ANSI C (Release 3) for the RISC OS operating system

Release Note

The ANSI C package

The following items are included in the ANSI C package:

- the ANSI C Guide
- three discs:

  Disc 1: Work disc

  Disc 2: Library support disc

  Disc 3: Reference disc
- four reference cards:

Card 1:	side 1: C compiler directory structure
	side 2: Contents of release disc 1
Card 2:	side 1: C compiler options
	side 2: Contents of release disc 2
Card 3:	side 1: ASD commands
	side 2: Contents of release disc 3
Card 4:	side 1: RISC OS library structure
	(side 2 is blank)

- Release Note (this leaflet), which includes the supplement entitled *Release 3 of C for RISC OS: additional notes*.
- a registration form and pre-paid envelope.

If any of these items is missing or damaged, notify your supplier immediately.

Configuration details

The C compiler system will run on any Acorn computer which supports the RISC OS operating system.

Network installation

You should obtain a site licence from Acorn Computers to use this software on an Econet network. It is a good idea to install the software at a time when your network is not being subjected to heavy use.

Notes

Errors and infelicities in the compiler and libraries that were known about at the time of release are given in the second part of this Release Note. The following observations should also be noted:

- The `!Cstart` program on Disc 1 is to serve as an example and to get started on Disc 1. It is not designed for general use and you would need to edit it for that purpose.
- It is not possible to nest invocations to the Acorn Symbolic Debugger (ASD); recursively calling ASD may result in a software crash. However, this is not something that you would normally find useful.
- The ASD command `where` has a problem addressing contexts further down the stack than the current context, so that contexts specified using the \–1 or \2 notation produce error messages. As a temporary solution, use

  ```
  context \-1;
  where;
  context \+1
  ```

  rather than

  ```
  where \-1.
  ```
- The compiler works by function rather than source line, and moves code around. For this reason, you may encounter a mismatch between reported line numbers from ASD and the actual line number in the source. However, this will not be encountered frequently, and when it does occur the actual source line will usually be adjacent or close to the reported line number.
- The `-gv` compiler option (for debugging with information on local variables) will fail when used on desktop applications, giving a `Fatal internal error` report. This is accompanied by an error message in a box of asterisks which you can ignore in this case. For example, `cc balls64 -gv -I$.RISC_OSlib` will generate the error.
- It is not possible to use `ptrace` selectively, though use of macros and conditionally-executed breakpoint commands aften allow the effect of selective `ptrace` to be simulated.
- The compiler is not as efficient in code generation when working in pcc mode (using the `-pcc` option) as it is in normal ANSI mode (the default).
- If you are using a multiscan monitor and compiling a desktop application, set the monitor type to 0 to allow enough memory for the compilation. Use the *Configure MonitorType command. After compiling, change the monitor type back to 1.

Customer Support

The Acorn Customer Support Service operates through our international network of authorised dealers and distributors. If you encounter problems with your Acorn software package, you should therefore contact your supplier. To help your supplier to provide an efficient support service, please have available:

- the complete package
- all discs involved
- details of the hardware and software used. For example: Archimedes 410/1 with RISC OS 2.00 and ANSI C version 3.00.
- if possible, some written evidence of the problem.

Release 3.00 of C for RISC OS: additional notes

Contents

Introduction

You will find these notes useful for reference if you are upgrading from a previous release of the Acorn C compiler, since they give details of changes in the C compiler system from Release 2 to Release 3. An overview of this subject is given in the Guide, in *Appendix A: New features of Release 3*, but this document goes into greater detail, is primarily concerned with software functionality, and tackles issues at a more technical level.

Known faults in the compiler and libraries are covered towards the end of these Notes.

The product

Release 3 of C contains all the basic tools you require to develop programs in C for RISC OS (except for a text editor, which is supplied with RISC OS). With Release 2, a text editor and the Software Developer's Toolbox had to be purchased separately.

Integration with the RISC OS desktop

Release 3 of C contains high-level interfaces to the RISC OS windowing system to make it easier to write window-based applications for RISC OS which have the same look and feel as Acorn's Edit, Paint and Draw applications distributed with RISC OS. However, the individual tools in Release 3 are still firmly command-line oriented.

