1991

MIDI control Language (MCL) : reference manual

Marshall Earl Edwards
San Jose State University

MIDI Control Language (MCL) reference manual

Edwards, Marshall Earl, M.A.

San Jose State University, 1991

MIDI CONTROL LANGUAGE (MCL)

REFERENCE MANUAL

A Thesis

Presented to

The Faculty of the Department of Music

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Arts

By

Marshall Earl Edwards

May, 1991

APPROVED FOR THE DEPARTMENT OF MUSIC

[illegible]

Professor J. Allen Strange

[illegible]

Professor Daniel Wyman

Pablo E. Furman

Dr. Pablo Furman

APPROVED FOR THE UNIVERSITY

[illegible]

ABSTRACT

MIDI CONTROL LANGUAGE (MCL)
REFERENCE MANUAL

by Marshall Earl Edwards

This thesis serves as a reference manual for the MIDI Control Language which was developed for use in the San Jose State University Electro-Acoustic Studio. MCL is a high level computer programming language with intrinsic MIDI functions for the generation of MIDI standard files.

The goal of this manual is to provide users with useful information about the language, it's application to the MIDI environment, and its compatibility with other MIDI application programs. Instruction is given in the use of MCL commands, operators and functions so users might develop their own MCL application programs to explore techniques of algorithmic composition.

Musical concepts in the 20th Century often defy expression with traditional musical notation and can be cumbersome to implement with MIDI sequencer programs. MCL fills the gap between the two by permitting the expression of musical concepts as algorithms. A significant portion of contemporary musical thought lends itself very well to algorithmic solutions and MCL is a good vehicle from which to do so.

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

BACKGROUND

Rather than to submit a composition to fulfil degree requirements for a Master of Arts Degree in Music Composition, the author--with many years of experience in Computer Science and Electro-Acoustic Music--was compelled to develop a programming language which incorporated operators and functions particularly useful in the generation of MIDI data. This interest lead to the design and implementation of the MIDI Command Language (MCL), which is targeted for composers who prefer a programming alternative over the more conventional music notation programs and sequencer programs that rely on tedious “cut and paste” manipulations.

ACKNOWLEDGEMENTS

In his book, Programming Languages (An Interpreter Based Approach).[1] Samuel N. Kamin provides an in-depth look into the principal concepts of non-imperative languages such as LISP, APL, SCHEME, SASL,CLU, SMALLTALK, and PROLOG. For each language under study, Kamin offers an interpreter emulation that presents the language in a syntactically and semantically simplified form. Although Kamin's interpreters are syntactically different from the real languages, they preserve the central concepts of each language. Kamin's book provides a point of departure from which to begin a trek into the integral elements of "state of the art" computer languages. It can save readers from light years of experimental oblivion and divert them from the pitfalls of compiler design.

MCL is modeled after Kamin's interpreter emulation of APL. Although its syntax is sometimes cumbersome and difficult to use, it is a viable alternative to real APL--which can be very cryptic and unreadable. Kamin's APL interpreter was first converted from Pascal to C (which was relatively painless thanks to the developers of Lightspeed Pascal and Lightspeed C). The next step was to modify the interpreter to operate within the Macintosh environment. Additional operators and MIDI functions were to added to augment the very basic set initially implemented in the Pascal version. Finally the text editor was added and MCL was forged into its present form.

1 S. N. Kamin, Programming Languages (An Interpreter-Based Approach), (New York: Addison-Wesley Publishing Company, Inc., 1990), p.v.

The author is very thankful to the International MIDI Association (IMA) for granting permission to include the document, "Standard MIDI File Format V 1.0," as an appendix in this manual. Users requiring further information on the MIDI protocol should contact IMA at the following address:

International MIDI Association (IMA)
5316 W. 57th St.
Los Angeles, CA 90056

Phone:	213-649-6434
FAX:	213-215-3380

DEDICATION

MCL is dedicated to the author's wife, Kathey, whose love provided the inspiration to complete this once abandoned ambition. Day by day, minute by minute, her support provided encouragement to keep plodding along line by line, and bug by bug...

MIDI CONTROL LANGUAGE (MCL)

REFERENCE MANUAL

Beta Test Version

B y

Marshall Earl Edwards

CHAPTER 1

INTRODUCTION

MCL--an acronym for MIDI Control Language--is a high level computer programming language with intrinsic functions specifically oriented for the generation of MIDI standard data files. MCL is not a music notation program; nor does it support the real time capture, editing, or performance of sequenced data. Its ambition, nevertheless, is to facilitate the generation of musical compositions through algorithmic means. A significant amount of contemporary musical thought lends itself very well to algorithmic solutions and--for composers who have a background in computer programming--MCL is a good vehicle from which to do so.

Composers who work within the MIDI environment are well aware that music can be quantized in terms of time durations, note numbers, and articulation parameters. What they may have overlooked, however, is the natural employment of mathematical set theory to manipulate these parameters. Musical data is easily stored in aggregate data structures such as vectors and arrays. MCL incorporates numeric and logical operators that manipulate such values in a simplified mathematical manner. When combined with MCL's control operators, which facilitate looping and branching--allowing complete programmer control over the flow of musical form, MCL is ideally suited for the development of algorithmic compositions.

Users must first, however, establish a firm grasp on the fundamentals of MCL before they can truly appreciate its power. The goal of this manual is to provide essential details on MCL commands, operators and intrinsic MIDI functions so that users will be proficient enough to develop their own MCL applications. In general, this manual serves to provide the user with useful information about the language, it's application to the MIDI environment, and its compatibility with other MIDI application programs.

MCL PHILOSOPHY

The goal of MCL is to provide composers with an environment that will free them to think algorithmically and to avoid the repetitive tasks artificially imposed upon them by scoring and sequencer programs. Composers should not be creatively bound by the mundane tasks of data entry. Computers are capable of much more than simple editing. They are most beneficial when processing raw data into useful information and can be used in this manner to generate musical information.

Data processing, however, requires that a computer code be developed that explicitly specifies the method in which information is derived. If computers are to be used to generate musical information, a composer's methods must be exactly stated in a computer program. MCL exists to make this process easier. The philosophy of MCL is to provide the composer with data processing operators that allow them to govern the process of composition rather than to be governed by the detailed specification of the compositional process; to approach composition cybernetically[1] in a self steering manner.

In 1985 a book entitled "Cybernetic Music" was published by an author named Jaxitron who suggests that:

> If we could describe in enough detail the way a composer manipulates information to produce a score, we could certainly reprogram the process for a computer. [2]

Jaxitron does not discount the process of composition; insinuating that composers can be easily replaced by computers. On the contrary, he is very much aware of the complex and sometimes cumbersome tasks associated with composition. Jaxitron proposes that computers provide assistance during the compositional process:

> We are going to impose the logic of music on computers so that computers can help us compose. We will also expose music to the logic of computing to free music from unreasonable outdated and inhibiting lines of thought. Computers can help in composition by performing quite routine tasks such as keeping track of the time durations allotted to each measure and transposing a sequence of pitches through a given interval. But we'll see that they can even compose melodies and harmonize them just as we would ourselves--that is, just as we would if the methods we describe to the computer really reflect our own methods. The key to Cybernetic music is the description of method itself.
>
> Standard musical notation and terminology may be used to demonstrate method, but they can not describe it. If we are to explain it to a computer, we must certainly be able to describe it to ourselves. Anyone who has ever listened to a composer trying to explain just what it is that he does will appreciate the difficulties of this task. Anyone who has ever tried to get a computer to follow his wishes--not his programmed statements--will appreciate the remaining difficulties.[3]

---

1 The term cybernetics is derived form the Greek word kybernetes, which is defined as steersman or governor.

2 Jaxitron, Cybernetic Music, (Blue Ridge Summit, PA: Tab Books Inc., 1985), p. 1.

3 Jaxitron, Cybernetic Music, (Blue Ridge Summit, PA: Tab Books Inc., 1985), p. v.

A musical score is merely a detailed list of instructions for a musician to use as a guideline for the act of performing. The actual performance is an interpretation of the score. In electronic music, however, instruments are not yet as complex as the human mind and require explicit instructions. Articulation details, whose specification is the responsibility of the composer, increase tremendously when the performer is an electronic device. Consequently, MIDI application programs have focused primarily on the monumental task of managing enormous amounts of sequenced data. Although they offer convenient forms of data entry and are very proficient at data base management, they have made very little inroads with the management of compositional method. They are simply editors of performance instructions.

MCL is a computer language and as in all languages a program must be developed to instruct the computer--on a step by step basis--what must be done in order to accomplish a particular task. In a way, the program is analogous to a musical score and the computer is a very unresourceful--but extremely precise--performer. A computer does only what it is instructed to do. It is the computer programmer's responsibility to specify the instructions.

A programmer must work through a problem, step by step, until each operation has been clearly defined and a algorithm has been designed. If composers want to use computers to generate musical events, they must specify each musical event in a step by step manner. During this process they might become aware that they are performing specific manipulations on very basic thematic material. They may be augmenting note durations in a theme, transposing a passage to the dominant key, or restating an opening motive in retrograde. These and countless other compositional techniques can be looked upon as "musical functions." MCL is ideally suited for the implementation of musical functions for the following reasons:

1) Numeric and logical operators shield users from the forbidding details of scalar, vector and matrix operations, greatly simplifying the mathematical manipulation of sets.

2) MIDI functions perform all the necessary memory management tasks associated with maintenance of a linked list of MIDI events. They also automatically compute the length of character strings and data buffers used in MIDI meta-events.

3) Users are able to define their own functions that are able to use another MCL operator, MIDI function, or previously defined user function, adding extensibility to the MCL programming environment. This allows users to build up a collection of valuable compositional tools.

These features facilitate easy implementation of musical functions that posses great musical utility. As a user's collection of tools grow larger, the process of composing becomes less tedious. Consider the following potential musical functions:

1) A function that incorporates the rules of serialism to transform a twelve tone row submitted by a user into a 12 by 12 transposition matrix.[4]

2) Functions that incorporate the rules of 16th, 18th, or 20th Century counterpoint to generate contrapuntal variations on thematic material submitted from a user.

3) Embellishment functions based on the structural theories of Javanese Gamelan to transform simple *balungan*[5] into complex layers of melody with overlapping and interlocking themes.

4) A function that generates a minimalist phasing between voices, which would be especially tedious to enter into a sequencer program.

5) Functions that use the random number generator to create random notes, durations, and articulation parameters for aleatoric compositions.

Although music generated from these suggested MCL applications is significantly different, they all share a single commonality. They posses clearly defined compositional methods that can be embodied within musical functions. Music, regardless of style and period, can be expressed as a set of manipulations of very basic thematic materials. Whether it is a motive or counter motive from a two part invention, simple silence, or the sound of fingernails scratching a black board, a composition is formless until the basic thematic material has been developed! Even aleatoric compositions must express the method in which it is performed.

Clearly every culture throughout the history of man displays a particular preference for certain compositional or improvisational methods. In the history of western music, each period is identified with an associated music theory that can usually be embodied within a rule based system and is by definition applicable to computer generation. Any compositional method can be broken down into its elementary components and expressed as a sequence of

4 Refer to the function invmat in Chapter 7.

5 Balungan means 'skeleton melody' in Javanese music. All embellishment patterns are derived from the balungan.

instructions. Computers understand instructions! Why not use them to perform work? The resulting compositions will always obey the specified rules of method. Remember, a composer is the creator, and forever retains the right to judge whether or not the result is aesthetically pleasing. A composer may--at any time--revise the rules, or simply choose to alter their methods.

MCL'S VALUE OPERATORS

MCL incorporates a subset of APL's[6] operators that extend arithmetic and logical operations to the "larger values" of vectors and matrices. In APL these values are manipulated as easily as scalar values are in other programming languages. The programmer is sheltered from work being done internally during the execution of an operation. APL users care little about the nuances of a matrix multiply or the summation of elements in a vector. Users are able to focus on the application of mathematical manipulations rather than to bog down in their implementation.

Leonard Gillman and Allen J. Rose comment in their book, APL (An Interactive Approach), on APL operators:

> ...Once one has mastered APL one has accomplished something of permanent value and benefit: programs of great value and interest can be written by one person that in other languages would require a team!
>
> The power of the APL language comes from its direct manipulation of aggregates of data in the form of arrays. Everyone recognizes that computers excel where aggregates are manipulated, where the descriptive details of a function do not grow with the size of the aggregates being manipulated, and where one description suffices to cover a large population of aggregates. Most other computer languages require their programs to penetrate these structures, manipulate the components individually even in order to achieve a uniform effort. It is not surprising that APL programs are significantly shorter and more lucid than programs in most other languages. In programming, clarity is not a consequence of discursiveness or low information density in the program text. [7]

6 APL is an acronym for "A Programming Language" and was originally conceived by Ken Iverson in 1962. Years later the language was developed at IBM by a group of programmers led by Iverson. The language is noted for its ability to perform operations on arrays with a single expression in "closed form" without iteration.

7 L.Gilman and A. J. Rose, APL (An Interactive Approach), (New York: John Wiley & Sons, Inc., 1984), p.i.

APL operators are well suited for musical applications. A musical phrase can be viewed as a single array with five columns: column one storing note durations, column two channel numbers, column three note values, and columns four and five attack and release velocities. A simple matrix subscript would extract a row from the array returning a vector of performance parameters for a single note event. Manipulation of vectors and arrays offer great potential to music applications.

DISCLAIMER

This initial implementation of MCL is a "proof of concept" and should be viewed as a beta test version. It does not profess to be fully operational, tested, or bug free. Therefore, in the hope of minimizing user frustration, the following list of known bugs is submitted:

1) Matrix operations require that arguments have a specific rank compatibility and system errors will likely result whenever these requirements are violated. In practice MCL attempts to handle known rank errors. However, there is a high probability that the current version of MCL has not accounted for all possible errors. Users should become acquainted with the mathematical theories of matrix operations and to apply them accordingly to reduce rank errors. Keep in mind that a system crash during a matrix operation indicates a possibility of a rank error.

2) This beta version does not perform any garbage collection duties. Memory dynamically allocated for large data arrays during run time will not be returned to the free memory heap when they are no longer needed. Consequently, as more and more free memory is consumed the operating system will take a longer time to allocate dynamic memory. MCL will eventually bog down when available free memory becomes scarce. A system error will occur if heap memory is totally consumed and MCL may or may not be able to recover from the error.

It is heavily recommended that a Macintosh II or SE/30 populated with an abundance of RAM be used in the evaluation of MCL. A Macintosh Plus (or early SE) will bog down very quickly when heap memory is nearly exhausted. Its slow processing speed and memory limitation of 4 megabytes make it a bad candidate to run the beta version. In a benchmark test case developed to repeat a MIDI note on and note off sequence, 10,000 iterations were executed on a DEC VAX 8700 in approximately 3 seconds. A Macintosh SE/30 with 8 megabytes of RAM performed 2,000 iterations in 10 seconds. The same test on a Macintosh Plus with 4 megabytes of RAM resulted in a system crash.

If users do not have access to a Macintosh II or an SE/30, they can still use MCL on a Macintosh Plus if they obey the following guidelines: Use MCL to generate small fragments of musical material that can imported into a sequencer program that is capable of importing standard MIDI files. The sequencer software can be used to glue together imported MCL fragments. This will prevent MCL from consuming large amounts of heap memory and will help alleviate system errors. In general, users should minimize data allocation and avoid deeply nesting operators and functions.

