DPKernel

Paul Manias

COLLABORATORS			
	TITLE : DPKernel		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Paul Manias	July 26, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

DPKernel

1.1 Library: DPKernel

```
SYSTEM DOCUMENTATION
Name:      DPKERNEL
Version:   0.9 Beta
Date:      December 1997
Author:    Paul Manias
Copyright: DreamWorld Productions, 1996-1997.  All rights reserved.
Notes:     This  document is still being written and will contain errors
           in  a  number  of  places.  The information within cannot be
           treated as official until this autodoc reaches version 1.0.

CHANGES VERSION 0.9B
Added:   Flush()
         Read() (moved from files.guide)
         Reset()
         Write() (moved from files.guide)
         Detach()
         AutoStop()

         Moved from the master.guide
         FastRandom()
         ReadKey()
         SlowRandom()
         WaitLMB()
         WaitTime()
         WriteDec()

Deleted: SetUserPrefs()
         WaitLMB()
         WriteDec()
         ReadKey()
         AutoSwitch()

Edited:  WaitTime()
         CloseDPK()
         Activate()
         AddSysObject()
```

1.2 DPKernel

```
OBJECTS
Module
RawData
Reference
Segment
SysObject
Task
Time

FUNCTIONS
AddSysEvent()
AddSysObject()
AllocMemBlock()
AutoStop()
CloseDPK()
DPKForbid()
DPKPermit()
FastRandom()
FindDPKTask()
FindSysObject()
FreeMemBlock()
GetMemSize()
GetMemType()
OpenModule()
RemSysEvent()
SlowRandom()
Switch()
TagInit()
TotalMem()
WaitTime()

Resource Tracking
AddTrack()
DeleteTrack()
InitDestruct()
SelfDestruct()

ACTIONS
Activate()
Clear()
CopyStructure()
Deactivate()
Detach()
Display()
Draw()
Flush()
Free()
Get()
Hide()
Init()
Load()
Lock()
Query()
Read()
```

```
 Reset()
 Save()
 Unlock()
 Write()
```

1.3 Kernel: Activate()

```
ACTION
Name:     Activate()
Short:    Perform the native action of the object.
Synopsis: LONG Activate(APTR Object [a0]);

DESCRIPTION
Executes the native action for a given object (but only if the object
supports Activate()). The action taken by each object class is extremely
varied, here are some examples of what some current objects do for
Activate():

  Sound     - Plays sample data over the speakers.
  Restore   - Replaces destroyed backgrounds.
  Directory - Grabs a list of directory contents.

Some objects will continue to perform their native action after they return
back to you, for example a Sound will play in the background while you
continue processing. If for some reason you wish to cancel an action, you
can attempt a call to Deactivate().

INPUT
Object - Pointer to an initialised object structure.

RESULT
Returns ERR_OK on success.

SEE ALSO
Kernel: Deactivate()
```

1.4 Kernel: AddSysEvent()

```
FUNCTION
Name:     AddSysEvent()
Short:    Adds a new event node to the system.
Synopsis: *Event AddSysEvent(LONG *Tags)

DESCRIPTION
Note: This area of the OS is currently under development, so you use it at
your own risk.

This function adds a new event node to the system. An event is a
representation of a particular occurance in the operating system. Such
examples of this are the appearance of a new task, detecting a disk
inserted by the user, low memory, a new screen on display, and so on.
```

Each event is handled by a particular area of the system. Sending floppy disk events is handled by the I/O routines for example. When a certain event is signalled by one of these handlers, the system will call all the nodes that are chained into that event. For an application or module to add a new node to one of these event chains, it needs to use AddSysEvent().

Currently available events are:

```
  OnNewTask
  OnRemTask
```

EXAMPLE
This section of code will create a node that is linked to the OnNewTask event. Note that the node that is created will need to be removed with RemSysEvent() before the Task exits.

```
  EVTOnNewTask = AddSysEventTags(TAGS,NULL,
     EVA_Number,   EVT_OnNewTask,        /* Event type */
     EVA_Routine,  &OnNewTask,           /* Pointer to event routine */
     EVA_Priority, 3,                    /* Priority setting */
     TAGEND);
```

INPUT
Tags - Pointer to the event initialisation tags. See the Event object's documentation for details.

RESULT
Returns a pointer to the newly created Event node.

SEE ALSO
Kernel: RemSysEvent()

1.5 Kernel: AddSysObject()

FUNCTION
Name: AddSysObject()
Short: Adds a new object to the system.
Synopsis: *SysObject AddSysObject(APTR Tags [a0]);

DESCRIPTION
This function adds a new object to the system. The objective of this is to allow your object to support the various "Action" functions, and to give the system the necessary information about your object. Example:

```
  JPEGObject = AddSysObjectTags(TAGS,  NULL,
                 SOA_ObjectID,         ID_HIDDEN,
                 SOA_ClassID,          ID_PICTURE,
                 SOA_Name,             "Picture~Jpeg",
                 SOA_CopyToUniverse,   &PIC_CopyToUniverse,
                 SOA_CopyFromUniverse, &PIC_CopyFromUniverse,
                 SOA_Load,             &PIC_Load,
                 TAGEND);
```

In this example, AddSysObject() will create a Child type (called Jpeg) that belongs to the Picture class. Because child types inherit actions from the

parent, this Jpeg object will also support actions such as Query(), Init(), Get() and Free().

