A universal format for object modules, designed to apply to a variety of microprocessors, permits linking and relocating to be isolated from the specification of target architecture.

The Microprocessor Universal Format for Object Modules

Proposed Standard

IEEE P695 Working Group

This standard specifies a universal format for object modules designed to apply to a variety of microprocessors, permitting the linking and relocating functions to be isolated from the specification of target architecture. For portability, object modules are represented as sequences of ASCII characters, although a binary representation is permitted. A uniform command structure is specified to support linkage edition, relocation, expression evaluation, and loading.

Foreword

This forward is not part of P695, Draft 3.1, 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.

0272-1732/83/0800-0048$01.00 © 1983 IEEE

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 even 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 is done at the same time as linking. The object module that comes out of the relocator 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 pur-

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

Benefits of a universal format for object code

Users of a standardized, universal format for object modules reap the benefits of a wide variety of powerful software tools that are not restricted to a single target computer. The proposed standard object module format differs substantially from normal manufacturers' formats since it is independent of any particular computer or operating system. This is especially important in cross-software development, where a multitude of incompatible formats has restricted progress.

The proposed standard format affects the whole range of program development. The object module output stage of compilers and assemblers is standardized, the linkage editor is standardized, the relocator—which embodies different types of relocation—is standardized, and the loader is standardized. The developer of a new compiler or assembler can concentrate his efforts on producing the necessary bit patterns that form the object code of the target machine. He is relieved of having to learn a different incompatible format for a new machine. Furthermore, he can develop and use his software on any available host computer that supports the standard.

In the worst case, when no loader for the standard format is available, the loading stage may be preceded by a format conversion which produces an absolute load module in the target machine's own absolute format. Since the information contained in an absolute module is restricted to addresses and bit patterns that are to be loaded at those addresses, this format conversion is a simple one-to-one transformation. It should be recognized that separate compilation (including type checking), relocation, and library facilities are available even when the target format does not support them. Also, the software developer is required to learn only the simplest subset of the target format.

Once the standard format's processors and library routines are established, the work required to produce a standard assembler for the standard format is roughly equivalent to that required to produce an absolute assembler for the target format.

Other advantages of this approach should also be taken into account. For example, there exist today microprocessors which are packaged by their manufacturers or by software houses, or which are targets for particular PROM programmers. Each of these microprocessors or programmers requires a different object module format. These differing requirements are due to the manufacturers' copyrights and to the generality of the PROM programmers, which accept different types of memory chips. In the best case, the microprocessors and programmers will convert to the proposed standard format, hence removing duplication of effort; in the worst case, they will be covered by a few format converters.

The proposed standard format will encourage the development of additional universal utilities such as program analyzers, symbolic debuggers, object module cross-referencers, and type checkers. The competition to produce high-quality cross-software products will only benefit the marketplace. The advent of powerful tools for library maintenance, separate compilation, and relocation will enhance the productivity of the microprocessor programmer.

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

poses 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 up 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.

Committee list. The following persons are responsible for contributions to this document. Comments and suggestions are welcomed and should be addressed to the chairman of the committee:

Tom Pittman
Itty Bitty Computers
PO Box 6539
San Jose, CA 95150

Committee members:

Geoff Baldwin
Richard James III
Gregg Marshall
Jean Montuelle
Tom Pittman
Stephen Savitzky
Ian Willers

Table of contents

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

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 via 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, or
(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 and reference documents

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.

2.1 Definitions

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 is 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.

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

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.

2.2 Reference documents

(1) Christopher W. Fraser and David R. Hanson, "A Machine-Independent Linker," *Software—Practice & Experience,* Vol. 12, 1982, pp. 351-366.

(2) Leon Presser and John R. White, "Linkers and Loaders," *Computing Surveys,* Vol. 4, No. 3, Sept. 1972, pp. 149-157.

(3) Jean Montuelle, "CUFOM: The CERN Universal Format for Object Modules," Report No. DD/78/21, CERN Data Handling Division, Geneva, Switzerland, Oct. 1978.

(4) J. Montuelle and I. M. Willers, "Cross Software Using a Universal Object Module Format, CUFOM," *Proc. Euro IFIP 79—European Conf. on Applied Information Technology,* P. A. Samet, ed., North-Holland Pub. Co., Amsterdam, pp. 627-632.

3. Introduction

3.1 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 might be

```
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.
```

3.2 Free-form syntax

An object module is treated as one contiguous string of ASCII characters (reference ANSI X3.4—1968). All ASCII control characters (such as a line feed or a carriage return) are ignored in processing.

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 (e.g., encompass several line feeds, carriage returns, or other control characters).

4. Basic constructions

4.1 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 |

item1 | item2 either item1 or item2 not both

Election is indicated by the metasymbol ?

item? either item or nothing

Example:

variable → letter (letter | digit)*

4.2 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"

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 → hexdigit hexdigit

char-string → char-string-length character*

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

MUFOM-variable → nonhexletter hexnumber?

a name which references an internal variable of a MUFOM process (see Section 6).

5. 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 → hexnumber | MUFOM-variable |
polish-expr | conditional-expr

polish-expr → (operand ",")* function

operand → expression

function → "@" identifier | operator

operator → "+" | "−" | "/" | "*" |
"<" | ">" | "=" | "#"

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.

5.1 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.

5.2. Functions with no operands

polish-expr → "@F"
polish-expr → "@T"

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

@F, @T—These functions return the logical values FALSE and TRUE. They have no operands.

