

Recognized as an American National Standard (ANSI)

IEEE Std 695-1990
Upgraded from trial use 1985)

IEEE Standard for Microprocessor Universal Format for Object Modules

Sponsor
**Microcomputers and Microprocessors Standards Subcommittee
of the
IEEE Computer Society**

Approved February 15, 1990
IEEE Standards Board

Approved July 24, 1990
American National Standards Institute

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

© 1990 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published in 1990. Printed in the United States of America

ISBN 1-559374-105-6

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE which have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least once every five years for revision or reaffirmation. When a document is more than five years old, and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
345 East 47th Street
New York, NY 10017
USA

Foreword

(This Foreword is not a part of IEEE Std 695, IEEE Trial-Use Standard for Microprocessor Universal Format for Object Modules.)

With the passage of time, microprocessor system capabilities are increasing, just as did those of the minicomputers of the previous generation and the mainframes before them. With larger memories come larger programs, and with them, the necessity of partitioning programs into smaller, more manageable units that are linked together into the final program. Thus arises the need for relocatable and linkable code modules.

Unlike their larger cousins, the minicomputers and mainframes, microprocessors are often used in multivendor situations where the same software must work with a variety of operating systems and configurations and usually where the original computer manufacturer does not hold a dominant sway over the software formats. The ability to relocate and link code from a variety of vendors is in the interest of the user, and to that end a standard object module format is desirable.

A second, perhaps even more compelling reason for a standard object module format arises from the fact that many users of microprocessors are not limited to one architecture or CPU. These users are compelled to support all of their target processors with all stages of the software translation process, even though many operations need not depend on the actual target CPU. With a standard format, only those functions that necessarily relate to the target machine architecture need specialization; linking and relocation of the object modules can be done for all processors with a single program.

MUFOM, the Microprocessor Universal Format for Object Modules, is designed to apply to a variety of target machines. It permits the linking and relocating functions to be isolated from the specification of target architecture, so that a user may employ a single program for these functions across all CPUs.

Processes

MUFOM is specified with the realization that an object module may undergo five processes in its life cycle and that these processes may be separate programs or may be combined into a single program.

The first process in the life of an object module is its creation by a compiler or assembler. Such a process almost always must be concerned with the architecture of the target machine and is not specified in this standard.

The second process is normally that of linking separately translated modules into one module and resolving external references. A partial linking may take several modules and produce a new module with still unresolved external references, which may require yet another linking. Linking may not be required for compilations that reference no external procedures.

The third process is relocation. This process assigns absolute memory addresses to module locations and adjusts address fields previously generated with address constants (or expressions) relative to the module entry points. Often relocation and linking are done simultaneously. The object module that comes out of the relocater is normally an absolute module with all address fields completely specified.

The fourth process evaluates loader expressions. These may be generated in the process of linking and relocating, or they may be initially generated by the source language translator. Usually, it is only after relocation that all necessary operands for evaluation are available. The result of the evaluation is a memory image that only needs loading. Aside from considerations such as word size and back links, expression evaluation does not depend on the target machine.

The fifth process is the loading of the absolute module into memory in the target machine. This may be specific to the word size of the target machine. Other than that dependency, loading may be done independently of the CPU architecture. Often the loading function is included with relocation and expression evaluation. For purposes of consideration in this standard, a translator that converts the MUFOM format absolute code module into a target-

machine-dependent format is considered to be doing the loading functions. The loading process marks the end of the life cycle of the object module.

MUFOM is defined consistently across all five processes so that a load module is in a format that is a proper subset of the format for an absolute module, which in turn is a proper subset of the format for a relocatable module, which similarly is a proper subset of the format for a linkable module. Thus, a single syntax defines all three formats, with the simple qualification that there are particular commands that are destined to be swallowed by the linker and that there are other commands for the relocater and still others for the loader.

Portability

MUFOM is based on CUFOM, the CERN Universal Format for Object Modules. MUFOM is designed for maximum portability of object modules between conforming systems. For this reason, all commands are specified as ASCII text derived from a minimum character set. For internal use an equivalent binary format is encouraged, and Appendix B gives an example of a binary implementation.

Many existing object module formats were considered to ensure their ease of translation into MUFOM in a direct way. In general, a MUFOM command represents a single function. Portability of MUFOM modules to other formats was not considered to be of sufficient value or feasibility to warrant serious attention.

