ACEReference.doc

COLLABORATORS			
	TITLE : ACEReference.doc		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 30, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

ACEReference.doc

1.1 Main Menu

```
+----------+
| ACE v2.3 |
+----------+
```


1.2 Introduction

Introduction

1.3 Getting started

Getting started

1.4 A hitch-hiker's guide to ACE

```
                A hitch-hiker's guide to ACE
                ----------------------------

General comments
The preprocessor and #include files
Data types, expressions, constants
Precedence of operators
Indirection operators
Identifiers
Files
Command line and Workbench arguments
Subprograms
Structures
Shared library function calls
Machine code calls
External references
Creating & using ACE subprogram modules
Windows
Screens
Gadgets
Menus
Requesters
Turtle Graphics
Loading & displaying IFF pictures
Sound
Event trapping
Interprocess Communication
Error handling
Notes for assembly programmers
Limitations
Known bugs
```

1.5 Stop bits

```
                        Stop bits
                        ---------

 Future versions
 A note to PD libraries and reviewers
 Disclaimer
 References
 Contacting the author
 ACE discussion list and FTP site
 Final word
```

1.6 What is ACE?

```
What is ACE?
------------
 AmigaBASIC Compiler with Extras?
```

```
 A Creative Environment?
 A Compiler for Everyone?
 A Cool Enterprise?
 Automatic Computing Engine (ala Alan Turing)?
 Dr Who's last companion?

Okay, seriously...

ACE is a freely distributable, recursive descent, peephole-optimising
Amiga BASIC compiler which produces A68K-compatible assembly source code.
ACE runs under Wb 1.3, 2.x and up, as do the executables it produces. ACE
will run in 512K, but more than this is required for programs beyond about
250 lines.

ACE supports a large subset of AmigaBASIC. It also provides a variety of
commands, functions and features not found in AmigaBASIC.

In many cases, ACE programs produce results which are similar or identical
to programs written in AmigaBASIC.

Any differences between the two are discussed in this document and the
command and function reference.

The following files constitute a complete ACE package:

        executables
        -----------
  bas   - A shell script which automates
          the production of ACE executables
        (1.3 and 2.x/3.x versions).
  app   - A simple preprocessor.
  ace   - The BASIC compiler.
  a68k    - Charlie Gibbs' 68000 assembler.
  blink   - The Software Distillery's linker.
  muchmore  - The file viewer by Fridtjof Siebert & Christian
        Stiens (1.3 and 2.x/3.x versions).

  documents
  ---------
  ace.doc/guide   - The document you are reading which describes ACE.
  ref.doc/guide   - A command and function reference for ACE.
  example.guide - Examples of ACE command and function usage.
  history         - A history of ACE's development.
  a68k.doc        - Documentation for the assembler.
  blink.doc       - Documentation for the linker.

  scanned libraries
  -----------------
  ami.lib   - A freely distributable version of amiga.lib.
  startup.lib - A library of routines needed at the start and
        end of an ACE program run.
  db.lib    - A library of assorted routines used by ACE
        programs.

  other stuff
  -----------
  includes  - Some useful ACE include files.
```

```
  icons    - ACE tool and document icons.
  utils    - Miscellaneous utilities.
  examples  - Example programs which demonstrate many of ACE's
        capabilities.
  AIDE     - A graphical front-end for ACE.
```

With one exception (see the discussion of file requesters under Wb 1.3 in the "Requesters" section) ACE programs do not require any special run-time shared libraries, so the executables which the compiler produces (via the assembler and linker) are completely portable, requiring only the standard Amiga shared libraries in your LIBS: directory. The three ".lib" files mentioned above are scanned libraries and code from these is included at link time.

ACE is written in C (Sozobon's ZC v1.01), while db.lib and startup.lib are written in assembler (~50%) and C. I may eventually switch to SAS C.

A68K and Blink are used to assemble and link the code produced by ACE.

The MicroEmacs (v1.3 & v2.1) editor has been used throughout every stage of ACE's development. It works for me.

The complete ACE package may be freely distributed.

1.7 Who is it for?

```
Who is it for?
--------------
```

ACE is intended for anyone who already knows BASIC and wants one or more of the following:

- Faster program execution.

- Independence from the BASIC interpreter, ie: standalone programs which are runnable from the CLI/Shell and Workbench.

- Extra commands, functions and features: turtle graphics, command-line (and Workbench) arguments, recursion, SUBs with return values, external references, named constants, structures, include files, better WAVE command, gadgets, requesters, ability to create subprogram libraries, interprocess communication, etc.

Maybe you don't wish to learn another high-level language, or perhaps you already use C or assembler but prefer to use BASIC for some tasks while still having the power of a compiled language.

ACE is a general purpose language so in theory at least, it can be applied to any programming task you like. In practice however, I find ACE to be most useful for writing small to medium sized programs where speed is important but so is ease of programming.

ACE is also a useful prototyping language. It allows you to get something up and running quickly to test an idea. You may later decide to re-code in C or assembler, or you may just add some polish to the existing ACE program.

```
The latter is becoming more feasible as ACE matures.

Here are some programs that have been written with ACE:

  - Shell utilities, eg: basic calculator (ACE:prgs/ShellUtils/bc.b).
  - An 8SVX sound sample player.
  - Fractal graphics programs.
  - Neural networks.
  - Astronomy programs, eg: galaxy collision simulator,
    Jovian satellite position predictor, Messier object database.
  - An Integrated Development Environment (AIDE - see separate
    archive).
  - A home security/wakeup system.
  - A program which matches section headings to line and page
    numbers to create the document you are now reading from an
    unformatted version of it.
  - A Workbench calculator with a difference.
  - A SpiroGraph simulator (Chuck Kenny).
  - A serial Stratego game (Dan Oberlin).
  - A variety of communications programs.
```

1.8 Installation

```
Installation
------------
You will need to open a shell to install ACE. Installation consists of:

  - Extracting the four archives found in the single supplied ACE
    archive.

  - Adding a few commands to your s:user-startup (Wb 2.x/3.x) or
    s:startup-sequence (Wb 1.3) file.

The distribution archive contains:

  - MAIN.lha    (main ACE files with a few examples)
  - DOCS.lha    (documentation for ACE,A68K and Blink)
  - PRGS.lha    (more example programs)
  - AIDE.lha    (a graphical front-end for ACE)

All four will fit onto a single floppy disk in their compressed form.

So, extract these four archives into RAM:, into a temporary hard disk
directory or onto an empty floppy disk with:

  lha x <ACE-archive>
eg.
  lha x ACE23

You must now extract the files from each archive with:

  lha -a x <archive>
eg.
  lha -a x MAIN
```

```
The "-a" switch preserves file attributes (eg. the "s" bit on shell scripts)
in the archive.

If you have a hard disk, just extract all four archives into a directory
created for ACE (eg: sys:ACE). Extract MAIN.lha first. This will set up the
directory structure and main ACE files.

The extraction of MAIN.lha also creates three subdirectories for the last
three archives. Extract DOCS.lha, PRGS.lha and AIDE.lha into the docs, prgs
and AIDE subdirectories.

If you are using a floppy-only system, extract MAIN.lha onto one disk and
the last three archives onto other disks to suit yourself. The disk on which
the main ACE files reside is the volume to which the shell commands discussed
below refer.

Next, add the following lines to your user-startup or startup-sequence
script:

  assign ACE: <volume or directory>
  path ACE:bin add

where <volume or directory> is the name of the disk or directory where
the main ACE files now reside (eg: assign ACE: sys:ACE).

In addition, you need to add three more statements to your startup-sequence
or user-startup script:

  assign ACElib:     ACE:lib  ; bas finds scanned libraries here.
  assign ACEbmaps:   ACE:bmaps  ; ace looks here for .bmap files.
  assign ACEinclude: ACE:include  ; app uses this for include files.

Now reboot your Amiga to let the above path and assign commands take effect.

That's it!

** Note: As an alternative to these startup script modifications, you can
use the "ACEsetup" script to be found in MAIN.lha. Read the comments at
the top of that script for usage details.
```

1.9 Using the compiler

```
Using the compiler
------------------
Starting with ACE v2.0 there are two ways to use the compiler:

  - From the shell/CLI.
  - Via an Integrated Development Environment: AIDE.

Whichever environment you choose to work with ACE in, read on.

ACE expects all BASIC source files to have a ".b" or ".bas" extension.

If you have a program called foo.b[as], you would invoke the compiler thus:
```

```
  ace foo    (or ace foo.b[as])
```

This would produce foo.s, an A68K-compatible assembly source (text) file.

If you wanted to preprocess, compile, assemble and link foo.b[as], you'd type:

```
  bas foo
```

which would yield foo (the executable).

The bas script sets the stack to 40000 bytes. Before running ACE by itself, you will need to set this. A minimum stack size seems to be around 5000 for many ACE compilations, but I recommend 40000 to be safe.

If your Amiga GURUs or hangs during a compile or produces garbage in the shell, you can be confident that the stack is too small.

You can either create a BASIC source file using an editor or in the AmigaBASIC environment. If you want to compile a program developed with the interpreter, just save the program in ASCII format thus:

```
  save "foo.b[as]",a
```

ACE will only compile ASCII source files, not AmigaBASIC's compressed format.

For those who don't have access to the AmigaBASIC interpreter but who wish to convert old AmigaBASIC programs not saved in ASCII format, see the ACE:utils directory for a utility called ab2ascii written by Stefan Reisner.

1.10 Compiler options

```
Compiler options
----------------
```

The full command line syntax for ACE is:

```
  ace [words | -bcEilmOw] <sourcefile>[.b[as]]
```

which indicates that there are currently eight optional compiler switches.

Before giving a description of these switches however, let me say something about the optional "words" parameter. If the command:

```
  ace words
```

is given, all the reserved words which are known to the compiler will be dumped to standard output (ie. to the shell/CLI). Using redirection thus:

```
  ace words > rwords
```

will result in a file containing this information. This feature will be useful for anyone wanting to know which words are reserved by the compiler. AmigaBASIC and ACE keywords are differentiated.

The switches can appear in any combination (eg: -bO, -clb, -O, -ObE) but *are* case sensitive (so -b does not equal -B).

The "b" switch tells the compiler to include code to check for ctrl-c breaks by the user. The inclusion of this code can result in noticeably larger assembly source files, but execution speed doesn't seem to suffer appreciably. When a ctrl-c is detected, the program will clean up and exit but user-defined windows and screens will remain open. The use of ON BREAK can get around this by allowing for user-controlled clean up (see the "Event Trapping" section).

The "c" switch includes each line of ACE source code as a comment in the final assembly source file. This was requested as a debugging aid. Warning: the presence of such comments interferes with peephole optimisation. Also be aware that ACE sometimes includes extra code apart from that which you would expect purely on the basis of the source code.