5.3 Monadic operators and functions

polish-expr → operand "," monad-f

monad-f → "@ABS" | "@NEG" | "@NOT" | "@ISDEF"

@ABS—The type of 'operand' must be integer. This function returns an integer which is the absolute value of 'operand.'

@NEG—The type of 'operand' must be integer. This function returns an integer which is 'operand' arithmetically negated.

@NOT—If the type of 'operand' is logical, the value returned is its logical complement. If the type of 'operand' is integer, the value returned is the one's complement of its value.

@ISDEF—This function returns the logical value TRUE if 'operand' contains no unassigned variables, and returns FALSE otherwise.

5.4 Dyadic operators and functions

polish-expr → operand1 "," operand 2 "," dyad-f

dyad-f → "+" | "−" | "/" | "*" | "@MAX" | "@MIN" | "@MOD" | "<" | ">" | "=" | "#" | "@AND" | "@OR" | "@XOR" |

For noncommutative functions, the conventional operand ordering is used. For example, in the case of division, the result is the value of operand1 divided by the value of operand2.

+, −, /, *—These perform the obvious arithmetic operations on two operands, which must be integer. The result is integer. Divide truncates toward zero, and its result is undefined if operand2 is zero.

@MAX, @MIN—These functions perform an arithmetic comparison between two operands, which must be integer. They return an integer which is the greater, or lesser, of the two operands.

@MOD—This takes two integer operands and returns the integer value operand1 modulo operand2. The result is undefined if either operand is negative, or if operand2 is zero.

<, >, =, #—These are relational operators for integer values. The operator '#' tests inequality. They return TRUE or FALSE.

@AND, @OR, @XOR—These are logical operations on the bit string representations of the values of the two operands or on the logical values TRUE or FALSE. If the operands are bit strings, a bit string is returned; if they are logical values, then TRUE or FALSE is returned.

5.5 Data manipulation operations

polish-expr → operand1 "," operand2 "," operand 3 "," "@EXT"

@EXT—This function extracts a bit string. The operands must be non-negative integers. Operand1 is the value from which the bit string is to be extracted. Operand2 and operand3 are the integer values of the starting and ending bit positions relative to the right end of operand1. The value is returned right-justified with binary zero fill. The least significant bit is numbered zero.

polish-expr → operand1 "," operand2 "," operand3 "," operand4 "," "@INS"

@INS—This function inserts a bit string. The operands must be non-negative integers. The value of operand2 is inserted inside operand1. Operand3 and operand4 are the integer values of starting and ending bit positions relative to the right end of the value of operand1. The value of operand2 is treated as a bit string and replaces the value of operand1 between these starting and ending bit positions, inclusive. The value of the function is the resultant packed bit string. Truncation or binary zero fill is on the left. The least significant bit is numbered zero.

5.6 Other operations

polish-expr → value "," condition "," errornum "," "@ERR"

value → expression

condition → expression

errornum → expression

@ERR—If condition is TRUE, then an error message is generated, with errornum becoming the error number. This function returns 'value.'

e.g., to emit external #3 into one byte and generate error #25 on overflow:

```
LR(X3,X3,FF,>,25,@ERR,1).
```

conditional-expr → condition "," "@IF" "," expression1 "," "@ELSE" "," expression2 "," "@END"

condition → expression

@IF—If the value of condition is TRUE, then expression1 is evaluated and its value and type become

the value and type of the function; expression2 is not evaluated. If the value of 'condition' is FALSE, then expression1 is not evaluated; expression2 is evaluated and its value and type become the value and type of the function.

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

6. Variables

Some internal variables of MUFOM processes (e.g., load pointer or symbol dictionary items) shall be manipulated according to operations expressed in the object module. Such variables are called MUFOM variables and their symbolic names may appear in expressions.

The internal variables of MUFOM are identified by a name with the following syntax:

MUFOM-variable → nonhexletter hexnumber?

The nonhexletter gives the class of the variable. If several variables of the same class exist, a hexnumber is required. If this hexnumber refers to the current section number, it may be omitted.

Except for the X-variable, the variables can appear in the left part of an AS-command (see Section 11).

6.1 G-variable (execution starting address)

G-variable → "G"

The G-variable contains the address of the first instruction to be executed. This value is to be loaded into the program counter to run the program. The implementer shall define the results of not assigning to a G-variable, or assigning to it more than once.

6.2 I-variables (external definition—symbol values)

I-variable → "I" hexnumber

The I-variable is assigned a value which is to be made available to other modules for the purpose of linkage edition. That value is also available in the module in which it is defined by reference to the I-variable. The value is associated with the external definition specified in the corresponding NI-command (see Section 10.1). The effect of more than one assignment to an I-variable is undefined.

6.3 L-variables (low limit)

L-variable → "L" section-number?

section-number → hexnumber

A section is loaded within a contiguous region of the target memory. The bounds of a section are given by its lowest address (L-variable) and its size (S-variable). The L-variable contains the value of the low address for the section specified by the section-number. There shall not be an assignment to an L-variable in a relocatable MUFOM section.

6.4 N-variables (name)

N-variable → "N" hexnumber

The N-variables are used for type checking in the TY-command (see Section 10.5). An N-variable refers to the name of a user's variable and is defined by the NN-command (see Section 10.3).

6.5 P-variables (load pointer)

P-variable → "P" section-number?

section-number → hexnumber

The P-variable of the section specified by section-number contains the address of the target memory location where the next element of binary information is to be loaded. The P-variable is automatically incremented whenever a new element is loaded. It may also be modified using an AS-command (see Section 11).

