lowlevel

COLLABORATORS			
	TITLE : lowlevel		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

lowlevel

1.1 lowlevel.doc

```
    AddKBInt()
    AddTimerInt()
    AddVBlankInt()
    ElapsedTime()
    GetKey()
    GetLanguageSelection()
    QueryKeys()
    ReadJoyPort()
    RemKBInt()
    RemTimerInt()
    RemVBlankInt()
    SetJoyPortAttrsA()
    StartTimerInt()
    StopTimerInt()
    SystemControlA()
```

1.2 lowlevel.library/AddKBInt

```
 NAME
AddKBInt -- adds a routine to the keyboard interrupt. (V40)

 SYNOPSIS
intHandle = AddKBInt(intRoutine, intData);
D0                   A0          A1

APTR AddKBInt(APTR, APTR);

 FUNCTION
This routine extends the functionality of the keyboard interrupt to
include intRoutine. Since this is an extention of the normal
keyboard interrupt all of the keyboard handshaking is handled. The
keyboard error codes are filtered out and not passed to intRoutine.

The routine is called whenever the user enters a key on the
keyboard.
```

```
The routine is called from within an interrupt, so normal
restrictions apply. The routine must preserve the following
registers: A2, A3, A4, A7, D2-D7. Other registers are
scratch, except for D0, which MUST BE SET TO 0 upon
exit. On entry to the routine, A1 holds 'intData' and A5
holds 'intRoutine', and D0 contains the rawkey code read
from the keyboard.

The routine is not called when a reset is received from the
keyboard.

This is a low level function that does not fit the normal Amiga
multitasking model. The interrupt installed will have no knowledge
of which window/screen currently has input focus.

If your program is to exit without reboot, you MUST call RemKBInt()
before exiting.

Only one interrupt routine may be added to the system.  ALWAYS
check the return value in case some other task has previously
used this function.

 INPUTS
intRoutine - the routine to invoke every vblank. This routine should
       be as short as possible to minimize its effect on overall
       system performance.
intData - data passed to the routine in register A1. If more than one
    long word of data is required this should be a pointer to
    a structure that contains the required data.

 RESULT
intHandle - a handle used to manipulate the interrupt, or NULL
      if it was not possible to attach the routine.

 SEE ALSO
RemKBInt()
```

1.3 lowlevel.library/AddTimerInt

```
 NAME
AddTimerInt -- adds an interrupt that is executed at regular
         intervals. (V40)

 SYNOPSIS
intHandle = AddTimerInt(intRoutine, intData);
D0                      A0          A1

APTR AddTimerInt(APTR, APTR);

 FUNCTION
Calling this routine causes the system to allocate a CIA timer
and set up 'intRoutine' to service any interrupts caused by the timer.
Although the timer is allocated it is neither running, nor enabled.
StartIntTimer() must be called to establish the time interval and
```

```
start the timer.

The routine is called from within an interrupt, so normal
restrictions apply. The routine must preserve the following
registers: A2, A3, A4, A7, D2-D7. Other registers are
scratch, except for D0, which MUST BE SET TO 0 upon
exit. On entry to the routine, A1 holds 'intData' and A5
holds 'intRoutine'.

Only a single CIA timer will be allocated by this routine. So this
routine may only be called once without an intervening call to
RemTimerInt().

The CIA timer used by this routine is not guaranteed to always be
the same. This routine utilizes the CIA resource and uses an
unallocated CIA timer.

If your program is to exit without reboot, you MUST match all
calls to this function with calls to RemTimerInt() before exiting.

Even if you only use the function once in your program; checking
the return value will make your program more tolerant for
mulititasking on the Amiga computer platforms.

 INPUTS
intRoutine - the routine to invoke upon timer interrupts. This routine
       should be as short as possible to minimize its effect on
       overall system performance.
intData - data passed to the routine in register A1. If more than one
    long word of data is required this should be a pointer to
    a structure that contains the required data.

 RESULT
intHandle - a handle used to manipulate the interrupt, or NULL
      if it was not possible to attach the routine.

 SEE ALSO
RemTimerInt(), StopTimerInt(), StartTimerInt()
```

