# Ctrans: portable ALGOL 68 compiler

Algol 68 to C translator and run-time system

$Id: ctrans.texi,v 1.2 2004/09/04 02:42:57 teshields Exp $

**Praxis Systems plc, Bath, UK**

# 1 Overview

Ctrans is a portable ALGOL 68 compilation system that allows Algol 68 programs to be compiled and run on most systems for which an ANSI C compiler is available.

The system consists of these components

- Algol 68 to C translator (`a68toc`)
- C header files providing type definitions and support macros required by the generated C code
- run-time support library including garbage-collected heap allocator
- an ALGOL 68 prelude library that provides interfaces to XPG3-compatible operating system facilities (note that these interfaces are not the same as the ALGOL 68 standard prelude defined in the Revised Report).
- a standard prelude library that provides facilities very similar, but not completely compatible, with the standard prelude described in the Revised Report.

Ctrans can be easily ported to most systems that possess an ANSI C compiler (e.g. `gcc`), and an XPG3 standard C library.

The heap management system works best on virtual memory systems, but can be made to work in other environments.

The reader needs to be familiar with ALGOL 68 RS (see Appendix B [Further reading], page 19) to fully understand this documentation.

# 2 ALGOL 68 language

The translator is based on the portable ALGOL 68 RS front-end implemented by RSRE (now DRA) and therefore accepts all standard RS extensions to the language (except where indicated in this section) such as `FORALL`. These are not discussed further here.

This chapter describes how the language accepted by Ctrans differs from standard ALGOL 68 RS in terms of restrictions, extensions and differences.

## 2.1 Language restrictions

### 2.1.1 Modules system

Only `CONTEXT VOID` modules (both declarations and closed-clause) are supported. This implies that there can be no `HERE` clauses (*holes*) or composition modules. A corollary of this is that there can be no standard prelude or postlude.

The general form of a `DECS` module is

```
DECS decstitle
CONTEXT VOID
[ USE uselist ]:

decsbody

KEEP keeplist
FINISH
```

and the general `PROGRAM` module is

```
PROGRAM progtitle
CONTEXT VOID
[ USE uselist ]

enclosed clause

FINISH
```

### 2.1.2 Standard prelude library

The set of identifiers that would normally be provided by the ALGOL 68 RS library mechanism (eg `print`, `newline`) are not available.

However, an alternative set of operating interface modules have been supplied (see Chapter 3 [Prelude Library], page 7) which may be used automatically via the library mechanism.

A POSIX-compatible standard prelude library may be made available in the future.

A standard prelude, very similar to the standard prelude described in the Revised Report is available provided that the phrase `USE standard` appears in the program header.

### 2.1.3 `FORMAT`

Formatted transput and the modes `FORMAT` are not supported.

### 2.1.4 Selection from arrays of structures

The selection of a field from an array of structures to yield an array is not supported. For example, in the following

```
[2] STRUCT (INT i,REAL r) sarray;
```

```
          REF [] REAL rr = r OF sarray;
```

the selection would be rejected by the translator. It is always possible to circumvent this restriction using loops.

### 2.1.5 XTYPE and YTYPE

The generalised modes `XTYPE` and `YTYPE` are not supported.

### 2.1.6 CYCLE operator

The `CYCLE` operator is not supported.

### 2.1.7 Rowing dynamic procedure results

The translator takes no action to ensure that stack frames of procedures containing dynamic results are not collapsed. Since the translator allocates all dynamically created data, such as arrays, on the heap this is not usually a problem. However, the following extract will give erroneous results due to the rowing coercion applied to `i`

```
     PROC p = REF [] INT: ( INT i ; i );
```

## 2.2 Language extensions

### 2.2.1 First-class procedures

The C code generated by the translator relaxes the usual scoping rules as applied to procedures and allows them to be returned as values from other procedures. The environment in which they were created is preserved for use in subsequent calls, as in the Flex implementation of RS Algol68.

The mechanism relies on all necessary data being allocated on the heap; in particular this applies to data which is local to one procedure but non-local to that procedure which is to be returned as a value for subsequent use. The user must ensure that data is suitably allocated.

For example, when translating the following program extract

```
     PROC a = ( INT i ) PROC INT:
     BEGIN
        INT k := i;
        PROC b = INT: k +:= 2;
        b
     END;
```

`k` has insufficient scope. To ensure correct execution of the program, the user should then amend the declaration of `k` to read

```
     HEAP INT k := i;
```

### 2.2.2 Star-edit preprocessor

A simple conditional compilation feature is built into the translator.

If a source line contains a `*` or `%` in the first column, then the next character is matched with the *starflags* for the current translation, which may be specified by the `-staredit` option (see Section A.1 [Translator options], page 15) or the `A68_STAREDIT` (see Section A.2 [Environment variables], page 17) environment variable.

If a match is found, the source line is translated if the first character is `*`, and suppressed if the first character is `%`.

If no match is found, the source line is suppressed if the first character is `*`, and translated if the first character is `%`.

## 2.3 Language differences

### 2.3.1 `LOC` generators

The translator treats `LOC` generators as if they were `HEAP` generators.

### 2.3.2 C inserts

A C insert is a primary of the form

```
[ mode ] CODE [ ( identifier [ , identifier ... ] ) ] "C code"
```

where

| | |
|---|---|
| *mode* | is the required mode of the primary (`VOID` by default) |
| *identifier* | is an identifier in the Algol68 program which is to be referred to in the C insert |
| *C code* | is the required C insertion using the specified Algol68 identifiers. |

If mode is not void, the C insert must contain an assignment to `RESULT` to represent the result of the clause. The translator encloses the insert in a block and generates pre-processor definitions for the required identifiers and result.

An example of the use of the code construct is

```
INT a, b;
...
IF b = INT CODE (a) "a += 42; RESULT = a*a;"
THEN
  ...
FI;
```

which is equivalent to

```
INT a, b;
...
IF b = ( a +:= 42; a*a )
THEN
  ...
FI;
```

### 2.3.3 `ALIEN` declarations

The ALGOL 68 RS alien construct has the form

```
MODE identifier =
  ALIEN "C identifier"
        "C definition"
      [ "C definition continued" ... ];
```

Note that it is only allowed in the form of an identity declaration.

The `ALIEN` construct is used to give access to functions and data items that are declared externally to ALGOL 68 modules, such as C library routines. The translator generates a C macro definition that maps the C name onto a Ctrans unique name, and copies the C definition unchanged to the output C file.

