELSE RETURN[FALSE];
      END;
    [] ← ProcessTypeMap[bcd2, TypeMap2];
    RETURN[FALSE];
    END;
  [] ← ProcessTypeMap[bcd1, TypeMap1];
  RETURN
  END;

CMMapItem: TYPE = RECORD [
  cti: CTIndex, cm: PrincOps.ControlModule, level: CARDINAL];

AssignControlModules: PROCEDURE [bcd: BcdBase, map: Map]
  RETURNS [cm: PrincOps.ControlModule] =
  BEGIN OPEN PrincOps;
  ctb: Base ← LOOPHOLE[bcd + bcd.ctOffset];
  mtb: Base ← LOOPHOLE[bcd + bcd.mtOffset];
```

```
    cti: CTIndex;
    mapIndex, maxLevel: CARDINAL ← 0;
    i: CARDINAL;
    cmMap: POINTER TO ARRAY [0..0) OF CMMapItem;
    MapControls: PROCEDURE [cth: CTHandle, cti: CTIndex] RETURNS [BOOLEAN] =
      BEGIN OPEN PrincOps, PrincOpsRuntime;
      cm: ControlModule;
      c: CTIndex;
      level: CARDINAL;
      IF cth.nControls = 0 THEN cm ← NullControl
      ELSE {
        i: CARDINAL;
        cm.list ← Frame.Alloc[
          RuntimeInternal.MakeFsi[
          cth.nControls + SIZE[CARDINAL] + SIZE[ControlModule]]];
        cm.list.nModules ← cth.nControls + 1;
        FOR i IN [0..cth.nControls) DO
          cm.list.frames[i + 1] ← GetFrame[GFT[map[mtb[cth.controls[i]].gfi]]];
          ENDLOOP;
        cm.multiple ← TRUE};
      FOR c ← ctb[cti].config, ctb[c].config UNTIL c = CTNull DO
        level ← level + 1; ENDLOOP;
      cmMap[mapIndex] ← [cti: cti, cm: cm, level: level];
      mapIndex ← mapIndex + 1;
      maxLevel ← MAX[maxLevel, level];
      RETURN[FALSE];
      END;
    GetControl: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN OPEN PrincOps, PrincOpsRuntime;
      frame: GlobalFrameHandle ← GetFrame[GFT[map[mth.gfi]]];
      IF mth.config # cti THEN RETURN[FALSE];
      IF frame.global[0] = NullControl THEN frame.global[0] ← GetModule[cm];
      RETURN[FALSE];
      END;
    IF bcd.nModules = 1 THEN
      BEGIN OPEN PrincOpsRuntime;
      frame: GlobalFrameHandle ← GetFrame[GFT[map[1]]];
      frame.global[0] ← NullControl;
      RETURN[[frame[frame]]];
      END;
    cmMap ← PilotLoaderOps.GetSpace[bcd.nConfigs*SIZE[CMMapItem]];
    [] ← BcdOps.ProcessConfigs[bcd, MapControls];
    FOR level: CARDINAL DECREASING IN [0..maxLevel] DO
      FOR index: CARDINAL IN [0..mapIndex) DO
        list: ControlModule;
        IF cmMap[index].level # level OR (cm ← cmMap[index].cm) = NullControl THEN
          LOOP;
        list ← cm;
        list.multiple ← FALSE;
        list.list.frames[1] ← SetLink[cm, list.list.frames[1]].frame;
        FOR i: CARDINAL IN [2..list.list.nModules) DO
          list.list.frames[i] ← SetLink[
            GetModule[[frame[list.list.frames[1]]]], list.list.frames[i]].frame;
          ENDLOOP;
        cti ← cmMap[index].cti;
        [] ← BcdOps.ProcessModules[bcd, GetControl];
        ENDLOOP;
      ENDLOOP;
```



```
    FOR index: CARDINAL IN [0..mapIndex) DO
      parent: CARDINAL;
      list: ControlModule;
      IF (list ← cmMap[index].cm) = NullControl THEN LOOP;
      list.multiple ← FALSE;
      IF (cti ← ctb[cmMap[index].cti].config) = CTNull THEN cm ← NullControl
      ELSE {
        FOR parent IN [0..mapIndex) DO
          IF cmMap[parent].cti = cti THEN EXIT; ENDLOOP;
        cm ← GetModule[cmMap[parent].cm]};
      list.list.frames[0] ← cm.frame;
      ENDLOOP;
    FOR i IN [0..mapIndex) DO
      IF ctb[cmMap[i].cti].config = CTNull THEN {
        cm ← GetModule[cmMap[i].cm]; EXIT};
      ENDLOOP;
    PilotLoaderOps.FreeSpace[cmMap];
    END;

  SetLink: PROCEDURE [cm: ControlModule, frame: GlobalFrameHandle]
    RETURNS [ControlModule] = {
    t: ControlModule = frame.global[0];
    frame.global[0] ← cm;
    RETURN[IF t = PrincOps.NullControl THEN [frame[frame]] ELSE t]};

  GetModule: PROCEDURE [cm: ControlModule] RETURNS [ControlModule] = {
    list: ControlModule;
    DO
      IF ~cm.multiple THEN RETURN[cm];
      list ← cm;
      list.multiple ← FALSE;
      cm.frame ← list.list.frames[1];
      ENDLOOP};

  END...
```

```
-- PilotLoaderSupport.mesa Modified by
-- Forrest/Johnsson, October 6, 1980 2:27 PM
-- Sandman, October 20, 1980 1:15 PM
-- Johnsson, October 27, 1980 12:02 PM

