MUAECL2 User's Manual ver. 2.0.1_alpha(Eng. ver.)

MUA Group(Modification Ultraman Alliance)

today

Contents

1 ABOUT

MUAECL is a ecl compiler developed by MUA Group. It can translate a C-like language (MUAECL language) to a ecl file, in order to be read by either MUA engine, or th15 engine.

2 INSTALLATION AND RUNNING

Download and unzip to any folder. The unzipped folder contains:

File name	Usage
`MUAECL2.exe`	the program
`action.csv`	jumping table
`ins.ini`	instruction translate table
`default.ini`	sub names contained in default.ecl
`include.ini`	predefined preprocess directive

These files need to be contained in the same folder.

MUAECL run in command line. Running "`MUAECL2.x.x.exe -h`"(The real exe name will change along with version number) in cmd or "`MUAECL2.x.x.exe --help`" will check the command line syntax help.

Command Line Argument		Usage
`-h`	`--help`	Display command line syntax help
`-c`	`--compile`	Compile an MUAECL2 source file to a raw ECl file
`-p`	`--preprocess`	Preprocess (and not compile) an MUAECL2 source file
`-i <filename>`	`--input-file <filename>`	Input filename
`-o <filename>`	`--output-file <filename>`	Output filename
`-s <filepath>`	`--search-path <filepath>`	Search path for includes, can assign multiple path
`-n`	`-no-preprocess`	Bypass the preprocessing step during compiling

Examples:

```
MUAECL2 -h
MUAECL2 -c -i st01.mecl -o st01.ecl
MUAECL2 -p -i st01.mecl -o st01p.mecl
MUAECL2 -c -n -i st01p.mecl -o st01.ecl
```

3 MUAECL Language

MUAECL Language is a C-like script language, which describes some information for MUA engine, such as bullet shooting or enemy entering.

3.1 Preprocess directives

First, the MUAECL preprocessor checks for backslash which appears at the end of a line. Delete those backslashes, and moves the text on the next line to th original line.

MUAECL preprocess directives begins with `#`. It contains those following types:

s directive (s stands for substitute) `#s` *deliminator pattern deliminator string deliminator name*

Usage: Substitute all matching *pattern* (checks for regular expression) with *string* in following texts, until a `ends` directive with the same *name*. If no matching `ends`, substitute until EOF.

Examples:

```
#s/abc/def/name
#s ((?:a|b|c)*)d(e?) $2$1 mysubs1
```

1. *deliminator* should be a non-identifier character `[^a-zA-Z0-9_]`.
2. Special character in *pattern* and *string* will be interpreted according to ECMAScript grammar, which is the same as what C++ standard library <reg_ex> use by default.
3. Read *pattern* and *string* all by literal, including space and tab, but not including newline.
4. *pattern* and *string* are not allowed to contain the non-identifier character used for *deliminator*, and not allowed to match or substitute to multiline text.
5. *name* could not contain non-identifier character.
6. Preprocessor will report an error when two *name* being the same.

unnamed `s` directive `#s` *deliminator pattern deliminator string deliminator*

Usage: Substitute all matching *pattern* with *string* in following texts until EOF.

ends directive `#ends` *deliminator name*

Usage: End the substituting whose name is *name*. The constrain of *deliminator* and *name* are the same as `s` directive.

define directive `#define` *space identifier space string*

Usage: Just like C, substitute keyword *identifier* to *string*.

Example:`#define abc def`
Will substitute `abc + abcd` to `def + abcd`.

1. Same as `s` directive who matches "`\b`*identifier*`\b`", while *name* defined to *identifier*
2. *identifier* could not contain non-identifier character.

function type define directive `#define` *space identifier{paraname1, paraname2, ...} space string*

Usage: Just like C, substitute keyword *identifier* with parameters to *string*. What do not like C is that during processing *paraname* within *string*, only substitute string but not identifier. **So PLEASE be cautious, DO NOT let paranames contain each other as substrings, or let *string* contain some of the paranames.**

Example: `#define abc{d,e,f} (d+e+f+def)`
Will substitute `abc{1,3,5}` to `(1+3+5+135)`.