The initial value of the P-variable is equal to the value of the L-variable (low limit) for that section, which is the starting load address.

6.6 R-variables (relocation reference)

R-variable → "R" section-number?

section-number → hexnumber

The R-variable is a reference point within a relocatable section. Addresses within the section may be specified relative to the R-variable. The initial value of the R-variable is equal to the value of the L-variable for that section.

When the linkage editor joins several sections into one, it may generate assignments to the R-variable of the resulting section in such a way that the relocation offsets remain unchanged. Addresses which are to be relocated will in general use, as relocation points, the R-variable of the section.

6.7 S-variables (section size)

S-variable → "S" section-number?

section-number → hexnumber

There is one S-variable for each section in an object module. The S-variable contains the size (in target machine MAUs) of the section given by section-number. The effect of assigning to any S-variable more than once is undefined. The L-variable and S-variable of an absolute section give bounds to that section. Relocatable sections shall be relocated outside such bounded regions.

6.8 W-variables (working register)

W-variable → "W" hexnumber

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

6.9 X-variables (external symbol reference)

X-variable → "X" hexnumber

The X-variables correspond to external references. They are mapped onto the I-variables of other modules. When an NX-command is encountered, it is compared with external definitions given by NI-commands (see Section 10.1). If an external definition is found, the external reference is resolved by setting the value of the X-variable to the value of the corresponding I-variable.

By definition, X-variables cannot be the left part of an AS-command. They occur mainly as parameters for relocation operations. The implementer shall specify the action of the loader if it should encounter an X-variable.

7. Module-level commands

7.1 MB (module begin) command

MB-command → "MB" target-machine-configuration
("," module-name)? "."

target-machine-configuration → identifier

module-name → char-string

e.g., MBI8080,07MONITOR.

The MB-command shall be the first command of an object module. The target-machine-configuration may define the target-machine's name, CPU type, and operating system. The module-name should appear in the load map.

7.2 ME (module end) command

ME-command → "ME."

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

7.3 DT (date and time of creation) command

DT-command → "DT" digit* "."

To specify the date and time of creation of the output of a process, the date and time record may be used. The first part is the year (4 digits), followed by month, day, hour, minute, second (each 2 digits), and the fraction of a second (any number of digits). Only the more significant portions of the date and time need be given; for example, on a machine that has only a date:

DT19810723.

This form approximates the standards ANSI X3.43-1977 and ISO 3307-1975. The absence of a period before the fraction of seconds is forced by the MUFOM convention of terminating a command by a period.

7.4 AD (address descriptor) command

AD-command → "AD" bits-per-MAU
("," MAUs-per-address
("," order)?)? "."

bits-per-MAU → hexnumber

MAUs-per-address → hexnumber

order → "L" | "M"

e.g., AD8,2,L. CO,05 8080.
AD24,1. CO,05 7094.

(See Section 8.1 for a description of the "CO" command.)

In order to facilitate the use of machine-independent processes, the AD-command is used to specify address formats, the maximum size of an address, and the number of bits in a minimum addressable unit (MAU). The AD-command overrides implementer-defined default values.

Bits-per-MAU specifies the number of bits in a MAU.

MAUs-per-address specifies the maximum number of MAUs in an address to be relocated. If the MAUs-per-address is omitted, it is assumed to be one. A standard relocator or loader need not accept any module for which the product of bits-per-MAU and MAUs-per-address in the AD-command exceeds (decimal) 64.

The order field of the AD-command need only be specified when the maximum address size exceeds one MAU and the order is other than the default value M.

M specifies that the most significant MAU occupies the lowest address in the target machine; L specifies that the

least significant MAU occupies the lowest address in the target machine.

An implementer-defined order code may specify another possible ordering. The AD-command has no effect on the linker.

8. Comments and checksum

8.1 Comments

MUFOM 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, a process may generate a load map describing the overall structure of a module (which can be complex if it is the result of multiple linkage editions). To facilitate comprehension, MUFOM comments may be copied into the load map.

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

CO-command → "CO" comment-level? "," comment-text "."

comment-level → hexnumber

comment-text → char-string

e.g., CO3,1AI/O CONTROLLER (15 JUNE 81).

The comment-text in a CO-command may be used by a process to pass information to the user. The comment-level may be used by the process to select or exclude the comment from transmission or output. The interpretation of the comment-level shall be implementer-defined. A process may discard a CO-command or it may copy the CO-command, changing only the comment-level, if desired.

8.2 CS (checksum) command

CS-command → "CS" checksum? "."

checksum → hexdigit hexdigit

The checksum is formed by the unsigned binary adding of the binary values of successive ASCII characters (control characters are not added) together (modulo 128). The last byte to be added is the 'S' of the CS-command. The checksum should agree with (be equal to) the integer value of hexdigit hexdigit above. The running checksum is reset to zero both at the start of a module and after encountering the period terminating a CS-command. A simple "CS." resets the running checksum to zero without checking it. Checksum commands may be inserted anywhere any other command might be expected.

9. 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 in accordance with 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 or by an ME-command.

9.1 SB (section begin) command

SB-command → "SB" section-number "."

section-number → hexnumber

e.g., SB5.

The implementer shall specify the maximum possible section-number. All section numbers between zero and the maximum (inclusive) shall be valid.

Until an SB-command is encountered, the section number is zero.

9.2 ST (section type) command

ST-command → "ST" section-number ("," section-type)* ("," section-name)? "."

section-number → hexnumber

section-type → identifier

section-name → char-string

Examples of section types:

ST0,R,U,04SUBR.	CO,0D code section.
ST1,S.	CO,0D unnamed data.
ST23,E,03COM.	CO,0D named common.
ST1F,M.	CO,0D blank common.
ST4,Z,S.	CO,12 data in zero page.

