CERN - DATA HANDLING DIVISION
DD/78/21
J. Montuelle
October 1978

C U F O M

The CERN Universal Format for Object Modules

D-ag

CONTENTS

1 INTRODUCTION

The function of a language translator (assembler or compiler) is to convert a source file to an object file: a source program is translated in an object module. In simple cases, such generated code is directly processed by a loader to initialize memory space before the program is executed by setting the program counter with a starting address. Basically, object modules consist of loading addresses and values which are to be loaded. But, normally, this is improved by offering program relocation, linkage edition and library facilities.

The program relocation allows programs to be loaded in regions which are dynamically allocated. In writing his source file, the programmer refers to data and instructions by names or labels without knowing their physical addresses. Object modules are generated in such a way that the loader can perform the address translation (the physical addresses are calculated according to the loading addresses).

The linkage editor processing allows the programmer to divide his program into several parts. Each part may be coded in the programming language best suited to it. Before loading the resulting object modules are combined and linked (the cross references between parts are resolved).

Object modules stored in a library can be placed in the input to the linkage editor, either upon request or automatically (if unresolved external references remain after all input to the linkage editor is processed, an automatic library call routine retrieves the modules required to resolve the references). The program which is devoted to the library management is called the "librarian".

To support all these facilities, object modules convey information of different kinds. Thus an object module format must be defined. This format specifies how to encode each component of the information. As the computer manufacturers include in their operating system their own loader, linkage editor and library representation, they define the format which is the most adapted to their own use.

Consequently, when a language translator is transported from one type of computer to another, a sizeable part of the translator must be modified. Roughly, we call the affected part the code generator and the stable part the syntactic analyser. These two parts may communicate by function call or by an intermediate code file. The modification of the code generator is necessary because the hardware (e.g. instruction set, word length, memory organization) is different. But, as the object module format and the type of processing change, this modification is generally a complete rewriting because even the functionnal decomposition cannot be kept. If the translation and the execution of all the programs are done on the same computer, it is nevertheless the simplest solution.

2 WHY A NEW OBJECT FORMAT?

When developing software for an installation which offers a limited range of service programs or a low availability (typical example: microprocessors) it is interesting, if possible, to use another installation which can provide a larger computing capacity (in speed, file space, number of concurrent users, editing flexibility, etc.). **Cross-software** is then implemented on this computer (we call it the host). By this means, an object module is constructed before its loading on the target machine. The loading can be done:

- by the loader existing on the target machine. If the format accepted by the loader is different to the one supported on the host, a format transformer (generally on the host) must be called before the effective loading.

- by a loader existing on the host if a direct access to the target memory exists (DMA).

All programs running on the host and involved into the loading step (e.g. format transformer, loader on the host) are called "pushers". Later on, we reserve the term "loader" for a loader running on the target machine. When the residency of the loader is of no importance, we will use the term "loading processor". The program which transforms an object module from the target format to the host one is called a puller (for example to link-edit a program with standard routines already provided by the manufacturer of the target).

The object format used until the loading step is usually the one defined by the manufacturer of the host machine, in order to use facilities that are already present (e.g. the linkage editor, the structure of a code generator). But, as the manufacturer's format is not adaptable (because it is defined for another hardware architecture) the advantages of using it are not so convincing.

> For example if the host is an IBM 370 (with a 24 bit address) and the target is an INTEL 8080 microprocessor (with a 16 bit address where the most significant bits are in the second byte) a pre-processor and a post-processor are needed for proper processing by the IBM linkage editor (address manipulation).

And if the architecture of the host and the target are very different, the host linkage editor cannot be used because it is too specific.

More complications arise if there are a wide range of host and target computers for which the same set of "cross-services" must be offered (e.g. the CERN microprocessor support). To limit the cost of implementing and maintaining a general cross-system for a large combination of hosts and targets, this software must be as portable as possible. First of all, the service programs must be written in a high level language which is available on all the hosts and which allows the greatest

possible program portability (in our case, this language is BCPL). The next step is to define a standard format for all the object modules. This standardization provides the following advantages:

- A host independency and a lower target dependency for the code generators. A code generator is only modified when a new target machine must be supported. This modification is only devoted to the peculiarities of the target hardware; the object module structure and encoding remain constant. It is even possible to design a general code generator which is driven by tables describing the target peculiarities.

- A linkage editor and a librarian which are host and target independent.

- A standard set of services (e.g. documentation, user interface). A user who changes from one computer combination to another must learn only those things which are logically host or target dependent (no tricks are needed to make a specific system to do what he wishes).

- A standard language which facilitates the implementation of programs manipulating object modules (e.g. loaders, "pushers").

All these considerations have lead to the following proposal for the CERN universal format for object modules (CUFOM) and a standard set of "CUFOM processors" for manipulating these object modules (i.e. the linkage editor, the "pushers" and "pullers").

3 PRELIMINARY TECHNICAL DISCUSSION

The most important design aim for CUFOM is the host-target independence of the linkage editor. Thus the definition of CUFOM and the description of the linkage editor processing cannot be dissociated. The following chapters explain the syntactical and semantical rules which must be obeyed for a correct linkage edition of the object modules. New rules may be added to define the format for object modules of a particular target machine. For example a loading processor must define what is a load module (by giving the subset of CUFOM it accepts). The module produced by the linkage editor is another object module in CUFOM and can be relinked with others (if some external references are still unresolved).

As for a given target machine, the code generators and the loading processor may only use a subset of the CUFOM possibilities, we shall try to point out what kind of subset may be compatible with the linkage edition. Nevertheless the best way for knowing if the output modules of the linkage editor fall into the subset of the input ones, is to run some tests.

4 OVERALL DESCRIPTION OF CUFOM

Object module files are text files: the unit of information is the character. Data which are not by themselves character strings (not a symbolic name, a CUFOM mnemonic or delimiter, etc.) are represented by the hexadecimal notation of their value.