Since the beta version of MCL is a "proof of concept" implemented to demonstrate the semantics and syntax of the language, it was thereby justified to omit garbage collection. MCL with a garbage collection facility warrants a commercially marketable software product, which is beyond the scope of a Master's thesis in Music Composition. Once the requirements of MCL have been rigidly defined and the most appropriate method can be determined, a garbage collection utility will be added. At that time MCL will mature and will operate satisfactorily on any Macintosh computer. Until then it is available solely on a test basis. It is essential that all bugs be reported directly to the author to facilitate a prompt fix. Whenever bugs are found or a user has suggestions as to how MCL can be improved, it would be greatly appreciated if the user would direct all correspondence to:

Marshall Earl Edwards
1105 Washington SE
Albuquerque, NM 87108

HOW TO USE THE MANUAL

Chapter two presents an overview of MCL's programming environment which includes specific details of the MCL interpreter and editor. The interpreter section introduces the user to MCL's commands, expressions, variables, constants, and user defined functions. The editor section covers file management and text editing.

Chapter three familiarizes users with MCL's Macintosh user interface. There is a listing of MCL's menus, and a description of the command line and edit windows. A brief tutorial is presented that demonstrates the extraction of operator help text into an edit window.

Chapter four provides a comprehensive listing of MCL's control operators, value operators, and MIDI functions.

Chapters five and six are concise glossaries of MCL operators and MIDI functions. The glossaries include proper operator syntax, a description of what the operator or function does, and a listing of the operator or function arguments. Experienced users may want to use these chapters as quick reference guides.

Chapter seven offers several example MCL applications. Among them are demonstrations of control and value operators; a function that converts a twelve tone row vector into a twelve by twelve transposition matrix; and aleatoric musical example that uses the random number generator.

CHAPTER 2

MCL PROGRAMMING ENVIRONMENT

The MIDI Control Language is an interactive interpreter that is bundled with a text editor for editing and managing source code files. Together they provide the user with a complete programming environment. This chapter provides an overview of the interpreter and the editor. The interpreter section presents the fundamentals of the MIDI Control Language. The second section covers the editor and provides details on file management and text editing.

THE INTERPRETER

When MCL is first entered the user is faced with a familiar Macintosh user interface. If an entry is typed into the command line the interpreter is invoked to evaluate user input as either a command, variable name, expression, or comment. If a command is entered then it is promptly executed.[8] If a variable name is entered its current values are displayed. MCL expressions are embodied in a matching pair of left and right parenthesis. A comment begins with a semicolon. If the user has entered none of the above, an error message will be displayed in an alert dialog informing the user that their entry is unknown. Any expression resulting in an error will also result in the display of an error message.

The interpreter can be used interactively from the command line or directed to load source code from a file. In either case interpreter response to a line of input is output to the currently opened edit window, which is referred to as the **output stream**. If no edit window is currently opened, then a new "Untitled" window will be opened to receive the interpreter response.

VALID ASCII CHARACTERS

The interpreter will accept any printable character from the keyboard. Spaces, tabs, parentheses, single quotes, double quotes, and semicolons are used as delimiters. The plus sign, minus sign, asterisk, and slash are are used as operators. The following special characters should be avoided in variable names and user function names:

( ) ' " + - * /

Illustration 2.1 Special Characters

8 Commands can also be selected from the Commands menu, which is covered in Chapter 3.

Spaces and tabs delimit words. Parentheses delimit the beginning and ending of expressions. Single quotes and double quotes delimit the beginning and ending of vector expressions. The semicolon is designated as the comment character and signals the interpreter to ignore all text until the end of the line. Lines that begin with semicolons are skipped over.

Command names, operator names, and function names are reserved as key words. Variable names and user function names should never contain a delimiter character or duplicate a key word. As long as these constraints are obliged, variable names and user function names may use any character on the keyboard.

VALUES

In MCL a value can be either scalar, vector, or matrix. They can be permanently stored in a variable or exist only as intermediate result returned from an operator or function. Result values can be used as an argument to another operator or function or simply displayed in the output stream. They are lost for further use, however, if they are not stored in a variable.

OPERATORS AND FUNCTIONS

MCL's control operators, value operators, and MIDI functions[9] are accessible when they are embedded within expressions and entered into the interpreter's command line; where the interpreter will evaluate the expression (the equivalent of running a program). If the last operator or function called in an expression returns a result value, the value will be displayed in the output stream. No output will occur if the last operator or function does not return a result value. As a rule, all value operators return result values, while control operators and MIDI functions do not .

EXPRESSIONS

The beginning of an expression is marked with a left parenthesis. It is followed by an operator or function name with an argument list. The end of an expression is marked with a closing right parenthesis. The following expression calls a function named **dog** with variables **a**, **b**, and **c** as arguments:

(dog a b c)

Illustration 2.2.1 Expression Syntax

9 MCL operators and functions are covered in detail in Chapter 4.

Arguments can be replaced with nested expressions.The return value from a nested expression can be used as an argument in another expression. Consider the following examples:

```
(dog a b (cat c))
(lion z (dog a b (cat c)))
```

Illustration 2.2.2 Nested Expressions

In the first example the outer expression calls a function named **dog** which has three arguments in its argument list. The first two arguments passed to **dog** are variables **a** and **b**. The third argument however is a nested expression that calls a function named **cat** with variable **c** as its argument. The result value returned from **cat** will be used as the third argument passed to the function named **dog**. The resulting value returned from dog will be printed in the output stream.

In the second example the result returned from the most nested expression--the expression that calls function **cat** with argument **c**--is used as the third argument in the next higher nested expression--the expression that calls function **dog**. The result value from this expression is used as the second argument for the outer most expression--which calls function **lion** with two arguments.

Vector constant expressions are used to specify vector constants. Only numeric and predefined constants can be used within a vector expression. Nested expressions are not allowed within a vector constant expression. Vector expressions can specify two types of vectors, numeric and character. A single quote identifies a numeric vector. A double quote identifies a character vector:

In the following example the first vector expression is numeric and the second is character:

```
'1 2 3 4 5 6 7 8 9 0x0A 0x0B WHOLE'
"(C) 1991 Marshall Earl Edwards"
```

Illustration 2.3 Numeric and Character Vector Expressions

The numeric vector in the previous example includes decimal constants, hex constants, and a predefined MID constant. The character vector is a string that might be used to specify a copyright notice. Notice that the string has parentheses within it. When a string has parentheses embedded within it, they must be always in order (i.e., a left parenthesis balanced with a right parenthesis) or else the interpreter will not be able to resolve the string.

Although MCL's expression syntax can be quite confusing at first when specifying mathematical equations, it should not be of any great hardship for users once they become accustomed to it. The following example demonstrates how a involved equation might be represented in MCL:

Mathematical Equation: **i = 2 * j - k/3**

MCL Expression: **(set i (- (+ (* 2 j) i) (/ k 3)))**

Illustration 2.4 Mathematical Equation Expression

PREDEFINED CONSTANTS

Scalar constants are always numeric and may entered in decimal or hex. MCL also allows users to use a set of predefined MIDI constants. The following list of predefined durations will be set according to the current default number of ticks per quarter note, which is set when the MIDI function **MidiFileHeader** is called:

WHOLE	HALF_DOT
HALF	QUARTER_DOT
QUARTER	EIGHTH_DOT
EIGHTH	EIGHTH_TRIPLET
SIXTEENTH	SIXTEENTH_TRIPLET
THIRTYSEC	THIRTYSEC_TRIPLET

Illustration 2.5.1 Table of Predefined MIDI Duration Constants

MIDI key numbers range from 0 to 127 and may be specified by one of the following predefined MIDI constants:

C-2	C-1	C0	C2	C3	C4	C5	C6	C7	C8	C9
C#-2	C#-1	C#0	C#2	C#3	C#4	C#5	C#6	C#7	C#8	C#9
Db-2	Db-1	Db0	Db2	Db3	Db4	Db5	Db6	Db7	Db8	Db9
D-2	D-1	D0	D2	D3	D4	D5	D6	D7	D8	D9
D#-2	D#-1	D#0	D#2	D#3	D#4	D#5	D#6	D#7	D#8	D#9
Eb-2	Eb-1	Eb0	Eb2	Eb3	Eb4	Eb5	Eb6	Eb7	Eb8	Eb9
E-2	E-1	E0	E2	E3	E4	E5	E6	E7	E8	E9
F-2	F-1	F0	F2	F3	F4	F5	F6	F7	F8	F9
F#-2	F#-1	F#0	F#2	F#3	F#4	F#5	F#6	F#7	F#8	F#9
Gb-2	Gb-1	Gb0	Gb2	Gb3	Gb4	Gb5	Gb6	Gb7	Gb8	Gb9
G-2	G-1	G0	G2	G3	G4	G5	G6	G7	G8	G9
G#-2	G#-1	G#0	G#2	G#3	G#4	G#5	G#6	G#7	G#8	G#9
Ab-2	Ab-1	Ab0	Ab2	Ab3	Ab4	Ab5	Ab6	Ab7	Ab8	Ab9
A-2	A-1	A0	A2	A3	A4	A5	A6	A7	A8	A9
A#-2	A#-1	A#0	A#2	A#3	A#4	A#5	A#6	A#7	A#8	A#9
Bb-2	Bb-1	Bb0	Bb2	Bb3	Bb4	Bb5	Bb6	Bb7	Bb8	Bb9
B-2	B-1	B0	B2	B3	B4	B5	B6	B7	B8	B9

Illustration 2.5.2 Table of Predefined MIDI Note Constants

VARIABLES

A variable is a reserved area of memory that is identified by a user specified name that stores a scalar, vector or matrix value. Variables can be redefined to another rank and shape. All variables are global with the exception of the arguments in a user function definition, which are local to the function.

USER DEFINED FUNCTIONS

Users may define their own functions by using the **define** operator. The following is an example of a user defined function:[10]

```
(define addVars (a b)
     (begin
          (set cglobal
               (+ a b)
          )
          (print cglobal)
     )
)
```

Illustration 2.6   Example User Defined Function

Notice that in this example expressions there are four nested levels. At Level one a call to **define** is made to initiate a user defined function. The name of the function is listed next. In this case the name is **addVars**. Care must be taken not to use any of the key words reserved for MCL's operators and functions. Next comes the argument list. The function **addVars** will have two arguments that will be local to the function by the names of **a** and **b**.

At level two it is time to specify the body of the function. In this example a series of operators will be called, therefore a call to the **begin** operator is made to block all expressions nested within level two. If a function only has but a single expression within its body, then **begin** does not have to be called.

At level three the first expression found within the **begin** block makes a call to **set** either create the variable **cglobal** or to redefine it. The variable **cglobal** is a global variable because it does not appear in the argument list. This expression has a nested expression within it bringing the nested level to four. This expression makes a call to the **+** operator which adds value **a** to value **b**. The result value is returned and used as an argument for the **set** operator which assigns the resulting sum of **a** and b to the global variable **cglobal**. The next expression within the begin block makes a call to the print operator to output the variable cglobal. Now MCL works its way back up through the levels of nesting until it reaches the interpreter. If any errors are encountered along the way they are reported in the output stream.

10 All operators in this example are covered in detail in Chapter 4.

INPUT AND OUTPUT

MCL inputs and outputs both text files an binary files. MCL source code is stored in text files and MIDI files are binary. Source code files are managed by the usual items under the **File** menu such as **New**, **Open**, **Close**, **Save**, **SaveAs**, and **Revert**. MIDI files are managed with the MCL commands **import** and **export** and may be entered either from the command line or the **Commands** menu.

MCL source code is saved in sequential ASCII files with variable length records. Files may contain any of the valid characters mentioned in the previous section but may not contain word processor formatting characters. Although MCL is capable of reading text files output from any editor, it can not deal with word processor files. Files created with word processors must be saved as simple text files before MCL will be able to read them. Source code files must contain valid MCL commands, expressions or comment lines. Commands are limited to a single line, while expressions may be entered on multiple lines.

MIDI standard binary files are formatted according to a document published by the International MIDI Association (IMA) entitled, "Standard MIDI File Format V 1.0,"[11] which is presented in the Appendix.

MCL COMMANDS

The following is a list of commands that are available from the command line or the **Commands** menu. They are listed here by their command line spellings. To obtain a listing of the MCL commands, enter **commands** into the command line. An listing of the commands will sent to the output stream window.

LOAD COMMAND

The **load** command reads a source code file and directs each line of text to the interactive interpreter which evaluates the line. Source code files should contain only valid MCL commands, expressions, or comments. If a variable name or user function name conflict with a previously defined name, the older version will be discarded and replaced with the newer version

11 Refer to the complete listing of the Standard MIDI File Format V1.0 in the Appendix for further information regarding the formatting of MIDI standard files.

EXPORT COMMAND

The **export** command starts at the top of the MIDI events list and as it works its way to the end, writes each event to a specified MIDI standard file which can be imported into other MIDI software products that support the MIDI file standard as specified in the IMA document, "Standard MIDI File Format V 1.0."[12]

EVENTS COMMAND

The **events** command will generate an ASCII listing of the events currently in the MIDI events list. Address pointers are stored in hex.

CLEAREV COMMAND

The **clearev** command will clear out any events stored in the current MIDI events list.

VARS COMMAND

The **vars** command will list each global variable currently present in the MCL system environment. Each variable is listed with its associated rank and shape.

VALS COMMAND

The **vals** command lists each global variable currently present in the MCL system environment with a printout of the elements currently assigned to the variable.

FUNCS COMMAND

The **funcs** command lists all user defined functions currently within the system environment with their associated argument lists.

12 Refer to Standard MIDI File Format V1.0 in the Appendix for further details.

THE EDITOR

MCL's text editor is capable of performing basic text edit functions on ASCII text files. The purpose of this section is to inform the user on the basic features of the editor. Although the editor is not very robust, it does work and will suffice if no other text editor is preferred. It's most redeeming value is that it is extremely easy to learn because it incorporates the standard editing techniques that users are familiar with in Macintosh applications. Experienced Macintosh users may wish to skip this section altogether and learn the editor through trial and error.

The editor included with this version of MCL will allow only one file to be opened at a time. When MCL is first entered an "Untitled" file is created and is the output stream for interpreter response to commands and expressions. If the user wants to create or open a text file they must first close the "Untitled" edit window. If the window contains interpreter response, they will be prompted by a dialog window to determine if the text should be saved in a file.

CREATING A NEW FILE

New files can be created only when no files are open. To create a new file select **New** from the **File** menu. The edit window will opened with the name "Untitled".

OPENING A FILE

Text files may be opened for editing by selecting the **Open** command in the **File** menu. The usual Macintosh dialog window will be displayed allowing users to move freely from folder to folder until they find the file they want to edit. The file can be opened by either clicking once on the file name and **Open** button, or by double clicking on the file name.

CLOSING A FILE

Files may be closed either by clicking in the close box of the output stream window or by selecting **Close** from the **File** menu. If changes have been made since the file was last saved, or if the file is new, or is an "Untitled" output stream window, the MCL editor will ask the user if they want to save their changes before closing the file.