In an ST-command, section-type defines the type of the section identified by section-number. If present, section-name gives a symbolic name to that section.

A section can be absolute or relocatable. A section is absolute when its loading addresses are absolute (known at assembly or compile time). A section is relocatable

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

when its loading addresses are relative to a relocation base whose value is initially not known.

Relocatable sections may be individually loaded into separate storage areas. However, in some cases, the loading addresses of sections shall be related. During the loading process, the user may have full control of the storage layout if the loader allows: for example, by using overlay directives. Except in special cases (e.g., overlays), MUFOM allows interdependency between sections of different modules. Thus, during the linkage edition, the module structure is built to correspond to the structure of the source programs.

For the linkage editor, it is possible to express three mutually exclusive kinds of relations among sections:

- Several sections are to be joined into a single one.
- Several sections are to be overlapped.
- Sections are not to coexist.

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

MUFOM categorizes sections in four ways. These provide the relocator with the information needed to lay out sections appropriately. The codes and/or categories may be extended for target machines where the standard is not sufficient. It may be meaningful to specify more, or less, than one attribute from each category. Some combinations of attributes are not meaningful.

Access. This attribute controls how the relocator groups sections together and lays them out in memory.

Writable (RAM, code W). This is the default in an ST-command if no access attribute is specified.

Read only (ROM, code R).

Execute only (code X).

Zero page (code Z).

Abs (code A). There shall be an assignment to the L-variable.

Named. This property is derived from the presence of the name.

Overlap. This attribute controls what to do with two sections if they have the same name and access attribute. (Two unnamed sections are said to have the same name if they have the same access attribute.)

Equal (code E). Error if lengths differ.

Max (code M). Use largest length encountered (e.g., blank common).

Unique (code U). Names should be unique (e.g., code).

Cumulative (code C). Concatenate sections together. The resulting section shall be aligned in such a way as to preserve the alignments of its components (see Section 9.3).

Separate (code S). No connection between sections. Multiple sections can have the same name, and the relocator may allocate them at unrelated places in memory.

When to allocate.

Now (code N). This is the normal case. This is the default if none is specified.

Postpone (code P). Relocator must allocate after all 'now' sections, thereby providing a way of getting "the last address allocated."

In an object module, if the ST-command is present, it may precede the corresponding SB-command. If no ST-command is given for a section, its type is absolute (A).

9.3 SA (section alignment) command

SA-command → "SA" section-number "," MAU-boundary? ("," page-size)? "."

MAU-boundary → expression

page-size → expression

Examples:

Start section 1 on a 2-MAU boundary:

```
SA1,2.
```

Force section 2 to reside in a 256-MAU page:

```
SA2,,100.
```

When a section is required to start on a boundary which is some integral multiple of more than one MAU, the SA-command is used.

If page size is supplied, the section may not be allocated across a page boundary.

10. Symbolic name and type declaration

Some symbolic names which appear in source programs must be retained in the object modules for further processing (e.g., linkage edition). In MUFOM, each symbol is declared by a separate command. Such commands begin with the character "N" followed by a letter which determines the class of that name. These commands are employed in the resolution of external references and in the specification of attributes associated with names.

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

10.1 NI (name of internal symbol) command

NI-command → "N" I-variable "," ext-def-name "."

I-variable → "I" hexnumber

ext-def-name → char-string

e.g., NIA,04SINE.

An NI-command shall be provided for each symbol which is defined in the current module and which may be referenced from another module. This is called external definition. It indicates that in the current module the I-variable (see Section 6.2) is associated with the external definition of ext-def-name. The effect of more than one NI-command in the same linkage edition with the same ext-def-name is undefined. An NI-command shall always come before all occurrences of its corresponding I-variable.

If the module produced by the linkage editor may be linked in a further step, the NI-commands are retained unless the implementer gives an option to the linkage editor to suppress them (see Section 13).

10.2 NX (name of external symbol) command

NX-command → "N" X-variable "," ext-ref-name "."

X-variable → "X" hexnumber

ext-ref-name → char-string

e.g., NX11,04SINE.

An NX-command shall be provided for each external symbol which is referenced in the current module. It indicates that in the current module the X-variable (see Section 6.9) is associated with the external reference of ext-ref-name. An NX-command shall always precede any occurrences of its corresponding X-variable.

Since the module produced by the linkage editor may be linked in a further step, the NX-commands which correspond to unresolved references are retained.

10.3 NN (name) command

NN-command → "N" N-variable "," name "."

N-variable → "N" hexnumber

name → char-string

An NN-command shall be provided for each symbol which is defined in the current module and which may not be referenced from any other module (i.e., is local to the module). It may be passed to a symbolic debugger, or it may provide symbols for the type table. It indicates that in the current module the corresponding N-variable (see Section 6.4) is associated with the local definition of name. The effect of more than one NN-command in the same module with the same name is undefined. An NN-command shall always come before all occurrences of its corresponding N-variable.

10.4 AT (attributes) command

AT-command → "AT" variable "," type-table-entry ("," lex-level ("," hexnumber)*)? "."

variable → I-variable | N-variable | X-variable

type-table-entry → hexnumber

lex-level → hexnumber

The AT-command is used to define attributes of a symbol such as type or debugging information. The name and value (i.e., load address) attributes are defined by other commands. The type-table-entry identifies an entry in the type table which defines the type for this symbol. The lex-level may be used to show the scope in block-structured language modules; additional hexnumbers are implementer-defined. Normally the AT-command designates I-, N-, or X-variables. In the last case, type checking is performed, and the type-table-entry for each X-variable must be compatible with the type-table-entry for the corresponding I-variable (see the TY-command below).