For example, if the following bit pattern:

11101001

is the binary representation of an instruction to load, it will be represented by

E9

in the CUFOM text.If the characters are coded in ASCII, its binary representation on the object module file would be:

0100010100111001

As an object module is almost entirely composed of values to load, its size in the CUFOM representation is double that of a direct binary representation. But much portability is retained by keeping the data in character form. We know from experience the problem of marrying a byte oriented machine like the IBM with the CDC which uses a 60 bit word. As the adopted representation does not favour a particular machine structure, it is easy to encode or decode the information on any computer and this in a standard way (host independence). As interactive facilities are usually text oriented, the text format allows easy reading, file manipulation (e.g. transfering files through a network with automatic character code translation) and, if necessary, editing operations. Also the programming facilities for text handling are better than that for binary.

An object module consists of a logical file containing a number of commands. These commands are introduced by a two letter command word and are terminated by a full stop. Thus

ME<body>.

is a well formed command. The <body> depends on the command type. In the case of the example, <body> is the empty string. This command signals the end of the object module ("ME" is the mnemonic for Module End).

The length (in characters) of a CUFOM command is variable and there is no limitation for that length. To map CUFOM logical files on physical files (with a limited line size), a single command can occupy several successive lines. This may occurs for the LD (load), EX (execute), AS (assign) and US (use) commands. The other CUFOM commands can be represented on a single line (with a sufficient length such as 72 characters). If such a command does not end with the current line, the "continuation sign" (:) must be the last character of the line (before the "end

of line" character).

e.g. LD<part1 of the body>:
<part2 of the body>:
<last part of the body>.

There is no fixed column number for receiving the "continuation sign". A command must be split logically. In a LD command the "continuation sign" separates two elementary values (e.g. instructions, addresses). In the three other commands, it may be used to split an expression or a parameter list on two lines. In such cases, it follows a comma sign (,) or a semicolon sign (;). Finally, in a EX command, it may separates two <EX-item> (see sect. 10.2).

When possible it is better to replace a multiple line command by several single line commands. Thus the command of the previous example, if it was not followed by an EX command , is equivalent to:

LD<part1 of the boby>.
LD<part2 of the body>.
LD<last part of the body>.

5 BASIC CONSTRUCTIONS

5.1 Notation used for syntax definitions

The meta-language used for the syntax definitions is derived from the Backus-Naur Form.

Repetitive concatenation of objects are indicated by the notation:

```
(          )j
< <object> >
(          )i
```

where i and j are respectively the minimum and the maximum number of repetions allowed. j may be an arbitrary (undefined) limit denoted by n. i may be 0, indicating that the <object> may be omitted. If i = 0 and j = 1, the notation {<object>} will be used.

In CUFOM, the spaces are not used as delimiters. Thus the spaces which may appear in syntax definitions are only there for readability and must by ignored. But the space character can be used in a <char string> as can any other <character>.

5.2 Elementary syntactic objects

CUFOM uses a small number of data types to provide standard means of expressing constructions. There are the follows:

```
<hexdigit>      ::=Ø|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

                   (             )n
<hexnumber>     ::=< <hexdigit> >
                   (             )1

<nonhexletter> ::=G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<letter>        ::=A|B|C|D|E|F|<nonhexletter>

<alphanum>      ::=<letter>|<hexdigit>

                   (             )n
<identifier>    ::=< <alphanum> >
                   (             )1

<character>     ::= any character of the host computer set

<string length>::=<hexdigit><hexdigit>

                                  (             )n
<char string>   ::=<string length>< <character> >
                                  (             )Ø
```

Where the hexadecimal value of <string length> must be equal to n; n must be less than 256.

e.g. 14A STRING WITH SPACES

<CUFOM variable name> ::=<nonhexletter>{<hexnumber>}

It is a name which references an internal variable of a CUFOM processor (see sect. 12).

<diadic operator> ::=+|-|*|/|>|<|&|\

with the respective meaning: plus,minus,times,divide,shift rigth,shift left,and,or.

<monadic operator>::=!

with the meaning: indirect (!E refers to the content of the storage cell whose address is given by the value of E).

5.3 Expression

Expressions are written in the reverse Polish form. Expressions are evaluated when the values of their operands are known (it is generally at loading time). The binary representation of each operand must fit into a word of the computer where the evalution is done.

```
<expression>::=<hexnumber>
               |<CUFOM variable name>
               |@<function call>
               |<expression>,<monadic operator>
               |<expression>,<expression>,<diadic operator>

<function call>::=<identifier> { ( <parm list> ) }
```

When a **function** call appears in an expression, it causes the computation to be performed on the values given in the <parm list> (if any) according to the function definition. The resulting quantity which must be a word value is put on the evaluation stack. No standard function is provided with CUFOM: the function definitions are target dependent.

```
                                (              )n
<parm list>    ::=<parameter>< ;<parameter> >
                                (              )Ø

<parameter>    ::=<expression>|"<char string>
```

e.g. R3,@MAX(K4;K6),3Ø,*,+
is evaluated as R3 + (MAX(K4,K6) * 3Ø)

An expression may be classified as being **absolute** or **relocatable.** A relocatable expression is affected by program relocation whereas an absolute expression is not. When an expression must be absolute, <absolute expression> is used in the syntax definitions.

We distinguish <l-expression> from the general <expression> to define what kind of expression must be on the left part of an "assignment command".

```
<l-expression>::=<CUFOM variable name>
                 |<expression>,!
```

The value of <expression> on the last line must be an address within the target memory.

e.g.	I5	references the 5th I variable
	FF,!	references the target memory location with address 255
	L3,4,+,!	references the 4th location of the region where section 3 will be loaded.

6 COMMENTS

CUFOM is an intermediate language which is usually not seen by the user. But even if object modules are not intended to be read or directly modified, comments may be useful in some cases.

