WG14 NØ33

Ø5 Dec 87

Remarks to X3J11
From ISO WG14

ISO WG 14 on the C language has several remarks about the draft C standard from X3J11. The first section lists the topics the working group feels must be addressed before the draft can become an ISO standard. The second section lists topics WG14 feels X3J11 should look into to clarify important issues.

Parenthesis grouping and unary +

We feel strongly that the C standard in section 3.3 should honor grouping in expressions. One member of WG14 (AFNOR) has stated that they will vote against an ISO standard that does not honor grouping.

This allows the special semantics for unary plus connected with grouping to be removed from the language. We are not in favor of removing the unary plus as a language construct.

We fully support the position taken by P.J. Plauger in his paper (X3J11/87-197).

Multibyte characters

WG14 supports the actions X3J11 have taken to date in support of multi-byte characters. One member of WG14 (AFNOR) has stated that they will vote against an ISO standard that does not complete the job of supporting multi-byte characters.

WG14 endorses an attempt to find a notation for wchar string literals and character constants.

WG14 has no interest in providing additional capability.

WG14 sympathizes with the Japanese with regard to their multi-byte character support requests. However, the committee is hesitant to support more ambitious goals than those stated above without the benefit of established prior art.

The term for international time

The description in section 4.12.3.3 uses the name "Greenwich Mean Time (GMT)" for international time specification. This name is not correct it should be "Universal Time Coordinated (UT)". WG14 strongly endorses this change.

NO23 Reconsideration of trigraphs

The ISO/TC97/SC22 second plenary meeting in Washington Ø8-11 Sep 87 adopted unanimously in its resolution WP-34 that:

> "ISO/IEC JTC1/SC22 requests its member bodies to communicate to the SC22 secretariat their requirements for character handling in programming languages, including multi-octet character sets."

Thus WG14 has considered the support for non-ASCII ISO 646 based character sets in ISO C. ISO 646 leaves 12 positions open to national standardisation bodies to define, and it should be possible to avoid using these positions in C-programs, as they are not well defined. A solution to this, named the trigraph solution, is included in the current ISO draft, but this seems inappropiate as the resulting programs are not very pleasant to read nor write. Examples of this are demonstrated in the paper ISO/TC97/SC22/WG14/NØ23, where a possible solution also is discussed.

The paper as amended by WG14 concludes in the following changes to the draft proposal dated 1987-Ø5-15:

Alternate bracketing tokens for {, }, [, and ], namely (/, /), !(, and ) respectively. Corrections:

page 27 line 13, add: !(
page 27 line 21, add: !( ),
page 27 line 22, add: The operator pair !( and ) shall be used synonymously for [ and ] respectively.
page 27 line 33, 35 add: (/ /) !( )
page 27 line 37, add: The punctuators (/ /) !( ) shall be used synonymously for { } [ and ] respectively.

! used alone is proposed to work as an infix operator for array subscripting.

page 33 line 28 add: postfix-expression ! expression
page 33 line 46 add after []: or after !

It is also necessary to introduce new definitions of logical operators, as they also use undefined ISO 646 characters. The proposal is to include the keywords: or cor and cand xor compl.

page 13 section 3.1.1 add: or cor and cand xor compl
page 27 line 18 add: or cor and cand xor compl
page 27 line 22, add: The operators or cor and cand xor compl shall be used synonymiously for | || & && ^ ~ respectively.

An alternate way to introduce these definitions is to include them in a header file with the following definitions:

```
#define or |
#define cor ||
#define and &
```

```
#define cand &&
#define xor ^
#define compl ~
```

These defines could be included in a header file, for instance the existing <locale.h> or a new <booldefs.h>.

The last non-defined ISO 646 character used in the ISO C draft is \ used in strings. We propose to alternatively use &. This should be included in section 3.1.3.4 with the addendum:

Alternatively the character & could be used for escape sequences.

A non-ISO 646 character is likely to be introduced in format strings for printf etc, namely $ to specify argument number. We propose to alternatively use !.

WG14 strongly (but not unanimously) supports the alternate notation for braces and brackets, and the proposed keywords (or macros). There was no support for using & as an alternate escape character within string literals.

Finland does not support any part of this proposal, except possibly some form of infix operator for subscripting.