Master Objects
Any object that is initialised with a specific ID will be treated as a master object. If you call AddSysObject() when the given ID is already in use by another master object in the system, AddSysObject() will not create a new SysObject. Instead, it will add your object specifications to the existing SysObject.

For example, if you attempt to add a Picture master with a tag of SOA_Activate, and the existing Picture does not support that action, your supporting function will be added to the existing Picture master. If you specify SOA_Activate|TREPLACE, then your Activate function will be placed in the Picture master even if it already has support for Activate(). Please do not take the TREPLACE flag lightly - only do this if absolutely necessary.

Hidden Objects
You can declare a "hidden" system object by passing ID_HIDDEN. What this creates is an object that has no identification characters. This means that no program will be able to specifically search for your object in the system. For example, the Get() action will be completely useless in looking for your object. The only way to get to your object will be via the FindSysObject() function.

The main purpose for creating hidden objects is for simple object types that will only be used internally (in the system). For example, XPK support was implemented via a hidden object, because as a programmer you wouldn't want to deal with unpacking when loading data. Instead, the Load() function will invisibly detect XPK files for you and call upon the hidden XPK object to deal with the data.

Child Objects
To declare a child object, use ID_HIDDEN in SOA_ObjectID, then use the class identifer in SOA_ClassID (e.g. ID_PICTURE). This is useful in situations where you might want to add support for a new data format to an object. As an example, when a jpeg picture is loaded the master will fail to recognise it, because it only uses IFF files. This failure will cause the Load() function to look for child objects belonging to the Picture class and use those instead.

The other advantage as a child object is that you get function inheritance from the master. So if any action is unsupported by your object, you will still be able to inherit support from the master's functions.

INPUT
Tags - Pointer to a standard tag list.

RESULT
Returns a pointer to the created SysObject or NULL if failure.

SEE ALSO
Kernel: Free()

1.6 Kernel: AddTrack()

```
FUNCTION
Name:     AddTrack()
Short:    Adds a resource tracking node.
Synopsis: LONG AddTrack(LONG Resource [d0], LONG Data [d3], void *Routine [a0])

DESCRIPTION
This function is intended for use by the system modules, but you may use it
in your program if necessary.

About Resource Tracking
Each  task has a special list of all system resources that are currently in
use.  Each  resource has its own list node that uniquely identifies it (eg
each  memory  allocation  has  its  own  resource node).  When a particular
resource  has  been  freed  the  list  is  checked  and its related node is
deleted.  When the program shuts down, any resources that are still in the
list  can  be  freed  by  the  kernel so that they are not lingering in the
system.

When you call AddTrack() you need to give it a resource ID which is made up
from  one  of the following resource types.  This is important as resources
are freed from the system in an order based on these ID's.  The order looks
like this:

     1. Free all hardware based resources (blitter, sound, etc).
     2. Free complex resources (both hardware and software).
     3. Free customised resources (user defined types etc).
     4. Free memory (always freed last).

As  you  can  see  the correct order is vital, if memory was freed first it
would  have  adverse  affects  when  freeing  the complex and user resource
types.  Always make sure that you give the correct ID when describing your
resource.   The  ID's  are  RES_MEMORY,  RES_HARDWARE,  RES_COMPLEX  and
RES_CUSTOM, as outlined in the include file "system/tasks.i".

Passing  a  Data  pointer  is  optional,  but  you will need it to uniquely
identify  your  resource  later on (eg AllocMemBlock() uses this to store a
pointer to the allocated memory).

The  Routine  points  to code that will free the resource if it is still in
use  when  the  program  exits.  It  will  be passed the Data field of the
resource  node  in  register  d0,  and  the Key field in register d1.  This
routine  must save all the CPU registers that it uses and can only free its
resource  - it may not make new allocations of any sort.  It may send debug
and  error  messages,  which  will  help to work out what the problem is if
anything goes wrong.

INPUT
Resource - Correct resource identifier from system/tasks.i.
Data     - Optional data pointer to store in the resource node.
Routine  - Pointer to the routine to call when freeing the resource.

RESULT
Returns  a  key that AddTrack() has used to uniquely identify the resource.
You will need to store the key somewhere for when you call DeleteTrack().
```

```
SEE ALSO
Kernel:  DeleteTrack()
Include: system/tasks.i
```