Major new features

The key additional features of Release 3 are:

- conformance with the latest ANSI draft (December 1988); see the section below entitled *ANSI Conformance*
- RISC OS library extensions
- support for developing the following types of program for RISC OS:
  - desktop applications
  - relocatable modules
  - overlaid programs

- improved portability to and from RISC OS
- new software tools and enhanced tools that were previously part of the Software Developer's Toolbox (in particular, the Acorn Symbolic Debugger, which now supports low-level debugging as well as high-level symbolic debugging).

New Procedure Call Standard

The Procedure Call Standard (PCS) obeyed by code compiled by the C compiler has been changed. The change was made to introduce support for writing modules which can run in SVC mode.

Object code compiled by Release 3 compilers is incompatible with earlier object code and libraries, whether compiled or assembled. It is not possible to warn of this at link time without causing annoyance to at least as many users as might be helped by a warning.

Old binaries which use the shared C library can still run using the new shared C library; new binaries cannot run using the old shared C library. Old binaries suffer a performance degradation, but this is imperceptible interactively.

The compiler has a command line option (`-zkA`) to compile code which uses the old procedure call standard. This is provided to allow those with a large investment in assembly language to make a gradual transition to the new PCS.

Obsolete and obsolescent features

Acorn Object Format (AOF)

The following features have been removed from the definition of AOF and are no longer supported by the Acorn linker:

- Absolute areas (there was never any linker support for them)
- the Position Independent Code (PIC) attribute (rarely used)
- the PC-relative relocation of 11-bit LDR offsets (never supported by the Acorn linker)
- the three AOF image formats
- alignment of AOF AREAs to other than 4-byte boundaries.

Dbug and AOF images

Dbug (a low-level debugger supplied with the Software Developer's Toolbox) is declared obsolete. Its functionality is subsumed by the Acorn Symbolic Debugger (ASD) supplied with Release 3 of C.

AOF Type-1 images are declared obsolescent. Type-1 Images can only be run under Dbug and Dbug is the only RISC OS application to use this format. For a transition period, the linker can still generate AOF Type-1 images (using `link -dbug`). This facility may be withdrawn in a future release of C.

cc -arthur/cc -super/ArthurLib

Cc `-arthur` and `cc -super` are now synonyms and both ArthurLib and SuperLib are declared obsolescent (ArthurLib is now supplied only with RISC OS; SuperLib only with SpringBoard Brazil). The functionality of SuperLib (the Brazil kernel interface library for SpringBoard) and the low-level functionality of ArthurLib (the Arthur 1.2 interface library) is subsumed by the kernel of the C library. The high-level functions of ArthurLib are now better provided via the RISC OS library on which the RISC OS applications Edit, Paint and Draw are based.

Further use of ArthurLib is deprecated; it will be withdrawn from the next release of C.

Additions, changes and upgrades

Low-level debugging and the Acorn Symbolic Debugger (ASD)

An (AIF) image containing low-level debugging data can now be run directly and runs at the same address whether it is being debugged or not (previously such an image could only be run under Dbug and was linked at a different base address when being debugged). Programs can now be debugged at the machine code level using ASD, supplied with this release of C.

Squeezed images can now be debugged using the Acorn Symbolic Debugger (ASD).

Self-relocating AIF images

A self-relocating AIF image can now be generated which, when entered, relocates itself at the top end of applications workspace, leaving a specified amount of working memory above itself (for its own heap, stack, etc). ASD, supplied with Release 3 of C, is just such an image.

Support for overlays

The linker now supports a static overlay scheme much like those common under MS-DOS. Overlays need no special support from the compiler, but can benefit from compiling each function into its own AOF AREA. This feature, enabled by `-zO`, causes AREAs called `C$$<fn_name>` to be generated, rather than a single area called `C$$code`. This allows a finer assignment of functions to overlay segments than would otherwise be possible (except by restructuring the source code). Code compiled `-zO` will suffer an increase in size because of the reduced sharing of literal pools.

