intel® Media Libraries Programming Guide

Version 1.32

1 Introduction

The int**e**nt® media libraries are a compact toolkit for providing powerful multimedia capabilities. It allows manufacturers to add value to their embedded products, while retaining their own unique branding. It is comprised of the following concepts:

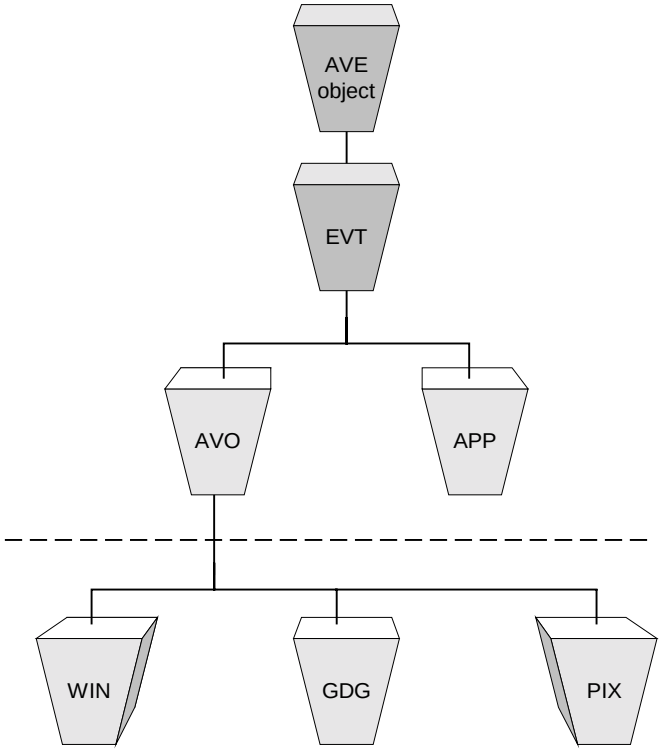
- *AVE device driver (dev/ave)*
- *AVE Toolkit (ave/toolkit/*)*
- *Audio Visual Objects (ave/avo)*
- *Event Target Object (ave/ave/evt/)*

The int**e**nt libraries can be described as consisting of three layers. The first of these, is the graphics drawing layer, with functions for drawing lines, circles and other drawing primitives, in plain, tiled and xor drawing modes. Secondly, there is a compositing layer that allows the hierarchical arrangement of multiple components (whether opaque, translucent, or transparent). Finally, a gadget layer consists of a number of toolkits that permit the creation of graphical user interface components that may be customised according to the user's requirements.

The term 'AVE' (an acronym for audio video environment) is often used interchangeably with the int**e**nt media libraries description in this document.

2 Conceptual Overview

2.1 Object Infrastructure



*Conceptual Overview of int**e**nt media libraries*

The most basic concept within the int**e**nt media libraries is the AVE object. Essentially, these are responsible for object creation and destruction, object memory allocation, reference counting and cloning. From this starting point, event target objects (EVTs) may be created, and from there applications objects (APPs) and audio visual objects (AVOs).

2.2 AVE Objects

int**e**nt's media object infrastructure consists of a standardised way to create, destroy, initialise, deinitialise, allocate and free object instances. All objects within the AVE inherit from this base class. The purpose of this new framework is to provide a simple, flexible, consistent, lightweight object creation mechanism for the int**e**nt media libraries and other sub-systems within int**e**nt. The AVE methods are briefly described below:

- *_size*

This ccallable method simply returns the size of an object of this class.

- *_new*

This ccallable method has the responsibility of allocating an area of memory for an object that is at least the size that *_size* returns and referencing the class tool.

- *_delete*

This method is the partner for *_new*. It is responsible for derefencing the class tool and freeing the object instance back to wherever the *_new* method allocated it from.

- *_init*

This method is paired with *_deinit*. This method is responsible for initialising the object instance returned by the *_new* method. It is often overridden, as initialisation is class specific.

- *_deinit*

This method is the partner for *_init*. It should free any resources that the *_init* method opened or created.

_open - ccalls _new then *_init.*

This method is responsible for creating a complete allocated and initialised object instance of the class. Whenever there is an *_init* method in a class there will be an _open method to match.

- *_close*

This calls *_deinit* then *_delete.* There is in general no need to ever have a _close method in a subclass.

- *_clone*

This method returns a new instance of this type of object that contains the same state as the input object.

- *_ref*

This method simply increments the objects reference count.

- *_deref*

This method decrements the objects reference count and if the count is now equal to 0 it calls the *_close* method.

2.3 Event Target Objects (EVTs)

Essentially, an EVT is an object capable of receiving events (as opposed to the Java language, where events are objects) but it is not an AVO, which serves as an encapsulation of a two dimensional interactive element. EVT objects inherit from the base class, and provide such functionality as a unique object reference number and the ability to be validated as an event target.

For information on the specific methods available to event target objects, please see *ave/evt/api.html.*

2.4 Application Objects (APPs)

The application object essentially acts as a container for all other EVTs and inherit from the EVT baseclass. The application object is fundamentally an AVE application. AVE applications must create at least one application object in order to use AVOs (see below). Please remember that the AVE device dispatcher mails the parent APP of the target AVO. Therefore, there must be an application object to which AVOs can be added or removed under program control; in order to add AVO objects to the screen they are added to the APP object. The APP object also receives events from the AVE device.

To display and activate AVOs they must be linked into the AVE. This is done by adding them to the application object returned by the 'open' call on the AVE device. This application object acts as a gateway for event messages to be passed to the AVOs, and represents the AVE's screen for the purposes of adding and removing AVOs.'

For information on the specific methods available to application objects, please see *ave/app/api.html.*

2.5 Audio Visual Objects (AVOs)

The fundamental building block of the intent media libraries is the Audio Visual Object (AVO). intent has a class hierarchy, the base class being *ave/avo/class.asm*. This is the class from which all AVOs inherit. The AVOs are:

- Pixelmap (*ave/avo/pix*)
- Image (*ave/avo/img*)
- Window (*ave/avo/win*)
- Gadget (*ave/avo/gdg*)
- Soundmap (*ave/avo/snd)*

These objects may be created, destroyed and organised by the application programmer, as with any other object in the system. Additionally the AVOs can be hierarchically organised depending on the needs of the application. AVOs can contain other AVOs. Any AVO can contain any other AVO.

AVOs can be grouped together, should this be required. For example a window AVO may contain several other AVOs so that when the window is moved the other AVOs move too. As an example, a typical hierarchy would be an application object containing one or more windows. Each window contains a mixture of gadgets, pixelmaps and soundmaps. The gadgets may in turn contain pixelmaps or soundmaps, or indeed other gadgets. It is important to note that this hierarchy is entirely flexible and the example given purely arbitrary. The AVO hierarchy can be thought of as a tree, with various parent AVOs having child AVOs added to them.

The differences between the AVOs are discussed in more detail below.

Each AVO provides its own xyz co-ordinate system for child AVOs, and clips children to its bounds. As well as acting as containers AVOs have other characteristics; they have positional data (x and y co-ordinates) and depth (z co-ordinate), they can be placed, moved, resized, hidden and clipped. In addition, each AVO possesses an index, which orders it within a z plane (0 being foremost). Every AVO inherits its own event handling method code (unless it explicitly overrides this). As mentioned an AVO can be added to another AVO, but it can also easily be removed from that AVO.

Every AVO has an associated cursor type; this determines the cursor image that is displayed when the mouse (or other pointer) is over the AVO. By default, the cursor type is dynamically inherited from the AVO's parent tree; setting a different cursor type overrides inheritance.

There are a number of methods, which apply to all AVOs, with each AVO type then having its own additional methods. Each AVO has a system for handling tokens inherited from the AVO base class. This is discussed in more detail later.

The base class code for all AVOs is in *ave/avo/class.asm*. All the methods of this class are documented in *ave/avo/api.html*. This class is subclassed to produce the standard AVOs listed above.

3 Standard AVOs

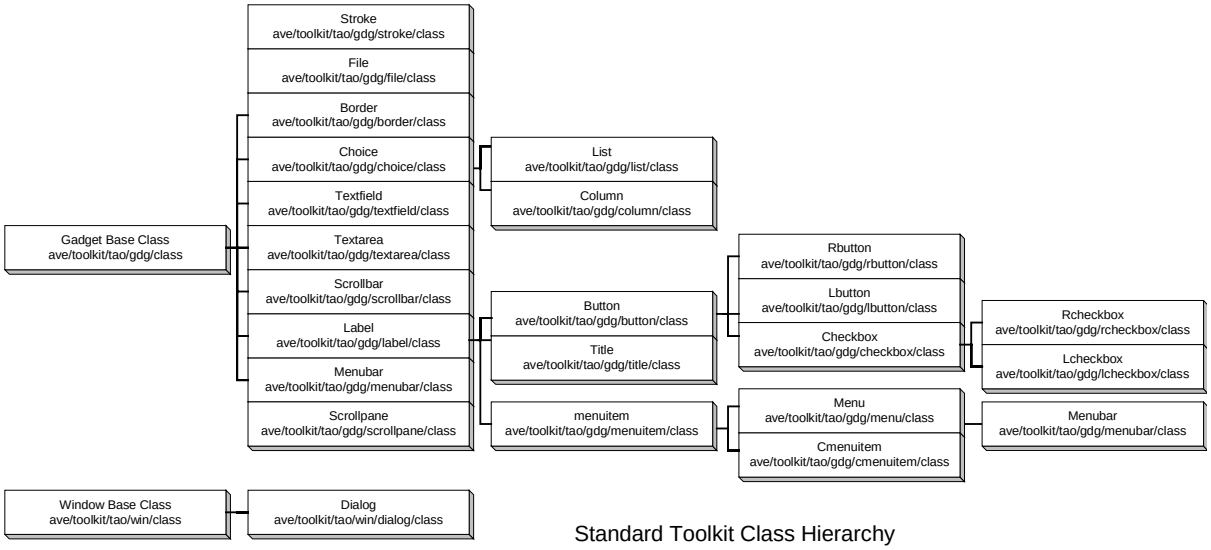
As mentioned before a number of standard AVOs are defined and these inherit from the AVO base class. The standard AVOs are:

- Pixelmap
- Window
- Image
- Gadget
- Soundmap

It is possible for user defined AVOs to be created simply by subclassing any of the above AVOs. This could be done to create a new type of gadget or window for example.