1. *identifier* and *paraname* could not contain non-identifier character.
2. Using the example above, it is the same as: substitute *paraname1* to '`$01`', *paraname2* to '`$02`', and so on in *string*:
   `#s \babc\{\s*(.*?\S)\s*,\s*(.*?\S)\s*,\s*(.*?\S)\s*\} ($01+$02+$03+$01$02$03) abc`

undef directive `#undef` *space name*

Usage: End the matching of *name*. Same as `#ends` *name*.

include directive `#include` *space includefile*

Declares in current file the subs defination and declaration in the target mecl file. Recursive including is allowed.

ecli including `#ecli` *space eclfile*

anim including `#anim` *space anmfile*

Add include ecl and anm files.

Example: `#ecli default.ecl`

All preprocess will not change line numbers afterwards.

3.2 Predefined macro variables

mecl includes some predefined macro variables:

Macro name	Usage
`__FILE__`	Expands to the name of the current file
`__DATE__`	Expands to the date when translating the current keyword
`__TIME__`	Expands to the time when translating the current keyword
`__LINE__`	Expands to the line number
`pi`	Expands to 3.14159265

4 BASIC CONCEPTS

4.1 COMMENTS

mecl allows two types of comments: C-style and C++-style:

```
1. /* some content */
2. // some content \n
```

Unlike C, the processing of comment is later than the preprocess substituting, which means: 1. Preprocess directives will be checked inside the comment range; 2. Comment token which is formed by preprocess substituting will be checked.

4.2 IDENTIFIER

Just like C, a *identifier* is an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters, and should not begin with digits. Identifiers are case-sensitive.

Examples of valid identifiers: `shedarshian`,`Temp0_float`,`_name_`

4.2.1 GLOBAL VARIABLES

mecl contains many global variables, such as the coordinate of player, the internal random number generator. A list will be found in ins.ini. If an identifier being the same as a global variable, it will be considered as that global variable.

4.3 TYPES

mecl contains types below:

void, int, float, string, point. (Current version do not support string type **VARIABLES**.)

In mecl, int and float occupies 4 bytes.

We call "int, float, point" as arithmetic type, and "int float" as fundamental arithmetic type.

4.3.1 IMPLICIT TYPE CONVERSION

Implicit type conversion will be check when needed (for example, when overload resolution happens, or in if/for expressions).

Implicit conversion includes: lvalue to rvalue conversion; int to float conversion, and void lvalue to any type lvalue conversion.

5 SUB-PROGRAMS

mecl is made up of serveral sub-programs (`sub`): *suffix* `sub` *subname* `(` *subvars* `)` *suffix* `{` *subcontents* `}`

suffix is an optional `no_overload` keyword. Parameter lists are comma-separated list of *parameter declaration*, each of which contains a type name and a variable name. These named formal parameters can be approached inside function body. Function body *subcontents* contains serveral statements.

Example:

```
sub MainSub01(float x) no_overload
{
        //some content
}
no_overload sub Main()
{
        //some content
}
sub Boss_at1(int num, float angle) {
        //some content
}
```

Sub-programs are allowed to have same names (**overload**), as long as parameter number or type differs. Overload resolution will take place when calling overloaded function.

By default, the compiler will use the parameter types to decorate the sub name. Keyword `no_overload` will tell the compiler not to decorate the sub name, and in result, forbid the overload of the current function.

The sub-program name is global, which means **usage before declaration** is allowed.

5.1 SUB-PROGRAM DECLARATION

Sub-programs can be declared without defination: *suffix* `sub` *subname* `(` *subvars* `)` *suffix* `;`

This is often used when the sub-program is defined in some other files. Declaration will assign the function signature into the list, which means that the same function is not allowed (and not needed) to be both declared and defined.

6 EXPRESSIONS

An expression is a sequence of *operators* and their *operands*, that specifies a computation.