The C definition part will usually consist of a `#include` of the C header file defining the C identifier being used. If portability is a major consideration, a macro definition may be used to invoke conversion macros (from `include/Atypes.h` on the arguments and result. There are many examples of this in the library prelude sources (`liba68prel/a/*.a68`).

#### 2.3.3.1 Prelude and postlude actions

Because Ctrans is restricted to `CONTEXT VOID` modules, it is impossible to implement preludes and postludes in the traditional way using contexts.

However, a simple mechanism has been provided to allow side-effects to occur before and after the main program.

At the beginning of every program Ctrans generates

```
#ifdef A_prelude
A_prelude(argc,argv,envp); /*envp added by SL*/
#endif
```

and at the end it generates

```
#ifdef A_postlude
A_postlude;
#endif
```

Actions can be specified by providing definitions for these macros. See `liba68prel/a/osshell.a68` for an example from the prelude library.

## 2.4 C naming conventions

The translator and run-time library reserve a set of identifiers for their own use. When writing C code to be combined with a translated Algol68 program (either as a separate module or as a C insert) the reserved identifiers must not be used. They are

- identifiers starting with seven uppercase letters followed by an underscore (the unique prefix, see Section 4.2 [Unique names], page 9)
- identifiers starting with `A_`, `Agc_` or `A68_`

# 3 Prelude Library

The prelude library `liba68prel.a` contains a number of useful procedures and operators that can be used in your ALGOL 68 programs. These replace the ALGOL 68 standard prelude facilties as defined in the Revised Report (but see the book "Programming Algol 68 Made Easy" for a full description of the QAD standard prelude which *does* provide most of the standard facilities).

The library contains

| | |
|---|---|
| `a68config` | operations on the configuration information generated by `a68toc` |
| `cif` | Conversions between C pointers and ALGOL 68 vectors. Some environment enquiries, and `CHAR` constants. |
| `messageproc` | General error handling mechanism. |
| `oscommon` | Filename parsing. |
| `oserrors` | Interpretation of operating system errors. |
| `osfiles` | File and directory handling, I/O. |
| `osgc` | Manipulation of garbage collector parameters. |
| `osif` | Library initialisation. |
| `osmisc` | Current time, CPU time, working directory, execute shell command, read environment variable etc. |
| `osshell` | Program argument handling. Shell meta-character expansion. |
| `ossignals` | Signal handling |
| `strops` | Memory-efficient concatenation of `VECTOR [] CHAR`s. Conversion of `INT` to `VECTOR [] CHAR` (i.e. like `sprintf`) |
| `usefulops` | Miscellaneous mode cheats and string operations. |

You can make these available to your ALGOL 68 compilations by specifying the directory containing the library modules with the `-lib` option of `a68toc`(see Section A.1 [Translator options], page 15) or the `A68_LIB` environment variable (see Section A.2 [Environment variables], page 17.

Consult the prelude library source files (`liba68prel`/*.a68) for further details.

# 4 Using the translator

## 4.1 Input and output files

All output files are created in the current working directory with the same basename as the input ALGOL 68 source file.

Both `PROGRAM` and `DECS` modules generate a C output file (`.c` suffix).

In addition, `DECS` modules generate a module information file (`.m` suffix) which is read by the translator when processing ALGOL 68 source files that `USE` that `DECS` module. The search path for module information files can be specified with the `-dir` option (see Section A.1 [Translator options], page 15) or the `A68_DIR` environment variable (see Section A.2 [Environment variables], page 17).

## 4.2 Unique names

In order to ensure the uniqueness of C identifiers, the translator prefixes ALGOL 68 identifiers of non-local scope with a string of seven upper-case letters followed by underscore. For example, `filename` might be translated into `GBDTFYV_filename`.

The unique prefixes are generated in a fixed sequence starting from a specified seed value determined from the `-uname` option (see Section A.1 [Translator options], page 15).

## 4.3 Multiple-module ALGOL 68 programs

The translator does not implement the full ALGOL 68 RS modules system (see Section 2.1.1 [Modules system restrictions], page 3).

The translator needs to access information about previously translated modules in order to

- check that the identifiers imported from `USE`d modules are used correctly;
- check whether the interface of the `DECS` module currently being translated is identical with a previously translated version.

As mentioned above, the information required by the translator is contained in files with a suffix of `.m`.

Note that a declaration of the form

```
PROC(REF FLOBJECT,VECTOR[]CHAR,INT)VOID
   fl set button shortcut = fl set object shortcut;
```

will not work due to an obscure error in the generated C code.

The form

```
PROC fl set button short cut =
   (REF FLOBJECT ob,VECTOR[]CHAR str,INT showit)VOID:
   fl set object shortcut(ob,str,showit);
```

will work.

When a previous version of a `DECS` module (i.e. a `.m file`) is found, the translator outputs a warning if the current version is incompatible with the previous version.

If the current version is compatible with the previous version, the translator outputs a comment and ensures that externally visible objects have the same C identifiers as in the previous version by propagating the unique prefixes. For a module to be compatible with its previous version

- the keeplist must contain the same identifiers in the same order
- the modes of all kept items must be unchanged

- the C representation of all kept items must be unchanged. This will not be true if, for example,

  ```
  INT i;
  ```

  is changed to

  ```
  REF INT i = j;
  ```

- the C identifier and definition part of kept `ALIEN` declarations must be identical. Thus, changing

  ```
  INT sig_block   = ALIEN "SIG_BLOCK" "#include <signal.h>";
  ```

  to

  ```
  INT sig_block   = ALIEN "SIG_BLOCK" "#include <sys/signal.h>";
  ```

  would render the current version incompatible.

- all modules that are `USE`d by the current module and contribute to its keeplist are compatible with the versions `USE`d by the previous version of the current module.

Note that the last two criteria are additional to those found in other RS Algol68 systems.

If a module is compatible with its previous version then modules depending on the module being translated need not be retranslated. Failure to retranslate modules that depend on a module that has changed its interface will result in subsequent inability to link the compiled objects.

# 5 Building complete ALGOL 68 programs

This section outlines how to compile ALGOL 68 modules and link them together into a runnable program.

The simplest and, perhaps, the best way, to compile and link ALGOL 68 source programs is to use the `mm` Module Manager. See Section "Overview" in `mm`. However, if the Module Manager is unavailable, then you may have to resort to using a `Makefile`.

The instructions are given in terms of rules for GNU make as this is widely available.

