

# **Async**

Michael Zucchi

Copyright © 1993 Michael Zucchi, All right reserved

<b>COLLABORATORS</b>
----------------------

	<i>TITLE :</i> Async		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Michael Zucchi	December 11, 2024	

<b>REVISION HISTORY</b>
-------------------------

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Async</b>	<b>1</b>
1.1	Async.guide . . . . .	1
1.2	Module overview . . . . .	2
1.3	The guy who wrote it . . . . .	2
1.4	as_open . . . . .	3
1.5	as_close . . . . .	3
1.6	as_read . . . . .	4
1.7	as_fgets . . . . .	5
1.8	as_fgetc . . . . .	5
1.9	as_nextbuffer . . . . .	6
1.10	Information about the examples . . . . .	6

# Chapter 1

## Async

### 1.1 Async.guide

Asynchronous file reading module for AmigaE2.5+

© 1993 Michael Zucchi  
All Rights Reserved

This document describes the usage of a suite of asynchronous file reading routines designed for the AmigaE language. The interface is designed to follow the V36 dos.library calls as closely as possible.

The following sections are available:

    Overview          some of the ideas behind the module

General functions

as\_Open()            to open a file  
as\_Close()          to close a file  
as\_Read()            reading from the file

High level functions

as\_FGetS()          reading text lines  
as\_FGetC()          reading character by character

Low level functions

as\_NextBuffer()      accessing input buffers direct

Examples

NOTE: This module requires Workbench 2.0 (V36) or higher! Please make sure that this version of the system libraries is present before using these functions.

## 1.2 Module overview

Just what is meant by 'asynchronous i/o'?

When most programs use `dos.library` to read/write files, they simply call `Read()` or `Write()`. What happens then is that these functions examine the filehandle passed to them for information about the handler that handles the file, and creates a dos 'packet' out of this information (see `dos/dosexens.m` to see what a packet looks like). This packet is then sent via the standard message passing system to the handler handling the file. The dos function calls then wait for a reply to the request via the process's message port `pr_MessagePort` - i.e. they wait until the filesystem and handler have retrieved the information before returning (incidentally, the fact that the replies come in through `pr_MessagePort` is the reason dos cannot be called from a standard 'task'). With a slow i.o device (e.g. floppy disk) all of this waiting can mean the cpu is sitting idle a lot of the time waiting for data to come in.

How do you fix this less than ideal situation? Its quite simple. You can create your own packets and send these packets direct to dos. This way, a custom reply port can be set up for the packets, and requests for reads (or writes) can be sent out immediately, and the data read when the packets are returned. If something needs to be done while the filesystem is fetching this data, then your program can get it done - without having to wait.

This is basically what `async.m` does. Currently only reading is supported, but writing will be added in the future, along with utility functions like `Seek()` etc. I got the idea from some code i got off the local BBS, something from one of the cool guys at Commodore i think.

## 1.3 The guy who wrote it

I wrote this code some time ago, mainly for a multi-threaded directory utility i've been working on from time to time. I found it so handy for adding just that extra bit of performance to just about everything i wrote that i thought other people might find it useful too.

Presently, i study 'from time to time' (:-) in order to obtain a Computer Systems Engineering degree from the Univerity Of South Australia. I'm 'Zed' of FRONTIER in my anti-os hours.

I can be contacted in the following ways:

Internet email:

9107047w@lux.levels.unisa.edu.au  
till the end of '94 at least - reliable

'Real Mode' (tm) mail:

Michael Zucchi  
PO BOX 824  
Waikerie

---

South Australia 5330  
slow, but very reliable - till mum sells the house :)  
  
Michael Zucchi  
110 Dunrobin Rd  
Warradale  
South Australia 5046  
to my door - till i move (?)

## 1.4 as\_open

async.m/as\_Open

async.m/as\_Open

### SYNTAX

```
file := as_Open( name:PTR TO CHAR,  
                mode:LONG,  
                count:LONG,  
                size:LONG );
```

### PURPOSE

Opens an asynchronous file, and returns a pointer to a (private) file handle. When called, packets will be sent to the appropriate handler to fill all buffers, and the return will call immediately.

### INPUTS

name A string, describing the name of the file to open  
mode Same as mode in dos.library/Open. Must be MODE\_OLDFILE for now.  
count Number of buffers to allocate. 3 works very well.  
size The size of each buffer to allocate. Above 5000 works well, must be a multiple of 4.

### OUTPUTS

file A pointer to a filehandle that may be passed to the other async functions.

### NOTES

No sanity checking is done on any of the input values. Use reasonable values for everything.  
The filehandle returned by as\_Open() is NOT compatible with normal dos filehandles, and system calls!

### SEE ALSO

as\_Close(), as\_NextBuffer(), as\_Read(), as\_FGetS(), as\_FGetC()

## 1.5 as\_close

async.m/as\_Close

async.m/as\_Close

### SYNTAX

---

```
as_Close( file:LONG );
```

**PURPOSE**

Closes the file, free's all memory buffers and cleans up all outstanding packets. This call may be made as any time on a valid async filehandle.