SAVING A FILE

Files may be periodically saved without closing them by selecting **Save** from the **File** menu. If the output stream is an "Untitled" window, the file has never been saved and the user must select **SaveAs...** from the menu to name the file. A standard Macintosh dialog window will appear prompting for the name the file. If the cursor is moved to the folder name and the mouse button is depressed and held down, a pop up menu will appear displaying the hierarchical listing of the current working folder. If the cursor is dragged to another folder listed in the menu and the mouse button is released, a new folder will be selected and will become the new working folder. Folders displayed in the new file list can be opened by double clicking on the name. In this manner it is possible to move around the desktop until the desired folder is found. Then name of the file should be typed into the input line of the dialog window. Once the name is entered select the **Save** button and the file will be saved. If the file name already exists in the working folder, a small dialog window will prompt the user to determine if they want to replace the file. If the **Yes** button is selected the file will be over written. If the **No** button is selected the **SaveAs...** dialog window will be displayed and the file name will have to be entered again. Once the file has been saved, the new name will be displayed in the title bar of the edit window and can be updated by using the **Save** command.

SAVING A FILE WITH ANOTHER NAME

An opened file may be saved with a different name by selecting **SaveAs...** from the **File** menu. Changes to the opened file will not be saved, however, unless a **Save** is done before the **SaveAs...** command.

ENTERING TEXT

Text is entered simply by typing into the keyboard. The insertion point may be moved anywhere in the edit window by simply moving the cursor to a new point of entry. Text entered past the right edge of the edit window will be wrapped around to a new line. However, although word processors never insert hard carriage returns at line wraps, the MCL text editor does in order to preserve the file as a pure ASCII text file without special formatting characters to specify margins. MCL will wrap lines that are longer than can be displayed in the window (eliminating the need for a horizontal scroll bar). When longer lines are needed, the edit window should be sized to accommodate the desired length.

SCROLLING

The vertical scroll bar is used to move forwards or downwards through the body of the text. There are three ways to scroll and they are:

1) If the cursor is placed on the up arrow or down arrow at the ends of the scroll bar and the mouse button is depressed and held down, the text will be scrolled smoothly--line by line--in the direction of the selected arrow. This method is desirable when moving to exact desired file positions.

2) If the cursor is placed above the thumb box and the mouse button is clicked once, a page scroll upward will occur. Likewise placing the cursor below the thumb box and then clicking the mouse button will produce a page scroll downward. This method should be used to page scroll to the point of interest.

3) The quickest way to move significant distances within the file is to place the cursor over the thumb button, depress the mouse button and drag the thumb button to a position within the file proportional to the length of the scroll bar. Although this method is the fastest, the new file position will not be displayed until the mouse button is released, making it somewhat difficult to gauge where to stop. The thumb box should be used to scroll to a general area in the file, or as an easy way to get to the top or bottom of the file.

Although scrolling displays a new file position in the edit window, the point of insertion will not change unless the cursor is moved to a point in the edit window and the mouse button is clicked once. If this is not done, with the very first key typed, the edit window will return to the original point of insertion.

USING THE ARROW KEYS

Arrow keys move the insertion point up, down, left and right. This editor does not support the selection of lines of text with a shift up or shift down key.

SELECTING TEXT

The MCL facilitates the selection of text with the usual mouse technique of depressing the mouse button at the beginning of a selection range and holding the button down while dragging the mouse to the end of the selection range. Individual words can be selected by double clicking at a point within a word. All selected text ranges are displayed in reverse video. Once text has been selected, the edit commands can be used to.cut, copy, paste and clear.

EDIT COMMANDS

The MCL editor supports the usual edit commands found in all Macintosh programs. These commands are used to perform typical cut, paste, and delete selected text regions and insertion points.

CUT

The **Cut** command deletes text from the selected range and places it into the paste buffer. It is the first step in a cut and paste operation.

COPY

The **Copy** command places the text in the selected range into the paste buffer. It is primarily used to replicate text elsewhere in the file.

PASTE

The **Paste** command copies the text in the paste buffer to the specified point of insertion. This is the second step in a typical cut and paste operation. If the point of insertion is a selected range of text, the text in the paste buffer will replace the selected region.

CLEAR

The **Clear** command deletes text in the selected range. The only way to retrieve cleared text is to use the **Undo** command immediately following the **Clear** command.

UNDO

If text is inadvertently deleted or accidently typed over in a selected text range, it can easily be recovered by selecting **Undo** from the **Edit** menu. **Undo** reverses the last thing done. When mistakes are made, they need to be immediately corrected with the **Undo** command before any other action is taken.

CHAPTER 3

THE MACINTOSH USER INTERFACE

MENUS

MCL uses standard Macintosh menus that conform to industry standards with regard to menu order, command key equivalents, and implementation of hierarchical menus. MCL displays the following menu bar:

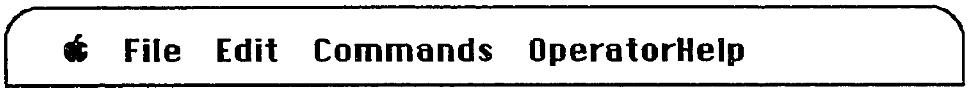


Illustration 3.1 MCL Menu Bar

MENU

The Apple Menu displays a listing of the desk top accessories that are currently installed in the System file. The first item listed is **About MCL...** and selecting it will display a dialog window that exhibits miscellaneous information about MCL. The desk top accessories can be invoked by selecting the desired item from the menu. The following is an example of a possible Apple menu that would be displayed under Finder:

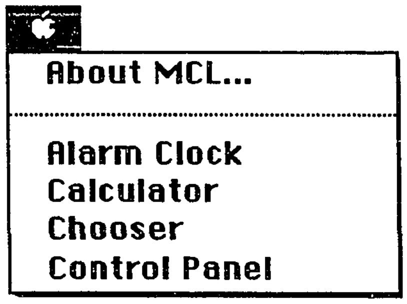


Illustration 3.2 Apple Menu

FILE MENU

The **File** menu displays menu items that perform various file handling and printing functions. Selecting **New** will create a new "Untitled" file. **Open** will open an old file. **Close** will close the current opened file. Save will update the current opened file. SaveAs... will save the current file with a specified name. **Revert** will ignore all changes since the file was last updated. **Page Setup...** will allow users to specify how text files are to be printed. **Print** will print the file using the current device as specified by **Chooser**. **Quit** will exit MCL.

Usage of the file menu items are described in greater detail in the editor section. The **File** menu is displayed as follows:

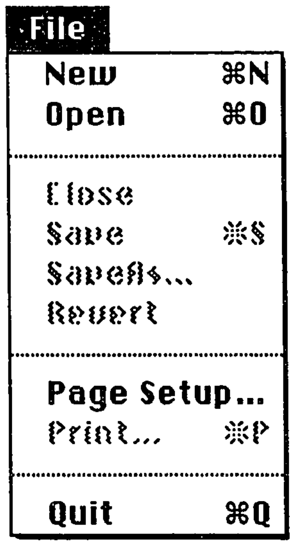


Illustration 3.3 File Menu

EDIT MENU

The **Edit** menu is used when the text editor has been invoked by clicking within the body of an edit window. Its usage is described in detail in the editor section. These menu items are also available to desk top accessories that require cut and paste capabilities. The **Edit** menu is displayed as follows:

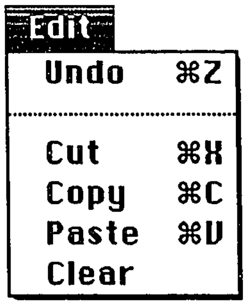


Illustration 3.4 Edit Menu

COMMANDS MENU

The commands menu provides access to MCL commands that include the loading and regeneration of MCL source code, the importation and exportation of MIDI files, functions that manage the MIDI events list, and display utilities that list variable names , variable contents, and a listing of user defined functions. The **Commands** menu is displayed as follows:

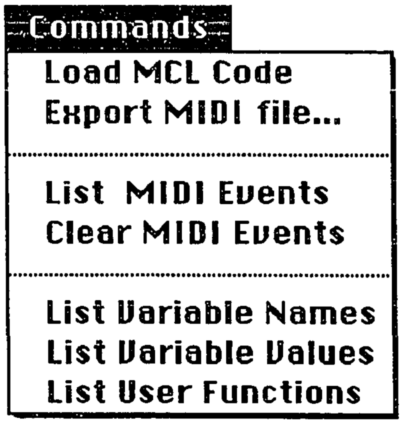


Illustration 3.5 Commands Menu

OPERATOR HELP MENU

Operator help is structured as a hierarchical menu subdivided into several submenus each of which represents a operator or function category.[13] When **OperatorHelp** is selected from the menu bar, a menu is displayed listing the available submenus. To display a submenu, keep the mouse button depressed and drag the cursor to the item. A submenu will pop up to the right of the main menu displaying the operators or functions available under the selected category. If the cursor is dragged to a particular operator or function item, help text will be displayed in a dialog window. Help text may be extracted into the edit window and this process is demonstrated in a brief tutorial in the next section.

The **OperatorHelp** menu is displayed as follows:

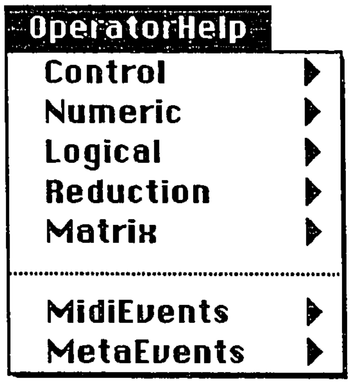


Illustration 3.6.1 Operator Help Menu

13 Refer to the Chapter 3 for a listing of the major classification of MCL operators and MIDI functions.

The **Control** submenu items are primarily used for branching and looping, or for the creation and subscription of variables. The **Control** submenu is displayed as follows:

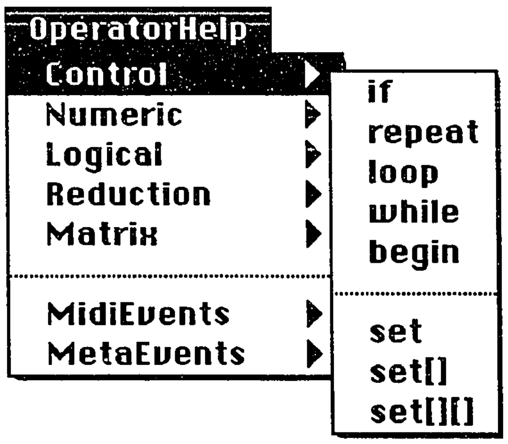


Illustration 3.6.2 Control Operators Submenu

The **Numeric** submenu items perform basic numeric operations on variables and constants.[14] The **Numeric** submenu is displayed as follows:

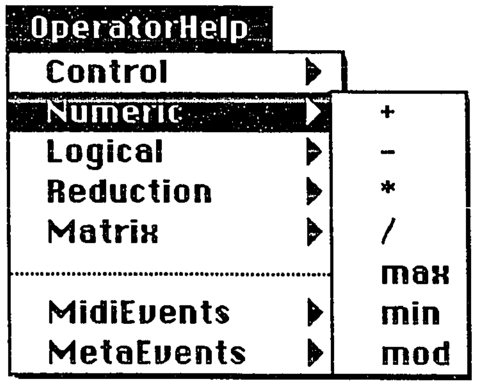


Illustration 3.6.3 Numeric Operators Submenu

14 In MCL the term constant has a larger scope than in most computer language. In addition to scalar constants, MCL supports vector and matrix constants. Their usage is covered in chapter 3.

The **Logical** submenu items perform various logical evaluations that determine relationships between variables and constants.

The **Logical** submenu is displayed as follows:

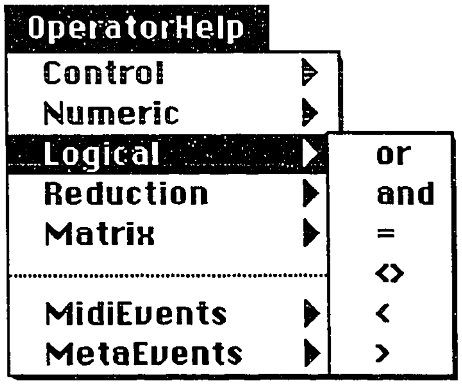


Illustration 3.6.4 Logical Operators Submenu

The **Reduction** submenu items perform miscellaneous numeric and logical operations on individual elements within a variable or constant.

The **Reduction** submenu is displayed as follows:

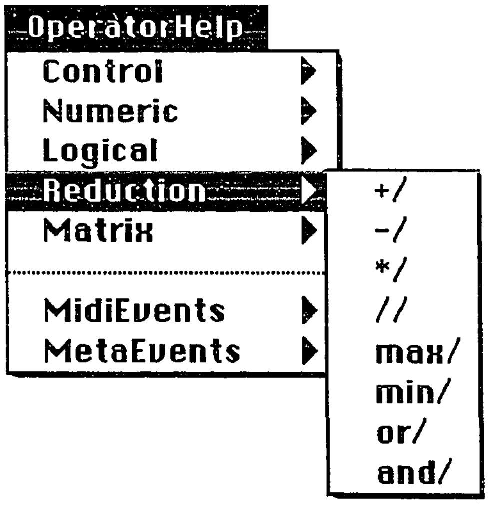


Illustration 3.6.5 Reduction Operators Submenu

The **Matrix** submenu items perform miscellaneous manipulations and operations on scalar, vector, and matrix variables and constants.

The **Matrix** submenu is displayed as follows:

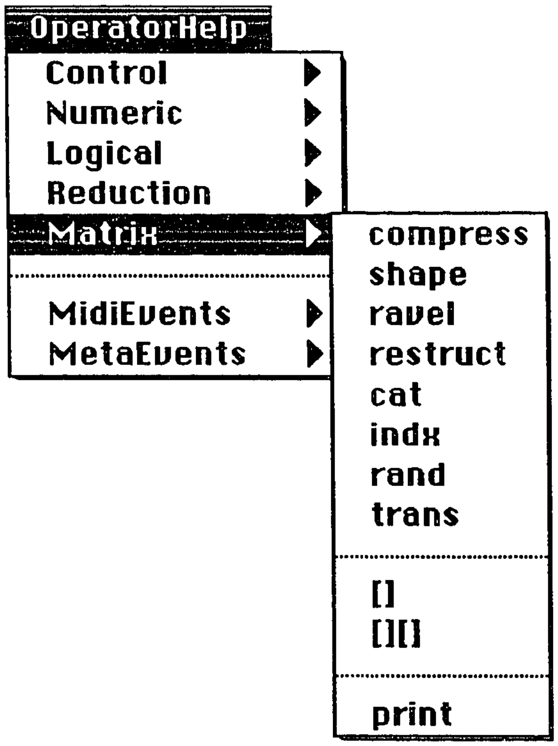


Illustration 3.6.6 Matrix Operators Submenu

The **MidiEvents** submenu items are intrinsic MIDI event functions. The **MidiFileHeader** function specifies basic information about a MIDI file including the file format, number of tracks, and number of ticks per beat. It is always the first MIDI function called when building a MIDI file events list. The number of ticks per beat is used to compute note duration constants. The remaining functions generate the time tagged MIDI events that are executed during real time performance.

