Devices

Contents

Chapter 1

Devices

1.1 Amiga® RKM Devices: 5 Gameport Device

```
The gameport device manages access to the Amiga gameport connectors for
the operating system.  It enables the Amiga to interface with various
external pointing devices like mice (two and three button), joysticks,
trackballs and light pens.  There are two units in the gameport device,
unit 0 and unit 1.

                       AMIGA GAMEPORT CONNECTORS

                Model       Unit 0              Unit 1
                -----   ---------------     --------------
                A3000   Front Connector     Back Connector
                A2000   Left Connector      Right Connector
                A1000   1                   2
                A500    1 JOYSTICK          2 JOYSTICK
```

1.2 5 Gameport Device / Gameport Device Commands and Functions

Command	Operation
CMD_CLEAR	Clear the gameport input buffer.
GPD_ASKCTYPE	Return the type of gameport controller being used.
GPD_ASKTRIGGER	Return the conditions that have been preset for triggering.
GPD_READEVENT	Read one or more gameport events.
GPD_SETCTYPE	Set the type of the controller to be used.

```
GPD_SETTRIGGER  Preset the conditions that will trigger a gameport event.


Exec Functions as Used in This Chapter
--------------------------------------
AbortIO()       Abort a command to the gameport device.

CheckIO()       Return the status of an I/O request.

CloseDevice()   Relinquish use of the gameport device.  All requests must
                be complete before closing.

DoIO()          Initiate a command and wait for completion (synchronous
                request).

OpenDevice()    Obtain shared use of one unit of the gameport device. The
                unit number specified is placed in the I/O request
                structure for use by gameport commands.

SendIO()        Initiate a command and return immediately (asynchronous
                request).

WaitIO()        Wait for the completion of an asynchronous request.  When
                the request is complete the message will be removed from
                reply port.


Exec Support Functions as Used in This Chapter
----------------------------------------------
CreateExtIO()   Create an extended I/O request structure of type IOStdReq.
                This structure will be used to communicate commands to the
                gameport device.

CreatePort()    Create a signal message port for reply messages from the
                gameport device.  Exec will signal a task when a message
                arrives at the port.

DeleteExtIO()   Delete an I/O request structure created by CreateExtIO().

DeletePort()    Delete the message port created by CreatePort().

   Who Runs The Mouse?
   -------------------
   When the input device or Intution is operating, unit 0 is usually
   dedicated to gathering mouse events.  The input device uses the
   gameport device to read the mouse events.  (For applications that
   take over the machine without starting up the input device or
   Intuition, unit 0 can perform the same functions as unit 1.)  See the
   "Input Device" chapter for more information on the input device.
```

1.3 5 Gameport Device / Device Interface

```
The gameport device operates like the other Amiga devices.  To use it, you
must first open the gameport device, then send I/O requests to it, and
```

then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The I/O request used by the gameport device is called IOStdReq.

```
    struct IOStdReq
    {
        struct  Message io_Message;
        struct  Device  *io_Device; /* device node pointer  */
        struct  Unit    *io_Unit; /* unit (driver private)*/
        UWORD   io_Command;       /* device command */
        UBYTE   io_Flags;
        BYTE    io_Error;         /* error or warning num */
        ULONG   io_Actual;        /* actual number of bytes transferred */
        ULONG   io_Length;        /* requested number bytes transferred*/
        APTR    io_Data;          /* points to data area */
        ULONG   io_Offset;        /* offset for block structured devices */
    };
```

See the include file exec/io.h for the complete structure definition.

 Opening The Gameport Device
 Gameport Device Controllers
 Closing The Gameport Device

1.4 5 / Device Interface / Opening The Gameport Device

Three primary steps are required to open the gameport device:

* Create a message port using CreatePort(). Reply messages from the device must be directed to a message port.

* Create an I/O request structure of type IOStdReq. The IOStdReq structure is created by the CreateExtIO() function. CreateExtIO() will initialize the I/O request with your reply port.