It is assumed that these macros are available in the Makefile:-

| | |
|---|---|
| `A68TOC` | the name of the `a68toc` executable file |
| `A68INCLUDE` | the name of the directory containing the Ctrans header files |
| `A68LIB` | the name of the directory containing the prelude library module information files (`*.m`) and object library (`liba68prel.a`) |

## 5.1 Compiling ALGOL 68 modules

Here is a rule that builds an object file from an ALGOL 68 source file.

```
%.o : %.a68
        $(A68TOC) $(A68TOCFLAGS) $<
        $(CC) $(CFLAGS) -I$($A68INCLUDE) -c $(addsuffix .c,$(basename $<))
```

If the prelude library is required, `$(A68TOCFLAGS)` must contain `-lib $(A68LIB)`.

The Ctrans header files must be made available to the C compiler.

There are C compiler options that enable debugging features (see Chapter 6 [Debugging], page 13), such as array bound checking.

## 5.2 Linking ALGOL 68 programs

ALGOL 68 programs are linked in exactly the same way as C programs. Modules must be linked in the following order if the heap management system is to work correctly

1. Object files that do not originate from ALGOL 68 sources (e.g. native C modules)
2. `library/Afirst.o`
3. Object files derived from ALGOL 68 sources.
4. Prelude library (e.g. liba68prel/liba68prel.a). Note that if none of the facilities in this library are used (for example, the `QAD` standard prelude is used instead), then this library should be replaced by that library, `-la68s`.
5. ALGOL 68 run-time library, `-la68`
6. Any other libraries referenced. You will always require `-lm -lc`

# 6 Debugging ALGOL 68 programs

Run-time failures fall into two main classes:-

- the program runs, but does not behave as expected
- the program crashes, usually with a memory access fault (e.g. segmentation fault or bus error), or with an assignment error

## 6.1 Program debugging

Debugging an ALGOL 68 program effectively means debugging the C generated by `a68toc` using your favourite debugger. Although the generated C is not intended for human consumption, it is not difficult to follow with a little practice. It can help to pass the C file through `indent` before you compile it.

The main problem is the unique prefix that is added to most C identifiers which makes tasks such as the setting of breakpoints on procedures more difficult, as you first have to find the C name of the corresponding function. There is no easy general way to do this, but some debuggers (e.g. `gdb`) help by allowing regular expression searches through the symbol table.

Another problem is how to find the ALGOL 68 code that corresponds to a particular line of C, or vice versa. The translator provides help here by identifying the ALGOL 68 source file and line number in the generated C source by either `#line` directives (for every C line) or comments (whenever the ALGOL 68 line number changes). You can choose between these by using the `-noline` and `-verbose` options (see Section A.1 [Translator options], page 15).

## 6.2 Program crashes

Program crashes are usually caused by one of:-

Indexing arrays and vectors with out-of-bound indexes

Run-time bound checking can be turned on by compiling your generated C files with `A68_CHECK` defined (i.e. `CFLAGS=-DA68_CHECK`). There is a performance penalty in doing this, so you may choose to do this only for the module where you suspect the fault lies.

Dereferencing `NIL`

This is usually caused by incorrect deferencing of `REF REF MODE` or `REF REF REF MODE` objects in list processing applications. All you can do is debug the C code with a debugger to find where to apply the appropriate cast in your ALGOL 68.

Scope errors

The scope of variables (that is the lifetime of its memory allocation) is not checked at run time.

Reading and writing through references to stack locations will produce unpredictable results, which you will have to debug by conventional means.

Writing through bad heap references will potentially confuse the heap management system by corrupting its private data structure. You can check if this is the case by either disabling the garbage collector (see Section C.5.6 [Debugging the garbage collector], page 33) or periodically checking that the heap is self-consistent by calling `Agc_check_heap`.

`a68toc` generates a call of the macros `A_PROC_ENTRY` and `A_PROC_EXIT` on each procedure entry and exit. These macros may be redefined (they are empty by default), to call `Agc_check_heap`, or your own debugging code.

Garbage collector errors

See Section C.5.6 [Debugging the garbage collector], page 33.

# Appendix A Summary of translator usage

The translator `a68toc` may be invoked as

```
a68toc [options] sourcefile
```

The behaviour of `a68toc` is also affected by certain environment variables.

## A.1 Translator options

`-verbose`
`-v` Causes some potentially helpful C comments to be generated.

`-noline`
`-n` Prevents the generation of C pre-processor `#line` directives. In its absence `-verbose`, causes comments containing Algol 68 line number and source file to be generated.

`-long`
`-l` Causes the translator to accept `LONG LONG`modes. This is the default.

`-short`
`-s` Causes the translator to accept `SHORT SHORT` modes. This option should be given if the QAD standard prelude is being used.

`-return_structs`
`-r` Prevents generated C functions from returning structure results directly, it uses an extra pointer parameter instead

`-mark_closures`
`-c` Causes closures to be marked with a special value for debugging purposes.

`-stack_closures`
`-C` Causes closures to be (unsafely) stacked rather than heaped.

`-f` *`integer`*
Truncates module names to integer characters long when searching for USE and library modules. eg. -f 8 for DOS FAT files.

`-staredit` *`starflags`*
Sets the required staredit characters to *starflags*. The characters specified in the `A68_STAREDIT` environment variable are overriden unless *starflags* starts with `+` or `-` in which cases the *starflags* are added to, or removed from, those specifed in `A68_STAREDIT`

`-dir` *`directory`*
Add *directory* to the front of the list of directories, specified in the `A68_DIR` environment variable, in which the translator searches for module information (`.m`) files.

`-cdir` *`directory`*
Add *directory* to the front of the list of directories, specified in the `A68_CDIR` environment variable, in which the translator searches for previously generated (`.c`) files when the `-uname cfile` option is specified.

`-lib` *`directory`*
Specifies that *directory* contains the modinfo (`.m`) files corresponding to library modules. This overrides any value set in the `A68_LIB` environment variable.

`-nolib` Turns off the library lookup mechanism.

`-stream` Causes RS stream language to be printed on standard output.

`-dot` Use dot stropping. (Note 2)

`-quote` Use quote stropping. (Note 2)

`-skip` Suppresses run-time errors when missing non-`VOID` `OUT` parts are selected.

`-tilde` Allows tilde as a valid character in identifiers. (Note 2)

`-optbool` Uses the optimised versions of `and(bool,bool)` and `or(bool,bool)`.

`-keeplist`
Allows extension of a keeplist to maintain compatibility. (Note 2)

`-trace` *`level`*
`-t` *`level`* Controls level of expression tree diagnostics.

