F CONTENTS

```
NAME
     CloseAsync -- close an async io file.

SYNOPSIS
     result = CloseAsync(file);

     LONG CloseAsync(struct AsyncFile *);

FUNCTION
     Closes a file, flushing any pending writes.  Once this call has been
     made, the file can no longer be accessed.

INPUTS
     file - the file to close.  May be NULL.

RESULT
     result - < 0 for an error, >= 0 for success.  Indicates whether cl
              the file worked or not.  If the file was opened in read-mod
              then this call will always work.  In case of error,
              dos.library/IoErr() can give more information.

SEE ALSO
     OpenAsync, dos.library/Close()
```

```
   NAME
        OpenAsync -- open a file for asynchronous IO.

   SYNOPSIS
        file = OpenAsync(fileName, accessMode, bufferSize);

        struct AsyncFile OpenAsync(STRPTR, UBYTE, LONG);

   FUNCTION
        The named file is opened and an async file handle returned.  If the
        accessMode is MODE_READ, an existing file is opened for reading.
        If the value is MODE_WRITE, a new file is created for writing.  If
        a file of the same name already exists, it is first deleted.  If
        accessMode is MODE_APPEND, an existing file is prepared for writing.
        Data written is added to the end of the file.  If the file does not
        exists, it is created.

        'fileName' is a filename and CANNOT be a simple device such as NIL:, a
        window specification such as CON: or RAW:, or "*".

        'bufferSize' specifies the size of the IO buffer to use.  There are
        in fact two buffers allocated, each of roughly (bufferSize/2) bytes
        in size.  The actual buffer size use can vary slightly as the size
        is rounded to speed up DMA.

        If the file cannot be opened for any reason, the value returned
        will be NULL, and a secondary error code will be available by
        calling the routine dos.library/IoErr().

    INPUTS
        name - name of the file to open
        accessMode - one of MODE_READ, MODE_WRITE, or MODE_APPEND
        bufferSize - size of IO buffer to use.  8192 is recommended as it
                     provides very good performance for relatively little
                     memory.

    RESULTS
        file - an async file handle or NULL for failure.  You should not access
               the fields in the AsyncFile structure, these are private to the
               async IO routines.  In case of failure, dos.library/IoErr() can
               give more information.

    SEE ALSO
        CloseAsync(), dos.library/Open()

asyncio/ReadAsync                                            asyncio/ReadAsync

   NAME
```

```
        ReadAsync -- read bytes from an a

   SYNOPSIS
        actualLength = ReadAsync(file,buff

        LONG ReadAsync(struct AsyncFile *

   FUNCTION
        Read() reads bytes of information
        into the buffer given.  'numBytes
        the file.

        The value returned is the length
        So, when 'actualLength' is greate
        'actualLength' is the the number
        ReadAsync() will try to fill up yo
        of zero means that end-of-file has
        by a value of -1.

    INPUTS
        file - opened file to read, as obta
        buffer - buffer where to put byte
        numBytes - number of bytes to rea

    RESULT
        actualLength - actual number of b
                       case of error, dos
                       information.

    SEE ALSO
        OpenAsync(), CloseAsync(), WriteA
        dos.library/Read()

asyncio/ReadCharAsync

   NAME
        ReadCharAsync -- read a single by
ad bytes from an async file.

ReadAsync(file,buffer,numBytes);

struct AsyncFile *file, APTR buffer, LONG numBytes);

tes of information from an opened async file
 given.  'numBytes' is the number of bytes to read from

ned is the length of the information actually read.
lLength' is greater than zero, the value of
is the the number of characters read.  Usually
l try to fill up your buffer before returning.  A value
hat end-of-file has been reached.  Errors are indicated

e to read, as obtained from OpenAsync()
 where to put bytes read
er of bytes to read into buffer

actual number of bytes read, or -1 if an error.  In
case of error, dos.library/IoErr() can give more
information.

oseAsync(), WriteAsync(), ReadCharAsync(),

                                        asyncio/ReadCharAsync

- read a single byte from an async file.
```

```
   SYNOPSIS
        byte = ReadCharAsync(file);

        LONG ReadCharAsync(struct AsyncFile *file);

   FUNCTION
        This function reads a single byte from an async fil
        returned, or -1 if there was an error reading, or
        was reached.