The "E" switch creates a file in the current directory called ace.err which contains all error messages generated during a compliation. Error messages are still displayed to the screen during compilation however.

The "i" switch tells ACE to make an icon for the executable resulting from the compilation. The file "ACE:icons/exe.info" must exist as it is used as the source file for the icon. This allows you to use an icon of your own if you so wish.

The "l" switch causes the compiler to display each line of ACE source code as it is being compiled.

The "m" switch creates a linkable module containing no startup code (See Creating & using ACE subprogram modules section).

The "O" switch causes the assembly source code produced by ACE to be optimised. A range of peephole optimisations is currently carried out. Assembly code size reductions of around 5% to 10% are usual. Speed improvements vary, depending upon the program, however I recommend the use of the -O switch for all programs where speed is the least bit important. ACE's optimiser has been improved since v2.0 and it will continue to be improved.

The "w" switch tells ACE to include checks for window close-gadget clicks. ACE checks all open windows and upon detecting a close-gadget click, the clicked window is closed and the program exits. However, any other open windows or screens will not be closed. The use of ON WINDOW can get around this by allowing for a user-defined clean-up subroutine (see the "Event Trapping" section below).

See also the OPTION command in ref.guide.

The syntax for the bas script is:

```
  bas [-bcEilmOw] <sourcefile> [<objectfile>]
```

where <sourcefile> is the program to be compiled (without the .b[as] extension) and <objectfile> is a C or assembly module which has been (compiled and) assembled to produce an object file.

The <objectfile> is linked with the output of ACE+A68K along with db.lib, startup.lib and ami.lib. This is necessary when an external function or variable in <objectfile> is referenced by an ACE program. For more about external functions, see the section "External references".

1.11 Running ACE programs

Running ACE programs

ACE programs can be run from either a shell/CLI or Workbench. In the latter case a tool icon must be created for the executable. One has been provided with the archive in the icons directory (exe.info). Refer to the "i" switch in "Compiler options" re: automatic icon creation by the compiler.

1.12 About the example programs

About the example programs

I have written a number of programs which illustrate most of the features of ACE up to this point and you should find these with the distribution.

Several programs are related to chaos theory and fractals. The remainder are an assorted bunch which demonstrate ACE's capabilities.

The information for the chaos/fractal programs came from a wide variety of sources. The algorithms for henon.b and lorenz.b came from "Dynamical systems and fractals: computer graphics experiments in Pascal" by Becker & Dorfler, 1990.

Some programs are optimised at the source level and some are not. You'll find that using integer variables can often result in quite dramatic improvements in program execution speed (eg: try replacing op% with op! and k% with k! in ifs.b and you'll see what I mean).

There are several examples which demonstrate the use of recursive subprograms in ACE (eg: misc/fact.b).

Another area of interest for me is neural networks and you'll find a program called gfx/hopnet.b, which shows graphically how a simple Hopfield network changes under various conditions.

Other programs include a talking clock (tclock.b), a sound sample player (sound/play.b) and a command-line calculator (ShellUtils/bc.b).

1.13 General comments

General comments

I made a decision very early on in the project to allow standard I/O (for

```
shell/CLI). All other windows are (as of ACE v2.0) Intuition windows.

The execution speed of most programs (especially with graphics, eg: ifs.b)
is, as you might expect, *fast* compared to interpreted AmigaBASIC.

No error messages are given at run-time (but some errors are reported via ERR
and ON ERROR), nor is there any stack overflow or array bounds checking.

Labels are supported and can be used with GOSUB and GOTO. Line numbers are
supported, but are only necessary for old BASIC programs. Also, ACE's rich
assortment of control constructs makes the use of GOTO largely redundant.

Available control constructs are: WHILE..WEND, REPEAT..UNTIL, IF..THEN..ELSE,
IF..THEN..ELSE..END IF, CASE..END CASE, ON..GOTO, ON..GOSUB, SUB..END SUB and
GOSUB..RETURN.

Apart from single line comments with REM and ' ACE allows block comments
with { and }. For example:

 { comments can span more than
   one line like this }

Multi-statements are also supported by ACE, eg:

 x$="hello":y$="there":say translate$(x$+" "+y$)

In ACE (as in AmigaBASIC) end-of-line characters (ASCII 10) are significant,
so a line continuation character (~) must be used to extend expressions and
parameter lists (etc) over more than one line, eg.

 DECLARE FUNCTION MyAmazingExternalFunction(theFirst, theSecond, theThird, ~
              theFourth) EXTERNAL

Note that this feature is ACE-specific.
```

1.14 The preprocessor and #include files

```
The preprocessor and #include files
-----------------------------------
The ACE preprocessor: APP, is modest when compared to the C preprocessor.

Its main function thus far is for the inclusion of files with the #include
directive. As in C, #include "filename" looks for the file as specified,
while #include <filename> looks for the file in a local include directory
(see "Installation"). A file will only ever be included once.

The #include directive can also be used in included files, but since file
inclusion is recursive, watch your stack size (40000 bytes is plenty).

APP also handles single-line comments (text following a "'") and block
comments (starting with "{" and ending with "}"). This is partly to allow
#include commands to be commented out and also to make less work for the
compiler. However, the compiler does still handle comments in case the pre-
processor isn't invoked. APP does not handle REM since this is a BASIC
statement.
```

```
The syntax for APP is:

  app <source> <dest>

The bas script uses APP by first preprocessing an ACE source file to the
RAM:T directory.

Use of #include has the effect of adding lines to the preprocessed ACE
source, which has an impact upon the physical location of lines from the
main ACE file when transfered to the destination file.

ACE include files have two purposes. As in C they can be used to include
constants and structure definitions. Second, as with files like WBarg.h,
ACE include files may contain subprogram definitions and although this is
quite possible in C, it seems to be less often done under the guise of .h
files. I have arbitrarily chosen to append all include files with .h but
there is no reason why this need be so. Purists will probably be aghast.

APP will be improved as time goes by.
```

1.15 Data types, expressions and constants

```
Data types, expressions and constants
-------------------------------------
The following fundamental data types are currently supported:

       - signed short integers (2 bytes = 16 bits)
       - signed long integers  (4 bytes = 32 bits)
       - single-precision: Motorola fast floating point (4 bytes = 32 bits)
       - strings (default to 1024 bytes including ASCII 0 at end of string)

Exponential and fixed-point formats are recognised by all ACE functions
and in program text for single-precision numbers.

Expression parsing is the same as for AmigaBASIC, as is the precedence of
operators. Evaluation of _all_ expressions proceeds from left to right.
This includes exponentiation, so 2^3^2 will be evaluated as (2^3)^2.

In addition, due to the higher precedence of exponentiation over unary
negation and the way ACE's recursive descent parser works, 4^(-2) is okay,
but 4^-2 isn't.

ACE supports full 32-bit and single-precision floating point math:

  addition, subtraction, multiplication:

    - 16-bit integer
    - 32-bit integer
    - single-precision

  division & modulo arithmetic:

    - 32-bit integer
    - single-precision
```

Increment and decrement operators are provided in ACE in the following form:

```
  ++<variable> OR --<variable>
```

The value of the simple or external variable is incremented or decremented by 1.

Notice that ++ and -- are pre-increment & pre-decrement operators ONLY. Those familiar with C will recognise these operators and their utility. In terms of efficiency: ++x is better than x=x+1.

Unlike interpreted AmigaBASIC, hexadecimal and octal constants can be either short or long values. This makes for nicer addressing with PEEK & POKE. Also, the prefixes &H and &O may precede an integer anywhere that it makes sense, including in strings submitted to the VAL function. The effect of &H and &O is to indicate that a hexadecimal or octal value follows. The use of these prefixes is consistent with AmigaBASIC.

Trailing characters (%&!#) after constants cause coercion from one numeric data type to another, as in AmigaBASIC, eg:

```
       Delay(50&)   '..50 is coerced from short to long integer
       x=12.5*65!   '..65 is coerced from short integer to single-precision
```

Using these type-qualifier characters means that ACE doesn't have to generate numeric conversion code. This leads to smaller assembly code source files.

As in AmigaBASIC expression evaluation, all operands in an expression are converted to the data type of the most precise operand. Logical operators (AND,EQV,IMP,NOT,OR,XOR) convert their operands to integer values as does the integer division operator "\". Relational operators (= <> > < >= <=) yield long integer results.

ACE's boolean values are as follows: 0=false, N=true where N is any non-zero long integer. Note that relational operations give -1 for true (since: NOT -1 = 0).

Please note that in addition to their use in relational expressions, the logical operators (AND, NOT...) actually work in bitwise fashion. This means that you can create bitwise AND masks and perform other operations on groups of bits, as desired. The SHL and SHR functions may also prove useful in this area (see ref.guide).

ACE allows you to define named global signed numeric constants with the CONST directive.

Strings have a default length of 1K instead of the usual 32K, since ACE programs reserve memory for each string immediately at run-time which could result in quite memory hungry executables if strings were too large. It is possible however, to define strings which are longer or shorter than 1K (see STRING command).

ACE strings are NULL terminated, ie: the last character is an ASCII 0, as in strings manipulated by C's standard library functions.

```
A string literal without a final '"' will be truncated at the end of the
line.
```

1.16 Precedence of operators

```
Precedence of operators
-----------------------
ACE follows AmigaBASIC in operator precedence, with the addition of
structure dereferencing and indirection operators.

              operators
              ---------

  1.  Structure Member Dereferencing,   ->
      Parentheses and Address Operator    () @
  2.  Indirection Operators     *% *& *!
  3.  Exponentiation        ^
  4.  Unary Negation        -
  5.  Multiplication and Floating-Point Division  * /
  6.  Integer Division        \
  7.  Modulo Arithmetic       MOD
  8.  Addition and Subtraction      + -
  9.  Relational Operators      = < > <= >= <>
  10. NOT
  11. AND
  12. OR and XOR
  13. EQV
  14. IMP

The use of parentheses in an expression forces the enclosed term to be
evaluated before adjacent terms. Expression evaluation always proceeds
from left to right in ACE and AmigaBASIC.
```

1.17 Indirection operators

```
Indirection operators
---------------------
ACE has four indirection operators: @,*%,*&, and *!. These are _similar_
to pointers in C.

      @<object>   - returns the absolute address of a data object.
                  - note that this is identical to VARPTR(<object>).
      *%<address> - peeks or pokes a short value at the specified address.
      *&<address> - peeks or pokes a long value at the specified address.
      *!<address> - peeks or pokes a single value at the specified address.

The indirection operators can therefore be used in a statement (poke)
and/or as part of an expression (peek), for example:

  address x
  y=23.25
  x=@y
```

```
  *!x := *!x + 2
  print y
```

will print a value of 25.25.

There are two things to notice here. First, the pointers are to addresses, not necessarily connected to variables. It would be quite legal to allocate an area of memory and then dereference it with these operators.