`-debug_level` *`level`*
`-d` *`level`* Controls level of general diagnostics.

`-debug_module` *`module-name`*
`-m` *`module-name`*
Turns on general diagnostics in *module-name*.

`-cstream file`
`-cstream memory`
Controls whether the translator's C streams are implemented as files or in memory (the default).

`-uname seedfile`
`-uname cfile`
`-uname` *`prefix`*
Controls how the translator's unique name generator is seeded.

`seedfile` Read the seed from the file specfed by the `A68_NAMESEED` environment variable. This is the default. The specified file is updated after translation.

`cfile` The seed is read from the first line of the C file produced by a previous translation of the current module, causing the same seed to be used as when the previous C file was generated. The C file is found by searching the directories specified using the `-cdir` option or `$A68_CDIR` environment variable.

*`prefix`* The specified *prefix* is used as the seed. The unique name should consist of seven uppercase letters. It is an error if the specified unique name contains invalid characters. If more than seven characters are specified, the extra ones are silently ignored. If fewer than seven characters are specified, the unique name is padded on the right with 'A's.

If any error is detected, a warning is issued, and the translator proceeds as if -uname AAAAAAA had been specified.

`-tmp` The generated C file, and C stream files (if `-cstream file` has been specified), are placed in `/tmp` rather than the current directory. This may improve performance on systems where `/tmp` is mapped into virtual memory. Beware of concurrent builds using this option.

`-check` Check ALGOL 68 syntax. No C file is generated, but module information is generated normally.

`-V` Outputs the current version of Ctrans to standard output.

`-version`
`-Version` Outputs detailed configuration information to standard output.

## A.2 Environment variables

`A68_NAMESEED`
The name of the file containing the unique name seed.

`A68_STAREDIT`
The set of required staredit characters. This may be over-ridden or modified by the `-staredit` option.

`A68_DIR` A colon-separated list of directories in which the translator will look for module information (`.m`) files. This may be modified by the `-dir` option.

`A68_CDIR` A colon-separated list of directories in which the translator will look for previously generated (`.c`) files when the `-uname cfile` option has been specified. This may be modified by the `-cdir` option.

`A68_LIB` The name of directory in which the translator will look for module information (`.m`) files corresponding to library modules. This may be overridden by the `-lib` or `-nolib` options.

`A68_GC_DEBUG`
Controls the amount of debugging information produced by the heap management functions of the Algol 68 run-time library. Its values should be as follows:-

| | |
|---|---|
| 0 | No output |
| 1 | Notification of garbage collection |
| 2 | plus small amount fixed amount of additional garbage collection statistics |
| 3 | plus further garbage collection statistics |
| 4 | plus function entry and exit (except Agc_Xalloc) |
| 5 | plus small amount fixed amount of additional information |
| 6 | plus tracing in loops |
| 7 | plus full general tracing (including Agc_alloc) |
| 8 | plus special debugging tracing |
| 9 | plus even more special debugging tracing |

`A68_GC_POLICY`
Controls behaviour when a memory allocation cannot be satisfied. The specified behaviour may be overridden by calling `set_gc_params` from your ALGOL 68 program. Possible values are as follows

| | |
|---|---|
| 0 | The garbage collector is called if the heap usage is above a certain threshold. This is the default. |
| 1 | The garbage collector is never called (i.e. the heap is grown). |
| 2 | The garbage collector is always called. The heap is grown only if absolutely necessary. |

Further policies may have been implemented by additions to `library/Ahpolicy.c`.

# Appendix B Further reading

Guide to ALGOL 68 for users of RS systems, Woodward and Bond, ISBN 0 7131 3490 9.

Revised Report on the Algorithmic Language ALGOL 68, van Wijngaarden et al, Springer-Verlag 1976.

Informal Introduction to ALGOL 68, Lindsey and van der Meulen, ISBN 0 7204 0726 5.

The RS compiler texinfo file (see Section "The RS Compiler" in `rscompiler`).

# Appendix C Implementation guide

The following sections describe in outline the source files that make up the various components of the Ctrans system. Most of these are intended to be easily portable to new architectures, so you should rarely need to examine these unless you want to find out more about how Ctrans is implemented.

The major exception is the part of the garbage collector concerned with the scanning of the ALGOL 68 program's data space. The procedure for finding a program's static data and stack are inherently architecture dependent, but can usually be implemented fairly quickly with code based on the default implementation provided.

Potential porting problems are noted where appropriate. See Section C.7 [Porting], page 34, for a collated checklist of porting concerns.

## C.1 Source directories

Each Ctrans component is contained in a single subdirectory.

`ctrans` `Makefile` and ALGOL 68 sources for the ALGOL 68 to C translator itself.

`include` C header files required to compile C sources generated by `a68toc`.

`liba68prel`
: `Makefile` and ALGOL 68 sources for `liba68prel.a`, a library providing a simple interface to operating system features which may be used by ALGOL 68 programs via the `-lib` option of `a68toc`. This library is used by `a68toc` itself.

`library` `Makefile` and C sources for `liba68.a`, which provide some ALGOL 68 operations that could not be easily generated in-line by the translator.

Also contains `Afirst.c`, which is part of the scheme for finding the ALGOL 68 static data in the address space of an ALGOL 68 program.

## C.2 The translator (`ctrans/a68toc`)

The data-flow of the translator can be summarised in a diagram

```
                ALGOL 68 source
                       |
                       V
                 ---------------
                |               |
    ----------->| rscompiler    |
    |           | (front-end)   |
    |           |               |
 --------        ---------------
| module |         |   |   |   'stream language'
|  info  |         |   |   |
| files  |         V   V   V
 --------        ---------------
   ^  |         |               |
   |  --------->|  compiler     |
   -------------|  (back-end)   |
                |               |
                 ---------------
                   |   |   |
                   |   |   |   C streams
                   V   V   V
                 ---------------
                |               |
                | C stream      |
                | collation     |
                |               |
                 ---------------
                       |
                       V
                  C source file
```

The ALGOL 68 to C translator has been created by incrementally modifying the Multics ALGOL 68 RS compiler. First of all the code for Multics machine code generation were stripped out, then routines were added to generate C streams, keeping the structure of the original code. The structure has changed in that some unnecessary modules have been removed, some have been added to avoid circularities in the dependencies, but most of these changes are low-level detail. The overall structure is still recognisable as the Multics compiler. Some parts of the translator have changed very little in content from the original version, while others have been completely replaced.