   INPUTS
        file - opened file to read from, as obtained from Op

   RESULT
        byte - the byte read, or -1 if no byte was read.
               dos.library/IoErr() can give more informati

   SEE ALSO
        OpenAsync(), CloseAsync(), WriteCharAsync(), ReadA
        dos.library/Read()

asyncio/WriteAsync

   NAME
        WriteAsync -- write data to an async file.
```

```
ile *file);

e from an async file.  The byte is
error reading, or if the end-of-file

s obtained from OpenAsync()

o byte was read.  In case of error,
ive more information.

CharAsync(), ReadAsync()

                   asyncio/WriteAsync
```

```
   SYNOPSIS
        actualLength = WriteAsync(file,buffer,numBytes);

        LONG WriteAsync(struct AsyncFile *file, APTR buffer, LONG numBytes);

   FUNCTION
        WriteAsync() writes bytes of data to an opened async file.  'numBytes'
        indicates the number of bytes of data to be transferred.  'buffer'
        points to the data to write.  The value returned is the length of
        information actually written.  So, when 'numBytes' is greater than
        zero, the value of 'numBytes' is the number of characters written.
        Errors are indicated by a value of -1.

    INPUTS
        file - an opened file, as obtained from OpenAsync()
        buffer - address of data to write
        numBytes - number of bytes to write to the file

    RESULT
        actualLength - number of bytes written, or -1 if error.  In case
                       of error, dos.library/IoErr() can give more
                       information.

    SEE ALSO
        OpenAsync(), CloseAsync(), ReadAsync(), WriteCharAsync(),
        dos.library/Write

asyncio/WriteCharAsync                                asyncio/WriteCharAsync

   NAME
        WriteCharAsync -- write a single byte to an async file.

   SYNOPSIS
```

```
       result = WriteCharAsync(file,byte);

       LONG WriteCharAsync(struct AsyncFile *, UBYTE byte);

   FUNCTION
       This function write a single byte to an async file.

   INPUTS
       file - an opened async file, as obtained from OpenAsync()
       byte - byte of data to add to the file

   RESULT
       result - 1 if the byte was written, -1 if there was an error.  In
                case of error, dos.library/IoErr() can give more information.

   SEE ALSO
       OpenAsync(), CloseAsync(), ReadAsync(), WriteCharAsync(),
       dos.library/Write
```

```
/* ASyncIO.h - Header File for ASyncIO.c */

#ifndef ASYNCIO_H
#define ASYNCIO_H


/*****************************************************************************/
```

```
#include <exec/types.h>
#include <exec/ports.h>
#include <dos/dos.h>


/*******************************************


struct AsyncFile
{
    BPTR                  af_File;
    struct MsgPort       *af_Handler;
    APTR                  af_Offset;
    LONG                  af_BytesLeft;
    ULONG                 af_BufferSize;
    APTR                  af_Buffers[2];
    struct StandardPacket af_Packet;
    struct MsgPort        af_PacketPort;
    ULONG                 af_CurrentBuf;
    UBYTE                 af_PacketPending;
    UBYTE                 af_ReadMode;
};


/*******************************************


#define MODE_READ   0  /* read an existing fil
#define MODE_WRITE  1  /* create a new file, d
#define MODE_APPEND 2  /* append to end of ex


/*******************************************


struct AsyncFile *OpenAsync(STRPTR fileName,
LONG CloseAsync(struct AsyncFile *file);
LONG ReadAsync(struct AsyncFile *file, APTR b
LONG ReadCharAsync(struct AsyncFile *file);
LONG WriteAsync(struct AsyncFile *file, APTR
LONG WriteCharAsync(struct AsyncFile *file, c


/*******************************************


#endif /* ASYNCIO_H */
```

```
;/* ASyncIO.c - Execute me to compile with S
lc -cfist -v -j73 asyncio.c
quit
*/
#include <exec/types.h>
#include <exec/exec.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
```

```
*******************************************************/




*af_Handler;
 af_Offset;
 af_BytesLeft;
 af_BufferSize;
 af_Buffers[2];
 af_Packet;
 af_PacketPort;
 af_CurrentBuf;
 af_PacketPending;
 af_ReadMode;



*******************************************************/


read an existing file                                 */
create a new file, delete existing file if needed */
append to end of existing file, or create new     */


