# FreeLC - API Programmers Guide


## by Hugh Jack (copyright2000 under GPL)


## Warning:

This software is educational and experimental and is not suitable for factory floor control


## Abstract:

This guide is intended for those who want to write new programs that include the FreeLC engine using languages such as C. This guide will explain how to interface to the controller at different levels for applications that embedded, or GUIs.


## 1. Introduction

### What is an API?

It is an Application Programming Interface. In practical terms it is a simple set of function calls designed for a user program. For example the GUI (Graphical User Interface) API uses fewer than 10 function calls, but these can be adapted to different applications.

### Why an API?

The programming interfaces are provided so that simple programs can be created without having to understand all of the underlying code. They also allow the original code to be upgraded or replaced with other code without having to rewrite the other applications.

## 2. The Architecture of FreeLC

The architecture of the entire FreeLC project was designed with modularity in mind. Figure 1 shows the major modules, and how they are connected. The ladder logic under the controller class can be used in an embedded environment. Examples of the embeddable API can be seen in the program ‘test.cpp’. Lower level modules are also available, and some examples can be seen in ‘testprog.cpp’, but I don’t recommend you do anything at this level, because I do plan to make modifications that will change these functions. You should also expect the functions in ‘test.cpp’ to change somewhat in the next few revisions.

The upper level has been designed so that it can be used for developing a graphical interface. To ease the production of a graphical interface I have included some ladder logic drawing routines.

But, to use these you will need to supply a few simple drawing routines in the file ‘x_gui_lowlevel.cpp’. Your main windowed program should be set up in ‘x_gui’, or whatever you want to call it. It will then call functions in ‘gui_api’ to make changes and get back information. The needed functions will be described in greater detail later.

When applying this engine to a real hardware you will need to add drivers to ‘io.cpp’. I am hoping that the Puffin project produces a high quality IO scanner that can be used here.