6.1 TYPES AND VALUE CATEGORIES

Each expression is characterized by two independent properties: a type and a value category. There are two value categories: lvalue and rvalue. Lvalue indicates its evaluation determines the identity of an object, while rvalue indicates its evaluation do not produce a long-lived object.

The following expressions are lvalue expressions:

1. The name of a variable.
2. Assignment and compound assignment expressions, such as `a = b`.
3. Indirection expressions, such as `*p`.

Lvalue's address may be taken`&(i = 1)`, and lvalue may be used as left-hand operand of the assignment and compound assignment operators.

The following expressions are rvalue expressions:

1. Literals, such as `72`.
2. Function call expressions, such as `g(1)`.
3. Arithmetic expressions, logical expressions, or comparision expressions, such as `a + 1 c && d`.
4. Address-of expressions, such as `&a`.
5. Type cast expressions, such as `(int)f`.

Rvalue's address may not be taken, and rvalue can't be used as left-hand operand of the assignment and compound assignment operators.

6.2 PRIMARY EXPRESSIONS

Primary expressions includes literals and identifiers.

Integer literals Integer literals contains:

1. Decimal literals, which is a non-zero decimal digit `1-9` followed by zero or more decimal digits `0-9`;
2. Octal literals, which is the digit `0` followed by zero or more octal digits `0-7`;
3. Hex literals, which is the character sequence `0x` followed by zero or more hexadecimal digits `0-9` or `A-F` or `a-f`.

For example, the following variables are initialized to the same value:

```
int d = 42;
int o = 052;
int x = 0x2a;
int X = 0x2A;
```

Floating-point literal Floating-point literal is a decimal number which contains and only contains one decimal separator, and a optional `f` in the end, such as `2.34`, `.1f`, `1.`.

String literal String literal is a character sequence surrounded by a pair of double-quote `""`. The backslash can be used to escape characters, since the character followed by a backslash will be recognized as itself. So it can be used to escape the double-quote character itself. String literals are of type string.

Predefined constant Predefined constant are defined in ins.ini, and will be converted to corresponding integer or float literal. Predefined constant supported will be listed around corresponding instruction.

Identifier Identifier can be used as function name, or to refer a variable.

6.3 OPERATORS

Operators contain following categories:

<table>
<tr><th>Assignment</th><th>Arithmetic</th><th>Logical</th><th>Comparison</th><th>Member access</th><th>Others</th></tr>
<tr>
<td>a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &&= b
a ||= b
a &= b
a |= b
a ^= b</td>
<td>-a
a + b
a - b
a * b
a / b
a % b
a & b
a | b
a ^ b</td>
<td>!a
a && b
a || b
a and b
a or b</td>
<td>a == b
a != b
a < b
a <= b
a > b
a >= b</td>
<td>a.b
a[b]
*a
&a</td>
<td>id(...)
a1 : a2 : a3 : a4
(type)a
(a)</td>
</tr>
</table>

Built-in operators have serveral built-in overload, overload resolution rule is used to specify which overload is used.

6.3.1 ARITHMETIC OPERATOR

Unary minus operator Returns the negative of its operand. Has the form `-expr`.

For every arithmetic type `A`, the following function signatures participate in overload resolution:

```
A operator-(A)
```

Add and minus operators Return the sum or subtract of its operand. Have the form `expr + expr`, `expr - expr`.

For every arithmetic type `A`, the following function signatures participate in overload resolution:

```
A operator+(A, A)
A operator-(A, A)
string operator+(string, string)
```

Multiplicative operators Return the product, division or remainder of its operand. Have the form `expr * expr`, `expr / expr`, `expr % expr`

For every fundamental arithmetic type `I`, the following function signatures participate in overload resolution:

```
I operator*(I, I)
I operator/(I, I)
I operator%(I, I)
point operator*(float, point)
point operator*(point, float)
point operator/(point, float)
```

Bitwise logic operators Return the bitwise AND, bitwise OR, or bitwise XOR of its operand. Have the form `expr & expr`, `expr | expr`, `expr ^ expr`

The following function signatures participate in overload resolution:

```
int operator&(int, int)
int operator|(int, int)
int operator^(int, int)
```

6.3.2 LOGICAL OPERATORS

Logical operators return the logical AND, logical OR, or logical NOT of the operand. Use integer to represent boolean value, non-zero for true and zero for false. For return value, 1 for true and 0 for false. They have the form `expr && expr`, `expr || expr`, `expr and expr`, `expr or expr`, `! expr`. There are two types of logical AND and OR: the symbol types are short-circuiting, while the word types are not short-circuiting. Short-circuiting stands for: for `&&`, if the first operand is false, the second operand is not evaluated; for `||`, if the first operand is true, the second operand is not evaluated.