For example, one of the functions of the MAP listing is to explain the overall structure of a CUFOM module (which can be complex if it is the result of a multiple linkage edition). To facilitate the comprehension, CUFOM comments can be copied in the MAP listing.

Comments may also be used to record the name of the source file to which it corresponds, the date of creation, a flag indicating if the compilation was successful, etc.

6.1 CO (COmment) command

<CO command>::=CO{<hexnumber>},<char string>.

e.g. CO3,1AI/O CONTROLLER (15 JUNE 78).

<char string> is the comment itself. <hexnumber> can be understood as a "level of comprehension": the higher the value, the more the comment participates in the detail of the overall structure of the final program.

The effect of the command depends on the processor which reads it. We only define what the linkage editor does. If the level indication is present and less than or equal to the "comment parameter" (given by the user at linking time) the <char string> is printed on the MAP listing preceded by its level value in brackets. All the CO commands are also copied to the output CUFOM file. But if the level indication is present and greater than 1, it is increased by 1 before the copying.

7 MODULE DELIMITERS

7.1 MB (Module Begin) command

<MB command>::=MB<identifier>{,<char string>}.

e.g. MBI8Ø8Ø,Ø7MONITOR.

The MB command must be the first command of an object module. <identifier> identifies the target machine. This information is used by the linkage editor to check if all the modules are homogeneous (destined for the same target machine). On the other hand, a loading processor accepts only modules for its specific target machine.

If present, <char string> gives the name of the module. This name can be set by a special directive in the source program (processed by the language translator) or given as an option to the processor (translator,linkage editor or "puller"). This name will appear in the MAP listing.

7.2 ME (Module End) command

<ME command>::=ME.

The ME command must be the last command in an object module. It defines the end of the module.

8 SECTIONING

An object module can be divided into one or more sections. A section is a separately controlled region of the program: it has a type which globally influences the actions performed according to the commands that it contains.

Within the same module, a section is identified by a "section number". A section is introduced (or resumed) by an SB command containing its number and it is terminated (or suspended) by a new SB command with a different section number.

8.1 SB (Section Begin) command

<SB command>::=SB<hexnumber>.

e.g. SB5.

The <hexnumber> represents the section number. As the CUFOM processors may not accept a section number greater than the maximal number of sections that they can manage, normally these numbers are the first of the integers (starting from Ø).

If no SB command appears in a module, that means there is only one section, the type of which is absolute. More generally, until an SB command is encountered, the type of the current section is considered as absolute.

8.2 ST (Section Type) command

<ST command>::=<hexnumber>,<letter> { ,<char string> } .

e.g. STØ,A.
ST3,C,Ø3JOE.

In a ST command, <letter> defines the type of the section identified by <hexnumber>. If present, <char string> gives a symbolic name to that section. This name is kept in the symbol dictionary of the linkage editor and thus can be referenced from outside the current module.

A section can be absolute or relocatable. A section is absolute when its loading addresses are absolute (known at assembly or compiling time). A section is relocatable when its loading addresses are relative to a "base" whose value is only known at loading time.

Relocatable sections may be individually loaded into separate storage areas. However, in some cases, the loading addresses of sections must be related. During the loading process, the user may have full control of the storage layout if the "pusher" allows, for example in using overlay directives. But such a description demands extra work from the user. Then, except in very special cases (e.g. overlays), it is preferable

that CUFOM allows interdependancy between sections of different modules. Thus, during the linkage edition, the program structure is built automatically according to the semantics of the source programs.

For the linkage editor, it is possible to express two kinds of relations between sections:

* Several sections must be joined into a single one.
* Several sections must be overlapped.

As these sections usually appear in different object modules, the combination is established according to their symbolic names.

For special storage layouts on the target machine, some conventions may be passed between language translators and a loading processor by using special section names (e.g. prefixed names).

> For example, using the prefix LCP/, the name LCP/GLOBAL indicates that the section is a COMMON preset data area which must be loaded into the low part of the memory. GLOBAL is the symbolic name given by the user. The LCP prefix is destined for the loading processor. But if there are, in different modules, several sections with LCPGLOBAL as name and of C type, they are overlapped during the linkage edition (see example 5).

CUFOM distinguishes between five different section types. In the following list, each type is given by the <letter> of the <ST command>.

A (Absolute) : The section is absolute.

G (General) : The section is relocatable and the linkage editor joins all G-sections into a single one in the order in which they are processed (see example 4). In a module, only one G section can be defined; but in different modules, G sections may be named differently.

R (Relocatable) : The section is relocatable. If R sections that have been processed by the linkage editor have the same name, they are joined into a single one (with that name) in the order in which they were processed. Unnamed R sections are not joined (see example 5).

C (Common) : The section is relocatable. If C sections processed by the linkage editor have the same name they are overlapped. Unnamed C section (blank common) are also overlapped. A warning message is printed if their sizes differ (see example 5).

U (Unique) : The section is relocatable and must have a name. If other U sections processed by the linkage editor have the same name, an error message is printed.

In an object module, the type of a section must be given by an ST command before the section is introduced by an SB command. By default the type is absolute (A).

9 SYMBOLIC NAME DECLARATION

Some symbolic names appearing in source programs must be kept in the object modules for further treatment (e.g. linkage edition). In CUFOM, each symbol is declared by a separate command. Such commands begin with the character "N" followed by a letter which gives the type of that name. These commands are devoted to the resolution of external references.

9.1 NI (Name of Internal symbol) command

<NI command>::=NI<hexnumber>,<char string>.

e.g. NIA,Ø4SINE.

An NI command must be provided for each symbol which is defined in the current module and may be referenced from another module (entry name definition). It indicates that, in the current module, the variable "I<hexnumber>" (see sect. 12.1.1) shall be associated with the internal symbol <char string>. An NI command must always come before all occurences of its corresponding I variable.