1.7 Kernel: AllocMemBlock()

```
FUNCTION
Name:     AllocMemBlock()
Short:    Allocate a new memory block.
Synopsis: APTR AllocMemBlock(LONG Size [d0], LONG MemType [d1]);

DESCRIPTION
This function allocates a memory block from the system memory pool and
returns it to your program. By default all memory is public. This open
memory model is used to enhance the communication levels between tasks and
functions. If you need private memory make sure that you ask for it and
that you will be the only Task using it. An AllocPrivate() macro has been
included to make this type of allocation easier.

Protection and Resource Tracking
Header and Tail ID's are used to offer a security system similar to
MungWall, acting as cookies at each end of a memory block. You will be
alerted by FreeMemBlock() if the ID's are damaged. This is a permanent
debugging feature, so there is little need to run MungWall for debugging
your programs.

Resource tracking is automatic, so you will be warned if you allocate
memory and forget to free it on exit (ie when you close down). Any memory
that is found will be freed for your convenience.

By default all memory is cleared before it is given to you. Here are the
memory types:

  MEM_DATA
  Suitable for storing data and variables. This is the default. Note that
  you are disallowed from running code from this type of memory.

  MEM_CODE
  This can store and execute CPU instructons. It also fits the
  requirements of MEM_DATA, so you can store variables and data in it.

  MEM_VIDEO
  Is for displaying graphics, and is also compatible with the Blitter
  interface.

  MEM_BLIT
  Is memory that is compatible with the blitter module. Currently this
  module only uses chip memory, but future versions could also support CPU
  drawing from fast if the graphic is located in that area.

  MEM_SOUND
  For memory that is compatible with the Sound interface.

You may also use the following extra flags when making your memory
```

```
allocation:

  MEM_PRIVATE
  If other programs should not have access to your memory.

  MEM_UNTRACKED
  If  you don't want resource tracking on your allocation.  You should only
  use this flag if the memory is part of a complex resource that is already
  being tracked.

INPUT
Size    - Size of the required memblock in bytes.
MemType - The type of memory to allocate, eg MEM_VIDEO.

RESULT
Pointer to the start of your allocated memblock or NULL if failure.

SEE ALSO
Kernel: FreeMemBlock()
        GetMemSize()
        GetMemType()
```

1.8 Kernel: AutoStop()

```
FUNCTION
Name:     AutoStop()
Short:    Pauses your task when the user leaves your screen or windows.
Synopsis: void AutoStop(void)

DESCRIPTION
If the user moves away from your task by switching to a different screen or
window, you may want to pause your actions until the user returns the focus
back  to  your  task.   This  is  very useful in games and other "non-stop"
programs  which  can  continue  regardless  of  any user interaction.  Such
non-stop  programs  can be counter productive if the user is at a different
task, and your program is stealing resources in the background.

If  this  function  finds that the user has moved away, it pauses your task
immediately.   When the user returns to your task, this function returns to
execute the next instruction in your program.

SEE ALSO
Kernel: Switch()
```

1.9 Kernel: Clear()

```
ACTION
Name:     Clear()
Short:    Clears an object's graphic from its container.
Synopsis: LONG Clear(APTR Object [a0])

DESCRIPTION
```

This action will clear an object's graphic from its container. For example, Clear(Bob) will clear a Bob graphic from its assigned Bitmap. Note that it will not restore what was previously under the graphic - it just clears it all away. This means that you will get a empty rectangle or mask of the object, most likely with a colour of black.

INPUT
Object - Pointer to an initialised object.

RESULT
Returns ERR_OK if successful.

SEE ALSO
Kernel: Draw()

1.10 Kernel: CloseDPK()

FUNCTION
Name: CloseDPK()
Short: Closes the kernel library.
Synopsis: void CloseDPK(void)

DESCRIPTION
Before your program exits you will have to call this function so that the system knows you are shutting down. If you do not close the kernel before you exit you will leave certain memory allocations unfreed and there may be other adverse system effects.

This function will perform a resource tracking check, so if you have not freed any system resources you will be notified here (if you get a yellow alert box, you will need to use IceBreaker to get a detailed list of the errors).

CloseDPK() is responsible for handling the OnRemTask event. This event will be activated as soon as CloseDPK() is called by your task.

NOTE
Remember that you may not call any more functions after calling CloseDPK().

This function is used in the STARTDPK macro and the dpk.o file, so C and assembler programmers do not need to call this function explicitly. Programs that are started natively or from the StartDPK executable, will not have to call CloseDPK(). This is because the call will be handled within the system for these cases.

1.11 Kernel: CopyStructure()

ACTION
Name: CopyStructure()
Short: Copies details from one structure to another.
Synopsis: LONG CopyStructure(APTR Source [a0], APTR Destination [a1]);

```
DESCRIPTION
Copies one structure's data across to another. The structures can be
similar or completely different (even a Screen -> Sound copy is possible,
although there won't be much of a result).

This action only copies fields that are currently present in the Universe
object. This action DOES NOT copy fields that contain option flags, fields
that contain pointers to the main area of data (eg screen bitplanes), or
fields designated as private.

Only the NULL fields in the Destination structure will be written to. If
the Destination structure has already been initialised, you may find that
CopyStructure() has no effect due to this condition.

INPUT
Source      - Points to the source object.
Destination - Points to the destination object.

RESULT
Returns ERR_OK on success, otherwise may fail due to unrecognised objects
or invalid arguments.
```

1.12 Kernel: Deactivate()

```
FUNCTION
Name:     Deactivate()
Short:    Stop the object from performing its native action.
Synopsis: void Deactivate(APTR Object [a0]);

DESCRIPTION
Stops the given object from continuing its native action. This function
only works if the object has recently had Activate() called on it. If the
object is not performing any actions, then this function will simply do
nothing.

INPUT
Object - Pointer to an initialised object structure.

SEE ALSO
Kernel: Activate()
```