DIRECTORY
  BcdDefs USING [
    Base, FTIndex, FTSelf, MTIndex, NameRecord, SGIndex, SpaceID, SPIndex,
    SPRecord, VersionID],
  BcdOps USING [
    BcdBase, FTHandle, MTHandle, NameString, ProcessModules, ProcessSegs,
    SGHandle, SPHandle],
  File USING [Capability, LimitPermissions, PageCount, PageNumber, read, write],
  Inline USING [BITAND],
  Heap USING [FreeMDSNode, MakeMDSNode, systemMDSZone],
  Mopcodes USING [zALLOC, zFREE],
  PilotLoaderOps USING [
    Binding, BindLink, FrameItem, FrameList, New, VersionMismatch],
  PilotLoadStateOps USING [Map],
  PrincOps USING [
    ControlLink, ControlModule, GFTIndex, GlobalFrameHandle, LastAVSlot,
    NullControl],
  PrincOpsRuntime USING [EmptyGFTItem, GetFrame, GFT],
  Runtime USING [ConfigErrorType],
  RuntimeInternal USING [Codebase, FrameSize],
  RuntimePrograms USING [Start],
  Space USING [
    Create, defaultBase, defaultWindow, Delete, GetAttributes, GetHandle,
    GetWindow, Handle, LongPointer, MakeReadOnly, MakeWritable, Map, mds,
    nullHandle, PageFromLongPointer, Pointer, virtualMemory, WindowOrigin,
    wordsPerPage],
  SpecialSpace USING [CreateForCode, MakeResident],
  SubSys USING [Handle];

PilotLoaderSupport: MONITOR
  IMPORTS
    BcdOps, File, Inline, Heap, PilotLoaderOps, PrincOpsRuntime, RuntimeInternal,
    RuntimePrograms, Space, SpecialSpace
  EXPORTS PilotLoaderOps, Runtime, SubSys
  SHARES File = PUBLIC

  BEGIN OPEN PilotLoaderOps, BcdDefs, BcdOps, PrincOps;

  BcdBase: PRIVATE TYPE = BcdOps.BcdBase;
  Map: PRIVATE TYPE = PilotLoadStateOps.Map;

  bcdCap: File.Capability;
  bcdSpaceBase: File.PageNumber;

  InvalidFile: PRIVATE SIGNAL [File.Capability] = CODE;

  LoadBcd: PRIVATE PROCEDURE [file: File.Capability, offset: File.PageCount]
    RETURNS [bcd: BcdBase] =
    BEGIN
    pages: CARDINAL;
    bcdSpace: Space.Handle;
    bcdSpaceBase ← offset;
    bcdSpace ← Space.Create[size: 1, parent: Space.virtualMemory];
```



```
    Space.Map[space: bcdSpace, window: [file: file, base: bcdSpaceBase]];
    bcd ← Space.LongPointer[bcdSpace];
    pages ← bcd.nPages;
    IF bcd.versionIdent # BcdDefs.VersionID OR bcd.definitions THEN
      ERROR InvalidFile[file ! UNWIND => Space.Delete[bcdSpace]];
    IF pages > 1 THEN
      BEGIN
      Space.Delete[bcdSpace];
      bcdSpace ← Space.Create[size: pages, parent: Space.virtualMemory];
      Space.Map[space: bcdSpace, window: [file: file, base: bcdSpaceBase]];
      bcd ← Space.LongPointer[bcdSpace];
      END;
    bcdCap ← file;
    RETURN
    END;

  ConfigError: PUBLIC ERROR [type: Runtime.ConfigErrorType] = CODE;

  LoadConfig: PUBLIC PROCEDURE [
    file: File.Capability, offset: File.PageCount, codeLinks: BOOLEAN]
    RETURNS [PROGRAM] = {RETURN[LOOPHOLE[Load[file, offset, codeLinks]]]};

  NewConfig: PUBLIC PROCEDURE [
    file: File.Capability, offset: File.PageCount, codeLinks: BOOLEAN] = {
    [] ← Load[file, offset, codeLinks]};

  RunConfig: PUBLIC PROCEDURE [
    file: File.Capability, offset: File.PageCount, codeLinks: BOOLEAN] = {
    Run[Load[file, offset, codeLinks]]};

  Load: PUBLIC ENTRY PROC [
    file: File.Capability, offset: File.PageCount, codeLinks: BOOLEAN]
    RETURNS [SubSys.Handle] =
    BEGIN
    ENABLE BEGIN UNWIND => NULL END;
    cm: ControlModule = New[
      LoadBcd[file, offset ! InvalidFile => GOTO invalidConfig], ~codeLinks !
      BadCode => GOTO badCode;
      PilotLoaderOps.VersionMismatch => GOTO versionMismatch;
      MissingCode => GOTO missingCode];
    RETURN[[cm]]
    EXITS
      invalidConfig => RETURN WITH ERROR ConfigError[invalidConfig];
      badCode => RETURN WITH ERROR ConfigError[badCode];
      versionMismatch => RETURN WITH ERROR ConfigError[versionMismatch];
      missingCode => RETURN WITH ERROR ConfigError[missingCode];
    END;

  Run: PUBLIC PROCEDURE [handle: SubSys.Handle] = {
    cm: ControlModule = LOOPHOLE[handle];
    IF cm # NullControl THEN RuntimePrograms.Start[cm]};

  AppendName: PUBLIC PROCEDURE [s: STRING, ssb: NameString, name: NameRecord] =
    BEGIN
    i: CARDINAL;
    FOR i IN [s.length..MIN[s.maxlength, ssb.size[name] + s.length]) DO
      s[i] ← ssb.string.text[name + i]; s.length ← s.length + 1; ENDLOOP;
    END;
```

```
-- Frame allocation/deallocation

AllocateFrames: PUBLIC PROC [size: CARDINAL, single, resident: BOOLEAN]
  RETURNS [f: FrameList] =
  BEGIN
  space: Space.Handle;
  p: POINTER;
  IF single THEN {
    index: CARDINAL;
    FOR index IN [0..PrincOps.LastAVSlot] DO
      IF RuntimeInternal.FrameSize[index] >= size THEN EXIT; ENDLOOP;
    p ← Alloc[index]}
  ELSE {
    space ← Space.Create[
      size: (size + Space.wordsPerPage - 1)/Space.wordsPerPage,
      parent: Space.mds];
    Space.Map[space];
    IF TRUE THEN SpecialSpace.MakeResident[space];
    p ← Space.Pointer[space]};
  f ← GetSpace[SIZE[PilotLoaderOps.FrameItem]];
  f.frame ← p;
  RETURN
  END;