**INPUTS**

file valid filehandle from as\_Open(), or NIL in which case nothing happens.

**OUTPUTS****NOTES****SEE ALSO**

as\_Open()

## 1.6 as\_read

async.m/as\_Read

async.m/as\_Read

**SYNTAX**

```
bytes := as_Read( file:LONG,  
                  buffer:PTR TO CHAR,  
                  number:LONG );
```

**PURPOSE**

as\_Read reads a number of bytes ('number') into the buffer specified by 'buffer', from the async file 'file'.

The number of bytes actually read in is indicated by the return value. A return of zero indicates end of file, and errors are flagged by a return value of -1.

**INPUTS**

file Only a valid filehandle from as\_Open() is allowed.  
buffer A pointer to at least 'number' bytes of memory to store the data. May be arbitrarily aligned.  
number Specifies the number of bytes to read. number=0 is ignored.

**OUTPUTS**

bytes The number of bytes actually stored in 'buffer'. A value of zero indicates end of file, and -1 that a file error has occurred, check IoErr() for detail.

**NOTES****SEE ALSO**

as\_Open(), as\_Close(), as\_NextBuffer(), as\_FGetS(), as\_FGetC()



## 1.7 as\_fgets

async.m/as\_FGetS

async.m/as\_FGetS

### SYNTAX

```
buffer := as_FGetS( file:LONG,  
                  buffer:PTR TO CHAR,  
                  number:LONG );
```

### PURPOSE

Reads upto 'size' bytes from the file 'file' into the buffer pointed to by the buffer parameter. Stops reading at end of file or once a NEWLINE (\$0a) character is encountered. Returns a pointer to that buffer or NIL on end of file or error.

The string stored in the buffer is NULL terminated.

### INPUTS

file A valid filehandle from as\_Open().  
buffer A pointer to at least 'number' bytes of memory to store the data. May be arbitrarily aligned.  
number Specifies the number of bytes to read, at maximum. This MUST be >2.

### OUTPUTS

buffer Same as 'buffer' passed as an input, or NIL on end of file or file error.

### NOTES

If the line is too long to fit, the input stream is not skipped till the next linefeed.

### SEE ALSO

as\_Open(), as\_Close(), as\_Read(), as\_NextBuffer(), as\_FGetC()

## 1.8 as\_fgetc

async.m/as\_FGetC

async.m/as\_FGetC

### SYNTAX

```
char := as_FGetC( file:LONG );
```

### PURPOSE

Reads the next character from the input file. Returns -1 on error or end of file. The character is an unsigned 32 bit quantity.

### INPUTS

file A valid filehandle from as\_Open().

### OUTPUTS

char The next available byte from the input stream, or -1 on error.

---

## NOTES

This call is about as efficient as possible.

## SEE ALSO

`as_Open()`, `as_Close()`, `as_Read()`, `as_NextBuffer()`, `as_FGetS()`

## 1.9 as\_nextbuffer

async.m/as\_NextBuffer

async.m/as\_NextBuffer

## SYNTAX

```
buffer,valid := as_NextBuffer( file:LONG );
```

## PURPOSE

Returns the next available data buffer in the file. If end of file has not yet been reached, and a buffer has now been made free, then another read request is sent to the filesystem, in an asynchronous manner.

## INPUTS

file A valid filehandle from `as_Open()`.

## OUTPUTS

buffer The address of the internal data buffer, 0 for end of file, or -1 on a file error.  
valid Number of valid bytes in the buffer. This will be the same as the size of each buffer as specified when the file was opened, unless it is the last buffer being read.

## NOTES

The call is NOT compatible with any of the other reading functions. If you call those functions, you must NOT call this function, and visa-versa. It is a low level function which both of the other read functions make use of directly, and should only be used (esclusively) where extra performance/lower memory use is required.

## SEE ALSO

`as_Open()`, `as_Close()`, `as_Read()`, `as_FGetS()`, `as_FGetC()`

## 1.10 Information about the examples

Included with this package are a few examples of using this module.

## typedef

This is a simple example demonstrating the use of the `as_FGetS()` call. It simply types a specified file to the current shell - quite a bit faster than `c:type` does.

usage:

---

```
typedef [Name] <filename>
```

### PlaySamp

This is a non-trivial example of the `as_Read()` function. It is a complete 'raw sample' player that can be used to play ANY sized sample from disk.

usage:

```
PlaySamp [Name] <file1> [<file2> ... ] RATE <rate>
```

### histogram

A simple example of using the `as_NextBuffer()` command. It counts the occurrannces of each byte in a file, and produces a report when done.

usage:

```
histogram [Name] <filename>
```

Coding a more useful example for `as_NextBuffer()` requires a bit more work than i have time for :)