1.4 lowlevel.library/AddVBlankInt

```
 NAME
AddVBlankInt -- adds a routine executed every vertical blank. (V40)

 SYNOPSIS
intHandle = AddVBlankInt(intRoutine, intData);
D0             a0   a1

APTR AddVBlankInt(APTR, APTR);

 FUNCTION
Lets you attach a routine to the system which will get called
everytime a vertical blanking interrupt occurs.

The routine is called from within an interrupt, so normal
```

```
restrictions apply. The routine must preserve the following
registers: A2, A3, A4, A7, D2-D7. Other registers are
scratch, except for D0, which MUST BE SET TO 0 upon
exit. On entry to the routine, A1 holds 'intData' and A5
holds 'intRoutine'.

If your program is to exit without reboot, you MUST call
RemVBlankInt() before exiting.

Only one interrupt routine may be added to the system.  ALWAYS check
the return value in case some other task has previously used this
function.

 INPUTS
intRoutine - the routine to invoke every vblank. This routine should
       be as short as possible to minimize its effect on overall
       system performance.
intData - data passed to the routine in register A1. If more than one
    long word of data is required this should be a pointer to
    a structure that contains the required data.

 RESULT
intHandle - a handle used to manipulate the interrupt, or NULL
      if it was not possible to attach the routine.

 SEE ALSO
RemVBlankInt()
```

1.5 lowlevel.library/ElapsedTime

```
 NAME
ElapsedTime -- returns the time elapsed since it was last called. (V40)

 SYNOPSIS
fractionalSeconds = ElapsedTime(context);
D0                              A0

ULONG ElapsedTime(struct EClockVal *);

 FUNCTION
This function utilizes the timer.device/ReadEClock() function to get
an accurate elapsed time value. Since the context needs to be
established the first call to this routine will return a nonsense
value.

The return value for this function only allows for sixteen bits
worth for the integer number of seconds and sixteen bits for the
factional number of seconds.

With sixteen bits worth of integer seconds this function can be
used to timer an interval up to about 16 hours. If the actual time
interval is larger this function will return this maximum value.

The sixteen bits for fractional seconds gives a resolution of
approximately 20 microseconds. However, it is not recomended
```

```
to expect this function to be accurate for a time interval of
less than 200 microseconds.

 INPUTS
context - pointer to an EClockVal structure. The first time you
    call this function, you should initialize the structure
    to 0s. You should then reuse the same structure for
    subsequent calls to this function, as this is how the
    elapsed time is calculated.

 RESULT
fractionalSeconds - The elapsed time as a fixed point 32-bit
                    number with the point fixed in the middle.
                    That is, the upper order sixteen bits represent
                    the number of seconds elapsed. The low order
                    sixteen bit represent the fractional number of
                    seconds elapsed. This value is limited to about
                    sixteen hours. Although this value is precise
                    to nearly 20 microseconds it is only accurate to
                    within 200 microseconds.

 WARNING
The first call to this function will return a non-sense value. Only
rely on its result starting with the second call.

 SEE ALSO
timer.device/ReadEClock()
```

1.6 lowlevel.library/GetKey

```
 NAME
GetKey -- returns the currently pressed rawkey code and qualifiers.
    (V40)

 SYNOPSIS
key = GetKey();
D0

ULONG GetKey(VOID);

 FUNCTION
This function returns the currently pressed non-qualifier key and
all pressed qualifiers.

This function is safe within an interrupt.

This is a low level function that does not fit the normal Amiga
multitasking model. The values returned by this function are
not modified by which window/screen currently has input focus.

 RESULT
key - key code for the last non-qualifier key pressed in the low
      order word. If no key is pressed this word will be FF. The
      upper order word contains the qualifiers which can be found
      within the long word as follows:
```

```
                Qualifier               Key
                LLKB_LSHIFT             Left Shift
                LLKB_RSHIFT             Rigt Shift
                LLKB_CAPSLOCK           Caps Lock
                LLKB_CONTROL            Control
                LLKB_LALT               Left Alt
                LLKB_RALT               Right Alt
                LLKB_LAMIGA             Left Amiga
                LLKB_RAMIGA             Right Amiga

 SEE ALSO
<libraries/lowlevel.h>
```