Alloc: PROCEDURE [CARDINAL] RETURNS [POINTER] = MACHINE CODE
  BEGIN Mopcodes.zALLOC END;

ReleaseFrames: PUBLIC PROCEDURE [
  bcd: BcdBase, frames: FrameList, map: PilotLoadStateOps.Map] =
  BEGIN
  i: CARDINAL;
  mtb: Base = LOOPHOLE[bcd + bcd.mtOffset];
  IF frames = NIL THEN RETURN;
  IF bcd.nModules = 1 THEN
    BEGIN
    Free: PROCEDURE [POINTER] = MACHINE CODE BEGIN Mopcodes.zFREE END;
    Free[frames.frame];
    END
  ELSE
    UNTIL frames = NIL DO
      Space.Delete[Space.GetHandle[Space.PageFromLongPointer[frames.frame]]];
      frames ← frames.next;
      ENDLOOP;
  FOR i IN [0..LENGTH[map]) DO
    PrincOpsRuntime.GFT[map[i]] ← PrincOpsRuntime.EmptyGFTItem; ENDLOOP;
  END;

-- Code management

FindCode: PUBLIC PROCEDURE [bcd: BcdBase, map: Map] =
  BEGIN
  sgList: SGList;
  GetCode: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
    BEGIN OPEN RuntimeInternal, PrincOpsRuntime;
    MatchCSeg: PROC [mmth: MTHandle, mmti: MTIndex] RETURNS [BOOLEAN] = {
      IF mth # mmth AND mth.code.sgi = mmth.code.sgi AND mth.code.offset =
        mmth.code.offset THEN
        frame.shared ← GetFrame[GFT[map[mmth.gfi]]].shared ← TRUE;
      RETURN[FALSE]};
```



```
    frame: GlobalFrameHandle = GetFrame[GFT[map[mth.gfi]]];
    IF mth.altoCode THEN InvalidModule[bcd, mth];
    frame.code.longbase ← SgiToLP[bcd, mth.code.sgi, sgList] + mth.code.offset;
    frame.code.out ← TRUE;
    [] ← BcdOps.ProcessModules[bcd, MatchCSeg];
    RETURN[FALSE];
    END;
  sgList ← FindSegments[bcd];
  AllocateCodeSpaces[bcd, sgList];
  [] ← BcdOps.ProcessModules[bcd, GetCode ! UNWIND => ReleaseCode[sgList]];
  FreeSpace[BASE[sgList]];
  RETURN
  END;

ReleaseCode: PROCEDURE [sgList: SGList] =
  BEGIN
  i: CARDINAL;
  space1, space2: Space.Handle;
  space1 ← FindMappedSpace[sgList[i ← 0].space];
  i ← 1;
  DO
    DO
      IF i = LENGTH[sgList] THEN EXIT;
      IF (space2 ← FindMappedSpace[sgList[i].space]) # space1 THEN EXIT;
      i ← i + 1;
      ENDLOOP;
    Space.Delete[space1];
    IF i = LENGTH[sgList] THEN EXIT;
    space1 ← space2;
    i ← i + 1;
    ENDLOOP;
  RETURN;
  END;

FindMappedSpace: PROCEDURE [space: Space.Handle] RETURNS [Space.Handle] =
  BEGIN
  mapped: BOOLEAN;
  parent: Space.Handle;
  DO
    [mapped: mapped, parent: parent] ← Space.GetAttributes[space];
    IF mapped THEN RETURN[space];
    space ← parent;
    ENDLOOP;
  END;

SGItem: TYPE = RECORD [sgh: SGHandle, space: Space.Handle];

SGList: TYPE = DESCRIPTOR FOR ARRAY CARDINAL [0..0) OF SGItem;

FindSegments: PROCEDURE [bcd: BcdBase] RETURNS [sgList: SGList] =
  BEGIN
  n: CARDINAL;
  sgItem: SGItem;
  CountSegs: PROCEDURE [sgh: SGHandle, sgi: SGIndex] RETURNS [BOOLEAN] = {
    IF sgh.class # code THEN RETURN[FALSE];
    IF sgh.file # FTSelf THEN BadFile[bcd, sgh.file];
    n ← n + 1;
    RETURN[FALSE]};
  AddSeg: PROCEDURE [sgh: SGHandle, sgi: SGIndex] RETURNS [BOOLEAN] = {
```

```
    IF sgh.class # code THEN RETURN[FALSE];
    sgList[n] ← [sgh: sgh, space: Space.nullHandle];
    n ← n + 1;
    RETURN[FALSE]};
  SiftUp: PROCEDURE [low, high: CARDINAL] = {
    k, son: CARDINAL;
    sgItem: SGItem;
    k ← low;
    DO
      IF k*2 > high THEN EXIT;
      IF k*2 + 1 > high OR sgList[k*2 + 1 - 1].sgh.base < sgList[
        k*2 - 1].sgh.base THEN son ← k*2
      ELSE son ← k*2 + 1;
      IF sgList[son - 1].sgh.base < sgList[k - 1].sgh.base THEN EXIT;
      sgItem ← sgList[son - 1];
      sgList[son - 1] ← sgList[k - 1];
      sgList[k - 1] ← sgItem;
      k ← son;
      ENDLOOP;
    RETURN};
  n ← 0;
  [] ← BcdOps.ProcessSegs[bcd, CountSegs];
  sgList ← DESCRIPTOR[GetSpace[n*SIZE[SGItem]], n];
  n ← 0;
  [] ← BcdOps.ProcessSegs[bcd, AddSeg];
  FOR n DECREASING IN [1..LENGTH[sgList]/2] DO
    SiftUp[n, LENGTH[sgList]] ENDLOOP;
  FOR n DECREASING IN [1..LENGTH[sgList]) DO
    sgItem ← sgList[1 - 1];
    sgList[1 - 1] ← sgList[n + 1 - 1];
    sgList[n + 1 - 1] ← sgItem;
    SiftUp[1, n];
    ENDLOOP;
  END;

