```
asyncio/SeekAsync                                             asyncio/SeekAsync

   NAME
        SeekAsync -- set the current position for reading or writing within
                     an async file.

   SYNOPSIS
        oldPosition = SeekAsync(file, position, mode);

        LONG SeekAsync(struct AsyncFile *, LONG, BYTE);

   FUNCTION
        SeekAsync() sets the read/write cursor for the file 'file' to the
        position 'position'. This position is used by the various read/write
        functions as the place to start reading or writing. The result is the
        current absolute position in the file, or -1 if an error occurs, in
        which case dos.library/IoErr() can be used to find more information.
        'mode' can be SEEK_START, SEEK_CURRENT or SEEK_END. It is used to
        specify the relative start position. For example, 20 from current
        is a position 20 bytes forward from current, -20 is 20 bytes back
        from current.

        To find out what the current position within a file is, simply seek
        zero from current.

    INPUTS
        file - an opened async file, as obtained from OpenAsync()
        position - the place where to move the read/write cursor
        mode - the mode for the position, one of SEEK_START, SEEK_CURRENT,
               or SEEK_END.

    RESULT
        oldPosition - the previous position of the read/write cursor, or -1
                      if an error occurs. In case of error, dos.library/IoErr()
                      can give more information.

    SEE ALSO
        OpenAsync(), CloseAsync(), ReadAsync(), WriteAsync(),
        dos.library/Seek()
```

ASyncIO.c

```
;/* ASyncIO.c - Execute me to compile with SAS/C 5.10b
sc data=near nominc strmer streq nostkchk saveds ign=73 AsyncIO.c
;lc -cfist -v -j73 asyncio.c
quit */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

#include <pragmas/exec_pragmas.h>
#include <pragmas/dos_pragmas.h>

#include "asyncio.h"

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

extern struct Library *DOSBase;
extern struct Library *SysBase;

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

/* this macro lets us long-align structures on the stack */
#define D_S(type,name) char a_##name[sizeof(type)+3]; \
                       type *name = (type *)((LONG)(a_##name+3) & ~3);
```

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

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

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

/* this function waits for a packet to come back from the file system. If no
 * packet is pending, state from the previous packet is returned. This ensures
 * that once an error occurs, it state is maintained for the rest of the life
 * of the file handle.
 *
 * This function also deals with IO errors, bringing up the needed DOS
 * requesters to let the user retry an operation or cancel it.
 */
static LONG WaitPacket(struct AsyncFile *file)
{
LONG bytes;

    if (file->af_PacketPending)
    {
        /* mark packet as no longer pending since we are going to get it */
        file->af_PacketPending = FALSE;

        while (TRUE)
        {
            /* This enables signalling when a packet comes back to the port */
            file->af_PacketPort.mp_Flags = PA_SIGNAL;

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

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

            bytes = file->af_Packet.sp_Pkt.dp_Res1;
            if (bytes >= 0)
            {
                /* packet didn't report an error, so bye... */
                return(bytes);
            }

            /* see if the user wants to try again... */
            if (ErrorReport(file->af_Packet.sp_Pkt.dp_Res2,
                            REPORT_STREAM,
                            file->af_File,NULL))
                return(-1);

            /* user wants to try again, resend the packet */
            SendPacket(file,file->af_Buffers[file->af_CurrentBuf]);
        }
    }

    /* last packet's error code, or 0 if packet was never sent */
    SetIoErr(file->af_Packet.sp_Pkt.dp_Res2);

    return(file->af_Packet.sp_Pkt.dp_Res1);
}

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

/* this function puts the packet back on the message list of our
 * message port.
 */
```

```
static VOID RequeuePacket(struct AsyncFile *file)
{
    AddHead(&file->af_PacketPort.mp_MsgList,&file->af_Packet.sp_Msg.mn_Node);
    file->af_PacketPending = TRUE;
}