At the time that this standard was approved for trial use the members of the Working Group on Microprocessor Relocatable Object Code Format were as follows:

Gregg Marshall, *Chair*

Geoff Baldwin
Richard James, III

Jean Montuelle
Tom Pittmann

Stephen Savitsky
Ian Willers

When the IEEE Standards Board approved this standard on February 15, 1990, it had the following membership:

Marco W. Migliaro, *Chair*

James M. Daly, *Vice Chair*

Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Fletcher J. Buckley
Allen L. Clapp
Stephen R. Dillon
Donald C. Fleckenstein
Jay Forster*

Thomas L. Hannan
Kenneth D. Hendrix
John W. Horch
Joseph L. Koepfinger*
Michael A. Lawler
Donald J. Loughry
John E. May, Jr.
Lawrence V. McCall

L. Bruce McClung
Donald T. Michael*
Stig Nilsson
Roy T. Oishi
Gary S. Robinson
Terrance R. Whittemore
Donald W. Zipse

*Member Emeritus

CLAUSE	PAGE
1. Scope	1
2. Definitions.....	1
3. References.....	3
4. Introduction	3
4.1 Commands	3
4.2 Free-Form Syntax	3
5. Basic Constructions.....	4
5.1 Notation Used for Syntax Definitions.....	4
5.2 Elementary Syntactic Objects	4
6. Expressions	5
6.1 Types of Operands	5
6.2 Functions With No Operands.....	5
6.3 Monadic Operators and Functions	6
6.4 Dyadic Operators and Functions.....	6
6.5 Data Manipulation Operations	6
6.6 Other Operations	7
7. Variables	7
7.1 G-Variable (execution starting address)	8
7.2 I-Variables (external definition—symbol values).....	8
7.3 L-Variables (low limit)	8
7.4 N-Variables (names)	8
7.5 P-Variables (load pointer).....	8
7.6 R-Variables (relocation reference).....	8
7.7 S-Variables (section size).....	9
7.8 W-Variables (working register)	9
7.9 X-Variables (external symbol reference).....	9
8. Module-Level Commands.....	9
8.1 MB (module begin) Command	9
8.2 ME (module end) Command.....	10
8.3 DT (date and time of creation) Command	10
8.4 AD (address descriptor) Command.....	10
9. Comments and Checksum.....	11
9.1 Comments	11
9.2 CS (checksum) Command	11
10. Sectioning.....	12
10.1 SB (section begin) Command	12
10.2 ST (section type) Command	12

CLAUSE	PAGE
10.3 SA (section alignment) Command	14
11. Symbolic Name and Type Declaration	14
11.1 NI (name of internal symbol) Command	14
11.2 NX (name of external symbol) Command	15
11.3 NN (name) Command	15
11.4 AT (attributes) Command	15
11.5 TY (type) Command	16
12. AS (assignment) Command	16
13. Loading Commands	16
13.1 LD (load) Command	17
13.2 IR (initialize relocation base) Command	17
13.3 LR (load relocate) Command	17
13.4 RE (replicate) Command	18
14. Linkage Edition	18
14.1 RI (retain internal symbol) Command	18
14.2 WX (weak external symbol) Command	19
15. Libraries	19
15.1 LI (Specify Default Library Search List) Command	19
15.2 LX (library external) Command	19
16. MUFOM Kernel	19
17. Binary Format	20
18. Bibliography	20
Annex A (informative) Collected Syntax	21
Annex B (informative) Suggested Binary Encoding	25
Annex C (informative) Command and Variable Usage by Processes	27

IEEE Standard for Microprocessor Universal Format for Object Modules

1. Scope

This standard specifies the format of linkable, relocatable, and absolute object modules for binary computers of arbitrary word size and architecture. Two levels of compliance are specified, minimum and full.

The minimum compliance level affords sufficient flexibility to link separately compiled modules, to relocate addresses in simple ways, and to load the resulting absolute object modules with a minimal loader.

The full compliance level affords all of the functionality of the minimum level and adds to it arbitrary address expression handling, type checking capability, librarian control commands, and other useful functions for full generality.