*******************************************************/


nc(STRPTR fileName, UBYTE mode, LONG bufferSize);
yncFile *file);
ncFile *file, APTR buf, LONG numBytes);
 AsyncFile *file);
yncFile *file, APTR buf, LONG numBytes);
t AsyncFile *file, char ch);


*******************************************************/
```

```
e to compile with SAS/C 5.10b
```

```
#include <stdio.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

#include "asyncio.h"

/*************************************************************

static VOID SendAsync(struct AsyncFile *file, APTR arg2)
{
    /* send out an async packet to the file system. */

    file->af_Packet.sp_Pkt.dp_Port = &file->af_PacketPort;
    file->af_Packet.sp_Pkt.dp_Arg2 = (LONG)arg2;
    PutMsg(file->af_Handler, &file->af_Packet.sp_Msg);
    file->af_PacketPending = TRUE;
}

/*************************************************************


static VOID WaitPacket(struct AsyncFile *file)
{
    /* This enables signalling when a packet comes back to the
    file->af_PacketPort.mp_Flags = PA_SIGNAL;

    /* Wait for the packet to come back, and remove it from the
     * list. Since we know no other packets can come in to the
     * safely use Remove() instead of GetMsg(). If other packet
     * we would have to use GetMsg(), which correctly arbitrate
     * a case
     */
    Remove((struct Node *)WaitPort(&file->af_PacketPort));

    /* set the port type back to PA_IGNORE so we won't be bothe
     * spurious signals
     */
    file->af_PacketPort.mp_Flags = PA_IGNORE;

    /* packet is no longer pending, we got it */
    file->af_PacketPending = FALSE;
}

/*************************************************************

struct AsyncFile *OpenAsync(STRPTR fileName, UBYTE mode, LONG bu
{
struct AsyncFile  *file;
struct FileHandle *fh;

    /* The buffer size is rounded to a multiple of 32 bytes. Th
     * DMA as fast as can be
     */

    bufferSize = (bufferSize + 31) & 0xffffffe0;

    /* now allocate the ASyncFile structure, as well as the rea
     * 15 bytes to the total size in order to allow for later d
     * alignement of the buffers
     */

    if (file = AllocVec(sizeof(struct AsyncFile) + bufferSize +
                        MEMF_ANY|MEMF_CLEAR))
    {
        if (mode == MODE_READ)
        {
            file->af_File     = Open(fileName,MODE_OLDFILE);
            file->af_ReadMode = TRUE;
        }
        else if (mode == MODE_WRITE)
        {
            file->af_File = Open(fileName,MODE_NEWFILE);
        }
        else if (mode == MODE_APPEND)
        {
            /* in append mode, we open for writing, and then se
             * end of the file. That way, the initial write wil
***********************************/

 APTR arg2)

_PacketPort;

***********************************/

 comes back to the port */

 remove it from the message
can come in to the port, we can
(). If other packets could come in,
correctly arbitrates access in such

PacketPort));

o we won't be bothered with

***********************************/

UBYTE mode, LONG bufferSize)

ple of 32 bytes. This will make

 as well as the read buffer. Add
o allow for later quad-longword

le) + bufferSize + 15,

MODE_OLDFILE);

_NEWFILE);

riting, and then seek to the
 initial write will happen at
```

```
         * the end of the file, thus extending it
         */

        if (file->af_File = Open(fileName,MODE_READWRITE))
        {
            if (Seek(file->af_File,0,OFFSET_END) < 0)
            {
                Close(file->af_File);
                file->af_File = NULL;
            }
        }
    }

    if (!file->af_File)
    {
        /* file didn't open, free stuff and leave */
        FreeVec(file);
        return(NULL);
    }

    /* initialize the ASyncFile structure. We do as much as we can here,
     * in order to avoid doing it in more critical sections
     *
     * Note how the two buffers used are quad-longword aligned. This helps
     * performance on 68040 systems with copyback cache. Aligning the data
     * avoids a nasty side-effect of the 040 caches on DMA. Not aligning
     * the data causes the device driver to have to do some magic to avoid
     * the cache problem. This magic will generally involve flushing the
     * CPU caches. This is very costly on an 040. Aligning things avoids
     * the need for magic, at the cost of at most 15 bytes of ram.