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

/* this function records a failure from a synchronous DOS call into the
 * packet so that it gets picked up by the other IO routines in this module
 */
VOID RecordSyncFailure(struct AsyncFile *file)
{
    file->af_Packet.sp_Pkt.dp_Res1 = -1;
    file->af_Packet.sp_Pkt.dp_Res2 = IoErr();
}

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


struct AsyncFile *OpenAsync(const STRPTR fileName, UBYTE accessMode, LONG bufferSize)
{
struct AsyncFile  *file;
struct FileHandle *fh;
BPTR               handle;
BPTR               lock;
LONG               blockSize;
D_S(struct InfoData,infoData);

    handle = NULL;
    file   = NULL;
    lock   = NULL;

    if (accessMode == MODE_READ)
    {
        if (handle = Open(fileName,MODE_OLDFILE))
            lock = DupLockFromFH(handle);
    }
    else
    {
        if (accessMode == MODE_WRITE)
        {
            handle = Open(fileName,MODE_NEWFILE);
        }
        else if (accessMode == MODE_APPEND)
        {
            /* in append mode, we open for writing, and then seek to the
             * end of the file. That way, the initial write will happen at
             * the end of the file, thus extending it
             */

            if (handle = Open(fileName,MODE_READWRITE))
            {
                if (Seek(handle,0,OFFSET_END) < 0)
                {
                    Close(handle);
                    handle = NULL;
                }
            }
        }

        /* we want a lock on the same device as where the file is. We can't
         * use DupLockFromFH() for a write-mode file though. So we get sneaky
         * and get a lock on the parent of the file
         */
        if (handle)
            lock = ParentOfFH(handle);
    }

    if (handle)
    {
        /* if it was possible to obtain a lock on the same device as the
         * file we're working on, get the block size of that device and
         * round up our buffer size to be a multiple of the block size.
         * This maximizes DMA efficiency.
         */

        blockSize = 512;
        if (lock)
        {
            if (Info(lock,infoData))
            {
                blockSize = infoData->id_BytesPerBlock;
                bufferSize =
                    (((bufferSize + blockSize - 1) / blockSize) * blockSize) * 2;
            }
            UnLock(lock);
        }

        /* now allocate the ASyncFile structure, as well as the read buffers.
         * Add 15 bytes to the total size in order to allow for later
         * quad-longword alignement of the buffers
         */

        if (file = AllocVec(sizeof(struct AsyncFile) + bufferSize + 15,MEMF_ANY))
        {
            file->af_File      = handle;
            file->af_ReadMode  = (accessMode == MODE_READ);
            file->af_BlockSize = blockSize;

            /* 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];
            file->af_CurrentBuf    = 0;
            file->af_SeekOffset    = 0;
            file->af_PacketPending = FALSE;

            /* 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_Tail     = NULL;
            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_Pkt.dp_Res1          = 0;
            file->af_Packet.sp_Pkt.dp_Res2          = 0;
            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 (accessMode == 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 transfers, 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;
                file->af_BytesLeft             = 0;
                if (file->af_Handler)
                    SendPacket(file,file->af_Buffers[0]);
            }
            else
            {
                file->af_Packet.sp_Pkt.dp_Type = ACTION_WRITE;
                file->af_BytesLeft             = file->af_BufferSize;
            }
        }
        else
        {
            Close(handle);
        }
    }

    return(file);
}

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

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

    if (file)
    {
        result = WaitPacket(file);
        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);
            }
        }

        Close(file->af_File);
        FreeVec(file);
    }
    else
    {
        SetIoErr(ERROR_INVALID_LOCK);
        result = -1;
    }

    return(result);
}

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

LONG ReadAsync(struct AsyncFile *file, APTR buffer, 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)
    {
        /* drain buffer */
        CopyMem(file->af_Offset,buffer,file->af_BytesLeft);

        numBytes           -= file->af_BytesLeft;
        buffer              = (APTR)((ULONG)buffer + file->af_BytesLeft);
        totalBytes         += file->af_BytesLeft;
        file->af_BytesLeft  = 0;

        bytesArrived = WaitPacket(file);
        if (bytesArrived <= 0)
        {
            if (bytesArrived == 0)
                return(totalBytes);