3.1 Gadget AVOs

The gadget AVO base class is defined in *ave/avo/gdg/class.asm* and the API for these is documented within *ave/avo/gdg/api.html.* The API for the gadget baseclass is defined in *ave/avo/gdg/api.html*. From this gadget baseclass all other gadget AVOs are created by inheritance. The subclass gadgets will define their own methods, but inherit all the methods of the base class. The gadgets created by subclassing the base gadget class are collected into 'toolkits'. A standard toolkit is provided which may be supplemented by additional toolkits to give a different look and feel. intent provides techniques for easily switching between toolkits to generate a different look and feel for an application's user interface. This concept of a toolkit also extends to windows. Note that each subclass will have its own API defined, which is documented in the api.html file found in the same folder as the subclass. The following diagram further illustrates the toolkit concept:



Standard Toolkit Class Hierarchy

The gadget subclasses for the standard toolkit (*ave/toolkit/tao/gdg/*)* include:

- border
- button
- checkbox
- choice
- cmenuitem
- column
- label
- lbutton
- lcheckbox
- list
- menu
- menubar

- menuitem
- rbutton
- rcheckbox
- scrollbar
- textarea
- textfield
- title
- scrollpane
- Progress

3.2 Window AVOs

The window AVO baseclass is located in *ave/toolkit/tao/win/class.asm*. Window AVO standard toolkit (*ave/toolkit/tao/win/*)* subclasses include dialog. This implements the titlebar, the border, and other controls normally associated with windows (e.g. buttons to minimise or close the window). The API for these is contained within *ave/avo/win/api.html.*

3.3 Pixelmap AVOs

Pixelmaps are memory areas that contain graphical data. They can be copied to the screen, in which case their contents becomes visible, or they may exist only in memory. Pixelmaps may be created, destroyed, moved, displayed, or manipulated by one of the many methods defined in *ave/avo/pix/api.html*. As pixelmaps are AVOs they could be added to other AVOs such as a window or gadget. Alternatively, they may just be displayed directly and not added to any other AVO. The baseclass pixelmap AVO is defined in *ave/avo/pix*. In addition to this, the standard pixelmap AVO is subclassed to provide the following pixelmap types:

- 1bit
- 4bit
- 8bit
- 12bit
- 15bit
- 16bit
- 24bit
- 32bit
- 16 bit YBR
- 32 bit YBR

A wide range of pixelmaps are provided to support displays with different capabilities. The API for these is contained within *ave/avo/pix/api.html.*

3.4 Image AVOs

These are procedural drawing tools, whose redraw method uses the low level drawing methods to describe their object's appearance. Standard subclaases include

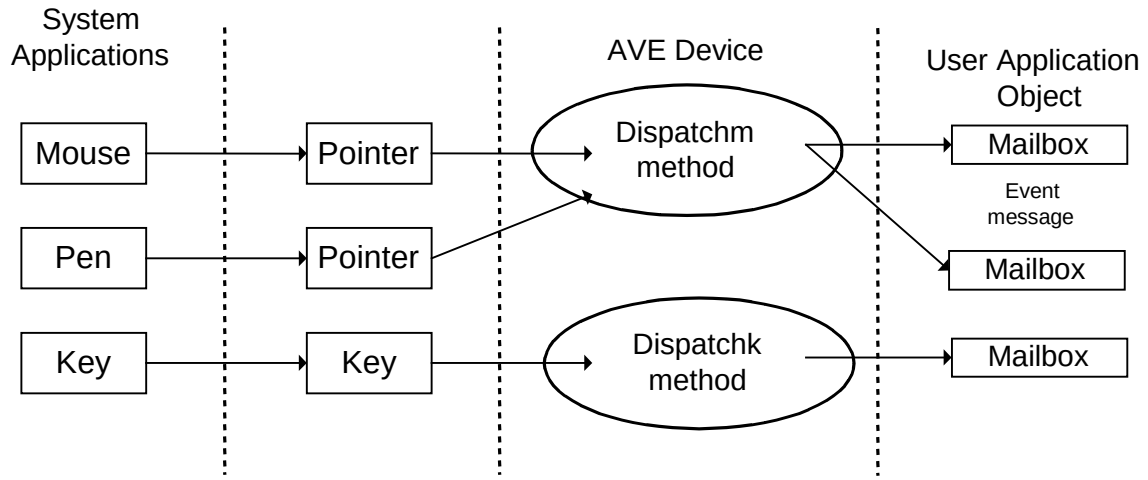
- RGB Mask
- Greyscale
- Verical Gradient
- Half Transparency
- False Colour
- Sepia Tone
-

The API for these is contained within *ave/avo/img/api.html.*

3.5 The AVE Device

The primary function of the AVE device is to process incoming events and to pass these on to the correct application object. All AVE applications lookup and open the AVE device in order to create an APP object, without which an AVE application cannot display content on the screen. When the AVE device is first opened, creating a screen through the GRF device driver (see chapter five) and running all command lines within the *dev/ave/auto.scr* file. This file typically contains command lines to start system applications (see chapter four). Height, width, pixel depth and font pitch can all be specified. When the last application is closed the AVE device will close the screen object it acquired from the GRF device.

The AVE device contains methods for the creation and destruction of APP objects and processing incoming events, namely *dispatchm* and *dispatchk*. The *dispatchm* method handles mouse and pen input events and *dispatchk* handles incoming keyboard events. The appropriate dispatch method is called directly by system applications handling input from the related device driver. This is illustrated in the diagram below:



The generic operation of a dispatch method is:

1) Determine the target object by current token owner or by collision detection. The application object can then be determined by performing a *findparent* method on the target object.

2) Create a standard format message including all necessary data, such as x and y co-ordinates, timestamp, target object, button status or key status etc.

3) Send the message to the mailbox of the application object that was determined in step 1) above.

3.6 Tokens and Token Allocation

Tokens are provided as a way of handling shared resources like the keyboard and pointer. The AVE device is also responsible for managing token allocation to AVOs. Note that only AVOs can have ownership of a token, not EVTs. Currently there are four token types defined:

DragToken	Gained by an AVO that is being dragged (this is not for user access)
Overtoken	Gained by an AVO that has the mouse cursor over it (this is not for user access)
Keytoken	Claimed by an AVO that is requesting keyboard input
PopupToken	Held by any temporary popup gadget

Tokens are used to control access to non-shareable devices. For example only one AVO should have access to the keyboard at any one time and so would need to obtain the keyboard token.

The AVE device maintains a list of which AVO has which token. User applications can control token handling by using the following methods:

settoken	Set token's target AVO
gettoken	Gets token's target AVO
clrtoken	Clears any token ownership of the target AVO
clrtokens	Clears all token ownership

When the *settoken* method is called the previous token owner is notified with a *losttoken* message, while the new owner is notified with a *gaintoken* message. For more information on these please see *ave/avo/api.html.*

3.7 AVE Device Calls

The following functions associated with the AVE device may be of use to an application:

- **Tools *dev/ave/tao/lock* and *dev/ave/tao/unlock***

The AVE lock can be used to prevent other processes from affecting the state, for example by issuing an update call. These tools should be used with care, and only around short sections of code.
See *demo/ave/boing*.

- **Method *opentoolkit***

This call returns a toolkit instance used to create gadgets.

- **Method *closetoolkit***

This closes a previously opened toolkit.

- **Methods *allocevent* and *freeevent***

These calls allocate and free a standard event message.

- **Method *getscreen***

This call returns information such as screen size and mode.

3.8 Scripts and the System Menu

int**e**nt script files list AVE applications with optional parameters. They may contain comments preceded by a hash (#), and use the '.scr' file extension. The system menu, *dev/ave/dsk/runapp*, is invoked by clicking on the int**e**nt backdrop. This enumerates the directory tree rooted in */app/start/* each script file found becomes a menu item, while subdirectories become submenus. Thus to add an option to the system menu, create a script file to invoke an application at the appropriate place in the /app/start/ directory tree (the script's filename, minus the extension, becomes the label).

4 System Applications

The AVE device driver is supported by a number of stand-alone additional processes. These applications, referred to as 'system applications,' handle communications with low level input device drivers and perform tasks such as launching the intent media libraries and providing a tiled backdrop for the main intent desktop. These applications are typically located within *dev/ave/dsk/* while command lines from them are contained within the *dev/ave/auto.scr* file. Currently the following system applications are available:

Input device system applications:

- Pointer
- Keyboard

The pointer provides the link to the mouse or pen driver supplying user input. The keyboard reads from the keyboard driver and dispatches the results through the AVE device *dispatchk* method.

Utility system applications:

- Tiles
- Eterm (Graphical shell – see *dev/ave/dsk/eterm.html* for more information)
- Launch (start desktop) and Runapp

Of these, tiles provides a tiled background that can be set to execute a specific command line script whenever the pointer button is clicked on the screen backdrop; at present this starts the runapp applications. Runapp serves to create an application launch menu. Standard entries within the menu include intent's graphical shell, eterm. Launch allows intent to be launched without having to run a user application. Command lines options for this allow the overriding of the AVE command lines parameters for width, height, pixel depth etc.

Rather than 'hard-wiring' input device handling and utility code into the main AVE device, a more layered approach was taken. With this concept the intent media libraries do not need to know the details of devices that might be connected to it. The low-level details of the device are handled by the appropriate device driver, whilst the system application acts as an interface between that driver and the AVE device. The system application understands the information coming from the device driver and can therefore use the correct *dispatch* method to call on the AVE device. For example, the pointer system application knows how to use the pointer device driver; it knows that that device returns absolute co-ordinates and it knows what information to get from the device driver. It then uses this information to call the *dispatchm* method of the AVE device.

The big advantage of this device concept is that the intent media libraries can quickly and easily be made to support new devices such as pens, mice or joypads.

All that is required is a new system application and this is relatively easy to write (this of course assumes that device drivers are available for the devices in question).

Generally, two types of input system application are made available:

- Synchronous
- Asynchronous

Of these two, the asynchronous type is recommended for most cases. The synchronous system applications simply poll the input device driver and if any change in state or new data becomes available this is minimally processed before calling the AVE dispatcher. The asynchronous system applications use an asynchronous read method (*reada*) on the input device driver. The *reada* method does not cause the calling application to block. This is based on a callback technique, that is, when

the device operation completes or generates some other activity a callback procedure is executed. The callback procedure then invokes the AVE dispatcher. The callback procedure itself is set up in the system application.

Another feature of the system application architecture is that it is possible to run several system applications feeding into the same dispatch method. For example, it would be possible to have a joystick and pointer all generating input data. The system applications handle this input and then invoke the same *dispatch* method. From the user application's perspective there is only one logical input device, although input may physically come from more than one device. The following diagram illustrates this:

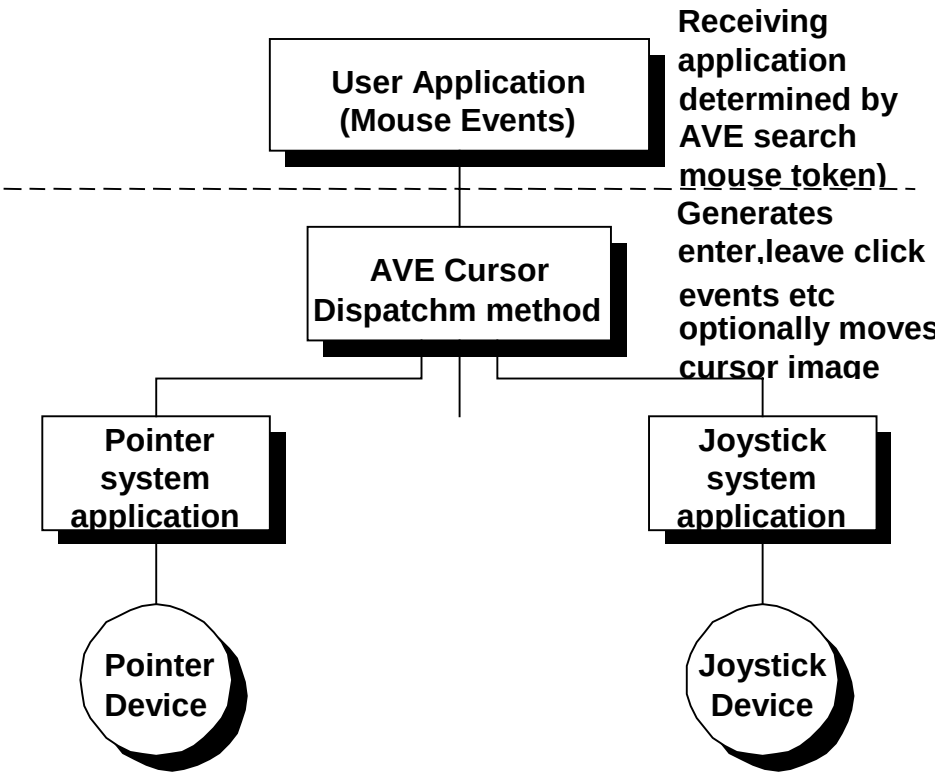


Diagram1. System applications activating the same dispatchm method on the AVE device.

The system applications can be found in the *dev/ave/dsk* directory.