1.7 lowlevel.library/GetLanguageSelection

```
 NAME
GetLanguageSelection -- returns the current language selection. (V40)

 SYNOPSIS
language = GetLanguageSelection();
D0

ULONG GetLanguageSelection (VOID);

 FUNCTION
Determine what the user has specified as a language.

 RESULT
language - user specified language, or zero if none has yet been
     specified. See <libraries/lowlevel.h> for a definition
     of the currently supported language.

 SEE ALSO
<libraries/lowlevel.h>, locale.doc
```

1.8 lowlevel.library/QueryKeys

```
 NAME
QueryKeys -- return the states for a set of keys. (V40)

 SYNOPSIS
QueryKeys(queryArray, arraySize);
          A0          D1

VOID QueryKeys(struct KeyQuery *, UBYTE);

 FUNCTION
Scans the keyboard to determine which of the rawkey codes
listed in the QueryArray are currently pressed. The state for each
key is returned in the array.

This function may be invoked from within an interrupt, but the size
```

```
of QueryArray should be kept as small as possible.

This is a low level function that does not fit the normal Amiga
multitasking model. The values returned have no knowledge
of which window/screen currently has input focus.

 INPUTS
queryArray - an array of KeyQuery structures. The kq_KeyCode fields
       of these structures should be filled with the rawkey
       codes you wish to query about. Upon return from this
       function, the kq_Pressed field of these structures
       will be set to TRUE if the associated key is down,
             and FALSE if not.
arraySize - number of key code entries in queryArray

 SEE ALSO
<libraries/lowlevel.h>
```

1.9 lowlevel.library/ReadJoyPort

```
 NAME
ReadJoyPort -- return the state of the selected joy/mouse port. (V40)

 SYNOPSIS
portState = ReadJoyPort(portNumber);
D0                      D0

ULONG ReadJoyPort(ULONG);

 FUNCTION
This function is used to determine what device is attached to the
joy port and the current position/button state. The user may attach
a mouse, game controller, or joystick to the port and this function
will dynamically detect which device is attached and return the
appropriatly formatted portState.

To determine the type of controller that is attached, this function
clocks the game controller and/or interprets changes in the joy
port data. Valid clocked data from the game controller is immediately
detected. However, to accuratly determine if a mouse or joystick is
attached, several calls to this function are required along with some
movement at the joy port by the user.

This function always executes immediatly.

This is a low level single threaded function that does not fit the
normal Amiga multitasking model. Only one task can be executing
this routine at any time. All others will return immediately with
JP_TYPE_NOTAVAIL.

The nature of this routine is not meant to encourage non-multitasking
friendly programming practices like polling loops. If your task
is waiting for a transition to be returned use a WaitTOF() between
calls to minimize the total system impact.
```

```
When called the first time, for each port, this function attempts to
acquire certain system resources. In order to aquire these resources
this function MUST be called from a task, or a DOS process. If this
function fails to aquire the necessary resources, it will
return with JP_TYPE_NOTAVAIL. Once the resources are aquired (return
value other than JP_TYPE_NOTAVAIL) this function may be used in
interrupts.

 INPUTS
portNumber - port to read, in the range 0 to 3.

 RESULT
portState - bit map that identifies the device and the current
            state of that device. The format of the bit map is
            dependant on the type of device attached.

            The following constants from <libraries/lowlevel.h>
            are used to determine which device is attached and
            the state of that device.

            The type of device can be determined by applying
            the mask JP_TYPE_MASK to the return value and comparing
            the resultant value with the following:

                JP_TYPE_NOTAVAIL        port data unavailable
                JP_TYPE_GAMECTLR        game controller
                JP_TYPE_MOUSE           mouse
                JP_TYPE_JOYSTK          joystick
                JP_TYPE_UNKNOWN         unknown device

            If type = JP_TYPE_GAMECTLR the bit map of portState is:
                JPF_BUTTON_BLUE         Blue - Stop
                JPF_BUTTON_RED          Red - Select
                JPF_BUTTON_YELLOW       Yellow - Repeat
                JPF_BUTTON_GREEN        Green - Shuffle
                JPF_BUTTON_FORWARD      Charcoal - Forward
                JPF_BUTTON_REVERSE      Charcoal - Reverse
                JPF_BUTTON_PLAY         Grey - Play/Pause
                JPF_JOY_UP              Up
                JPF_JOY_DOWN            Down
                JPF_JOY_LEFT            Left
                JPF_JOY_RIGHT           Right

            If type = JP_TYPE_JOYSTK the bit map of portState is:
                JPF_BUTTON_BLUE         Right
                JPF_BUTTON_RED          Fire
                JPF_JOY_UP              Up
                JPF_JOY_DOWN            Down
                JPF_JOY_LEFT            Left
                JPF_JOY_RIGHT           Right

            If type = JP_TYPE_MOUSE the bit map of portState is:
                JPF_BUTTON_BLUE         Right mouse
                JPF_BUTTON_RED          Left mouse
                JPF_BUTTON_PLAY         Middle mouse
                JP_MVERT_MASK           Mask for vertical counter
                JP_MHORZ_MASK           Mask for horizontal counter
```

```
 SEE ALSO
SetJoyPortAttrs()
```

