A C Cross-Compiler for the Motorola 680x0 Family

Tamas Gaa1* and David G. Foster

* Scientific associate from the Central Research Institute for Physics, Budapest

(To be presented at the IFIP/IFAC Working Conference on Hardware and Software for Real-Time Process Control, Warsaw, 30 May-1 June 1988)

DD-vf
A C Cross-Compiler for the Motorola 680x0 Family

Tamas Gaal, David G. Foster
CERN DD Division
1211 Geneva 23 Switzerland

Abstract

A C front-end compiler has been written as a member of the CERN set of cross-compilers for the Motorola 680x0 microprocessor family [3,4,6], keeping in mind the coming X3J11 ANSI C standard [9]. After a review of the peculiarities of C, the implementation principles and difficulties will be discussed. Overall advantages, disadvantages, and some performance characteristics of the new C cross-compiler are also mentioned.

Key phrase: "C programmers use all the possibilities of C"

1. Short History, Aims

The Motorola 680x0 microprocessor family comprises several 32-bit processors [1,2]. The lack of commercially available software support for embedded systems necessitated the development of cross software tools [3]. Among others, a cross assembler and compiler suite was planned and implemented. The result of the processing of the compiler front-ends (Pascal, Modula2, C, FORTRAN77) is an abstract-tree-format code which is further processed by an optimizing code generator, which, in turn, creates another abstract-format object code called CUFOM [3]. This code is linkable and loadable. A slightly modified CUFOM has become an industrial standard in the USA (MUFOM) [10]. The present working hosts are VAX/ULTRIX and VAX/VMS [and see 3], although the software has been ported to many different machines. The targets are M680x0 processors. The programs are loaded over terminal lines or network interfaces.

2. C-Peculiarities - a short reminder

The three most relevant C sources are: Kernighan, Ritchie: The C Programming Language, Harbison, Steele: C a Reference Manual, and the X3J11 Draft C Standard ([7], [8] and [9]). Unfortunately, they differ in the language they define. Our basis has been [8]. We wish to avoid comparing C with other programming languages, instead emphasis will be given to some of the more interesting features. Details concerning the pre-processor and the C program library are not included in this talk.

2.1. C Features

C has strong typing but it can be circumvented by the programmer. C has separate compilation but initialised external variables may considerably change the semantics of a program. C programs can be separated into a declaration (.h) and definition (.c) part. These may depend on other declaration parts which are merged by a simple macro pre-processor.

C has an entity - the declarator - by which arbitrarily complicated data structures can be devised and such structures can be effectively used in expressions: there is a common syntactical reference. This feature renders C suitable for object-oriented style programming. C code can be very effective especially in system programming; in number crunching applications the automatic promotion of float-type variables into double-type variables slows down the run-time processing: although it could be optimised during the code generation phase.

C does not have a full constructor facility: type name aliases and structure, union, and enumeration types can be created, they can be coupled by declarations, but the operations on such objects are predefined by the language. The operators of C are rich, well classified by precedence which encourages concise programming. C can handle bit level operations and bit types can also be created, however, although bit manipulation expressions are an integral part of the language, the bit-field facility is not well defined and quite obscure. C structures can be assigned in an assignment expression.

C functions can be called with an arbitrary number of parameters and there is no check on the types of defined (formal) and used (actual) parameters. The returned result can be of almost any type with the exception of arrays and functions. Functions can be recursive but nesting of functions is not allowed. One can, however, define
unnamed blocks with inner declarations inside to an arbitrary level of nesting. The nesting of declarations either in declarators or in composite types (structures) is also allowed: in the latter case one can choose named or unnamed inner structures and so on. C I/O is not defined by the language and is done by quasi-standard library routines.

The above mentioned features together with other uncommon, and obscure, ones often means considerable difficulty for the compiler writer: the underlying semantics can be interpreted slightly differently which implies that previously good programs will not run when ported between compilers. It is advisable not to be too restrictive when compiling C programs.

Everything is allowed: the programmer is always right.