1.13 Kernel: DeleteTrack()

```
FUNCTION
Name:     DeleteTrack()
Short:    Delete a resource tracking node.
Synopsis: void DeleteTrack(LONG Key [d1]);

DESCRIPTION
Deletes a resource node allocated from AddTrack(). If resource nodes are
not deleted they will stay linked to the program and the system will
attempt to free them when the program shuts down.
```

```
Note  that  this  function only deletes the resource node, it will
not attempt to deallocate the resource from its deallocation function.

INPUT
Key - A key that was obtained from AddTrack().

SEE ALSO
Kernel: AddTrack()
```

1.14 Kernel: Detach()

```
ACTION
Name:     Detach()
Short:    Detach an object from a parent structure.
Synopsis: LONG Detach(APTR Object [a0], APTR Parent [a1])

DESCRIPTION
This  action  will  separate  an  object  from  a parent structure, without
causing  any undue harm to either object.  If this operation is successful,
you can then use the detached object independantly of its parent.

Some detachments are impossible or too dangerous to attempt, e.g.  a Screen
object cannot survive without its Bitmap.

INPUTS
Object - The object that needs to be detached.
Parent - Pointer to the parent object that you want to be detached from.

RESULT
Returns  ERR_OK  if  successful.  This call can fail if the object does not
support the Detach() action, or if it is not attached to a parent object in
the first place.
```

1.15 Kernel: Display()

```
ACTION
Name:     Display()
Short:    Displays an object inside its Container.
Synopsis: LONG Display(APTR Object [a0]);

DESCRIPTION
Calling this action will display the object in its specified container.  If
the  object  cannot  be  displayed,  or if the required container was never
specified on Init(), the the call will fail.

INPUT
Object - The object to be displayed.

RESULT
Returns ERR_OK on success.
```

```
SEE ALSO
Kernel: Hide()
```

1.16 Kernel: DPKForbid()

```
FUNCTION
Name:     DPKForbid()
Short:    Stop other tasks/processes from executing.
Synopsis: void DPKForbid(void)

DESCRIPTION
Stops all other tasks and processes from executing until you call
DPKPermit(). This call will not turn off interrupts.

NOTE
This function has little effect in systems that do not multi-task.

SEE ALSO
Kernel: DPKPermit()
```

1.17 Kernel: DPKPermit()

```
FUNCTION
Name:      DPKPermit()
Short:     Allow other tasks to continue their processing.
Synopsis:  void DPKPermit(void);

DESCRIPTION
Reverses a previous call to DPKForbid(), so that all tasks can continue
their normal processes.

SEE ALSO
Kernel: DPKForbid
```

1.18 Kernel: Draw()

```
ACTION
Name:     Draw()
Short:    Draws an object's graphic to its container.
Synopsis: LONG Draw(APTR Object [a0]);

DESCRIPTION
This action will draw an object's graphic to the container that it was
originally initialised to. If the object was never initialised to a
container that supports drawing, this action will have no effect.
Secondly, if the object has no graphical representation, then nothing can
be drawn.

This action is most often used in the drawing of Bobs and MBobs.
```

```
INPUT
Object - Points to an initialised object that you wish to draw.

RESULT
Returns ERR_OK if successful.

SEE ALSO
Kernel: Clear()
```

1.19 Kernel: FastRandom()

```
FUNCTION
Name:   FastRandom()
Short:    Generate a random number between 0 and <Range>.
Synopsis: LONG FastRandom(WORD Range [d1]);

DESCRIPTION
Creates a random number as quickly as possible. The routine uses one
divide to determine the range and will automatically change the random seed
value each time you call it. This routine has now been fully tested and
generates 100% patternless numbers.

Remember that all generated numbers fall BELOW the Range. Add 1 to your
range if you want this number included.

INPUTS
Range - A range between 1 and 32767. An invalid range of 0 will result in
        a division by zero error.

RESULT
A number greater or equal to 0, and less than Range.

SEE ALSO
Kernel: SlowRandom()
Demos:  demos/randomplot
```

1.20 Kernel: FindDPKTask()

```
FUNCTION
Name:     FindDPKTask()
Short:    Find the DPKTask structure for the current task.
Synopsis: *DPKTask FindDPKTask(void)

DESCRIPTION
This function will return the DPKTask structure for the task that called
it. The DPKTask structure is used for storing data that is specific to
your task - things like preference settings for example. Almost all of the
DPKTask fields are private and you you cannot write to this structure
unless you are a system module.

For the curious, it only takes 3 assembler instructions to grab the task
node, so there is no time wasted in calling this function.
```

```
RESULT
Pointer to the DPKTask structure.

SEE ALSO
Include: system/tasks.i
```