The **MidiEvents** submenu is displayed as follows:

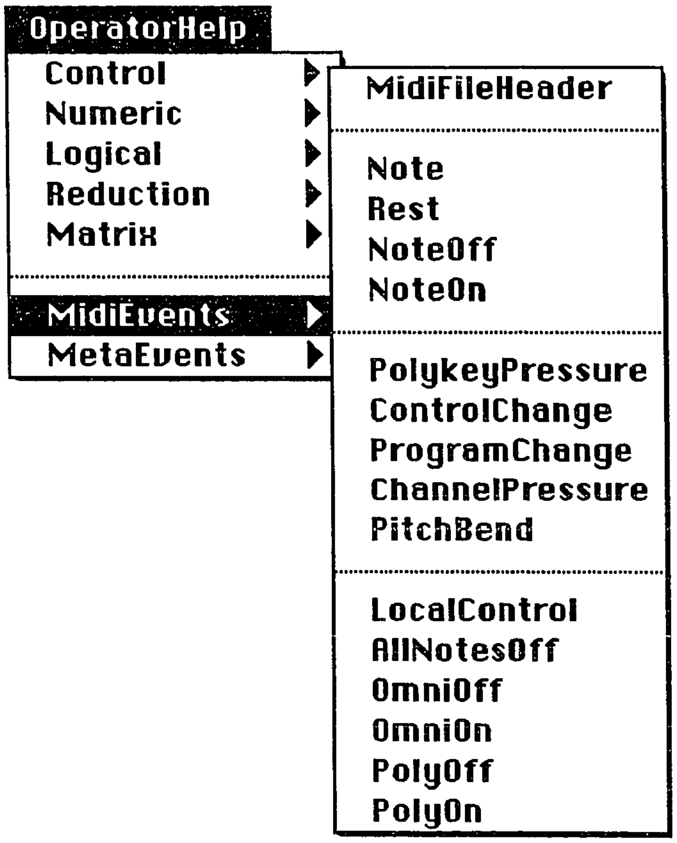


Illustration 3.6.7   MIDI Event Functions Submenu

The **MetaEvents** submenu items are functions that generate MIDI meta-events. Meta-events specify information about a MIDI file or an individual track within a file. These events are not time tagged and are usually included early on in a file.

The **MetaEvents** submenu is displayed as follows:

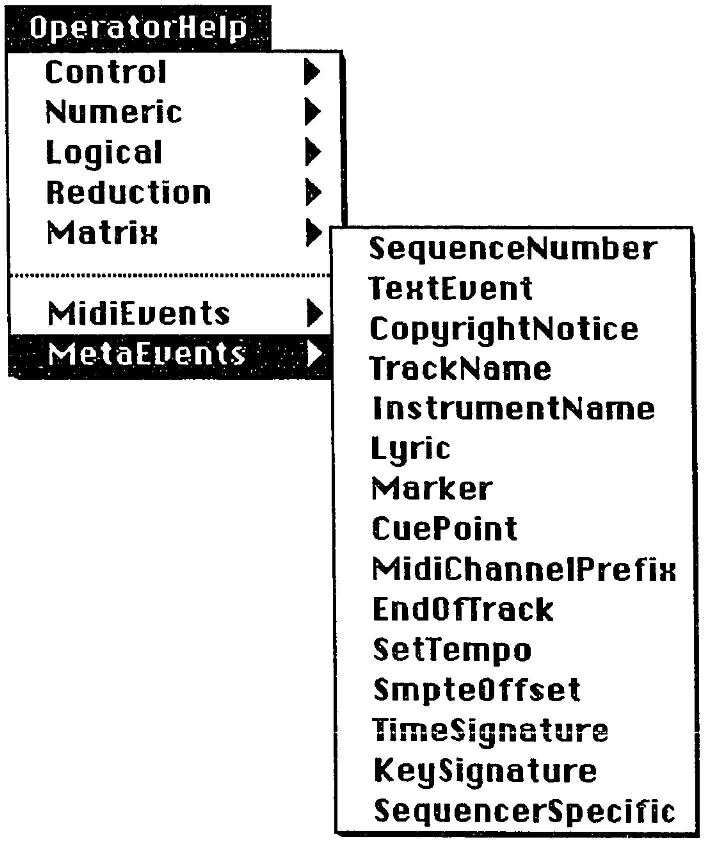


Illustration 3.6.8 MIDI Meta-Event Functions Submenu

COMMAND LINE DIALOG

The command line dialog appears just below the menu bar. It's purpose is for the entry of MCL commands and expressions. Commands available from the **Commands** menu are also available from the command line. The command line is a not a modal dialog and will allow a user to suspend data entry, go off and execute another task, and to later resume entry into the command line. Modal dialog windows require users to select an **Ok** or **Done** button before they are free to do anything else, which is undesirable if they need to look up the syntax of a particular operator or function, or to perform some other task.

The command line dialog displays two different prompts. The **?** prompt is displayed normally unless the user has entered an expression that does not have a matching right parenthesis for each left parenthesis. In this case the **..)** command line prompt will be displayed until all left parenthesis have been balanced. The command line dialog is normally displayed below the menu bar as follows:

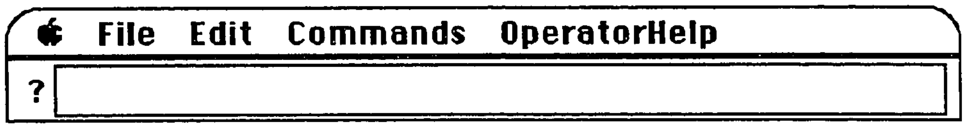


Illustration 3.7 Command Line Dialog Window

OUTPUT STREAM WINDOW

The output stream window is where all command line entries and resulting interpreter output is directed. The window is fully accessible to the text editor which is invoked by clicking within the body of the window. All interpreter output is commented to insure that any output stream saved as an MCL source code file will load without errors. It is, however, aesthetically advantageous to delete commented interpreter output before saving the output stream in a file. Otherwise when MCL loads the source code, it will display its normal commented response and then display the commented response from the previous session.

By directing command line entries and interpreter response to the output stream, users are able to work interactively knowing that they always retain the option to save their session in a log file, eliminating the edit, compile, debug, and edit loop that aggravates many programmers. In MCL users are able to test as they go, knowing that they may later extract good code from the output stream and save it in a source code file for future use.

Upon entry into MCL the user is in command line mode and the output stream window is displayed as follows:

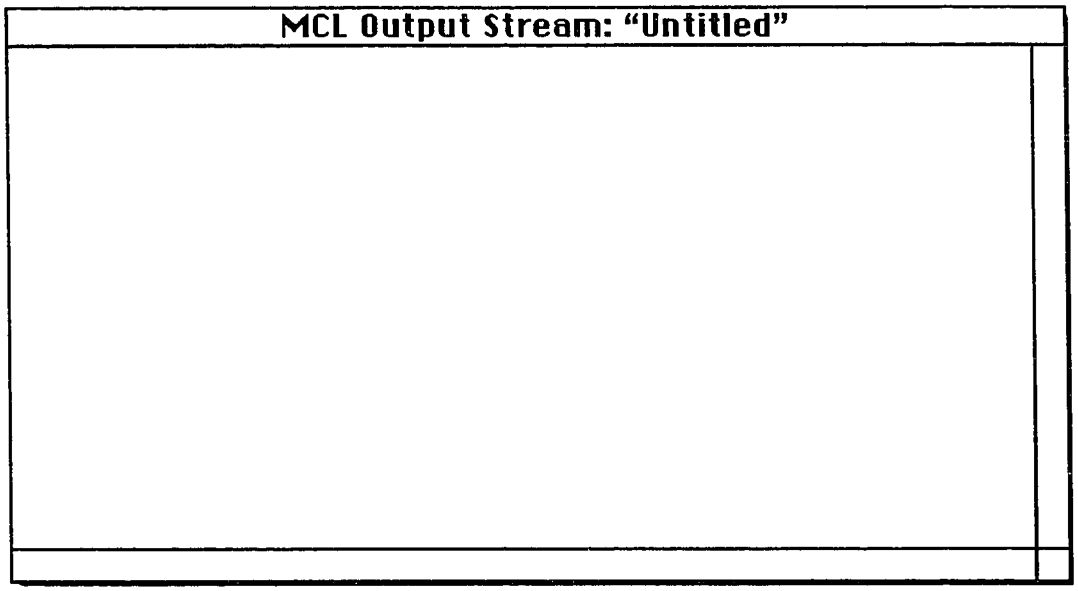


Illustration 3.8.1 Output Stream Window — Editor Not Invoked

The editor must be activated before text can be entered into the output stream. If the title bar is not highlighted (as displayed in the previous illustration) the text editor must be activated by clicking the mouse button within the body of the window. Once the editor is activated, the window will be highlighted with parallel horizontal lines. Now the editor is ready for users to start typing and will appear as follows:

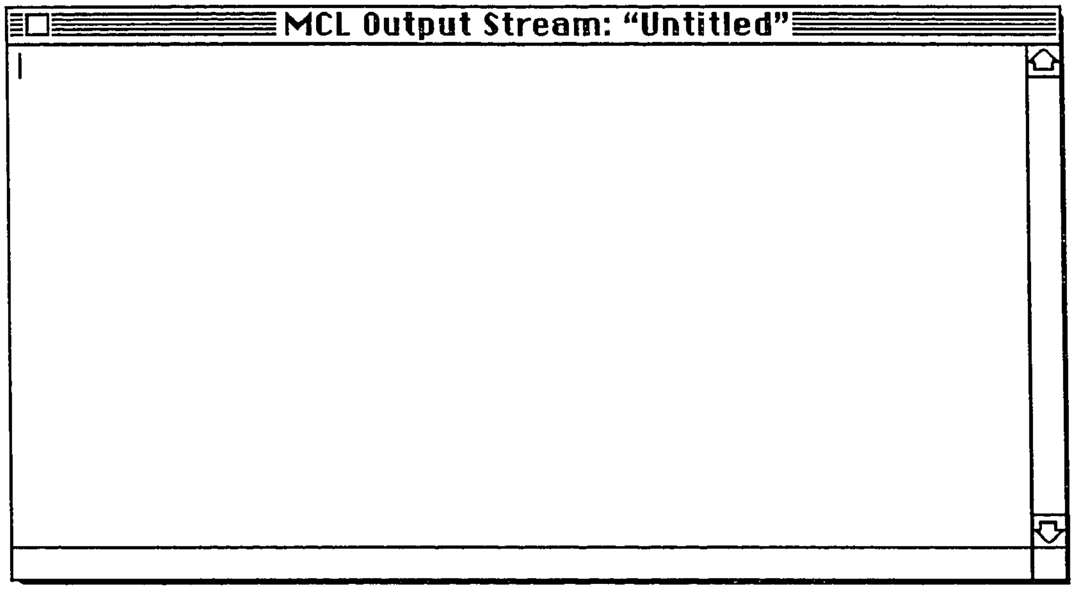


Illustration 3.8.2 Output Stream Window — Editor Invoked

EXTRACTING OPERATOR HELP TEXT

The following brief tutorial demonstrates how to extract help text for a particular operator or function and insert it into an edit window. In this example help text for the logical **and** operator is extracted. This can be accomplished by executing the following steps:

1) Drag the mouse to the menu bar and position the cursor on **OperatorHelp**. Depress the mouse button and the operator help menu will appear.

2) While keeping the mouse button depressed, drag the cursor to the **Logical** menu item and pause. The **Logical** submenu will pop up to the right of the operator help menu. (Remember to keep the mouse button depressed.)

3) Drag the cursor to the **and** item and pause. At this time the cursor, **OperatorHelp** menu and **Logical** submenu will be displayed as follows:

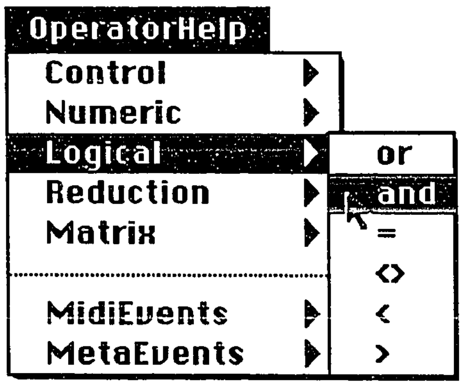
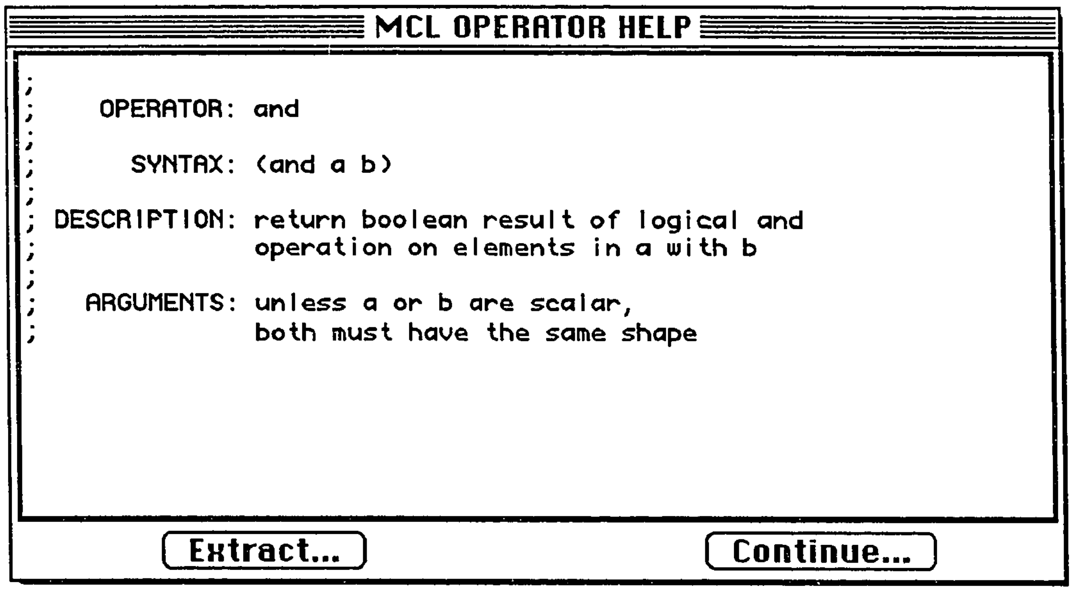


Illustration 3.10.1 Selecting operator from OperatorHelp Menu

4) Release the mouse button and the following help window will be displayed:

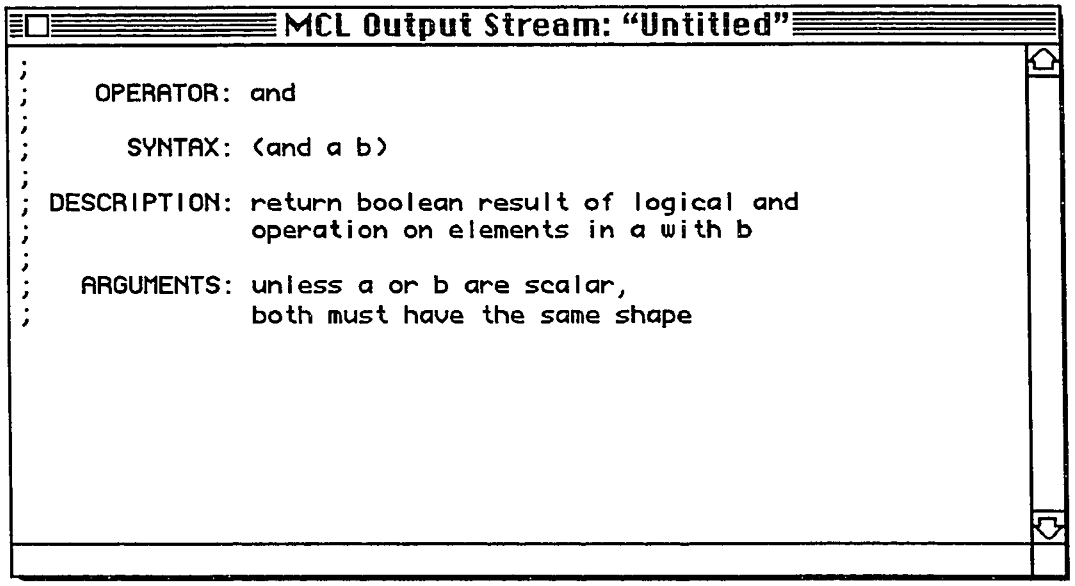


```
;
;     OPERATOR: and
;
;       SYNTAX: (and a b)
;
; DESCRIPTION: return boolean result of logical and
;              operation on elements in a with b
;
;   ARGUMENTS: unless a or b are scalar,
;              both must have the same shape
```

Illustration 3.10.2 Operator Help Window

5) Notice that there are two buttons at the bottom of the window labeled **ExtractText** and **Ok**. Select the **ExtractText** button and help text will be output to the edit window.