`adicops.a68`

This is part of the evaluation mechanism, it contains all the built in monadic and dyadic operators. These are

- monadic operators: `UPB`, `LWB`, `NOT`, `ABS`, `BIN`, `REPR`, `LENG`, `SHORTEN`, `ODD`, `SIGN`, `ROUND`, `ENTIER`, `+`, `-`
- dyadic operators: `+`, `-`, `UPB`, `LWB`, `AND`, `OR`, `<`, `>`, `<=`, `>=`, `=`, `/=`, `*`, `/`, `DIV`, `OVER`, `MOD`, `**`, `ELEM`, `SHL`, `SHR`, `IS`, `ISNT`
- assignment operators: `+:=`, `-:=`, `*:=`, `OVERAB`, `DIVAB`, `MODAB`

`biops.a68`

Generates code for built-in operators (e.g. `BIOP 99`), introduced via the `BIOP` keyword in ALGOL 68 source.

`centities.a68`
: Modes and simple procedures related to manipulating fragments of C code.

`clauses.a68`
: Here are the code generation routines for the following constructs:
  - units, the returned values are stored in temporaries
  - closed clauses, `BEGIN` *serial clause* `END`
  - `IF-THEN-ELSE-FI`
  - `CASE-IN-OUT-ESAC`, both integer and conformity versions
  - optbools, `ANDTH` and `OREL`
  - exits, the `EXIT` bold-word
  - code inserts, the `CODE` bold-word

`common.a68`
: This module is just a keeplist of the modes required by `rscompiler` and the rest of the translator.

`compiler.a68`
: This is the top-level module of the back-end which reads the stream language generated by the front end. It contains a number of recursive routines to activate the various code generation routines in `clauses`, `modules` and the evaluator modules.

`coutput.a68`
: This is a self-contained mechanism for collating the C output streams used by the translator for each procedure level. The C streams may be in memory or in files, controlled by the `-cstream` option of `a68toc`.

`ctrans_version.a68`
: Contains the current version number of Ctrans. `a68toc` generates code that inserts this into the `A_CONFIG_INFO` structure.

`denotations.a68`
: Supplied.

`entryandreturn.a68`
: Generates code for function headers and trailers, and the start and end of modules.

`environ.a68`
: Supplied.

`environment.a68`
: This module is responsible for
  - reading the command line and environment variables, and setting option flags accordingly
  - opening and closing the source file
  - output of diagnostic and error messages
  - aborting on a fatal error

`evalbase.a68`
: This module exists to overcome a circularity in the keeplists of the modules comprising the evaluation mechanism. It is for routines which decompose a value into calls to `evaluate`. It also contains some useful facilities for tracing an evaluation in progress.

`evaluator.a68`

This module contains the lookup table for all the semantic routines in the evaluation mechanism. The actual routines themselves are defined in `adicops` and `operators`.

The main entry point for the module, `evaluate` (defined in `evalbase`) is a `REF PROC` to which is assigned one of `eval trace` or `eval no trace` at run-time. `evaluate` is called at the end of a phrase to read values from the valuestack and produce object code. The semantic routine for each operation is obtained by indexing into the table of procedures given in this module. If it is an arithmetic operator then further selection by the same method takes place in `adicops`.

The C code is yielded by assignment to the parameter of `evaluate`. The operator semantics may emit lines of code directly in order to set up temporary values for the code fragment yielded.

`identifiers.a68`

This contains all the routines to process identifiers. It contains procedures to create new entries in the identifier table (see `idtable`) and procedures to emit:

- identity declarations
- name declarations
- external declarations
- union choice (conformity clause) declarations
- label declarations
- C routines

`idtable.a68`

This contains the declaration of the identifier table which contains all information known to the translator about an identifier. There are routines to initialise the table and to yield a field of the table given an index into the table.

`incenviron.a68`

This contains the mechanism for tracking non-local data—the `ENVIRONSTACK`. The procedures and functions to manipulate the stack of non-locals and to determine the current nesting level are defined here.

`incid.a68`

This contains some predefined constants for use in the identifier table.

`incimperatives.a68`

This is a set of declarations of the format of stream imperatives on the input to the back-end as returned by `next stream imperative`. The stream imperatives output by `rscompiler` are converted into imperatives of the form given by these declarations.

`incinstallation.a68`

This module contains some installation-defined constants, such as the maximum length of an identifier (in the identifier table) the upper bound of the mode table, the length of the unique name prefix, and the largest indexable structure.

`incmode.a68`

This module contains a set of symbolic constants for all the fixed mode numbers (1-31).

`incoperfn.a68`

This contains a set of symbolic constants for some of the indexes to the table of operator semantics in `evaluator`.

`incvalue.a68`
: This contains the definition of a `VALUE` in the evaluation tree. The `EXTRA` field can take on many different types prior to evaluation, and becomes a fragment of C code during evaluation. To pass extra information about a `VALUE` during evaluation, a set of flags is defined here called the *attributes* of a value.

`initialiser.a68`
: Several modules have initialisation procedures. By convention these are called from this module, so that only one call is necessary in the `shell` to start up the back end.

`liblookup.a68`
: This module attempts to map undeclared identifiers onto library modules by decoding the RS compiler keeplists of the modules in the `-lib` directories.

`loads.a68`
: The `compiler` module calls the routines in this module to add a `VALUE` to operand stack, prior to calling `oper` to construct a tree from the operators which act upon them. This is called from `compiler` when a `LOAD` stream imperative is read.

`lookup.a68`
: This is the bold-word lookup table for `rscompiler`.

`loops.a68`
: This contains the code generation routines for loops, both `FOR-FROM-BY-TO-WHILE-DO-OD` loops and `FORALL-IN-DO-OD` types. The routines in this module are called from `compiler`.

`message.a68`
: This contains a procedure which maps an error number onto a message.

`mnemonics.a68`
: Returns a human-readable representation of a stream imperative for debugging.

`modes.a68`
: This contains routines to manage the mode table (see also `incmode`). The mode table is provided as an array by `rscompiler`.

  There are a number of routines of importance in declaring C objects:

  `find deflexed mode`
  : gives the static equivalent of a `FLEX` mode.

  `ctype`
  : gives a C type declaration of the specified mode for use in casts and the declaration of temporaries.

  `write c typedef`
  : returns a C `typedef` statement for a particular mode.

  `declare c temporary`
  : emits a declaration on the local declaration stream and returns a temporary.