As the module produced by the linkage editor can be linked in a further step, the input NI commands are retained (except if an option is given to suppress them).

9.2 NX (Name of eXternal symbol) command

<NX command> ::=NX<hexnumber>,<char string>.

e.g. NX11,Ø4SINE.

An NX command must be provided for each external symbol which is referenced in the current module. It indicates that, in the current module, the variable "X<hexnumber>" (see sect. 12.1.2) shall be associated with the external symbol <char string>. An NX command must always come before all occurences of its corresponding X variable.

As the module produced by the linkage editor can be linked in a further step, the input NX commands which correspond to unresolved references are retained.

10 LOADING COMMANDS

The most important part of an object module is formed by the internal representation of instructions, addresses and data as translated from the source program. Due to the sequential use of memory addresses, it is possible to isolate one or more "blocks" (generally a whole section) where the information is destined to be loaded into contiguous locations. If all the information is completely known, as for some absolute loading, each block represents an exact image (coded in the CUFOM representation) of a memory region after the loading. In such a case, the block may be simply split into one or more successive LD commands (see example 1). Otherwise it is also split but as some values are not the actual ones to load in sequence, complementary information is given in an EX command immediately following the LD command concerned (see example 2).

10.1 LD (LoaD) command

<LD command>::=LD<hexnumber>.

e.g. LD1D7F1D8033C63006A144A144A16010415A2B2A3D2D403C3E.

The <hexnumber> is the concatenation of the hexadecimal representation of the values to be loaded. The way to read this string, in conjunction with a possible EX command, depends on the target machine characteristics (word length, address representation, etc.). Only the processors which are related to a particular target machine (e.g. code generator, "pusher", loader) can encode or decode it. The linkage editor never uses or modifies the LD commands, it copies them.

10.2 EX (EXecute) command

```
                          (            )n
<EX command>        ::=EX< <EX-item>  > .
                          (            )0

<EX-item>           ::=<loading operation>{<hexnumber>}

<loading operation>::=<nonhexletter>{ ( <parm list> ) }
```

e.g. EXL5RLR(XA,X3,-)L3M(2;1B)LR.

The EX command indicates those loading operations which are to be applied during the loading of the values represented in the previous LD command.

The optional <hexnumber> is a repetition factor. If its value is n, it means that the current <loading operation> must be repeated n times. Thus, for example, the string LLLLL is shortened to L5. The <nonhexletter> specifies the operation to

be applied. The <parm list>, if any, represents its parameters.

According to the loading operation, a certain number of hexadecimal digits are scanned in the previous LD command body. The next invoked operation in the EX command is performed in relation with the next hexadecimal digits of the LD command body. In simple cases of loading (simple operations), all the values of the LD command can be loaded first and the (relocation) operations are applied next, directly on the target memory (no buffering is needed).

An LD command and its related EX command cannot be separated by other commands. The way that LD and EX commands are organized into a module depends on the processing type of the language translator and processors which follow the linkage edition. For example, if an object format transformation is needed before the loading, the final format (which is imposed) may induce a special organization into CUFOM modules. Each LD command is, if possible, contained in a single line, to prevent the buffering of several lines. An EX command follows if necessary (see example 2). But some language translators must first output all the code and then the relocation information. In such a case, a program section is translated in one LD command followed by one EX command (see example 3).

10.2.1 Loading operations

The semantics of a <loading operation> depend on the target machine. "L" is the only <nonhexletter> which is reserved in CUFOM. It does not accept any parameters. It only means "load" the current value. Its presence is necessary in the EX command because it permits the movement of the scanning pointer over the LD command body for a fixed number of hexadecimal digits (related to the target machine unit of storage). Its presence is not required if there is no other operation to perform until the end of the LD command.

e.g. EXL3R2L7. is shortened to EXL3R2.
EXL9. can be suppressed.

Normally "R" is the <nonhexletter> for the relocation operation. If it is followed by one expression, it means that the address represented by the current digits within the LD command must be offset by the value of the expression; if it is not, the offset is the value of the R variable of the current section (see sect. 12.2.2).

"L" and "R" are the most frequently used loading operations but it is possible to define other operations for special purposes. For example M(x;y) may mean "multiple load" : the x next bytes encoded into the LD command must be loaded, as a whole, y times into the next memory locations. The definition of the new loading operations is a convention passed between the code generators and the loading processor related to a particular machine. The linkage editor introduces no distortion because it only replaces some CUFOM variables by equivalent expressions. It never has to know the meaning of the <nonhexletter>.

11 EVALUATION OF GENERAL EXPRESSIONS

The loading processor uses and updates a counter which gives the address where the next piece of code (instruction, address or data) must be stored. For absolute loading, this counter may be initialized with a value given by the language translator. More generally, a new value for the counter is required each time that code is to be stored in the memory at some address other than the next location. The object module must be able to convey such directives.

To achieve linkage edition, each external reference appearing in a relocation operation must be replaced with an expression giving the value of the symbol. Thus the object module must provide means of associating such an expression with the symbols it declares.

Consequently some internal variables of CUFOM processors (e.g. load counter or symbol dictionary items) must be manipulated according to operations expressed in the object module. Such variables are called "CUFOM variables" and their symbolic names may appear in <expression>.

Expressions are not only used as parameters of functions or loading operations but they can also be used as left and right part of "assignment commands".

11.1 AS (ASsignment) command

<AS command>::=AS<l-expression>=<expression>.

```
e.g.   ASP=AØØ.
       ASI7=R,8C,+.
       ASK8,2,-,!=X8,X3,-,1,>.
```

The value of <expression> is assigned to the location referenced by <l-expression>. According to the semantics of its left part, an AS command is destined for a particular CUFOM processor.

