ACK/CEM Compiler
Reference Manual

Erik H. Baalbergen

Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

11 February 2005

ACK/CEM Compiler
Reference Manual

Erik H. Baalbergen

Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

1. C Language

This section discusses the extensions to and deviations from the C language, as described in [1]. The issues are numbered according to the reference manual.

2.2 Identifiers

Upper and lower case letters are different. The number of significant letters is 32 by default, but may be set to another value using the **−M** option. The identifier length should be set according to the rest of the compilation programs.

2.3 Keywords

`asm`

The keyword `asm` is recognized. However, the statement

```
asm(string);
```

is skipped, while a warning is given.

`enum`

The `enum` keyword is recognized and interpreted.

`entry`, `fortran`

The words `entry` and `fortran` are reserved under the restricted option. The words are not interpreted by the compiler.

2.4.1 Integer Constants

The type of an integer constant is the first of the corresponding list in which its value can be represented. Decimal: `int, long, unsigned long`; octal or hexadecimal: `int, unsigned, long, unsigned long`; suffixed by the letter L or l: `long, unsigned long`.

2.4.3 Character Constants

A character constant is a sequence of 1 up to `sizeof(int)` characters enclosed in single quotes. The value of a character constant '$c_1 c_2 \cdots c_n$' is $d_n + M \times d_{n-1} + \cdots + M^{n-1} \times d_2 + M^n \times d_1$, where M is 1 + maximum unsigned number representable in an `unsigned char`, and d_i is the signed value (ASCII) of character c_i.

2.4.4 Floating Constants

The compiler does not support compile-time floating point arithmetic.

2.6 Hardware characteristics

The compiler is capable of producing EM code for machines with the following properties

- a `char` is 8 bits
- the size of `int` is equal to the word size
- the size of `short` may not exceed the size of `int`
- the size of `int` may not exceed the size of `long`
- the size of pointers is equal to the size of either `short`, `int` or `long`

4 What's in a name?

`char`

Objects of type `char` are taken to be signed. The combination `unsigned char` is legal.

`unsigned`

The type combinations `unsigned char`, `unsigned short` and `unsigned long` are supported.

`enum`

The data type `enum` is implemented as described in *Recent Changes to C* (see appendix A). *Cem* treats enumeration variables as if they were `int`.

`void`

Type `void` is implemented. The type specifies an empty set of values, which takes no storage space.

Fundamental types

The names of the fundamental types can be redefined by the user, using `typedef`.

7 Expressions

The order of evaluation of expressions depends on the complexity of the subexpressions. In case of commutative operations, the most complex subexpression is evaluated first. Parameter lists are evaluated from right to left.

7.2 Unary operators

The type of a `sizeof` expression is `unsigned int`.

7.13 Conditional operator

Both the second and the third expression in a conditional expression may include assignment operators. They may be structs or unions.

7.14 Assignment operators

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of operands taking part must be the same.

8.2 Type specifiers

The combinations `unsigned char`, `unsigned short` and `unsigned long` are implemented.

8.5 Structure and union declarations

Fields of any integral type, either signed or unsigned, are supported, as long as the type fits in a word on the target machine.

Fields are left adjusted by default; the first field is put into the left part of a word, the next one on the right side of the first one, etc. The `-Vr` option in the call of the compiler causes fields to be right adjusted within a machine word.

The tags of structs and unions occupy a different name space from that of variables and that of member names.

9.7 Switch statement

The type of *expression* in

`switch (`*expression*`)` *statement*

must be integral. A warning is given under the restricted option if the type is `long`.

10 External definitions

See [4] for a discussion on this complicated issue.

10.1 External function definitions

Structures may be passed as arguments to functions, and returned by functions.

11.1 Lexical scope

Typedef names may be redeclared like any other variable name; the ice mentioned in §11.1 is walked correctly.