The following function signatures participate in overload resolution:

```
int operator&&(int, int)
int operator||(int, int)
int operator and(int, int)
int operator or(int, int)
int operator!(int)
```

6.3.3 COMPARISON OPERATORS

Comparison operators return the compares the operands. They have the form `expr == expr`, `expr != expr`, `expr < expr`, `expr <= expr`, `expr > expr`, `expr >= expr`

For every one of the four types `T` (int, float, point, string), the following function signatures participate in overload resolution:

```
int operator==(T, T)
int operator!=(T, T)
int operator<(T, T)
int operator<=(T, T)
int operator>(T, T)
int operator>=(T, T)
```

6.3.4 ASSIGNMENT OPERATORS

Colon-separated expression are allowed to be the right operand of assignment operators, and not allowed in other operators. Assignment operators expects a modifiable lvalue as its left operand and an rvalue expression as its right operand.

Direct assignment operator Direct assignment operator modifies its left operand with the value of its right operand. It has the form `expr = exprf`. It returns an lvalue identifying the left operand after modification.

For every one of the four types `T` (int, float, point, string), the following function signatures participate in overload resolution:

```
T& operator=(T&, T)
```

Compound assignment operators Compound assignment operators computes the result of its operand, and use it to modify its left operand. `E1 op= E2` is exactly the same as the behavior of the expression `E1 = E1 op E2`, except that the expression `E1` is evaluated only once. They have the form `expr += exprf`, `expr -= exprf`, `expr *= exprf`, `expr /= exprf`, `expr %= exprf`, `expr &&= exprf`, `expr ||= exprf`, `expr &= exprf`, `expr |= exprf`, `expr ^= exprf`. Here the logical AND assignment operator and the logical OR assignment operator are short-circuiting.

For every arithmetic type `A`, every fundamental arithmetic type `I`, every one of the four types `T` (int, float, point, string), the following function signatures participate in overload resolution:

```
T& operator+=(T&, T)
A& operator-=(A&, A)
I& operator*=(I&, I)
point& operator*=(point&, float)
I& operator/=(I&, I)
point& operator/=(point&, float)
int operator%=(int, int)
int operator&&=(int, int)
int operator||=(int, int)
int operator&=(int, int)
int operator|=(int, int)
int operator^=(int, int)
```

6.3.5 MEMBER ACCESS OPERATORS

Indirection operator Indirection operator takes out the object pointed-to by the pointer operand. It has the form `*expr`. Since there is no pointer type in current version, integer is used to replace pointer, so indirection operator returns a void lvalue, and type cast is needed to specify the type of the object taken out. Especially, void lvalue can be implicitly type-cast to any type, so the object type may be deduced by context. Expression whose type cannot be deduced is an error. More information can be approached in overload resolution.

The following function signatures participate in overload resolution:

```
void& operator*(int)
```

Address-of operator Address-of operator creates a pointer pointing to the object operand. It has the form `&expr`. Since there is no pointer type in current version, address-of operator returns a integer.

For every one of the four types `T` (int, float, point, string), the following function signatures participate in overload resolution:

```
int operator&(T&)
```

Member of object operator `expr.expr`, no effect in current version.

Subscript operator `expr[expr]`, no effect in current version.

6.3.6 OTHER OPERATORS

Function call operator Has the form `id(exprf, exprf, ...)`. `id` is a identifier identifies function name. The name may be a sub-program defined by user, or a built-in instruction name or mode name. The parameter can be a colon-separated expression or normal expression. User-defined sub-program name may be overloaded, overload resolution rule will take place when calling overloaded function. Almost all built-in instruction and mode will return void. Only a few built-in instruction that has a return value are math calculation.