6) Select the **Ok** button to exit the dialog window. The edit window will now contain help text for the **and** operator as follows:



```
;
;    OPERATOR: and
;
;      SYNTAX: (and a b)
;
; DESCRIPTION: return boolean result of logical and
;              operation on elements in a with b
;
;   ARGUMENTS: unless a or b are scalar,
;              both must have the same shape
```

Illustration 3.10.3 Extracted Help Text in Edit Window

CHAPTER 4

MCL OPERATORS AND MIDI EVENT FUNCTIONS

MCL operators are classified into two major categories. Those that control program flow and those that manipulate values whether scalar, vector, or matrix. Control operators are primarily used to allocate storage, initialize variables, control branching, and to manage looping during program execution. Value operators are further classified into four categories which include numeric, logical, reduction and matrix operators. Numeric and logical operators perform operations on multiple variables; reduction operators process individual elements within a vector or matrix; and matrix operators perform miscellaneous operations on vectors and matrices. The following lists of MCL operator names are designated as key words and can not be used as function or variable names:

CONTROL OPERATORS

if	repeat	loop	while
set	set[]	set[][]	begin

NUMERIC OPERATORS

+	-	*	/
max	min	mod	

LOGICAL OPERATORS

or	and	=
<>	<	>

REDUCTION OPERATORS

+/	-/	*/	//
max/	min/	or/	and/

MATRIX OPERATORS

compress	shape	ravel	restruct
cat	indx	rand	trans
[]	[][]	print	

Illustration 4.1 Table of Control and Value Operators

MIDI EVENT FUNCTIONS

MIDI functions are classified as channel event or meta-event functions. Channel event functions generate time tagged events (meaning that a specified delay occurs in real time before the event is executed). Meta-event operators are not time tagged and specify such miscellaneous information about a MIDI file as the sequence number, end of a track, lyrics, or even copyright notice. They usually are located near the beginning of a MIDI file so that application programs will know from early on that meta-events are present in the file. It is not required that MIDI software handle meta-events and it is up to the software to handle them. MIDI files are structured so that MIDI application programs can either choose to ignore them or to use them as specified. The following lists of MCL MIDI function names are designated as key words and can not be used as function or variable names:

CHANNEL EVENTS

MidiFileHeader	**Note**
Rest	**NoteOff**
NoteOn	**PolykeyPressure**
ControlChange	**ProgramChange**
ChannelPressure	**PitchBend**
LocalControl	**AllNotesOff**
OmniOff	**OmniOn**
PolyOff	**PolyOn**

META-EVENTS

SequenceNumber	**TextEvent**
CopyrightNotice	**TrackName**
InstrumentName	**Lyric**
Marker	**CuePoint**
MidiChannelPrefix	**EndOfTrack**
SetTempo	**SmpteOffset**
TimeSignature	**KeySignature**
SequencerSpecific	

Illustration 4.2 Table of MIDI Functions

CHAPTER 5

GLOSSARY OF OPERATORS

DEFINE OPERATOR

OPERATOR: define

SYNTAX:

```
(define myfunc (a1 a2)
        (begin
              (e1 a1)
              (e2 a2)
              (e3 a1 a2)
        )
)
```

DESCRIPTION: define a user function named **myfunc** with arguments **a1** & **a2**, that executes expressions **e1**, **e2**, & **e3** within the **begin** block

CONTROL OPERATORS

OPERATOR: if

SYNTAX: (if (e1) (e2)) or (if (e1) (e2) (e3))

DESCRIPTION: logical if operation

If expression **e1** is true
then do expression **e2**

If expression **e1** is true
then do expression **e2**
else do expression **e3**

ARGUMENTS:

e1:	expression returns 1 for true or 0 for false
e2:	expression e2 is evaluated if e1 is true
e3:	expression e3 is evaluated if e1 is false

OPERATOR: repeat

SYNTAX: (repeat n (e))

DESCRIPTION: **repeat** expression **e**, argument **n** number of times

ARGUMENTS:

n:	number of times to repeat expression e1
e:	expression to repeat n times

OPERATOR:	**loop**
SYNTAX:	(loop min max step (e))

DESCRIPTION:	**loop** from **min** to **max**, increment with **step**:
	    evaluate expression **e**
	end loop

ARGUMENTS:	**min**: starting index **max**: ending index **step**: index step **e**: expression evaluated during loop
OPERATOR:	**while**
SYNTAX:	(while (e1) (e2))
DESCRIPTION:	**while** expression **e1** is true do expression **e2**
ARGUMENTS:	**e1**: expression returns 1 for true or 0 for false **e2**: expression is evaluated while e1 is true
OPERATOR:	**begin**
SYNTAX:	(begin (e1) (e2) ... (eN))
DESCRIPTION:	**begin** a block of expressions **e1** thru **eN**
ARGUMENTS:	**e1**...**eN**: expressions to be blocked
OPERATOR:	**set**
SYNTAX:	(set a (e))
DESCRIPTION:	**set** creates variable **a** and initializes it to the value returned from expression **e**
ARGUMENTS:	**a**: variable name **e**: expression returning the value

OPERATOR:	**set[]**
SYNTAX:	(set[] a s (e))
DESCRIPTION:	**set[]** initializes variable **a**, singly subscripted by value **s**, with value returned from expression **e**
ARGUMENTS:	**a**: variable being subscripted
	s: subscript one
	e: expression returning value

OPERATOR:	**set[][]**
SYNTAX:	(set[][] a s1 s2 (e))
DESCRIPTION:	**set[][]** initializes variable **a**, doubly subscripted by values **s1** and **s2**, with value returned from expression **e**
ARGUMENTS:	**a**: variable being subscripted
	s1: subscript one
	s2: subscript two
	e: expression returning values

NUMERIC OPERATORS

OPERATOR:	**+**
SYNTAX:	(+ a b)
DESCRIPTION:	return result of argument **b** added to argument **a**

OPERATOR:	**-**
SYNTAX:	(- a b)
DESCRIPTION:	return result of argument **b** subtracted argument value **a**

OPERATOR:	*****
SYNTAX:	(* a b)
DESCRIPTION:	return result of argument **a** multiplied by argument **b**

OPERATOR:	**/**
SYNTAX:	(/ a b)
DESCRIPTION:	return result of argument **a** divided by argument **b**
OPERATOR:	**max**
SYNTAX:	(max a b)
DESCRIPTION:	return value with maximum elements in arguments **a** and **b**
OPERATOR:	**min**
SYNTAX:	(min a b)
DESCRIPTION:	return value with minimum elements in arguments **a** and **b**
OPERATOR:	**mod**
SYNTAX:	(mod a b)
DESCRIPTION:	return modula of argument **a** divided by argument **b**

LOGICAL OPERATORS

OPERATOR:	**or**
SYNTAX:	(or a b)
DESCRIPTION:	return boolean result of logical **or** for elements in **a** with **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape
OPERATOR:	**and**
SYNTAX:	(and a b)
DESCRIPTION:	return boolean result of logical **and** for elements in **a** with **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape

OPERATOR:	**=**
SYNTAX:	(= a b)
DESCRIPTION:	return boolean result of **a** equals **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape
OPERATOR:	**<>**
SYNTAX:	(<> a b)
DESCRIPTION:	return boolean result of **a** not equal to **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape
OPERATOR:	**<**
SYNTAX:	(< a b)
DESCRIPTION:	return boolean result of **a** less than **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape
OPERATOR:	**>**
SYNTAX:	(> a b)
DESCRIPTION:	return boolean result of **a** greater than **b**
ARGUMENTS:	unless **a** or **b** are scalar, both must have the same shape

REDUCTION OPERATORS

OPERATOR:	**+/**	
SYNTAX:	(+/ a)	
DESCRIPTION:	return the sum of the elements in argument **a**	
ARGUMENTS:	if **a** is scalar:	do nothing but return the argument
	if **a** is vector:	(+ v1 (+ v2 (... (+ vN-1 vN))))
	if **a** is matrix:	do vector operation on each row

OPERATOR:	**-/**
SYNTAX:	(-/ a)
DESCRIPTION:	return the reduction subtraction of elements in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if **a** is vector: (- v1 (- v2 (... (- vN-1 vN)))) if **a** is matrix: do vector operation on each row
OPERATOR:	*** /**
SYNTAX:	(*/ a)
DESCRIPTION:	return the product of elements in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if **a** is vector: (* v1 (* v2 (... (* vN-1 vN)))) if **a** is matrix: do vector operation on each row
OPERATOR:	**//**
SYNTAX:	(// a)
DESCRIPTION:	return the reduction division of elements in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if **a** is vector: (/ v1 (/ v2 (... (/ vN-1 vN)))) if **a** is matrix: do vector operation on each row
OPERATOR:	**max/**
SYNTAX:	(max/ a)
DESCRIPTION:	return the maximum element in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if **a** is vector: (max v1 (max v2 (... (max vN-1 vN)))) if **a** is matrix: do vector operation on each row

OPERATOR:	**min/**
SYNTAX:	(min/ a)
DESCRIPTION:	return the minimum element in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if a is vector: (min v1 (min v2 (... (min vN-1 vN)))) if **a** is matrix: do vector operation on each row

OPERATOR:	**or/**
SYNTAX:	(or/ a)
DESCRIPTION:	return the logical **or** of elements in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if a is vector: (or v1 (or v2 (... (or vN-1 vN)))) if **a** is matrix: do vector operation on each row

OPERATOR:	**and/**
SYNTAX:	(and/ a)
DESCRIPTION:	return the logical **and** of elements in argument **a**
ARGUMENTS:	if **a** is scalar: do nothing but return the argument if **a** is vector: (and v1 (and v2 (... (and vN-1 vN)))) if **a** is matrix: do vector operation on each row

MATRIX OPERATORS

OPERATOR:	**compress**
SYNTAX:	(compress a b)
DESCRIPTION:	**compress** argument **b** by boolean argument **a**
ARGUMENTS:	**a**: boolean compression vector **b**: vector or matrix to be compressed

RETURN:	if **b** is vector: return compressed vector
	if **b** is matrix: return compressed matrix

OPERATOR:	**shape**
SYNTAX:	(shape a)
DESCRIPTION:	returns the shape vector of argument **a**
RETURN:	if **a** is scalar: shape vector length is one if **a** is vector: shape vector length is one if **a** is matrix: shape vector length is two

OPERATOR:	**ravel**
SYNTAX:	(ravel a)
DESCRIPTION:	return vector with elements in argument **a**, regardless of shape

OPERATOR:	**restruct**
SYNTAX:	(restruct s a)
DESCRIPTION:	restructure argument **a** to shape specified in argument **s**
ARGUMENTS:	**s**: shape vector **a**: value being restructured
RETURN:	value of shape **s** with elements from **a**

OPERATOR:	**cat**
SYNTAX:	(cat a b)
DESCRIPTION:	catenation of elements from argument **a** and argument **b**
RETURN:	vector of catenation of (ravel a) and (ravel b)

OPERATOR:	**indx**
SYNTAX:	(indx n)
DESCRIPTION:	return a vector of **n** index values: 1,2,3, n

OPERATOR:	**rand**
SYNTAX:	(rand n min max)
DESCRIPTION:	return a vector of **n** random values ranging from **min** to **max**

OPERATOR:	**trans**
SYNTAX:	(trans a)
DESCRIPTION:	transpose matrix **a**
RETURN:	transposed matrix (scalars & vectors are returned unchanged)

OPERATOR:	**[]**
SYNTAX:	([] a s)
DESCRIPTION:	single subscript argument **a** with argument **s**
ARGUMENTS:	**a**: vector or matrix **s**: subscript vector/scalar
RETURN:	subscripted portion of argument **a**, returned with same rank as **s**

OPERATOR:	**[][]**
SYNTAX:	([][] a s1 s2)
DESCRIPTION:	double subscript argument **a** with arguments **s1** and **s2**
ARGUMENTS:	**a**: matrix **s1**: subscript vector/scalar 1 **s2**: subscript vector/scalar 2
RETURN:	subscripted portion of argument **a**, returned as a matrix

OPERATOR:	**print**
SYNTAX:	(print a)
DESCRIPTION:	print elements in argument **a**

CHAPTER 6

GLOSSARY OF MIDI FUNCTIONS

CHANNEL EVENT FUNCTIONS

FUNCTION: MidiFileHeader

SYNTAX: (MidiFileHeader format ntracks nticks)

DESCRIPTION: This MIDI function specifies MIDI file header parameters that control MIDI file format, number of tracks in file, and the number of MIDI ticks per quarter note.

ARGUMENTS:

format:	MIDI file format number, i.e., 0, 1, or 2
ntracks:	number of tracks in the MIDI file
nticks:	number of ticks per quarter note

FUNCTION: Note

SYNTAX: (Note dT ch k a r)

DESCRIPTION: **Note** performs the following for channel **ch**:

1) **NoteOn** : 0 delta time,
key **k**,
attack velocity **a**

2) **NoteOff**: **dT** delta time,
key **k**,
release velocity **a**

ARGUMENTS:

dT:	delta time to delay before executing note off event
ch:	MIDI channel identifier (0-15)
k:	key number
a:	attack velocity
r:	release velocity

FUNCTION: Rest

SYNTAX: (Rest dT)

DESCRIPTION: increment system rest duration for added delay to next dT > 0

ARGUMENTS: **dT**: delta time to delay before executing note off event

FUNCTION: **NoteOff**

SYNTAX: (NoteOff dT ch k v)

DESCRIPTION: **NoteOff** specifies the release of key **k** with velocity **v** on channel **ch**

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
k:	key number
v:	release velocity

FUNCTION: **NoteOn**

SYNTAX: (NoteOn dT ch k v)

DESCRIPTION: **NoteOn** specifies the striking of key **k** with velocity **v** on channel **ch**

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
k:	key number
v:	attack velocity

FUNCTION: **PolykeyPressure**

SYNTAX: (PolykeyPressure dT ch k a)

DESCRIPTION: **PolykeyPressure** sets after touch **a** for key **k** on channel **ch**

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
k:	key number
a:	after touch

FUNCTION: **ControlChange**

SYNTAX: (ControlChange dT ch n v)

DESCRIPTION: **ControlChange** sets control number **n** and control value **v** for channel **ch**

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
n:	control number
v:	control value

FUNCTION:	**ProgramChange**
SYNTAX:	(ProgramChange dT ch n)
DESCRIPTION:	**ProgramChange** selects program number **n** for channel **ch** to select new instrument sound bank
ARGUMENTS:	**dT**: delta time to delay before executing event **ch**: M!DI channel identifier (0-15) **n**: program number

FUNCTION:	**ChannelPressure**
SYNTAX:	(ChannelPressure dT ch p)
DESCRIPTION:	**ChannelPressure** specifies pressure value **p** for channel **ch**
ARGUMENTS:	**dT**: delta time to delay before executing event **ch**: MIDI channel identifier (0-15) **p**: pressure value

FUNCTION:	**PitchBend**
SYNTAX:	(PitchBend dT ch p)
DESCRIPTION:	**PitchBend** specifies pitch bend value **p** for channel **ch**
ARGUMENTS:	**dT**: delta time to delay before executing event **ch**: MIDI channel identifier (0-15) **p**: pitch bend value

FUNCTION:	**LocalControl**
SYNTAX:	(LocalControl dT ch on)
DESCRIPTION:	**LocalControl** toggles local control on or off for channel **ch**
ARGUMENTS:	**dT**: delta time to delay before executing event **ch**: MIDI channel identifier (0-15) **on**: flag to toggle local control on or off

FUNCTION:	**AllNotesOff**
SYNTAX:	(AllNotesOff dT ch)
DESCRIPTION:	**AllNotesOff** performs a **NoteOff** for all keys for channel **ch**
ARGUMENTS:	**dT**: delta time to delay before executing event **ch**: MIDI channel identifier (0-15)

FUNCTION: **OmniOff**

SYNTAX: (OmniOff dT ch)

DESCRIPTION: **OmniOn** toggles omni mode off...