A conforming implementation may extend the command or function set of MUFOM for greater efficiency in dealing with machine-specific requirements, but object modules containing such extensions shall not be said to be conforming to this standard. Such extensions are not specified in this standard.

This standard also does not specify

- 1) The source language features to be supported by way of MUFOM modules
- 2) The structure of relocatable code libraries
- 3) The format of the optional binary equivalent
- 4) Algorithms for linking, relocating, and loading
- 5) The architecture of the host machine
- 6) The architecture of the target machine, including word size, size of memory, memory organization, or instruction format.

This standard is derived from documents in the public domain and contains no patented or proprietary material.

2. Definitions

The following definitions give the meanings of the technical words as they are used in this standard and are intended to reflect the accepted use of the defined words.

absolute code: Data or executable machine code in memory or an image thereof. Distinguish from **relocatable code**.

absolute loader: A process which can load one or more sections of **absolute code** only at the locations specified by the **sections**. See: **relocator**.

binary information: Bit patterns to be loaded into memory.

character form: The (printable) character representation of **binary information** as opposed to the bit pattern representation.

checksum: A deterministic function of a file's contents. If a file is copied and the **checksum** of the copy is different from the original, there has been an error in copying.

code: Data or executable machine code. See: **absolute code** and **relocatable code**.

command: Control information for a **linker**, **relocator** or **loader**. Distinguish from **code**.

comments: Information that is readable by people. To be distinguished from **binary information** and **character form**.

external definition: The definition, within a **section**, of a symbol which is referenced by other **sections**. An exported global symbol.

external reference: The specification, within a **section**, of a symbol which is defined outside that **section**. An imported global symbol.

format: The language in which **object modules** are specified.

librarian: The **process** which performs operations, such as maintenance, upon a **library**.

library: A set of **object modules**.

linkage editor: A **process** which combines **object modules** into a single **object module** satisfying links between the **object modules**.

linker: Same as **linkage editor**.

load pointer: A pointer for a **section** which is dynamically maintained by the **loader**. It indicates where the next item of **code** is to be loaded. It is initialized to a **starting load address**.

loader: Same as **absolute loader**.

MAU: Abbreviation for **minimum addressable unit**.

minimum addressable unit: For a given **processor**, the amount of memory located between an address and the next address. It is not necessarily equivalent to a word or a byte. Abbreviated **MAU**.

module: Same as **object module**.

MUFOM: Acronym for Microprocessor Universal Format for Object Modules.

object module: A set of **sections** of **absolute code** or **relocatable code**, together with **commands**.

process: A program which is executed by a **processor**. In this document the denotation is restricted to **linker**, **relocator**, expression evaluator, and **loader**.

processor: A computing machine, albeit perhaps a virtual machine.

program: A set of data and instructions which defines the operations to be performed by a **processor**.

relocatable code: Position-independent code which can be loaded at an arbitrary memory location. It generally requires both **relocation** and expression evaluation to become **absolute code**.

relocator: A process which assigns absolute addresses to a **section** of **relocatable code** to produce **absolute code**.

relocation: The function of the **relocator**.

replication: The loading of repetitive data by means of an abbreviated specification.

section: A part of a **program** with ancillary information (**commands**) which becomes a segment when loaded.

segment: A contiguous region in memory with arbitrary boundaries.

shall: Is required to ... , are required to

should: Is advised to ... , are advised to

starting load address: The address at which the loading of a **section** begins.

type: An attribute of a symbol, value, or **section**. The definition of a symbol type is left to the implementer.

weak external: An **external reference** which need not be satisfied during linking.

3. References

[1] ANSI X3.4-1977, American National Standard Code for Information Interchange.¹

[2] ANSI X3.43-1977, American National Standard Representations of Local Time of the Day for Information Interchange.

[3] ISO 3307-1975, Representations of Time of the Day—Information Interchange.²

4. Introduction

4.1 Commands

A MUFOM object module is a sequence of commands. These commands are introduced by a two-letter mnemonic and are terminated by a period. Thus,

```
MBI8080.
ASP,100.
LDC30001.
ME.
```

is a well-formed module. Detailed descriptions of the commands in this example are contained in following sections, but an informal paraphrase of this example is Module begin “Intel 8080.”

Assign load pointer value to be 256 (decimal).

Load three bytes with the binary information represented by the hexadecimal specification “C3,” “00,” and “01.”