12 CUFOM VARIABLES

The internal variables of a CUFOM processor which are made accessible for command interpretation are identified by a name with the following syntax:

<CUFOM variable name>::=<nonhexletter>{<hexnumber>}

The <nonhexletter> gives the type of the variable. If several variables of the same type exist, a <hexnumber> is needed. Sometimes this <hexnumber> has a precise meaning (e.g. a section number) and may be omitted if the context provides a default value (e.g. the number of the current section).

As the linkage editor can replace variables by an equivalent expression, the allowed <nonhexletter> are restricted to predefined types. Thus if a name has a completely defined meaning, it should not be used for a different one even when this meaning is not currently in use (for that target machine).

Related to the type of a variable, CUFOM defines some rules to restrict the use of variables inside an expression. The non-observation of these rules may lead to incorrect processing by the linkage editor which does not necessarily check if all the rules are obeyed. Obviously other rules can be added as a convention between language translators and a loader (or "pusher", or format transformer) because, generally, only a limited subset of the CUFOM possibilities is used.

12.1 Variables related to the linkage edition

12.1.1 I variables (Internal symbol values)

<I variable name>::=I<hexnumber>

The I variables are internal variables of the linkage editor. During the first pass on the input, when an NI command is encountered with n as <hexnumber>, the In name references the element of the symbol dictionary which is associated with the symbol defined in the NI command.

In the same module and after this NI command, there must exist one, and only one, "assignment command" which gives a value to this variable. The syntax of such a command must be:

ASI<hexnumber>=<hexnumber>.
if the value is absolute.

or

ASI<hexnumber>=Rn,<hexnumber>,+.
ASI<hexnumber>=Rn,<hexnumber>,-.
if the value is relocatable. n is the number of the section where the symbol is declared (may be omitted).

e.g. ASI7=48A.
ASI1B=R,3C,+.

It is not necessary to write a NI command and a ASI command for the name of a section. By default the value given to a section name is equal to the value of its R variable (or L variable if the section is of type R and is named).

12.1.2 X variables (External symbol references)

<X variable name>::=X<hexnumber>

The X variables are internal variables of the linkage editor and are mapped onto the I variables. The symbol dictionary is built up, as described above, during the first pass. On the second pass when an NX command is encountered with n as <hexnumber> and if the symbol it declares is found in the dictionary, the Xn names appearing in the same module are replaced by an expression giving the value of that symbol. This expression is a <hexnumber> if it corresponds to an absolute address; it is "Rn,<hexnumber>,+" if it corresponds to a relocatable address.

By definition X names cannot be the left part of an "assignment command". Their main use is as parameters of relocation operations. X names must never exist in an object module submitted to a "pusher" or a loader.

12.2 Variables related to the loading

The following variables are internal variables of the processor which performs the loading. The <hexnumber> which may follow the <nonhexletter> of their names represents the number of the related section. If the variable number is the same as the current section number, it may be omitted. If no SB command exists before, the variable is related to the absolute section.

12.2.1 L variables and U variables (Low and Upper limits)

<L variable name>::=L{<hexnumber>}

<U variable name>::=U{<hexnumber>}

A relocatable section is loaded within a region of the target memory. This region is characterized by two values which are its lowest and highest addresses. Ln and Un contain these values for the section n. Normally these variables are not used for absolute sections, but by convention they may contain the lowest and the highest address of the whole target memory.

As L and U variables are only known at loading time and remain constant, they cannot be the left part of an "assignment command". They are initialized by the loading processor when the section is entered for the first time.

12.2.2 R variables (Relocation bases)

<R variable name>::=R{<hexnumber>}

At assembly (or compiling) time, addresses pointing to a location in a relocatable section (which is generally the section currently being processed) can only be translated to an offset relative to a "section origin". At loading time, as the origin value will be known, the actual addresses will be calculated and loaded (program relocation). It is Rn which will contain the origin value for the section n. R variables are not used for absolute sections.

An R variable is initialized by the loading processor when the corresponding section is entered for the first time.

When the linkage editor joins several sections (from separately compiled modules) into a single one, it updates the R variable of the resulting section in such a way that the relocation offsets remain unchanged (see example 5). This is the reason why all addresses which are to be relocated must use, as relocation base, the R variable of the section into which they point. This is also the reason why CUFOM allows "assignment commands" with R name as left part but with the following syntax:

12.2.3 P variables (loading Pointer)

<P variable name>::=P{<hexnumber>}

The variable Pn contains the address of the target memory location where the next element (instruction, address or data) of the section n should be loaded. Thus it is automatically updated each time a new element is loaded. But it may be modified in using an "assignment command". As it is a pointer to a section location, the syntax of such a command must be:

ASPn=<absolute expression>.
if the section is absolute,
or
ASPn=Rn{,<absolute expression>,+}.
ASPn=Rn{,<absolute expression>,-}.
if the section is relocatable,
or
ASPn=Pn{,<absolute expression>,+}.
ASPn=Pn{,<absolute expression>,-}.
only to update Pn (independently of the section type)

where n is a <hexnumber> (or may be omitted).

When the first load command of a relocatable section is encountered and if the P variable was not initialized before by such an "assignment command", it is assumed that the value of the P variable is equal to the one of the R variable (ASP=R. is the default command when a new section is entered).

12.2.4 E variable (Execution starting address)

<E variable name>::=E

The E variable contains the address of the first instruction to be executed. This value is loaded in the program counter to run the program.

An E variable name can be the left part of an "assigment command". Usually such a command appears, at most, once in a module directly produced by a language translator. But after a linkage edition there may exist several of them. Logically it is the last assignment which decides the E value. Nevertheless the actual behaviour depends on the loading processor convention (target dependence). As the E variable need not be assigned in a module, the loading processor can provide either a warning message, a default value, or a parameter set by the user at loading time (e.g. an absolute address, an internal symbol name).

12.3 Variables related to the linkage edition and the loading

12.3.1 S variables (section Size)

<S variable name>::=S{<hexnumber>}

The S variables are internal variables of the linkage editor and of the loading processors. There is one S variable per relocatable section in an object module. The variable Sn contains the size of the section n (in target machine unit of storage). Assigning a size to an absolute section is meaningless.