New image format for overlay segments and ROM-ing

The linker now supports a plain, unadorned binary image type (in support of overlay segments and of putting code into ROM).

Improved system( ) function

The C library `system()` function has been improved so that an application can transparently call another application as a subprogram using `system("program")`. The Release 2 variants `system("CALL:prog")` and `system("CHAIN:prog")` are still available, but the default is now `"CALL:"` (`CHAIN:` causes a transfer of control with no return to the calling application).

`System("CALL:...")` is implemented by copying the caller to the top of application workspace then running the callee in the remaining space. This may not work if there is insufficient memory of if the wimp slot is too small. In these cases, `system()` returns `_kernel_ERROR`. If the call succeeds, the called application indicates success or failure by setting `Sys$ReturnCode` 0 or non-0, respectively (`exit(0)`, or return 0 from `main`, sets it to 0 for you). `System()` returns the value set by the called application. All programs linked with the C library now set `Sys$ReturnCode` to the value passed to `exit()` or returned from `main()`. If the called sub-program is a built-in command which fails, `system()` now returns `_kernel_ERROR`.

If `system()` returns `_kernel_ERROR`, for any reason, `_kernel_last_oserror()` returns a pointer to a RISC OS error block describing the fault.

Improved command line redirection for programs written in C

The UNIX and MS-DOS 'standards' for the redirection of `stdin`, `stdout` and `stderr` are now supported by the C library. Redirections of the form `foo { > out }` still specify RISC OS's standard redirection (all terminal output sent to `out`, lines termined by CR LF) whereas a redirection of the form `foo > out` redirects only `stdout` to the file `out` (no additional CRs inserted). The complete set of simple redirections of `stdin`, `stdout` and `stderr` as recognised by UNIX's C shell (csh) and Bourne shell (sh) and MS-DOS, are recognised by the C library. For example:

`foo >& 'out'`	(`stdout` and `stderr` to 'out', csh notation)
`foo > out 2>&1`	(the same, sh notation)
`foo 2> out`	(`stderr` only sent to 'out', sh notation).

`stdout` and `stderr` are now separately redirectable, overcoming a major deficiency of Release 2 of C.

Use of file descriptors other than 1 and 2 is not supported in sh-like stream redirections.

As all the tools in the C release, including the C compiler itself, use the C library, these rules apply to them with equal force.

Special characters in quoted command line arguments

An argument enclosed in double quotes is treated as a single argument word. Such arguments may contain spaces, '<'s, '>'s, etc. Within a quoted word, '"' is represented by `\"` and '\' by `\\`. This allows an argument to `main` to contain spaces and apparent redirections. For example: `prog "> out" arg2 arg3`.

ANSI conformance

Release 3 of C seeks to conform to the December 1988 draft ANSI specification for the programming language C. Previous releases sought to conform to the October 1986 draft. No attempt has been made to conform to intermediate drafts and, in fact, the December 1988 draft is, in most visible respects, closer to the previous release of C than any of the intermediate specifications.

As yet there is no formal standard for C and therefore no formal way to measure compliance with it. Acorn has been using the first draft C Validation Suite (CVS) as distributed by BSI. Because of the status of the draft standard at the time of going to press, it is not possible to claim full draft ANSI conformance or seek validation for this release.

A list of known violations of the current draft standard is geven in the section entitled *Known faults in Release 3 of the C compiler and library*.

C language changes since Release 2

In most respects, changes since Release 2 have been minor and will go unnoticed by most users. The following areas are most worthy of note:

- external linkage and tentative definitions
- mixing old-style (K&R/pcc) and ANSI function prototypes.

External linkage and tentative definitions

In ANSI mode, the compiler now conforms strictly to the ANSI specification. Release 2 contained bugs in this area and earlier releases did not permit such ANSI-isms as: `static int x; static int x = 2; extern int x;`. These changes are unlikely to cause problems to users as the new specification is, in general, more permissive than either the previous release or old style C. Furthermore, while the compiler is fussy about diagnosing errors in this area, it can recover and generate sensible code (ie what a user should expect) in most error situations. When it cannot, it refuses to generate an object module.