Second, when assigning a value to a dereferenced memory location as in the above example, the ":=" symbol must be used, simply because of the way the parser processes statements. Pascal programmers will recognise this as the assignment operator.

See also the section below for information about how to use these operators to implement variable parameters (call-by-reference) for simple variables in ACE subprograms.

1.18 Identifiers

```
Identifiers
-----------
```
As in AmigaBASIC an identifier can consist of a combination of letters, numbers and periods (".") up to a maximum length of 40 characters.

In ACE the underscore ("_") character is also legal. An ACE identifier must start with either a letter or an underscore character.

An identifier can be used to represent the following:

- labels
- arrays
- variables
- parameters
- structures
- subprograms
- defined functions
- named constants
- shared library functions
- external functions and variables

Note that in ACE, a simple variable and an array with the same name cannot coexist. For example, a single-precision variable V cannot coexist with a single-precision array, eg: DIM V(100). However 'V!' *can* live with V(100) since V and V! are different variables in ACE. See the discussion of qualifier characters below.

Labels are global in ACE, so a main program label and a SUB label cannot have the same name. The format for a label definition is:

```
  <name>:
```

Labels are used by GOTO and GOSUB. For example:

```
  GOSUB HaveSomeFun
```

```
  STOP

  HaveSomeFun:
    PRINT "Are we having fun yet?"
    RETURN
```

Identifiers can have a qualifier character (%&$!#) appended in order to indicate data type, where:

```
    % = short integer
    & = long integer
    ! = single-precision
    # = double-precision -> not supported yet
    $ = string
```

Examples of valid identifiers are:

```
    x3
    num&
    _putchar
    play.sound
```

An identifer with no qualifier has a default type of single-precision. The DEFxxx compiler directives (see command and function reference) have the same effect as the qualifier characters except they affect all identifiers starting with a certain letter. Qualifier characters have higher priority than DEFxxx directives.

For shared library functions and external references, a qualifier is used merely to declare data type. So for example, an external function might be declared thus:

```
    external function RangeRand%
```

but can later be referred to as RangeRand.

The declaration of external functions/variables and shared library functions is global no matter where the declaration occurs.

Defined constants are unaffected by qualifier characters. The _value_ of a defined constant determines its type. Thus CONST x&=1.2 is a single-precision - NOT a long integer - constant. Needless to say therefore, it is unwise to use qualifier characters for named constants.

The declaration of constants (with CONST) is always global whether the declaration takes place in the main program or a subprogram.

Structure variables hold a long integer value (address), so trailing characters have no effect.

Structure type definitions are global, but structure variable declarations are local.

ACE allows for *optional* variable declarations with the SHORTINT, LONGINT, ADDRESS, SINGLE and STRING directives. Such declarations are useful in that:

```
    (i) They ensure that a variable has a NULL or zero
      value.

    (ii)  They prevent dangerous errors which result from
      the misspelling of variable names.

    (iii) Most languages have them and they serve to
      document variable usage explicitly.

    (iv)  They provide a "cleaner" way of establishing a
      variable which is to be shared by a subprogram
      in ACE.
```

My feeling on the matter of variable declarations is that in a small program they probably aren't necessary so long as you are careful, but in a large program all major variables should be declared for safety. I plan to add a compiler switch which could be used to enforce mandatory variable declarations.

Variable declarations override the DEFxxx compiler directives and qualifier characters and are local to the current level (main program or subprogram).

Summary of identifier properties:

Identifier	Local/Global	Affected by %&!$
ARRAY	LOCAL	YES
SIMPLE VAR	LOCAL	YES
STRUCTURE VAR	LOCAL	NO
PARAMETER	LOCAL	YES
STRUCTURE DEF	GLOBAL	NO
LABEL	GLOBAL	NO
NAMED CONST	GLOBAL	NO
SUBPROGRAM	GLOBAL	YES
DEF FN	GLOBAL	YES
LIBRARY FUNC	GLOBAL	YES (declaration)
EXT VAR/FUNC	GLOBAL	YES (declaration)

1.19 Files

Files

AmigaBASIC sequential files are supported and random files are on the list of things to do.

The commands and functions for manipulating sequential files in ACE are:

commands:
- OPEN
- CLOSE
- PRINT#
- WRITE#
- INPUT#

```
  - LINE INPUT#

functions:
  - INPUT$
  - EOF
  - LOF
  - HANDLE *

  * not found in AmigaBASIC
```

Note that for any command which is immediately followed by a # there should be at least one space between the keyword and the #, even though I may refer to such commands as <name># in the text of this document and in ref.guide.

See the command and function reference for details of each of these.

When WRITE# is used, the result is identical to AmigaBASIC. For example:

```
  X=12 : Y=-3.2 : Z$="fun eh?"
  OPEN "O",#1,"stuff"
    WRITE #1,X,Y,Z$
  CLOSE #1
```

results in a one-line file of the following format:

```
  12,-3.2,"fun eh?"
```

On the other hand, if the following is used instead:

```
  PRINT #1,X,Y,Z$
```

the file format will be:

```
   12     -3.2    fun eh?
```

while if semicolons are used:

```
  PRINT #1,X;Y;Z$
```

the file format becomes:

```
   12 -3.2 fun eh?
```

INPUT# (eg: INPUT #1,X,Y,Z$) can be used to read values from a file in any of the above formats, but bear in mind that strings that are not delimited by quotes, but contain spaces or tabs will be seen as more than one string by INPUT#. So, in the example formats above, while

```
  "fun eh?"
```

is one string,

```
  fun eh?
```

is two strings as far as INPUT# is concerned.

The formats of sequential files in ACE and AmigaBASIC are now very nearly identical, the only differences being in ACE tabs (produced by comma delimiters in PRINT -- see PRINT in ref.guide) and the number of decimal places written for single-precision values (usually more in ACE).

If you find ACE file I/O too slow, you may want to use the dos.library functions (eg: xRead, xWrite). For this reason, I have included the HANDLE function which returns the AmigaDOS handle of a file opened with ACE's OPEN command. You may also wish to use the ami.lib buffered file I/O functions which also require this handle. If HANDLE returns 0, the file can't be opened.

See prgs/IO/print.b for an example of opening a sequential file to a printer.

Although SER: may be opened as a sequential file, it is not possible to specify parameters for the serial port (baud rate etc) by this method as is possible in AmigaBASIC.

Instead, ACE provides a set of special serial I/O commands. See ref.guide for details of SERIAL OPEN/CLOSE etc, and prgs/IO/aterm.b for a simple terminal program.

1.20 Command line and Workbench arguments

```
Command line and Workbench arguments
------------------------------------
```
When called from a Shell or CLI, an ACE program may have arguments, eg:

```
  tree 30
```

Arguments can be accessed by two ACE functions:

```
  ARGCOUNT and ARG$(n)
```

The former returns a short integer value indicating the number of arguments for the current program, while the latter returns the nth argument as a string where n ranges from 0 to argcount. The zeroth argument (ie: ARG$(0)) is the name of the program.

Workbench arguments are currently supported by ACE in the form of four functions in the include file WBarg.h: WBargcount, WBarg$(n), WBargPath$(n) and WBargLock&(n).

The first three are the most useful. The fourth is mainly for use by WBargPath$.

WBargcount returns the number of arguments passed to a program. This includes the program name.

As with ARG$(0), the zeroth Workbench argument is the name of the program.

To pass arguments to a program via Workbench one of the shift keys is held down while the icons which represent the arguments to be passed are activated. While still depressing the shift key, the application icon is

double clicked.

An alternative method of passing arguments is to change the default tool of a project icon (eg: document) with the Info option from Workbench.

When this project icon is double clicked, the default tool will be loaded. In this case, if the source code of the default tool (the program) had a line such as:

```
x$ = WBarg$(1)
```

x$ would contain the name of the project file.

WBargPath$(n) is used to find the full path of the file name and includes trailing ":" and "/" characters.

See the include file WBarg.h for further descriptions of each function.

1.21 Subprograms

Subprograms

Subprograms are supported by ACE, but differ from AmigaBASIC subprograms in a number of ways. Namely, ACE subprograms:

- Don't use the STATIC keyword,
- Allow recursion,
  - Can be assigned return values.

ACE subprograms don't make use of the STATIC keyword because they are non static. This means that between calls to a specific subprogram, variables local to the subprogram cease to retain any meaningful value since the memory used to store them may be reused for other purposes.

Recursive subprograms are an important feature of modern general programming languages. For several examples of the use of recursion, see the included programs (eg: fact.b, hanoi.b, tree.b). See also ACEinclude:WBarg.h.

A word of warning about recursion: it can be stack hungry, so it's a good idea to set your stack to 20000 or so, just to be safe, although in most cases, this will be a lot more than you need. From Workbench, simply change the tool's stack size with Info, or with the STACK command in a shell.

As with AmigaBASIC, ACE subprogram declarations cannot be nested.

The syntax of a subprogram call is the same as in AmigaBASIC:

```
[CALL] sub-name[(parameter-list)]
```

The only difference is that the parentheses around the parameter list are not optional when CALL is omitted -- unless there are NO parameters.

CALL *must* be used after THEN in a single-line IF..THEN statement.

By default, every subprogram has a return type of single-precision (just like variables). The DEFxxx directives can be used to change the default data type of subprograms, as can the trailing characters !#$&%. A subprogram name can also be preceded by SHORTINT,LONGINT,ADDRESS,SINGLE or STRING as yet another alternative to setting the subprogram's return type.

The fact that ACE subprograms can be easily used as functions pretty much obviates the need for DEF FN. However for reasons of compatibility with AmigaBASIC and other BASICs, as well as its utility for simple functions, ACE supports DEF FN (as of ACE v2.0).

A subprogram is given a value either inside the body of the relevant subprogram or in the main program (eg: to zero it) - ala Pascal - thus:

```
  sub-name = <expression>
```

However, subprograms cannot be assigned a value in any other way (eg: with INPUT or READ).

A subprogram can be used in an expression, whereupon the subprogram is called and its value pushed onto the stack for inclusion in the final result of the expression, eg:

```
  x = n*pow(n)
```

where "pow" is a subprogram with one parameter.

While subprogram declarations can appear anywhere within the program text, ACE requires that declarations precede calls. So:

```
  sub test
    print "hello"
  end sub

  test
```

is legal, but:

```
  test

  sub test
    print "hello"
  end sub
```

is not, and will yield an "undeclared subprogram" error. To get around this, a forward declaration can be used:

```
  declare sub test

  test

  sub test
    print "hello"
  end sub
```

Forward declarations can include a parameter list. If you later declare

the actual SUB with a different parameter list and you've already called the subprogram after a forward declaration, the results will be unpredictable. I may place tighter controls on this at some stage.

Actual parameters are checked for number and type against formals, and parameter count mismatches result in a compilation error. Actual parameters are coerced to the corresponding formal parameter's type.