* Open the gameport device. Call OpenDevice(), passing the I/O request and and indicating the unit you wish to use.

```
   struct MsgPort *GameMP;   /* Message port pointer */
   struct IOStdReq *GameIO;  /* I/O request pointer */

     /* Create port for gameport device communications */
   if (!(GameMP = CreatePort("RKM_game_port",0)))
       cleanexit(" Error: Can't create port\n",RETURN_FAIL);

     /* Create message block for device I/O */
   if (!(GameIO = CreateExtIO(GameMP,sizeof(struct IOStdReq))))
       cleanexit(" Error: Can't create I/O request\n",RETURN_FAIL);

     /* Open the right/back (unit 1, number 2) gameport.device unit */
   if (error=OpenDevice("gameport.device",1,GameIO,0))
       cleanexit(" Error: Can't open gameport.device\n",RETURN_FAIL);
```

The gameport commands are unit specific. The unit number specified in the call to OpenDevice() determines which unit is acted upon.

1.5 5 / Device Interface / Gameport Device Controllers

The Amiga has five gameport device controller types.

GAMEPORT DEVICE CONTROLLERS

Controller Type	Description
GPCT_MOUSE	Mouse controller
GPCT_ABSJOYSTICK	Absolute (digital) joystick
GPCT_RELJOYSTICK	Relative (digital) joystick
GPCT_ALLOCATED	Custom controller
GPCT_NOCONTROLLER	No controller

To use the gameport device, you must define the type of device connected to the gameport and define how the device is to respond. The gameport device can be set up to return the controller status immediately or only when certain conditions have been met.

When a gameport device unit reponds to a request for input, it creates an input event. The contents of the input event will vary based on the type of device and the trigger conditions you have declared.

* A mouse controller can report input events for one, two, or three buttons and for positive or negative (x,y) movements. A trackball controller or car-driving controller is generally of the same type and can be declared as a mouse controller.

* An absolute joystick reports one single event for each change of its current location. If, for example, the joystick is centered and the user pushes the stick forward and holds it in that position, only one single forward-switch event will be generated.

* A relative joystick, on the other hand, is comparable to an absolute joystick with "autorepeat" installed. As long as the user holds the stick in a position other than centered, the gameport device continues to generate position reports.

* There is currently no system software support for proportional joysticks or proportional controllers (e.g., paddles). If you write custom code to read proportional controllers or other controllers (e.g., light pen) make certain that you issue GPD_SETCTYPE (explained below) with controller type GPCT_ALLOCATED to insure that other applications know the connector is being used.

GPCT_NOCONTROLLER.

The controller type GPCT_NOCONTROLLER is not a controller at all, but a flag to indicate that the unit is not being used at the present time.

1.6 5 / Device Interface / Closing The Gameport Device

```
Each OpenDevice() must eventually be matched by a call to CloseDevice().

All I/O requests must be complete before CloseDevice().  If any requests
are still pending, abort them with AbortIO() and remove them with WaitIO().

    if (!(CheckIO(GameIO)))
        {
        AbortIO(GameIO);  /* Ask device to abort request, if pending */
        }
    WaitIO((GameIO);   /* Wait for abort, then clean up */
    CloseDevice(GameIO);
```

1.7 5 Gameport Device / Gameport Events

```
A gameport event is an InputEvent structure which describes the following:

   * The class of the event - always set to IECLASS_RAWMOUSE for the
     gameport device.

   * The subclass of the event - 0 for the left port; 1 for the right port.

   * The code - which button and its state.  (No report = 0xFF)

   * The qualifier - only button and relative mouse bits are set.

   * The position - either a data address or mouse position count.

   * The time stamp - delta time since last report, returned as frame
     count in tv_secs field.

   * The next event - pointer to next event.


    struct InputEvent GameEV
    {
        struct InputEvent *ie_NextEvent;  /* next event */
        UBYTE    ie_Class;                /* input event class */
        UBYTE    ie_SubClass;             /* subclass of the class */
        UWORD    ie_Code;                 /* input event code */
        UWORD    ie_Qualifier;            /* event qualifiers in effect */
           union
           {
             struct
             {
              WORD   ie_x;                /* x position for the event */
              WORD   ie_y;                /* y position for the event */
             }  ie_xy;
             APTR ie_addr;
           } ie_position;
        struct timeval ie_TimeStamp;      /* delta time since last report
    }
```

See the include file devices/inputevent.h for the complete structure definition and listing of input event fields.

1.8 5 / Gameport Events / Reading Gameport Events

You read gameport events by passing an I/O request to the device with GPD_READEVENT set in io_Command, the address of the InputEvent structure to store events set in io_Data and the size of the structure set in io_Length.

```
struct InputEvent  GameEV;
struct IOStdRequest *GameIO;  /* Must be initialized prior to using */

void send_read_request()
{
GameIO->io_Command = GPD_READEVENT;  /* Read events */
GameIO->io_Length = sizeof (struct InputEvent);
GameIO->io_Data = (APTR)&GameEV;     /* put events in GameEV*/
SendIO(GameIO);                      /* Asynchronous */
}
```

1.9 5 / Gameport Events / Setting Gameport Event Trigger Conditions

You set the conditions that can trigger a gameport event by passing an I/O request to the device with GPD_SETTRIGGER set in io_Command and the address of a GamePortTrigger structure set in io_Data.