As the S variable values may be used by the CUFOM processors, they must be calculated by the language translator and set by an "assignment command" with the following syntax:

ASSn=<absolute expression>.

where n is the section number (or may be omitted) and <absolute expression> must be evaluable at linking time.

In a module, there must exist one (and only one) such "assignment command" for each relocatable section.

The S variables are used by the linkage editor for:

joining several input G or R sections into a single one. If x is the address which will be assigned to the R variable of the resulting section and y an offset calculated at compiling (or assembly) time, the address of the corresponding element will be the sum of x,y and the sizes of all the previously joined sections. The size of the resulting section is the sum of the sizes of all the joined sections (see example 5).

overlapping B or C sections. The size of the resulting section is the size of the largest overlapped section. If the sizes are not all equal an appropriate message is printed.

12.4 Variables with a meaning dependent upon the target machine

The following variables can be internal variables of the loading processor. The language translator knows the general specification of the target machine (e.g. the instruction set, the address and number representation, the word length, etc.) but it may ignore some peculiarities of a given installation (e.g. the size of the memory, the addresses of I/O ports, the ROM location, etc.). To solve such a possible case, it can generate expressions containing K variable names. At loading time these K variable will have a constant value related to the target installation.

Finally, if the set of all the previous CUFOM variables is not sufficient, it is possible to give a precise meaning to W variables which are normally understood as working registers. Then, for example, it is possible to pass a parameter between separately compiled modules, using reserved W variable names. The only requirement is to link-edit the module which contains the assignment to such W variable before the others.

The linkage editor never replaces or deletes K or W variables names from the expressions that it reads. It never puts K or W variables in the new commands it generates.

12.4.1 K variables (constants)

<K variable name>::=K<hexnumber>

The meaning and the use of these variables are the responsability of the CUFOM user. But as they are intended to contain constant values, they cannot be the left part of an "assignment command".

12.4.2 W variables (Working registers)

<W variable name>::=W<hexnumber>

No special meaning is attached to these variables: in different parts of the same module, a W variable name may be used for different purposes (e.g. to store a value temporary).

13 THE GENERAL COMMAND

For some particular cases of loading on a specific target machine, it may happen that the previous commands are not appropriate. With the US (use) command, new commands can be defined to solve these particular problems.

<US command>::=US<identifier>{ (<parm list>) }

When such a command is encountered at loading time, the routine specified by <identifier> is applied, using the values of the <parm list> (if any). The definition of that routine is target dependent.

14 CHECKSUM

When object modules are sent for loading through data links which do not allow hardware checking (e.g. INDEX line), CUFOM may be authorized to add error detection information into the object module. In such a case, the "pusher" adds a checksum field or at the end of each line, or at the end of each command, either at the end of the module (in the "module end command"). The choice depends on a convention related to a target machine.

If present, the checksum field is between the full stop (or the continuation sign) and the "end of line" character. It contains a <hexnumber> which represents the value of the checksum. The way that this checksum is calculated depends again on a convention related to target machine. For example, it may be the value of the lower byte of the sum of the ASCII codes of each character in the line and preceding the full stop or the semicolon. As the word size may differ between the host and the target computer, the term of "sum" must be understood as "sum as calculated on the target machine".

Object modules with a checksum field are not accepted by the linkage editor which cannot have any special convention with a target machine. The checksum must be calculated afterwards by a "pusher" (pre-loading processing).

15 EXAMPLES

15.1 Example 1

The following object module is an example for absolute loading. The values represented in three first LD commands must be loaded in successive locations starting at the address 2560 (A00). The values represented in the last LD command must be loaded in successive locations starting at the address 2836 (B14). The execution must begin with the instruction loaded at the address 2568 (A08).

```
MBJB007.
ASP=A00.
LD03FA350564DC67FF0000000010002FB7A45702A2BE3759537F8D412007FA4.
LD4EAB6389110AAF372994140B2AF4C76E3000006lFE64A8B429D700540002.
LD437B00020AF38380.
ASP=B14.
LD4546474800000FF01AC3E.
ASE=A08.
ME.
```

15.2 Example 2

The following module is an example of loading with address relocation. There exists an absolute section (number Ø) and a relocatable section (number 1). The length of the relocatable section is 112 (in bytes in the case of the TEXAS INSTRUMENTS TMS9900 microprocessor). No space is reserved at the beginning of the program. Firstly 37 words (or 74 bytes) must be loaded in successive locations. Each loading operation ("L" or "R") is applied on a word (" bytes). Thus the words 1,7,11,14,2Ø,23,25,27 with the respective values (in hexadecimal): ØØ5Ø,ØØ4A,ØØ2Ø,ØØ2Ø,ØØ3C,ØØ3C,ØØ3C,ØØ3C must be relocated in adding to them the address of the word Ø. Next, in the absolute section, 2 words must be loaded respectively in the locations 2Ø and 22 (for interrupt handling). The location 22 must contain the address of the first word of the relocatable section. Finally after the 37 words previously stored (relocatable section), 3 new words must be loaded. The 32 last bytes are not initialized. To execute this program, the program counter must be set with the address of the first word of the relocatable section.

```
MBTMS9900,Ø7EXAMPLE.
STØ,A.
ST1,G.
SB1.
ASS=7Ø.
LDØ2EØØØ5ØØ3ØØØØØØØ2ØC1ØØØ2Ø2ØØ4AD2321103Ø6AØØØ2Ø1ØFBØ6AØ.
EXLRL5RL3R.
LDØØ2ØØ3401FØA13ØCC14BØ6AØØØ3C1EØA06AØØØ3CØ6AØØØ3CØ6AØØØ3C.
EXRL5RL2RLRLR.
LD1DØA04551FØD16FE32Ø81FØB16FE1DØBØ44B.
SBØ.
ASP=14.
LDØ1180000.
EXLR(R1).
SB1.
ASP=R,2Ø,+.
LD48454C4C4FA1.
ASE=R.
ME.
```