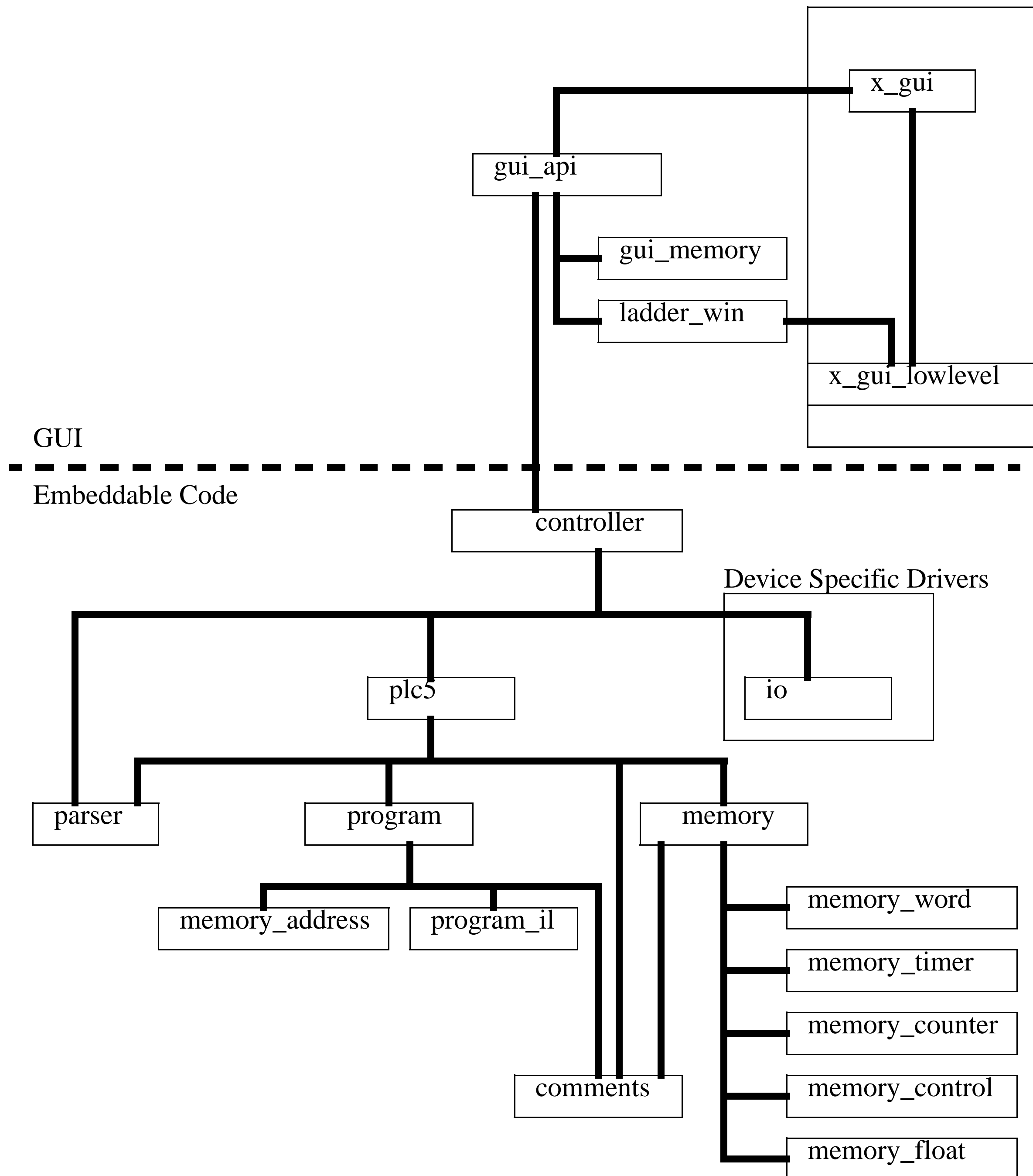


Figure 1 - Hierarchy of Major Classes and Modules

## 3. The File Format

I will discuss the file formats first. This is primarily because it makes the format of the program and data memory clear. A sample program file is shown in Figure 2. The equivalent ladder logic is shown to the right. The first statement identifies the start of a new program file '#PROGRAM', the memory location '2' and the program type '_IL_PROGRAM'. The first instruction 'SOR' marks that start of a rung. If this and the 'EOR' end of rung statements are left off, the logic engine will still work, but you will not be able to draw the program as ladder logic. In the logic engine the 'SOR' pushes a true on the stack, and sets a flag indicating the logic is in ladder form. The next instruction pulls the top value from the stack, performs an AND operation with a memory location, and pushes the results back on the stack. The 'OTE' instruction will retrieve the bit from the top of the stack, and set to memory location. If the logic engine is in 'ladder mode' then the bit will be pushed back on the stack.

In the second rung a branch instruction is encountered for the first time. When the 'BST' command is encountered the value on the top of the stack will be pulled off, and then pushed back on twice (so there are now two copies there). After the logic on the main branch is completed the 'NXB' instruction will exchange the top two results on the stack, and then solve the branch logic. At the end of the branch the 'BND' statement will pull two values from the top of the stack, and perform an OR operation. The 'XIO' operation is similar to the 'XIC', except that it inverts the value retrieved from memory. The following statement is a GT greater than comparison. If the result on the top of the stack is true, it will execute. This function will pull two values from memory, and if the first is greater than the second it can be true. If the comparison was executed the results will be pushed on the stack. Finally, the rung ends with our old friend the timer. The last rung contains an END statement. These should be at the end of each program to ensure proper termination. If it is not there there should be a warning.

```
#PROGRAM 2 _IL_PROGRAM
SOR
XIC I1:0/0
OTE O0:1/0
EOR
SOR
XIC I1:0/1
BST
XIO I1:0/2
NXB
XIC I1:0/3
BND
GT N7:0 N7:1
TON T4:0 0.010000 20 0
EOR
SOR
END
EOR
```

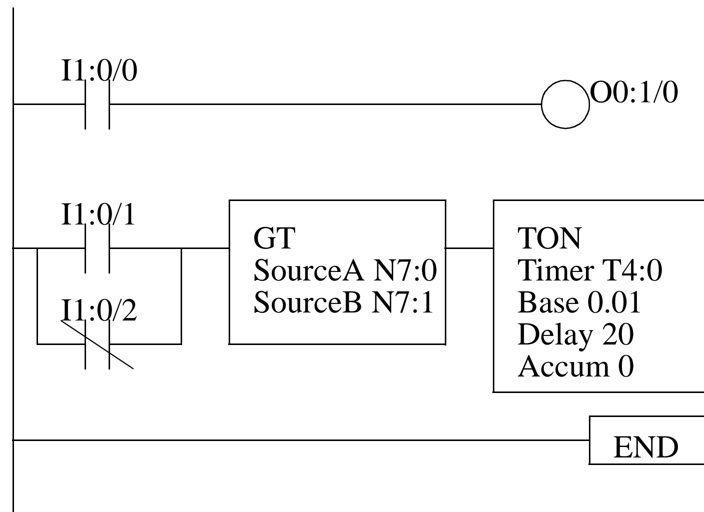


Figure 2 - A Sample Program File

The memory file is shown in Figure 3. The memory blocks begin with '#MEMORY', and the memory block number follows '0'. The type of memory follows this '_OUTPUT', and finally the size of the memory. The following lines contain values to be copied in, or written from the memory. When the PLC is started, it defines the allen bradley default memory types from 0 to 8.

```
#MEMORY 0 _OUTPUT 8
0 0 0 0
0 0 0 0
#MEMORY 1 _INPUT 6
0 0 0 0 0 0
#MEMORY 7 _INTEGER 4
0 0 0 0
```

Figure 3 - A Memory File Example

## 4. The GUI API

Every windowing interface has common concepts. But, even though two different systems may look the same the programs remind me of a million line BASIC program. So...... I have set up the

API so that you can use a minimal number of function calls in your program to implement a GUI front end for the logic engine. For you to write the GUI you need to write two pieces. The low level file is called ‘x_gui_lowlevel.cpp’. It contains the functions described in Figure 4. These will be called by the API. You also need to modify the file ‘x_gui_lowlevel.h’ to include the windows handles needed to draw in the ladder logic window.