4.1 Event Handling

As we have seen, input events from the system applications are further processed by the AVE device. An event format message is created and mailed to the mailbox of the parent application EVT of the target AVO. This section discusses the EVT event handling scheme. It is important to note at the outset that this scheme is defined in the EVT base class and is therefore inherited by all EVTs and AVOs.

The EVT event handling scheme uses a three layered architecture.

Once an event comes into an EVT. it can be processed by up to three layers of methods. Of these, the first two layers are applicable to event target objects, and the third is applicable to audio visual objects. This last layer is not available in cases of inheritance from an EVT, as the AVO object overrides the second level methods. These are shown in the table below:

[EVT]	[EVT]	[AVO]
1st level event handler method	2nd level event handler methods	3rd level event handler methods
event	systemevent	-
	tokenevent	gainevent loseevent
	mouseevent	buttondownevent buttonupevent buttonheldevent trackingevent enterevent leaveevent
	keyevent	keyupevent keydownevent keycharevent inputevent
	commsevent	dropevent
	userevent	-

The first level event handler has a switch statement to detect the type of event and then call the correct second level handler method. The second level handler methods also have a switch statement to detect the type of event and call the appropriate third level event handler method.

By default the third level event handler methods simply return. They should therefore be overridden in order to perform processing of events. It is also possible to override the event handler methods at the first or second level should this be required. For example, if all mouse events must be processed then it may be convenient to override the second level mouse event handler *mouseevent.*

Furthermore it is also possible to process events without using any of the above event handler methods. To do this a call to *getevent* is made and then the event type determination and handling code is performed within the event loop of the user application. The user application could also call the first level event handler method ‘event’ to ensure that other EVTs that need to process this event can do so.

4.2 Handling Events

Events are received by calling the application's *getevent* method, with an optional timeout. This call sleeps until a (valid) event is received, then returns the message, target and type of event. The application can either handle the event itself, or pass it to the target. Either way, the event should then be freed by calling the AVE *freeevent* method (but note that a *getevent* timeout returns an event pointer of 0, which must not be freed).

4.3 Modality

Mouse and key events are grouped as input events. Within an application, an AVO tree can be set to be the modal root, in which case any input events targeted at AVOs that are outside the tree are silently ignored. The modal root is normally the application object, in which case all events are accepted.

- *ncall app,setowner,(app,root_avo:-)* - sets the modal root to root_avo
- *ncall app,setowner,(app,app:-)* - resets (clears) the modal root

4.4 Event Message Format

intent is a system based on message passing. The programmer must be aware of the standard message structure in order for the application to extract necessary information.

Additional information may be added, as defined by the event type. The relevant api.html documents document the precise types of information. The message structure is contained within the file *ave/evt/class.inc*

4.5 Event Linking

The intent media libraries allow the creation of links to other application objects. Toolkit objects have event channels that can be listened to by APP or AVO objects. This system allows a number of target objects to be notified should certain activity or state changes occur within the source object. When the source EVT generates a state change or action an event is posted into the event handling system, so that all objects 'listening' into that channel will receive that event. The target EVT's main event handling loop will be written to handle the events of interest. This normally involves detecting the event message type and extracting the required information from the message.

Each EVT typically has a number of channels that can be linked. These channels can be linked to any number of other EVTs. Typically EVTs have two channels; channel 0 and channel 1.

Channel 0 represents a change state event. This means that if the state of the EVT changed, an event message is sent to the target EVT or EVTs linked to this channel. The target EVTs then decode the message and process the information. Usually the state flag information is appended to the end of the user event message sent. The actual state flags used depend on the type of EVT (see documentation for the source EVT).

Channel 1 represents an action event. When the EVT is 'activated', for example by clicking, an event message is sent to all target EVTs linked to channel 1.

Other channels may be defined depending on the AVO.

To manage event linking three methods of the AVO base class are provided, namely:

- *addlink*
- *postlink*
- *sublink*

Addlink allows a target EVT to listen to a particular channel and can be called any number of times to add more EVTs to a channel.

Postlink allows a user defined event message to be sent to EVTs linked to a specified channel under program control. This is the method that is called by the source AVO to post user event messages to all target EVTs linked to its channels.

Sublink removes a link between a channel and a target EVT.

Each user defined event link set up by the use of *addlink* needs an event type identifier. This is defined as EV_USER+xx, where xx is any convenient number. Typically in the event handling loop of an EVT a switch statement analyses the event message types and processes the types required.
This is illustrated by the following code snippet :

```
; event handler loop
...
            switch
                  whencase i0 = EV_BUTTONDOWN
                  whencase i0 = EV_TRACKING
                  whencase i0 = EV_USER+00
                  whencase i0 = EV_USER+01
                  ; whencase other events as required
                  otherwise
                        break
                  case i0 = EV_BUTTONDOWN
                        ; process this event type
```

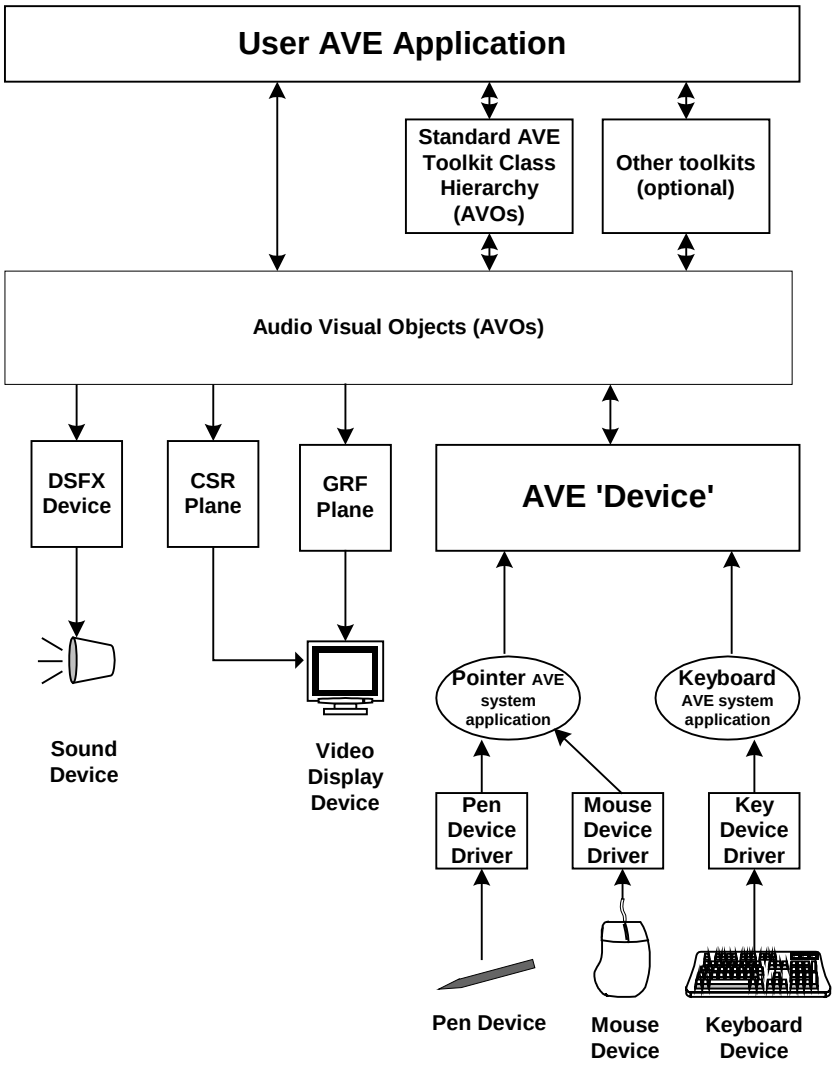
```
                        break

                case i0 = EV_USER+00
                        ; process this event type
                        break

                case i0 = EV_USER+01
                       ; process this event type
                        break

                case  i0 = EV_TRACKING
                        ; process this event type
                        break
            endswitch
...
```

It is possible to supply a system event type identifier to *postlink*, instead of EV_USER+XX. A typical example of this is in linking a dialog window's DIALOG_ACTION channel to an application using EV_QUIT. This means the application will close both because of a system request and a click on the dialog's close button (see *demo/ave/buttons.asm*).

5 Planes and DSFX Drivers

As we have seen, the intent system applications utilise standard device drivers for input from various devices such as mouse, keyboard, pen and joystick. In order to perform graphical, video, and cursor output the intent media libraries call on the services of the planes sub-system. To perform sound output the DSFX (digital sound effects) sub-system is used.

The inter-relationship between these components is shown in the following diagram:



5.1 The Planes Device Driver

Planes drivers allow the control of display planes such as graphics, video and cursors. As such, graphics drivers have been superseded by graphic plane objects, managed by planes drivers. However, extant graphics drivers written to the previous GRF API will continue to function once reassembled.

Essentially, the plane device driver is responsible for providing access to plane objects. The co-ordinate system for a plane's position priority and size is specified in 16.16 signed fixed-point values. The x/y position and priority can be the full range given by this number format. The visible display

area is a cube in plane co-ordinate space, extending from top-left-front (0,0,0) to bottom-right-back (1,1,1).

The plane objects controlled by the plane device driver are typically pixel, cursor or video subclasses of the plane object. Each plane object is responsible for controlling one or more displayable entities; for example a pixel plane object supports the allocation and display of drawing surfaces (the AVE's 'screen pixelmap').

In the case of a cursor plane object, there are a number of predefined cursor image entities, of which at most one may be visible at any time. The position of the cursor on the display is controlled by the position of this plane.

A plane mixer is responsible for selecting planes associated with a particular output. The default behaviour is for all active plane to be included in all outputs. The model assumes that each mixer is associated with a single output encoder.

5.1.1 The Cursor

The cursor manager class provides an interface between the AVE and hardware or software cursor implementations, and defines a fixed minimum set of cursor types (although some of these types may share images). By default, intent has no cursor manager set; the '-c' command line parameter sets the optional cursor manager class. This can be set using the sysbuild utility by specifying the option -DAVE_CURSOR_CLASS=classname".

If the cursor manager is controlling a cursor plane, then the cursor image may be moved anywhere within the plane coordinate space, and is not necessarily restricted to the bounds of the AVE screen pixelmap.

The displayed cursor image is selected automatically by the AVE *dispatchm* method, or by setting the cursor type on an AVO if that AVO currently 'owns' the cursor.

5.2 The DSFX Device Driver

A digital sound effects device driver allows digitally sampled sounds in linear PCM format to be played. The superclass for all digital sound effects device drivers is *dev/dsfx/class*, which provides basic functionality.

DSFX drivers are asynchronous and support both sample transfer notification for efficient streaming and play position notification for synchronisation. The API supports the concepts of voices. A voice relates to sound output hardware and drivers normally support as many voices as are made available by the underlying hardware. A DSFX driver is expected to support a minimum of one voice.

A session consists of a number of actions on voices. Its purpose is to allow the synchronisation of actions over a number of voices and to identify data queued for playing.

A voice/session combination supports the concept of data and control command queues. A data queue consists of a sequence of sample play requests that the driver is required to stream. A control queue consists of a single control command from the set of commands.

The exec call is used to action commands queued on voices associated with a particular session.

Drivers generally make no attempt to emulate functionality that is not supported in hardware. Minimum expected functionality from a driver, where it exists, is detailed in the API.

The provision and use of a specific driver memory object is optional and intended to be used as a performance optimisation mechanism. Typically, this may be used to allow applications to cache frequently used samples and drivers to ensure that buffer alignment requirements are met.

6 AVE Programming

6.1 intent Program Structure

Most intent applications have a similar structure. As a minimum, a typical intent program will need to perform the following:

```
option processing (use lib/opts)
lookup ave device (service)
open app object
open toolkit factory
create all parts (objects) needed by app
assemble parts
add main object(s) to app object

//start event loop
repeat
   wait for an event (optionally with timeout) using getevent
   call event method of the target object to pass event handling down chain
   process application specific events
   freeevent
until quit event (EV_QUIT)
//end event loop

// clean up
call app closeall method to make sure all objects are destroyed
close toolkit factory
close app object
exit
```

This program is discussed further in the following sections.

