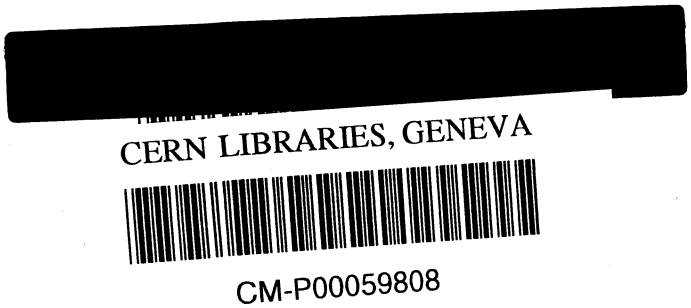
CERN-DATA HANDLING DIVISION
DD/78/20
I. Willers
J. Montuelle
October 1978



Cross Software Using a Universal Object Module Format, CUFOM

Submitted to Euro IFIP 79
(London 25-28 September 1979)

Montuelle,Willers

ABSTRACT

The CERN Universal Format for Object Modules, CUFOM, has been designed to be machine independent. The format of a CUFOM object module does not depend upon the computer on which the object module is to be loaded. Also, the object module may be stored and manipulated in a machine independent manner.

The main area of application is in cross software, where compilation or assembly, linkage edition and part of the loading process takes place on a host computer. One surprising result is that the linkage editor is independent of both the host and the target computer.

1 INTRODUCTION

An object module is a stored representation of the information necessary to load the bit patterns, that are the machine instructions and data, into the computer's memory to form part of, or all of, an executable program. The format in which this is stored is known as the object module format.

At present there are a large number of object module formats each of which is associated with a particular computer, range of computers or manufacturer. An essential requirement of any computer is the ability to load executable programs into its memory. So the 'loader' and its associated object module format appear early in the life of a particular type of computer. Some of the design decisions taken at this early stage cannot be revised since many other basic processors require that certain design features remain constant. In the case of cross software and distributed processing, where program preparation may take place on different computers, a common format is desirable. CUFOM, CERN Universal Format for Object Modules, is an object module format which is designed to represent object modules for different computers in a uniform manner.

The programs that produce object modules are called compilers or assemblers. They translate a readable human representation of a program, or part of a program, into an object module. These object modules are manipulated using a linkage editor. A linkage editor will assemble many object modules into a single object module while establishing the links between modules. A loader then forms an

executable program in the computer's memory. The diagram in fig.1 represents this process in a schematic way.

