The FTS C Programming Style Guide.

This are the official C programming guidelines for the FTS team.

- Why a style guide ?
- Some good principles on optimization
- Functions
  - ANSI prototypes
  - No unused static functions
  - Non local functions
  - Return types
  - Static functions
- Variables
  - Automatic variable initialization
  - Global variables
  - Global variables initialization
  - No unused constants
  - No unused variables
  - Register variables
  - Static variables
- Control structures
  - Assignment in `if', `while' and `for' conditions
  - Goto statements
  - No `for' loops with empty body
  - Only iteration variable in the `for' clauses
  - The comma operator
- Expressions
  - Bit operators
  - Pointers and integers
- Naming
  - Structure fields names
  - Variables and functions names
- Code presentation
  - Coding indentation style
  - Commented code
  - Comments on a line
  - Debug code
  - Tab size

- Memory management
  - Freeing memory
- Compilation
  - ANSI C
  - Compilation warnings

Why a style guide ?

External objects programming in Max derive from times where ANSI C was still a dream, and "static" denotated a program that don't move too much.

It is very likely, and already happened, that code written by external objects programmer will be included in official Max/Fts distribution, and that the FTS team will be required to support this code.

As the introduction of `tabpeek` and `tabpoke` proved, putting some code in a directory called "unsupported" don't prevent it to be widely used if the functionality implemented is needed in the user base.

For this reason, no future release will include unsupported C code written in Ircam; if an object is included in the official Ircam distribution, it is fully supported by the development team.

But adding source code to support to a system already difficult to maintain raises the maintenance cost, and our resources are limited.

In order to reduce this cost, we require that any contribution under the form of C source code have to comply with the following guidelines in order to included in an official distribution.

In general, if your C source do not fully comply with the the guidelines will *not* be accepted, under any condition; for some of the point we use the term "recommended"; this means that we may decide to accept some source C also if they don't comply with some of the recommended guidelines.

Some good principles on optimization

The main purpose in writing a program is making it readable to other human being, including yourself.

Some programmers write the code with the attitude of helping the compiler to generate faster code; this programmers are wrong.

Most of the time, the assumption made by these programmers refer to a compiler technology of the 70s; modern compiler can perform low-level and function-level optimization a lot better than most of the better assembler programmer; more over, the analysis on the source code is so deep and complex that most of the programming tricks at the source level are worthless, and sometime get also get the reverse effect.

On the contrary, writing a clear and readable code help the programmer, the designer of the system, to find optimization at the architectural and algorithmic level, that the compiler cannot find.

Nevertheless, sometime you may need to perform hand tuning on the code; if you do need this, avoid blind optimization based on your personal feelings; first, write the programs in the most readable way, and care about low level optimization only when you can prove, that the performance problem can be solved at this level (most of the time, it cannot).

In many platforms there are fine analysis tools (`pixie`, for example, on SGI or Dec machines) that allow you to

know exactly where FTS spend time; on the results of these analysis, that are often counterintuitive, spend your time in optimize only the critical functions.

Also, don't try to invent the wheel again and again; for most of the time consuming tasks, there are well known optimization techniques; ask around, check with your local vector computation guru, or try to ask us.

Functions

ANSI prototypes

All the functions should be defined and declared using full ANSI prototypes. For example, this declaration is not allowed:

```
void foobar();
```

Assuming that foobar don't take arguments, the correct declaration is:

```
void foobar(void);
```

No unused static functions

The code should not include functions that are not called; if they are debugging function, see the debug code section.

Non local functions

Functions that are not declared static should be correctly prototyped in ".h" file, included with the sources, and a minimal documentation should be included on their purpose and arguments.

A file referring to a global function should include the proper ".h" header file, and not having a local copy of the function declaration.

This documentation may be included in the C source or in the header file as a comment.

The only exception to this rule, for the time being, is "_config" function of an external object.

Return types

Return types for functions should be always specified, i.e. a function not returning a value should be declared "void", and a function returning an int should be declared "int".

Examples:

```
incrp(int *x)
{
  (*x)++;
}
```

Is illegal because it don't return a value, but is declared implicitly int; the return type should be declared "void".

```
addone(int x)
{
   return x+1;
```

```
}
```

Must be declared of type int.

Static functions

All the functions that are not intended to be called by name from other files should be declared static.

It must be remembered that the "static" keyword declare the function name local to the file; this do not means that a pointer to a static function cannot be used to call the function from an other file.

In the case of an external objects, usually the only function that should not be static is the "_config" function; the FTS module structure also usually encourage a programming style where the only exported symbol is the module itself.