12 Compiler control lines

Lines which do not occur within comment, and with `#` as first character, are interpreted as compiler control line. There may be an arbitrary number of spaces, tabs and comments (collectively referred as *white space*) following the `#`. Comments may contain newline characters. Control lines with only white space between the `#` and the line separator are skipped.

The `#include`, `#ifdef`, `#ifndef`, `#undef`, `#else` and `#endif` control lines and line directives consist of a fixed number of arguments. The list of arguments may be followed an arbitrary sequence of characters, in which comment is interpreted as such. (I.e., the text between `/*` and `*/` is skipped, regardless of newlines; note that commented-out lines beginning with `#` are not considered to be control lines.)

12.1 Token replacement

The replacement text of macros is taken to be a string of characters, in which an identifier may stand for a formal parameter, and in which comment is interpreted as such. Comments and newline characters, preceeded by a backslash, in the replacement text are replaced by a space character.

The actual parameters of a macro are considered tokens and are balanced with regard to `()`, `{}` and `[]`. This prevents the use of macros like

```
CTL([)
```

Formal parameters of a macro must have unique names within the formal-parameter list of that macro.

A message is given at the definition of a macro if the macro has already been `#defined`, while the number of formal parameters differ or the replacement texts are not equal (apart from leading and trailing white space).

Recursive use of macros is detected by the compiler.

Standard #`defined` macros are

`__FILE__`	name of current input file as string constant
`__DATE__`	curent date as string constant; e.g. `"Tue Wed  2 14:45:23 1986"`
`__LINE__`	current line number as an integer

No message is given if *identifier* is not known in

`#undef` *identifier*

12.2 File inclusion

A newline character is appended to each file which is included.

12.3 Conditional compilation

The `#if`, `#ifdef` and `#ifndef` control lines may be followed by an arbitrary number of

`#elif` *constant-expression*

control lines, before the corresponding `#else` or `#endif` is encountered. The construct

```
#elif constant-expression
some text
#endif /* corresponding to #elif */
```

is equivalent to

```
#else
#if constant-expression
some text
#endif /* corresponding to #if */
#endif /* corresponding to #else */
```

The *constant-expression* in `#if` and `#elif` control lines may contain the construction

`defined(`*identifier*`)`

which is replaced by `1`, if *identifier* has been #`defined`, and by `0`, if not.

Comments in skipped lines are interpreted as such.

12.4 Line control

Line directives may occur in the following forms:

```
#line constant
#line constant "filename"
#constant
#constant "filename"
```

Note that *filename* is enclosed in double quotes.

14.2 Functions

If a pointer to a function is called, the function the pointer points to is called instead.

15 Constant expressions

The compiler distinguishes the following types of integral constant expressions

- field-width specifier

- case-entry specifier
- array-size specifier
- global variable initialization value
- enum-value specifier
- truth value in `#if` control line

Constant integral expressions are compile-time evaluated while an effort is made to report overflow. Constant floating expressions are not compile-time evaluated.

2. Compiler flags

−C Run the preprocessor stand-alone while maintaining the comments. Line directives are produced whenever needed.

−D*name=string-of-characters*
Define *name* as macro with *string-of-characters* as replacement text.

−D*name*
Equal to **−D***name*=**1**.

−E Run the preprocessor stand alone, i.e., list the sequence of input tokens and delete any comments. Line directives are produced whenever needed.

−I*path*
Prepend *path* to the list of include directories. To put the directories "include", "sys/h" and "util/h" into the include directory list in that order, the user has to specify

```
-Iinclude -Isys/h -Iutil/h
```

An empty *path* causes the standard include directory (usually `/usr/include`) to be forgotten.

−M*n*
Set maximum significant identifier length to *n*.

−n Suppress EM register messages. The user-declared variables are not stored into registers on the target machine.

−p Generate the EM **fil** and **lin** instructions in order to enable an interpreter to keep track of the current location in the source code.

−P Equivalent with **−E**, but without line directives.