6.1.1 Device Lookup

Before calling any methods the object pointer must first be obtained. As with other device drivers this is done by looking up the AVE device in the mount table by calling *sys/kn/dev/lookup*. This call returns an object pointer (if the lookup is successful), that can then be used to call intent methods.

6.1.2 Opening an Application Object

Having obtained an object pointer the next thing to do is create an application object. This is achieved by calling the open method. The open method returns a handle, which in the case of the AVE is a reference to an application object. As with many other device drivers the AVE device supports multiple opens as it manages multiple resources and each returned open handle points to a "resource," in this case an intent application object.

The first call to the open method carries out some basic initialisation of the AVE. This includes:

1) Opening a screen via the planes device driver.
2) Processing the *dev/ave/auto.scr* which starts background programs and systems applications such as tiles, pointer and keyboard event handler.

6.1.3 Opening a Toolkit Factory

The toolkit factory provides a layer of abstraction to allow easy manipulation of the look and feel of an interface. A standard toolkit is provided, but third parties can produce additional toolkits with different styles of gadgets e.g. different shaped buttons, round windows etc. Using the factory concept an application program can be given a completely different look and feel by simply opening a new toolkit at the start of the program. No other code in the application needs to be changed.

A good example of this is toolkit factory's *createbutton* method. As the title implies, *Createbutton* will create a new button, but the look and feel of that button will be determined by the actual toolkit opened. The current toolkit can be set by calling the AVE device settoolkit method, and is set by default through a AVE command line option. Note that each toolkit implements its own gadget properties, which are specific to that toolkit. This allows the user to change the look of the toolkit without having to assemble the toolkit from source. For example, the provided Tao toolkit uses the file located at *ave/tooolkit/tao/default.prp.* The API is described within *ave/toolkit/api.html.*

6.1.4 Adding Objects to the Application Object

Once the required application specific objects have been created, they must be added to the application object.

The hierarchy of created AVOs can be added to the application by adding the root container object. For example, if a window has been created and the window contains various other gadgets that have been added to it, then we simply need to add the window to the application object. Of course, multiple objects can also be added directly to the application object. Update is then called on the window to draw it. If we added multiple objects (not contained in a window object) then we must call their update methods too.

6.1.5 Event Loop

This is the main "engine" of an AVE application. The basic event handling loop idea is the same as for other event driven systems, such as Windows®.

The first requirement is to get an event. For efficiency (particularly pertinent to portable devices) getting an event is done by sleeping (descheduling the process) until an event arrives or a timeout is reached. If an AVE event occurs the application process wakes and the *getevent* method returns with the target EVT (the EVT the event is for), the event message and the type of event. Although the type of event is part of the event message it is returned as a separate parameter as a convenience to the programmer.

The next step is to process the event. Each AVO in the system already has built in event handling method code. For example, the Lbutton AVO has implementation within a method called *buttondownevent*, to handle the particular case when it is clicked. These event methods are called through a three level hierarchy described in section 3.1. Usually an application will make use of this built in event handling code by calling the event method to pass the event down the event handling chain. It can do any further event processing required once the default processing has completed. In a simple case this would include handling a quit event message (EV_QUIT). Once default processing has been completed the event message resources should be deallocated by calling the freeevent method.

6.1.6 Cleanup

This is primarily a question of "closing" (with a call to the close method) everything that was "opened" (with the open method).

6.2 Code Example

The following simple example provides an idea of how to put these concepts into practice:

```
. include 'tao'
.include 'ave/toolkit/toolkit'
.include 'ave/toolkit/rsc/rsc'
.include 'ave/avo/pix/class'

DIALOG_SIZE=256
BLENDINC=512/DIALOG_SIZE

structure
size AVEOBS_START
pointer GP_DIALOG
pointer GP_LAYOUT
size AVEOBS_END
pointer GP_PIX
pointer GP_NULL
pointer GP_APP
pointer GP_APPNAME
pointer GP_DIALOGBORDER
pointer GP_DIALOGTITLE
pointer GP_DIALOGCNT
int32 GP_DIALOGFLAGS
int32 GP_PIXTYPE
int32 GP_PIXWIDTH
int32 GP_LAYOUTFLAGS
int32 GP_LAYOUTPADDING
size GP_SIZE

tool 'demo/ave/blend',VP,F_MAIN,16384,GP_SIZE

      ent -:-

      defbegin 0
      defp ave,app,tkit,dialog,avo,msg,table
      defi evt,xy,w,h,col

      ;clear globals, init args
      qcall lib/memseti,(gp,0,GP_SIZE:p~)
      qcall lib/argcargv,(-:p~,i~)

      ;find the ave
      __string '/device/ave/Blend'
      qcall sys/kn/dev/lookup,(__string_param.p:ave,app)
      ifnoterrno ave,true
            ;open ave
            ncall ave,open,(ave,app,0,0:app)
            ifnoterrno app,true
                  ;save app
                  cpy app,[gp+GP_APP]

                  ;open toolkit
                  ncall ave,opentoolkit,(ave,app:tkit)
                  if tkit!=0,true
                        ;build application
```

```
                        ifnoterrno table
                              ;set alphamode and clear opaque
                              cpy [gp+GP_PIX],avo
                              ncall
avo,setflags,(avo,FAVO_ALPHA,FAVO_ALPHA|FAVO_OPAQUE:-)

                              ;draw alpha blended circles
                              clr xy
                              cpy DIALOG_SIZE,w
                              cpy RGBWHITE,col
                              ncall avo,fbox,(avo,xy,xy,col,w,w:-)
                              repeat
                                    ncall avo,foval,(avo,xy,xy,col,w,w:-)
                                    add BLENDINC<<24,col
                                    inc xy
                                    sub 2,w
                              until w<=0

                              ;add to app, update dialog
                              cpy [gp+GP_DIALOG],dialog
                              ncall app,addavo,(app,0.p,dialog,0:-)
                              ncall dialog,update,(dialog:-)

                              ;event loop
                              repeat
                                    ;wait till msg arrives
                                    ncall app,getevent,(app,-
1.l:avo,msg,evt)
                                    continueif avo=0

                                    ;pass out to event handler
                                    ncall avo,event,(avo,msg,evt:-)

                                    ;free event
                                    ncall ave,freeevent,(ave,msg:-)
                              until evt=EV_QUIT

                              ;hide dialog
                              ncall dialog,hide,(dialog:-)
                        endif

                        ;close any open objects
                        qcall
ave/toolkit/rsc/closeobs,((gp+AVEOBS_START),(gp+AVEOBS_END):-)

                              ;close toolkit
                              ncall ave,closetoolkit,(ave,tkit:-)
                  endif

                  ;close ave
                  ncall ave,close,(ave,app:i~)
            endif

            ;exit
            qcall lib/exit,(0:-)
      endif
      ret

      defendnz
```

```
      data

buildtable:
      rsc_int32
GP_DIALOGFLAGS,(FDI_INNER|FDI_CLOSE|FDI_BORDER|FDI_TITLE|FDI_CONTENT|FDI_DR
AG|FDI_CLOSE|FDI_DEPTH)
      rsc_int32 GP_PIXTYPE,PIX_TYPE_32
      rsc_int32 GP_PIXWIDTH,DIALOG_SIZE
      rsc_int32 GP_LAYOUTFLAGS,(FAVO_ALIGNCENTER|FAVO_NOFLOW)

      rsc_string GP_APPNAME,'Blend'
      rsc_dialog
GP_DIALOG,GP_APPNAME,GP_NULL,GP_DIALOGFLAGS,GP_DIALOGBORDER,GP_DIALOGTITLE,
GP_DIALOGCNT
      rsc_addlink GP_DIALOG,CH_DIALOG_ACTION,GP_APP,EV_QUIT

      rsc_pix GP_PIX,GP_PIXWIDTH,GP_PIXWIDTH,GP_PIXTYPE
      rsc_add GP_DIALOGCNT,GP_PIX,0

      rsc_flowlayout GP_LAYOUT,GP_LAYOUTFLAGS,GP_LAYOUTPADDING
      rsc_setlayout GP_DIALOGCNT,GP_LAYOUT

      rsc_setprefsize GP_DIALOG

      rsc_end

toolend

.end
```

This file may be found at *demo/ave/blend.asm.*

6.2.1 Resource Constructors

These are an optional set of macros, as used above, designed to simplify the procedure of constructing applications. Instead of defining functions individually, functions are constructed in a table driven manner. Resource constructors can also be useful for simplifying error handling.

A pointer is provided to a data structure for storing object instances, as well as a pointer to a data table that contains a list of byte codes and byte code parameters for the creation of all of the common AVE objects. Macros are provided to simplify the creation of the byte code table. For an example of the use of resource constructors please see the files *demo/ave/blend.asm*, or for alternative approaches *demo/ave/togrey.asm*.

6.2.2 C Bindings

Note that C bindings for writing intent programs may be found in lang/cc/include/elate/ave.

6.3 Event Types

As shown in the previous examples, it is occasionally necessary to detect and process events in the application's main event loop. For list of defined constants see the ave/*evt/class.inc* file.

6.4 Gadget Programming

6.4.1 Gadget Types

The AVE standard gadget toolkit (*ave/toolkit/tao/gdg*) currently supports the following gadgets :

Border	A border
Button	A push button
Checkbox	A push checkbox
Choice	Selected item list
Cmenuitem	A menu item that can be toggled
Column	
File	File Browser
Label	Textual label
Lbutton	Latched button
Lcheckbox	Latched check box
List	Scrollable, selectable list
Menu	Pop up menu
Menubar	Pop up menu bar
Menuitem	Item on pop up menu
Progress	Progress Indicator
Rbutton	Radio button
Rcheckbox	Radio checkbox
Scrollbar	Standard scrollbar
Scrollpane	Scrollable Area
Stroke	Mouse/pen stroke capture
Textarea	Scrollable textual area
Textfield	Editable textual area
Title	A title

The toolkit gadget class (*ave/toolkit/tao/gdg/class.asm*) has two main methods in addition to all those inherited from the AVO base class:

- *setstate* – this method allows individual state flags to be set, cleared or toggled. The lower 16 bits are reserved for system use, while the top 16 are gadget specific. Any change of state generates a *postlink* call to channel 0 containing the new state. A mask can be set for the bits that are to cause an automatic call to update the gadget's appearance. This makes use of the methods getupdate and setupdate.

- *getstate* – this method returns the current state flags of the gadget.

Furthermore, each subclass of *ave/toolkit/tao/gdg/* may define additional methods.

Each gadget has a fully documented API. See the file api.html in the gadget subdirectory. The details vary somewhat between gadget types.