ACE's parameter passing mechanism for subprograms is NOT the same as that used for assembly code routines or C functions. In other words, the standard C parameter passing mechanism is not used for SUBs. This may be changed in the future as it makes object modules written in ACE incompatible with C or assembler object modules in this respect.

Changes made to a formal parameter have no effect upon the actual parameter in a simple call to an ACE subprogram, but see "Limitations" re: overwriting of strings/arrays during recursive calls; see also "Structures".

The formal parameter list consists of identifiers separated by commas. Each identifier may also be preceded by: SHORTINT,LONGINT,ADDRESS,SINGLE or STRING to avoid the use of a qualifier (%&!$).

Actual parameters can basically be any type of expression. A whole array cannot be passed as a value parameter in ACE however.

There is an arbitrary upper limit of 40 parameters per subprogram at the moment, which may be removed at some stage.

Main program variables and arrays can be accessed and modified within subprograms via the SHARED directive. All shared variables are passed by reference to a subprogram.

Multiple SHARED statements are allowed within a single subprogram.

DIM SHARED is not allowed. An array is declared to be shared in exactly the same way as simple variables, for example:

```
DIM x(10)

sub thing
      shared x
      .
      .
      end sub
```

Note that parentheses are not required after an array in the shared statement, nor are they legal in ACE.

Keep the following in mind with regard to shared variables in ACE:

- Shared variables only allow access to main program variables from a subprogram, and do not provide a mechanism for changing the value of variables in one subprogram from another.

- The name of a variable to be shared must correspond

```
    to the name of an existing (ie: already referenced/declared)
          main program variable.
```

Although variable parameters are not explictly provided by ACE there are two ways to implement them: using indirection operators for simple variables and the ADDRESS option of DIM and STRING (see also "Structures" section).

Here's an example of call-by-reference parameters for simple variables:

```
  sub doub(address x)
    *!x := *!x * 2    '...n! = n!*2 [note the ":=" symbol!]
  end sub

  n!=22.5
  print n!
  doub(@n!)  '..pass the address of n!
  print n!
```

which passes the single-precision variable n! by reference to the subprogram doub, where n! is doubled. This will first print 22.5 and then 45.

For an array, the following can be done:

```
  sub test(address x)
  dim a(10) address x
    a(3)=a(3)+12
  end sub

  dim n(10)
  n(3)=2
  print n(3)
  test(@n)  '..pass address of array n.
  print n(3)
```

which would print first 2 and then 14.

The same mechanism can be used to pass a string variable by reference (see the STRING command's ADDRESS option).

These variable parameter mechanisms are most useful when used to pass data *between* subprograms, otherwise it is simpler to use SHARED variables.

The following table shows the possibilities regarding parameters and shared variables in ACE:

Data Type / Object	Shared	Value param	Call by Reference parameter
SHORTINT VARIABLE	YES	YES	YES - *%addr
LONGINT/ADDRESS VAR	YES	YES	YES - *&addr
SINGLE VARIABLE	YES	YES	YES - *!addr
STRING VARIABLE	YES	YES	YES - STRING x ADDRESS addr

```
| EXTERNAL VARIABLE   | NO     | YES       | YES - *%, *&, *!, STRING .. |
|         |        |       |           |
| ARRAY         | YES    | NO      | YES - DIM x ADDRESS addr    |
|         |        |       |                 |
| STRUCTURE       | YES    | NO      | YES - See "Structures"     |
+---------------------+--------+-------------+----------------------------+
```

Note: In the above table, "addr" is a long integer address. VARPTR or @ can be used to obtain this.

1.22 Structures

Structures

Structures have been included in ACE mainly because of their utility in gaining access to operating system functions.

Structure members may be of the following type: BYTE (in structures only), SHORTINT, LONGINT, ADDRESS, SINGLE, STRING. The latter can have an optional size specification. A structure may also have as a member another structure.

If you want to have a pointer to a structure (or a pointer to anything else) as a member, simply declare it to be of type ADDRESS.

Allowing structures to have other structures as members makes converting system structures from C to ACE much easier than it otherwise would be, eg:

```
  STRUCT DateStamp
    LONGINT days
    LONGINT secs
    LONGINT ticks
    END STRUCT

  STRUCT myFirstStruct
    DateStamp ds
    STRING name SIZE 30
  END STRUCT

  DECLARE STRUCT myFirstStruct mine
  DECLARE STRUCT DateStamp *myDate

  .
  .

  myDate = @mine->ds  '..assign address of mine->ds to myDate.
  myDate->days = 23

  .
  .
```

If an array is required as a structure member, it is currently necessary to use STRING <ident> SIZE <bytes>. Most system structures use character arrays (strings) anyway. As an example, if you wanted a SHORTINT array member with 50 elements (0..49) you could say:

```
  STRUCT mySecondStruct
    STRING myArray SIZE 100 '..reserve space for the array
    .
    .
  END STRUCT

  DECLARE STRUCT mySecondStruct aStruct

  DIM N%(100) ADDRESS @aStruct->myArray
```

You can of course use SIZEOF to determine the number of bytes ACE would set aside for a particular array. A two line program would accomplish this:

```
  DIM a_short_array%(49)
  PRINT SIZEOF(a_short_array%)
```

The value thus derived can then be used when declaring a structure such as the one shown above.

Note that in the case of the array and structure members above, it is necessary to take the address of the member and assign it to a normal array or structure (pointer to structure - see below) variable.

When declaring a structure, the only difference between the following two forms:

```
    DECLARE STRUCT mystructtype mystruct
  and
    DECLARE STRUCT mystructtype *mystruct
```

is that for the former, an appropriate data object is created (on a long word boundary), but not for the latter.

In both cases, mystruct contains the start address of a structure of type mystructtype. In the second case, the address is NULL until assigned a value (eg: with ALLOC). In both cases, the address can be reassigned at will, although this should only really be done for structure pointers (the second form).

Since both forms of structure declaration result in an address being stored (in mystruct in the example), the dereferencing operator is always "->".

```
  examples:
  --------

  PRINT mystruct       -  prints the start address of the structure.

  PRINT mystruct->mins -  prints the value of a member called mins.
```

The SIZEOF function can be used to determine the size of a structure type if allocating memory for a structure (see prgs/misc/linkedlist.b).

ACE structures are stand-alone data objects, and cannot be elements in an array, although structure *addresses* _can_ be. For an example of the latter, see prgs/misc/array_of_structs.b.

An ACE structure can be SHARED to allow its member's values to be
modified, or a structure's address can be passed to a subprogram, eg:

```
  struct my
    longint one
    longint two
  end struct

  sub test(addr&)
  declare struct my *second
    second=addr&
    second->one = second->one * 2
  end sub

  '..main
  declare struct my first
  first->one=12
  print first->one
  test(first)
  print first->one
```

which will print 12 followed by 24.

The following code allocates enough memory to hold a structure of type
"my", gives values to the structure's 2 members, and changes the address
held by the structure variable "third" to the start of the newly allocated
memory area:

```
  declare struct my *third

  sub create(ADDRESS a_struct)
  declare struct my *temp
    temp = ALLOC(sizeof(my))
    temp->one = 16
    temp->two = 10
    *&a_struct := temp  '..change structure variable's value
  end sub

  '..main
  create(@third)
  .
  .
```

Finally, it is not currently possible to use INPUT, (LINE) INPUT# or READ
in conjunction with structures.

1.23 Shared library function calls

```
Shared library function calls
-----------------------------
```
ACE provides access to shared libraries in the same way as AmigaBASIC does
with the exception that you MUST declare a function in order to use it.

Also, the library in question must either be in LIBS: or in ROM.

As of version 2.0, ACE and AmigaBASIC are otherwise pretty much the same. The ACE commands also retain their earlier syntax for backward compatibility and convenience.

The commands are as follows:

LIBRARY <libname>

- Where <libname> is the name of a shared library with or without quotes (eg: "graphics", "graphics.library", graphics).

- A ".library" or ".bmap" suffix is allowed but optional.

    - The LIBRARY command opens the shared library and provides a copy of its base address for use internally by function calls.

- If a library can't be opened at run-time, the program will abort.

LIBRARY CLOSE [<libname>]

- Closes the specified shared library or all open libraries if no library name is given.

- Closing a library more than once will cause no harm.

Notes about standard libraries used by ACE:

    - There are currently six standard libraries which are often opened by ACE routines during a program run. These are: dos, intuition, graphics, mathffp, mathtrans and translator libraries.

- If one of these six is opened by the LIBRARY command it will be opened at the start of the program *and* closed at the end. Any other library will be opened and closed at the points in the program specified by you.

- You don't actually have to close any of the six libraries mentioned above, but it won't hurt.

- Moreover, you never have to open or close the dos.library since ACE opens it for EVERY program.

DECLARE FUNCTION <funcname>[%&!#$][(param-list)] LIBRARY [<libname>]

    - Where <funcname> is the case sensitive name of a function in a shared library.

    - <funcname> may have a trailing character (&%#!$) to indicate type, otherwise default data type rules apply for the function's return value. This character is optional when CALLing the function.

- The optional parameter-list is for documentation purposes only and is otherwise ignored.

- If <libname> (same as for LIBRARY and LIBRARY CLOSE) is specified, ACE only looks in the bmap file for that library, otherwise ACE looks for the function in the bmap files for all open libraries and all the standard libraries known to the compiler. Needless to say that specifying <libname> results in faster bmap file entry lookups. This option is not given by AmigaBASIC however.

- Example: DECLARE FUNCTION SetSoftStyle LIBRARY

[CALL] <funcname>[(parameter-list)]

- Transfers control to the function <funcname>, loading the appropriate registers before doing so, according to the information about that function in the library's bmap file.

- The return value of a function can be accessed by calling a function as part of an expression, eg: addr& = AllocMem(100,2).

Function declarations are GLOBAL. They are are NOT optional in ACE.

** PLEASE NOTE ***
No type checking of parameters is performed, so expect weirdness if you pass values of the wrong type to a shared library function. ACE _does_ however now automatically promote all short integers to long integers by sign-extension.

When passing strings as parameters it is not necessary to add a CHR$(0) to the end of a string since ACE strings are already NULL terminated.

Either VARPTR or SADD can safely be used to find the address of a string variable or constant. Actually, the use of SADD or VARPTR for strings passed to library functions is optional, but it's probably a good idea to use one or the other all the time, for consistency's sake. These comments also apply to calling machine code routines and external functions.

It is up to YOU to open and close libraries correctly. ACE doesn't keep track of this, and will try to jump to a library function so long as it finds a reference to it in a bmap file even if the library hasn't been opened! As mentioned above, it is not necessary to open and close dos.library because _every_ ACE program does this.

ACE expects the bmap file for a library to be in the directory ACEbmaps: (see "Installation").

Given Commodore's liquidation this year and the resultant confusion over who's now in charge of all this kind of stuff, I have no idea when or if the .bmaps will be provided with the ACE archive. However, this doesn't really matter so long as you have the .FD files. Read on...