The information needed for gameport trigger setting is placed into a GamePortTrigger data structure which is defined in the include file devices/gameport.h.

```
struct GamePortTrigger
{
    UWORD    gpt_Keys;      /* key transition triggers */
    UWORD    gpt_Timeout;   /* time trigger (vertical blank units) */
    UWORD    gpt_XDelta;    /* X distance trigger */
    UWORD    gpt_YDelta;    /* Y distance trigger */
}
```

A few points to keep in mind with the GPD_SETTRIGGER command are:

* Setting GPTF_UPKEYS enables the reporting of upward transitions. Setting GPTF_DOWNKEYS enables the reporting of downward transitions. These flags may both be specified.

* The field gpt_Timeout specifies the time interval (in vertical blank units) between reports in the absence of another trigger condition. In other words, an event is generated every gpt_Timeout ticks.

```
   Vertical blank units may differ from country to country (e.g 60 Hz
   NTSC, 50 Hz PAL.)  To find out the exact frequency use this code
   fragment:

   #include <exec/execbase.h>
   extern struct ExecBase *SysBase;

   UBYTE get_frequency(void)
   {
   return((UBYTE)SysBase->VBlankFrequency);
   }

*  The gpt_XDelta and gpt_YDelta fields specify the x and y distances
   which, if exceeded, trigger a report.
```

For a mouse controller, you can trigger on a certain minimum-sized move in either the x or y direction, on up or down transitions of the mouse buttons, on a timed basis, or any combination of these conditions.

For example, suppose you normally signal mouse events if the mouse moves at least 10 counts in either the x or y directions. If you are moving the cursor to keep up with mouse movements and the user moves the mouse less than 10 counts, after a period of time you will want to update the position of the cursor to exactly match the mouse position. Thus the timed report of current mouse counts would be preferred. The following structure would be used:

```
#define XMOVE 10
#define YMOVE 10

struct GamePortTrigger GameTR =
{
    GPTF_UPKEYS | GPTF_DOWNKEYS, /* trigger on all key transitions */
    1800,                   /* and every 36(PAL) or 30(NTSC) seconds */
    XMOVE,                  /* for any 10 in an x or y direction */
    YMOVE
};
```

For a joystick controller, you can select timed reports as well as button-up and button-down report trigger conditions. For an absolute joystick specify a value of one (1) for the GameTR_XDelta and GameTR_YDelta fields or you will not get any direction events. You set the trigger conditions by using the following code or its equivalent:

```
struct IOStdReq *GameIO;

void set_trigger_conditions(struct GamePortTrigger *GameTR)
{
GameIO->io_Command = GPD_SETTRIGGER;    /* set trigger conditions */
GameIO->io_Data = (APTR)GameTR;         /* from GameTR */
GameIO->io_Length = sizeof(struct GamePortTrigger);
DoIO(GameIO);
}
```

```
Triggers and Reads.
-------------------
If a task sets trigger conditions and does not ask for the position
```

```
reports the gameport device will queue them up anyway. If the trigger
conditions occur again and the gameport device buffer is filled, the
additional triggers will be ignored until the buffer is read by a
device read request (GPD_READEVENT) or a system CMD_CLEAR command
flushes the buffer.
```

1.10 5 / Gameport Events / Determining The Trigger Conditions

```
You determine the conditions required for triggering gameport events by
passing an I/O request to the device with GPD_ASKTRIGGER set in
io_Command, the length of the GamePortTrigger structure set in io_Length
and the address of the structure set in io_Data.  The gameport device will
respond with the event trigger conditions currently set.

    struct IOStdReq *GameIO;  /* Must be initialized prior to using */

    struct GamePortTrigger GameTR;

    void get_trigger_conditions(struct GamePortTrigger *GameTR)
    {
    GameIO->io_Command = GPD_ASKTRIGGER;    /* get type of triggers */
    GameIO->io_Data = (APTR)GameTR;         /* place data here */
    GameIO->io_Length= sizeof(GameTR);
    DoIO(GameIO);
    }
```