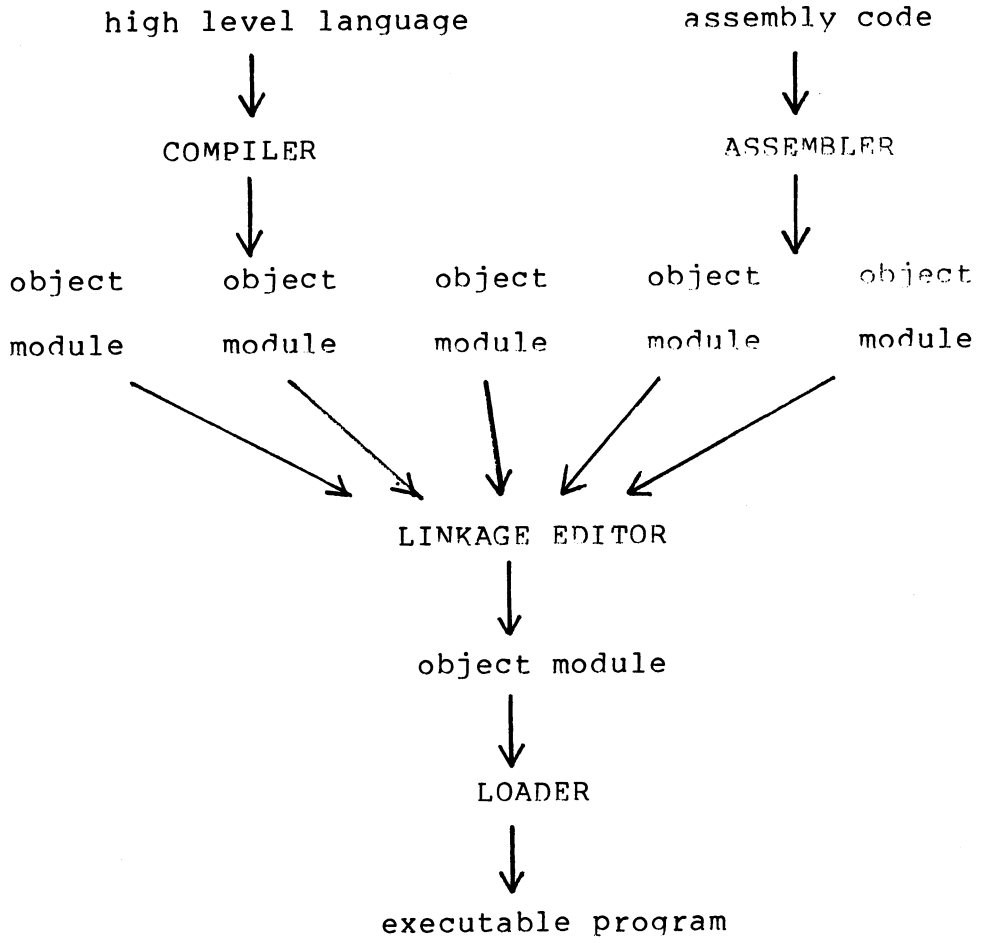


Fig.1 Forming an executable program

In cases where no linking is required, an object module may be directly presented to the loader.

CERN's main computing centre consists of two IBM and three CDC computers. There is a medium sized centre based upon a DEC 10 computer. The principal minicomputers are NORD 10's, PDP 11's, HP 2100's and Modcomp II's. The main microprocessors which are

officially supported are the 8080 family, 6800 family, Texas Instrument's TMS 9900 and Feranti's F100-L. The CERN network (see {1}), CERNET, now enables computers of various types to communicate with each other with a previously unknown ease. INDEX (see {2}), a hardware type of telephone exchange, enables lower speed communication between microprocessors and multi-user time sharing systems on other computers.

Small configurations of microprocessors and minicomputers have only basic program development tools. In a network environment much of this program development process is better done using a friendlier system which exists on another computer. Most substantial gains are made in reliable and large file storage, fast execution of program development aids and a multi-user environment.

Our concern was to supply cross software support on computers, called hosts, to generate programs destined to execute on other computers, called targets. An initial restriction of three host types and seven target types gave us an upper limit of twenty one sets of programs. At this time, fifteen possible combinations are covered by this software.

2 EARLY DECISIONS

We studied the process of compiling or assembly to program loading so that it could be as host and target independent as possible. Here are some of those early decisions.

(i) Write programs in a portable language. Our software is written in BCPL {3}. This language is known to be well suited for writing compilers and assemblers {4}, and it is available, in CERN, on the prospective hosts. The language is closely controlled at CERN so that no new extensions, side effects or peculiarities can be introduced into the standard language as described in {5}.

(ii) The BCPL compiler is the only compiler supplied for users. This is not such a satisfactory decision since the language may be unsuitable for the user's requirements. However, for the implementors, the lack of manpower made this an ideal early decision since any target computer with such a compiler becomes a potential host type.

(iii) The effort involved in the production of a new asembler for a new assembly code and/or a new target computer should be minimised. The techniques of assembler design have already formed the subject of papers, see {6} and {7}. In our case, the code translation is table driven but the instruction formats which are referenced by the table are directly described by the BCPL program which analyses them. The common parts of all the assemblers are parameterised in such a way that the programming effort is equivalent to the one needed with an assembler generator, but without introducing a specific language and its processing.

(iv) Design a machine independent object module format. This format and its effect on portability forms the subject of this paper.

3 DESIGN FEATURES

During the design stage of CUFOM certain aims were specified and later achieved. The important features are:

(i) CUFOM, in order to be universal, must be capable of expressing the constructs in use on the various computers. All design features are extendible in an attempt to cover all future possibilities. It is possible to assign a new type or a new function whenever that is necessary.