AllocateCodeSpaces: PROCEDURE [bcd: BcdBase, sgList: SGList] =
  BEGIN
  start, end, startBase, pages, offset: CARDINAL;
  space, subSpace: Space.Handle;
  start ← end ← 0;
  DO
    startBase ← sgList[start].sgh.base;
    pages ← sgList[start].sgh.pages;
    DO
      end ← end + 1;
      IF end = LENGTH[sgList] OR pages + sgList[end].sgh.pages > 255 THEN EXIT;
      pages ← pages + sgList[end].sgh.pages;
      ENDLOOP;
    space ← SpecialSpace.CreateForCode[
      size: pages, parent: Space.virtualMemory, base: Space.defaultBase];
    Space.Map[
      space: space,
      window: Space.WindowOrigin[
      file: File.LimitPermissions[bcdCap, File.read],
      base: bcdSpaceBase + sgList[start].sgh.base - 1]];
    offset ← 0;
    FOR index: CARDINAL IN [start..end) DO
      sgh: SGHandle ← sgList[index].sgh;
```



```
      sph: SPHandle ← FindSPHandle[bcd, sgh];
      IF sph = NIL THEN
        subSpace ← Space.Create[size: sgh.pages, parent: space, base: offset]
      ELSE {
        FOR sp: CARDINAL DECREASING IN [0..sph.length) DO
          subSpace ← Space.Create[
            size: sph.spaces[sp].pages, parent: space,
            base: offset + sph.spaces[sp].offset];
          IF sph.spaces[sp].resident THEN SpecialSpace.MakeResident[subSpace];
          ENDLOOP};
      sgList[index].space ← subSpace;
      offset ← offset + sgh.pages;
      ENDLOOP;
    IF end = LENGTH[sgList] THEN EXIT ELSE start ← end;
    ENDLOOP;
  RETURN;
  END;

FindSPHandle: PUBLIC PROCEDURE [bcd: BcdBase, sgh: SGHandle]
  RETURNS [sph: SPHandle] =
  BEGIN
  sgb: BcdDefs.Base = LOOPHOLE[bcd + bcd.sgOffset];
  spb: BcdDefs.Base = LOOPHOLE[bcd + bcd.spOffset];
  spi: SPIndex;
  FOR spi ← FIRST[SPIndex], spi + SIZE[SPRecord] + sph.length*SIZE[SpaceID]
    UNTIL spi = bcd.spLimit DO
    sph ← @spb[spi]; IF @sgb[sph.seg] = sgh THEN RETURN[sph]; ENDLOOP;
  RETURN[NIL]
  END;

SgiToLP: PROCEDURE [bcd: BcdBase, sgi: SGIndex, sgList: SGList]
  RETURNS [LONG POINTER] =
  BEGIN
  sgb: BcdDefs.Base = LOOPHOLE[bcd + bcd.sgOffset];
  i: CARDINAL;
  FOR i IN [0..LENGTH[sgList]) DO
    IF @sgb[sgi] = sgList[i].sgh THEN
      RETURN[Space.LongPointer[sgList[i].space]];
    ENDLOOP;
  RETURN[NIL]
  END;

BadCode: PUBLIC SIGNAL [name: STRING] = CODE;
MissingCode: PUBLIC SIGNAL [name: STRING] = CODE;

InvalidModule: PROCEDURE [bcd: BcdBase, mth: MTHandle] =
  BEGIN
  name: STRING ← [40];
  dummy: BOOLEAN ← TRUE;
  AppendName[s: name, ssb: LOOPHOLE[bcd + bcd.ssOffset], name: mth.name];
  IF dummy THEN ERROR BadCode[name];
  END;

BadFile: PROCEDURE [bcd: BcdBase, fti: FTIndex] =
  BEGIN
  name: STRING ← [40];
  fth: FTHandle = @LOOPHOLE[bcd + bcd.ftOffset, Base][fti];
  dummy: BOOLEAN ← TRUE;
```

```
  AppendName[s: name, ssb: LOOPHOLE[bcd + bcd.ssOffset], name: fth.name];
  IF dummy THEN ERROR MissingCode[name];
  END;

-- Binding and Map management

InitBinding: PUBLIC PROCEDURE [bcd: BcdBase] RETURNS [binding: Binding] =
  BEGIN
  i: CARDINAL;
  p: POINTER ← GetSpace[bcd.nDummies*SIZE[BindLink]];
  binding ← DESCRIPTOR[
    p - CARDINAL[bcd.firstdummy*SIZE[BindLink]], bcd.nDummies];
  FOR i IN [bcd.firstdummy..bcd.firstdummy + bcd.nDummies) DO
    binding[i] ← [whichgfi: 0, body: notbound[]]; ENDLOOP;
  END;

ReleaseBinding: PUBLIC PROCEDURE [bcd: BcdBase, binding: Binding]
  RETURNS [Binding] =
  BEGIN
  IF BASE[binding] # NIL THEN
    FreeSpace[BASE[binding] + bcd.firstdummy*SIZE[BindLink]];
  RETURN[DESCRIPTOR[NIL, 0]];
  END;

InitializeMap: PUBLIC PROCEDURE [bcd: BcdBase]
  RETURNS [map: PilotLoadStateOps.Map] =
  BEGIN
  i: CARDINAL;
  map ← DESCRIPTOR[GetSpace[bcd.firstdummy], bcd.firstdummy];
  FOR i IN [0..bcd.firstdummy) DO map[i] ← 0; ENDLOOP;
  END;

DestroyMap: PUBLIC PROCEDURE [map: PilotLoadStateOps.Map] =
  BEGIN IF BASE[map] # NIL THEN FreeSpace[BASE[map]]; END;

-- Link management

links: LONG POINTER TO ARRAY [0..0) OF PrincOps.ControlLink;
long, writeable: BOOLEAN;