`modules.a68`
: This module contains the procedures for constructing a keeplist for USEing a module.

  This module has all the data structures relating to the modules involved in the current compilation, and the procedures which operate on them. It contains code to decode a ‘module specification’ and create new module information for the current module.

`moduletracer.a68`
: Controls module-specific tracing.

`oper.a68`
: This module constructs a subtree from the evaluation stack. The procedure `oper` takes the operands off the value stack and replaces them with an operator. It removes the operands into a new `VALUELIST`, leaving the `OPER` at the top of the value stack pointing at the operands.

`operators.a68`
: This module contains operator semantic routines for the evaluation mechanism. This includes language coercions (widening, rowing, dereferencing, voiding, uniting), procedure calling, selection of structures, assignment, jumps, indexation, trimming, straightening, space generation.

`rscompiler.a68`
: This is the portable front-end of the translator.

`shell.a68`
: The shell controls the streams interface between the front-end and the back-end of the translator. It calls the front-end (`rscompiler`) and back-end (`compiler`) and collates the C streams output by the back-end prior to their being written to the C file by `close coutput`.

`tracer.a68`
: This contains a debugging routine to print out the value tree during an evaluation.

`unionaids.a68`
: Here are several routines to assist in the scanning of united modes in the mode table.

`uniquenameserver.a68`
: This module generates a unique prefix for all the C constants and variables output by the translator. The prefix is an alphabetic string of characters which is incremented as if it were a base-26 number each time a new identifier is required. The starting value of the unique prefix is read from a seed-file at the start of translation, or may be specified in other ways with the `-uname` option. The final value is written back into the file at the end of translation.

`values.a68`
: This module contains a collection of operators and procedures which take a `VALUE` as an argument.

  There are a number of predicates on `VALUE`s, e.g. to determine which constituent of the `EXTRA` field is present.

  A routine used by most operator semantics is `GETCFRAGMENT` which turns most kinds of leaf node in the evaluation tree into a fragment of C code.

  There are two routines which logically belong to the evaluator tracing mechanism here; one to give a human readable representation of a value, the other to give a printable name of an operator.

  Finally, there are a set of functions and operators, to assist in concatenating fragments of C prior to output, and to assign partial results to temporaries. This is the set of operators using the definition of `UVALUE`.

## C.3 The ALGOL 68 header files (`include/*.h`)

`Aassign.h`
: C macros concerned with assignment and copying of vectors and rows. The corresponding functions are in `library/Aassign.c`.

`Acoerce.h`
: Widening of `BITS` to `VECTOR [] BOOL`. The corresponding functions are in `library/Acoerce.c`.

`Aconfig.h`
: Type definition of `A_CONFIG_INFO`. `a68toc` generates C code to initialise this structure. Facilities for manipulating this structure are available in `liba68prel/a68config.a68`.

`Asupport.h`
: Includes all the other header files.

`Alibrary.h`
: C `extern` declarations for all interfaces in the run-time support library `liba68.a`.

`Amacros.h`
: C macros for miscellaneous operations and features, including indexing, uniting, rowing, straightening, trimming and procedure environments.

`Aalloc.h`
: C macros for dynamic space allocation (`HEAP` and `LOC` generators).

`Astrings.h`
: C macros for operations on `[] CHAR` and `VECTOR [] CHAR`. The corresponding functions are in `library/Astrings.c`.

`Atypes.h`
: Type definitions mapping ALGOL 68 modes onto C concrete types; macros mapping between ALGOL 68 primitive modes and C abstract types (such as `size_t`). These should be reviewed whenever porting Ctrans to a new architecture (see Section C.7 [Porting], page 34).

## C.4 The ALGOL 68 run-time support library (`library/liba68.a`)

See Section C.5 [Heap management], page 28, for the majority of the run-time library which is concerned with heap management and garbage collection.

This section outlines the other parts of the library.

`Aassign.c`
: These contain the run-time functions for array and union assignments. The interfaces fall into these groups
  - Vector and row assignments of the data from the data pointer of one descriptor to the data pointer of the other.
  - Vector and row assignments of the data from one descriptor to dynamic storage generated by an invocation of the assignment macro.
  - Union assignments

`Acoerce.c`
: This contains functions associated with coercions. Currently this is only the widening of `BITS` to `[]BOOL`. There is a version of this call for each length of `BITS` value. The widening `BITS` to a row is unusual because it returns a vector. The returned vector is assigned to a temporary provided by the C translator. This assignment is encapsulated in the macro `A_WB_CALL`.

`Aerror.c`
: This contains a routine to display an error message, stop and if possible, core-dump. An error message is currently a literal string of characters.

`Afirst.c`
: Not strictly part of the library, this module is used by the garbage collector to deduce the extent of static data belonging to ALGOL 68 modules.

`Aindex.c` This is run-time support for array trimming and indexing. The calling of the functions to perform trimming is only done if array bound checks are enabled, otherwise the macro implementation of the indexing function is used.

`Amath.c` An implementation of `LONG INT` to the power of an `INT`.

`Astrings.c`
This contains vector and row versions of the string operators `EQ`, `GT`, `+` and `*`.

## C.5 Heap management and garbage collection (`library/liba68.a`)

The garbage collection scheme relies on the ability to find pointers into the heap simply by looking for words (pairs of words in the case of arrays and vectors) with the right format. The major task of porting Ctrans to a new architecture is teaching the garbage collector how to find the data areas to scan.

It is, of course, possible that a non-pointer may be taken to be a pointer resulting in some garbage not being collected. The probability of this happening depends on the format of pointers in a particular architecture, but is usually small. However, it will usually increase as the size of the heap increases. Because of the possible misinterpretation of non-pointers, the garbage collector is not allowed to modify heap references effectively forcing the use of a non-compacting algorithm.

The heap is arranged in segments according to the size of an allocated object (or element size for arrays). The garbage collector can therefore deduce the size of a heaped object given just its address. The garbage collector knows the format of array descriptors (they are marked with a special bit pattern, see `A_GC_MARK` in `include/Aalloc.h`) and hence the number of elements in the heaped array.

Note that an indexable structure has no descriptor, and must therefore be allocated in the heap segment appropriate to its total size, rather than element size.

NB. Unfortunately, due to the limitations of the implementation of the heap, all Algol 68 code can only appear in static code. Only code written in C/C++ can appear in shared-object libraries.

### C.5.1 Internal macro interface

Most of the heap management and garbage collection code should be easily ported. The C modules have been written to an internal macro interface, implemented by the header files, behind which is hidden the architecture dependence.