     */

    fh                  = BADDR(file->af_File);
    file->af_Handler    = fh->fh_Type;
    file->af_BufferSize = bufferSize / 2;
    file->af_Buffers[0] =
            (APTR)(((ULONG)file + sizeof(struct AsyncFile) + 15) & 0xfffffff0);
    file->af_Buffers[1] =
            (APTR)((ULONG)file->af_Buffers[0] + file->af_BufferSize);
    file->af_Offset     = file->af_Buffers[0];

    /* this is the port used to get the packets we send out back.
     * It is initialized to PA_IGNORE, which means that no signal is
     * generated when a message comes in to the port. The signal bit number
     * is initialized to SIGB_SINGLE, which is the special bit that can
     * be used for one-shot signalling. The signal will never be set,
     * since the port is of type PA_IGNORE. We'll change the type of the
     * port later on to PA_SIGNAL whenever we need to wait for a message
     * to come in.
     *
     * The trick used here avoids the need to allocate an extra signal bit
     * for the port. It is quite efficient.
     */

    file->af_PacketPort.mp_MsgList.lh_Head     =
            (struct Node *)&file->af_PacketPort.mp_MsgList.lh_Tail;
    file->af_PacketPort.mp_MsgList.lh_TailPred =
            (struct Node *)&file->af_PacketPort.mp_MsgList.lh_Head;
    file->af_PacketPort.mp_Node.ln_Type        = NT_MSGPORT;
    file->af_PacketPort.mp_Flags               = PA_IGNORE;
    file->af_PacketPort.mp_SigBit              = SIGB_SINGLE;
    file->af_PacketPort.mp_SigTask             = FindTask(NULL);

    file->af_Packet.sp_Pkt.dp_Link          = &file->af_Packet.sp_Msg;
    file->af_Packet.sp_Pkt.dp_Arg1          = fh->fh_Arg1;
    file->af_Packet.sp_Pkt.dp_Arg3          = file->af_BufferSize;
    file->af_Packet.sp_Msg.mn_Node.ln_Name  = (STRPTR)&file->af_Packet.sp_Pkt;
    file->af_Packet.sp_Msg.mn_Node.ln_Type  = NT_MESSAGE;
    file->af_Packet.sp_Msg.mn_Length        = sizeof(struct StandardPacket);

    if (mode == MODE_READ)
    {
        /* if we are in read mode, send out the first read packet to the
         * file system. While the application is getting ready to read
         * data, the file system will happily fill in this buffer with
         * DMA transfer, so that by the time the application needs the data,
         * it will be in the buffer waiting
         */
```

```
            file->af_Packet.sp_Pkt.dp_Type = ACTION_READ;
            if (file->af_Handler)
                SendAsync(file,file->af_Buffers[0]);
        }
        else
        {
            file->af_Packet.sp_Pkt.dp_Type = ACTION_WRITE;
            file->af_BytesLeft             = file->af_BufferSize;
        }
    }

    return(file);
}

/*****************************************************************************/

LONG CloseAsync(struct AsyncFile *file)
{
LONG result;
LONG result2;

    result = 0;
    if (file)
    {
        if (file->af_PacketPending)
            WaitPacket(file);

        result  = file->af_Packet.sp_Pkt.dp_Res1;
        result2 = file->af_Packet.sp_Pkt.dp_Res2;
        if (result >= 0)
        {
            if (!file->af_ReadMode)
            {
                /* this will flush out any pending data in the write buffer */
                result  = Write(file->af_File,
                          file->af_Buffers[file->af_CurrentBuf],
                          file->af_BufferSize - file->af_BytesLeft);
                result2 = IoErr();
            }
        }

        Close(file->af_File);
        FreeVec(file);

        SetIoErr(result2);
    }

    return(result);
}

/*****************************************************************************/

LONG ReadAsync(struct AsyncFile *file, APTR buf, LONG numBytes)
{
LONG totalBytes;
LONG bytesArrived;

    totalBytes = 0;

    /* if we need more bytes than there are in the current buffer, enter the