Note also that a static declaration may in some case allow the compiler to remove the function body completely, when all the call to the functions can be inlined.

Variables

Automatic variable initialization

The Initialization of an automatic variable is allowed in a declaration only when the initialization is intrinsically part of the semantic of the variable, i.e. if the variable is part of a complex data structure, where the consistency is guaranteed by assigning a known value to the variable.

Example:

```
{
  t_atom table_of_values[MAX_VALUES];
  int table_fill_pointer = 0;

  ...
}
```

is OK, and recommended, because the pair (table_of_values, table_fill_pointer) together form a data structure that would be inconsistent without assigning 0 to the table_fill_pointer.

On the contrary, assignment that are part of the algorithm in the strict sense are not allowed, like:

```
{
  int new_size = old_size * GROW_RATE;
  int alloc_size = new_size - old_size;
  int do_it_carefully = alloc_size > MAX_RECOMMENDED_SIZE:

  ...
}
```

This code example (another classic from the MAX original source code) hide a significant part of the algorithm inside the variable declarations, making them a lot more difficult to read, and to comment; moreover, introduce dependency on the order of declarations, that may introduce errors later.

The code should be coded as:

```
{
  int new_size;
  int alloc_size;
  int do_it_carefully;

  new_size = old_size * GROW_RATE;
  alloc_size = new_size - old_size;
  do_it_carefully = alloc_size > MAX_RECOMMENDED_SIZE:

  ...
}
```

Another case where initialization of automatic variables is not allowed is in the case of convenience or iteration variables; iteration variables should be initialized in the context of the loop (in the for statement or before the while).

Example:

```
{

  int i = 0;

  for (; i < MAX; i++)
        ....
}
```

Is not allowed; the i variable should be initialized in the for statement, because its value have no meaning outside the loop.

Global variables

In principle, global variables are a "bad idea"; so in general, we recommend to use no global variables; if they are really needed they should be defined at the beginning of the file, within a code section that is explicitly marking as containing global variables definitions.

Every global variable must be properly documented with a comment.

A file referring to a global variable should include the proper ".h" header file, and not having a local copy of the variable declaration.

Example of good code:

```
#include "...."

/* Global variables for memory handling */

/* count how many bytes have been globally allocated */

long fts_byte_counter = 0;
```

Global variables initialization

If the program assume that a variable (global or local, static or automatic) have a specific initial value, the variable should be explicitly initialized to that value.

While ANSI C guarantee that static and external variables are initialized to zero, explicitly initializing them to zero express that the initial value is part of the semantic of the variable, and document the fact that the zero value is assumed.

Good Example:

```
static int foobar_selected;
static int foobar_counter = 0;
```

In this case that the initial value of foobar_selected is not used, and so probably a zero value is meaningless; while the foobar_counter is significant also at zero.

Bad example:

```
static int zapn;

void
dozap(void)
{
   zapn++;
}
```

It is not clear from the zapn definition that its initial value is significant.

No unused constants

There should be no unreferred #define'd constants.

No unused variables

There should be no unused variables in the code, either local to a function or global to a file.

Register variables

Don't use register declarations; they are useless, because all the modern compiler ignore the register declarations, and make you think you are in control of the generated code, while you aren't.

Static variables

All the variables whose purpose and use is local to file must be declared static.

As for functions, r-ember that the "static" keyword declare that the name of a variable (and not the variable itself) is local to the current file; the variable (i.e. the data structure itself) can be still access from other files by means of pointers.

Control structures

Assignment in `if', `while' and `for' conditions

Is is recommended to don't use assignments in if conditions.

Example:

```
        if (file = open(filename, "r"))
        {
           .....
        }
```

Should be written as

```
        file = open(filename, "r");

        if (file)
        {
           .....
        }
```

The only allowed assignment inside an if condition is at the expression top level.

In no case an assignment can be nested inside a logical expression, like the following:

Wrong example:

```
        int new_size;

        ........

        if ((need_size > old_size)) && ((new_size = old_size * GROW_RATE) <
MAX_SIZE))
        {
                realloc_table(new_size);
        }
```

This example (a classic from the Max original sources) is not readable, because in the condition we hide an essential part of the algorithm, that should be coded separately; moreover, the single steps of the algorithm cannot be properly commented, and the new_size variable get a scope bigger than the needed.

```
        .....

        if (need_size > old_size)
        {
           int new_size;

           /* need reallocation: compute new possible size */

           new_size = old_size * GROW_RATE;
```

```
        /* if not too big, reallocate it */

        if (new_size  < MAX_SIZE)
             realloc_table(new_size);
        else
             .....
    }
```

About the "while" statement, there are coding styles where the result is readable, like:

```
    while (c = getchar())
    {
        ... DO SOMETHING WITH c ...
    }
```

Sometimes, a for would do a better job.