OpenLinkSpace: PROCEDURE [frame: GlobalFrameHandle, mth: BcdOps.MTHandle] =
  BEGIN
  IF frame.codelinks THEN {
    long ← TRUE; links ← RuntimeInternal.Codebase[LOOPHOLE[frame]]}
  ELSE {long ← FALSE; links ← LOOPHOLE[LONG[frame]]};
  links ← links - mth.frame.length;
  writeable ← FALSE;
  END;

WriteLink: PROCEDURE [offset: CARDINAL, link: PrincOps.ControlLink] = {
  IF long AND ~writeable THEN {
    cap: File.Capability;
    space: Space.Handle;
    writeable ← TRUE;
    space ← FindMappedSpace[Space.GetHandle[Space.PageFromLongPointer[links]]];
    cap ← Space.GetWindow[space].file;
    cap.permissions ← cap.permissions + File.write;
    Space.MakeWritable[space, cap]};
```



```
  links[offset] ← link};

ReadLink: PROCEDURE [offset: CARDINAL] RETURNS [link: PrincOps.ControlLink] =
  BEGIN RETURN[links[offset]]; END;

CloseLinkSpace: PROCEDURE [frame: GlobalFrameHandle] = {
  OPEN Space;
  IF long AND writeable THEN
    MakeReadOnly[FindMappedSpace[GetHandle[PageFromLongPointer[links]]]]};

-- Free storage management

heap: MDSZone ← Heap.systemMDSZone;

GetSpace: PUBLIC PROCEDURE [nwords: CARDINAL] RETURNS [p: POINTER] = {
  RETURN[Heap.MakeMDSNode[z: heap, n: nwords]]};

FreeSpace: PUBLIC PROCEDURE [p: POINTER] = {Heap.FreeMDSNode[z: heap, p: p]};


END....
```

```
-- RuntimeLoader>PilotLoadState.mesa (last edited by Luniewski on July 8, 1980 3:23 PM)

DIRECTORY
  BcdOps USING [BcdBase],
  Environment USING [PageNumber, PageCount, wordsPerPage],
  Inline USING [LongCOPY],
  PilotLoaderOps USING [FreeSpace, GetSpace, PilotLoaderSupport],
  PilotLoadStateFormat USING [
    PilotVersionID, ConfigIndex, LoadState, ModuleInfo, ModuleTable, NullConfig,
    NullModule],
  PilotLoadStateOps USING [EnumerationDirection, Map],
  PrincOps USING [GFTIndex],
  RuntimeInternal USING [loadStatePage],
  RuntimePrograms USING [],
  SDDefs USING [SD, sGFTLength],
  Space USING [
    Create, GetAttributes, GetHandle, Handle, LongPointer, LongPointerFromPage,
    Map, PageFromLongPointer, virtualMemory, VMPageNumber];

PilotLoadState: MONITOR
  IMPORTS Inline, RuntimeInternal, Space, PilotLoaderOps
  EXPORTS PilotLoadStateOps, RuntimePrograms =

  BEGIN OPEN PilotLoadStateOps, PilotLoadStateFormat, PrincOps;

  loadstate: LONG LoadState; -- the working copy of the load state.
  gft: LONG ModuleTable;
  queue: CONDITION;
  inuse: BOOLEAN ← TRUE;

  LoadStateFull: PUBLIC SIGNAL = CODE;
  LoadStateInvalid: PUBLIC SIGNAL = CODE;
  InitializePilotLoadState: PUBLIC PROCEDURE [
    pageInitialLoadState: Environment.PageNumber,
    countInitialLoadState: Environment.PageCount] = -- Module initialization:
    BEGIN OPEN Space;
    workingLoadState: Space.Handle ← Space.Create[
      countInitialLoadState, Space.virtualMemory];
    Space.Map[workingLoadState];
    Inline.LongCOPY[
      from: LongPointerFromPage[pageInitialLoadState],
      to: loadstate ← Space.LongPointer[workingLoadState],
      nwords: countInitialLoadState*Environment.wordsPerPage];
    gft ← DESCRIPTOR[@loadstate.gft, SDDefs.SD[SDDefs.sGFTLength]];
    RuntimeInternal.loadStatePage ← Space.VMPageNumber[workingLoadState];
    -- set everyone to now use the working load state.
    ReleaseLoadState[];
    START PilotLoaderOps.PilotLoaderSupport;
    END;

  InputLoadState: PUBLIC ENTRY PROCEDURE RETURNS [ConfigIndex] =
    BEGIN
    ENABLE UNWIND => NULL;
    WHILE inuse DO WAIT queue ENDLOOP;
    IF loadstate.versionident # PilotVersionID THEN ERROR LoadStateInvalid;
    inuse ← TRUE;
    RETURN[loadstate.nBcds]
    END;
```



```
  ReleaseLoadState: PUBLIC ENTRY PROCEDURE =
    BEGIN ENABLE UNWIND => NULL; inuse ← FALSE; NOTIFY queue; END;

  UpdateLoadState: PUBLIC PROCEDURE [config: ConfigIndex, bcd: BcdOps.BcdBase] =
    BEGIN
    size: CARDINAL;
    IF bcd = NIL THEN ERROR LoadStateInvalid;
    IF config >= LAST[ConfigIndex] THEN ERROR LoadStateFull;
    size ← Space.GetAttributes[
      Space.GetHandle[Space.PageFromLongPointer[bcd]]].size;
    loadstate.bcds[config] ←
      [exports: bcd.nExports # 0, typeExported: bcd.typeExported, pages: size,
        base: bcd];
    IF config >= loadstate.nBcds THEN loadstate.nBcds ← loadstate.nBcds + 1;
    END;

  RemoveConfig: PUBLIC PROCEDURE [map: Map, config: ConfigIndex] =
    BEGIN
    i: CARDINAL;
    FOR i IN [1..LENGTH[gft]) DO
      IF gft[i].config > config AND gft[i].config # NullConfig THEN
        gft[i].config ← gft[i].config - 1;
      ENDLOOP;
    FOR i IN [1..LENGTH[map]) DO gft[map[i]] ← NullModule; ENDLOOP;
    FOR i IN [config..loadstate.nBcds) DO
      loadstate.bcds[i] ← loadstate.bcds[i + 1]; ENDLOOP;
    END;