As of version 2.0, I have provided a program (FD2BMAP) which is functionally equivalent to ConvertFD (since this may NOT be freely redistributed) so that bmap files for new libraries can be created. The program FD2BMAP can be found in the ACE:utils/fd2bmap directory and was written in ACE by Harald Schneider,

with some modifications from me.

The 1.3 FD files can be found in the BasicDemos drawer on the Extras disk.

The FD files for Release 2.x/3.0 are available from Commodore (who?) for about $30 (you get six disks of developer goodies: ask for the Native Developer Kit).

The FD files can also be found on Fred Fish's CD-ROMs and on the InterNet FTP site: ftp.rz.uni-wuerzburg.de in /pub/amiga/frozenfish/bbs/cbm. Note that FDs obtained from these sources are not freely redistributable and must be obtained on an individual basis, presumably to prevent modification when being passed through many hands.

AmigaBASIC cannot handle functions which use address register a5. This is not true for ACE. Neither ACE nor AmigaBASIC allow the use of functions which use register a6 however.

See the programs in prgs/Library for examples of how to use shared library functions in ACE.

1.24 Machine code calls

Machine code calls

ACE supports AmigaBASIC's mechanism for calling machine code routines and the passing of parameters to such routines. AmigaBASIC's stack conventions are also followed (ie: C style parameter passing).

The syntax for calling such a routine is:

```
  CALL long-integer-variable-name[(parameter-list)]
```

Note that CALL is NOT optional. Also, the variable containing the address of the routine *must* be a long integer (LONGINT OR ADDRESS) in ACE.

For example,

```
  CALL caps&(length&,addr&)
```

will set up the stack like this:

```
  8(sp) = addr&
  4(sp) = length&
  0(sp) = return address
```

on entry to the machine code subroutine caps&.

On exit from a routine, ACE cleans up the stack by POPping all parameters.

You can use a short integer array, a string or an allocated area of memory (eg. with ACE's ALLOC function) to poke the bytes of a machine code routine into. I prefer the latter method.

Note that because ACE treats ASCII 0 as the end-of-string character, don't

use the string-building method, eg:

```
  z$=""
  for i=1 to N
      read b
    z$=z$+chr$(b)
        next
```

since if b=0, chr$(b) will be the NULL string. If you want to use a string, do the following:

```
  z$="" '..or STRING z$ SIZE 100 (if there are 100 bytes of MC).
  addr&=sadd(z$)
  for i&=0 to N-1
    read b%
    poke addr&+i&,b%
  next
  call addr&
```

The latter is okay, so long as you don't allocate other strings with odd sizes. But if you want to be sure that you have an area of memory which is long-word aligned, use ALLOC (or AllocMem), eg:

```
  addr&=Alloc(100)  '..100 bytes of ANY memory
  IF addr& = 0& THEN STOP
  for i&=0 to N-1
    read b%
    poke addr&+i&,b%
  next
  CALL addr&
```

The above examples assume the presence of appropriate DATA statements. See the prgs/MC directory for examples.

ACE also supports primitive inline assembly code inclusion. See ref.guide under ASSEM..END ASSEM for details.

1.25 External references

```
External references
-------------------
```

Reference can be made to a variable or function in another file which is resolved at link time. You may for instance, have written a function in C or assembler. It is possible to pass parameters to, call and obtain return values (as with ACE SUBs) from such a function in ACE after declaring an external reference to the function with the EXTERNAL FUNCTION or DECLARE FUNCTION ... EXTERNAL directive (see command and function reference for syntax).

When passing parameters, standard C parameter passing conventions are used. Although some C compilers seem to pass all parameters as 4 bytes per parameter on the stack, ACE allows 2 (short words) or 4 byte parameters. Be aware of this! See prgs/ExternFunc for examples.

External variables can be assigned values like normal variables, eg:

```
    external RangeSeed&
    RangeSeed=5276&

Instead of the "&" qualifier, the following is also legal:

    external longint RangeSeed

Note however that externally referenced string variables are assumed to be
arrays of characters ala C, eg.

    char my_buffer[80];
or
    char my_string[] = "Hello World!";

but _not_ a character pointer, eg:

    char *a_char_pointer = "Hello World";

Use a C function to return a character pointer, and externally reference it
from your ACE program.

All external reference identifiers have an underscore prefixed by ACE but
this is optional when declaring or using an external reference. C compilers
always seem to prefix referenceable symbols with an underscore, so ACE does
too.

Note that the names of external references (all except external SUBs -
see below) *are* case sensitive.

Also, the bas script can take as a third argument the name of the object
file (ie. .o or .lib file) produced from the original source which contains
the external function or variable to be linked with your ACE program.

You can't easily call ACE SUBs from C or assembler because ACE SUBs don't
use C parameter passing conventions and ACE code relies heavily upon
linking code from run-time libraries (db.lib and startup.lib).
```

1.26 Creating & using ACE subprogram modules

```
Creating & using ACE subprogram modules
---------------------------------------
It is now possible to create libraries of ACE subprograms and so to have
multi-file ACE projects. For example in one file you can have:

  SUB lines(n) EXTERNAL
    FOR i=1 to n
      LINE (RND*640,RND*200)-(RND*640,RND*200)
    NEXT
  END SUB

and in another file:

  '..main
  DECLARE SUB lines(n) EXTERNAL
```

```
  LIBRARY "graphics.library"
  WINDOW 1,,(0,0)-(640,200)
  lines(500)
  WINDOW CLOSE 1
  END
```

If the latter file is called say, main.b and the former is called lines.b then the following sequence of shell commands will give you an executable called main:

```
  ace -Om lines   { The -m switch is the key here }
  a68k lines.s
  bas -O main lines.o
```

The first two commands can be replaced by:

```
  module lines
```

which is a shell script in the bin directory. The effect of the last of the 3 commands shown above can also be achieved via the Linker menu in AIDE, while the Compiler menu "Create Linkable Module" option is the equivalent of the first two.

There are a few things to be kept in mind when using ACE modules:

1. Only subprograms - not subroutines (ie. via GOTO/GOSUB) - in such modules can be called from other modules.

2. The main module *MUST* open all standard libraries required by the other ACE modules linked to it. If in doubt add the following lines of code at the top of your main program:

```
    LIBRARY mathffp
    LIBRARY mathtrans
    LIBRARY graphics
    LIBRARY intuition
    LIBRARY translator  '..only if TRANSLATE$ used
```

If your code works as a single-module program but you invoke the GURU or your program just doesn't work anymore when part of it is placed into a linkable module, the above should fix it.

3. The "m" switch creates an assembly source module containing bare code with no calls to the startup functions normally invoked by an ACE program. Because of this, you need to keep the following in mind:

- Command-line arguments must either be handled in the main program module or ARGCOUNT or ARG$ must be used one or more times in the main module if they are going to be used in a module produced with ACE's "m" switch. A line of code in the main module such as:

  n = ARGCOUNT

  will suffice.

- Likewise, if ALLOC is used in a module, it must also be used at least once in the main program (eg. x=ALLOC(0) - no bytes will be allocated).

In both cases, it is a matter of letting the compiler know that these features are required by the final executable program which the above actions will do.

- In order to use DATA/READ within a module, make sure you issue a RESTORE command (in a subprogram) before the _first_ READ is executed. This RESTORE command *must* be in a subprogram.

  For example, in the calling module:

```
library mathffp
declare sub data_test external
data_test
```

  and in the library module:

```
sub data_test external
    restore
    read x
    print x
    data 1.345  '..DATA lines _can_ be outside of SUBs
end sub
```

- ON TIMER normally has special startup code associated with it. Since there _is_ no startup code in modules, you will have to include the following lines of code such that they will be executed before the ON TIMER code:

```
external function ontimerstart
ontimerstart
```

  For example, in the calling module:

```
  library mathffp
  declare sub timer_test external
  timer_test
```

  and in the library module:

```
  sub timer_test external
SHORTINT count
  external function ontimerstart    '..can be outside SUB
  ontimerstart
    on timer(1) gosub do_beep
    timer on
    while -1
        if count=5 then exit sub
    wend
  do_beep:
      ++count
      beep
      return
end sub
```

In addition, while subprograms can be treated as functions when in modules, because there is no startup code, SUBs in a module are allocated no space for return value storage. This means that function return values must be passed by some method other than via the stack frame. In this case, ACE uses the 680x0 register d0. Since d0 can be overwritten at any time, the last thing a subprogram (in a module) should do is to assign the final return value to the subprogram. C forces you to do this anyway, it's just that ACE normally gives you more flexibility than C. For example, the following:

```
SUB even(n) EXTERNAL
  m = n MOD 2
  IF m=0 THEN
    even = -1
  ELSE
    even = 0
  END IF
END SUB
```

is acceptable since the final action of the subprogram is to assign itself a return value. On the other hand, the following will cause d0 to be overwritten before the subprogram can return its value to the caller:

```
SUB even(n) EXTERNAL
  m = n MOD 2
  IF m=0 THEN
    even = -1
  ELSE
    even = 0
  END IF
    PRINT n;"mod 2 is";m  ' trashes d0!!
END SUB
```

If need be of course, you can simply use a temporary local variable to hold the final return value to be assigned later.

Note that the d0 convention ONLY applies when the "m" compiler switch is used and that it applies to all subprograms in a module. However, even if a SUB is declared to be external, the d0 parameter passing convention will not be used unless either: (i) the "m" switch is used for the module in which the SUB is defined, or (ii) a SUB is used after being externally referenced via DECLARE SUB ... EXTERNAL.

Instead of an external SUB it is legal to have an external DEF FN. The forward declaration is still the same as for a SUB. The definition for the "even" function (in a module) looks like this:

```
DEF even(n) EXTERNAL = -(n MOD 2)
```

Remember also that only variables which are local to SUBs and subprogram parameters may be used in a module. Global variables are not provided for correctly when declared in a module produced with the "m" switch, again because of the lack of startup code. This may be rectified in a future revision.

Finally, be careful to match up the return and parameter types for an

externally defined subprogram and its forward (external) declaration.

1.27 Windows

Windows

You can open up to nine user-defined windows per program. See the command and function reference for the syntax of the WINDOW statement.

All user-defined windows are now (as of ACE v2.0) Intuition windows with each characteristic being configurable via the "type" parameter as per AmigaBASIC (see ref.guide).

Windows can be opened on the Workbench screen or on a user-defined screen.

The zeroth window (the shell/CLI, if the program was CLI launched) is now the only instance of a DOS console window in ACE.

The WINDOW function takes a single parameter and returns information about the current output window. See ref.guide for details.

Note that for user-defined windows, close-gadget clicks must be handled by the use of ON WINDOW event trapping or via the "w" compiler switch (or the OPTION w+ command).

Please read the next section for more information about how windows relate to screens in ACE.