In for, most of the time an assignment in the condition clause is sign of a badly written for; most of the time the assignment should be in the iteration clause; anyway, if you really need to put an assignment in a conditional, we ask you to find the FSF coding conventions, and add an extra pair of parenthesis, like in:

```
    while ((c = getchar()))
    {
        ... DO SOMETHING WITH c ...
    }
```

So to mark clearly that it is not a case of a mistyped equality test.

Goto statements

Goto statements are not allowed.

Goto in C are needed only in very complex situation, to handle the weakness of C in handling multi-level exit from complex nested loops; anyway, most of the time such multi-level nested loops are not needed or a sign of bad programming style; remember, you are not programming in Assembler any more. Most of the time a "break" or "continue" statement can substitute the goto; other time, the duplication of cleanup code, or the introduction of a clean up function.

The following is an other, quit extreme, example from the Max source code:

```
    void
    fun(...)
    {

    for (i = 0; i < MAX; i++)
        {
          .....
          if (error())
             goto exit;
        }

    exit:
        return;
```

```
}
```

No `for' loops with empty body

There should be no for loops with an empty body. A for with an empty body hide it's real purpose in the loop controlling clauses; it should probably be rewritten as while, to make a lot clearer it's purpose, that programmer look for in the loop body, and not control.

For example, the statement:

```
for (p = list; p->next; p = p->next);
```

Should be written as:

```
p = list

while (p->next)
  p = p->next;
```

That put in evidence that the purpose of the loop is to advance the pointer (the body of the loop), until we don't have successor (the loop control clause).

Only iteration variable in the `for' clauses

The "for" statement initialization and iteration clause should be used to modify only the iteration variables, i.e. the variables tested inside the condition of the for.

Examples:

```
for (p = table, i = 0; i < table_size; i++, p++)
        .....
```

Is wrong, because the variable p do not define the loop, but is part of the algorithm the loop execute; seeing p in the iteration and initialization condition suggest that the loop is defined in term of the pointer, while is defined in term of the counter i.

In general, too many initialization or iterations clauses are usually a bad programming style indication.

The comma operator

The C comma operator is allowed only inside "for" clauses. It should not be used as a substitute for a C block.

Using commas instead of blocks prevent the use of macros that expand to blocks. For instance, in the following example (taken from Max/FTS original sources), defining `outlet_int` as a macro instead of a function leads to a syntax error :

```
if (x->s_vel || !x->s_sust)
    outlet_int(x->s_out2, x->s_vel),
    outlet_int(x->s_ob.o_outlet, n);
else
    x->s_slink = slink_new(x->s_slink, n);
```

This should be coded as:

```
if (x->s_vel || !x->s_sust)
  {
    outlet_int(x->s_out2, x->s_vel);
    outlet_int(x->s_ob.o_outlet, n);
  }
else
  x->s_slink = slink_new(x->s_slink, n);
```

Expressions

Bit operators

The use of Bitwise operators used to implement arithmetic operations is forbidden.

A typical example is using the bitwise and operator to implement a modulo operation with a power of two constant, on the assumption that the and operator is faster then the modulo (division) operator. What this argument forgot is that all the modern compiler is able to automatically perform this kind of optimization; the programmer, as said at the beginning, should concentrate in writing readable code, and let the compiler do low level optimizations.

Examples:

```
#define SIZE 4096
#define MASK SIZE-1

i = (i + 1) & MASK;
```

Should be code as

```
#define SIZE 4096

i = (i + 1) % SIZE;
```

This coding would work also with a SIZE different from a power of two.

Another example taken from the Max is performing a comparison with a power of two by means of a bitwise and; first, we do not known any architecture were this would be possibly faster than a standard comparison, and second, the compiler would find out that by its own.

The code using this kind of tricks is completely unreadable, like in:

```
if (size & SIZE_MASK)
```

The semantic of the test is not known unless you do know the value of SIZE_MASK.

Bit operators should be left for real bitmask oriented operations.

Pointers and integers

The following is another good example taken from the original Max/FTS code :

```
sig = (t_sig *)(av[1].a_w.w_long);
```

Such expressions are guaranteed to be non portable.

Naming

Structure fields names

We recommend to don't follow the naming convention given in the Max Opcode external object programmers manual about structure fields.