10.5 TY (type) command

TY-command → "TY" type-table-entry ("," parameter)+ "."

type-table-entry → hexnumber

parameter → hexnumber | N-variable | "T" type-table-entry

The TY-command specifies an entry in the type table. The type-table-entry is an access number that is used in this module to refer to this entry. The remainder of the command specifies a list of type parameters. A hexnumber parameter is a literal; an N-variable refers to the character string that is the name of that variable; and a T-number refers recursively to another entry in the type table.

Type compatibility is established between two entries if the strings represent identical type trees: that is, they have the same number of parameters of the same form; the numeric parameters are respectively equal; the N-variable entries refer to variables with the same names; and the type tree references refer to compatible type table entries.

'Don't care' types are defined with entries of zero: a numeric parameter of zero is considered equal to any numeric parameter; N0 is considered to be the same name as any other N-variable; and T0 is considered to be compatible with any type.

Example—in the following table, types T3 and T4 are compatible:

```
T1,13,N0,88.
T2,13,N8,0.
T3,44,T7,T1.
T4,44,T0,T2.
```

11. AS (assignment) command

AS-command → "AS" MUFOM-variable "," expression "."

e.g., ASP,A00.
ASI7,R,8C,+.

The value of expression is assigned to the variable.

12. 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 "blocks" where the information is destined to be loaded into contiguous locations. If all the information is completely known, as is the case for absolute loading, each block represents an exact image (encoded in the MUFOM representation) of a memory region after the loading. In such a case, the block may be specified by one or more successive LD-commands. Otherwise the information may be divided between LD and LR commands, which are successively loaded in the order processed.

12.1 LD (load) command

LD-command → "LD" load-constant+ "."

load-constant → hexdigit+

e.g., LD1D7F1D8033C63006A144A144A16010415A 2B2A3D2D403C3E.

The load-constant is a value which is loaded into one or more MAUs. The load-constant shall be a fixed number of digits with the most significant first. The number of digits shall be defined by the implementer. Only processes which are specific to a particular target machine (e.g., code generators, loaders) are required to encode or decode load constants; the linker and relocator do not use or modify the LD-commands, but only copy them.

12.2 IR (initialize relocation base) command

IR-command → "IR" relocation-letter "," relocation-base ("," number-of-bits)? "."

relocation-letter → nonhexletter

relocation-base → expression

number-of-bits → expression

e.g., IRQ,X4,10.
IRG,R3,45,−,8.
IRT,XA,3.

The IR-command initializes a relocation base (designated by the relocation-letter) for use in the LR (load-relocate) command. The relocation-base is evaluated by the relocator at the time the IR command is encountered and used to offset each address that designates this base in all subsequent LR-commands, until another IR command reinitializes this base.

The number-of-bits associates an address field size with this relocation base, for all subsequent references. This size shall not exceed the product of MAUs-per-address and bits-per-MAU as specified in the AD-command (see Section 7.4). If number-of-bits is not specified, it is equal to that product.

12.3 LR (load relocate) command

LR-command → "LR" load-item+ "."

load-item → relocation-letter offset "," | load-constant | "(" load-value ("," number-of-MAUs)? ")"

relocation-letter → nonhexletter

offset → hexnumber

load-value → expression

number-of-MAUs → expression

e.g., LR0000HB2,0001(P,10,+,2).

The LR-command is a compact notation for loading and for relocating information. Three different kinds of load-item may coexist in the command:

A load-constant is absolute (unrelocated) code that is to be loaded into as many MAUs as are required, in the same form as the LD-command (0000 and 0001 in the example above).

An offset preceded by relocation-letter (HB2 above) is added to the designated relocation base, which will have been specified by its most recent IR-command (see Section 12.2). As many MAUs are loaded as are required to contain the number of bits specified in the IR-command.

A parenthesized expression permits arbitrary relocation expressions (load-values). The number-of-MAUs, if omitted, is assumed to be the MAUs-per-address specified in the AD-command (see Section 7.4). The specified or default number of MAUs is loaded with the value of the expression, right-justified (i.e., the most significant bits are discarded or filled out with binary zeros, as appropriate). In the example above, the last load item is sixteen plus the location counter P, filling 2 MAUs.

If, in the example above, it is assumed that P is initially hex 0123, and relocation base H has been initialized with a value of hex 1A and a field width of 5 bits, and a MAU is defined as 16 bits wide, then the result of that LR command is the following bit pattern (shown in hex):

0000 00AC 0001 0000 0136

The first MAU results from the load constant 0000. The second MAU results from the relocation-letter H with offset B2 (note the sum of 1A + B2 has the carry-out of the fifth bit suppressed). The third MAU results from the load constant 0001. The fourth and fifth MAUs result from the expression in parentheses (note that P has incremented to 0126 at the point of evaluation).

12.4 RE (replicate) command

RE-command → "RE" expression "."

e.g., REA.LR0000.
RE64.LR(7F800000,P,+,4).

The RE-command evaluates the expression to determine the number of times the immediately following LR-command is to be replicated (see Section 12.3). The maximum length of the LR-command portion to be processed by the RE-command is implementation-dependent, but it shall be at least 64 characters.