1.11 5 Gameport Device / Setting and Reading the Controller Type

```
 Determining The Controller Type
 Setting The Controller Type
```

1.12 5 / Setting and Reading Controller Type / Determining The Controller Type

```
You determine the type of controller being used by passing an I/O request
to the device with GPD_ASKCTYPE set in io_Command, 1 set in io_Length and
the number of the unit set in io_Unit. The gameport device will respond
with the type of controller being used.

   struct IOStdReq *GameIO;  /* Must be initialized prior to using */

   BYTE GetControllerType()
   {
   BYTE controller_type = 0;

   GameIO->io_Command = GPD_ASKCTYPE;          /* get type of controller */
   GameIO->io_Data = (APTR)&controller_type;  /* place data here */
   GameIO->io_Length = 1;
   DoIO(GameIO);
   return (controller_type);
   }
```

```
The BYTE value returned corresponds to one of the five controller types
noted above.
```

1.13 5 / Setting and Reading the Controller Type / Setting The Controller Type

```
You set the type of gameport controller by passing an I/O request to the
device with GPD_SETCTYPE set in io_Command, 1 set in io_Length and the
address of the byte variable describing the controller type set in io_Data.

The gameport device is a shared device; many tasks may have it open at any
given time. Hence, a high level protocol has been established to prevent
multiple tasks from reading the same unit at the same time.

 Three Step Protocol for Using the Gameport Device
```

1.14 5 / Setting Controller Type /Three Step Protocol for Using Gameport Device

```
Step 1:
    Send GPD_ASKCTYPE to the device and check for a GPCT_NOCONTROLLER
    return. Never issue GPD_SETCTYPE without checking whether the desired
    gameport unit is in use.

Step 2:
    If GPCT_NOCONTROLLER is returned, you have access to the gameport.
    Set the allocation flag to GPCT_MOUSE, GPCT_ABSJOYSTICK or
    GPCT_RELJOYSTICK if you use a system supported controller, or
    GPCT_ALLOCATED if you use a custom controller.

    struct IOStdReq *GameIO;  /* Must be initialized prior to using */

    BOOL set_controller_type(type)
    BYTE type;
    {

    BOOL success = FALSE;
    BYTE controller_type = 0;

    Forbid();                           /*critical section start */
    GameIO->io_Command = GPD_ASKCTYPE;  /* inquire current status */
    GameIO->io_Length = 1;
    GameIO->io_Flags = IOF_QUICK;
    GameIO->io_Data = (APTR)&controller_type; /* put answer in here */
    DoIO(GameIO);

    /* No one is using this device unit, let's claim it */
    if (controller_type == GPCT_NOCONTROLLER)
        {
        GameIO->io_Command = GPD_SETCTYPE;/* set controller type */
        GameIO->io_Length = 1;
        GameIO->io_Data = (APTR)&type;  /* set to input param */
        DoIO( GameIO);
        success = TRUE;
        UnitOpened = TRUE;
        }
    Permit(); /* critical section end */

    /* success can be TRUE or FALSE, see above */
    return(success);
    }

Step 3:
    The program must set the controller type back to GPCT_NOCONTROLLER
    upon exiting your program:

    struct IOStdReq *GameIO;  /* Must be initialized prior to using */

    void free_gp_unit()
    {
    BYTE type = GPCT_NOCONTROLLER;
    GameIO->io_Command = GPD_SETCTYPE;  /* set controller type */
    GameIO->io_Length = 1;
    GameIO->io_Data = (APTR)&type;      /* set to unused */
    DoIO( GameIO);
    }

This three step protocol allows applications to share the gameport device
in a system compatible way.

    A Word About The Functions.
    ---------------------------
    The functions shown above are designed to be included in any
    application using the gameport device.  The first function,
    set_controller_type(), would be the first thing done after opening
    the gameport device. The second function, free_gp_unit(), would be
    the last thing done before closing the device.
```

1.15 5 Gameport Device / Additional Information on the Gameport Device

```
Additional programming information on the gameport device can be found in
the include files and the Autodocs for the gameport and input devices.
Both are contained in the Amiga ROM Kernel Reference Manual: Includes and
Autodocs.

                        Gameport Device Information
                    -----------------------------------
                    INCLUDES       devices/gameport.h
                                   devices/gameport.i
                                   devices/inputevent.h
                                   devices/inputevent.i

                    AUTODOCS       gameport.doc
```