  ForceDirty: PUBLIC PROCEDURE = BEGIN END;

  EnterModule: PUBLIC PROCEDURE [rgfi: GFTIndex, module: ModuleInfo] = {
    gft[rgfi] ← module; };

  GetModule: PUBLIC PROCEDURE [rgfi: GFTIndex] RETURNS [module: ModuleInfo] = {
    RETURN[gft[rgfi]]; };

  MapConfigToReal: PUBLIC PROCEDURE [cgfi: GFTIndex, config: ConfigIndex]
    RETURNS [rgfi: GFTIndex] =
    BEGIN
    IF cgfi = 0 THEN RETURN[0];
    FOR rgfi IN [0..LENGTH[gft]) DO
      IF gft[rgfi].config = config AND gft[rgfi].gfi = cgfi THEN RETURN[rgfi];
      ENDLOOP;
    RETURN[0];
    END;

  MapRealToConfig: PUBLIC PROCEDURE [rgfi: GFTIndex]
    RETURNS [cgfi: GFTIndex, config: ConfigIndex] = {
    RETURN[gft[rgfi].gfi, gft[rgfi].config]; };

  GetMap: PUBLIC PROCEDURE [config: ConfigIndex] RETURNS [map: Map] =
    BEGIN
    max: CARDINAL ← 0;
    i: GFTIndex;
    FOR i IN [0..LENGTH[gft]) DO
      IF gft[i].config = config THEN max ← MAX[max, gft[i].gfi]; ENDLOOP;
    map ← DESCRIPTOR[PilotLoaderOps.GetSpace[max + 1], max + 1];
    FOR i IN [0..LENGTH[map]) DO map[i] ← 0; ENDLOOP;
```

```
    FOR i IN [0..LENGTH[gft]) DO
      IF gft[i].config = config THEN map[gft[i].gfi] ← i; ENDLOOP;
    END;

  ReleaseMap: PUBLIC PROCEDURE [map: Map] = {
    IF BASE[map] # NIL THEN PilotLoaderOps.FreeSpace[BASE[map]]; };

  AcquireBcd: PUBLIC PROCEDURE [config: ConfigIndex]
    RETURNS [bcd: BcdOps.BcdBase] = {RETURN[loadstate.bcds[config].base]; };

  ReleaseBcd: PUBLIC PROCEDURE [bcd: BcdOps.BcdBase] = {};

  EnumerateModules: PUBLIC PROCEDURE [
    proc: PROCEDURE [GFTIndex, ModuleInfo] RETURNS [BOOLEAN]] RETURNS [GFTIndex] =
    BEGIN
    i: GFTIndex;
    FOR i IN [0..LENGTH[gft]) DO IF proc[i, gft[i]] THEN RETURN[i]; ENDLOOP;
    RETURN[0]
    END;

  EnumerateBcds: PUBLIC PROCEDURE [
    dir: EnumerationDirection, proc: PROCEDURE [ConfigIndex] RETURNS [BOOLEAN]]
    RETURNS [config: ConfigIndex] =
    BEGIN
    SELECT dir FROM
      recentfirst =>
        FOR config DECREASING IN [0..loadstate.nBcds) DO
          IF proc[config] THEN RETURN[config]; ENDLOOP;
      recentlast =>
        FOR config IN [0..loadstate.nBcds) DO
          IF proc[config] THEN RETURN[config]; ENDLOOP;
      ENDCASE;
    RETURN[NullConfig]
    END;

  BcdUnresolved: PUBLIC PROCEDURE [bcd: ConfigIndex] RETURNS [BOOLEAN] =
    BEGIN
    i: GFTIndex;
    FOR i IN [0..LENGTH[gft]) DO
      IF bcd = gft[i].config AND ~gft[i].resolved THEN RETURN[TRUE]; ENDLOOP;
    RETURN[FALSE];
    END;

  BcdExports: PUBLIC PROCEDURE [bcd: ConfigIndex] RETURNS [BOOLEAN] = {
    RETURN[loadstate.bcds[bcd].exports]; };

  BcdExportsTypes: PUBLIC PROCEDURE [bcd: ConfigIndex] RETURNS [BOOLEAN] = {
    RETURN[loadstate.bcds[bcd].typeExported]; };

  END.
LOG
Time: January 30, 1980 8:32 AM      By: Sandman      Action: Created file.
Time: January 31, 1980 4:50 AM      By: Forrest      Action: ?
Time: April 14, 1980 11:19 AM  By: Knutsen      Action: Module now STARTed by InitializePil
**otLoadState[]
Time: May 17, 1980 4:17 PM     By: Forrest      Action: Mesa 6
Time: July 8, 1980 3:23 PM     By: Knutsen      Action: Pass parameters for initial loadState.
**Create working loadState on the fly.
```

```
-- PilotUnLoader.mesa; edited by Sandman on August 28, 1980 2:02 PM

DIRECTORY
  BcdDefs USING [Base, MTIndex],
  BcdOps USING [BcdBase, MTHandle, ProcessModules],
  Inline USING [BITAND],
  PilotLoaderOps USING [
    CloseLinkSpace, FreeSpace, GetSpace, OpenLinkSpace, ReadLink, WriteLink],
  PilotLoadStateFormat USING [ModuleInfo],
  PilotLoadStateOps USING [
    AcquireBcd, ConfigIndex, EnterModule, EnumerateBcds, GetMap, GetModule,
    InputLoadState, MapRealToConfig, ReleaseBcd, ReleaseLoadState, ReleaseMap,
    Map, RemoveConfig],
  Mopcodes USING [zFREE],
  PrincOps USING [
    ControlLink, FrameCodeBase, GFTIndex, GFTNull, GlobalFrameHandle,
    NullGlobalFrame, PrefixHandle, TrapLink, UnboundLink],
  PrincOpsRuntime USING [EmptyGFTItem, GetFrame, GFT],
  Runtime USING [GlobalFrame, UnNew],
  RuntimeInternal USING [GetNextGlobalFrame],
  Space USING [
    Delete, GetAttributes, GetHandle, Handle, PageFromLongPointer, Pointer,
    wordsPerPage],
  SubSys USING [];