- win::win() - a constructor for the class
- win::~win - a destructor for the class
- win::draw_line(int x1, int y1, int x2, int y2) - a line drawing function that will draw a line from pixels (x1, y1) to (x2, y2)
- win::draw_circle(int x, int y, ind d) - a circle drawing utility that will draw a circle of diameter ‘d’ pixels centered at (x, y)
- win::draw_text(int x, int y, char *text) - a routine to draw a string ‘text’ starting at pixel location (x, y)
- win::text_size(char *text, int &width, int &height) - a text string size calculator. When this function is passed a string ‘text’, it will return a width and height of the string in pixels.
- win::foreground(int color) - a function to change the color of the drawing pen to ‘color’. The current values are ‘_REGULAR’, ‘_BACKGROUND’ and ‘_HIGHLIGHT’. Right now these functions are not used effectively, but color will become essential in a release soon.

Figure 4 - Low Level GUI Functions

You will also need to set up high level routines that will set up the windowed interface, and then manage it. There function calls that can be made are listed in Figure 5.

- gui_api::gui_api(char *file_name) - the constructor can also open a set of files for a project with the name ‘file_name’
- gui_api::~gui_api() - the destructor will shut things down and deallocate memory
- gui_api::command(int op, int a1, int a2, int a3, int a4, int a5, char *text) - the command will used given operations ‘op’ to determine how to apply the variable arguments. This will be explained later.
- gui_api::request(int op, int a1, int a2, int a3, int &result, int *text) - this function will return data for use in other function. This will also be described later.
- gui_api::window_get(int prog, win*W) - this function will give the high level winowing functions access to the low level windowing function in the ‘x_gui_lowlevel.h’ definition. I tried to avoid global variables.

Figure 5 - High Level GUI Functions

The commands that can be issued to the API with the 'command' function are described in Figure 6. The API allows multiple interface windows to be defined, so when dealing with ladder logic, you can have over 10 windows. When calling ladder logic oriented functions, the second arguments will be 'window'. For a simple interface, leave this value as '0'. But, for more advanced GUIs you may want to have some specialized switching, or ladder logic windows viewable simultaneuously.

- (API_DELETE_LADDER_WINDOW, int window, int prog, 0, 0, 0, NULL) - This will delete a ladder logic program, from 'window' with program number 'prog'.
- (API_CREATE_LADDER_WINDOW, int window, int prog, 0, 0, 0, NULL) - This will create a new window, and associate it with program file number 'prog'.
- (API_SET_LADDER_LINE, int window, int line, 0, 0, 0, NULL) - This will set ladder 'window' to start drawing the ladder logic at 'line'.
- (API_DRAW_LADDER, int window, 0, 0, 0, 0, NULL) - This routine will use the low level gui routines to draw the ladder logic for 'window'.
- (API_LADDER_SELECT_BOX, int window, int x1, int y1, int x2, int y2, NULL) - Given a box defined by corners (x1,y1) and (x2,y2), this function will select instructions to focus on. The resulting selection box will be shown on the ladder diagram. It is used later by the edit functions.
- (API_EDIT_LADDER, int window, int operation, 0, 0, 0, char *text) - The edit function will perform operations based upon the selection box. Commands that can be used are '__INSERT', '__APPEND', '__REPLACE', '__DELETE', '__BRANCH', '__RUNG', '__COPY', __PASTE'.
- (API_EDIT_MEMORY, int window, int operation,  int file, int location, 0, char *text) - This function will interpret the text string 'text', and place the value in memory 'file' and 'word'. The permitted operations include '__EDIT_DELETE_MEMORY', '__EDIT_CREATE_MEMORY' and '__EDIT_CHANGE_MEMORY'.
- (API_SAVE_PROGRAMS, 0, 0, 0, 0, 0, char *name) - This will save all programs to the given file name.
- (API_SAVE_MEMORY, 0, 0, 0, 0, 0, char *name) - This will save all of the memory to the given file name.
- (API_SAVE_IO, 0, 0, 0, 0, 0, char *name) - This will save all of the IO configuration to the given file name.
- (API_LOADS_PROGRAMS, 0, 0, 0, 0, 0, char *name) - This function will load programs from the given file name.
- (API_LOAD_MEMORY, 0, 0, 0, 0, 0, char *name) - This function will load memory from the given file name.
- (API_LOAD_IO, 0, 0, 0, 0, 0, char *name) - This function will load IO configurastions from the given file name.(API_PLC_SCAN, 0, 0, 0, NULL, NULL) - This will cause one scan of the ladder logic.