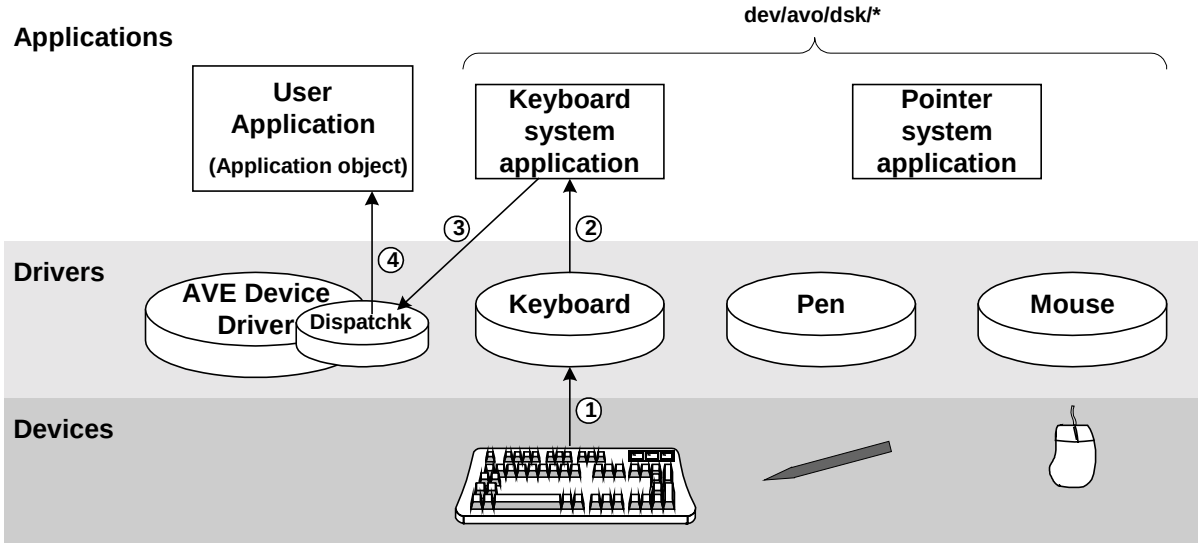
6.5 Gadget State and Action Channels

As mentioned previously it is possible to create event links between an AVO and a target EVT using the *addlink* method.

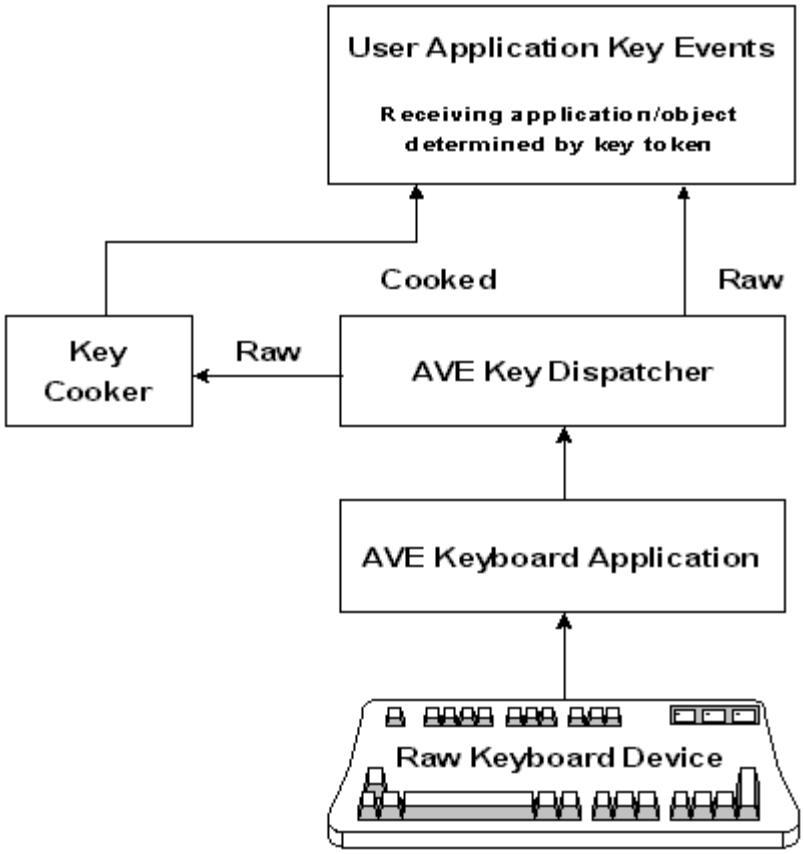
7 Anatomy of a System Application

System applications were introduced in section 3 of this manual. This section looks at the source for a simple system application and highlights the most important features.

The code analysed is for the synchronous keyboard application. This uses a polling loop to check for keyboard activity from the keyboard device driver. When keyboard activity is detected (2) the *dispatchk* method of the AVE device is invoked (3). The raw keyboard character is passed to *dispatchk* as a parameter. *Dispatchk* then creates an event message and posts it to the correct application AVO mailbox(4). (1) represents keyboard input into the keyboard device driver. This is illustrated in the following diagram :



The *dispatchk* method creates an event message that contains both raw and cooked key characters. This is achieved by the *dispatchk* method calling the keyboard cooker device, to cook the raw key code. This is shown in the diagram below:

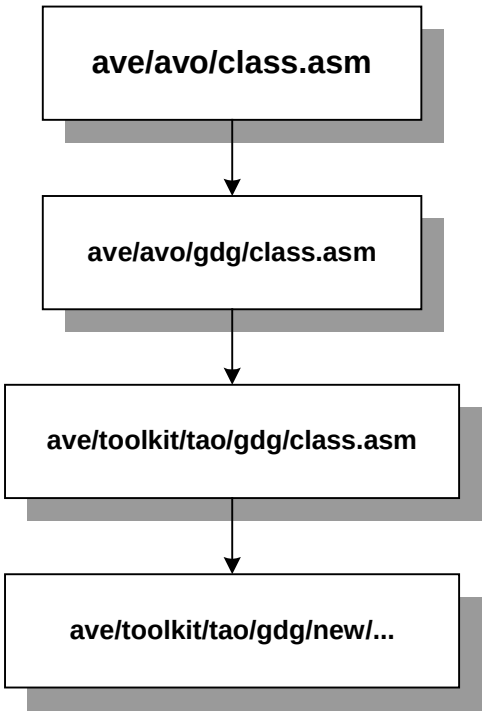


8 Creating New AVOs

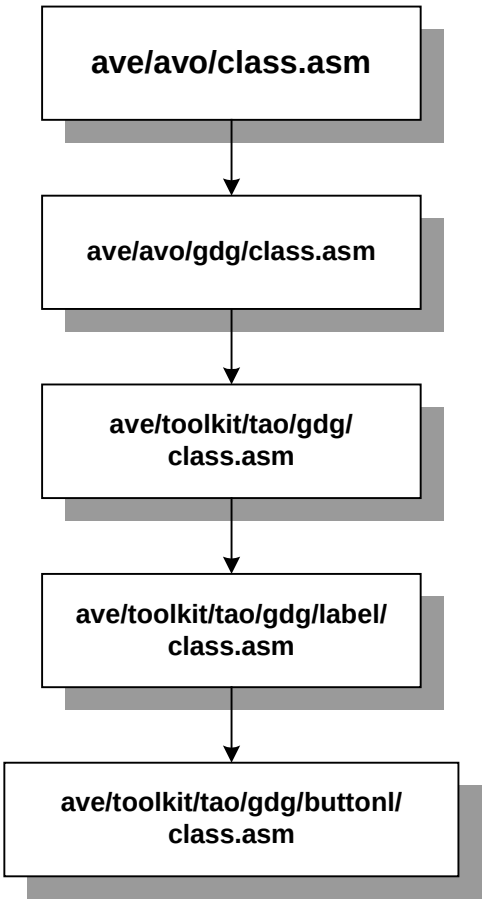
Creating new AVOs such as new dialog box types or new gadgets needs awareness of the hierarchical class structure of the AVE system. In fact intent itself is heavily based on an object-oriented class hierarchy, with use of inheritance and method overriding concepts.

The task of creating a new AVO involves writing a subclass at the appropriate level. This also involves selecting which methods to override, such as third level event handler methods, and adding new methods appropriate to the new AVOs features.

The following diagram shows an example of a new gadget that is created as part of the standard toolkit:



The diagram below shows how the button gadget is developed from the label gadget:



9 Displaying AVOs

AVOs are updated to screen by explicit or implicit update calls. The explicit calls are the AVO *update* and *updatepatch* methods, while implicit calls are from certain state changes to gadgets (such as gaining focus), or manipulation of AVOs (e.g. calling the change method).

During the update cycle, the redraw method on the avo will be called with an appropriate graphics context. The standard gadget (*ave/avo/gdg/std*) pre-processes this call, and then calls the *_redraw* method, where the actual gadget drawing occurs. Note that depending on whether the gadget is buffered or not, it will be passed either a graphics context or a pixelmap as the display surface; although most of the pixelmap API is reproduced in the context class, the *pick* method does not exist and must not be used.

AVOs other than gadgets just overload the *redraw* method. Pixelmaps call an appropriate *blit* method, and images typically make use of *modify* or one of the pixelmap drawing methods. Note that most *redraw* methods assume they are redrawing the whole of the AVO's area; the context handles any clipping.

9.1 Utilities - Patches

The intent media library frequently uses rectangular patches, which are small structures allocated from a memory heap object. There is a set of tools in *ave/rec/,* for maintaining lists of mutually exclusive patches (see the api.html document in the same directory for detailed information). The following operations are available: paste a region into a patch list; remove a region from a patch list; copy a region between patch lists; cut a region between patch lists; and clip a patch list to a bounding region.

The following routines are used in AVO methods to allow regions of the AVO to be registered and unregistered for updating, thus:

- addpatch (x,y,width,height) - pastes the region into the update list.
- subpatch (x,y,width,height) - removes the region from the update list.
- updatepatches() - updates the regions on the update patch list, then resets the list

9.2 Contexts

The intent media libraries make use of graphical context objects when updating the screen. These objects support a similar API to pixelmaps, but contain additional information about visible areas. Normally, an application will never need to use a graphics context directly (they are hidden within the *update* methods); however the AVO *redraw* methods may be passed in a context as the destination.

Contexts support the *read* and *modify* methods, but not the *pick* method.

10 Sizing and Layouts

Each AVO can return its current, minimum, maximum and preferred sizes. By default, preferred=minimum=current, and maximum is 2^31 square.

The *size* methods can be used in arranging AVOs within a containing gadget. To help arrange multiple children, a containing gadget can have a layout object added. The layout object overrides the gadget's *getXXXsize* methods, by interrogating the children. In addition, there is a layout method that resizes and repositions the children to fit the containing gadget. Layout occurs automatically when the containing gadget changes size.

Four standard layout objects are available. Horizontal and Vertical put all child AVOs in a single row or column respectively; Grid fits child AVOs into a regular grid, and Flow positions AVOs as per characters on a page.

11 Pix and Img Programming

11.1 Pixelmap AVOs

Pixelmaps are another subclass of the AVO baseclass and inherit all the capabilities of the AVO baseclass. Pixelmaps are rectangular areas onto which pixels can be drawn by various methods that are available to the pixelmap class. The pixelmap class methods include those for drawing polygons, circles, lines and pixels.

11.1.1 Colour Encoding

For pixelmaps all colours are specified internally in a 32-bit ARGB format (alpha, red, green, blue). Alpha refers to the transparency of the colour. Using this technique it is possible to create screens with multiple pixelmaps such that one pixelmap will be visible through another pixelmap which has, for example, 50% alpha (half transparent, 100% is fully transparent).

In certain circumstances pixelmaps may be created that are specified by the programmer with a colour depth that is not comparable with this 32-bit ARGB encoding scheme (e.g. 4-bit pixelmap). In this case, the internal ‘engine’ uses an algorithm to translate between the non 32-bit scheme and the internal format.

11.1.2 Alpha

This refers to the transparency of a pixelmap. An alpha of 0% (0) corresponds to an opaque pixelmap. An alpha of 100% (255) means that a pixelmap is entirely transparent. Pixelmaps incorporate an exceptionally fast software alpha-blending algorithm, enabling sophisticated effects without specialised underlying hardware.

11.1.3 Blitting and Copying

The pixelmap class includes various methods for copying and blitting (block image transfer) pixelmaps and pixelmap areas. Blits are used to transfer 2D graphical data from one pixelmap to another. Copies are used to transfer 2D graphical data from one area of a pixelmap to another area of the same pixelmap; copies are slower than blits as they have to correctly handle the source and destination overlapping in memory. Blits can be used to transfer within the same pixelmap, but only if the areas do not overlap.