−R Interpret the input as restricted C (according to the language as described in [1]).

−T*path*
Create temporary files, if necessary, in directory *path*.

−U*name*
Get rid of the compiler-predefined macro *name*, i.e., consider

`#undef` *name*

to appear in the beginning of the file.

−V*cm.n*, **−V***cm.ncm.n ...*
Set the size and alignment requirements. The letter *c* indicates the simple type, which is one of **s**(short), **i**(int), **l**(long), **f**(float), **d**(double) or **p**(pointer). If *c* is **S** or **U**, then *n* is taken to be the initial alignment of structs or unions, respectively. The effective alignment of a struct or union is the least common multiple of the initial struct/union alignment and the alignments of its members. The *m* parameter can be used to specify the length of the type (in bytes) and the *n* parameter for the alignment of that type. Absence of *m* or *n* causes the default value to be retained. To specify that the bitfields should be right adjusted instead of the default left adjustment, specify **r** as *c* parameter.

−w Suppress warning messages

−−*character*

Set debug-flag *character*. This enables some special features offered by a debug and develop version of the compiler. Some particular flags may be recognized, others may have surprising effects.

d Generate a dependency graph, reflecting the calling structure of functions. Lines of the form

`DFA:` *calling-function*`:` *called-function*

are generated whenever a function call is encountered.

f Dump whole identifier table, including macros and reserved words.

h Supply hash-table statistics.

i Print names of included files.

m Supply statistics concerning the memory allocation.

t Dump table of identifiers.

u Generate extra statistics concerning the predefined types and identifiers. Works in combination with **f** or **t**.

x Print expression trees in human-readable format.

References

[1] Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*

[2] L. Rosler, *Draft Proposed Standard - Programming Language C,* ANSI X3J11 Language Subcommittee

[3] Erik H. Baalbergen, Dick Grune, Maarten Waage, *The CEM Compiler,* Informatica Manual IM-4, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

[4] Erik H. Baalbergen, *Modeling global declarations in C,* internal paper

Appendix A - Enumeration Type

The syntax is

enum-specifier:
`enum` { *enum-list* }
`enum` *identifier* { *enum-list* }
`enum` *identifier*

enum-list :
enumerator
enum-list , *enumerator*

enumerator :
identifier
identifier = *constant-expression*

The identifier has the same role as the structure tag in a struct specification. It names a particular enumeration type.

The identifiers in the enum-list are declared as constants, and may appear whenever constants are required. If no enumerators with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

Enumeration tags and constants must all be distinct, and, unlike structure tags and members, are drawn from the same set as ordinary identifiers.

Objects of a given enumeration type are regarded as having a type distinct from objects of all other types.

Appendix B: C grammar in LL(1) form

The **bold-faced** and *italicized* tokens represent terminal symbols.

external definitions

program: external-definition*

external-definition: ext-decl-specifiers [declarator [function | non-function] | '**;**'] | asm-statement

ext-decl-specifiers: decl-specifiers?

non-function: initializer? ['**,**' init-declarator]* '**;**'

function: declaration* compound-statement

declarations

declaration: decl-specifiers init-declarator-list? '**;**'

decl-specifiers: other-specifier+ [single-type-specifier other-specifier*]? | single-type-specifier other-specifier*

other-specifier: **auto** | **static** | **extern** | **typedef** | **register** | **short** | **long** | **unsigned**

type-specifier: decl-specifiers

single-type-specifier: *type-identifier* | struct-or-union-specifier | enum-specifier

init-declarator-list: init-declarator ['**,**' init-declarator]*

init-declarator: declarator initializer?

declarator: primary-declarator ['**(**' formal-list ? '**)**' | arrayer]* | '*****' declarator

primary-declarator: identifier | '**(**' declarator '**)**'

arrayer: '**[**' constant-expression? '**]**'

formal-list: formal ['**,**' formal]*

formal: identifier

enum-specifier: **enum** [enumerator-pack | identifier enumerator-pack?]