1.21 Kernel: FindSysObject()

```
FUNCTION
Name:     FindSysObject()
Short:    Finds a system object based on the ID.
Synopsis: *SysObject FindSysObject(LONG ID [d0], *SysObject [a0]);

DESCRIPTION
This  function begins finding a SysObject structure when given an ID and an
initial SysObject of NULL.  If a matching SysObject is found in the system,
it  will  be returned immediately.  Otherwise this function will go through
the  object  reference  list to see if the object is available on disk.  If
this  is  the  case,  then  the relevant module will be loaded and then the
SysObject will be returned.  If not, NULL is returned to indicate a failure
in the search.

The  first matching SysObject to be returned is always the master object of
that particular class.  If you want to find the child objects of the class,
call  FindSysObject()  again  and supply the previously returned SysObject.
The  next  matching  object  will  be found and returned, otherwise NULL is
returned  if  no more objects are left.  Note that if you are looking for a
specific  child or hidden object you will need to check the SysObject->Name
string.

NOTE
The only way to find hidden objects are to search on ID_HIDDEN.

INPUT
ID        - A system object ID as specificed in system/register.i.
SysObject - Last  received SysObject if continuing a search, otherwise NULL
            to start a new search.

RESULT
Pointer  to the SysObject that has been found as a result of the search, or
NULL if no matching objects were found.

SEE ALSO
Include: system/register.i
```

1.22 Kernel: Flush()

```
ACTION
Name:     Flush()
Short:    Flush buffered data from an object.
Synopsis: LONG Flush(APTR Object [a0])
```

```
DESCRIPTION
This  action  will  flush  all  buffered data from an object.  It is mostly
intended for file objects that may buffer data to speed up processing time.
As a result of flushing, any un-written data will be dumped to its physical
location, if this is practical for the object in question (eg files).

Note  that  any object using a buffering technique will automatically flush
its data when you Free() it.

INPUT
Object - Pointer to an initialised object.

RESULT
Returns ERR_OK on success.  All errors returned from Flush() are non-fatal,
so you may continue to use the object after failure.

SEE ALSO
Kernel: Reset()
```

1.23 Kernel: Free()

```
ACTION
Name:     Free()
Short:    Frees an object and any of its associated parts.
Synopsis: void Free(APTR [a0])

DESCRIPTION
This action will take any system object and free its resources.  It accepts
lists for multiple deallocations.

If there is no Free action for the object, then it will be assumed that the
object does not require freeing.  However, such an instance would be highly
unusual.

Once  the  object  has  been freed, the original pointer that you passed to
Free() immediately becomes invalid.  For this reason, any references to the
object  should  be  driven  to  NULL to prevent bugs from appearing in your
program.

INPUT
APTR - Pointer to one of the following:

         Object, ListV1, ListV2, TagList, ObjectList

SEE ALSO
Kernel: Get()
        Init()
```

1.24 Kernel: FreeMemBlock()

```
FUNCTION
Name:     FreeMemBlock()
Short:    Free a previously allocated mem block.
Synopsis: void FreeMemBlock(APTR MemBlock [d0])

DESCRIPTION
Frees  a  memory  area  allocated  by  AllocMemBlock(),  AllocVideoMem(),
AllocBlitMem(),  or AllocSoundMem().  If the mem header or tail is missing,
then  it  is assumed that something has written over the boundaries of your
memblock,  or  you  are  attempting  to  free  a  non-existant  allocation.
Normally  this  would  cause a complete system crash, but instead we simply
send a message to IceBreaker and leave the memory block in the system.

Bear  in  mind that it does pay to save your work and reset your machine if
such a message appears, as it indicates that important memory data may have
been destroyed.

NOTE
Never attempt to free the same MemBlock twice.

INPUT
MemBlock - Points  to  the  start  of a memblock.  If NULL, then no
     action will be taken (function exits).

SEE ALSO
Kernel: AllocMemBlock()
```

1.25 Kernel: Get()

```
FUNCTION
Name:     Get()
Short:    Gets the latest version of a specified object.
Synopsis: APTR Get(LONG ID [d0])

DESCRIPTION
This function will get the latest version of any object that you specify by
the  ID  argument.  That  is  of  course, if the object has been correctly
installed and is registered within the sytsem.

Screens, Pictures and Sounds are permanent and therefore always recognised,
while  something  like  a  CardSet is an extra and must have been installed
first  [Get()  will  find  such  objects  by  using  the reference files in
GMS:System/References/].

All  objects  going  through  Get() are tagged with a resource key.  Before
your program exits you will need to free the object with the Free() action.
If  you  want  your program to exit and leave certain objects in the system
(ie  for  other  programs  to use), then you can logical OR the ID argument
with GET_NOTRACK.

The  structure  will  return with empty fields, so you can fill them out to
suit your requirements.

This  function  is the only way you can legally obtain a system object.  If
```

```
you ever think of compiling objects directly inside your program, forget it
- your program will crash.

INPUT
ID - One of the ID's as specified in the system/register.i file.

RESULT
The  latest  version  of the specified object or NULL if failure (caused by
lack of memory or unrecognised ID).

SEE ALSO
Kernel:  Free()
         Init()
Include: system/register.i
```