PilotUnLoader: PROGRAM
  IMPORTS
    BcdOps, Inline, PilotLoaderOps, PilotLoadStateOps, PrincOpsRuntime, Runtime,
    RuntimeInternal, Space
  EXPORTS Runtime, SubSys =
  BEGIN OPEN PrincOps, RuntimeInternal;

  ConfigIndex: TYPE = PilotLoadStateOps.ConfigIndex;
  Map: TYPE = PilotLoadStateOps.Map;
  BcdBase: TYPE = BcdOps.BcdBase;
  GlobalFrameHandle: TYPE = PrincOps.GlobalFrameHandle;
  NullGlobalFrame: GlobalFrameHandle = PrincOps.NullGlobalFrame;

  Item: TYPE = RECORD [next: List, space: Space.Handle, pages: CARDINAL];
  List: TYPE = POINTER TO Item;

  fList, cList: List;

  UnNewConfig, UnLoad: PUBLIC PROC [link: UNSPECIFIED] =
    BEGIN OPEN PilotLoadStateOps;
    config: ConfigIndex;
    map: Map;
    bcd: BcdBase;
    frame, f: GlobalFrameHandle;
    frame ← LOOPHOLE[Runtime.GlobalFrame[link]];
    IF frame.copied THEN
      FOR f ← GetNextGlobalFrame[NullGlobalFrame], GetNextGlobalFrame[f] UNTIL f =
        NullGlobalFrame DO
        IF f # frame AND SameCode[frame, f] AND ~f.copied THEN
          BEGIN frame ← f; EXIT END;
        ENDLOOP;
    [] ← InputLoadState[];
    BEGIN
```



```
    ENABLE UNWIND => ReleaseLoadState[];
    [config: config] ← MapRealToConfig[frame.gfi];
    bcd ← AcquireBcd[config];
    map ← GetMap[config];
    CollectSpaces[map];
    UnBindConfig[config, bcd, map];
    CleanupFrames[map];
    RemoveConfig[map, config];
    ReleaseMap[map];
    ReleaseBcd[bcd];
    END; -- ELBANE
    ReleaseLoadState[];
    RETURN
    END;

  CollectSpaces: PROC [map: Map] =
    BEGIN OPEN PrincOpsRuntime, Space;
    frame: GlobalFrameHandle;
    i: CARDINAL;
    fList ← cList ← NIL;
    FOR i IN [1..LENGTH[map]) DO
      frame ← GetFrame[GFT[map[i]]];
      IF frame = NullGlobalFrame OR GFT[map[i]].epbias # 0 THEN LOOP;
      IF LENGTH[map] > 2 THEN
        fList ← AddSpaceToList[list: fList, space: FindMappedSpace[frame]];
      cList ← AddSpaceToList[
        list: cList, space: FindMappedSpace[frame.code.longbase]];
      ENDLOOP;
    RETURN
    END;

  FindMappedSpace: PROCEDURE [lp: LONG POINTER] RETURNS [space: Space.Handle] =
    BEGIN
    mapped: BOOLEAN;
    parent: Space.Handle;
    space ← Space.GetHandle[Space.PageFromLongPointer[lp]];
    DO
      [mapped: mapped, parent: parent] ← Space.GetAttributes[space];
      IF mapped THEN RETURN[space];
      space ← parent;
      ENDLOOP;
    END;

  AddSpaceToList: PROCEDURE [list: List, space: Space.Handle] RETURNS [List] =
    BEGIN
    l: List;
    FOR l ← list, l.next UNTIL l = NIL DO
      IF l.space = space THEN RETURN[list]; ENDLOOP;
    l ← PilotLoaderOps.GetSpace[SIZE[Item]];
    l↑ ← [next: list, space: space, pages: Space.GetAttributes[space].size];
    RETURN[l];
    END;

  SameCode: PROC [f1, f2: GlobalFrameHandle] RETURNS [BOOLEAN] =
    BEGIN
    fcb1, fcb2: PrincOps.FrameCodeBase;
    fcb1 ← f1.code;
    fcb2 ← f2.code;
    fcb1.out ← fcb2.out ← FALSE;
```

```
  RETURN[fcb1 = fcb2];
  END;

DestroyCopy: PROC [f: GlobalFrameHandle] RETURNS [BOOLEAN] =
  BEGIN Runtime.UnNew[LOOPHOLE[f]]; RETURN[FALSE]; END;

Free: PROC [POINTER] = MACHINE CODE BEGIN Mopcodes.zFREE END;

CleanupFrames: PROC [map: Map] =
  BEGIN OPEN PrincOpsRuntime, Space;
  frame: GlobalFrameHandle;
  f, c: List;
  i, ep: CARDINAL;
  FOR i IN [1..LENGTH[map]) DO
    frame ← GetFrame[GFT[map[i]]];
    ep ← GFT[map[i]].epbias;
    IF frame = NullGlobalFrame OR ep # 0 THEN LOOP;
    [] ← EnumerateCopies[frame, DestroyCopy];
    IF frame.alloced THEN
      BEGIN
      Align: PROC [POINTER, WORD] RETURNS [POINTER] = LOOPHOLE[Inline.BITAND];
      IF frame.codelinks THEN Free[frame]
      ELSE
        BEGIN
        nlinks: [0..256) ← FindNLinks[frame];
        Free[Align[frame - nlinks, 177774B]];
        END;
      END;
    ENDLOOP;
  FOR f ← fList, fList UNTIL f = NIL DO
    Space.Delete[f.space]; fList ← f.next; PilotLoaderOps.FreeSpace[f]; ENDLOOP;
  FOR i IN [1..LENGTH[map]) DO GFT[map[i]] ← EmptyGFTItem ENDLOOP;
  FOR frame ← GetNextGlobalFrame[NullGlobalFrame], GetNextGlobalFrame[frame]
    UNTIL frame = NullGlobalFrame DO
    space: Space.Handle ← FindMappedSpace[frame.code.longbase];
    FOR c ← cList, c.next UNTIL c = NIL DO
      IF space = c.space THEN {c.pages ← 0; EXIT}; ENDLOOP;
    ENDLOOP;
  FOR c ← cList, cList UNTIL c = NIL DO
    IF c.pages # 0 THEN Space.Delete[c.space];
    cList ← c.next;
    PilotLoaderOps.FreeSpace[c];
    ENDLOOP;
  RETURN
  END;