At the end of each header file is a section of the form

```
#ifndef XXXX
#error XXXX is not defined
#endif
```

which defines the macro interface required to be exported by that header file.

A portable default definition is provided for most macros, protected by `#ifndef` so that it may be overriden with a definition appropriate for a specific architecture.

You can enable the default definitions for a particular header file by defining `A_GC_xxx_DEFAULTS`, where `xxx` is the variable part of the header file name (`area`,`basics`,etc.). `Aharea.h` has a number of logical sections each protected by its own `A_GC_xxx_DEFAULTS` macro. Remember that these defaults can be overriden with architecture-specific definitions that preceed them in the header file.

All defaults in all headers can be enabled by defining `A_GC_DEFAULTS`. This should only be done whilst debugging a port. Once it is established that the defaults are satisfactory, they should be explicitly enabled.

The generic form of the header files is therefore

```
/* architecture-specific section */
#if arch1
#define XXX  xxx   /* specific to arch1 */

#define A_GC_XXX_DEFAULTS   /* use defaults for everything else */
#endif

#if arch2
#define YYY yyy
#define A_GC_XXX_DEFAULTS
#endif

/* defaults section */
#ifdef A_GC_XXX_DEFAULTS
#ifndef XXX
#define XXX default_xxx
#endif

#ifndef YYY
#define YYY default_yyy
#endif

#endif

/* interface check section */
#ifndef XXX
#error XXX is not defined
#endif

#ifndef YYY
#error YYY is not defined
#endif
```

### C.5.2 Garbage collector header files

`Aharea.h` This header file defines the type `AREA` which represents an area of memory to be traced for heap pointers by `Agc_trace`.

It also provides `TRACINGSTACK` which is a stack of `AREA`s (with `PUSH` and `POP` macros) which is used to follow nested heap references. This should be portable.

Finally, it defines `INIT_AREA` and `NEXT_AREA` which return the first and subsequent `AREA`s to be traced. These will have to be implemented for each new architecture (see below).

This header file is divided into a number of distinct sections, each protected by its own `A_GC_xxx_DEFAULTS` macro. These are

`A_GC_AREA_DEFAULTS`
The type `AREA`, plus `STEPAREA`, `NILAREA` and `PTRINAREA`. You should not need to change these for new architectures.

`A_GC_TRACINGSTACK_DEFAULTS`
The type `TRACINGSTACK` and associated operations. Again, You should not need to change these for new architectures.

`A_GC_STATIC_DEFAULTS`
: This is concerned with finding the area of memory occupied by the static data associated with ALGOL 68 modules so that it can be traced for heap references. The default mechanism looks at the addresses of symbols defined in `library/Afirst.o` and the run-time library (see `Agc_last_statics` in `Ahtrace.c` to deduce the extent of the static area. The mechanism relies on the linker and loader locating static data incrementally in the same order in which the modules are presented to the loader. Then all that is necessary is to ensure that the ALGOL 68 program is linked with all ALGOL 68 modules (including the prelude library `liba68prel.a`) sandwiched betweeen `Afirst.o` and `liba68.a`.

`A_GC_STACK_DEFAULTS`
: This is concerned with finding stack frames so that they can be traced for heap references. It is essential that all registers that may contain heap references are flushed, or inspected some other way. It is impossible to do this in a portable way, so *there is no default implementation.*

  It is not usually necessary to understand the red tape associated with each stack frame—it is normally safe to regard the whole stack as a single `AREA`.

  `Agc_main_frame` is initialised by `Agc_startup` to the address of `argc`, which may help in detecting the last stack frame to trace.

`A_GC_TRACINGAREA_DEFAULTS`
: This contains the definitions of `INIT_AREA` and `NEXT_AREA`, usually in terms of operations defined in the previous two sections.

`Ahbasics.h`
: This contains miscellaneous type definitions and general macro definitions which should not require modification for porting.

`Ahbitmap.h`
: This file contains some general bit manipulation macros adapted from a set of public domain macros, plus some macros specific to the bitmaps used to mark used areas of the heap in `Agc_trace`. All these macros should not require modification for porting.

`Ahclr.h`
: This file contains macros for the clearing of elements on allocation, freeing and so on. There are various compile-time options that control when clearing is done, and what bit pattern is used on initialisation. The default is to zeroise elements. All these macros should not require modification for porting.

`Ahdesc.h`
: This file contains portable macros for decoding ALGOL 68 array descriptors. No modification should be necessary.

`Ahptr.h`
: This file contains pointer manipulation macros. It defines the following macros, which may require modification

  `ALIGN_NEXT`
  : Aligns pointer to appropriate boundary for specified object size.

  `VALIDPTR`
  : Decides whether a word could possibly be a pointer (prior to checking that it points into the heap).

  `RESIZE1ALLOCATION`
  : Modifies the size of a single object to be allocated. This must take into account any alignment constraints. It also allows object sizes to

be rounded up so that fewer heap segments are required in total (fewer different sizes of objects), at the expense of wasting space within segments.

`RESIZENALLOCATION`
: Similarly for array allocations.

`Ahseg.h` Type definitions for segments (`SEGCTL`), free list elements (`EL` and operations on them. These should not require modification for porting.

The pointers chaining together free list items must be disguised so that they are not traced. Such pointers are defined as `CODEDPTR`s, and the operations `ENCODEPTR` and `DECODEPTR` convert them from/to real addresses. These should be reviewed when porting to a new architecture.

### C.5.3 Garbage collector modules

`Ah1alloc.c`
`Ah1alloc.h`
Implements a fast allocation cache for single small objects. The cache is accessed via in-line code (see macro `A_GC_1ALLOC`), and slots are filled on demand using `Agc_nalloc`. The cache is not traced during garbage collection, so the cache slots must be emptied after each garbage collection.

`Ahalloc.c`
`Ahalloc.h`
This module implements the following interfaces for heap allocation

`Agc_nalloc`
: for allocating arrays

`Agc_1alloc`
: for allocating single objects

`Agc_alloc4`
: optimised for allocating arrays of words

`Ahcollec.c`
`Ahcollec.h`
This module implements the entry point to the garbage collector, `Agc_collect`, and an initialisation function, `Agc_startup`, which is called by generated C code at the start of `main`.

`Agc_startup` is typically used to remember the base of `main`'s stack frame to limit the extent of pointer tracing in the stack.