11.1.4 Drawing Primitives

The pixelmap class also has methods for drawing graphics primitives, for example, circles, ovals, boxes, ellipses, arcs, polygons, lines and points. In addition, enclosed shapes can be drawn filled in a specified colour if required. Note that the algorithms used in drawing primitives have been hand optimised for speed and code compactness.

Six combinations exist for the use of any drawing primitive. To be specific, there are three operations, and two filling types. Primitives may be created with a solid fill comprised of one single colour, or they may possess a tiled fill with the pixels being selected from a tile pattern. Following this, they may be alpha blended with the destination, undergo colour replacement or be XORed. These combinations are depicted in the following table:

	Colour Replacement	Alpha	XOR
Plain Fill	Yes	Yes	Yes
Tiled Fill	Yes	Yes	Yes

These methods are further documented in *ave/avo/pix/api.html*.

11.1.5 Pixelmap Types

The following pixelmap types can be created :

Type	Colour scheme
Type 1	1 bit monochrome
Type 4	4 bit grayscale
Type 8	8 bit 3:3:2 RGB
Type 12	4:4:4:4 ARGB
Type 15	5:5:5 RGB
Type 16	5:6:5 RGB
Type 16 YBR	6:5:5 YCBCR
Type 24	8:8:8 RGB
Type 32	8:8:8:8 ARGB
Type 32 YBR	8:8:8:8 AYCBCR

11.1.6 Xmethods

Although the ncall is generally an efficient way of invoking object method code, there is a more suitable technique for situations where it is necessary to execute method code repetitively. An example application is where a large number of pixels need to be plotted sequentially, say for instance, to perform a fill operation.

If a method is likely to be invoked repetitively in this manner it is better to declare the method as an xmethod :

```
xmethod plot
            ent p0 i0 i1 i2:-

            qcall ave/avo/pix/12bit/12plot,(p0,i0,i1,i2:-),VIRTUAL|FIXUP
            ret
```

This allows an ncall of the type shown below to be made on the method:

```
            ncall p0,@plot,(p0:p1)
```

In this case p0 is the pixelmap object pointer. p1 returned by this call is actually a pointer to the method code. Once a pointer to the method code is obtained it is possible to invoke that code using a gos instruction. Performing a *gos* to method code is faster than performing a ncall. Continuing with the above example the plot method code could be executed using a call such as :

```
            gos p1,(p0,i0,i1,i2)
```

Notice that the parameters passed are now those normally passed to the method.

Xmethods are used extensively in the pixelmap class.

11.1.7 Blitting, Copying

The various methods available for transferring graphics data from one pixelmap to another are tabulated below. There are two method types :

- blit
- copy

The precise ordering of combinations open to the *blit* and *copy* methods is shown below. Initially either the *blit* or *copy* methods are subject to identical operations; as is the case with any drawing primitive.

	Colour Replacement	Alpha	XOR
blit	Yes	Yes	Yes
copy	Yes	Yes	Yes

The alpha variation ensures that the source and destination pixelmaps are blended according to the source alpha value. For the source alpha value 0 is taken as 100% source (i.e. source is opaque) and 255 is taken as 100% destination (i.e. source is completely transparent and so only the destination will be visible). With the XOR version of the method an exclusive-or operation is performed between the source and destination pixels.

Following this they are then subject to masking and scaling. The masked method of transfer fills the given region with pixels from the source pixelmap. Masked pixels in the source are not written to the destination.

It is possible to combine blitting, alpha-blending and masking or copying, XOR and scaling. It is also possible to combine masking and scaling. However, it is not possible to combine alpha-blending and XOR'ing, or blitting and copying operations.

	Masked	Scaled
Colour Replacement	Yes	Yes
Alpha	Yes	Yes
XOR	Yes	Yes

11.2 Image AVOs

Image AVOs provide many special effects. Currently available are:

filled – plain fill	htrans – fill given colour at half-transparency
rgbmask – logical colour mask	tiled – tile given image
table – table based colour conversion	mshade – shadow following masked area
togrey – colour to greyscale conversion	vgrad – vertical graduation between colours

To illustrate the use of image AVOs the vertically graded image AVO is used in *demo/ave/vgrad.asm* A vertically graded image is one where the top and bottom colours are specified, the colours and translucency being smoothly graduated between these in the vertical direction.

12 Sounds

Sounds are implemented by calling on the services provided by the DSFX device driver (dev/dsfx) and toolkit (*ave/avo/snd*). See *dev/dsfx/api.html* for more information.

13 The intent Font System

The Font System consists of three layers: glyph caching technology, vector rendering engine and format decoder plug-ins. Plug-ins include TrueType®, PostScript® Type 1, Windows® Resource Fonts and ROM Fonts. Features include 1 bit, 4 bit or 8 bit anti-aliasing, on-the-fly gamma correction, configurable cache memory, Panose™ matching, Micro-spaced printing, and co-existing plug-ins.

The font manager is responsible for maintaining a list of available fonts and fonts engines. It is used to open fonts, which are then used to render text.

When the font manager is first invoked, it examines the contents of a system file, which contains a listing of the font engines that the system wants to declare (as well as the relevant fonts and cache details). This approach means that no font engine is 'hardcoded' into the font system, but the required font engines are instead defined when the engine is initialised.

The system file *ave/font/store.map* is a plain text file containing the font manager information, as in this example snippet:

```
# Truetype font engine and fonts ...
.engine= ave/font/ttf/      # The TrueType Font Engine
fonts/ANMSSN__.TTF
fonts/ANSNB___.TTF
"fonts/this line has spaces.TTF"
fonts/ANSN____.TTF
fonts/ARCTIC.ttf
device/hostfs/c:/other_os/fonts/times.ttf   # map to file outside of Elate tree
```

Font engines are dynamically loaded by the font manager in order to ascertain the validity of the selected font engines. Each of these must comply to a specific interface, which is defined elsewhere. The engines are uniform across the system, since each engine's 'open' tool creates font object classes (which inherit from *ave/font/class*). A font object class is used for method calls and for drawing strings.

String drawing is the process of sourcing characters, and linking them together to produce the best typographical representation of the string. The string drawing group of tools provided within the int**e**nt font system are helper tools, which group together a number of tools and routines into a uniform way of producing text output. At present, these have only a very minor effect on text, but are used to uniformly handle spacing and microspacing across the system.

When creating a tool to use fonts with the int**e**nt media libraries the following methods are available, examples of which may be found in *demo/ave/font.asm*.

ave/font/open	Opens a font
ave/font/opens	This tool will open a font based on a single string description *
ave/font/close	Closes an opened font, and frees any associated resources.

* For example, 'Nimrod, 10points' or 'Andale Sans, 12points, italic'

For more information upon available methods, please see *ave/font/api.html*

14 Streaming

It should be noted that intent includes a Media Streaming architecture, a mechanism for streaming data across an object based network, which can be used to provide a user with a media experience, be that audio streaming, video playback or static image display. It could also represent a conversion of media from one type to another. Any type of data or media can be streamed across the network.

A media stream is a heterogeneous network of stream objects (also known as streamers). These are connected to each other via channels, which conceptually represent the inputs and outputs of a streamer. Please note that the term channel should not be confused with audio channels (i.e. left or right channel). The objects communicate media data using the channels. The media data, referred to as a media buffer, is used as a container for the media being communicated. In practice this would mean that a streamer would create a media buffer and pack it with data. This is then sent to the channel object. The channel object implements most of the inter-stream object communications. The channel object can be considered as an internal part of the stream object. From this point the media data is passed to a media buffer belonging to each registered observer, such as a second streamer.

Media Buffers

A media stream is a flow of media buffers, where the entire stream is represented as a group of smaller buffers. These smaller buffers are the media buffers. A media buffer is a class, which encapsulates a block of media data. There are several types of media buffer, each of which is derived from the media buffer base class. They will often extend the API to include a mechanism to note their position in the media stream. For example, a streamer that outputs blocks from a file, or network pipe, would set the index for the buffer, i.e. the buffer index would be the byte offset index into the file.

Types of Stream Object

For more detailed information on the API for media streaming, please refer to *ave/media/api.html.*

Active Source	An active source stream object is a streamer that actively produces data for the media stream. An example of such an object could be a byte stream producer.
Active Sink	An active sink stream object, consumes the media stream, either displaying an image, playing back audio, or displaying subtitles.
Passive Source	A passive source streamer produces data into the media. Normally a passive source streamer will generate a media stream as the result of an external event such as a mouse click or a timer callback.
Passive Sink	A passive sink consumes data in a media stream but is not considered as a main data sink. E.g. a visualisation module.
Codec	A codec converts one or more input datatypes to a different datatype.
Filter	A filter modifies the data in a stream, but does not change the input datatype.

15 Import/Export Utilities.

The *ave/cnv* directory contains a number of tools for import and export of images and sounds. The main tools are *ave/cnv/load* and *ave/cnv/save*.

15.1 Loading.

The intent toolkit can import to pixelmaps files of the following types:

CPM	intent native compressed pixelmap
RPM	intent native raw (uncompressed) pixelmap
JPG	
PCX	
BMP	
XBM	
GIF	
PNG	

The img2cpm shell command may be used to convert any of these graphics file formats to the cpm format. More information on this command may be found in the *'Reference Manual for the intent Shell and Shell Commands.'*

The intent toolkit can import to soundmaps files of the following types:

CSM	intent native compressed soundmap
RSM	intent native raw (uncompressed) soundmap
WAV	
AU	

The file type being loaded is determined by the file extension of the inputted filename.

15.2 Saving

The intent toolkit can export from pixelmaps files of the following types:

CPM	intent native compressed pixelmap
RPM	intent native raw (compressed) pixelmap
TGA	
BMP	
PPM	

The intent toolkit can export from soundmaps files of the following types:

CSM	intent native compressed soundmap
RSM	intent native raw (uncompressed) soundmap

The file type being saved is determined by the file extension of the inputted filename.

In addition, there is a caching loader (*ave/cnv/share*). This is similar to *ave/cnv/load*, except that the bitmap data portion of the pixelmap is cached, so subsequent calls to share with the same parameters return a pixelmap pointing to the shared data. Note that in order to be shared, pixelmaps or soundmaps must be of the same type.

16 Glossary of all Terms

- AIFF

An acronym for Audio Interchange File Format. A format developed by Apple Computer Inc. for storing high-quality sampled audio and musical instrument information.

- Alpha

For this document, alpha refers to the transparency of a pixelmap. An alpha of 0% (0) corresponds to an opaque pixelmap. An alpha of 100% (255) means that a pixelmap is entirely transparent.

- ARGB

The colour model made use of by the intent media libraries. RGB refers to the traditional combination of red, green and blue colours, while A refers to the alpha value.

- AU

An audio format developed by Sun Microsystems.

- AVO

An acronym for audio visual object. The fundamental building block of intent media libraries.

- Blitting

A block Image Transfer, i.e. a process of transferring a 2D region from one memory space to another.

- BMP

A bitmap file format developed by Microsoft®.

- DSFX

An abbreviated term for the intent digital sound effects device driver.

- Events

Typically, this term includes actions such as mouse button presses, mouse movement and the pressing or releasing of keys.

- Fonts

A typeface which has a particular style, and point size. A font is used to perform text string operations like printing and measuring the graphical size of a string.

- Gadgets

A variety of components provided by intent for deployment in the construction of a graphical user interface. For example, menus, windows and scrollbars.

- Glyph

Either the actual shape (bit pattern, outline) of a character image, or a surface form derived from some combination of underlying characters in some specific context.

- GIF

Acronym for Graphics Interchange Format. A standard for digitised images compressed with the LZW algorithm, defined in 1987 by CompuServe.

- GRF

An abbreviated term for the int**e**nt graphics driver.

- GUI

Acronym for graphical user interface. A graphical environment through which a user may interact with the software running a particular device.