15.3 Example 3

The following module is another example of loading with relocation, but only one LD command and one EX command are used. Space for 32 memory units (e.g. bytes) is reserved at the beginning of the section.

```
MBB52,XWZ.
CO0,28****A NEW VERSION (ZWX) IS AVAILABLE****.
ST0,G.
SB0.
ASS=70.
ASP=R,20,+.
LD02E0005003000000020C0100020200 4AD2321103 06A0002010FB06A0:
0020034 01F0A130CC14B06A0003C1E0A06A0003C06A0003C06A0003C:
000045A736C3000000000000FFF20004D543E604D23A00000003A3B1:
1D0A04551F0D16FE32081F0B16FE1D0B044B48454C4C4FA1.
EXLRL3M(1;2E)RL3RL2RL5RL2RLRLRL3RLRL2RL3RL4RL2M(1;A2)L2R:
EXL2R.
ASE=R,20,+.
ME.
```

15.4 Example 4

The three following modules are an example of a linkage edition where external references are resolved.

The two following modules, listed in two columns, are the input.

In the first module, with name GOLD, the four external symbols: T1,T2,T3,P5 are referenced and the internal symbols: V1,V2,V3 are declared as entry names. The values of V1 and V2 are absolute and are equal respectively to 8 and 10 (A).The value of V3 is relative to the origin of the section 1 with a positive offset of 22 (16). We assume that the target machine has a byte addressing, a 16 bit word and an address length equal to the word length. Thus, for the absolute section, 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 10 (000A). For the relocatable section, the execute command expresses that the values which will be loaded in the relative address: 16 (10) onwards will be respectively:

* the value of T1,
* the value of T1 plus 4,
* the value of T1 plus 8,
* 19948 (4DEC),
* the value of T2 minus the value of T1 plus 10 (A),
* the value of P5.

```
MBJB007,04GOLD.
CO1,11"GOLD"(10 JUN 78).
ST0,A.
ST1,G.
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.
CO1,13"FINGER"(13 JUN 78).
ST0,A.
ST1,G.
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).
ASE=R,2,+.
ME.
```

The next module illustrates the manner in which the two previous modules are linked.

As this module may be linked with other ones in a further step, the NI commands are retained. The external symbol P5 was discovered unresolved. An appropriate message is printed on the MAP listing of the current linkage edition and a NX command is produced to allow a further linkage. In the execute commands the other external references are replaced by an equivalent expression. 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,+.". Thus in the execute commands, each call without parameter to the relocation operation (in this example: the R follwing L2 in the last EX command) are not to be replaced by a call with a parameter (which would be R(R,1C,+) in this example). The command "ASP=R." which is a command by default in the module B is now generated because the next LD command is no longer the first one of the section.

```
MBJBØØ7,ØAGOLDFINGER.
CO1,32"GOLDFINGER" LINKED ON IBM37Ø/168, DATE= 29 SEP 78.
CO2,11"GOLD"(1Ø JUN 78).
STØ,A.
ST1,G.
SB1.
NXØ,Ø2P5.
NIØ,Ø2V1.
NI1,Ø2V2.
NI2,Ø2V3.
ASIØ=8.
ASI1=A.
ASI2=R,16,+.
ASS=26.
SBØ.
ASP=8.
LDØØØA0506.
EXR(R1,24,+).
SB1.
ASP=R,1Ø,+.
LDØØØØØØ4ØØØ84DECØØØAØØØØ.
EXR(R,2Ø,+)3LR(R,1E,+,R,2Ø,+,-)R(XØ).
CO2,13"FINGER"(13 JUN 78).
NI3,Ø2T2.
NI4,Ø2T1.
NI5,Ø2T3.
ASR=R,1C,+.
ASI3=R,2,+.
ASI4=R,4,+.
ASI5=R,8,+.
ASP=R.
LDFFF4ØØ2D13ØCØØØ4Ø6AØ.
EXR(8)L2RR(R,6,-).
ASE=R,2,+.
ME.
```

The following figure shows the MAP listing which was produced during this linkage edition.

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

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


==1==>  "GOLD"(10 JUN 78)
         ABSOLUTE SECTION      0
         RELOCATABLE SECTION   1   SIZE=0026(   38)
           UNRESOLVED REFERENCE                                          P5
           DEFINED IN SECTION  0                     OFFSET=0008         V1
           DEFINED IN SECTION  0   (NOT REFERENCED)  OFFSET=000A         V2
           DEFINED IN SECTION  1                     OFFSET=0016         V3

==1==>  "FINGER"(13 JUN 78)
           DEFINED IN SECTION  1                     OFFSET=001E         T2
           DEFINED IN SECTION  1                     OFFSET=0020         T1
           DEFINED IN SECTION  1                     OFFSET=0024         T3


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


>>>>>>>  0  ERROR  DETECTED   <<<<<<<
```

15.5 Example 5

The three following modules are an example of a linkage edition where several sections are combined together. It concerns the FERRANTI F1ØØL microprocessor (16 bit words).

A program in store essentially consists of an area to hold the program instructions and two data areas: the lower and the upper areas. The lower area must lie within the addresses range 1-2Ø47, while the program and the upper data areas may fall above this range. This separation arises because the address field of a single word instruction is 11 bits wide. Instruction may refer to addresses higher than 2Ø47 by use of a second word (or by means of pointers having addresses in the range 1-255). The upper and lower divisions are further subdivided into nine "storage classes" which are (in the order of increasing addresses):

- Lower Data (LD): local variables.
- Lower Common Data (LCD): common variables.
- Lower Preset Data (LPD): local preset constants.
- Lower Common Preset Data (LCPD): common preset constants.
- Program Instructions (PR).
- Upper Preset Data (UPD).
- Upper Common Preset Data (UCPD).
- Upper Data (UD).
- Upper Common Data (UCD).

When program segments are combined to form a complete program, corresponding areas are merged together in the order of presentation to the linkage editor.

The following figures shows a source segment (SEGA) and its generated object module. The segment consists of 1 program block and 7 data blocks. The data blocks are of various kinds, thus the first declarations concern the Lower Common Preset Data area.

As the addresses may be represented on a 11 bit field (single word instruction) or on a 15 bit field (data or double word instruction), two relocation operations (EX commands) are distinguished: respectively S and R.