1.26 Kernel: GetMemSize()

```
FUNCTION
Name:     GetMemSize()
Short:    Identifies the size of a given memory block.
Synopsis: LONG GetMemSize(APTR MemBlock [a0])

DESCRIPTION
This  function  will get the size of any memory block legally obtained from
AllocMemBlock().  Illegal pointers will result in a return of NULL.

INPUT
MemBlock - Pointer to the start of the memory block to be identified.

RESULT
The size of the memory block in bytes.

SEE ALSO
Kernel: GetMemType()
```

1.27 Kernel: GetMemType()

```
FUNCTION
Name:     GetMemType()
Short:    Identifies the type of memory in use by a particular memory block.
Synopsis: LONG GetMemType(APTR MemBlock [a0])

DESCRIPTION
This function will get the memory type of any memory block legally obtained
from AllocMemBlock().  Illegal pointers will result in a return of -1.

INPUT
MemBlock - Pointer to the start of the memory block to be identified.

RESULT
Type - Memory flags identifying the block (see AllocMemBlock()).
```

```
SEE ALSO
Kernel: GetMemSize()
```

1.28 Kernel: Hide()

```
ACTION
Name:     Hide()
Short:    Hides a displayed object from view.
Synopsis: void Hide(APTR Object [a0])

DESCRIPTION
If you have successfully displayed an object, then you can call Hide() to
take it off the display. If the object has not been displayed, then the
call is ignored.

INPUT
Object - Pointer to an initialised object that has been displayed.

SEE ALSO
Kernel: Display()
```

1.29 Kernel: Init()

```
ACTION
Name:     Init(), InitTags()
Short:    Initialises an object so that it is ready for active use.
Synopsis: APTR Init(APTR Data [a0], APTR Container [a1])
          APTR InitTags(APTR Container, LONG tag1type, ...)

DESCRIPTION
This function initialises any recognised system object. The container
argument is dependent on the type of object that you are initialising (eg
Bob requires a Screen or Bitmap) so it is not always necessary to supply
one.

If you provide a List or ObjectList then be aware that Init() will pass the
original container to ALL the objects in the list. For this reason all
objects must share some commonalities to the container (eg do not
initialise a Sound in a list of Bobs).

If the initialisation action fails, then Init() will look for any child
classes that can handle initialisation of the object. If a child class
succeeds it will gain full ownership of the object and the master will lose
it.

This action will return NULL on error, you can get very informative error
messages by using IceBreaker.

Note to Module Programmers
If your module fails to initialise the object, Init() will free the object
for you - do not attempt to free the object yourself.
```

```
INPUT
Data      - Pointer to an:  Object, TagList, ListV1, ListV2 or an ObjectList.
Container - Some objects need to be initialised to a "container" or parent
            object.  If this is the case, specify that object here.

RESULT
Pointer to the initialised object.

SEE ALSO
Kernel: Free()
        Get()
```

1.30 Kernel: InitDestruct()

```
FUNCTION
Name:     InitDestruct()
Short:    Initialise the task for use of SelfDestruct().
Synopsis: void InitDestruct(APTR DestructCode [a0], APTR DestructStack [a1])

DESCRIPTION
This is a special function that is called in the STARTDPK macro, gms.o
startup file and StartDPK program only.  You should never call this
function explicitly unless you are writing your own startup code.

InitDestruct() will prepare your task so that it may be destroyed by the
SelfDestruct() function.  DestructCode must point to the exit code for your
task.  The exit code must call CloseDPK() at some point if you are to free
your tasks resources.  DestructStack must point to the correct stack area
for your exit code, otherwise your task cannot return to the system
correctly.

To see an example of how this function works look at the STARTDPK macro in
file "dpkernel.i".

INPUTS
DestructCode  - Points to the code at which your task makes its exit.
DestructStack - Points to the stack that will be used for the exit.

SEE ALSO
Kernel: SelfDestruct()
```

1.31 Kernel: Load()

```
ACTION
Name:     Load()
Short:    Load a file and initialise it as a system object.
Synopsis: APTR Load(APTR Source [a0], LONG ID [d0])

DESCRIPTION
Loads in a file, finds its native object and then returns an object that
has been initialised and ready for use.
```

If you supply a special ID of ID_MEMBLOCK, Load() will allocate a MEM_DATA memory block, load all of the files contents into it, and then return the memory block pointer back to you. You will need to free this pointer with FreeMemBlock() when you are finished with it.

The file will be loaded in according to the object's preferred format, eg loading of Pictures will always result in the data being in video ram. If you require more power in the loading of a particular object, use the Init() function instead of Load().

NOTE
If this function cannot find an object that immediately recognises this structure, Load() will most probably open the file using the RawData object.

INPUT
Source - File name or Memory location pointer.
ID - Forces the type of object that you want to be returned. Use NULL if you want Load() to assess the file and return its appropriate object.

RESULT
Pointer to the initialised object or NULL if failure. Remember that RawData is used if no other object recognises the source file.

SEE ALSO
Kernel: Free()
Init()

1.32 Kernel: OpenModule()

FUNCTION
Name: OpenModule()
Short: Provides a quick way of opening a module.
Synopsis: *Module OpenModule(LONG ID [d0], BYTE *Name [a0])

DESCRIPTION

INPUT
ID - ID of the module to open (can be NULL if name is supplied).
Name - Name of the module to open (not required if ID is known).