Colon separated expression Colon separated expression is a easy-to-use difficulty switch. It receives four parameters, which stand for ENHL four difficulty. It has the form `expr : expr : expr : expr`. It can only be used in assignment operator's right operand and function calling.

For every one of the four types `T` (int, float, point, string), the following function signatures participate in overload resolution:

```
T operator:(T, T, T, T)
```

C-style type cast Type cast expression can convert one type to another type. It has the form `(type)expr`. There is no restrict on type cast.(???)

Round braket expression Round braket expression is used to specify order of operations. It has the form `(expr)`.

6.4 OPERATOR PRECEDENCE

The following table lists the precedence and associativity of mecl operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`()`	Round braket	Left-to-right
2	`a.b`	Member-of	Left-to-right
	`a[b]`	Subscript	
3	`!`	Logical NOT	Right-to-left
	`*a &a`	Reference and dereference	
	`-a`	Unary minus	
4	`a*b a/b a%b`	Multiplication Division Remainder	Left-to-right
5	`a+b a-b`	Addition Subtraction	Left-to-right
6	`a<b a<=b`	Less-than Less-or-equal	Left-to-right
	`a>b a>=b`	Larger-than Large-or-equal	
7	`a==b a!=b`	Equal Not-equal	Left-to-right
8	`a&b`	Bitwise AND	Left-to-right
9	`a^b`	Bitwise XOR	
10	`a|b`	Bitwise OR	
11	`a and b`	Non-short-circuiting Logical AND	
12	`a or b`	Non-short-circuiting Logical OR	
13	`a&&b`	Short-circuiting Logical AND	
14	`a||b`	Short-circuiting Logical OR	
15	`= += -= *= /=`	All assignment	Right-to-left
	`%= &&= ||= &= ^= |=`		
16	`a1:a2:a3:a4`	Colon-separated	-

6.5 OVERLOAD RESOLUTION

When choosing a overload of a function or operator, use the following rules:

For each overload of the function or operator, the compiler first test whether the parameter number are the same, and whether the type of each input parameter can implicitly type cast to the target type. Then the compiler choose a BEST in all overloads that can be changed to. "BEST" is defined as follow: a implicit type cast that has high precedence is better than a implicit that does not have it. The precedence of implicit cast is:

`void&`→`point`, `void&`→`string`, `void&`→`float`, `void&`→`int`, `int`→`float`, `lvalue`→`rvalue`.

6.6 LITERAL OPERATION OPTIMIZATION

The compiler will compute operation on literal value during compile time, and saves the result in ecl file. For example, `a*(1./2)` will change to `a*0.5` in ecl files.

7 STATEMENTS

Mecl has following types of statements (*stmt*):

8 EXPRESSION STATEMENTS

An expression followed by a semicolon is a statement.

expr `;`

Example:

```
a += 1;                         //assignment statement
wait(20);               //ins calling
a;                                //discard value expression
```

8.1 STATEMENTS BLOCK

`{` *stmts* `}`

Using brace to enclose sequences of statements will be treated as one statement.

8.2 VARIABLE DECLARATION

Variable declaration statement declares (and optionally initialize) one or more variables. It has the form `type` *vdecl* ;

Where *vdecl* is a comma-separated list of one or more following term: `id` or `id =` *inif*

inif can be a initializer-list or expression with or without colon-separated.

Each variable declared in mecl are sub-program local, which means **usage before declaration** is allowed as long as they are in the same sub-program. Also there are no **scopes** in mecl.