            return(-1);
        }

        /* ask that the buffer be filled */
        SendPacket(file,file->af_Buffers[1-file->af_CurrentBuf]);

        if (file->af_SeekOffset > bytesArrived)
            file->af_SeekOffset = bytesArrived;

        file->af_Offset      = (APTR)((ULONG)file->af_Buffers[file->af_CurrentBuf]
                                + file->af_SeekOffset);
        file->af_CurrentBuf  = 1 - file->af_CurrentBuf;
        file->af_BytesLeft   = bytesArrived - file->af_SeekOffset;
        file->af_SeekOffset  = 0;
    }

    CopyMem(file->af_Offset,buffer,numBytes);
    file->af_BytesLeft -= numBytes;
    file->af_Offset     = (APTR)((ULONG)file->af_Offset + numBytes);

    return (totalBytes + numBytes);
}
/*****************************************************************************/

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

    if (file->af_BytesLeft)
    {
        /* if there is at least a byte left in the current buffer, get it
         * directly. Also 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 the main read
     * routine. This has the effect of sending a request to the file system to
     * have the current buffer refilled. After that request is done, the
     * character is extracted for the alternate buffer, which at that point
     * 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 buffer, 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[0];
            file->af_BytesLeft = file->af_BufferSize;
            return(numBytes);
        }

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

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

        if (WaitPacket(file) < 0)
            return(-1);

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

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

    CopyMem(buffer,file->af_Offset,numBytes);
    file->af_BytesLeft -= numBytes;
    file->af_Offset     = (APTR)((ULONG)file->af_Offset + numBytes);

    return (totalBytes + numBytes);
}

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

LONG WriteCharAsync(struct AsyncFile *file, UBYTE 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.
         */

        *(UBYTE *)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));
}

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

LONG SeekAsync(struct AsyncFile *file, LONG position, BYTE mode)
{
LONG  current, target;
LONG  minBuf, maxBuf;
LONG  bytesArrived;
LONG  diff;
LONG  filePos;
LONG  roundTarget;
D_S(struct FileInfoBlock,fib);

    bytesArrived = WaitPacket(file);

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

    if (file->af_ReadMode)
    {
        /* figure out what the actual file position is */
        filePos = Seek(file->af_File,OFFSET_CURRENT,0);
        if (filePos < 0)
        {
            RecordSyncFailure(file);
            return(-1);
        }

        /* figure out what the caller's file position is */
        current = filePos - (file->af_BytesLeft+bytesArrived);

        /* figure out the absolute offset within the file where we must seek to */
        if (mode == MODE_CURRENT)
        {
            target = current + position;
        }
        else if (mode == MODE_START)
        {
            target = position;
        }
        else /* if (mode == MODE_END) */
        {
            if (!ExamineFH(file->af_File,fib))
            {
                RecordSyncFailure(file);
                return(-1);
            }

            target = fib->fib_Size + position;
        }

        /* figure out what range of the file is currently in our buffers */
        minBuf = current - (LONG)((ULONG)file->af_Offset -
                     (ULONG)file->af_Buffers[1 - file->af_CurrentBuf]);
        maxBuf = current + file->af_BytesLeft
                     + bytesArrived;  /* WARNING: this is one too big */

        diff = target - current;

        if ((target < minBuf) || (target >= maxBuf))
        {
            /* the target seek location isn't currently in our buffers, so
             * move the actual file pointer to the desired location, and then
             * restart the async read thing...
             */

            /* this is to keep our file reading block-aligned on the device.
             * block-aligned reads are generally quite a bit faster, so it is
             * worth the trouble to keep things aligned
             */
            roundTarget = (target / file->af_BlockSize) * file->af_BlockSize;

            if (Seek(file->af_File,roundTarget-filePos,OFFSET_CURRENT) < 0)
            {
                RecordSyncFailure(file);
                return(-1);
            }