NO27 Preprocessor tokenisation Phases of Translation: 2.1.1.2

Nowhere is the conversion of preprocessing tokens to so-called "(normal) tokens" described. Surely the result of preprocessing, i.e. the output of the preprocessor is a stream of tokens which require no further processing. If this is not the case, then the standard should clarify what exactly takes place when "preprocessing tokens are converted into (normal) tokens." Tokenisation

The standard implies in 3.1 that the following example is correct:

```
#define a +
...a+b...
yields
...plus_token plus_token identifier...
and not
...increment_token identifier...
```

because 'a' only gets expanded after "parsing" ie tokenisation. The text is decomposed into: identifier plus_token identifier; then expanded to plus_token plus_token identifier.

If preprocessing is implemented as a separate text-to-text prepass (see 3.8.3 claim in Rationale), then this text becomes

```
...++b...
```

which is indistinquishable from the token sequence: increment_token identifier.

It should be clearly stated that the preprocessor is token based.

Semantics of Conditional Inclusion: 3.8.1

The order of replacement of identifiers defined as macro names is significant; and as this is undefined, it will be a source of ambiguity as the following example illustrates:

```
#define a defined
#define b c
#if a b
```

If 'a' is replaced first, then the identifier 'b' is "modified by defined" and hence an exception i.e. not replaced. If 'a' is not replaced before 'b', then the identifier 'b' is a candidate for replacement.

Defining the order of replacement, e.g. left to right, will solve this problem.

Evaluation of ## operations in a replacement list: 3.8.3.3

No clear semantics is given for the order of evaluation of the ## operator if it appears several times. Left to right would be a sensible rule; and be easy to understand and implement.

The current draft states that the result of a paste operation must be a valid token (3.8.3.3); the reason for this restriction is unclear.

Consider:

```
#define p3(a,b,c) a##b##c
...p3(text,9999,text)...
```

If the paste of b to c takes place before the paste of a to b, then the result of the paste '9999text' is not a valid token and more significantly the behaviour is undefined. If a is pasted to b first, the behaviour is defined.

Defining an order of evaluation allowing paste operations to be ordered so that their behaviour is defined will solve this. Alternatively, dropping the requirement would provide a better solution. After all the paste operations have been carried out in the replacement list, it must be a sequence of valid tokens; and that would be a more reasonable constraint.

Lines: 4.9.2

This section implies very strongly that vertical tab and form feed are space analogues, not newline analogues (i.e. a form feed character does not of itself terminate a line). By association, we take this to apply to carriage return as well, though the latter is not mentioned in 4.9.2. This has extremely serious consequences.

Firstly, both ANSI Fortran 77 and ANSI C have concepts of text files with pagination and overprinting, but it is impossible to define a 1-1 mapping between these. The constraint in 4.9.2 that certain text files are unchanged on reinput to C further confuses the issue. A letter mentioning this problem has been sent to X3J3 indicating this is a common problem.

It is proposed that the following simple solution be applied: define a line as terminated by any of newline, form feed, carriage return or vertical tab. This solution is that adopted by C's recognised ancestor, BCPL.

Change to section 4.9.2, second paragragh:

> A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating newline, form feed, carriage return or vertical tab.

There will be consequential changes throughout the document in the section on Input/Output (4.9).

Alternatively, drop the requirement that text files are unchanged on reinput if they contain form feed or vertical tab characters. Thus, the condition for rereading a text file unchanged is that it contains only printing characters, horizontal tab and newline.

Truncated Files: 4.9.3

It does not seem to be explicitly mentioned anywhere when, if ever, files are truncated (except, of course, when they are opened). The normal form of sequential access under IBM MVS and CMS provides all of the file access facilities of ANSI C (including binary update and fsetpos), with the interpretation that any write operation causes the file to be immmediately truncated beyond that point.

As the draft stands, this would be conforming implementation, though it would cause most programs imported from UNIX to fail horribly. While it is desirable that this 'magnetic tape' form of update be available, it is clearly essential that 'UNIX style' direct-access be available. This confusion is heightened by the absence of any ftruncate function to truncate a file explicitly, and that fact that ANSI Fortran 77 uses the 'magnetic tape' style for update.

Some clarification on this point is essential, even if merely to say that the state is implementation defined.

Add to section 4.9.3:

> Binary files are not truncated, except as defined in section 4.9.5.3. Thus every character written to a file remains in that file until it is overwritten, the file is reopened for write mode (letter w), or the file is removed. It is implementation-defined whether writing a character to a text file causes the file to be truncated immediately beyond that point, or whether text files are handled in the same way as binary files.

This is a possible solution; there are other solutions. The UK position is that the standard requires a solution to be specified.

Equality operators, relation operators and (void *)

Both the contraints for equality operators(3.3.9) and for relational operators(3.3.8) are to tight. For example: the current draft does not allow to compare two (void *) pointers to be compared for equality.