Module end.

4.2 Free-Form Syntax

An object module is treated as one contiguous string of ASCII characters according to ANSI X3.4-1977 [1].³ All ASCII control characters (such as a line feed or a carriage return) are ignored in processing.

¹ANSI publications are available from Sales Department, The American National Standards Institute, 1430 Broadway, New York, NY 10018.

²ISO publications are available from Sales Department, The American National Standards Institute, 1430 Broadway, New York, NY 10018.

³The numbers in brackets correspond to the references listed in Section 3.

The length of a MUFOM command is variable and there is no limitation on that length. To represent a MUFOM object module as a text file on a host processor (with limited line size), a single command can occupy several successive lines (for example, encompass several line feeds, carriage returns, or other control characters).

5. Basic Constructions

5.1 Notation Used for Syntax Definitions

The metalanguage for the syntax definitions is derived from the Backus-Naur form and from regular expression notation.

Terminal symbols (literal characters shown exactly as used) are enclosed in quotation marks; nonterminal symbols (names of syntactic forms) are (possibly hyphenated) words without quotation marks. Parentheses are used to show grouping. Spaces are not significant. Concatenation is indicated by the juxtaposition of symbols. Repetition is indicated by recursion or by the two metasymbols + or *

item * = zero or more occurrences of item

item + = one or more occurrences of item

Alternation is indicated by the metasymbol | item 1 | item 2 either item 1 or item 2 not both Electionis indicated by the metasymbol ? , item? either item or nothing

EXAMPLE:

variable → letter (letter | digit)*

5.2 Elementary Syntactic Objects

MUFOM uses a small number of data types to provide standard means of expressing constructs. They are as follows:

digit → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

hexletter → "A" | "B" | "C" | "D" | "E" | "F"

hexletter → "A" | "B" | "C" | "D" | "E" | "F"

hexdigit → digit | hexletter

hexnumber → hexdigit +

nonhexletter → "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z"

letter → hexletter | nonhexletter

alphanum → letter | digit

identifier → letter alphanum*

character → any ASCII graphic character

char-string-length $\rightarrow$ hexdigit hexdigit

char-string $\rightarrow$ char-sizing-length character*

where the hexadecimal value of string-length shall be equal to the number of characters, which shall be less than 128, for example, 14A STRING WITH SPACES,

MUFOM-variable $\rightarrow$ nonhexletter hexnumber?

a name which references an internal variable of a MUFOM process (see Section 7).

6. Expressions

Expressions are written in postfix Polish notation. The binary representation of each operand shall not exceed 64 bits. An implementation shall accept operands of at least 15 bits.

expression $\rightarrow$ hexnumber | MUFOM-variable | polish-expr | conditional-expr

polish-expr $\rightarrow$ (operand “,”)* function

operand $\rightarrow$ expression

function $\rightarrow$ “@” identifier | operator

operator $\rightarrow$ “+” | “-” | “/” | “*” | “<” | “>” | “=” | “#” | |

When a function or an operator appears in an expression, it specifies the computation to be performed on the values given in any preceding expression or expressions according to the function definition. Some standard functions are provided with MUFOM. However, it is expected that the implementer will enhance this set in order to more easily accommodate his target machine.

6.1 Types of Operands

The result of an operation performed on operands depends on the types and values of those operands. If the type or range of an operand is not allowed by the operation applied to it, the result is undefined. In MUFOM, only two types of operands are distinguished: logical and integer. Integer operands can be positive or negative. Logical operands can have only the two values TRUE and FALSE.

The type of an operand written as ‘hexnumber’ is integer, and its value is equal to that of the ‘hexnumber’ considered as a positive hexadecimal number. The type of an operand written as ‘MUFOM-variable’ depends on the last assignment to the corresponding variable. The type of an operand written as ‘polish-expr’ will be defined below. The type of an operand written as ‘conditional-expr’ is the type of the operand for which the condition is true.

For bit-wise logical operations (@AND, @OR, @XOR, @INS, @EXT), a MUFOM integer is represented by a binary number aligned on the right, zero-filled or truncated on the left, and considered as unsigned.

6.2 Functions With No Operands

polish-expr $\rightarrow$ “@F”

polish-expr $\rightarrow$ “@T”