Example:

```
int i;
float a = 1., b;
int j = 123 * 456 : 234 * 567 : 345 * 678 : 456 * 789;
```

8.2.1 INITIALIZER LIST

`{` *expr1, expr2,...* `}`

Initializer-list is used to initialize compound arithmetic type (such as point). For example:

```
point p = {1., 2.};
point p1 = {1., 2.}:{3., 4.}:{5., 6.}:{7., 8.};
```

8.3 SELECTION STATEMENTS

Selection statements choose between one of several flows of control. They have the form:

```
if ( condition ) true_stmt
if ( condition ) true_stmt else false_stmt
```

If the condition yields **NOT ZERO**, *true_stmt* is executed. Otherwise *false_stmt* is executed.

condition need to be implicitly casted to int rvalue.

In the second form of `if` statement (the one including `else`), if *true_stmt* is also an `if` statement then that inner `if` statement must contain an `else` part as well (in other words, in nested `if`-statements, the `else` is associated with the closest `if` that doesn't have an `else`)

If *true_stmt* is entered by goto, *false_stmt* is not executed.

Example:

```
if (i == 1) {
        et_shape(0, ET_small, BLUE16);
        et_shoot(0);
        wait(30);
}
else if (i == 2)
        if (j == 2)
                wait(20);
        else                        //This else is associated with 'j == 2'
                wait(30);
```

8.4 Looping statements

Loopint statements repeatedly execute some code. They has the form:

```
while ( condition ) stmt
do stmt while ( condition ) ;
loop ( times ) stmt
```

`break` and `continue` may be used to control flow in looping statements.

8.4.1 WHILE LOOP

`while` loop executes *stmt* repeatedly, until the value of *condition* becomes 0.

condition need to be implicitly casted to int rvalue.

Example:

```
float s = 10.;
while (s >= 0) {
        et_dir(0, RanDEG, 0.);
        et_speed(0, s, 0.);
        et_shoot(0);
        wait(1);
        s -= 0.1;
}
while (1)
        wait(1000);                         //This is a dead loop
```

8.4.2 DO-WHILE LOOP

`do-while`acts just like `while` loop, excepts that *stmt* is always executed at least once.

Example:

```
float c = 1, s = 0;
do {
        s += c;
        c /= 2;
} while (c > 0.00001);
```

8.4.3 LOOP LOOP

`loop` loop executes *stmt* *times* time

times need to be implicitly casted to int rvalue.

Example:

```
loop (100) {
        et_shoot(1);
        wait(1);
}
```

8.5 LABEL STATEMENT

Label statement set up a label for `goto` statement to jump. It has the form:

```
id :
```

8.6 JUMP STATEMENTS

Jump statements unconditionally transfer flow control. They have the form:

```
break ;
continue ;
goto id ;
```

8.6.1 BREAK STATEMENT

`break` statement terminates the enclosing `while`, `do-while`, `loop` loop.

A `break` statement cannot be used to break out of multiple nested loops. The `goto` statement may be used for this purpose.

Example:

```
while (1) {
        et_shoot(0);
        wait(30);
        if (Yplayer <= 120.)
                break;
}
et_shoot(1);
```

8.6.2 CONTINUE STATEMENT

`continue` statement skips the remaining portion of enclosing `while`, `do-while`, `loop` and continue next loop, as if by `goto` to the end of the loop body.

Example:

```
loop (1000) {
        et_shoot(0);
        wait(10);
        if (Y_absol <= 224.)
                continue;
        et_shoot(1);
}
```

8.6.3 GOTO STATEMENT

`goto` statement transfers control to the location specified by label. The `goto` statement must be in the same sub-program as the label it is referring, it may appear before or after the label.

8.7 THREAD STATEMENT

`thread` statement opens a thread, which simultaneously execute with the following program. It has the form:

```
thread id ( exprf, exprf, ... ) ;
```

Example:

```
thread BossCard1_at1(1., 20, pi/20, RED32);
thread BossCard1_at1(2., 20, pi/20, GREEN32);
thread BossCard1_at1(1.5, 10, pi/10, CYAN32);
while (1) wait(1000);
```

8.8 RAWINS STATEMENT

`rawins` construct raw ecl content using data. It has the form:

```
__rawins {
        insdata ;
        insdata ;
        ...
}
```

Each *insdata* are:
`{` *id difficulty_mask pop_count param_count param_mask, param1 param2 …* `}`