Figure 6 - Argument Lists for Calls to 'command'

Figure 7 shows the requests that can be made with the API. As before, when ladder logic is being interrogated, you will have to supply a window number. If you always use '0', you can only have one ladder logic window, but it will be simpler.

- (API_GET_LADDER_SIZE, int window, 0, 0, NULL, NULL) - This function will return the total number of lines for the program in 'window'.
- (API_LADDER_DESCRIPTION, int window, int number, 0, NULL, char *text) - This will return a description of all ladder logic and return a text description.
- (API_GET_MEMORY_VALUE, int file, int word, 0, NULL, char *text) - This function will get a value from the memory at 'file' and 'word' and return it as a printable text string.
- (API_GET_MEMORY_DESCRIPTION, int file, 0, 0, NULL, char *text) - This will get descriptions for memory files that can be printed.
- (API_PROGRAM_SIZE, 0, 0, 0, NULL, NULL) - This will return the maximum number of program files in the PLC.
- (API_MEMORY_SIZE, 0, 0, 0, NULL, NULL) - This will return the maximum numbers of memory files in the PLC.

Figure 7 - Request Argument Types

## 5. How The Logic Engine Works

The logic engine resides in 'plc5.cpp'. This section will be written after development of the basic logic engine is complete.

## Appendix A - Functions

This is a list of functions, crudely cut and pasted from the source code. Whe it is complete I will 'pretty it up' and add detailed descriptions.

ACS(Source, Dest) - ArcCosine
ADD(SourceA, SourceB, Dest) - Addition
AFI() - Always False
ANB() - AND Using Stack
ASN(Source, Dest) - ArcSine
ATI() - Always True
ATN(Source, Dest) - ArcTangent

AND(Source) - AND Stack with Memory
AVE(File, Dest, Control, Length, Position) - Average
BAND(SourceA, SourceB, Dest) - Boolean AND
BOR(SourceA, SourceB, Dest) - Boolean OR
BND() - Branch End
BNOT(Source, Dest) - Boolean Not
BST() - Branch Start
BXOR(SourceA, SourceB, Dest) - Boolean XOR
CLR(Dest) - Clear
COS(Source, Dest) - Cosine
CTU(Counter, Preset, Accumulator) - Count UP
CTD(Counter, Preset, Accumulator) - Count Down
DEG(Source, Dest) - To Degrees
DIV(SourceA, SourceB, Dest) - Divide
END() - End of Program
EOR() - EOR with Stack
EQ(SourceA, SourceB) - Equals
FRD(Source, Dest) - From BCD
GT(SourceA, SourceB) - Greater Than
GE(SourceA, SourceB) - Greater of Equals
JMP(Label) - Jump to Label
JSR(Program) - Jump to Subroutine
LBL(Label) - Label
LD(Source) - Load Stack
LE(SourceA, SourceB) - Less or Equal
LIM(High Limit, Test Value, Low Limit) - Limit Test
LN(Source, Dest) - Natural Log
LOG(Source, Dest) - Base 10 Log
LT(SourceA, SourceB) - Less Than
MEQ(SourceA, Mask, SourceB) - Masked Equals
MOV(Source, Dest) - Move Data
MUL(SourceA, SourceB, Dest) - Multiply
MVM(Source, Mask, Dest) - Move Masked
NE(SourceA, SourceB) - Not Equal
NEG(Source, Dest) - Negative
NOP() - No Operation
NOT() - Not Stack
NXB() - Next Branch
OR(Source) - OR Stack with Memory
ORB() - OR with Stack
OTE(Dest) - Output Enable
OTL(Dest) - Output Latch
OTU(Dest) - Output Unlatch
RAD(Source, Dest) -To Radians
RES(Dest) - Reset
RET() - Return from JSR

RTO(Timer, Base, Preset, Accumulator) - Retentive Timer OFF
SIN(Source, Dest) - Sine
SOR() - Start of Rung
SQR(Source, Dest) - Square Root
SUB(SourceA, SourceB, Dest) - Subtract
TAN(Source, Dest) - Tangent
TND() - Temporary End
TOD(Source, Dest) - To BCD
TON(Timer, Base, Preset, Accumulator) - On-delay Timer
TOF(Timer, Base, Preset, Accumulator) - Off-delay Timer
XIC(Source) - Examine if Closed (NO)
XIO(Source) - Examine if Open (NC)
XOR(Source) - XOR Stack and Memory
XPY(SourceA, SourceB, Dest) - X to Power Y