```
$SEGMENT  SEGA
$LOWER    COMMON/C1
(3 WORDS ARE RESERVED WITH PRESET VALUES: 1,2,3
L1-       1,2,3
$UPPER    COMMON/C2
L2-       71,72
$LOWER
(3 WORDS ARE RESERVED WITH NAMES: A,B,C
          A,B,C
$LOWER    COMMON/C1
L3-       4,5
$UPPER
L4-       A,B,C
$PROGRAM
          LDA   A
          STO   E
$LOWER
L5-       9,10
$UPPER    COMMON/C3
          D,E
$END
```

```
MBFER100L,04SEGA.
CO1,2F"SEGA" COMPILED ON IBM 370/168, DATE= 17 AUG 78.
ST0,R,02LD.
ST1,R,03LPD.
ST2,R,02PR.
ST3,R,03UPD.
ST4,R,02UD.
ST5,C,07LCPD/C1.
ST6,C,07UCPD/C2.
ST7,C,06UCD/C3.
ASS0=5.
ASS1=2.
ASS2=2.
ASS3=3.
ASS4=0.
ASS5=5.
ASS6=2.
ASS7=2.
SB5.
LD000100020003.
SB6.
LD00470048.
SB5.
LD00040005.
SB3.
LD00000010002.
EXR(R0)3.
SB2.
LD80004001.
EXS(R0)S(R7).
SB1.
LD0009000A.
ME.
```

The following figures shows another source segment (SEGB) and its generated object module.

```
$SEGMENT     SEGB
$UPPER       COMMON/C3
             F,G
$LOWER       COMMON/C1
L6-          41,42,43
$LOWER
             H,I
$PROGRAM
             LDA    G
             STO    L
$UPPER
L7-          6,7
$LOWER
L8-          11,12,13
$END
```

```
MBFER100L,04SEGB.
CO1,2F"SEGB" COMPILED ON IBM 370/168, DATE= 23 AUG 78.
ST0,R,02LD
ST1,R,03LPD.
ST2,R,02PR.
ST3,R,03UPD.
ST4,R,02UD.
ST5,C,06UCD/C3.
ST6,C,07LCPD/C1.
ASS0=2.
ASS1=3.
ASS2=2.
ASS3=2.
ASS4=0.
ASS5=2.
ASS6=3.
SB6.
LD00290002A002B.
SB2.
LD80014002.
EXS(R5)S(R7).
SB3.
LD00060007.
SB1.
LD000B000C000D.
ME.
```

The next figure illustrates the manner in which the various "storage classes" from the SEGA and SEGB segments are combined together by the linkage editor.

```
MBFER100L,04SEGC.
CO1,2C"SEGC" LINKED ON IBM370/168, DATE= 29 SEP 78.
CO2,2F"SEGA" COMPILED ON IBM 370/168, DATE= 17 AUG 78.
ST3,R,02LD.
ST4,R,03LPD.
ST5,R,02PR.
ST6,R,03UPD.
ST7,R,02UD.
ST8,C,07LCPD/C1.
ST9,C,07UCPD/C2.
STA,C,06UCD/C3.
ASS3=7.
ASS4=5.
ASS5=4.
ASS6=5.
ASS7=0.
ASS8=5.
ASS9=2.
ASSA=2.
SB8.
LD0001000020003.
SB9.
LD00470048.
SB8.
LD00040005.
SB6.
LD0000000010002.
EXR(R3)3.
SB5.
LD80004001.
EXS(R3)S(RA).
SB4.
LD0009000A.
CO2,2F"SEGB" COMPILED ON IBM 370/168, DATE= 23 AUG 78.
SB8.
ASP=R.
LD00290002A002B.
SB5.
ASR=R,2,+.
ASP=R.
LD80014002.
EXS(RA)S(R0).
SB6.
ASR=R,3,+.
ASP=R.
LD00060007.
SB4.
ASR=R,2,+.
ASP=R.
LD000B000C000D.
ME.
```

Finally the following figures shows the produced MAP listing.

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

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


==1==>  "SEGA" COMPILED ON IBM 370/168, DATE= 17 AUG 78
         RELOCATABLE SECTION    3   SIZE=0007(    7)   OFFSET=0000    LD
         RELOCATABLE SECTION    4   SIZE=0005(    5)   OFFSET=0000    LPD
         RELOCATABLE SECTION    5   SIZE=0004(    4)   OFFSET=0000    PR
         RELOCATABLE SECTION    6   SIZE=0005(    5)   OFFSET=0000    UPD
         RELOCATABLE SECTION    7   SIZE=0000(    0)   OFFSET=0000    UD
         COMMON SECTION         8   SIZE=0005(    5)   OFFSET=0000    LCPD/C1
         COMMON SECTION         9   SIZE=0002(    2)   OFFSET=0000    UCPD/C2
         COMMON SECTION        10   SIZE=0002(    2)   OFFSET=0000    UCD/C3

==1==>  "SEGB" COMPILED ON IBM 370/168, DATE= 23 AUG 78
>>> COMMON SECTION   8 : OLDSIZE=0003 (    3),NEW SIZE=0005 (    5) <<<


>>>>>>>  0  ERROR  DETECTED   <<<<<<<
```

INDEX