            SendPacket(file,file->af_Buffers[0]);

            file->af_SeekOffset = target-roundTarget;
            file->af_BytesLeft  = 0;
            file->af_CurrentBuf = 0;
        }
        else if ((target < current) || (diff <= file->af_BytesLeft))
        {
            /* one of the two following things is true:
             *
```

```
         * 1. The target seek location is within the current read buffer,
         * but before the current location within the buffer. Move back
         * within the buffer and pretend we never got the pending packet,
         * just to make life easier, and faster, in the read routine.
         *
         * 2. The target seek location is ahead within the current
         * read buffer. Advance to that location. As above, pretend to
         * have never received the pending packet.
         */

        RequeuePacket(file);

        file->af_BytesLeft -= diff;
        file->af_Offset     = (APTR)((ULONG)file->af_Offset + diff);
    }
    else
    {
        /* at this point, we know the target seek location is within
         * the buffer filled in by the packet that we just received
         * at the start of this function. Throw away all the bytes in the
         * current buffer, send a packet out to get the async thing going
         * again, readjust buffer pointers to the seek location, and return
         * with a grin on your face... :-)
         */

        diff -= file->af_BytesLeft;

        SendPacket(file,file->af_Buffers[1-file->af_CurrentBuf]);

        file->af_Offset
            = (APTR)((ULONG)file->af_Buffers[file->af_CurrentBuf] + diff);
        file->af_CurrentBuf = 1 - file->af_CurrentBuf;
        file->af_BytesLeft  = bytesArrived - diff;
    }
  }
  else
  {
    if (Write(file->af_File,
              file->af_Buffers[file->af_CurrentBuf],
              file->af_BufferSize - file->af_BytesLeft) < 0)
    {
        RecordSyncFailure(file);
        return(-1);
    }

    /* this will unfortunately generally result in non block-aligned file
     * access. We could be sneaky and try to resync our file pos at a
     * later time, but we won't bother. Seeking in write-only files is
     * relatively rare (except when writing IFF files with unknown chunk
     * sizes, where the chunk size has to be written after the chunk data)
     */

    current = Seek(file->af_File,position,mode);

    if (current < 0)
    {
        RecordSyncFailure(file);
        return(-1);
    }

    file->af_BytesLeft  = file->af_BufferSize;
    file->af_CurrentBuf = 0;
  }

  return(current);
}
```

AsyncIO.h

```
#ifndef ASYNCIO_H
#define ASYNCIO_H

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

#ifndef EXEC_TYPES_H
#include <exec/types.h>
#endif

#ifndef EXEC_PORTS_H
#include <exec/ports.h>
#endif

#ifndef DOS_DOS_H
#include <dos/dos.h>
#endif

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

/* This structure is public only by necessity, don't muck with it yourself, or
 * you're looking for trouble
 */
struct AsyncFile
{
    BPTR                  af_File;
    ULONG                 af_BlockSize;
    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;
    ULONG                 af_SeekOffset;
    UBYTE                 af_PacketPending;
    UBYTE                 af_ReadMode;
};


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


#define MODE_READ   0  /* read an existing file                             */
#define MODE_WRITE  1  /* create a new file, delete existing file if needed */
#define MODE_APPEND 2  /* append to end of existing file, or create new     */

#define MODE_START   -1   /* relative to start of file         */
#define MODE_CURRENT  0   /* relative to current file position */
#define MODE_END      1   /* relative to end of file         */


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


struct AsyncFile *OpenAsync(const STRPTR fileName, UBYTE accessMode, LONG bufferSize);
LONG CloseAsync(struct AsyncFile *file);
LONG ReadAsync(struct AsyncFile *file, APTR buffer, LONG numBytes);
LONG ReadCharAsync(struct AsyncFile *file);
LONG WriteAsync(struct AsyncFile *file, APTR buffer, LONG numBytes);
LONG WriteCharAsync(struct AsyncFile *file, UBYTE ch);
LONG SeekAsync(struct AsyncFile *file, LONG position, BYTE mode);

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

#endif /* ASYNCIO_H */
```