     * read loop
     */

    while (numBytes > file->af_BytesLeft)
    {
        /* this takes care of NIL: */
        if (!file->af_Handler)
            return(0);

        WaitPacket(file);

        bytesArrived = file->af_Packet.sp_Pkt.dp_Res1;
        if (bytesArrived <= 0)
        {
            /* error, get out of here */
            SetIoErr(file->af_Packet.sp_Pkt.dp_Res2);
            return(-1);
```

```
        }

        /* enable this section of code if yo
         * reads bigger than the buffer size
         */
#ifdef OPTIMIZE_BIG_READS
        if (numBytes > file->af_BytesLeft + b
        {
            if (file->af_BytesLeft)
            {
                CopyMem(file->af_Offset,buf,f

                numBytes   -= file->af_BytesL
                buf         = (APTR)((ULONG)
                totalBytes += file->af_BytesL
                file->af_BytesLeft = 0;
            }

            if (bytesArrived)
            {
                CopyMem(file->af_Buffers[file-

                numBytes   -= bytesArrived;
                buf         = (APTR)((ULONG)
                totalBytes += bytesArrived;
            }

            bytesArrived = Read(file->af_File

            if (bytesArrived <= 0)
                return(-1);

            SendAsync(file,file->af_Buffers[0]
            file->af_CurrentBuf = 0;
            file->af_BytesLeft  = 0;

            return(totalBytes + bytesArrived
        }
#endif

        if (file->af_BytesLeft)
        {
            CopyMem(file->af_Offset,buf,file->

            numBytes   -= file->af_BytesLeft,
            buf         = (APTR)((ULONG)buf
            totalBytes += file->af_BytesLeft,
        }

        /* ask that the buffer be filled */
        SendAsync(file,file->af_Buffers[1-file-

        file->af_Offset     = file->af_Buffers
        file->af_CurrentBuf = 1 - file->af_Cur
        file->af_BytesLeft  = bytesArrived;
    }

    if (numBytes)
    {
        CopyMem(file->af_Offset,buf,numBytes)
        file->af_BytesLeft -= numBytes;
        file->af_Offset     = (APTR)((ULONG)f
    }

    return (totalBytes + numBytes);
}

/*****************************************

LONG ReadCharAsync(struct AsyncFile *file)
{
char ch;

    if (file->af_BytesLeft)
    {
        /* if there is at least a byte left
         * directly. Also update all counter
         */
ection of code if you want special processing for
han the buffer size

le->af_BytesLeft + bytesArrived + file->af_BufferSize)



ile->af_Offset,buf,file->af_BytesLeft);

   -= file->af_BytesLeft;
    = (APTR)((ULONG)buf + file->af_BytesLeft);
es += file->af_BytesLeft;
BytesLeft = 0;




ile->af_Buffers[file->af_CurrentBuf],buf,bytesArrived);

   -= bytesArrived;
    = (APTR)((ULONG)buf + bytesArrived);
es += bytesArrived;


= Read(file->af_File,buf,numBytes);




,file->af_Buffers[0]);
entBuf = 0;
sLeft  = 0;

ytes + bytesArrived);





af_Offset,buf,file->af_BytesLeft);

= file->af_BytesLeft;
= (APTR)((ULONG)buf + file->af_BytesLeft);
= file->af_BytesLeft;


buffer be filled */
e->af_Buffers[1-file->af_CurrentBuf]);

   = file->af_Buffers[file->af_CurrentBuf];
uf = 1 - file->af_CurrentBuf;
t  = bytesArrived;




ffset,buf,numBytes);
t -= numBytes;
   = (APTR)((ULONG)file->af_Offset + numBytes);


numBytes);


**********************************************************/

t AsyncFile *file)





t least a byte left in the current buffer, get it
o update all counters
```

```
        ch = *(char *)file->af_Offset;
        file->af_BytesLeft--;
        file->af_Offset = (APTR)((ULONG)file->af_Offset + 1);

        return((LONG)ch);
    }

    /* there were no characters in the current buffer, so call
     * routine. This has the effect of sending a request to the
     * have the current buffer refilled. After that request is d