- JPG / JPEG

Joint Photographic Experts Group. The original name of the committee that designed the standard image compression algorithm of the same name. Also abbreviated to jpg.

- Kerning

To adjust the intercharacter spacing in character groups (i.e. words), thereby improving their appearance.

- MPEG

An acronym for Moving Picture Experts Group. The term also refers to the family of digital video compression standards and file formats developed by the group.

- MP3

The file extension for MPEG, audio layer 3. Layer 3 is one of three coding schemes for the compression of audio signals. It uses perceptual audio coding and psychoacoustic compression to remove all superfluous information.

- Panose

The PANOSE Typeface Matching System was developed by Benjamin Bauermeister and is licensed to Hewlett-Packard Corporation. A system for matching static or distortable fonts, by objectively classifying fonts according to their visual characteristics. The name "PANOSE" is derived from the six letters used to distinguish font attributes.

- PNG

An acronym for Portable Network Graphics. An extensible file format for the lossless, portable and well-compressed storage of raster images.

- Pixelmap

An AVO defining a 2D graphical region containing graphical information.

- PostScript®

A font standard developed by Adobe Systems Inc. PostScript Type One fonts contain hinting information which allows fonts to be rendered more readable at lower resolutions and small type sizes.

- TGA

An acronym for Targa Graphics Adaptor. The Truevision Targa Graphics Adaptor file format.
The TGA format is a common bitmap file format for storage of 24-bit images.

- TrueType®

A font standard initially developed by Apple Computer as an alternative to the PostScript standard propounded by Adobe Systems Inc. Like PostScript Type 1 fonts, it is an outline font format that allows for the scaling of fonts, thereby adjusting them to any size.

- Typeface

The features by which a character's design is deemed to be recognisable.

- Unicode

A character encoding system designed to support the interchange, processing, and display of written texts in a diversity of languages. The unicode standard defines a unique number for every character.

- WAV

Wav standard. A sound format developed by Microsoft and used extensively in Microsoft Windows.

- xmethods

Methods that allow their method code to be called directly. This is faster than the usual named method call (ncall).

- XOR

An exclusive or operation.

---

17 Appendix: Directory Structure

The main intent components can be found in the */ave* directory of the intent system. The */ave* directory is divided into a number of sub-directories:

AVOs:

/ave/avo/gdg	Contains the gadget AVO class
/ave/avo/pix	Contains the pixelmap AVO class
/ave/avo/win	Contains the window AVO class
/ave/avo/snd	Contains the sound AVO class
/ave/avo/img	Contains the image AVO class

Media Streaming Architecture:

ave/media/bf	The media buffer class, and classes derived from it
ave/media/mgr	Contains the media manager
ave/media/msc	Media stream controller.
ave/media/strm	Contains the media stream objects.

Other Functionality:

/ave/cnv	Pixelmap conversion tools
/ave/csr	Cursor Managers
/ave/flm	Film playing and recording tools
/ave/font	Bitmapped font rendering
/ave/heap	Heap memory management tools
/ave/rec	Region patch tools
/ave/input	Input Method Tools
/ave/ctx	Graphics Contexts
/ave/layout	Gadget Layout Tools
/ave/rec	Rectangle Patches Library
/ave/toolkit	Look & Feel Toolkits
/ave/util	Utilities e.g. wrapper classes for sound

Drivers:

dev/ave	Contains the main device driver
dev/dsfx	Sound device driver
dev/plane	Planes drivers

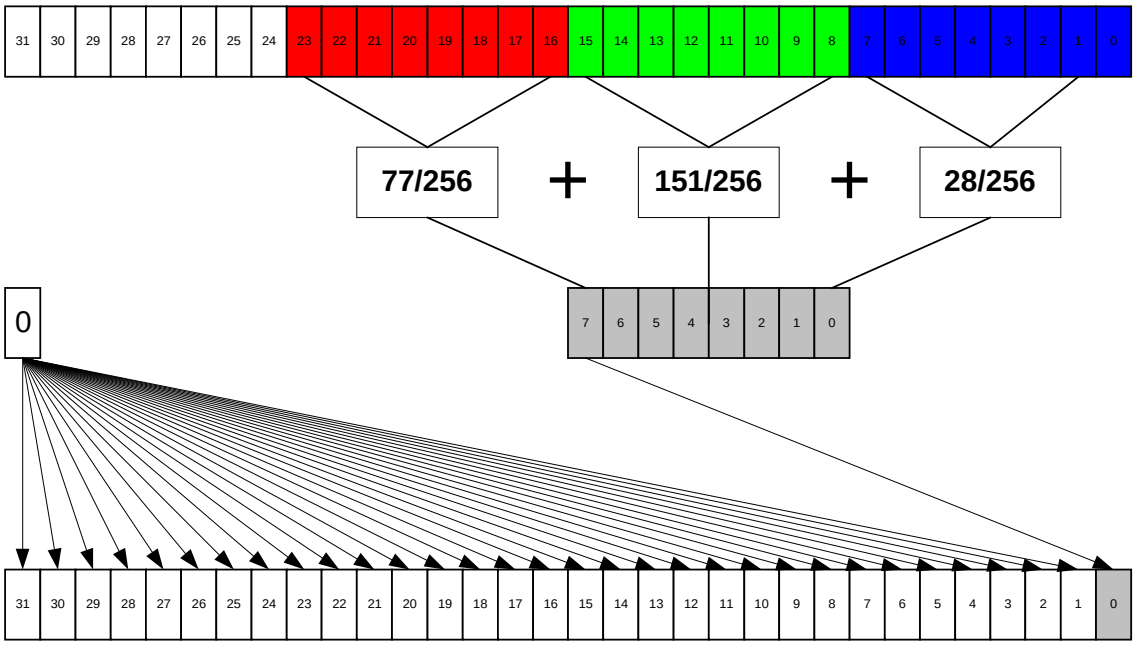
Demonstration Programs:

demo/ave	Demonstration Programs

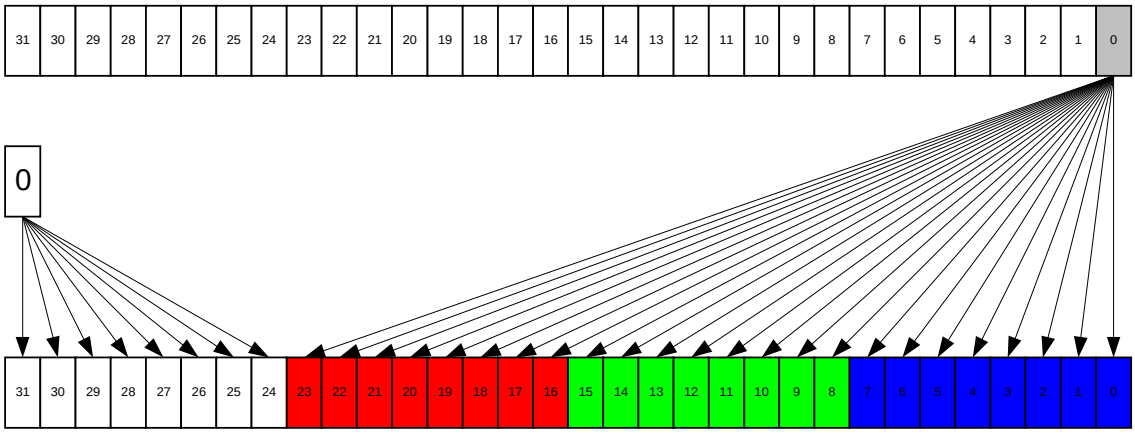
18 Appendix 2: Colour Depth Conversion

Type 1	1 bit monochrome
Type 4	4 bit grayscale
Type 12	4:4:4:4 ARGB
Type 15	5:5:5 RGB
Type 16	5:6:5 RGB
Type 24	8:8:8 RGB
Type 32	8:8:8:8 ARGB