1.28 Screens

Screens

A single program can have open nine screens at once (memory permitting).

By default, when a screen is opened, a (BORDERLESS+BACKDROP) window the same size as the screen is also opened. Subsequent output is directed to and input received from this window until the screen is closed (unless other windows are defined for the screen). Note that this feature is not found in AmigaBASIC. Note also that this default window is *not* counted as having a window-id of 0. That privilege is reserved for the shell window if the program was shell-launched. Read on...

The main use for such a default window is to provide a simple graphics output "slate" for quick-and-dirty programs. The borderless+backdrop characteristic prevents user-defined windows which may later be opened onto the screen from accidentally being depth arranged behind an invisible window where they would stay for the remainder of that screen's life. With a backdrop window this cannot happen since it will _always_ be the rear-most window for a screen.

Text and graphics positions will be different on a screen's default window than for user-defined windows. For example, text in the first row will be

partially off the top of the screen, so LOCATE may need to be used to adjust this. In short, I recommend that default windows now ONLY be used for rough output. If you _still_ want a borderless window, use the WINDOW command and ensure that the window-type has 32 as a component.

Avoid mixing the use of default and user-defined windows. Except for the simple case in which you use nothing but screens and their default windows you should consider your own windows to be the primary output destination for graphics and text.

Windows with depth gadgets cannot be sent behind the default window, but one screen can be sent to the back of other screens. It can also be moved vertically (and possibly horizontally). SCREEN BACK|FORWARD can be used to shuffle screens under program control.

When a screen is closed, ACE makes the screen with the next highest id the current one, so it is advisable to open and close screens in ascending and descending order.

A special SCREEN function exists in ACE which returns pointers to various Intuition structures (window,screen,rastport,viewport,bitmap) and x,y font sizes. This is detailed in the command and function reference (ref.guide) as are the following commands: SCREEN, SCREEN CLOSE, PALETTE and PRINTS. The latter is now redundant since all commands and functions can - as of ACE v2.0 - be used transparently for screens, user-defined windows and the shell/CLI.

1.29 Gadgets

Gadgets

ACE supports the Amiga's three standard gadget types: boolean, proportional (vertical and horizontal), and string (including long integer).

Since one of my aims is to support all Amigas running everything from Wb 1.3 to Wb 3.0, I have chosen to stick with simple Intuition gadgets for now.

Even so, ACE now supports the 3D bevel-box look of GadTools gadgets found under Wb 2.x/3.0. Moreover, a BEVELBOX command allows the programmer to create such boxes at will.

A future revision may support other gadget types, such as radio buttons, check boxes, list boxes etc.

Memory permitting, up to 255 gadgets can be created during a single program run.

The GADGET command creates a gadget with specific features while GADGET CLOSE removes the gadget from the window. Once created, a gadget can be enabled or disabled, indeed it can be disabled upon creation if so desired.

The GADGET MOD command modifies the state of a slider (knob size and position).

Having created a gadget or gadgets, you must then decide how to receive and handle information from them. ACE provides four methods: standard event trapping (ON GADGET), polling (via the GADGET function), WAITing for a specific gadget or WAITing for any gadget.

Where it is possible to make your programs modal (ie: focussed upon a single event or event type) you can use the GADGET WAIT command.

The following commands set up a window with two boolean gadgets and a close gadget. The latter is set up by Intuition with the WINDOW command.

The program traps WINDOW and GADGET events. The comments should help you understand the code.

```
CONST having_fun = -1&

WINDOW 1,"Gadgets",(0,0)-(640,200),8

GADGET 1,ON,"Hit Me",(3,3)-(75,20),BUTTON,1
GADGET 2,OFF,"Quit",(100,150)-(200,175),BUTTON,2

ON GADGET GOSUB gadget_handler
GADGET ON

ON WINDOW GOTO quit
WINDOW ON

'..main loop (actually does nothing, but is necessary for event trapping)
WHILE having_fun
  '..have a nap while nothing's happening
  '..(don't hog the machine by busy waiting)
  SLEEP
WEND

gadget_handler:
  '..find out which gadget was selected
  gad = GADGET(1)
  LOCATE 12,40:PRINT "<<";gad;">>"
  if gad = 2 then quit
RETURN

quit:
  GADGET CLOSE 2
  GADGET CLOSE 1

  WINDOW CLOSE 1
END
```

Alternatively, you could poll for a gadget to the exclusion of other events:

```
.
.
'..await a gadget selection
REPEAT
  WHILE NOT GADGET(0)
    SLEEP  '..be a little nice to other tasks
```

```
  WEND

  '..which one?
  gad = GADGET(1)
  LOCATE 12,40
  PRINT "<<";gad;">>"
UNTIL gad=2
.
.
```

Finally, you can wait for a gadget:

```
.
.
GADGET WAIT 2   '.."GADGET WAIT 0" waits for ANY gadget! BEST method!
.
.
```

See the program prgs/GUI/ACEgadgets.b for an example of gadget programming in ACE.

For boolean gadgets you can - if you need to - get information about the width and height of the gadget's text font by calling SCREEN(5) and SCREEN(6). If you need more precise width information, use the graphics library TextLength or TextExtent (v36) function.

String and LongInt gadgets now have associated with them a 1K buffer.

For more details about the GADGET commands and function, see ref.guide.

1.30 Menus

```
Menus
-----
```

ACE supports menus ala AmigaBASIC, with two additions: menu item keyboard equivalents and a MENU WAIT command. The latter puts the program to sleep until a menu event occurs. The former is specified by an optional parameter to the MENU command, for example:

```
  MENU 1,5,1,"Quit","Q"
```

defines menu item number 5 in menu number 1 to be the 'Quit' option and sets up a command-key sequence (Amiga-Q) for that item. The third parameter enables the menu item as per AmigaBASIC.

Note that although ACE adjusts menu text for font size and type as set via preferences, some fonts may require you to pad your menu title/item names with blanks when using command keys to avoid overlaps, so it is a good idea to add a couple of spaces to the end of a menu item string.

For an example of menu programming with ACE see prgs/ifs.b. For a better example, see the source code for AIDE. Over time I will modify some of the other example programs in the archive so that they are menu-driven.

Note that in ACE, MENU CLEAR replaces MENU RESET.

See ref.guide for more details about the MENU commands and function.

1.31 Requesters

Requesters

As of version 2.0, ACE supports 3 standard requesters:

- System requester
- File requester
- Input requesters (STRING and LONGINT)

Visual Basic for Windows has had an influence upon me and led me to add these to ACE since I have now come to expect them. You can get the most commonly needed requesters with a single line of ACE code!

If you are running Wb 2.x and above, ACE generates an ASL file requester.

For Wb 1.3 an ARP file requester is invoked for two simple reasons:

- It is quite acceptable.
- The arp.library is common on Wb 1.3 systems.

See MSGBOX, FILEBOX, INPUTBOX and INPUTBOX$ in ref.guide for more.

1.32 Turtle Graphics

Turtle Graphics

You may be wondering one or more of the following:

- what the heck is Turtle Graphics?
- isn't that for kids?
- why did he include THAT?

To answer the first question: Turtle Graphics (TG) originated as a subset of the language LOGO invented by Seymour Papert et al at MIT. LOGO was originally intended as a language for learning. Children are able to write simple programs to draw shapes on the computer's screen or move a Turtle - a dome-shaped robot - on a sheet of paper on the floor, learning about geometry "by doing" and having fun to boot.

LOGO also has many Lisp-like qualities and so can be used as a language for AI work, although to my knowledge, it's not.

But I digress. Apart from the fun kids can have with TG, it's actually possible to construct quite complex shapes with it. Combined with recursion, TG is a powerful tool. It is particularly useful in plotting many fractal shapes (see snowflake.b, dragon.b).

Since the first LOGO, there have been many manifestations of TG. Turbo

Pascal for the PC and the Mac have both had TG.

A few years ago, I wrote a pure TG subset of LOGO which used the same syntax as the original language and allowed recursive procedures. I've also written TG functions in C. Both of these have been useful to me and I've often wished that BASIC came with TG built-in. Well, now one dialect does!

For some examples of the use of Turtle Graphics in ACE, see the following programs:

- tree.b
- flower.b
- boxit.b
- torus.b
- dragon.b
- snowflake.b
- bst.b

The above discussion should have answered the second question. As for the third, the answer is: because I wanted to!! :^)

Okay, enough philosophy. Here's the ACE stuff:

```
BACK n    - move turtle back by n.
FORWARD n - move turtle forward by n.
HEADING   - return turtle's current heading in degrees (0..359).
HOME    - move turtle back to its home position.
PENDOWN   - put turtle's pen down.
PENUP   - lift turtle's pen up.
SETHEADING degs - change turtle's heading to degs.
SETXY x,y - change turtle's current x,y location.
TURN degs - rotate turtle by degs.
TURNLEFT degs - turn turtle left by degs.
TURNRIGHT degs  - turn turtle right by degs.
XCOR    - return turtle's current x-coordinate.
YCOR    - return turtle's current y-coordinate.
```

where:
- n is pixels
- degs is a signed short integer representing degrees with the turtle starting at a 270 degree orientation -- pointing up.
- home is the turtle's x,y start location (0,0).

Note that the X:Y ratio can be modified thus:

```
EXTERNAL SINGLE tg_xy_ratio
tg_xy_ratio = 1.125
```

but this should not be necessary (from v2.19) since ACE determines the correct aspect ratio for each user-defined screen when it is opened. A hi-res, non-interlaced Workbench screen is assumed at startup however. The correct value for tg_xy_ratio are show below:

Screen mode	Aspect ratio
Lo-res, non-interlaced	0.9375

```
  Hi-res, non-interlaced    1.875
  Lo-res, interlaced    0.46875
  Hi-res, interlaced    0.9375
  Hold and Modify (HAM)   0.9375
  Extra-Halfbrite     0.9375
```

Most LOGO environments use a coordinate system where 0,0 is at the center of the screen and positions to the left and down of this origin are negative while those up and to the right are positive. ACE's TG system however, uses the Amiga's normal graphics coordinate system with 0,0 at the top left of the screen/window so as to maintain consistency with ACE's normal graphics commands and functions (eg: POINT, PSET, LINE, PAINT, CIRCLE, AREAFILL).

If a negative value is specified for the turnleft or turnright commands, the turtle will be rotated in the opposite direction to that indicated by the the command name. Note that there is also a TURN command.

When using ACE's TG system, it's best to think of an imaginary turtle (in LOGO it's usually a small triangle on the screen) which rotates and moves according to your whim. The turtle can either have its pen lowered or raised - and will therefore draw or not - which is useful when you need to move in a relative fashion from one location to another without drawing anything.

SetXY is like the graphics library Move() command and may need to be preceded by PENUP unless you want to draw a line as the turtle finds its new position.

To change the colour of the drawing pen, use the COLOR command.