1.10 lowlevel.library/RemKBInt

```
NAME
RemKBInt -- remove a previously installed keyboard interrupt. (V40)

SYNOPSIS
RemKBInt(intHandle);
         A1

VOID RemKBInt(APTR);

FUNCTION
Remove a keyboard interrupt routine previously added with AddKBInt().

INPUTS
intHandle - handle obtained from AddKBInt(). This may be NULL,
      in which case this function does nothing.

SEE ALSO
AddKBInt()
```

1.11 lowlevel.library/RemTimerInt

```
 NAME
RemTimerInt -- remove a previously installed timer interrupt. (V40)

 SYNOPSIS
RemTimerInt(intHandle);
            A1

VOID RemTimerInt(APTR);

 FUNCTION
Removes a timer interrupt routine previously installed with
AddTimerInt.

 INPUTS
intHandle - handle obtained from AddTimerInt(). This may be NULL,
      in which case this function does nothing.

 SEE ALSO
AddTimerInt(), StopTimerInt(), StartTimerInt()
```

1.12 lowlevel.library/RemVBlankInt

```
 NAME
RemVBlankInt -- remove a previously installed vertical blank routine.
    (V40)

 SYNOPSIS
RemVBlankInt(intHandle);
       A1

VOID RemVBlankInt(APTR);

 FUNCTION
Removes a vertical blank interrupt routine previously added with
AddVBlankInt().

 INPUTS
intHandle - handle obtained from AddVBlankInt(). This may be NULL,
      in which case this function does nothing.

 SEE ALSO
AddVBlankInt()
```

1.13 lowlevel.library/SetJoyPortAttrsA

```
 NAME
SetJoyPortAttrsA -- change the attributes of a port.  (V40.27)
SetJoyPortAttrs -- varargs stub for SetJoyPortAttrsA().  (V40.27)

 SYNOPSIS
success = SetJoyPortAttrsA(portNumber, tagList);
D0                         D0          A1

BOOL SetJoyPortAttrsA(ULONG, struct TagItem *);

Success = SetJoyPortAttrs(portNumber, firstTag, ...);

BOOL SetJoyPortAttrs(Tag, ...);


 FUNCTION
This function allows modification of several attributes held by
ReadJoyPort() about both it's operation and the type of controller
currently plugged into the port.

ReadJoyPort()'s default behavior is to attempt to automatically
sense the type of controller plugged into any given port, when
asked to read that port. This behavior is beneficial, to allow
simple detection of the type of controller plugged into the port.
Unfortunately, rare cases are possible where extremely fine
mouse movements appear to be real joystick movements. Also, this
ability to auto-sense the controller type causes most reads to
take longer than if there were no auto-sensing.

SetJoyPortAttrs() is intended to provide for both of these cases.
It allows the programmer to notify ReadJoyPort() to stop spending
```

```
time attempting to sense which type of controller is in use -- and,
optionally, to force ReadJoyPort() into utilizing a certain
controller type.

 INPUTS
portNumber - the joyport in question (0-3).
tagList - a pointer to an array of tags providing parameters
          to SetJoyPortAttrs(); if NULL, the function
          will return TRUE, but do nothing.

 TAGS
SJA_Type (ULONG) - Sets the current controller type to the mouse,
    joystick, or game controller. Supply one of
    SJA_TYPE_GAMECTLR, SJA_TYPE_MOUSE, SJA_TYPE_JOYSTK,
    or SJA_TYPE_AUTOSENSE. If SJA_TYPE_AUTOSENSE is used,
    ReadJoyPort() will attempt to determine the type of
    controller plugged into the given port automatically.
    If one of the other types is used, ReadJoyPort() will
    make no attempt to read the given controller as
    anything other than the type specified. The default
    type is SJA_AUTOSENSE.

              Note -- if you set the type to anything other than
              auto-sense, it's your responsibility to return it
              to auto-sense mode before exiting.

SJA_Reinitialize (VOID) - Return a given port to it's initial state,
        forcing a port to deallocate any allocated resources;
              return the implied type to SJA_TYPE_AUTOSENSE.

 RESULT
success - TRUE if everything went according to plan, or FALSE upon
    failure

 SEE ALSO
ReadJoyPort(), <libraries/lowlevel.h>
```