2.2. C Problems

The biggest problem is the lack of a defining grammar for the C syntax. Worse, the existing published versions are error-prone. The C compiler writers are forced to define their own version of the C grammar but it is inherently unreliable as these grammars are aimed usually at a specific parsing method and are not suitable or require effort when used with another method. C is LALR(1) which is normally associated with a bottom-up style of parsing.

The syntax of C assumes to a large extent that bottom up parsing methods will be used and grammatical representations of the language tend to re-enforce this view. The fundamental construct in C is the expression and there is no enforced boundary between syntactic and semantic constructs. Syntactic constructs can be used at several funny places e.g. one can place a declaration into an expression in the case of a well-chosen size of expression (syntactically correct) but the underlying semantic checks should restrict the consequences of such constructs.

Fortunately, there are a number of programming paradigms which were kept in mind when designing C (l-value: left-hand-value, conversion rules, address space, pointer handling) that can be described by more or less precise rules so the often heard "spirit of C" really exists, nevertheless C remains inherently imprecise (e.g. undefined overflow effects).

3. The CERN C front-end cross-compiler

3.1. Basic Implementation Principles

The whole set of the CERN compilers was designed and implemented by observing the needs of Portability, Ease of Maintenance, Correspondence to the Language Standard. We do not claim these to be overall governing principles in our environment but with different host machines, complicated network architectures, etc they are compulsory requirements.

The C front-end is a one-pass compiler as C makes this possible. There is a symbolic debugging facility for the running code.

3.2. Implementation

C compilers are often offsprings of the Portable C Compiler (Bell Labs) [11] and the parsing is usually done by using a parser generator ("compiler-compiler") after having transformed the grammar into a suitable LALR(k) form.

This is not the case with our compiler. The reasons why not are mostly practical ones: the other CERN compilers were written in Pascal using top-down parsing and this approach has a non-negligible advantage: these compilers are readable and therefore maintainable. Another advantage in this approach is that the decisions of former developers do not force compromises about the accepted language and one can re-think obscure points in the language: in our case, the adherence to the forthcoming C standard has required this kind of re-thinking. This from-the-scratch approach has its pitfalls and dangers and obviously requires more work - we think it has been worth it.

The compiler is a one-pass, top-down, recursive descent parser generating a tree-format intermediate code for the optimizing code generator. Optimization was not implemented in the front end. Error recovery is done by a synchronising symbol deletion/insertion mechanism. The symbol table, identifier, and keyword table handling are based on the known methods of binary tree, hashing, list handling.

At several places look-aheads are necessary to decide about ambiguous constructs, and to be able to give good error messages: incoming symbols are arranged in a stack of proper length. This is a consequence of the grammar chosen: with transformed grammars look-aheads may be eliminated, reduced, or shifted to less sensitive places. By using look-aheads we want to avoid backtracking which is considered harmful. Our choice of insisting
on the recursive descent method even in the particular case of expression analysis is arguable: we admit that a table-driven sub-part expression analyser may have been a more suitable, faster way of processing expressions - at the price of giving up a part of readability.

The analysis of a C program and the creation of a semantically equivalent intermediate tree is a syntax-driven process but as the syntactic analysis progresses the more semantic attributes are deduced, resulting in a mixed syntax/semantics based compilation. The sequence of processing follows loosely the top-down tree walking; there are local divergences when e.g. the final determination of a declarator name embedded into a complicated declarator is delayed to the latest possible point in order to gather most possible information.

In C, a type, a declarator and instances of the resulting entity (possibly initialised) can be written in a single declaring phrase. This concise notation requires the compiler-writer to know about more than one thing at a given point of the analysis: in the implementing language adequate data abstraction features are needed to correctly meet this demand.

In certain statements the statement syntax is not precisely defined grammatically, resulting in context sensitiveness (case label, break, etc.). This sounds horrifying but these cases can be handled quite simply by using semantic information.

The target constant evaluation is done in the host according to the host arithmetic rules so the target code inherits the host properties in this respect; it may cause problems in extreme floating-point examples. The fact that it is a cross-compiler meant unexpected difficulties: in C, one can have a specific construct, a so-called constant expression which can be present at several places and it should evaluate to a constant value; the compiler must recognize these situations and pass a single value to the next step. The problem is that in certain cases either a constant or an arbitrary expression can be present and the front end should decide whether to evaluate the expression or pass it to the code generator.