RESULT
Pointer to an initialised Module structure or NULL if failure.

SEE ALSO
Object: Module
Kernel: Init()

1.33 Kernel: Query()

```
ACTION
Name:     Query()
Short:    Gets the latest information on a particular object.
Synopsis: LONG Query(APTR Object [a0])

DESCRIPTION
Calling the Query() action will update all fields in the given object to
reflect any changes since the last Query() or initialisation.

Examples of using the Query() action are to get information on picture
files (such as width, height, amount of colours) and obtaining consistently
changing data, such as joystick information.

INPUT
Object - Pointer to an object allocated from Get().

RESULT
ErrorCode - ERR_OK on success.
```

1.34 Kernel: Read()

```
ACTION
Name:     Read()
Short:    Read data from an object into a buffer.
Synopsis: LONG Read(APTR Object [a0], APTR Buffer [a1], LONG Length [d0]);

DESCRIPTION
This action will read the amount of bytes as determined by Length, from the
Object and into the given memory Buffer.  The read will start at the
position determined by the object's Byte Position field.  This field will
be incremented to BytePos+Length if the call succeeds.  Further calls to
Read() will therefore start from where you left off.

If the Length exceeds the total size of the data then this function will
only read as many bytes as there are left in the file.  You can always
compare the BytePos and Size fields to see how many bytes are left to read
in the object.

INPUT
Object - Pointer to an initialised Object.
Buffer - Pointer to a memory area in which the data will be written to.
Length - Amount of bytes to read from the object.

RESULT
Returns the total amount of data read into the buffer.  NULL can indicate
an error, or maybe there is no more data left to read.

SEE ALSO
Kernel: Write()
```

1.35 Kernel: RemSysEvent()

```
FUNCTION
Name:     RemSysEvent()
Short:    Removes an event tag from the system.
Synopsis: void RemSysEvent(*Event [a0])

DESCRIPTION
Not documented yet.

INPUT
Event -

SEE ALSO
Kernel: AddSysEvent()
```

1.36 Kernel: Reset()

```
ACTION
Name:     Reset()
Short:    Resets an object to a receptive state.
Synopsis: LONG Reset(APTR Object [a0])

DESCRIPTION
This action will reset an object to a state that is receptive to new
processes. Its behaviour depends greatly on the object at hand. Files
will reset their byte position to 0 for all further I/O operations, while
the Restore object dumps all current restore states and forgets about any
buffered data.

NOTE
Reset() is not the same as Flush(). Some objects will flush themselves
before performing the reset action, others may not. It depends on the
circumstances, but if you want the data flushed then use the Flush()
action.

INPUT
Object - Pointer to an initialised object.

RESULT
An error will be returned if the action could not execute. This can happen
if the object does not support or require the Reset() action.

SEE ALSO
Kernel: Flush()
```

1.37 Kernel: Save()

```
ACTION
Name:     Save()
Short:    Writes all of an object's information to a destination file.
Synopsis: LONG Save(APTR Object [a0], APTR Destination [a1])
```

```
DESCRIPTION
This  action will save an object's data to a file, in a data format that is
suitable  for  the  Load() action.  For example:  As a default, the Picture
object  saves  all pictures in IFF format, and it also supports the loading
of pictures in IFF format.

It will be possible for a child object to save a file in its own format (ie
instead  of  the  master  format).  This  way  the  user  can perform file
conversions,  such  as  load a picture as IFF and save it out as JPEG.  The
method  for  doing this is still under-way, but to simplify the process for
programmers,  the  user  will  probably  be able to choose preferred saving
types from GMSPrefs.

INPUT
Object      - Pointer to an initialised object.
Destination - Pointer to a Source/Destination structure, such as a
              FileName, MemPtr or File object.

RESULT
Returns ERR_OK on success.

SEE ALSO
Kernel: Load()
```

1.38 Kernel: SelfDestruct()

```
FUNCTION
Name:     SelfDestruct()
Short:    Destroys the task and frees resources.
Synopsis: void SelfDestruct(void)

DESCRIPTION
Destroys  the  task that called this function and then proceeds to free all
of  its  resources  according  to the resource nodes.  This is a completely
safe  and  effective  way  of  destroying  a  task,  and  can  be  used for
deconstructing a task when it has got into unrecoverable circumstances.

You  must  have called InitDestruct() before calling this function.  If you
are  programming  in  C  or assembler this initialisation is already in the
STARTDPK macro and gms.o startup file, so this does not concern you.

NOTE
This  function  will  not  return.  However if InitDestruct() has not been
called  then  the  function will not be able to do anything and will return
back to the task.

SEE ALSO
Kernel: InitDestruct()
```

1.39 Kernel: SlowRandom()

```
FUNCTION
Name:     SlowRandom()
Short:    Generate a random number between 0 and <Range>.
Synopsis: LONG SlowRandom(WORD Range [d1])

DESCRIPTION
Generates  a  very good random number in a relatively short amount of time.
This routine takes approximately two times longer than FastRandom(), but is
guaranteed of giving excellent random number sequences.

Remember  that  all  generated numbers fall BELOW the Range.  Add 1 to your
range if you want this number included.

INPUTS
Range - A range between 1 and 32767.

RESULT
A number greater or equal to 0, and less than Range.

SEE ALSO
Kernel: FastRandom()
Demos:  demos/randomplot
```