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)

FUNCTION: **OmniOn**

SYNTAX: (OmniOn dT ch)

DESCRIPTION: **OmniOn** toggles omni mode on...

When Omni is set on, an instrument will ignore the channel identifier and will attempt to respond to all incoming messages.

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)

FUNCTION: **PolyOff**

SYNTAX: (PolyOff dT ch n)

DESCRIPTION: **PolyOff** toggles poly mode off

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
n:	number of channels

FUNCTION: **PolyOn**

SYNTAX: (PolyOn dT ch)

DESCRIPTION: **PolyOn** toggles poly mode on

ARGUMENTS:

dT:	delta time to delay before executing event
ch:	MIDI channel identifier (0-15)
n:	number of channels

META-EVENT FUNCTIONS

This section covers the MIDI functions that generate meta-events. Minimal detail is provided here. See appendix A for further details.

FUNCTION:	**SequenceNumber**
SYNTAX:	(SequenceNumber ssss)
DESCRIPTION:	**SequenceNumber** sets the sequence number for a particular section in a MIDI file and must be called before any timed events
ARGUMENTS:	**ssss:** sequence number
FUNCTION:	**TextEvent**
SYNTAX:	(TextEvent text)
DESCRIPTION:	**TextEvent** specifies track name, lyrics, cue points, etc.
ARGUMENTS:	**text:** character vector
FUNCTION:	**CopyrightNotice**
SYNTAX:	(CopyrightNotice text)
DESCRIPTION:	**CopyrightNotice** specifies composer's copyright notice text and must be the 1st event in the 1st track at delta time 0
ARGUMENTS:	**text:** character vector
FUNCTION:	**TrackName**
SYNTAX:	(TrackName text)
DESCRIPTION:	**TrackName** specifies track name, unless it is the 1st track in a format 1 file, when it specifies the name of a sequence
ARGUMENTS:	**text:** character vector
FUNCTION:	**InstrumentName**
SYNTAX:	(InstrumentName text)
DESCRIPTION:	**InstrumentName** specifies type of instrument used in a track
ARGUMENTS:	**text:** character vector

FUNCTION:	**Lyric**
SYNTAX:	(Lyric text)
DESCRIPTION:	**Lyric** specifies a lyric (usually one lyric event per syllable)
ARGUMENTS:	**text**: character vector

FUNCTION:	**Marker**
SYNTAX:	(Marker text)
DESCRIPTION:	**Marker** specified a rehearsal letter, or section name, etc.
ARGUMENTS:	**text**: character vector
EXAMPLE:	(Marker "(First Verse))

FUNCTION:	**CuePoint**
SYNTAX:	(CuePoint text)
DESCRIPTION:	**CuePoint** specifies an staged event in a film score
ARGUMENTS:	**text**: character vector

FUNCTION:	**MidiChannelPrefix**
SYNTAX:	(MidiChannelPrefix cc)
DESCRIPTION:	**MidiChannelPrefix** indicates that channel **cc** is to be associated with any subsequent meta-events and system exclusive events and is to remain effective until a channel event is specified
ARGUMENTS:	**cc**: channel number

FUNCTION:	**EndOfTrack**
SYNTAX:	(EndOfTrack n)
DESCRIPTION:	**EndOfTrack** specifies an exact end of track **n**
ARGUMENTS:	**n**: track number

FUNCTION: SetTempo

SYNTAX:	(SetTempo tttttt)
DESCRIPTION:	**SetTempo** as specified by argument **tttttt** for a tempo change
ARGUMENTS:	**tttttt**: tempo value (μ seconds per quarter note)

FUNCTION: SmpteOffset

SYNTAX:	(SmpteOffset hr mn se fr ff)
DESCRIPTION:	**SmpteOffset** sets a track's SMPTE starting time

ARGUMENTS:

hr:	hours
mn:	minutes
se:	seconds
fr:	frame
ff:	fraction frames

FUNCTION: TimeSignature

SYNTAX:	(TimeSignature nn dd cc bb)
DESCRIPTION:	**TimeSignature** sets the time signature

ARGUMENTS:

nn:	numerator
dd:	denominator expressed as a power of 2
cc:	clocks per quarter (dotted quarter in 6/8, etc.)
bb:	number of 32nd notes per MIDI quarter note

EXAMPLE: (TimeSignature 6 3 36 8)

nn:	6	numerator

dd:	3	denominator = 2^3
		meter = 6 / 2^3 = 6/8

cc:	36	MIDI clocks per dotted quarter note
bb:	8	number of 32nd notes per MIDI quarter note.

FUNCTION:	**KeySignature**
SYNTAX:	(KeySignature sf mi)
DESCRIPTION:	**KeySignature** set the key signature
ARGUMENTS:	**sf**: number of sharps or flats **mi**: flag sets major or minor key if sf is negative then set flats else set sharps

EXAMPLE:	sf = -7: 7 flats
	sf = 0: key of C
	sf = 7: 7 sharps
	
	mi = 0: major key
	mi = 1: minor key

FUNCTION:	**SequencerSpecific**
SYNTAX:	(SequencerSpecific data)
DESCRIPTION:	**SequencerSpecific** specifies system exclusive information
ARGUMENTS:	**data**: vector (the 1st element must specify manufacturer ID)

CHAPTER 7

EXAMPLE MCL APPLICATIONS

VALUE OPERATORS AND FUNCTIONS

The following listing demonstrates various MCL operators:[15]

CREATING VARIABLES

```
(set V '2 4 6')              ;3 elements, vector [2 4 6]

      2    4    6

(set C '1 0')                ;2 elements, boolean vector

      1    0

(set D '1 0 1')              ;3 elements, boolean vector

      1    0    1

(set A                       ;A is a...
      (restruct              ;     restructured...
            '2 3'            ;     2 by 3 array...
            (indx 6)         ;     of vector [1 2 3 4 5 6]
      )
)

      1    2    3
      4    5    6
```

15 Actual MCL code appears in bold type while comments are to the right delimited with a semicolon. Resulting output from each expression is listed below.

NUMERIC AND LOGICAL OPERATORS

```
(* A A)                 ;matrix multiply A by A

      1    4    9
     16   25   36

(- V 1)                 ;subtract scalar 1 from vector V

      1    3    5

(> A 4)                 ;returns boolean array, set 1 when Aij<4

      0    0    0
      0    1    1

(+/ V)                  ;reduction sum of vector V

     12

(max/ A)                ;reduction max of array  A

      3    6
```

MATRIX OPERATORS

```
(compress D V)          ;compress vector V by boolean vector D

      2    6

(compress C A)          ;compress array A by boolean vector C

      1    2    3

(shape A)               ;get the shape of array A

      2    3

(ravel A)               ;ravel array into a vector of nr*nc length

      1    2    3    4    5    6
```

```
(restruct (shape A) V))           ;restructure V to shape of A

      2   4   6
      2   4   6

(restruct (shape V) C))           ;restructure C to shape of V

      1   0   1

(cat A C)                         ;catenate vector C to array A

      1   2   3   4   5   6   1   0

(indx 5)                          ;generate vector from 1 to 5

      1   2   3   4   5

(trans A)                         ;transpose array A

      1   4
      2   5
      3   6

([] V (indx 2))                   ;subscript V by 1,2

      2   4

([] A 1)                          ;subscript A by 1 to get row 1

      1   2   3

([] (trans A)(indx 2))            ;subscript trans A to get rows
1 & 2

      1   4
      2   5
```

CONTROL OPERATORS

```
(if 1 (print 11))                 ;test is true, print 11

     11

(if 0 (print 11))                 ;test is false, don't print 11

      1

(if 1 (print 11)(print 22))       ;test is true, print 11

     11

(if 0 (print 11)(print 22))       ;test is false, print 22

     22
```

```
(repeat 3 (print 33))              ;print expression 3 times:

     33
     33
     33

(loop 2 6 (print 55))              ;loop over print from 2 to 6

     55
     55
     55
     55
     55

(set i 1)                          ;set index to one

      1

(while (< i 7)                     ;while index < 7 do:
      (begin                       ;     begin expression block:
            (print i)              ;         print index i
            (set i (+ i 1))        ;         increment index i
      )                            ;     end block
)                                  ;end while loop

      1
      2
      3
      4
      5
      6
```

INVERSION MATRIX FUNCTION

The following listing of the user defined function *invmat* presents a musical utility that converts a twelve tone serial row into a twelve by twelve inversion matrix:

```
(set indx2to12 (+ 1 (indx 11)))

(define invmat (toneRow)
     (begin
          (set      resultMatrix (restruct '12 12' 0))
          (set[]    resultMatrix 1 toneRow)
          (set      inversionVector
                    ([] (- 12 toneRow) indx2to12)
          )
          (set[][]  resultMatrix
                    indx2to12 1 inversionVector
          )
          (set      iindex 2)
          (while (< iindex 13)
               (begin
                    (set iindexLast (- iindex 1))
                    (set interval
                    (- ([][] resultMatrix iindex 1)
                    ([][] resultMatrix iindexLast 1)))
                    (set inversionVector
                         (+   ([][]   resultMatrix
                                      iindexLast
                                      indx2to12
                              )
                              interval
                         )
                    )
                    (set[][]  resultMatrix
                              iindex indx2to12
                              inversionVector
                    )
                    (set      iindex  (+ iindex 1))
               )
          )
          (set resultMatrix
               (+ resultMatrix
                    (* (< resultMatrix  1) 12)
               )
          )
          (set resultMatrix
               (mod resultMatrix
                    (* (> resultMatrix 12) 12)
               )
          )
          resultMatrix
     )
)
```

Once the tone row is created it can be passed to the invmat function that will return a 12 by 12 matrix of inversion rows as is demonstrated in the following:

```
(set r '12 11 1 2 10 9 3 4 8 7 5 6')

     12   11    1    2    0    9    3    4    8     7    5    6

(set m (invmat r))

     12   11    1    2   10    9    3    4    8     7    5    6
      1   12    2    3   11   10    4    5    9     8    6    7
     11   10   12    1    9    8    2    3    7     6    4    5
     10    9   11   12    8    7    1    2    6     5    3    4
      2    1    3    4   12   11    5    6   10     9    7    8
      3    2    4    5    1   12    6    7   11    10    8    9
      9    8   10   11    7    6   12    1    5     4    2    3
      8    7    9   10    6    5   11   12    4     3    1    2
      4    3    5    6    2    1    7    8   12    11    9   10
      5    4    6    7    3    2    8    9    1    12   10   11
      7    6    8    9    5    4   10   11    3     2   12    1
      6    5    7    8    4    3    9   10    2     1   11   12
```

MUSICAL EXAMPLE

This presents a musical example that generates 32 measures of random notes for four parts according to the following grand design:

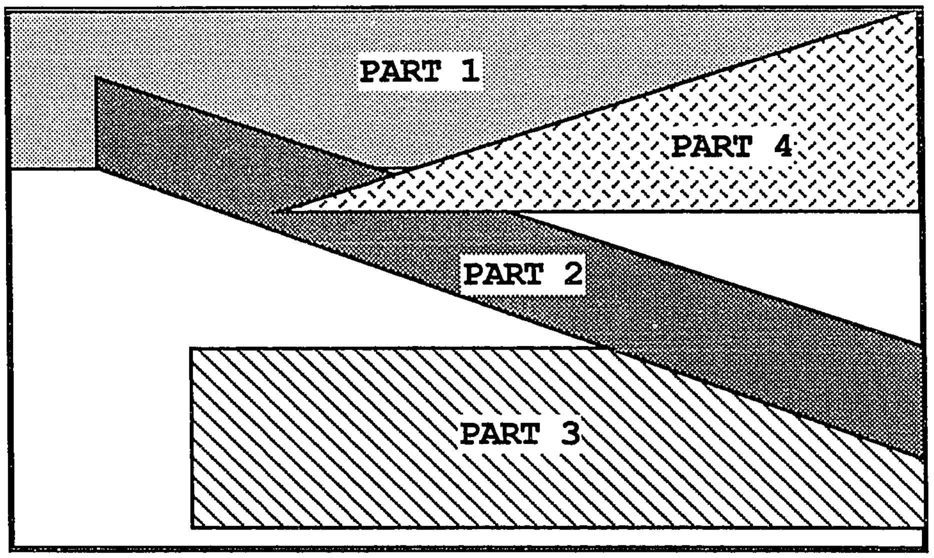


Illustration 7.1   Design for Random Music Example

This short work was designed for a Korg Symphony Orchestra Module with the first part alternating between the Strings 2 and Chorus tones. Part two alternates between the Strings 1 and Strings 2 voices. Part three alternates between the Organ 1 and Organ 2 voices. Part four alternates between the Brass, Bass/Guitar/Drums, and Organ 1 voices.