     * character is extracted for the alternate buffer, which a
     * becomes the "current" buffer
     */

    if (ReadAsync(file,&ch,1) > 0)
        return((LONG)ch);

    /* We couldn't read above, so fail */

    return(-1);
}

/****************************************************************

LONG WriteAsync(struct AsyncFile *file, APTR buf, LONG numBytes)
{
LONG totalBytes;

    totalBytes = 0;

    while (numBytes > file->af_BytesLeft)
    {
        /* this takes care of NIL: */
        if (!file->af_Handler)
        {
            file->af_Offset    = file->af_Buffers[file->af_Current
            file->af_BytesLeft = file->af_BufferSize;
            return(numBytes + totalBytes);
        }

        if (file->af_BytesLeft)
        {
            CopyMem(buf,file->af_Offset,numBytes);

            numBytes   -= file->af_BytesLeft;
            buf         = (APTR)((ULONG)buf + file->af_BytesLeft
            totalBytes += file->af_BytesLeft;
        }

        if (file->af_PacketPending)
        {
            WaitPacket(file);

            if (file->af_Packet.sp_Pkt.dp_Res1 <= 0)
            {
                /* an error occurred, leave */
                SetIoErr(file->af_Packet.sp_Pkt.dp_Res2);
                return(-1);
            }
        }

        /* send the current buffer out to disk */
        SendAsync(file,file->af_Buffers[file->af_CurrentBuf]);

        file->af_CurrentBuf   = 1 - file->af_CurrentBuf;
        file->af_Offset       = file->af_Buffers[file->af_CurrentB
        file->af_BytesLeft    = file->af_BufferSize;
    }

    if (numBytes)
    {
        CopyMem(buf,file->af_Offset,numBytes);
        file->af_BytesLeft -= numBytes;
        file->af_Offset     = (APTR)((ULONG)file->af_Offset + num
    }

    return (totalBytes + numBytes);
af_Offset + 1);

nt buffer, so call the main read
ng a request to the file system to
r that request is done, the
ate buffer, which at that point

**********************************/

buf, LONG numBytes)

ers[file->af_CurrentBuf];

+ file->af_BytesLeft);

kt.dp_Res2);

f_CurrentBuf]);

urrentBuf;
rs[file->af_CurrentBuf];

le->af_Offset + numBytes);
```

```
}

/*****************************************************************************/

LONG WriteCharAsync(struct AsyncFile *file, char ch)
{
    if (file->af_BytesLeft)
    {
        /* if there's any room left in the current buffer, directly write
         * the byte into it, updating counters and stuff.
         */

        *(char *)file->af_Offset = ch;
        file->af_BytesLeft--;
        file->af_Offset = (APTR)((ULONG)file->af_Offset + 1);

        /* one byte written */
        return(1);
    }

    /* there was no room in the current buffer, so call the main write
     * routine. This will effectively send the current buffer out to disk,
     * wait for the other buffer to come back, and then put the byte into
     * it.
     */

    return(WriteAsync(file,&ch,1));
}
```

```
;/* ASyncExample.c - Execute me to compile me with SAS/C 5.10b
LC -cfistq -v -y -j73 ASyncExample.c
Blink FROM LIB:c.o,ASyncExample.o TO ASyncExample LIBRARY
LIB:LC.lib,LIB:Amiga.lib,asyncio.o
quit ;*/

#include <exec/types.h>
#include <exec/exec.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <stdio.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
```

```
#include "asyncio.h"

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

VOID main(VOID)
{
struct AsyncFile *in;
LONG              num;
struct AsyncFile *out;

    if (in = OpenAsync("s:Startup-Sequence", MODE_READ, 8192))
    {
        if (out = OpenAsync("t:test_sync", MODE_WRITE, 8192))
        {
            while ((num = ReadCharAsync(in)) >= 0)
            {
                WriteCharAsync(out,num);
            }
            CloseAsync(out);
        }
        CloseAsync(in);
    }
}
```

♣