(ii) Extendibility to new target dependent pecularities is assured by providing programming possibilities via function calls. It was tempting to include function definition within CUFOM. However, this would have made CUFOM approach the complexity of a programming language. We prefered to describe these specific functions within the BCPL programs which process the object modules.

(iii) Complexity of CUFOM should be kept to a minimum. The simple structure of a CUFOM module gives fast processing and object modules which are easily understood. It is easy to teach the principles of linkage edition and loading without referencing a particular computer.

(iv) CUFOM must be transportable and must be handled by standard BCPL routines. Both these considerations enforced a character representation for the object module. Both INDEX and CERNET were able

to accurately handle character data. Character files are handled by BCPL in a uniform way on all computers.

(v) The software must remain as host and target independent as possible. It is now possible to produce compilers and assemblers for specific targets which are host independent. The linkage editor is both host and target independent. The only changes that are made are in providing a natural user interface for each computer, with defaults set to be appropriate to that host. In all cases the BCPL code which is to be changed is kept in well defined program modules.

4 SOFTWARE IMPLEMENTATION AND MACHINE INDEPENDENCY

In this section each of the programs that form the cross software are examined in more detail. The amount of machine independency that was possible was greatly increased by the use of CUFOM.

4.1 Compilers

The BCPL compiler and an associated BCPL library form a complete BCPL language implementation. The definition of the language remains constant for all the host and target combinations. The syntax analysis stage of the compiler is machine independent. It is constructed as a separate process which produces a machine independent intermediate code known as OCODE.

The next stage of compilation, OCODE to object module, is target dependent by its very nature. It is implemented as a separate process taking OCODE as input and producing an object module as output. It is easy to change from one target to another by interchanging one code generator with another. Originally a change of host computer also implied a change of object module format with its associated difficulties and was rarely undertaken. With CUFOM, however, the code generator becomes host independent and may be moved with little or no change.

4.2 Assemblers

The assemblers are designed to conform to the manufacturer's description of his assembler and assembly code. However, there are many aspects that assemblers have in common such as fast look up tables for names, macro expansion capability, statement by statement processing, maximum of two passes on the source code etc. These common functions are built into a general assembler so that it is necessary only to describe the instruction codes and formats for a complete assembler to be built. The instruction codes and format types are specified in a table. The corresponding instruction formats are analysed within a separate program module. Common functions are used to evaluate the label parts, operation parts and and address parts of instructions. The user interface normally needs changing in order to conform to local standards and character conversion may be necessary.

The object module that is produced is in CUFOM. A high degree of host independency has now been obtained, thus avoiding the previously time consuming process of converting assemblers to produce an object module with a different format. The CUFOM generating module became a standard or common function for all assemblers.

4.3 Machine independent linkage editor

So far all the design decisions have aided portability so that it is cheaper to move programs from one host to another. Now the extra cost will begin. If the manufacturer's linkage editor was used we would not have to write our own. It turned out however that only one program was necessary since that program was both host and target independent. Host independency was obtained due to the standard way CUFOM files are handled from a BCPL program. The target independency was obtained by using CUFOM in a way which will be described in the section 5.4.

The input to the linkage editor will be two or more CUFOM object modules and the output is a single object module, again in the format of CUFOM. During the linkage edition, references within object modules to values defined in other input object modules are replaced by the appropriate values. Any reference which is not satisfied remains in the object module and the user is warned of this fact. This new CUFOM object module may be presented to the linkage editor for further processing with other CUFOM object modules. When no

outstanding references remain the CUFOM object module is said to be loadable.

At the cost of writing this program a greater degree of machine independency was obtained in a larger set of compilers and assemblers.

4.4 Loading processors

Some process must be responsible for turning an object module into an executable program. In the case of cross software, the actual loading process may include format conversion and a transfer to the target computer. This will include programs which exist on both the host and the target computers. The user, however, will view the whole process as single user initiated operation. The suite of programs necessary for loading is known collectively as the loading processor. For the implementor, there are a variety of possibilities.