MCL source code for the random example is stored in five files. The first file stores the first function *prep* which is used to prepare the MIDI file header. The argument specifies the number of tracks the user wants to generate. One is added to this number to account for the tempos track. The initial tempo is at 120 beats per minute. The following is a listing of the prep function and functions that generate each part:

```
(define prep(n) ;n is the number of tracks
     (begin
          (set n2 (+ n 1))
          (MidiFileHeader 1 n2 96 )
          (CopyrightNotice
               "(C) 1991 Marshall Earl Edwards"
          )
          (TrackName "Random Example 1")
          (SetTempo               500000 )
          (EndOfTrack              WHOLE )
     )
)
```

The remaining four files store the functions that generate data for each part. The following pages present listings for each function:

```
(define part1()
     (begin
          (set j 1)
          (repeat 4 ;measures 01-04
               (set p (rand 16 C4        C7        ))
               (set d (rand 16 THIRTYSEC SIXTEENTH))
               (set i 1)
               (repeat 16
                    (set j (* j -1))
                    (if (< j 0) (set ch 3) (set ch 2))
                    (Note ([] d i) ch ([] p i) 40 40)
                    (set i (+ i 1))
               )
               (Rest (- WHOLE (+/ d)))
          )
          (repeat 4 ;measures 05-08
               (set p (rand 8 C4         C7    ))
               (set d (rand 8 SIXTEENTH EIGHTH))
               (set i 1)
               (repeat 8
                    (set j (* j -1))
                    (if (< j 0) (set ch 3) (set ch 2))
                    (Note ([] d i) ch ([] p i) 40 40)
                    (set i (+ i 1))
               )
               (Rest (- WHOLE (+/ d)))
          )
          (repeat 4 ;measures 08-11
               (set p (rand 4 C4      C7      ))
               (set d (rand 4 EIGHTH QUARTER))
               (set i 1)
               (repeat 4
                    (set j (* j -1))
                    (if (< j 0) (set ch 3) (set ch 2))
                    (Note ([] d i) ch ([] p i) 40 40)
                    (set i (+ i 1))
               )
               (Rest (- WHOLE (+/ d)))
          )
          (repeat 20 ;measures 12-32
               (set p (rand 2 C4       C7   ))
               (set d (rand 2 QUARTER HALF))
               (set i 1)
               (repeat 2
                    (set j (* j -1))
                    (if (< j 0) (set ch 3) (set ch 2))
                    (Note ([] d i) ch ([] p i) 40 40)
                    (set i (+ i 1))
               )
               (Rest (- WHOLE (+/ d)))
          )
          (EndOfTrack 0)
     )
)
```

```
(define part2()
     (begin
          (print "PART2 - 4 MEASURES REST, 28 MEASURES")
          (Rest (* WHOLE 4))
          (set p1 C4)
          (set p2 C5)
          (set v1 60)
          (set v2 40)
          (set j   1)
          (repeat 28 ;measures
               (repeat 4 ;beats
                    (set i 1)
                    (set dT
                         (rand 4 THIRTYSEC SIXTEENTH))
                    (set p0 (rand 4 p1 p2))
                    (repeat 4 ;sixteenth notes per beat
                         (set j (* j -1))
                         (if (< j 0)
                              (Note   ([] dT i) 1
                                   (-  ([] p0 i) 12)
                                        v1 v1
                              )
                              (Note   ([] dT i) 2
                                      ([] p0 i) v2 v2
                              )
                         )
                    )
                    ;rest unused portion of beat
                    (Rest ( - QUARTER (+/ dT)))
               )
               (set v1 (+ v1 1))
               (set v2 (+ v2 1))
               (set p1 (- p1 1))
               (set p2 (- p2 1))
          )
          (EndOfTrack 0)
     )
)
```

```
(define part3()
     (begin
          (print "PART3 - 16 REST, 16 MEASURES")
          (Rest (* WHOLE 16))
          (set p1 G2)
          (set p2 C3)
          (set v1 10)
          (set v2 40)
          (set j   0)
          (repeat 16 ;measures
               (repeat 4 ;beats
                    (set i 1)
                    (set dT
                         (rand 4 THIRTYSEC SIXTEENTH)
                    )
                    (set p0 (rand 4 p1 p2))
                    (repeat 4 ;sixteenth notes per beat
                         (set j (+ j 1))
                         (if (= j 1)
                              (Note   ([] dT i) 6
                                      ([] p0 i) 1 1
                              )
                         )
                         (if (= j 2)
                              (Note   ([] dT i) 4
                                      ([] p0 i) v1 v1)
                         )
                         (if (= j 3)
                              (Note   ([] dT i) 5
                                      ([] p0 i) v2 v2
                              )
                         )
                         (if (> j 3)(set j 0))
                    )
                    (set p2 (+ p2 1))
                    (set v1 (+ v1 1))
                    (set v2 (+ v2 1))
                    ;rest unused portion of beat
                    (Rest ( - QUARTER (+/ dT)))
               )
          )
          (EndOfTrack 0)
     )
)
```

```
(define  part4()
      (begin
            (prep      3)
            (doPart4  6)
            (doPart4  7)
      )
)
(define  doPart4(ch)
      (begin
            (print "PART4 - 12 REST, 20 MEASURES")
            (Rest  (* WHOLE 12))
            (set totalTime  (* WHOLE 20))
            (set n 1)
            (set d (rand 1 SIXTEENTH WHOLE))
            (while  (> totalTime  (+/  d))
                  (begin
                        (set d  (cat  d
                              (rand  1  SIXTEENTH  WHOLE))
                        )
                        (set  n  (+  n  1))
                  )
            )
            (print  "Number  of  notes")(print  n)
            (set  p  (rand  n  C1  C4))
            (set  v  90)
            (set  i  1)
            (repeat  n  ;measures
                  (print  i)
                  (Note   ([]  d  i)  ch  ([]  p  i)  v  v)
                  (set  v  (-  v  1))
                  (set  i  (+  i  1))
            )
            (EndOfTrack  0)
      )
)
```

APPENDIX

STANDARD MIDI FILE FORMAT V 1.0

By Opcode for International MIDI Association (IMA)

Introduction

This document is a specification of MIDI Files. The purpose of MIDI Files is to provide a standard way of interchanging time-stamped MIDI data between different programs on the same or different computers. One of the primary design goals is compact representation, which makes it very appropriate for a disk-based file format, but which might make it inappropriate for storing in memory for quick access by a sequencer program. (It can be easily converted to a quickly-accessible format on the fly as files are read in or written out.) It is not intended to replace the normal file format of any program, though it could be used for this purpose if desired.

MIDI Files contain one or more MIDI streams, with time information for each event. Song, sequence, and track structures, tempo and time signature information, are all supported. Track names and other descriptive information may be stored with the MIDI data. This format supports multiple tracks and multiple sequences so that if the user of a program which supports multiple tracks intends to move a file to another one, this format can allow that to happen.

This spec defines the 8-bit binary data stream used in the file. The data can be stored in a binary file, nibbleized, 7-bit-ized for efficient MIDI transmission, converted to Hex ASCII, or translated symbolically to a printable text file. This spec addresses what's in the 8-bit stream. It does not address how a MIDI File will be transmitted over MIDI. (It is the current feeling of the MMA that a MIDI transmission protocol will be developed for files in general, and MIDI Files will use this scheme.)

Sequences, Tracks, Chunks: File Block Structure

Conventions

In this document, bit 0 means the least significant bit of a byte, and bit 7 is the most significant.

Some numbers in MIDI files are represented in a form called a variable-length quantity. These numbers are represented 7 bits per byte, most significant bits first. All bytes except the last have bit 7 set, and the last byte has bit 7 clear. If the number is between 0 and 127, it is thus represented exactly as one byte.

Here are some examples of numbers represented as variable-length quantities:

Number (hex)	Representation (hex)
00000000	00
00000040	40
0000007F	7F
00000080	81 00
00002000	C0 00
00003FFF	FF 7F
00004000	81 80 00
00100000	C0 80 00
001FFFFF	FF FF 7F
00200000	81 80 80 00
08000000	C0 80 80 00
0FFFFFFF	FF FF FF 7F

The largest number which is allowed is 0FFFFFFF so that the variable-length representation must fit in 32 bits in a routine to write variable-length numbers. Theoretically, larger numbers are possible, but 2 x 108 96ths of a beat at a fast tempo of 500 beats per minute is four days, long enough for any delta-time!

Files

To any file system, a MIDI File is simply a series of 8-bit bytes. On the Macintosh, this byte stream is stored in the data fork of a file (with file type 'Midi'), or on the Clipboard (with data type 'Midi'). Most other computers store 8-bit byte streams in files--naming or storage conventions for those computers will be defined as required.

Chunks

MIDI files are made up of chunks. Each chunk has a 4-character type and a 32-bit length, which is the number of bytes in the chunk. This structure allows future chunk types to be designed which may easily be ignored if encountered by a program written before the chunk type is introduced. Your programs should *expect* alien chunks and treat them as if they weren't there.

Each chunk begins with a 4-character ASCII type. It is followed by a 32-bit length, most significant byte first (a length of 6 is stored as 00 00 00 06). This length refers to the number of bytes of data which follow: the eight bytes of type and length are not included. Therefore, a chunk with a length of 6 would actually occupy 14 bytes in the disk file.

This chunk architecture is similar to that used by Electronic Arts' IFF format, and the chunks described herein could easily be placed in an IFF file. The MIDI File itself is not an IFF file: it contains no nested chunks, and chunks are not constrained to be an even number of bytes long. Converting it to an IFF file is as easy as padding odd-length chunks, and sticking the whole thing inside a FORM chunk.

MIDI Files contain two types of chunks: header chunks and track chunks. A header chunk provides a minimal amount of information pertaining to the entire MIDI file. A track chunk contains a sequential stream of MIDI data which may contain information for up to 16 MIDI channels. The concepts of multiple tracks, multiple MIDI outputs, patterns, sequences, and songs may all be implemented using several track chunks.

A MIDI file always starts with a header chunk, and is followed by one or more track chunks.

```
MThd    <length of header data>
               <header data>
MTrk     <length of track data>
                <track data>
MTrk     <length of track data>
                <track data>
```

Chunk Descriptions

Header Chunks

The header chunk at the beginning of the file specifies some basic information about the data in the file. Here's the syntax of the complete chunk:

Header Chunk> = <chunk type> <length> <format> <ntrks> <division>

As described above, <chunk type> is the four ASCII characters 'MThd'; <length> is a 32-bit representation of the number 6 (high byte first).

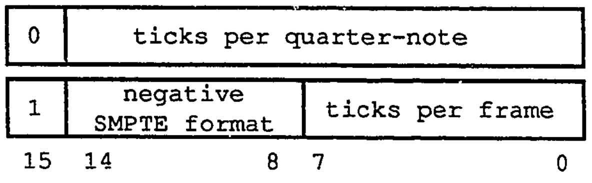
The data section contains three 16-bit words, stored most-significant byte first. The first word, format, specifies the overall organization of the file. Only three values of format are specified:

0	the file contains a single multi-channel track
1	the file contains one or more simultaneous tracks (or MIDI outputs) of a sequence
2	the file contains one or more sequentially independent single-track patterns

More information about these formats is provided below.

The next word, ntrks, is the number of track chunks in the file. It will always be 1 for a format 0 file.

The third word, division, specifies the meaning of the delta-times. It has two formats, one for metrical time, and one for time-code-based time:



If bit 15 of division is a zero, the bits 14 thru 0 represent the number of delta-time "ticks" which make up a quarter-note. For instance, if division is 96, then a time interval of an eighth-note between two events in the file would be 48.

If bit 15 of division is a one, delta-times in a file correspond to subdivisions of a second, in a way consistent with SMPTE and MIDI time code. Bits 14 thru 8 contain one of the four values -24, -25, -29, or -30, corresponding to the four standard SMPTE and MIDI time code formats (-29 corresponds to 30 drop frame), and represents the number of frames per second. These negative numbers are stored in two's complement form. The second byte (stored positive) is the resolution within a frame: typical values may be 4 (MIDI time code resolution), 8, 10, 80 (bit resolution), or 100. This system allows exact specification of time-code-based tracks, but also allows millisecond-based tracks by specifying 25 frames/sec and a resolution of 40 units per frame. If the events in a file are stored with bit resolution of thirty-frame time code, the division word would be E250 hex.

Formats 0, 1, and 2

A Format 0 file has a header chunk followed by one track chunk. It is the most interchangeable representation of data. It is very useful for a simple single-track player in a program which needs to make synthesizers make sounds, but which is primarily concerned with something else such as mixers or sound effect boxes. It is very desirable to be able to produce such a format, even if your program is track-based, in order to work with these simple programs. On the other hand, perhaps someone will write a format conversion from format 1 to format 0 which might be so easy to use in some setting that it would save you the trouble of putting it into your program.

A Format 1 or 2 file has a header chunk followed by one or more track chunks. Programs which support several simultaneous tracks should be able to save and read data in format 1, a vertically one-dimensional form, that is, as a collection of tracks. Programs which support several independent patterns should be able to save and read data in format 2, a horizontally one-dimensional form. Providing these minimum capabilities will ensure maximum interchangeability.

In a MIDI system with a computer and a SMPTE synchronizer which uses Song Pointer and Timing Clock, tempo maps (which describe the tempo throughout the track, and may also include time signature information, so that the bar number may be derived) are generally created on the computer. To use them with the synchronizer, it is necessary to transfer them from the computer. To make it easy for the synchronizer to extract this data from a MIDI File, tempo information should always be stored in the first MTrk chunk. For a format 0 file, the tempo will be scattered through the track and the tempo map reader should ignore the intervening events; for a format 1 file, the tempo map must *(starting in 0.04)* be stored as the first track. It is polite to a tempo map reader to offer your user the ability to make a format 0 file with just the tempo, unless you can use format 1.

All MIDI files should specify tempo and time signature. If they don't, the time signature is assumed to be 4/4, and the tempo 120 beats per minute. In format 0, these meta-events should occur at least at the beginning of the single multi-channel track. In format 1, these meta-events should be contained in the first track. In format 2, each of the temporally independent patterns should contain at least initial time signature and tempo information.

We may decide to define other format IDs to support other structures. A program encountering an unknown format ID may still read other MTrk chunks it finds from the file, as format 1 or 2, if its user can make sense of them and arrange them into some other structure if appropriate. Also, more parameters may be added to the MThd chunk in the future: it is important to read and honor the length, even if it is longer than 6.

Track Chunks

The track chunks (type MTrk) are where actual song data is stored. Each track chunk is simply a stream of MIDI events (and non-MIDI events), preceded by delta-time values. The format for Track Chunks (described below) is exactly the same for all three formats (0, 1, and 2: see "Header Chunk" above) of MIDI Files.

Here is the syntax of an MTrk chunk (the + means "one or more": at least one MTrk event must be present):

<Track Chunk> = <chunk type> <length> <MTrk event>+

The syntax of an MTrk event is very simple:

<MTrk event> = <delta-time> <event>

<delta-time> is stored as a variable-length quantity. It represents the amount of time before the following event. If the first event in a track occurs at the very beginning of a track, or if two events occur simultaneously, a delta-time of zero is used. Delta-times are always present. (Not storing delta-times of 0 requires at least two bytes for any other value, and most delta-times aren't zero.) Delta-time is in some fraction of a beat (or a second, for recording a track with SMPTE times), as specified in the header chunk.

<event> = <MIDI event> | <sysex event> | <meta-event>

<MIDI event> is any MIDI channel message. Running status is used: status bytes of MIDI channel messages may be omitted if the preceding event is a MIDI channel message with the same status. The first event in each MTrk chunk must specify status. Delta-time is not considered an event itself: it is an integral part of the syntax for an MTrk event. Notice that running status occurs across delta-times.

<sysex event> is used to specify a MIDI system exclusive message, either as one unit or in packets, or as an "escape" to specify any arbitrary bytes to be transmitted. A normal complete system exclusive message is stored in a MIDI File in this way:

F0 <length> <bytes to be transmitted after F0>

The length is stored as a variable-length quantity. It specifies the number of bytes which follow it, not including the F0 or the length itself. For instance, the transmitted message F0 43 12 00 07 F7 would be stored in a MIDI file as F0 05 43 12 00 07 F7. It is required to include the F7 at the end so that the reader of the MIDI file knows that it has read the entire message.