Designing recursive routines, inside the front end, requires care: arbitrarily deep nesting levels can be achieved when analysing complex expressions.

C adds to this recursive complexity: declarations can also be recursive so recursiveness can extend to "three dimensions" at a time: On a lower block level (dimension one) one can have a complicated declarator (dimension two) coupled with a structure definition that contains further inner nested structures (dimension three). If, for joy, such a variable object is supplied with a handy initialiser we have got a new recursive dimension.

Nested blocks and nested structures are similar in several respect: they can be conceptually modelled similarly (irrespective of their corresponding semantics) which proved useful when determining respective name spaces, scope, and visibility rules. It is not trivial and is not well defined in the case of structures (and unions).

Another problem area is the proper handling of the referencing/defining declaration (occurrence) of a given object. In C one can give several declarations of the same variable provided the declarations are all referencing ones. If a defining declaration occurred, the following declarations of the same identifier must all be referencing ones. This sounds simple but there is another attribute, the so-called storage class, which can exert influence on the previous rule: inter-compilation-unit references are not easy to separate from references confined to the name space of the processed compilation unit and it is really delicate when the variable space allocation, the creation of export and import attributes from the single "extern" storage class determiner (which can be left out: in certain places it is equivalent to an "assumed extern" which is not exactly equal to an "explicit extern" .) should be decided: we are not allowed to invent a new, tailor-suited scheme because storage allocation and function definition are pre-defined (by the existing code generator) activities. Consequently the degree of freedom is really narrow.

Existing compilers handle generously the question of the storage class in the context of the referencing/defining declaration but we were bound by the diverging storage allocation strategies.

The "initialiser" is again an interesting phenomenon: on one hand, it can make a referencing declaration into a defining one so adding to the referencing/defining problem; on the other hand, it can determine a partially determined object (e.g. an instance of an unbounded array type is given boundaries by the number of the initialisers of its elements - so storage allocation is postponed to this point). Its syntax is not an exact match of that of the variable declaration to be initialised - it is similar but there are exceptions (e.g. an array initialised with a char string) and alleviations (the compiler must keep in mind that e.g. a multi-dimensional array is initialised and not the programmer) so it needs a separate analysis. Besides, it is not independent from the code-generator because the two kind of initialisations (static/automatic) require different space allocation, assignment, linking and run-time activities.
C cross-compiler for the Motorola 680x0

Other areas are the 64 bit (double) constants and the conversions. C can automatically or explicitly convert variables: the principle is Widening to Make Uniform If Not Otherwise Stated. Some conversions are done by using software calls. Here again bit-fields can make it complicated.

The pre- and post-increment/decrement expressions may seem to be a well-established part of C however they are not trivial and we have valid C tests that killed existing compilers: in some cases the optimizer caused the malfunction (e.g. a++ + a++) in some other cases it was simply bad - but valid - C style (e.g. doublevar++, etc).

The programmer may overwrite code parts and data too by improper programming. This is usually true with other C compilers.

A specific difficulty was to achieve a minimal number of changes in the front-end - back-end interface. The back-end is an optimizing code generator which has been in use for several years so - sensibly - real changes were flatly denied only extensions were reluctantly allowed.

After having finished the first experimental version of the front-end substantial parts had to be changed to meet specific implicit needs of the code generator.

The development environment has been an APOLLO workstation and a VAX 8530 under ULTRIX. The interchangeable and inter-usable file system (through a TCP-IP network) and the good debugging facilities of the work-station has resulted in a convenient development configuration.

3.3. Evaluation

C seems to be a small language but a C compiler is not trivial because the compiler should serve the programmer’s whims and should confuse him/her only in case of absolute necessity. A new compiler should not diverge from the existing ones. The front-end described is the core of our C cross-compilation but other items (pre-processor, library, code generator, linker, pusher, debugger, monitor) make it usable.

3.3.1. Testing experience

Our testing philosophy was the well-known extreme-case testing. We tried conceiving complex and extreme cases in the worst possible programming style at each stage of the development and we were quite confident to meet real C programs after having successfully compiled these horrible things.