13. Linkage edition

This section contains commands which refer only to linkage edition. They give additional control over the actions of the linkage editor. The actions of these commands may further depend on options available with the linkage editor. For example, the option to retain NI-commands (see Section 10.1) might be overridden by a user option given to the linkage editor.

13.1 RI (retain internal symbol) command

RI-command → "R" I-variable ("," level-number)? "."

I-variable → "I" hexnumber

level-number → hex-number

This command indicates that the symbolic information previously defined by an NI-command (with the same I-variable) is to be retained in the object modules produced by any subsequent processing. The optional level number may be used to specify under what circumstances the symbol is to be retained.

13.2 WX (weak external symbol) command

WX-command → "W" X-variable
("," default-value)? "."

X-variable → "X" hexnumber

default-value → expression

This command indicates that the external symbol specified by a previous NX-command (with the same X-variable) is to be flagged as a weak external.

In the case that the external reference is unsatisfied, the evaluated expression is the value of the X-variable.

14. Libraries

These commands refer specifically to the problem of loading routines which are contained in libraries. They are meant to give suitable defaults and to overcome the problem of two or more routines with the same name being present in separate libraries.

14.1 LI (specify default library search list) command

LI-command → "LI" char-string ("," char-string)* "."

This command specifies the default order in which libraries are to be searched to resolve unsatisfied external references. Char-string identifies a library, and the search is made over the libraries specified by these char-strings in left-to-right order. LI overrides any previous LI-command within the same module.

14.2 LX (library external) command

LX-command → "L" X-variable ("," char-string)+ "."

The LX-command specifies the library or libraries from which the individual external reference corresponding to the X-variable is to be satisfied.

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

15. MUFOM kernel

The following features shall be supported by a minimally conforming implementation.

Expressions:

Operators: +, −, @NEG
(Implied stack at least 2 values deep)

Variables:

G, I, L, P, R, S, X (only W, N missing)

Commands:

MB
ME
SB
ST (without N or P attributes)
SA
NI, NX
LR, IR (without options)
AS

16. Binary format

The MUFOM format is specified for media portability. An optional binary format may be implemented for internal use where file space or I/O time is critical. An implementation of the binary format shall provide bidirectional translation between it and the character form. See Appendix B for an example of a binary encoding.

Appendix A—Collected syntax

This appendix is not a part of P695/D3.1 and is for information only.

variable → letter (letter | digit)*

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

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 → hexdigit hexdigit

char-string → char-string-length character*

MUFOM-variable → nonhexletter hexnumber?

expression → hexnumber | MUFOM-variable | polish-expr | conditional-expr

polish-expr → (operand ",")* function

operand → expression

function → "@" identifier | operator

operator → "+" | "−" | "/" | "*" | "<" | ">" | "=" | "#"

polish-expr → "@F"

polish-expr → "@T"

polish-expr → operand "," monad-f

monad-f → "@ABS" | "@NEG" | "@NOT" | "@ISDEF"

polish-expr → operand1 "," operand 2 "," dyad-f

dyad-f → "+" | "−" | "/" | "*" | "@MAX" | "@MIN" | "@MOD" | "<" | ">" | "=" | "#" "@AND" | "@OR" | "@XOR" |

polish-expr → operand1 "," operand2 "," operand3 "," "@EXT"

polish-expr → operand1 "," operand2 "," operand3 "," operand4 "," "@INS"

polish-expr → value "," condition "," errornum "," "@ERR"

value → expression

condition → expression

errornum → expression

conditional-expr → condition "," "@IF" "," expression1 "," "@ELSE" "," expression2 "," "@END"

condition → expression

MUFOM-variable → nonhexletter hexnumber?

G-variable → "G"

I-variable → "I" hexnumber

L-variable → "L" section-number?

section-number → hexnumber

N-variable → "N" hexnumber

P-variable → "P" section-number?

section-number → hexnumber

R-variable → "R" section-number?

section-number → hexnumber

S-variable → "S" section-number?

section-number → hexnumber

W-variable → "W" hexnumber

X-variable → "X" hexnumber

MB-command → "MB" target-machine-configuration ("," module-name)? "."

target-machine-configuration → identifier

module-name → char-string

ME-command → "ME."

DT-command → "DT" digit* "."

AD-command → "AD" bits-per-MAU ("," MAUs-per-address ("," order)?)? "."

bits-per-MAU → hexnumber

MAUs-per-address → hexnumber

order → "L" | "M"

CO-command → "CO" comment-level? "," comment-text "."

comment-level → hexnumber

comment-text → char-string

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

CS-command → "CS" checksum? "."

checksum → hexdigit hexdigit

SB-command → "SB" section-number "."

section-number → hexnumber

ST-command → "ST" section-number ("," section-type)* ("," section-name)? "."