Mixing old-style (K&R/pcc) and ANSI function prototypes

In previous releases, an in-scope, old-style function *definition* was treated (erroneously) as if it were an ANSI prototype, causing actual arguments to calls to the function to be coerced appropriately. Now, such arguments are warned of, but not coerced.

In fact, this change causes little trouble: the greatest potential for trouble exists in an implementation in which `sizeof(int) < sizeof(long)` or `sizeof(int) < sizeof(void *)`, or which did not widen char, short and float arguments. Earlier releases were free of these traps and, in ANSI mode, strict about the conformance of function declarations and function definitions, so there are no new problems. Indeed, an old pcc-mode trap disappears (in which calls compiled before the compilation of an old-style definition could be compiled differently to calls made after the definition).

Pcc-mode compatibility

Pcc-mode compatibility has been improved in a number of minor ways: for example, chars are now signed by default (despite the code-space penalty thereby incurred on the ARM); `sizeof()` returns an int (*not* an unsigned, as in ANSI mode); calls to old-style functions no longer have their actual arguments coerced to the formal argument types, as used to be (erroneously) done when an old-style function definition was in scope; several minor preprocessor incompatibilities have been removed.

External linkage in pcc mode has been disentangled from ANSI mode external linkage and made compatible with that defined by the 4.3 BSD VAX C compiler (cc) and system loader (ld). Linkage errors are now signalled exactly when pcc (or as) would signal an error. However, the compiler is, in general, more tolerant than the VAX pcc and will repair most errors in a sensible manner.

Interactive input to cc

`cc -E`, `-M` and `-S` accept input from `stdin` (eg interactively) if '–' is specified as the input file.

Support for maintaining makefiles

`cc -M` generates, on `stdout`, a list of all the files included by the source file being compiled. In conjunction with I/O redirection, this is useful for generating makefiles.

Using ObjAsm via cc

The compiler can now invoke the assembler ObjAsm to assemble an assembler source provided (i) ObjAsm can be found on the Run$Path and (ii) there is sufficient main memory to support the co-residence of cc and ObjAsm (this probably requires an A440). Thus the following style of usage now works:

```
cc -o command command.c net.s util.o
```

(compile `c.command` to `o.command`, assemble `s.net` to `o.net`, then link `o.command`, `o.net`, `o.util` and the C library to make a program).

Location of standard libraries

The assumed location of the C library (if C$libroot is unset) is now `$.clib` (it was previously `$.arm.clib`). The default C library is now `$.clib.o.stubs`, the interface to the shared C library module.

Return codes and error behaviour

The compiler now sets a non-zero return code if there are any errors during compilation, where previously it only did this for serious errors. Similarly, it does not perform a (requested) link step if there are errors, where previously linking was omitted for serious errors only. However, an object file is still produced in the presence of errors so the following is acceptable:

```
cc foo.c  fails with errors
.... ;    user checks that recovery is what was expected...
cc foo.o; and does the omitted link step.
```

This is more like BSD UNIX behaviour and integrates better with the new Acorn Make Utility (AMU).

Common Subexpression Elimination (CSE)

A new optimisation, CSE, has been implemented and is enabled by default. Occasionally this may cause problems when compiling big procedures on small machines, as more memory is required, both for CSE itself and for the subsequent register allocation phase. CSE can be disabled from the command line by `-zpz0` (or, equivalently, by #pragma `no_optimise_cse`). In general, the problems now associated with CSE are less than those associated with register allocation in Release 2.

Usually, CSE makes compiled code run faster; often it turns out smaller as well. Although in theory it can become bigger, this has rarely been encountered. In rare cases, code may run more slowly when CSE is enabled, as CSE sometimes seriously perturbs register allocation. Bear this in mind if you are particularly concerned about performance in a critical routine. Usually CSE yields improved code, so it is enabled by default.

Cross-jumping

Cross-jumping is a space-saving optimisation which shares common code sequences at the cost of additional branches. In general, its run-time cost is negligible and it is therefore enabled by default. If performance is your priority, note that in rare pathological cases cross-jumping can slow code by an arbitrary amount (eg a cascade of N common tails within an inner loop can insert N additional branches into the critical path). Cross-jumping can be disabled from the command line (`-zpj0`) or by #pragma `no_optimise_crossjump`.