1.40 Kernel: Switch()

```
FUNCTION
Name:     Switch()
Short:    Stops the task that called this function.
Synopsis: void Switch(void);

DESCRIPTION
Switches  your task over to the next task in the queue.  This function will
not return until the user reactivates your task, so your tasks execution is
effectively  stopped.  Any secondary processes and interrupts that you have
spawned will continue to execute, so multi-tasking can still be effective.

If  the next task is screen-based, then your screen display will be removed
and  the  new  screen  will  be displayed.  If you have any secondary tasks
running,  then  take  note:   You must not allow them to use the
drawing/blitter  operations as your display memory may be temporarily moved
to  free  up  video  memory.  Blitting  to  an  invisible  display is also
considered  to  be  bad  practice  as  most GMS tasks require all available
blitter time.  We also ask you to refrain from using the audio functions as
the next task will probably be needing all available channels.

If  there  are  no  more  tasks  in the queue, then the screen display will
return  to  intuition.  GMS  supports  two  methods of screen switching to
intuition, Switch-To-Window and Switch-To- Screen.  The method used depends
on the setting in the GMSPrefs utility.

Switch-To-Window  drops out to workbench and places a window on the screen.
It  will  wait  until the close gadget is pressed, whereupon your game will
continue where it left off.
```

```
Switch-To-Screen  opens an intution screen and busy-waits until that screen
comes  to the front.  At that point the intuition screen will be closed and
your game will resume execution.

SEE ALSO
Kernel: AutoStop()
```

1.41 Kernel: TagInit()

```
FUNCTION
Name:     TagInit()
Short:    Initialise a structure according to a tag list.
Synopsis: LONG TagInit(APTR Structure [a0], APTR TagList [a1])

DESCRIPTION
This  function  is  intended  for  system modules but may be used by normal
programs if required.

It will process a standard tag list and store specified values in the given
structure,  which should be empty although this is not a pre-requisite.  It
is important that the tags themselves have been correctly defined using the
TBYTE, TWORD and TLONG flags.  Check the include files for examples.

This  function  has  some software based memory protection and will prevent
values from being written outside of the structure's memory area.  Detected
errors will be sent to the system debugger.

INPUTS
Structure - Pointer to allocated structure memory.
TagList   - Pointer to a standard tag list (see tags).

RESULT
Returns ERR_OK if successful.

SEE ALSO
Kernel: Get()
```

1.42 Kernel: TotalMem()

```
FUNCTION
Name:     TotalMem()
Short:    Gets the total amount of memory used by a task.
Synopsis: LONG TotalMem(*DPKTask [a0], LONG Flags [d0])

DESCRIPTION
Gets the total amount of memory currently in use by a Task.  This total can
be  calculated  from a specific type of memory, or a total of all memory in
use  if  -1  is specified.  If DPKTask is passed as NULL then this function
will sum up the total amount of memory used by all Tasks in the system.

INPUT
DPKTask - Pointer to a DPKTask object, or NULL to calculate from all tasks.
```

```
Flags   - The  memory  type  that  you  want  to  be calculated (MEM_VIDEO,
    MEM_SOUND,  MEM_DATA  or MEM_BLIT).  If you want a complete total
    of all memory in use, specify -1 here.

RESULT
The total amount of memory in use.
```

1.43 Kernel: WaitTime()

```
FUNCTION
Name:     WaitTime()
Short:    Wait for a specified amount of micro-seconds.
Synopsis: void WaitTime(LONG MicroSeconds [d0])

DESCRIPTION
Waits  for  a  specified amount of micro-seconds.  During this time it will
reduce the task priority and make regular calls to AutoStop() for you.

INPUT
MicroSeconds - Amount of micro-seconds to wait for (100 = 1 Second).
```

1.44 Kernel: Write()

```
ACTION
Name:     Write()
Short:    Writes a given amount of bytes to an object's data space.
Synopsis: LONG Write(APTR Object [a0], APTR Buffer [a1], LONG Length [d0])

DESCRIPTION
This  action will write the amount of bytes as determined by 'Length', from
the  memory  buffer and into the Object's data space.  The write will start
at  the position determined by the object's BytePos field.  This field will
be incremented to BytePos+Length if the call succeeds.

If  the  BytePos+Length exceeds the total size of the file, then the object
may  increase  its  data  space  to  cope  with writing out the rest of the
buffer.   This will always happen with Files, but memory based objects will
rarely be able to do this.

INPUTS
Object - Pointer to an initialised object.
Buffer - Pointer to data that will be written to the object.
Length - Amount of bytes to write.

RESULT
Returns the amount of bytes written.  NULL indicates an error.

SEE ALSO
Kernel: Read()
```

1.45 Kernel: ()

```
FUNCTION
Name:       ()
Short:
Synopsis:

DESCRIPTION

NOTE

INPUT

RESULT

SEE ALSO
```