section-number → hexnumber

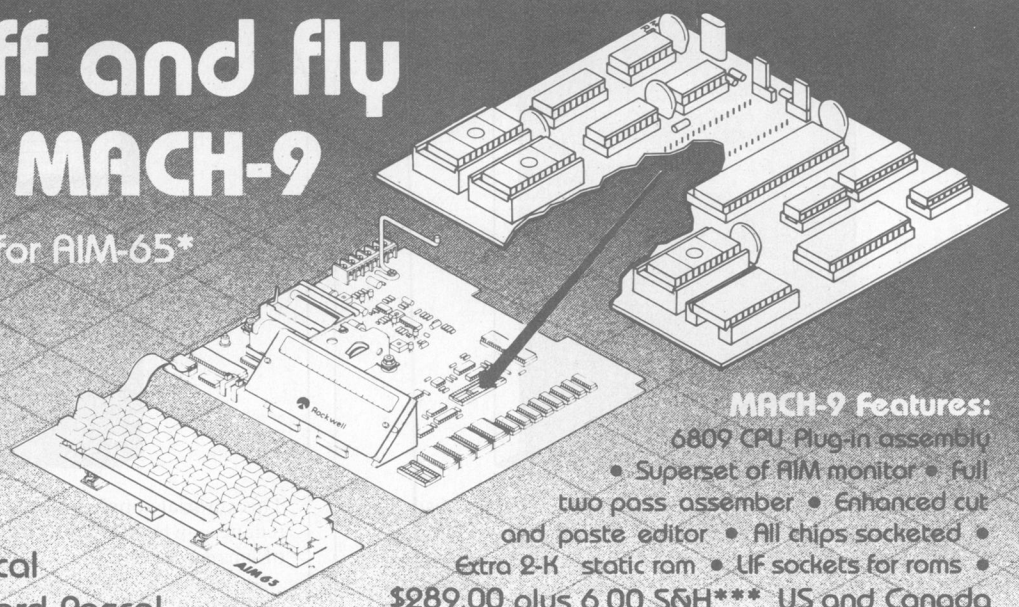
section-type → identifier

section-name → char-string

SA-command → "SA" section-number "," MAU-boundary? ("," page-size)? "."

Take off and fly with the MACH-9

The 6809 adaptor for AIM-65*



MACH-9 Features:
6809 CPU Plug-in assembly • Superset of AIM monitor • Full two pass assembler • Enhanced cut and paste editor • All chips socketed • Extra 2-K static ram • LIF sockets for roms •
$289.00 plus 6.00 S&H*** US and Canada

"Just Released"
MACH-9 Control Pascal
A superset of standard Pascal
No rom expansion board necessary

Sieve Benchmark**

Compiled Bytes	Total Bytes	Comp + Load	Execute
154	154	12 sec	264 sec

Introductory Price $69.00 plus $5.00 S&H US and Canada

For more Information Contact:
Modular Mining Systems, Inc. • 1110 E. Pennsylvania St.
Tucson, Arizona • 85714 • (602) 746-0418

In the UK Contact:
ACS Microsystems Ltd. • Gresham House
Twickenham Rd. • Feltham Middlesex •
TW13 6HA • 01-898-3775.

VISA

master charge

*AIM-65 is a trademark of Rockwell International • **Byte Magazine Sept. 1981 pg. 192 • ***$20.00 S&H for overseas.

Reader Service Number 3

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

MAU-boundary → expression

page-size → expression

NI-command → "N" I-variable "," ext-def-name "."

I-variable → "I" hexnumber

ext-def-name → char-string

NX-command → "N" X-variable "," ext-ref-name "."

X-variable → "X" hexnumber

ext-ref-name → char-string

NN-command → "N" N-variable "," name "."

N-variable → "N" hexnumber

name → char-string

AT-command → "AT" variable "," type-table-entry ("," lex-level ("," hexnumber)*)? "."

variable → I-variable | N-variable | X-variable

type-table-entry → hexnumber

lex-level → hexnumber

TY-command → "TY" type-table-entry ("," parameter)+ "."

type-table-entry → hexnumber

parameter → hexnumber | N-variable | "T" type-table-entry

AS-command → "AS" MUFOM-variable "," expression "."

LD-command → "LD" load-constant+ "."

load-constant → hexdigit+

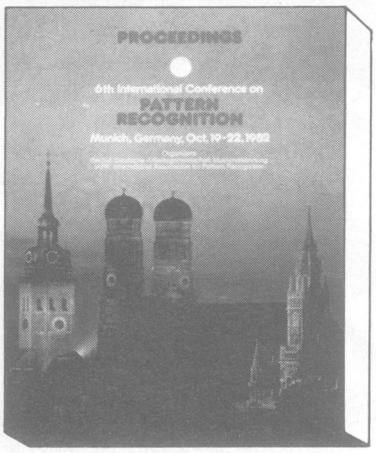
IR-command → "IR" relocation-letter "," relocation-base ("," number-of-bits)? "."

relocation-letter → nonhexletter

relocation-base → expression

number-of-bits → expression

LR-command → "LR" load-item+ "."



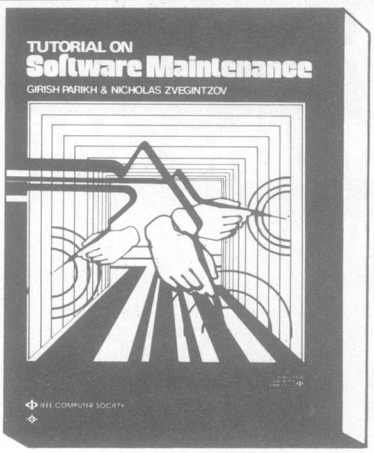
Major aspects covered in the proceedings include statistical, structural, and syntactic methods for classification of patterns, determining symbolic descriptions of patterns, implementations, and applications (character and speed recognitions, industrial and medical applications, remote sensing, and other fields). Two volumes. 1241 pp.

Order #436

Proceedings—Sixth International Conference on Pattern Recognition

October 19-22, 1982

Members—$48.00
Nonmembers—$70.00



This tutorial supplies a systematic overview of software maintenance—what it is, how to do it how to manage it, and the areas of current research.

It features thirty-one papers by thirty-seven leading authorities—the papers most often requested from their authors, papers in hard-to-find sources, papers that are the foundations of modern thinking in the topic and papers that extend the frontiers of research.