Sharing of literals

The sharing of literals – especially character string literals – has been improved since Release 2. Identical instances of a literal are now shared if they occur within an addressing span (1020 words or so; the use of `cc -z0` to compile one AOF AREA per function perturbs this).

Global register variables

It is now possible to force a global variable into a register. The cost is the loss of a register allocatable to a local variable. In general you are not recommended to use more than two global register variables. A small example shows how to do this:

```
struct frame {struct frame *fp; int sp, pc;};       /* example structure*/
#pragma -r1                        /* put the next global variable in the*/
extern int sp;                     /* 1st (integer) global register.    */
#pragma -r2                        /* put the next global variable in the*/
extern struct frame *fp;           /* 2nd (integer) global register.    */
#pragma -r0                        /* no more use of global registers.  */
```

Support for writing RISC OS modules in C

Code implementing a module must be specially compiled `-zM`, so that its static data will be relocatable at run time. It must also be specially linked (`link -M`) together with the output of the C Module Header Generator (CMHG).

C library definition changes since Release 2

The library has suffered many detailed changes as a result of being brought into line with the latest ANSI draft standard. For example, some objects previously defined as macros (eg `size_t`) are now no longer so defined, but are defined by typedefs. Some macros have been renamed (eg `OPEN_MAX` is now `FOPEN_MAX`, `CLK_TCK` is now `CLOCKS_PER_SEC`) and some behaviour has been clarified. In general these changes should cause minimal difficulties.

Libraries, Procedure Call Standard and tracebacks

The linkable library AnsiLib uses the new Procedure Call Standard. A version (called `AnsiLib_A`) is supplied which is compatible with the old APCS-A procedure call standard.

Linkable libraries are compiled to allow tracebacks through them (functions are named). The shared C library module does not support function names, so tracebacks will be meaningless (this saves some space in the shared C library). Old binaries which use the Release 2 shared C library will still run using the Release 3 shared C library, but with somewhat degraded performance (generally not perceptible interactively).

Library performance improvements

Divide has been speeded up significantly.

The Acorn Make Utility (AMU)

AMU has been bundled with this release and upgraded to implement most of the facilities of UNIX's 'make' command, including:

- incremental execution of commands (this was previously done offline, and offline execution via an EXEC script has been retained as an option)
- stop on error (the default; controllable with command prefixes '@' and '–' and the AMU command line flags `-i` and `-k`)
- VPATH
- rule patterns such as `.c.o:; $(CC) $(CFLAGS) -o $@ c.$*)` and `.SUFFIXES`
- acceptance of most UNIX and MS-DOS file-names (which are translated to the nearest obvious RISC OS equivalents).

The Acorn Symbolic Debugger (ASD)

ASD has been upgraded significantly and is now supplied with Release 3 of C. The following improvements are incorporated:

- Several dozen reported problems with Release 2 of ASD have been fixed.
- There is better (less rigid) use of the available memory.
- There is support for low-level debugging (subsumes Dbug's functionality).
- There is an expanded command set with better abbreviations and command macros.
- There is support for remote debugging via an RS232/RS423 link from another computer or computer terminal.
- There is partial support for debugging overlaid images.

The C Module Header Generator (CMHG)

CMHG is new to Release 3 of C. It is a special-purpose assembler for making a RISC OS module header for a module written in C. It also makes the entry veneers needed to interface the module header to the module body.

Squeeze, an image compactor

Squeeze encodes a binary program image, creating a new, smaller image that can unsqueeze itself when it is entered. Unsqueezing is extremely fast: faster than loading the additional bytes from a Winchester disc. In general, Squeeze reduces C programs to just over half their initial size (it can do better if they contain large arrays of zeros).