That's probably enough about Turtle Graphics. Oh by the way, if you want to know more about the origins and uses of Logo, read Papert's "Mindstorms" and his recent book entitled "The Children's Machine: Rethinking School in the Age of the Computer". They make for interesting reading.

While I think of it, Sherry Turkle wrote a book called "The Second Self: Computers and the human spirit", which I recommend if you're at all interested in the psychological/social effects of computing.

1.33 Loading & displaying IFF pictures

```
Loading & displaying IFF pictures
---------------------------------
```

IFF graphics files can now be loaded and displayed with ACE by using a few simple commands and functions.

The short example program prgs/gfx/iff.b demonstrates typical usage.

IFF READ uses the freeware ILBM.library but you don't need to have this since it will be created by ACE automatically at run-time if need be. If LIBS:ilbm.library is not found at run-time ACE will use the library's binary image - which is stored in db.lib - to create ram:ILBMtmp/ilbm.library. This file and the directory ILBMtmp will be removed when the program exits. The idea of using a binary image

like this was Roland Acton's.

See ref.guide for details of the IFF commands and function.

1.34 Sound

Sound

ACE provides you with similar functionality as AmigaBASIC for sound generation. It also allows you to do some things that AmigaBASIC doesn't.

See prgs/sound/sound.b for an example of how to use ACE's sound facilities in general.

How many times have you wished that AmigaBASIC would let you produce white noise easily like the good ol' C64 and Vic-20 did?

Well, you'll be pleased to know that ACE allows you to do this. All you have to do is allocate about 4000 or more bytes of Chip RAM (upwards of 4000 bytes yields better quality white noise), poke it with random values (between -128 and 127), call WAVE and you're set (see sound.b)!

Moreover, you can actually play sound samples (IFF or otherwise) in the same way, using just the two commands WAVE and SOUND (see prgs/sound/play.b).

As with AmigaBASIC, a sine waveform is the default, but through the WAVE statement you can create any waveform you wish including sawtooth, triangle, square and random (white noise).

WAVE has the following syntaxes:

```
  WAVE voice,SIN
and
  WAVE voice,waveform-address,byte-count
```

where waveform-address is the start of a block of memory where the waveform resides (an area of ALLOC'd CHIP memory) and byte-count is the number of bytes in the waveform table.

The SOUND statement syntax is as follows:

```
  SOUND period,duration[,volume][,voice]
```

This is different to AmigaBASIC in a number of ways. First, in ACE you specify the sampling period NOT the frequency. This was easier to implement and still provides the same functionality, but if you want specific notes, you'll have to do the calculations yourself (see equations below).

Sampling period is inversely proportional to frequency, so a high sampling rate corresponds to a low frequency and vice-versa. ACE allows you to specify a sampling period in the range 124..32767.

The duration is a single-precision value as in AmigaBASIC but can range from 0..999 (instead of 0..77). This range is somewhat arbitrary, but gives plenty of scope for large sound samples. This specifies the length of time

```
that a tone should be played for. A duration of 18.2 corresponds to about 1
second.

Volume defaults to 64 if not specified and can range from 0..64.

The voice can be in the range 0..3 - since there are 4 audio channels -
with 0 & 3 corresponding to the left speaker and 1 & 2 to the right.
The default voice is 0.

At the moment, ACE's SOUND statement isn't very good when used to produce
a series of short pulses, although this is somewhat dependent upon the
waveform in use. In any case, more work needs to be done in this area to
prevent "popping" between SOUND statements when the audio hardware is
turned on and off in rapid sequence.

ACE sound is produced by programming the hardware directly. A future
version will utilise the audio device instead. Indeed, the SOUND statement
in ACE may change in the future to be more in line with AmigaBASIC (see also
"Future Versions").

Finally, here's some useful equations for use in conjunction with ACE's
SOUND statement:

  samples/second to period:
  -------------------------
  period = 3579546 / samples-per-second

  musical note to period:
  -----------------------
  period = 3579546 / (length * frequency)

  where length is the size of the waveform table in bytes
  (32 bytes for ACE's sine waveform) and frequency is the
  note itself (eg: middle C is 523.25 Hz).

  duration value for one waveform cycle:
  --------------------------------------
  duration = .279365 * period * length / 1E6 * 18.2
```

1.35 Event trapping

```
Event trapping
--------------
ACE provides for AmigaBASIC-style event trapping. The following event
types are supported:

  BREAK     -  user break: ctrl-c.
  MOUSE     -  left mouse button press.
  TIMER(n)  -  cause a trap every n seconds.
  ERROR   -  I/O and other errors.
  MENU      -  menu selection.
  WINDOW    -  window event: close-gadget click.
  GADGET    -  user-defined gadget selection.

Event trapping in ACE works by checking for a given event at strategic
```

points in a program (before NEXT, WEND, GOTO, CALL, PRINT etc) and if an event is detected, control is passed to a trap handling routine. Hence, trapping here does not refer to CPU traps (exceptions).

Even if your program expects to do nothing but trap events, you'll need a loop like this:

```
WHILE -1
  SLEEP   '..don't hog the CPU  [SLEEP is optional but nice]
WEND
```

if you wish to have any events handled by your program.

The specification for the trapping of an event is:

ON <event> GOSUB | GOTO <label>|<line-number> (eg: ON BREAK GOTO quit)

which indicates the routine to which control is to be passed when an event is trapped. This is followed at some stage by:

```
<event> ON  (eg: BREAK ON)
```

which causes the compiled program from that point UNTIL the trap handling routine, to contain event trapping code.

It is a good idea to put trap handlers at the end of a program, since once the handler for an event is found by the compiler, no more event trapping code is generated for that event even if there is code below the handing routine. In other words, the equivalent of an <event> OFF (see below) command is issued once the trap handling code is found by the compiler.

Other commands are:

```
<event> STOP
```

which disables trapping for the event until another <event> ON is issued, and:

```
<event> OFF
```

which disables trapping for the event permanently.

Just so there is no misunderstanding, the latter two commands prevent the inclusion of event trapping code for a specific event in your program at the assembly source level. They do this from the point in an ACE program at which they are issued.

Here's a typical example:

```
ON BREAK GOTO quit
BREAK ON

for i=1 to 1000000
 print i
next

quit:
```

```
    PRINT "**break!"
    STOP
```

Simultaneous trapping of several different events is possible and in general works very well. The use of INKEY$ and ON BREAK together when a user-defined window is the current output window leads to some competition between the two. You may simply need to hit ctrl-c a few times in this circumstance for a user break to be accepted.

If you wish to trap only one kind of event you should consider the use of WAITing (only for menus and gadgets currently - see MENU/GADGET WAIT).

1.36 Interprocess Communication

```
Interprocess Communication
--------------------------
```
ACE provides a simple IPC mechanism centered around a set of MESSAGE commands (see ref.guide for full details).

ACEports represent a half-duplex, blocking/non-blocking, named IPC mechanism, similar to Berkeley Unix domain datagram sockets.

The mechanism is based upon Exec message ports and provides a simple way for concurrently running ACE programs to communicate across unique, safe channels. Strings can be sent as messages to a particular message port, the name of which must be known in advance.

Probably the best way to find out how the MESSAGE commands are used in ACE is to compile, run and study the programs in prgs/ACEports.

I consider this feature to be experimental and am interested in your feedback regarding it. The ACEports mechanism will probably be improved as time goes by.

ARexx capabilities are also planned for ACE at some stage.

1.37 Error Handling

```
Error Handling
--------------
```
The compiler messages generated by ACE are often different to the ones in the AmigaBASIC interpreter (and ACE doesn't beep at you with each error) but they are usually fairly clear.

Syntactically incorrect programs can lead ACE to produce a bunch of spurious error messages. In such cases, it's best to ignore all but the first one or two, unless there are "clusters" of messages which are separated by periods of error-free compilation.

If you leave out END IF, WEND, UNTIL, NEXT, END SUB, END STRUCT or END CASE, there will be a corresponding number of error messages at the end of the compile. If you leave off two WENDs, you'll get 2 "WHILE without

```
WEND" error messages.

ACE generally reports the first error in a line of code and ignores the
rest of the "bad" line. A typical message consists of the line containing
the error, a carat ("^") marker, and the error message itself. More work
still needs to be done on ACE's compile-time error handling, but it's
bearable.

No error messages are issued by ACE programs at run-time. Generally, when
a program runs into something it can't do, or an erroneous request - like
trying to open two files to the same file number or trying to open a
library that doesn't exist - the program will either quit or just not have
the desired effect.

Note that while the ERR function and ON ERROR event trapping are supported,
only file I/O, serial I/O, IFF and IPC (MESSAGE) and window/screen open errors are ↩
    currently
reported via these mechanisms.
```

1.38 Notes for assembly programmers

```
Notes for assembly programmers
------------------------------
I've tried to make the assembly source files that ACE produces as
readable as possible by using meaningful data object names. See also
"Compiler options" re: the compiler's "-c" switch.

Linked library routines use data registers d0-d6 and address registers
a0-a3, while d7 is used for array index calculations. Also, a4 and a5 are
used as stack frame pointers for variables and parameters.

Most db.lib routines don't save and restore registers via the stack, but
the use of registers is internally consistent (ie: all registers are up
for grabs but interdependent routines are written in such a way so as not
to conflict). External function calls in ACE programs now _do_ save and
restore registers.

The use of linked libraries means that the size of all executables is
fairly large. But given that disk space and memory are cheap, I'd rather
this than the alternative of having every executable be dependent upon one
or more special shared libraries at run-time. However, I will try to reduce
the size of executables. Some improvement has been made with the current
revision. Kendall Sears has suggested the creation of smaller versions of
db.lib and startup.lib for use with shell-based programs. I like this idea
and will implement it when I have time.

Due to BASIC's tendency to coerce data types so much for the programmer,
the resulting code can look a little nasty, and big increases in
efficiency can be gained by careful combinations of data types in
expressions.

Writing ACE has so far been a learning experience for me and if when I
started I knew what I know now, I would have done many things differently.

My original rationale for passing parameters via registers to ACE (and
```

shared) library functions was to improve execution speed. However, since
I call lots of other functions (eg: in ami.lib) which require their
parameters to be on the stack, I would probably call ALL functions in this
way if I did it again.

Moreover, my desire for internal consistency led me to a rather odd method
of passing parameters to SUBs. This allowed me to treat parameters in the
same way as variables which is all very nice, but it led to other problems,
chief among them being the need to use a Forbid()/Permit() pair when
sending parameters to a SUB. This works fine however, so I'm taking the
view that if it 'aint broke, I probably shouldn't fix it.

1.39 Limitations

```
Limitations
-----------
```
Variables do NOT get a default zero or NULL value, so don't assume ANYTHING
about the contents of an uninitialised variable, eg.

```
  PRINT X
```

will yield garbage if X has not been given a value. The optional variable
declarations provided in ACE are therefore worth using since they DO give
variables an initial zero or NULL value.