(i) CUFOM loader on the target computer. The loader accepts CUFOM as input. This can be combined with a program on the host computer which produces a simple CUFOM loadable module. Hence any residual linking information may be removed and relocation may take place on the host computer. This simplified load module then contains few CUFOM commands which may be analysed by the target loader more easily.

(ii) CUFOM to target format converter on the target computer. The program that receives the CUFOM object module converts it into the

local format. This solution may require local secondary storage which is normally present in the case of a minicomputer and may be absent in the case of a microcomputer.

(iii) CUFOM to target format converter on the host computer. This program is the same as in (ii) above but resides on the host computer. This solution is attractive since processing remains on the host computer and, at CERN, it is the most commonly used method. In some cases, the transfer medium may have to handle the transfer of binary data since many object modules are in a binary format.

The above list of possibilities is not exhaustive. There are other examples of the loading process, e.g. host has direct memory access, DMA, to the target computer's memory or the host may even share a memory module with the target computer. In both these cases the host computer will load directly into the target computer's memory.

The software problem centres round format conversion programs. Each of these programs is divided into two program modules which are the CUFOM analyser and target format generator. By designing a suitable interface between these modules it has been possible to simplify the writing of these programs. Relocation, which may be needed at this point, is handled by the CUFOM analyser. CUFOM expression evaluation now takes place since all the necessary information is available. The CUFOM analyser must know details concerning the target computer such as word length, address storage etc. Most of the target dependent properties can be parameterised and the code be written to adapt to different circumstances. The target

format generator remains simple.

5 CUFOM DESCRIPTION AND LINKAGE EDITION

CUFOM object modules appear as the result of a compilation or assembly, linkage edition and, sometimes, the first part of the loading process. These three types of modules are called linkable, loadable and simple loadable. The whole range of CUFOM can be required for a linkable object module, a subset of CUFOM for a loadable object module and a further subset for the simple loadable object module. These sets may be defined according to the requirements of each target computer while preserving the linkage edition possibilities.

Some of the more commonly used CUFOM commands will now be described. For a full description of the CERN universal object module format it is necessary to refer to {8} which is an aid to the implementor of programs that produce or handle CUFOM object modules.

In order to be easily transportable and easily stored in any file system the object module is stored in the format of a character file. Each line normally has a limit of 72 characters with the possibility of either terminating a line with a full stop or indicating continuation on the next line with a colon. Thus the length of a 'CUFOM line' is unlimited. In practice an artificial limit is applied when buffering is required in the loading processor.

A CUFOM line consists of a CUFOM command, which is always two letters, and data relating to that command. Some of these commands and their data are outlined in the following sections. The data itself can represent the bit patterns to be loaded or can be expressions or functions connected with the linking or loading process. Various sets of variables are defined, some with special meaning and some which are left to the implementor as working variables. All expressions are in reverse Polish notation for quick processing. All character strings in CUFOM are preceded by a two digit hexadecimal length.

The object module types are now described in order of increasing complexity. The CUFOM commands necessary for a given type are also necessary for the more complex types.

5.1 Simple loadable module

Every object module that is processed must begin with a module begin, MB, command and end with a module end, ME, command. Each object module may be named and also carries the name of the target computer. Normally, the body of a module consists of one or more numbered sections. In the case of the simple loadable module there is just one section of type absolute.

The assign, AS, command can be used to set CUFOM variables, such as the start address, loading pointer etc., and to modify the contents of locations pointed to by CUFOM variables while using the full expression manipulation that is allowed. The hexadecimal representation of the information that is to be loaded is placed in the data of a load, LD, command.

5.2 Loadable module

A loadable module can consist of many sections of different type. Absolute sections are of type A, simple relocatable sections are of type R and common relocatable sections are of type C. The section type, ST, command associates a type and, optionally, a name with the different section numbers. Each section has a loading pointer called P. The loading pointer of a section may be referenced by using the number of that section, e.g. P9 is the loading pointer for section 9. P, not followed by a number, refers to the current section. The P pointer may be set using the assign, AS, command.