Dialect translation utilities

Dialect translation utilities `topcc` and `toansi`, as used by Acorn, are supplied in source form. These can be used to help translate code between ANSI and pcc-style dialects of C. Acorn, for example, keeps the C compiler in ANSI C, but in a form that can be mechanically translated to pcc-style C by `topcc` (so that it can be compiled on UNIX systems that do not support ANSI C). `topcc`, of course, cannot do a complete job, but it helps. Similarly, `toansi` can assist with bringing old-style C sources into Acorn's ANSI C world (this is an alternative to using the compiler's `-pcc` mode).

Known faults in Release 3 of the C compiler and library

Faults in cc's preprocessor

Historically, preprocessing has been the least well specified aspect of compiling a C program. This has been reflected in the evolution of the draft ANSI standard for C and is reflected in the relatively large proportion of preprocessor faults in the list of known faults.

Faults which may cause valid programs to be rejected

The preprocessor predefines ARM and arm, which is forbidden by ANSI. For the purposes of Release 3, this predefinition is disabled by `-fussy`. Users should beware that a future release will predefine `__arm` and `__ARM` instead.

There is some unexpected token 'glueing' by the preprocessor (in effect, its behaviour is too like that of UNIX's cpp, which will cause no surprise to users of character processing preprocessors like cpp, but may cause surprises to users of token-oriented preprocessors in which there can be no accidental glueing caused by token juxtaposition). For example:

```
#define f(x) +x
    f(+y)
```

yields ++y rather than + +y as required by the draft standard.

Faults which may cause invalid programs to be accepted

A preprocessor redefinition of the form: `#define A 1 #define A() 1` is not faulted (the draft standard forbids redefinition of non-function-like macros as function-like macros, and vice versa). However, warning is always given of redefinition to a non-identical value. For example: `#define A 2` elicits a warning.

Similarly, redefining a macro with a different number of arguments, or a different spelling of the formal arguments, is not faulted (though it *is* warned of if the new macro body is not identical to the old one). The preprocessor does not fault supplying too few arguments to a macro invocation (though it does fault supplying too many).

The preprocessor will erroneously 'stringify' *any* token, not just macro parameters (eg `#define A 1; #define B(a) #A; B(2)` wrongly yields `'A'`, rather than a diagnostic relating to `#A`).

The preprocessor erroneously ignores a `##` at the start or end of a replacement list and this should be faulted. An empty `#line` directive is not faulted (eg `#line`).

The class of expression allowed in a `#if` preprocessor directive is not sufficiently restricted. For example, the compiler allows almost any expression it can evaluate at compile time whereas it should fault, amongst others, casts and `sizeof()` (For example, `#if (int)1==1` is accepted silently, as is `#if sizeof(char)==1`, but both should be faulted). These laxities cause anomalies between cc –c and cc –E.

A preprocessor directive like `#^Ldefine` is not faulted (only space and tab should be allowed between # and `define`).

Other cc faults

Faults which may cause valid programs to be rejected

Multiple local declarations are mishandled and faulted with a 'duplicate definition of <thing>' message. For example: `int f() {inf f1(); int f1();...}`.

In -pcc mode, a bitfield which is aligned on a non-word boundary will have incorrect access code compiled for it. For example:

```
struct bad_bitfields {
  short s;
  int bf1:1;      /* bad code compiled for accessing this */
  int bf2:1;
```

```
    /*...*/
};
int bf1(struct bad_bitfield bbf) {return bbf.bf1;}
```

This problem cannot occur in ANSI mode.

During type checking, an old-style function definition compared against an ANSI declaration involving qualified types is erroneously faulted (though correct code is generated) because, for example, `int f(const char *);` doesn't match `int f(s) char *s; {...}`. These cases are really marginal, especially since those involving default argument widening (such as `int f(int); int f(ch) char ch; {...}`) *are* correctly matched.

In `char x; int y = sizeof(0, x); sizeof()` returns 4 (char promoted to int in expression context) rather than 1 (as required by the draft ANSI standard). Of course, `sizeof(x) == 1`, as you would expect.

A struct, union or enum tag declared within a function prototype is wrongly given file scope rather than prototype scope (if the prototype is a function declaration) or function scope (if the prototype is part of a function definition).