Another form of sysex event is provided which does not imply that an F0 should be transmitted. This may be used as an "escape" to provide for the transmission of things which would not otherwise be legal, including system realtime messages, song pointer or select, MIDI Time Code, etc. This uses the F7 code:

F7 <length> <all bytes to be transmitted>

Unfortunately, some synthesizer manufacturers specify that their system exclusive messages are to be transmitted as little packets. Each packet is only part of an entire syntactical system exclusive message, but the times they are transmitted at are important. Examples of this are the bytes sent in a CZ patch dump, or the FB-01's "system exclusive mode" in which microtonal data can be transmitted. The F0 and F7 sysex events may be used together to break up syntactically complete system exclusive messages into timed packets.

An F0 sysex event is used for the first packet an a series--it is a message in which the F0 should be transmitted. An F7 sysex event is used for the remainder of the packets, which do not begin with F0. (Of course, the F7 is not considered part of the system exclusive message).

(New to 0.06) A syntactic system exclusive message must always end with an F7, even if the real-life device didn't send one, so that you know when you've reached the end of an entire sysex message without looking ahead to the next event in the MIDI file. If it's stored in one complete F0 sysex event, the last byte must be an F7. If it is broken up into packets, the last byte of the last packet must be an F7. There also must not be any transmittable MIDI events in between the packets of a multi-packet system exclusive message. This principle is illustrated in the paragraph below.

Here is an example of a multi-packet system exclusive message: suppose the bytes F0 43 12 00 were to be sent, followed by a 200-tick delay, followed by the bytes 43 12 00 43 12 00, followed by a 100-tick delay, followed by the bytes 43 12 00 F7, this would be in the MIDI File:

```
              F0 03 43 12 00
                          81 48      200-tick delta-time
     F7 06 43 12 00 43 12 00
                             64      100-tick delta-time
           F7 04 43 12 00 F7
```

When reading a MIDI File, and an F7 sysex event is encountered without a preceding F0 sysex event to start a multi-packet system exclusive message sequence, it should be presumed that the F7 event is being used as an "escape". In this case, it is not necessary that it end with an F7, unless it is desired that the F7 be transmitted.

<meta-event> specifies non-MIDI information useful to this format or to sequencers, with this syntax:

FF <type> <length> <bytes>

All meta-events begin with FF, then have an event type byte (which is always less than 128), and then have the length of the data stored as a variable-length quantity, and then the data itself. If there is no data, the length is 0. As with chunks, future meta-events may be designed which may not be known to existing programs, so programs must properly ignore meta-events which they do not recognize, and indeed, should *expect* to see them. *New for 0.06*: programs must never ignore the length of a meta-event which they do recognize, and they shouldn't be surprised if it's bigger than they expected. If so, they must ignore everything past what they know about. However, they must not add anything of their own to the end of a meta-event.

Sysex events and meta-events cancel any running status which was in effect. Running status does not apply to and may not be used for these messages.

Meta-Events

A few meta-events are defined herein. It is not required for every program to support every meta-event.

In the syntax descriptions for each of the meta-events a set of conventions is used to describe parameters of the events. The FF which begins each event, the type of each event, and the lengths of events which do not have a variable amount of data are given directly in hexadecimal. A notation such as dd or se, which consists of two lower-case letters, mnemonically represents an 8-bit value. Four identical lower-case letters such as wwww refer to a 16-bit value, stored most-significant-byte first. Six identical lower-case letters such as tttttt refer to a 24-bit value, stored most-significant-byte first. The notation len refers to the length portion of the meta-event syntax, that is, a number, stored as a variable-length quantity, which specifies how many data bytes follow it in the meta-event. The notations text and data refer to however many bytes of (possibly text) data were just specified by the length.

In general, meta-events in a track which occur at the same time may occur in any order. If a copyright event is used, it should be placed as early as possible in the file, so it will be noticed easily. Sequence Number and Sequence/Track Name events, if present, must appear at time 0. An end-of-track event must occur as the last event in the track.

Meta-events initially defined include:

FF 00 02 ssss Sequence Number

This optional event, which must occur at the beginning of a track, before any nonzero delta-times, and before any transmittable MIDI events, specifies the number of a sequence. The number in this track corresponds to the sequence number in the Cue and Tempo message discussed at the summer 1987 MMA meeting (refer to MMA TSBB #10: this is not yet a part of MIDI, so it's not further discussed here). In a format 2 MIDI file, it is used to identify each "pattern" so that a "song" sequence using the Cue message to refer to the patterns. If the ID numbers are omitted, the sequences' locations in order in the file are used as defaults. In a format 0 or 1 MIDI file, which only contain one sequence, this number should be contained in the first (or only) track. If transfer of several multitrack sequences is required, this must be done as a group of format 1 files, each with a different sequence number.

FF 01 len text Text Event

Any amount of text describing anything. It is a good idea to put a text event right at the beginning of a track, with the name of the track, a description of its intended orchestration, and any other information which the user wants to put there. Text events may also occur at other times in a track, to be used as lyrics, or descriptions of cue points. The text in this event should be printable ASCII characters for maximum interchange. However, other character codes using the high-order bit may be used for interchange of files between different programs on the same computer which supports an extended character set. Programs on a computer which does not support non-ASCII characters should ignore those characters.

(New for 0.06). Meta event types 01 through 0F are reserved for various types of text events, each of which meets the specification of text events(above) but is used for a different purpose:

FF 02 len text Copyright Notice

Contains a copyright notice as printable ASCII text. The notice should contain the characters (C), the year of the copyright, and the owner of the copyright. If several pieces of music are in the same MIDI file, all of the copyright notices should be placed together in this event so that it will be at the beginning of the file. This event should be the first event in the first track chunk, at time 0.

FF 03 len text Sequence/Track Name

If in a format 0 track, or the first track in a format 1 file, the name of the sequence. Otherwise, the name of the track.

FF 04 len text Instrument Name

A description of the type of instrumentation to be used in that track. May be used with the MIDI Prefix meta-event to specify which MIDI channel the description applies to, or the channel may be specified as text in the event itself.

FF 05 len text Lyric

A lyric to be sung. Generally, each syllable will be a separate lyric event which begins at the event's time.

FF 06 len text Marker

Normally in a format 0 track, or the first track in a format 1 file. The name of that point in the sequence, such as a rehearsal letter or section name ("First Verse," etc.).

FF 07 len text Cue Point

A description of something happening on a film or video screen or stage at that point in the musical score ("Car crashes into house," "curtain opens," "she slaps his face," etc.)

FF 20 01 cc MIDI Channel Prefix

The MIDI channel (0-15) contained in this event may be used to associate a MIDI channel with all events which follow, including System Exclusive and meta-events. This channel is "effective" until the next normal MIDI event (which contains a channel) or the next MIDI Channel Prefix meta-event. If MIDI channels refer to "tracks," this message may help jam several tracks into a format 0 file, keeping their non-MIDI data associated with a track. This capability is also present in Yamaha's ESEQ file format.

FF 2F 00 End of Track

This event is *not* optional. It is included so that an exact ending point may be specified for the track, so that it has an exact length, which is necessary for tracks which are looped or concatenated.

FF 51 03 tttttt Set Tempo (in microseconds per MIDI quarter-note)

This event indicates a tempo change. Another way of putting "microseconds per quarter-note" is "24ths of a microsecond per MIDI clock". Representing tempos as time per beat instead of beat per time allows absolutely exact long-term synchronization with a time-based sync protocol such as SMPTE time code or MIDI time code. This amount of accuracy provided by this tempo resolution allows a four-minute piece at 120 beats per minute to be accurate within 500 usec at the end of the piece. Ideally, these events should only occur where MIDI clocks would be located--this convention is intended to guarantee, or at least increase the likelihood, of compatibility with other synchronization devices so that a time signature/tempo map stored in this format may easily be transferred to another device.

FF 54 05 hr mn se fr ff SMPTE Offset

(New in 0.06 - SMPTE Format specification) This event, if present, designates the SMPTE time at which the track chunk is supposed to start. It should be present at the beginning of the track, that is, before any nonzero delta-times, and before any transmittable MIDI events. The hour *must* be encoded with the SMPTE format, just as it is in MIDI Time Code. In a format 1 file, the SMPTE Offset must be stored with the tempo map, and has no meaning in any of the other tracks. The ff field contains fractional frames, in 100ths of a frame, even in SMPTE-based tracks which specify a different frame subdivision for delta-times.

FF 58 04 nn dd cc bb Time Signature

The time signature is expressed as four numbers. nn and dd represent the numerator and denominator of the time signature as it would be notated. The denominator is a negative power of two: 2 represents a quarter-note, 3 represents an eighth-note, etc. The cc parameter expresses the number of MIDI clocks in a metronome click. The bb parameter expresses the number of notated 32nd-notes in a MIDI quarter-note (24 MIDI Clocks). This was added because there are already multiple programs which allow the user to specify that what MIDI thinks of as a quarter-note (24 clocks) is to be notated as, or related to in terms of, something else.

Therefore, the complete event for 6/8 time, where the metronome clicks every three eighth-notes, but there are 24 clocks per quarter-note, 72 to the bar, would be *(in hex):*

FF 58 04 06 03 24 08

That is, 6/8 time (8 is 2 to the 3rd power, so this is 06 03), 36 MIDI clocks per dotted-quarter (24 hex!), and eight notated 32nd-notes per MIDI quarter note.

FF 59 02 sf mi Key Signature

```
sf = -7:  7 flats
sf = -1:  1 flat
sf =  0:  0 flat and 0 sharps, key of C
sf =  1:  1 sharp
sf =  7:  7 sharps

mi =  0:  major key
mi =  1:  minor key
```

FF 7F len data Sequencer-Specific Meta-Event

Special requirements for particular sequencers may use this event type: the first byte or bytes of data is a manufacturer ID (these are one byte, or, if the first byte is 00, three bytes). As with MIDI System Exclusive, manufacturers who define something using this meta-event should publish it so that others may know how to use it. After all, this is an *interchange* format. This type of event may be used by a sequencer which elects to use this as its *only* file format; sequencers with their established feature-specific formats should probably stick to the standard features when using *this* format.

Program Fragments and Example MIDI Files

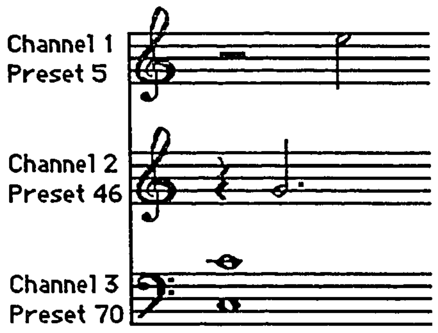
Here are some of the routines to read and write variable-length numbers in MIDI Files. These routines are in C, and use getc and putc, which read and write single 8-bit characters from/to the files infile and outfile.

```
WriteVarLen (value)
    register long value;
{   register long buffer;
    buffer = value & 0x7f;
    while ((value >>= 7) > 0)
    {   buffer <<= 8;
        buffer |= 0x80;
        buffer += (value & 0x7f);
    }
    while (TRUE)
    {   putc(buffer,outfile);
        if (buffer & 0x80)
            buffer >>= 8;
        else
            break;
    }
}

doubleword ReadVarLen ()
{   register doubleword value;
    register byte c;

    if ((value = getc(infile)) & 0x80)
    {
        value &= 0x7f;
        do
        {
            value=(value << 7) + ((c = getc(infile)) & 0x7f);
        } while (c & 0x80);
    }
    return (value);
}
```

As an example, MIDI Files for the following excerpt are shown below. First, a format 0 file is shown, with all information intermingled; then, a format 1 file is shown with all data separated into four tracks: one for tempo and time signature, and three for the notes. A resolution of 96 "ticks" per quarter note is used. A time signature of 4/4 and a tempo of 120, though implied, are explicitly stated.



The contents of the MIDI stream represented by this example are broken down here by Delta Time (decimal), Event Code (hex), Other Bytes (decimal), and Comment:

0	FF 58	04 04 02 24 08	4 bytes: 4/4 time, 24 MIDI clocks/click, 8 32nd notes/24 MIDI clocks
0	FF 51	03 500000	3 bytes: 500,000 µsec per quarter-note
0	C0	5	Ch. 1, Program Change 5
0	C1	46	Ch. 2, Program Change 46
0	C2	70	Ch. 3, Program Change 70
0	92	48 96	Ch. 3 Note On C2, forte
0	92	60 96	Ch. 3 Note On C3, forte
96	91	67 64	Ch. 2 Note On G3, mezzo-forte
96	90	76 32	Ch. 1 Note On E4, piano
192	82	48 64	Ch. 3 Note Off C2, standard
0	82	60 64	Ch. 3 Note Off C3, standard
0	81	67 64	Ch. 2 Note Off G3, standard
0	80	76 64	Ch. 1 Note Off E4, standard
0	FF 2F	00	Track End

The entire format 0 MIDI file contents in hex follow. First, the header chunk:

```
        4D 54 68 64                  MThd
        00 00 00 06          chunk length
              00 00              format 0
              00 01             one track
              00 60    96 per quarter-note
```

Then, the track chunk. Its header, followed by the events (notice that running status is used in places):

```
                   4D 54 72 6B                   MTrk
                   00 00 00 3B      chunk length (59)

Delta-time                   Event           Comments
        00   FF 58 04 04 02 18 08      time signature
        00      FF 51 03 07 A1 20               tempo
        00                  C0 05
        00                  C1 2E
        00                  C2 46
        00               92 30 60
        00                  3C 60      running status
        60               91 43 40
        60               90 4C 20
     81 40               82 30 40   two-byte delta-time
        00                  3C 40      running status
        00               81 43 40
        00               80 4C 40
        00               FF 2F 00        end of track
```

A format 1 representation of the file is slightly different. Its header chunk:

```
        4D 54 68 64                  MThd
        00 00 00 06          chunk length
              00 01              format 1
              00 04           four tracks
              00 60    96 per quarter-note
```

First, the track chunk for the time signature/tempo track. Its header, followed by the events:

```
                   4D 54 72 6B                   MTrk
                   00 00 00 14      chunk length (20)

Delta-time                   Event           Comments
        00   FF 58 04 04 02 18 08      time signature
        00      FF 51 03 07 A1 20               tempo
     83 00               FF 2F 00        end of track
```

Then, the track chunk for the first music track. The MIDI convention for note on/off running status is used in this example:

```
                          4D 54 72 6B                MTrk
                          00 00 00 10      chunk length (16)

Delta-time                      Event             Comments
        00                      C0 05
     81 40                   90 4C 20
     81 40                      4C 00      Running status:
                                             note on, vel=0
        00                   FF 2F 00          end of track
```

Then, the track chunk for the second music track:

```
                          4D 54 72 6B                MTrk
                          00 00 00 0F      chunk length (15)
Delta-time                      Event             Comments
        00                      C1 2E
        60                   91 43 40
     82 20                      43 00       running status
        00                   FF 2F 00         end of track
```

Then, the track chunk for the third music track:

```
                          4D 54 72 6B                MTrk
                          00 00 00 15      chunk length (21)
Delta-time                      Event             Comments
        00                      C2 46
        00                   92 30 60
        00                      3C 60       running status
     83 00                      30 00     2-byte deltaTime
                                            running status
        00                      3C 00       running status
        00                   FF 2F 00         end of track
```

BIBLIOGRAPHY

Gilman, L. and Rose, A. *APL: An Interactive Approach.* John Wiley & Sons, Inc., New York, 1984.

Jaxitron. *Cybernetic Music.* Tab Books Inc., Blue Ridge Summit, PA, 1985.

Kamin, S. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley Publishing Company, Inc., New York, 1990.

International MIDI Association. "Standard MIDI File Format V 1.0."