An example of this naming convention is the following:

```
struct foobar
{
   int f_foo;
   int f_bar;

   struct foobar f_next;
};
```

Structure field name are prefix with a letter corresponding to the structure name initial and by an "_"; the Opcode documentation report this as a "UNIX" programming convention.

Actually, it is not, and is a very old (70s) C programming convention coming from the old times where the name space of structure fields was unique, so all the structure fields defined in the same program had to be different (so the prefix).

This convention in not used since the late 70s, and all the C compilers follow the rule that every structure have its field name spaces, so field name can be reused in different structure definition.

Avoiding this convention will made the code a lot clearer.

Variables and functions names

Variable (also local variables and structure fields) and functions names, should be significant; don't use variables with funny but meaningless names for your amusement, like "shit", for example (an other classic in MAX original sources).

In particular, we recommend that you don't use the "x" name inside object methods to identify the object itself; "x" is not meaningful as an object name, and it often induce confusion in objects where a geometrical space is used in the semantic (x coordinates, for examples).

You should use a name clearly referring to the object, like in the classic object oriented languages, like "self", "this" or "me", for example; we use the "this" name, following the C++ rules.

Code presentation

Coding indentation style

The Fts team use the vertical indentation style; i.e. any open "{" or closed "}" is on its own new line.

A function should be indented like this:

```
        <return type>
        <function-name> ( <args> )
```

```
{
   <declarations>

   <body>
}
```

All the code supported by this team will be converted to this indentation style; we recommend, if possible, to use this indentation style for the submitted sources.

Commented code

There should be no commented code at all. I.e. nothing like:

```
x = x + 1;

/* printf("The x value is %d", x); */
```

Comments on a line

Comments to the right of a code on the same line should comment *only* that line; comments for blocks or functions or that refer to more than one line of code should be on one (or more) line by themselves.

Correct Example:

```
  new_size = old_size * GROW_RATE;   /* compute the new size */
  alloc_size = new_size - old_size;   /* find the needed new memory */
```

Wrong example:

```
if (x > 3)    /* If we have more than 3 foobar, frob them */
{
    ... frob  foobar ...
}
```

This example should be coded either:

```
/* If we have more than 3 foobar, frob them */

if (x > 3)
{
    ... frob  foobar ...
}
```

or:

```
if (x > 3)
{
    /* If we have more than 3 foobar, frob them */

    ... frob  foobar ...
}
```

Debug code

There should be no uncommented "forgotten" debug code.

There should be no #ifdef'ed debug code, unless:

1. There is a precise reason to leave it there after that the code has been debugged.
2. The #ifdef argument is a symbol, (i.e. don't use #if 0) that is documented in a comment at the beginning of the file, for *all* the file where it is used.

For example, this is not allowed:

```
        x = x + 1;
#if 0
        printf("The x value is %d", x);
#endif
```

While this is allowed:

```
        x = x + 1;
#ifdef INCREMENT_TEST
        printf("The x value is %d", x);
#endif
```

if and only if the following (or a similar comment) appear at the head of the file:

```
/*
        Compilation Flags

        INCREMENT_TEST  enable the code to test the frobar incrementer.
                        Left here because the frobbar implementation will
                        soon change and we need to keep the test.
*/
```

Tab size

If tabs are used in source, they have to be equivalent to 8 chars; some editor allow you to set a different tab size; this is a bad idea, because almost any other UNIX tool assume a 8 char tab size.

If your editor does not allow you a nice formatting of C code with any tab size, we suggest you to use a better editor (namely, GNU Emacs).

Memory management

Freeing memory

Care should be taken in assuring that the allocated memory is freed when not needed anymore.

There are two typical cases we detected in the Max and MaxLib sources:

```
        {
          char *p;
```

```
        p = malloc( ....);

       if (error)
              return;
      }
```

In this example, some memory is allocated, then an error test is performed, and if an error situation exist the function return, without freeing the memory; wile not very likely, it is n example of a memory leak difficult to catch.

Another more frequent and dangerous problems have to do with Max externals: sometimes, and object allocate some buffer at creation time, and forgot to free them when the object is destroyed.

Compilation

ANSI C

The code must be compiled with a full ANSI compliant compiler.

Some compilers provides a flag to signal calls to non-prototyped functions. If available, this flag must be set.

Compilation warnings

You code should generate no warning when compiled with gcc with the "-Wall" option, or equivalent flags for other compilers, or at least no warning you know how to fix.

This option enable all the possible warning the compiler can issue, including also a lot of style related warning; actually, many of the guidelines reported in this document are covered by this compiler option.

If you don't use gcc, and your compiler do not provide similar feature, use lint to check your source code.

Authors : Maurizio De Cecco, François Déchelle