enumerator-pack: '**{**' enumerator ['**,**' enumerator]* '**,**'? '**}**'

enumerator: identifier ['**=**' constant-expression]?

struct-or-union-specifier: [**struct** | **union**] [struct-declaration-pack | identifier struct-declaration-pack?]

struct-declaration-pack: '**{**' struct-declaration+ '**}**'

struct-declaration: type-specifier struct-declarator-list '**;**'?

struct-declarator-list: struct-declarator ['**,**' struct-declarator]*

struct-declarator: declarator bit-expression? | bit-expression

bit-expression: '**:**' constant-expression

initializer: '**=**'? initial-value

cast: '**(**' type-specifier abstract-declarator '**)**'

abstract-declarator: primary-abstract-declarator ['**(**' '**)**' | arrayer]* | '*****' abstract-declarator

primary-abstract-declarator: ['**(**' abstract-declarator '**)**']?

statements

statement:

 expression-statement

| label '**:**' statement

| compound-statement

| if-statement
| while-statement
| do-statement
| for-statement
| switch-statement
| case-statement
| default-statement
| break-statement
| continue-statement
| return-statement
| jump
| '**;**'
| asm-statement
;

expression-statement: expression '**;**'
label: identifier
if-statement: **if** '(' expression ')' statement [**else** statement]?
while-statement: **while** '(' expression ')' statement
do-statement: **do** statement **while** '(' expression ')' '**;**'
for-statement: **for** '(' expression? '**;**' expression? '**;**' expression? ')' statement
switch-statement: **switch** '(' expression ')' statement
case-statement: **case** constant-expression '**:**' statement
default-statement: **default** '**:**' statement
break-statement: **break** '**;**'
continue-statement: **continue** '**;**'
return-statement: **return** expression? '**;**'
jump: **goto** identifier '**;**'
compound-statement: '**{**' declaration* statement* '**}**'
asm-statement: **asm** '(' *string* ')' '**;**'

expressions
initial-value: assignment-expression | initial-value-pack
initial-value-pack: '**{**' initial-value-list '**}**'
initial-value-list: initial-value ['**,**' initial-value]* '**,**'?
primary: *identifier* | constant | *string* | '(' expression ')'
secundary: primary [index-pack | parameter-pack | selection]*
index-pack: '**[**' expression '**]**'
parameter-pack: '(' parameter-list? ')'
selection: ['**.**' | '**->**'] identifier
parameter-list: assignment-expression ['**,**' assignment-expression]*
postfixed: secundary postop?

unary: cast unary | postfixed | unop unary | size-of

size-of: **sizeof** [cast | unary]

binary-expression: unary [binop binary-expression]*

conditional-expression: binary-expression ['**?**' expression '**:**' assignment-expression]?

assignment-expression: conditional-expression [asgnop assignment-expression]?

expression: assignment-expression ['**,**' assignment-expression]*

unop: '*' | '**&**' | '−' | '**!**' | '˜' | '++' | '−−'

postop: '++' | '−−'

multop: '*' | '/' | '**%**'

addop: '+' | '−'

shiftop: '<<' | '>>'

relop: '<' | '>' | '<=' | '>='

eqop: '==' | '**!=**'

arithop: multop | addop | shiftop | '**&**' | '^' | '|'

binop: arithop | relop | eqop | '**&&**' | '||'

asgnop: '=' | '+' '=' | '−' '=' | '*' '=' | '/' '=' | '**%**' '='

| '<<' '=' | '>>' '=' | '**&**' '=' | '^' '=' | '|' '='

| '+=' | '−=' | '*=' | '/=' | '**%**='

| '<<=' | '>>=' | '**&**=' | '^=' | '|='

constant: *integer* | *floating*

constant-expression: assignment-expression

identifier: *identifier* | *type-identifier*