Relocation information is expressed as a series of function calls which form the data of an execute, EX, command. These functions are represented by a single letter, may take arguments and may be followed by a repetition count. They are only executed at loading time and act upon the data of the previous load command. In the case of a simple load where no relocation is required, the load function, L, is called. The simple relocation function, R, may be used but other more

complicated functions may be defined. To load five items and then relocate two items, the data of the execute command appear as 'L5R2'. The relocation function may have one argument to indicate relocation with respect to some value which is not the base pointer of the current section.

5.3 Linkable module

A linkable module contains and requires information which is needed by or contained in another linkable module. For example the internal values within a section of one module may be required within another module. The result of the linkage edition is itself capable of further linking since the internal information can be preserved.

An internal symbol is defined by a name internal, NI, command and its value assigned to an internal, I, variable. An external symbol is defined by a name external, NX, ccommand and referenced by an external, X, variable. The names of externals are matched with the names of internals in order to accomplish the linking process.

5.4 Linkage edition

The linking process is demonstrated by an example. The two object modules in fig.2 are linked in order to provide the object module shown in fig.3.

In the first module, with name GOLD, the four external symbols: T1, T2, T3 and P5 are referenced and the internal symbols: V1, V2 and V3 are declared as entry names. The values of V1 and V2 are absolute and are equal respectively to 8 and hexadecimal A. The value of V3 is relative to the origin of section 1 with a positive offset of 16. In this example let the target computer have byte addressing, a 16 bit word and an address length equal to the word length. Thus, for the absolute section in module GOLD, the execute command expresses that the value which will be loaded in the word of address 8 will be the value of the symbol T3 plus hexadecimal value A. For the relocatable section in module GOLD, the execute command expresses that the values which will be loaded in the relative address hexadecimal 10 onwards will be:

the value of T1,

the value of T1 plus 4,

the value of T1 plus 8,

the hexadecimal value 4DEC,

the value of T2 minus the value of T1 plus hexadecimal A,

the value of P5.

```
MBJB007,04GOLD.
ST0,A.
ST1,R.
SB1.
NX1,02T1.
NX2,02T2.
NX3,02T3.
NX4,02P5.
NI1,02V1.
NI2,02V2.
NI3,02V3.
ASI1=8.
ASI2=A.
ASI3=R,16,+.
ASS=1C.
SB0.
ASP=8.
LD000A0506.
EXR(X3).
SB1.
ASP=R,10,+.
LD0000000400084DEC000A0000.
EXR(X1)3LR(X2,X1,-)R(X4).
ME.
```

```
MBJB007,06FINGER.
ST0,A.
ST1,R.
SB1.
NI1,02T2.
NI2,02T1.
NX1,02V1.
NI3,02T3.
NX2,02V3.
ASI1=R,2,+.
ASI2=R,4,+.
ASI3=R,8,+.
ASS=A.
LDFFF4002D130C000406A0.
EXR(X1)L2RR(X2).
ME.
```

Fig.2 Two linkable modules

In fig.2 the relocation function has, as an argument, various expressions which include external variables. When the value of an external variable is known the linkage editor replaces references to it by the appropriate expression. For example, the value referenced by X3 is 8 after the relocatable base of section 1 in the produced module.