Order #453

TUTORIAL ON SOFTWARE MAINTENANCE

By Girish Parikh and Nicholas Zvegintzov

Members—$18.75
Nonmembers—$32.00

Use order form on p. 105.

load-item → relocation-letter offset "," | load-constant | "(" load-value ("," number-of-MAUs)? ")"

relocation-letter → nonhexletter

offset → hexnumber

load-value → expression

number-of-MAUs → expression

RE-command → "RE" expression "."

RI-command → "R" I-variable ("," level-number)? "."

I-variable → "I" hexnumber

level-number → hex-number

WX-command → "W" X-variable ("," default-value)? "."

X-variable → "X" hexnumber

default-value → expression

LI-command → "LI" char-string ("," char-string)* "."

LX-command → "L" X-variable ("," char-string)+ "."

Appendix B—Suggested binary encoding

This appendix is not a part of P695/D3.1 and is for information only.

A simple interference matrix of the syntactical forms in MUFOM shows that there is no place where there is a choice between character-string, small-hexnumber, load-constant, or identifier. Thus the same internal representation may be used for the first byte of all four of these without danger of ambiguity. All other syntactic forms defined in this standard can be uniquely encoded in 128 or fewer codes, leaving the remaining 128 codes to identify the first byte of the four.

The following encoding, while not intended to be definitive, shows how all the standard commands of this document may be encoded with gaps to add implementation-defined enhancements:

0nnnnnnn CCC . . . C	Character string, to 127 bytes
0xxxxxxx	Small (hex) number, 0 to 127*
0nnnnnnn BBB . . . B	Load constant, to 127 bytes**
0nnnnnnn CCC . . . C	Identifier, to 127 bytes**
10000000	Hex number omitted
1000nnnn BBB . . . B	Hex number, 0 to 8 bytes
1001xxxx	Implementer-defined functions
101xxxxx	Standard functions (see list)
110xxxxx (0nnnnnnn)	Standard variables and identifiers (see list)
111xxxxx	Standard command names (see list)
xxxxxxxx	Checksum—modulo 256***

*If small hexnumbers is optional and left out, hexnumber omitted symbol must be used.

**Can be arbitrary length in the character form.

***This checksum value will in general differ from the checksum value for the character form (see Section 15.1 above).

Encodings for standard command names, functions, and identifiers:

#	FUNCTION	IDENTIFIERS AND VARIABLES	COMMANDS
0	@F		MB
1	@T	A	ME
2	@ABS	B	AS
3	@NEG	C	IR
4	@NOT	D	LR
5	+	E	SB
6	−	F	ST
7	/	G	SA
8	*	H	NI
9	@MAX	I	NX
10	@MIN	J	CO
11	@MOD	K	DT
12	<	L	AD
13	>	M	LD
14	=	N	CS (with sum)
15	#	O	CS.
16	@AND	P	NN
17	@OR	Q	AT
18	@XOR	R	TY
19	@EXT	S	RI
20	@INS	T	WX
21	@ERR	U	LI
22	@IF	V	LX
23	@ELSE	W	RE
24	@END	X	
25	@ISDEF	Y	
26		Z	
27			
28			
29			
30	(		
31	)		

Proposed Standard

IEEE P695

Preliminary
Subject to Revision

Appendix C—Command and variable usage by processes

This appendix is not a part of P695/D3.1 and is for information only.

This appendix lists the MUFOM commands and variables against the conceptual processes.

This is a list of the conceptual processes, what they act upon, and what they produce. An implementation may also provide for "directives" into any or all processes. An implementation is likely to combine some of the conceptual processes into a single process.

INPUT/OUTPUT	CONCEPTUAL PROCESS
Several modules	
	Linkage editor (the process that combines several relocatable modules, resolving the externals)
Several sections	
	Relocator (the process that assigns an absolute starting address for each module)
Single "section"	
	Expression evaluator (the process that evaluates all expressions in the MUFOM commands)
Absolute section	
	Loader (the process that brings the binary information into memory)
Memory filled	

UTILITY MATRIX

Codes:

U	Used by the process.
. . .	Sent on (if present); i.e., ignored by process.
C	Possibly created by process (an implementation may create others, too).
M	Modified and then sent on.
UC	(Used and created) is a stronger transformation than M—the outgoing item may bear no relationship to the incoming item in contents and/or position.

Commands & variables	Linkage editor	Relocator	Expr. Eval.	Loader	
Meta:					
MB, ME	. UC				(required)
DT		C			
CO	. . C	. . . C	. . . C	. . . .	
CS	. UC	. . . UC	. . UC	. . U	
Descriptive:					
AD		U	. . . U	. . . U	
SB, ST	. UC	. . . U			
SA	. . U	. . . U			
NI,NX,RI, WX,LI,LX	. . U				
NN, AT, TY	. UC				U (on past loader)
Binary info:					
LD				U	
LR, IR	. . M	. . . M	. . . M	. . . U	
RE			. . . . . M	. . . U	(excluding LR part)
Active:					
AS	. UC	. . . UC	. MU	. . U	(see variables)
Variables:					
P		C	. . . U	. . . U	
R		M	. . . U		
W			. . . . . U		
L		U			
S	. UC	. . . U			(required*)
G			. . . . . U	. . . U	
I, X	. . U				
N	. . C				U (on past loader)
Operators:					
@NEG, @INS		C	. . . UC	. . U	
others		C	. . . U		
binary info				C . .	U (into memory)

*If the relocator does not textually substitute values for S-variables, then they need to be propagated to the expression evaluator.