The precision of exponentiation begins to falter with large numbers (where
the exponent is around 23 or higher) because all exponentiation is
currently done with the single-precision math library function SPPow(). Use
either long integer multiplication or ACE's SHL function for greater
accuracy, where integers are applicable. For example, compare SHL(2,22)
with INT(2^23). You may also find that integer exponentiation sometimes
yields a non-integer result, eg.

```
  PRINT 2^3
```
gives:
```
  8.0000009
```

This is an artifact of single-precision exponentiation. I intend to make ACE
take a more intelligent approach to integer exponentiation in a future
revision (probably the next one).

If you are only interested in an integer result, apply CINT or CLNG, thus:

```
  PRINT CLNG(2^3)
```
to get:
```
  8
```

See also the LONGINT(n) function for getting around the problem of
extracting a large integer value from a string.

While strings can be defined to be longer (or shorter) than 1K, there
are some ACE commands and functions which still assume a 1K limit, namely:
STRING$, SPACE$, LINE INPUT#, INPUT and SWAP.

Strings and arrays which are local to a subprogram will be overwritten if

the SUB has recursive calls to itself. The same applies to string parameters. In all these cases a single static data item is being referenced.

If you issue RETURN from within a FOR loop, the return address will *not* be the top item on the stack. Instead, FOR..NEXT loop data will be. A GURU will almost certainly result. It is probably better to use a while or repeat loop if you must RETURN from within a loop. See also EXIT FOR in ref.guide which allows for the safe, early termination of FOR loops.

A shared variable cannot be used as a FOR loop index in ACE. Any attempt to do so will result in a compile-time error.

IF..THEN NEXT will not have the desired effect (it's bad coding anyway). NEXT must always appear on a line by itself or as part of a multi-statement.

ACE does not consider "=>" to be the same symbol as ">=". In fact, ACE doesn't recognise the former at all. The same is true of "=<".

The compiler only responds to ctrl-c during the main compilation phase, and not during optimisation or when target code is being written.

Don't mix the compiler's "b" option with BREAK event trapping, as they will conflict.

1.40 Known Bugs

Known Bugs

The SAY(n) *function* works under release 2.x but not under 1.3. Since the function uses no 2.x-specific code, this is puzzling. The SAY command works under both 1.3 and 2.x however.

A68K sometimes complains about string literal definitions produced by ACE if they are much longer than a single line.

Some fonts cause the INPUTBOX[$] display (string gadget) to be corrupted. Stick to topaz for this where possible. A future revision may use a GadTools or BOOPSI requester (for 2.x/3.x machines).

1.41 Future versions

Future versions

Random files and double-precision floating-point math are high on the agenda.

More graphics (eg: GET,PUT) commands and functions are planned and I also intend to fix any remaining differences between ACE and AmigaBASIC in this area.

AGA screen modes are likely to be supported soon (especially since I recently purchased an A1200 :).

I may provide support for sprites at some stage.

Thus far, I've taken the approach of implementing what I most often use and what I have often wished for in BASIC.

In recent times I have been impressed by three things in the programming world: the rise of Object-Oriented Programming (OOP), Resources (as found on the Macintosh and MS-Windows) and Visual BASIC for Windows.

The idea of resources (pioneered by the Macintosh resource fork and ResEdit) is a powerful one and Microsoft and Borland have picked up on this in MS-Windows. I wish the Amiga had them as standard, but alas it doesn't.

I am trying to add more "visual" stuff to ACE, hence: gadgets, menus, requesters, AIDE. You can expect to see more "visualising" of ACE as time goes by, including a GUI designer for ACE programs. To make ACE like Visual BASIC would take a major rewrite, but I can at least try to take advantage of some of its features.

1.42 A note to PD libraries and reviewers

A note to PD libraries and reviewers

First, to those magazines who have reviewed ACE so far, let me say a big THANK YOU. Good publicity is always appreciated as is acknowledgement of the time I've spent on ACE.

I'd appreciate it if you would check with me (if possible) to ensure you have the latest version of ACE before including it in your library or reviewing it for a magazine. If you don't have e-mail access, I don't expect you to do this (snail-mail is a pain isn't it!?).

Lastly, the wider ACE travels the happier I am, so I'm pleased to see ACE turning up in the odd PD library. Please note however that I don't wish people to profit financially from the distribution of ACE. You may charge a fee which covers the cost of the disk and the copying thereof, but no more.

1.43 Disclaimer

Disclaimer

Although every care has been taken in the development and testing of the compiler and its libraries, the author will not be held liable for damages caused either directly or indirectly as a result of the use of ACE.

1.44 References

References

The following references have been used in developing ACE:

+---+
| "Amiga BASIC" (manual), 1985, Commodore-Amiga Inc. and Microsoft Inc. |
| |
| "Amiga ROM Kernel Reference Manual: Libraries", 1992, Commodore-Amiga |
| |
| "Amiga ROM Kernel Reference Manual: Devices", 1991, Commodore-Amiga |
| |
| Anderson & Thompson, 1990, "Mapping the Amiga", COMPUTE! Publications Inc.|
| |
| Bleek, Jenrich & Schulz, 1989, "Amiga C for Advanced Programmers", Abacus |
| |
| Choi, 1990, "Advanced Programming Techniques", University of Tasmania |
| |
| Dittrich, 1989, "Amiga Machine Language", Abacus |
+---+

Despite its not infrequent errors, "Mapping the Amiga" remains an excellent resource. I have also often referenced "Advanced Amiga C programming".

Naturally, the two RKM volumes listed above and the "Amiga BASIC" manual are used regularly as well.

Although not listed, Commodore's Autodocs for the Amiga (supplied with the Native Developer Kit) are also constantly used.

Young Choi's Advanced Programming notes in many ways provided the impetus for the development of ACE. They were used in a compiler construction course I took as an undergraduate. I thank Young for introducing me to the joys of compiler writing, although I know he didn't intend me to spend *all* my time writing programming language translators of one kind or another. :-)

Including the Pascal Minus compiler I wrote for that course, and the PC BASIC interpreter I wrote during the same period, I've since written a version of Logo (turtle graphics subset), a Forth interpreter and ACE.

1.45 Contacting the author

Contacting the author

I am contactable via e-mail on: the Internet, Compuserve and Discovery 40/80. Of course, there is also the telephone and snail-mail. Information about reaching me via these media is given below.

+---+
| Internet: D.Benn@appcomp.utas.edu.au |
| |
| Compuserve: 100033,605 |
| |
| Discovery 40: 032432850 |

| |
| Phone: (003) 317 680 [home] |
| (003) 243 529 [work] |
| |
| Address: 181 St John Street, Launceston, |
| Tasmania, Australia, 7250 |
+---+

1.46 ACE discussion list and FTP site

ACE discussion list and FTP site

In February 1994 I was allowed to establish a listserver-based discussion list for ACE on one of the Unix boxes at my place of work, the University of Tasmania at Launceston's Department of Applied Computing and Mathematics.

The purpose of the list is to allow for the dissemination of information about ACE, to discuss bugs, problems, solutions and ACE programming in general.

To subscribe to the list send (Internet) e-mail to:

  Listserver@appcomp.utas.edu.au

The subject-line of the message is unimportant, but the first line of the message must be of the following format:

  subscribe ace FirstName LastName

For example, in my case:

  subscribe ace David Benn

would do the trick. The listserver will send you a welcome message and information about how to participate in the list.

At about the same time as the ACE list was established, an anonymous FTP server was set up:

  ftp.appcomp.utas.edu.au

ACE-related files are stored in the directory:

  /pub/amiga/ACE

I'd like to thank Tony Gray (Technical Services Manager) and Christian McGee for maintaining the listserver and Young Choi (Head of Department) for allowing me to use the department's facilities for this purpose.

1.47 Final word

Final word

Let me offer my thanks to Charlie Gibbs for his reliable assembler and to the Software Distillery for Blink. Without these excellent programs, far fewer compilers would have seen the light of day, including ACE.

Sozobon C (ZC) has always been a reliable workhorse for me, so a vote of thanks goes to Sozobon as well. Isn't freeware great?

I'd like to thank those people who have tested ACE so far. They are too numerous to mention here, but special thanks goes to Addison Laurent, Alan-Peyton Smith, Peter Zielinski, John Stiwinter, and all the members of the ACE discussion list.

These guys have given ACE a good workout on a variety of platforms ranging from an A1000 running Wb 1.3 to 68030 machines running Wb 3.0.

I'd especially like to thank Michael Zielinski for discovering a particularly serious bug which prevented branches of greater than 32K in length prior to v1.02. Also, Enforcer hits reported by him started me on a trail which led to a nasty string-related bug (see entry for 13/4/93 in docs/history).

Jarto 'Robin' Tarp pointed out how inefficient my first implementation of INSTR was. It's considerably faster now (since v1.02).

Let me stress that any remaining bugs in ACE are entirely my fault.

Others have given me a great deal of encouragement and made useful comments throughout 1993 and 1994. I hope this continues. The sense of community on the virtual world of the Net is a refreshing change from the general weirdness of the "real" world.

I'd like to thank John Stiwinter for all his excellent work in putting together the AmigaGuide documents for ACE. Having ACE's reference material in this format has enhanced its utility enormously. He has also done other stuff in the background for which I am grateful.

Many of the subscribers to the ACE discussion list have given me plenty to think about, plenty of bug reports and plenty of support. In particular, thanks goes Kendall Sears for keeping me on my toes and for providing expert technical knowledge of the Amiga.

Jeff Harris, Chuck Kenney, Dan Oberlin, Sean Miller and Kenneth Brill have all contributed to ACE in their own ways. Thanks guys! Also, to all those who have e-mailed or written to me, thanks. You know who you are!

I'd also like to thank my wife Karen, for her encouragement and support, for putting up with the number of hours I spend at the computer, and just for tolerating me generally. :-) IFS is her favourite program so it's dedicated to her (try the "Green Fern").

Despite ACE's problems (see "Limitations" and "Known Bugs") it is already proving to be a useful tool for me, and if others can derive benefit from it, well, that's great. ACE is gradually coming to support the features I and others want it to, but there's still plenty of work to do.

The provision of support for shared library functions, external functions, machine code routines, inline assembly code, include files and linkable SUB modules should go a long way towards making up for ACE's intrinsic shortcomings.

I hope you enjoy ACE and find it useful. I'm learning a great deal by developing it and having a lot of fun in the process. If you feel the need to send me some money, please go ahead, but I certainly don't expect it.

What I DO want is feedback. If you have any problems, requests, queries or suggestions, I want to hear from you and I'd like to hear about interesting programs you've written with ACE (send me the source code if you like). I'm always on the lookout for good example programs to include in the ACE archive.

Remember that ACE is FreeWare, so redirect flames to NIL :^).

Happy programming!

Regards, David Benn
22nd October 1994