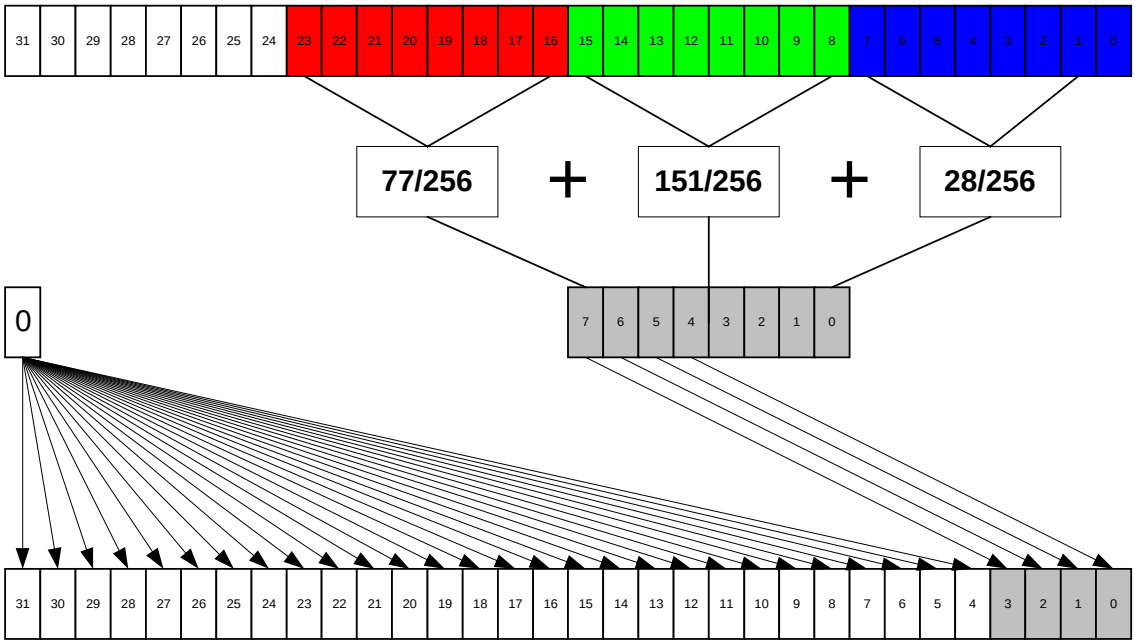
Type 1 Pack



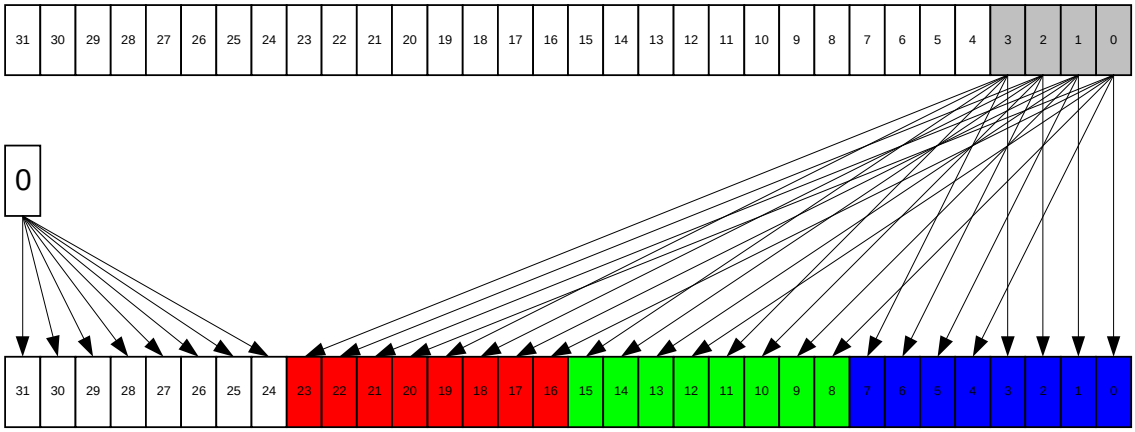
Type 1 Unpack



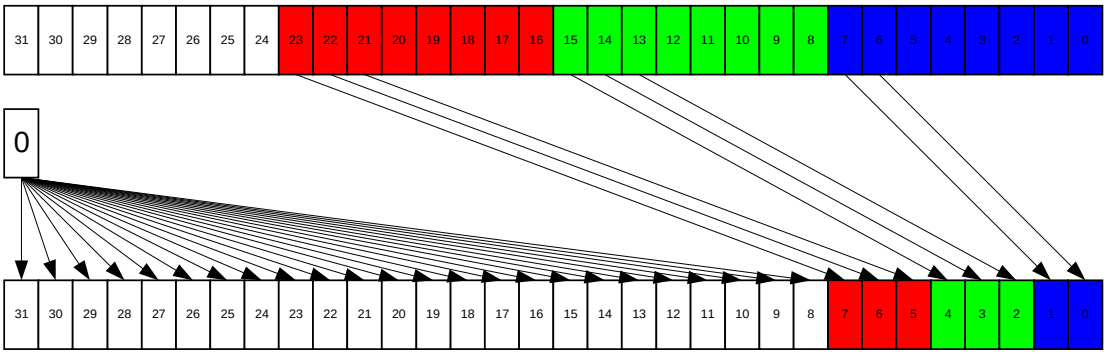
Type 4 Pack



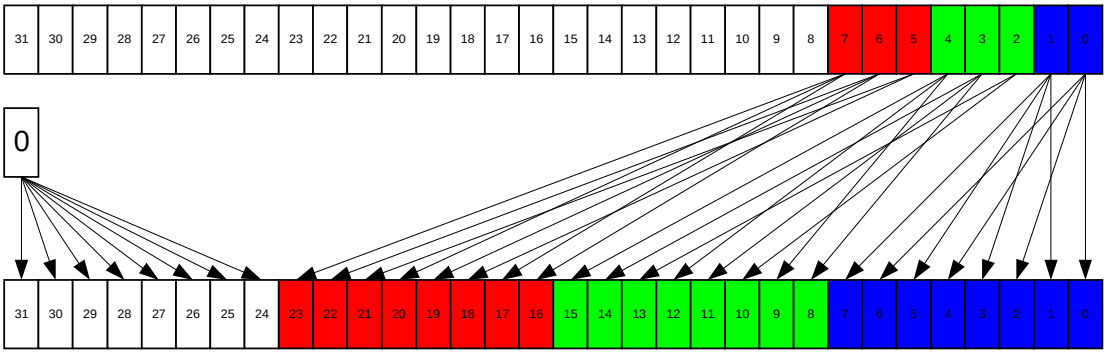
Type 4 Unpack



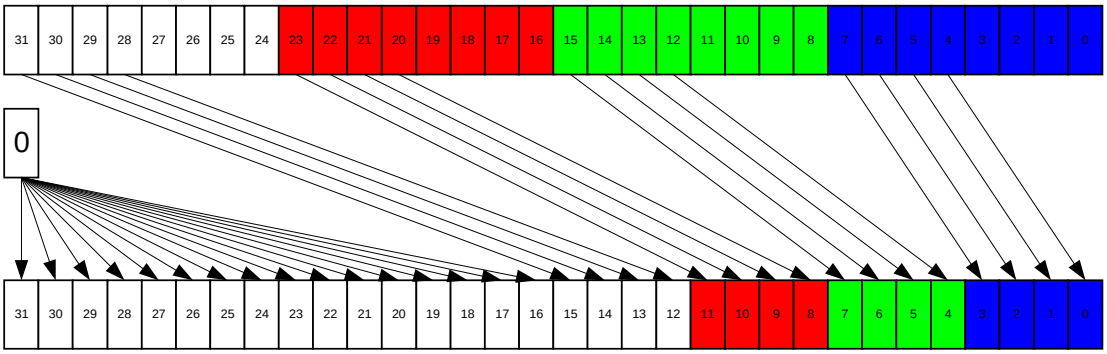
Type 8 Pack



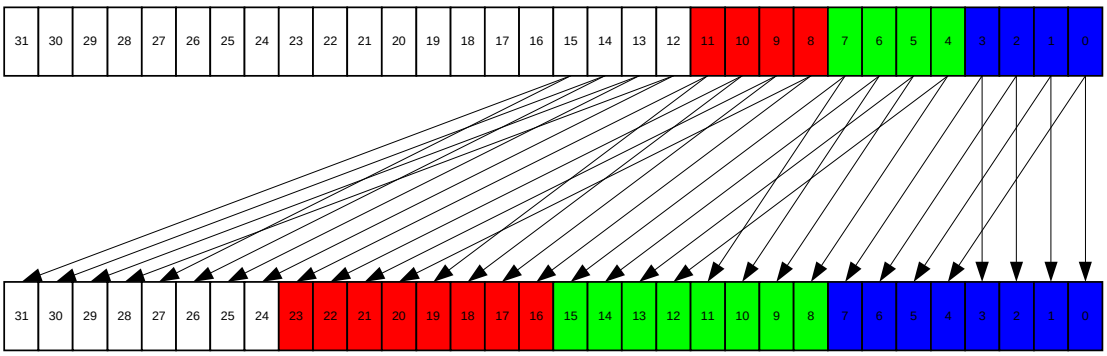
Type 8 Unpack



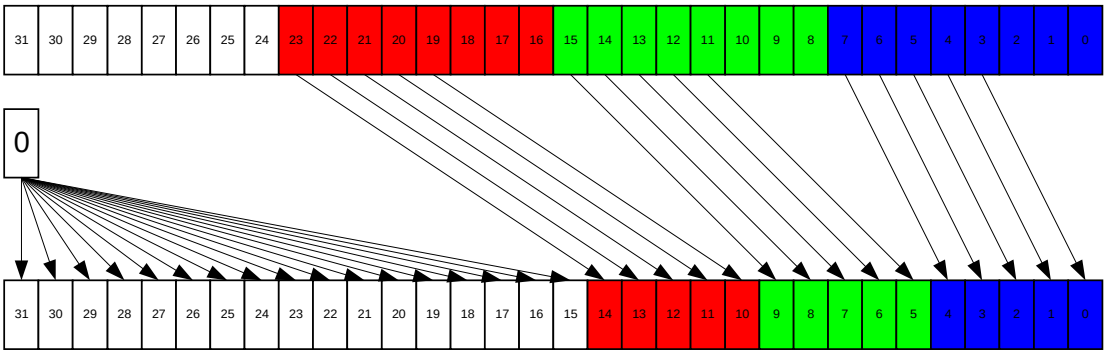
Type 12 Pack



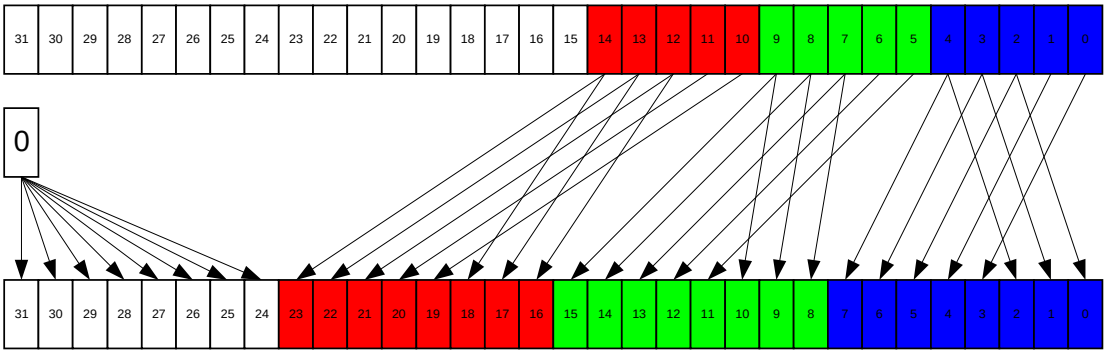
Type 12 Unpack



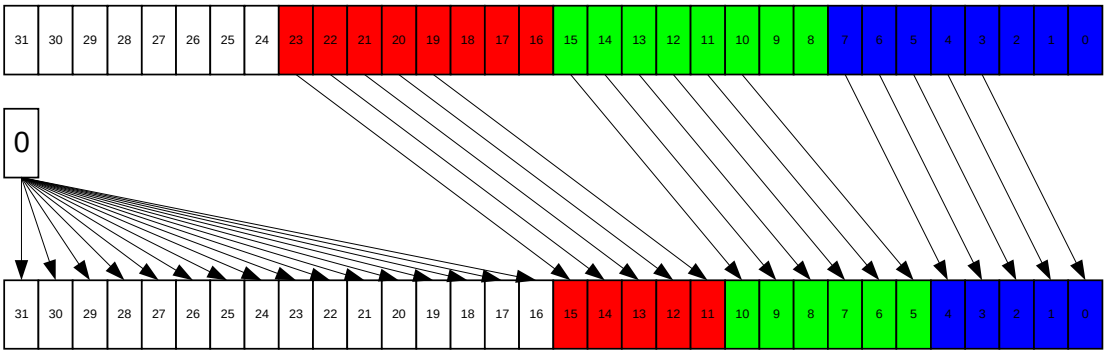
Type 15 Pack



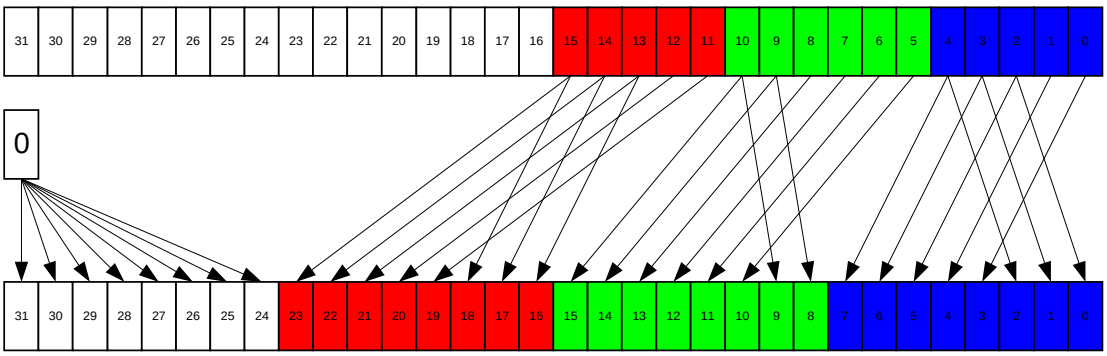
Type 15 Unpack



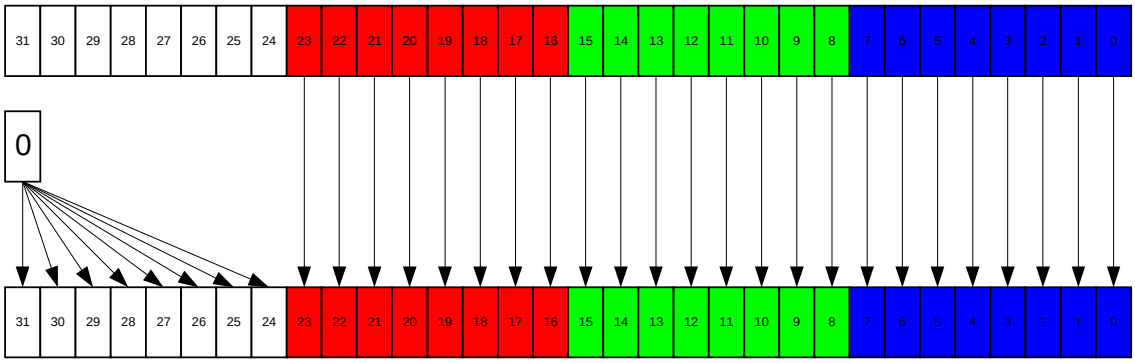
Type 16 Pack



Type 16 Unpack



Type 24 Pack



Type 24 Unpack