The following should work, but doesn't:

```
typedef void VOID;
VOID fn() {...}
```

Faults which may cause invalid programs to be accepted

An empty initialiser list within braces is not faulted (eg `int x[10] = {};`). An empty structure declaration (eg `struct foo {};`) is erroneously treated as the corresponding struct tag declaration (eg `struct foo;`). However, any attempt to use the empty declaration in a manner not compatible with use of the corresponding struct tag declaration, is faulted.

The compiler erroneously accepts 'register' as an orthogonal attribute of a variable. Thus the following are wrongly accepted without demur:

```
register auto x;
register static y;   /* not at top level */
```

Adding insult to injury, `&y` then elicits a warning about taking the address of a variable with the register attribute, even though it is treated as static!

An enum constant larger than `INT_MAX` is not faulted (it wraps around silently).

A file containing no external definitions is not faulted (as required by ANSI).

In `-pcc` mode, an initialised auto array elicits two spurious warnings rather than being faulted.

Miscellaneous compiler faults

The following faults cause neither rejection of valid input programs nor silent acceptance of invalid ones. Rather, they relate to unfriendly or unexpected behaviour by the compiler.

Invalid local redeclarations are faulted, but with a misleading message about 'redefinition' (rather than 'redeclaration'). For example:

```
int f() {int f1(int); int f1(int, int); ... }
```

Some seriously damaged sources can crash the compiler. The best (smallest) example available is:

```
int C(int c) {
    switch (c) {
     case 1:  do {
        /*  } while (1); */
     case 2:;
    }
  return 0;
}
```

Recovery from illegal and undefined casts is sometimes wrong and may crash the compiler. For example:

```
int f() {union {int i; char c;} u = 1; }
```

gives a fatal internal error. However, if '1' is enclosed in braces, as it should be, all is well.

The (illegal) unary negation of a double causes a fatal internal error. For example:

```
void f() {double d; int ff(); ff(~d);}
```

Code compiled by the C compiler does not strictly obey the ARM Procedure Call Standard, which demands that all one-word aggregate results from functions be returned in R0 (all aggregates are returned via an additional, first address argument to the function).

In `-cc` mode, cc generates bad code for extracting subfields from non-word-aligned bitfields. For example:

```
struct {char x; int y:4;} z = {0,1};
                ...
                int zy = z.y;          /* bad code */
```

This cannot happen in ANSI mode, or if a string of bitfields is word-aligned. For example:

```
struct {int x:8; int y:4;} z = {0,1};
        int zy = z.y;                  /* code is OK */
```

Known faults in the C library

Using the C library, it is not possible to perform an `ungetc()` immediately after a call to `scanf()` which performed an `ungetc()`. `CLK_TCK` remains defined as a synonym for `CLOCKS_PER_SEC`, in violation of the latest draft ANSI standard.

Known infelicities in Release 3 of cc

Expressions involving signed bytes and both flavours of short integer are not well optimised (chars are unsigned in ANSI mode; signed in `-pcc` mode).

Sequences of statements accessing bitfields belonging to the same word of a structure are not well optimised.

Use of '=' in conditional contexts is not uniformly well checked. Simple cases are caught and a warning is issued. Complicated cases go unnoticed, but this may be what you want.

The diagnostic generated by `typedef int INTFN(int); INTFN f {return 0;}` gives `ancient form of initialisation`, which is less than helpful.

The diagnostic generated by `int f(int i) int i; {return i;}` generates `duplicate type specification of formal parameter i`, which is less than helpful. `cc` should say something like `no declarator list may follow a prototype declarator`.

Some preprocessor messages refer to `#ifdef` when the fault occurs in a `#ifndef`; similarly with `#if/#elif`.

Potential side effects (overflow) of FP operations are ignored in dead-code elimination. Thus code like the following, in which the result of the computation is not used, will not fault at run-time. For example:

```
int f() {double x = 1e38; x *= x; x *= x; x *= x; x *= x; x *= x; x *= x; }
```

© Copyright Acorn Computers Limited
Published by Acorn Computers Limited
Fulbourn Road, Cambridge CB 1 4JN
August 1989
Part number 0470,109 Issue 1