1.14 lowlevel.library/StartTimerInt

```
 NAME
StartTimerInt -- start the timer associated with the timer interrupt.
     (V40)

 SYNOPSIS
StartTimerInt(intHandle, timeInterval, continuous);
              A1         D0            D1

VOID StartTimerInt(APTR, ULONG, BOOL);

 FUNCTION
This routine starts a stopped timer that is assocatied with a
timer interrupt created by AddTimerInt().

 INPUTS
intHandle - handle obtained from AddTimerInt().
```

```
timeInterval - number of micoseconds between interrupts. The
               maximum value allowed is 90,000. If higher values
               are passed there will be unexpected results.
continuous - FALSE for a one shot interrupt. TRUE for multiple
             interrupts.

 SEE ALSO
AddTimerInt(), RemTimerInt(), StopTimerInt()
```

1.15 lowlevel.library/StopTimerInt

```
 NAME
StopTimerInt -- stop the timer associated with the timer interrupt.
    (V40)

 SYNOPSIS
StopTimerInt(intHandle);
             A1

VOID StopTimerInt(APTR);

 FUNCTION
Stops the timer associated with the timer interrupt handle passed.
This is used to stop a continuous timer started by
StartTimerInt().

 INPUTS
intHandle - handle obtained from AddTimerInt().

 SEE ALSO
AddTimerInt(), RemTimerInt(), StartTimerInt()
```