Fig.3 illustrates the result of linking the two modules shown in fig.2. Fig.4 shows the map listing. As the module may be linked with others in a further step, the internal names are retained. Two comment commands are introduced to separate the lines which will be printed on the map listing when a symbol will be encoutered. The external symbol P5 was not defined. An appropriate message is printed on the map listing of the current linkage edition and a name external command is produced to allow a further linkage. In the execute commands the other external references are replaced by equivalent expressions. In the part related to the second input module, the origin of the relocatable section of the output is modified by the assignment command 'ASR=R,1C,+.'.

```
MBJBØØ7,ØAGOLDFINGER.
CO2,13 FROM MODULE : GOLD.
STØ,A.
ST1,R.
SB1.
NX1,Ø2P5.
NI1,Ø2V1.
NI2,Ø2V2.
NI3,Ø2V3.
ASI1=8.
ASI2=A.
ASI3=R,16,+.
ASS1=26.
SBØ.
ASP=8.
LDØØØA0506.
EXR(R1,24,+).
SB1.
ASP=R,1Ø,+.
LDØØØØØØ4ØØØ84DECØØØA0000.
EXR(R,2Ø,+)3LR(R,1E,+,R,2Ø,+,-)R(X1).
CO2,15 FROM MODULE : FINGER.
NI4,Ø2T2.
NI5,Ø2T1.
NI6,Ø2T3.
ASR=R,1C,+.
ASI4=R,2,+.
ASI5=R,4,+.
ASI6=R,8,+.
ASP=R.
LDFFF4ØØ2D13ØCØØØ4Ø6AØ.
EXR(8)L2RR(R,6,-).
ME.
```

Fig.3 A linked module

```
PAGE    1                        CUFOM LINKAGE EDITOR ( 27 SEP 78 )

==0==>  "GOLDFINGER" LINKED ON IBM370/168, DATE= 29 SEP 78


==1==>  FROM MODULE : GOLD
         ABSOLUTE SECTION      0
         RELOCATABLE SECTION   1   SIZE=0026(   38)
           UNRESOLVED REFERENCE                                        P5
           DEFINED IN SECTION  0                      VALUE=0008       V1
           DEFINED IN SECTION  0   (NOT REFERENCED)   VALUE=000A       V2
           DEFINED IN SECTION  1                      OFFSET=0016      V3

==1==>  FROM MODULE : FINGER
           DEFINED IN SECTION  1                      OFFSET=001E      T2
           DEFINED IN SECTION  1                      OFFSET=0020      T1
           DEFINED IN SECTION  1                      OFFSET=0024      T3


>>>>>>>   1 UNRESOLVED REFERENCE <<<<<<<

>>>>>>>   NO  ERROR  DETECTED   <<<<<<<
```

Fig.4 A map listing

All linkage edition is done in this way. Load, LD, commands are not examined by the linkage editor and so the correspondance between the load command's data and the execute command's functions need not be known. The linkage editor's sole function is to manipulate CUFOM variables and expressions in order to resolve external references and section combinations.

ACKNOWLEDGEMENTS

The birth of CUFOM took place during informal meetings which included, in alphabetical order, Dave Garnett (Cambridge University), Steve Mellor (now at Lawrence Berkeley Laboratory), Martin Standley (now at Norsk Data Ltd) and Tim Streater. Subsequent practical experience was gained by Chris Adams (Rutherford Laboratory) and Steve Mellor. The final version of CUFOM was produced by Jean Montuelle. Ian Willers assured the continuity of this work as well as contributing during all of the stages described above.

Thanks should also go to Julian Blake, Tor Bloch, Dietrich Wiegandt and all others who encouraged us to continue this work.

REFERENCES

{1} J.M. Gerard, CERNET, The CERN Packet-switching Network, Data Handling Division report DD/77/20, CERN, December 1977.

{2} H. Slettenhaar, INDEX A Digital 'Telephone Exchange' System, Data Handling Division report DD/77/11, CERN, September 1977.

{3} M. Richards, The BCPL Programming Manual, The Computer Lab., University of Cambridge, 1973.

{4} M. Richards, BCPL - a tool for compiler writing and system programming, Spring Joint Computer Conference, 1969, 557-566.

{5} Steve Mellor and Ian Willers, The BCPL mini-manual, Data Handling Division, CERN, Febuary 1978.

{6} S. Antoy and F. Cordano, CPCA: a general purpose cross assembler, Euromicro, vol.4, no.4, July 1978, 222-226.

{7} J.D. Nicoud, Common Instruction Mnemonics for Microprocessors, Microscope, vol.1, no.3, April 1975, 22-29.

{8} Jean Montuelle, CUFOM, The CERN universal format for object modules, Data Handling Division report DD/78/ , CERN, October 1978.