FindNLinks: PROC [frame: GlobalFrameHandle] RETURNS [nlinks: CARDINAL] = INLINE
  BEGIN
  RETURN[LOOPHOLE[frame.code.longbase, LONG PrefixHandle].header.info.nlinks]
  END;

NextMultipleOfFour: PROC [n: UNSPECIFIED] RETURNS [UNSPECIFIED] = INLINE {
  RETURN[n + Inline.BITAND[-n, 3B]]};

UnBindConfig: PROC [config: ConfigIndex, bcd: BcdBase, map: Map] =
  BEGIN OPEN PilotLoadStateOps;
  bmap: Map;
  UnBind: PROC [c: ConfigIndex] RETURNS [BOOLEAN] =
    BEGIN
```



```
    bindee: BcdBase;
    IF c = config THEN RETURN[FALSE];
    bindee ← AcquireBcd[c];
    IF bindee.nImports # 0 THEN
      BEGIN
      bmap ← GetMap[c];
      [] ← BcdOps.ProcessModules[bindee, AdjustLinks];
      ReleaseMap[bmap];
      END;
    ReleaseBcd[bindee];
    RETURN[FALSE]
    END;
  AdjustLinks: PROC [mth: BcdOps.MTHandle, mti: BcdDefs.MTIndex]
    RETURNS [BOOLEAN] =
    BEGIN OPEN PrincOpsRuntime;
    gfi: PrincOps.GFTIndex = bmap[mth.gfi];
    frame: GlobalFrameHandle = GetFrame[GFT[gfi]];
    changed: BOOLEAN;
    AdjustCopies: PROC [f: GlobalFrameHandle] RETURNS [BOOLEAN] = {
      [] ← AdjustFrame[f, mth]; RETURN[FALSE]};
    IF frame = NullGlobalFrame OR mth.frame.length = 0 THEN RETURN[FALSE];
    changed ← AdjustFrame[frame, mth];
    IF changed AND frame.shared AND ~frame.codelinks THEN
      [] ← EnumerateCopies[frame, AdjustCopies];
    IF changed THEN
      BEGIN
      info: PilotLoadStateFormat.ModuleInfo ← GetModule[gfi];
      info.resolved ← FALSE;
      EnterModule[rgfi: gfi, module: info];
      END;
    RETURN[FALSE]
    END;
  AdjustFrame: PROC [frame: GlobalFrameHandle, mth: BcdOps.MTHandle]
    RETURNS [changed: BOOLEAN] =
    BEGIN
    i: CARDINAL;
    link: PrincOps.ControlLink;
    changed ← FALSE;
    PilotLoaderOps.OpenLinkSpace[frame, mth];
    FOR i IN [0..mth.frame.length) DO
      link ← PilotLoaderOps.ReadLink[i];
      IF BoundHere[link, mth.frame.frag[i].vtag = var] THEN
        BEGIN
        link ← IF link.proc OR link.indirect THEN UnboundLink ELSE TrapLink;
        PilotLoaderOps.WriteLink[i, link];
        changed ← TRUE;
        END;
      ENDLOOP;
    PilotLoaderOps.CloseLinkSpace[frame];
    RETURN
    END;
  BoundHere: PROC [link: ControlLink, var: BOOLEAN] RETURNS [BOOLEAN] =
    BEGIN
    i: CARDINAL;
    gfi: GFTIndex ← GFTNull;
    SELECT TRUE FROM
      var => {
        op: ORDERED POINTER = LOOPHOLE[link];
        f: List;
```

```
        IF LENGTH[map] = 2 THEN RETURN[op IN [frameStart..frameEnd)]
        ELSE
          FOR f ← fList, f.next UNTIL f = NIL DO
            frameStart ← LOOPHOLE[Space.Pointer[f.space]];
            frameEnd ← frameStart + Space.wordsPerPage*f.pages;
            IF op IN [frameStart..frameEnd) THEN RETURN[TRUE];
            ENDLOOP;
          RETURN[FALSE]};
      link.proc => gfi ← link.gfi;
      ENDCASE => RETURN[FALSE];
    FOR i IN [1..LENGTH[map]) DO IF map[i] = gfi THEN RETURN[TRUE]; ENDLOOP;
    RETURN[FALSE];
    END;
  frameStart, frameEnd: ORDERED POINTER;
  IF bcd.nExports = 0 AND bcd.nModules # 1 THEN RETURN;
  frameStart ← LOOPHOLE[PrincOpsRuntime.GetFrame[PrincOpsRuntime.GFT[map[1]]]];
  IF bcd.nModules = 1 THEN
    BEGIN OPEN BcdDefs;
    mth: BcdOps.MTHandle = @LOOPHOLE[bcd + bcd.mtOffset, BcdDefs.Base][
      FIRST[BcdDefs.MTIndex]];
    frameEnd ← LOOPHOLE[frameStart + mth.framesize]
    END;
  [] ← EnumerateBcds[recentfirst, UnBind];
  RETURN
  END;

EnumerateCopies: PROC [
  frame: GlobalFrameHandle, proc: PROC [GlobalFrameHandle] RETURNS [BOOLEAN]]
  RETURNS [result: GlobalFrameHandle] =
  BEGIN
  f: GlobalFrameHandle;
  IF ~frame.shared THEN RETURN[NullGlobalFrame];
  FOR f ← GetNextGlobalFrame[NullGlobalFrame], GetNextGlobalFrame[f] UNTIL f =
    NullGlobalFrame DO
    IF f # frame AND f.copied AND SameCode[frame, f] AND proc[f] THEN EXIT;
    ENDLOOP;
  RETURN[f];
  END;

END...
```