To our surprise, in several cases existing programs - written in good style - were clever enough to beat the first version of the compiler. They have been an invaluable help in testing and debugging - and we still have some bug reports.

3.3.2. Comparison (performance hints)

The following statements concerning performance characteristics are intended to give a basic qualitative estimate and do not constitute a meticulous measurement.

Our C language reference basis has been extended by using - examining, testing - existing C compilers: cc on Berkeley 4.2 UNIX and ULTRIX, the APOLLO/DOMAIN cc compiler and the ACE C cross-compiler [12]. This latter has particular significance because it can generate code for the same processors as we do.

3.3.2.1. C Language:

We claim that our interpretation is a full one, that is, the recognised language is the C language as it should be. PCC [11] is deficient in this respect: not surprisingly, it implements a more orthodox, earlier version of C. Interestingly, the error handling and recovery seems to be more rudimentary than ours.

APOLLO/DOMAIN/CC is nearer to the expected standard but there are some points that need clarification/debugging. On the other hand, its error recovery and report is much more satisfactory compared with PCC. Our error messages are detailed enough; warnings could be improved.

Probably the vendors of commercial compilers will improve their compilers to reach the standard soon so our declared advantage is not very significant. If not then it would mean a dead standard.

3.3.2.2. Compilation time:

With cross-compilers compilation speed is not an important factor; we do not have results of comprehensive standard mix compilations yet. A rough estimate (from several sequential/parallel compilations of valid, large programs) is as follows: ULTRIX/CC is better with a factor of 4; DOMAIN/CC is better with a factor of only 1.25 which is shockingly favourable to our compiler.
The experimental version was a debug version with several debug listing phases inside and with simple input/output functionality which is going to be improved.

3.3.2.3. Run-time:

This is the most important factor: here again we do not have sufficient data: presently the ACE compiler is better than ours: we have been improving the code-generation phase together with changes in the front-end tree passed to the back-end.

4. Acknowledgements

Let me begin with a personal experience: when I started writing the C compiler I was filled with Ada. C was a somewhat abstract object of my activity: I had not known C before. Compared to other languages it seemed to be rather weird, even poor. Ever since I gathered more profound knowledge and experience about it my opinion has changed: I can’t help admiring its developers, their concise knowledge on software, on system programming, and their view of the programming paradigm. Firstly I wish to express my esteem towards Dennis M. Ritchie and Ken Thompson. I do not claim C to be the best programming language ever: but it is excellent in its expressive power and efficiency.

It was David Foster who introduced me into the secrets of the previous CERN compilers and the layout of the parsing phase, the lexical analysis, and the error recovery parts of the C compiler do not only reflect his ideas but mostly are borrowed (and modified) from his previous code. He is the co-author of this front-end: his revealing suggestions have been always inspiring.

Julian Blake’s consulting advice, his clear view of the whole cross-software suite, his extensions to the Intermediate Language and his knowledge on the details of compiler writing have been a necessity to achieve a working version.

Jonathan Caves implemented the required changes in the code generator (back-end) in a short time. He has also extensively tested the front-end and took an active part in planning decisions.

5. Bibliography

Motorola ISBN 0-13-566878-6

4th ed. Motorola 1986

[3] Blake, J.; H. von Eicken; D. Foster: Developing Programs for the Motorola
68000 Microprocessor at CERN
in: Computer Physics Communications 38 (1985) 315-318
North-Holland, Amsterdam

CERN DD SW Report 1987

for the Cross Compiler Suite
CERN DD SW Report 1984

Software - Practice & Experience, Vol. 16(2) 101-106 (1986)

Prentice Hall 1978

Prentice Hall 1984
C cross-compiler for the Motorola 680x0

[9]
X3J11 Draft C Standard Oct. 1986 in:
X/Open Portability Guide Vol 4: Programming Languages

[10]
MUFOM: IEEE Std. 695 for Microprocessor Universal Format
Object Modules Order No. DC 954

[11]
Johnson, S.C.: A Portable Compiler: Theory and Practice
Proc. of the Fifth ACM Symposium on Principles
of Programming Languages 1978

[12]
The ACE Expert Compilers
ACE Associated Compiler Experts Amsterdam 1986