1.16 lowlevel.library/SystemControlA

```
 NAME
SystemControlA - Method for selectively disabling OS features. (V40)
SystemControl - varargs stub for SystemControlA().

 SYNOPSIS
failTag = SystemControlA(tagList);
D0                       A1

ULONG SystemControlA(struct TagItem *);

failTag = SystemControl(firstTag, ...);

ULONG SystemControl(Tag, ...);

 FUNCTION
This function is used to alter the operation of the system. Some of
the alterations involve controlling what are normally regarded as
system resources. In order to minimize confusion only one task is
```

```
allowed to control any part of the system resources. This prevents
the possiblity of two tasks fighting, each controlling a part of the
system. If a tag is identified as task exclusive, it means that
only one task can hold (set to TRUE) that tag. If
another task attempts to set the same tag to TRUE, the call to
SystemControl() will fail.

It is important to remember that SystemControl() can fail.

This is a low level function and certain tags do not fit the normal
Amiga multitasking model.

 INPUTS
tagList - pointer to an array of tags listing the features of the
          system to be enabled/disabled.

 TAGS
SCON_TakeOverSys (BOOL)
  TRUE -  Takes over the CPU to ensure that a program gets every
    ounce of CPU time (with the exception of crucial
    interrupts). When in this mode, the CPU will belong
    completely to the program. Task switching will be
    disabled and the program will get all CPU cycles. This
    means any calls to the OS that involve multitasking in
    some way will not execute correctly. Other tasks will
    not run until this tag is used with FALSE. However,
    during a Wait() on a signal, multitasking will
    automatically be turned back on until the signal is
    received. Once received, multitasking will again be
    disabled and the CPU will be exclusive to the owning
    program.
  FALSE - Relinquishes the CPU and reenables multitasking.
    This tag is task exculsive. This tag nests. A task may
    take over the CPU several times before relinquishing
    it.

SCON_KillReq (BOOL)
       TRUE -   Disables system requesters.  These are the reasons for
    NOT disabling system requesters:

                    1- No calls in the program will cause a system
            requester.
                    2- The only thing that could cause a requester
            to appear is the lack of a CD in the drive and
            SCON_CDReboot is set to CDReboot_On, therefore a
            requester can't appear.
             3- The only disk I/O is via a CD with SCON_CDReboot
            set to CDReboot_On and/or nonvolatile.library.

        When requesters should not be disabled.
        GAME PROGRAMS:
        No DOS calls are used after loading; or SCON_CDReboot
    is CDReboot_On; and nonvolatile.library is used for
    loading and saving user data.

        This fits the above case since; After loading either
    DOS calls are not used fitting reason 1, or the game
```

```
    is accessing the CD and has SCON_CDReboot set to
    CDReboot_On fitting reason 2. The game accesses high
    scores, game position, etc through nonvolatile.library,
    fitting reason 3.

  FALSE - Enables requesters for the program.

    This tag nests. Tasks may disable requesters several times before
    enabling them.  However, there must be a matching number of calls.

SCON_CDReboot (ULONG)
  CDReboot_On - Ejecting the CD will cause a reboot of the
    system. Use this only if the program cannot deal with
    error conditions.
  CDReboot_Off - Ejecting the CD will not cause a reboot of the
        system. Use this if the program needs to insert CDs
    while running.
  CDReboot_Default - Restore the default reboot behavior for this
    system. This should be used upon exit, if this tag had
    been used to change the reboot behaviour. For the CD32
    this value is synonomous with CDReboot_On. For Amiga
    computers this value is synonomous with CDReboot_Off.

    Note that the default reboot behavior differs depending on the
    platform. If a program requires a specific behavior it must
    use this function to set the behavior. For example, a CD audio
    mixer would use this tag with the data CDReboot_Off. This will
    allow the changing of audio CDs on the game machine as well as
    Amiga computers.

    If, however, there is no error detection code at all this tag
    should be used with the data CDReboot_On.

    It is hoped that no program will require CDReboot_On. If all
    programs check for error condition and recover gracefully such a
    call should never be necessary. With the default behavior the
    CD32 will always reset on disk ejects, and programs
    run from Amiga computers will not reset. Thus, leaving the
    default will increase the market for a program to include both
    types of platforms.

    This tag does not nest.

SCON_StopInput (BOOL) - When TRUE, stops input.device from using any
    CPU cycles. Also prevents input.device from passing along any
    events from either the keyboard and/or port 0.

    This tag is task exclusive. This tag is NOT reversible.
    Attempting to reverse will result in confused/garbled input
    events.

SCON_AddCreateKeys (ULONG) - Starts creating rawkey codes for the
    joystick/game controller on the given unit. The unit value is
    checked for validity and must be either 0 or 1. Each different unit
    used results in some code added to the VBlank interrupt chain.
    This tag nests. The tag SCON_RemCreateKeys is used to undo this
    tag. Tasks may create rawkey codes several times before stopping
```

```
    them.

    Note that when operating in an Intuition window, the controller's
    blue button is the equivilent of the mouse menu button. Therefore,
    Intuition will be capturing most blue button events. If
    notificiation of these events is important, review the
    documentation for WFLG_RMBTRAP in the
    intuition.library/OpenWindow() autodoc.

SCON_RemCreateKeys (ULONG) - stops rawkey codes for the joystick/game
    controller on the given unit. The unit value is checked for
    validity and must be either 0 or 1.

 RESULT
failTag - zero if all tags succeeded. A non-zero return indicates a
    tag that has failed. It is possible that other tags may
          fail as well.

          If any tag fails there will be no change in the system due
          to other tags.

 SEE ALSO
<libraries/lowlevel.h>
```