The main functions of the garbage collector (marking used data, and freeing unused data) is imported from `Ahtrace.c` and `Ahsweep.c`.

`Ahdebug.c`
`Ahdebug.h`
This module implements various aids to debugging, including a publicly accessible heap consistency check, `Agc_check_heap` (see Section C.5.6 [Debugging the garbage collector], page 33).

`Ahparam.c`
`Ahparam.h`
This module collects together the parameters that affect the operation of the garbage collector (see Section C.5.4 [Garbage collector parameters], page 32). A function, `Agc_param`, is provided to inspect and modify these which is accessible from ALGOL 68 through the prelude library procedures `get_gc_param` and `set_gc_params`.

`Ahpolicy.c`
`Ahpolicy.h`
: When the allocator cannot satisfy an allocation request it must decide whether to proceed by extending the heap (i.e. allocating a new segment), or invoking garbage collection. It decides by invoking a function that implements the required policy. This module implements a set of three simple policies that may be selected at run-time either via the `A68_GC_DEBUG` environment variable, or from within the ALGOL 68 program via the `set_gc_params` procedure. Further policies may be added if needed.

`Ahstats.c`
`Ahstats.h`
: This module defines various statistics that can help in tuning the heap management for a particular application. Some statistics that are expensive to collect are only collected if the library is compiled with `A_DEBUG` defined (see Section C.5.5 [Garbage collector statistics], page 33.

`Ahsweep.c`
`Ahsweep.h`
: This module implements `Agc_sweep` (internal to the library). This function traverses the list of heap segments examining the bitmap (set up by `Agc_trace`) which marks used areas of the heap. In each segment, unmarked areas are returned to the free list, merging contiguous free list elements when possible.

`Ahtrace.c`
`Ahtrace.h`
: This module defines the function `Agc_trace` which scans areas of memory for potential pointers into the heap. The areas to be traced are returned by `INIT_AREA` and `NEXT_AREA` which are defined in `Ahseg.h`.

## C.5.4 Garbage collector parameters

A number of parameters that affect the behaviour of the garbage collector can be inspected and/or modified at run-time using the `Agc_param` run-time library function in `library/Ahparam.c`, or the prelude library procedures `get_gc_param` and `set_gc_params`.

Each parameter is identified by a name string as follows

'`MAX HEAP SIZE`'
: The maximum number of bytes to be allocated to the heap. The default value is 1Gb, which effectively means as much memory as can be obtained via `malloc`.

: It may be set to a smaller size if you wish to constrain the process size, e.g. on non-virtual memory architectures.

'`MIN SEGMENT SIZE`'
: The minimum number of bytes to be allocated to a segment. The default value is 4kb. This parameter interacts with the adjustment of allocated object sizes (`RESIZE1ALLOCATION` and `RESIZENALLOCATION` in `Ahptr.h`), to determine the number and size of segments that are allocated, and hence the fragmentation properties of the heap.

'`POLICY`'
: Controls the behaviour when a memory allocation cannot be satisfied (see Section A.2 [Environment variables], page 17, `A68_GC_POLICY`).

The following parameters affect the behaviour of the default policy

'`MIN HEAP SIZE`'
: The minimum number of bytes to be allocated to the heap.

'HEAP INCREMENT'
: The amount by which the collection threshold should be increased after each garbage collection. A positive value indicates an absolute number of bytes. A negative value indicates a fraction (denominator of 256) of the current number of bytes in use. Thus, a value of `-128` indicates that the threshold should increase by half the amount currently in use. The default value is `-256`, i.e. increase the threshold by the amount currently in use.

'COLLECTION THRESHOLD'
: Collect garbage if the number of allocated bytes exceeds this value. This is automatically adjusted at the end of every garbage collection. Its initial value is zero, so the first garbage collection is triggered when the amount in use exceeds `MIN HEAP SIZE`.

### C.5.5 Garbage collector statistics

The following C integers are declared in `Ahstats.c` and maintained by the heap management system

`Agc_collections`
: The number of garbage collections performed.

`Agc_s_grabbed`
: The number of heap segments currently in use.

`Agc_b_grabbed`
: The number of bytes occupied by the segments currently in use.

`Agc_b_allocated`
: The number of bytes in use by the ALGOL 68 program.

If the run-time library was built with `A_DEBUG` defined, the following extra statistics are available

`Agc_allocations`
: The total number of allocations performed.

`Agc_s_examined`
: The number of heap segments examined in satisfying those allocations.

`Agc_e_examined`
: The number of free list elements examined in satisfying those allocations.

`Agc_s_freed`
: The number of heap segments freed (i.e. returned to the operating system).

for each object size up to 512 bytes

| | |
|---|---|
| `alloc_1` | the number of calls of `Agc_1alloc`. |
| `refills` | the number of times the fast allocation cache has been filled. |
| `alloc_n` | the number of calls of `Agc_nalloc`, including those to fill the fast allocation cache |
| `elements` | the total number of array elements requested in calls of `Agc_nalloc`. |

### C.5.6 Debugging the garbage collector

By defining `A_DEBUG` when you build the library, you can turn on extensive consistency checking and diagnostic reports within the garbage collector.

The quantity of diagnostics is controlled by setting `Agc_debug_level` by one of

- setting the `A68_GC_DEBUG` environment level before you run the ALGOL 68 program

- calling set_gc_params from your ALGOL 68 module
- using a debugger

You can disable garbage collection at any time by calling the prelude library procedure disable_garbage_collector (module osgc) from your ALGOL 68 code.

## C.6 The ALGOL 68 prelude library (liba68prel/liba68prel.a)

The sources of these modules assume the existence of an XPG3 compliant library. Porting to other operating system environments will require changes to the ALIEN definitions of the prelude library.

## C.7 Porting checklist

include/Atypes.h
: Review the type definitions and mapping macros.

library/Ahbasic.h
: Review this file. You should not need to change any definitions.

library/Ahptr.h
: Review this file bearing in mind alignment constraints.

library/Ahseg.h
: Review CODEDPTR and its operations.

library/Aharea.h
: Implement INIT_AREA and NEXT_AREA. You can test allocation independent of garbage collection by defining these to return NIL_AREA, and running your program with garbage collection disabled by setting the A68_GC_POLICY environment variable to 1.

liba68prel/*.a68
: Review operating system interfaces used in ALIEN declarations. Check ossignals.a68 particularly. Ensure that the signals are declared in the right order for signal_facility.

# Option index

# Symbol index

# Concept index

## Short Contents

# Table of Contents