

PyASM User's Guide V. 0.3

by Grant Olson

Introduction

PyASM is a full-featured dynamic assembler written entirely in Python. By dynamic, I mean that it can be used to generate and execute machine code in python at runtime without requiring the generation of object files and linkage. It essentially allow 'inline' assembly in python modules on x86 platforms.

PyASM can also generate object files (for windows) like a traditional standalone assembler, although you're probably better off using one of the many freely available assemblers if this is you primary goal.

Installation

PyASM requires python 2.4 or later. To the best of my knowledge, the only 2.4 specific feature I've used is *from x import (a,b,c)*, but I need to draw a line in the sand somewhere as far as the installations I'll test and support.

Linux Install:

- Download and extract [pyasm-0.2.tar.gz](#)
- python setup.py install

Windows Install:

- Download and run [pyasm-0.2.win32-py2.4.exe](#)
- A source distribution [pyasm-0.2.zip](#) is available, but you'll need VS7.0 to compile the excmem module.

Hello World!

A simple Windows version of a hello_world.py program is as follows:

```
#
# Hello World in assembly: pyasm/examples/hello_world.py
#
#

from pyasm import pyasm

pyasm(globals(),r"""
    !CHARS hello_str 'Hello world!\n\0'

    !PROC hello_world PYTHON
    !ARG self
    !ARG args

    PUSH hello_str
```

```

        CALL PySys_WriteStdout
        ADD ESP, 0x4
        MOV EAX,PyNone
        ADD [EAX],1
!ENDPROC
""")

```

```
hello_world()
```

A brief description of what is happening durring the pyasm call:

1. the globals() statement tells pyasm where to bind newly created python functions
2. The !CHARS directive creates a string constant.
3. The !PROC and !ARG directives create a procedure that matches the standard CPythonFunction signature [PyObject* hello_world(PyObject* self, PyObject* args) and create procedure initialization code.
4. The procedure calls python's PySys_WriteStdout function. Since python functions use CDECL calling conventions, we:
 - a) PUSH the paramters onto the stack from right to left
 - b) CALL the function
 - c) Cleanup the stack ourselves
5. PyCFunctions must return some sort of python object, so we:
 - a) Load PyNone into the EAX register, which will become the return value.
 - b) Add one to the reference count
6. The !ENDPROC directive ends the procedure and creates function cleanup code.

Block quote ends without a blank line; unexpected unindent.

This creates a procedure called hello_world that would have the C signature of *PyObject* hello_world(PyObject* self, PyObject* args)*. The procedure loads hello_str onto the stack, calls the python interpreters PySys_WriteStdout function,

7. Calling hello_world() executes the newly created function.

Everyday usage

Assembler Syntax

Like most assemblers, the command-line assembler contains a very simple parser. There two basic statements that can be used. An *instruction statement* and an *assembler directive*. *Assembler directives* contain information that makes your assembly a little easier to read than raw assembly code, such as the begining and ending of function; declaration of parameters, variables, constants and data; and other stuff. *Instruction Statements* consist of real assembly instructions such as *MOV [EAX+4],value*

Additional notes specific to this assembler are as follows:

- Numbers use python's formatting scheme, so hex is represented as 0xFF and not FFh.

Instruction Statements

Instruction statements are reasonably straightforward if you know x86 assembly language. Instruction mneumonics and register names are in all capitals.

Assembler Directives

Assembler directives begin with an exclamation mark, followed by the directive itself, and followed by any applicable parameters. Keep in mind that these directives are provided for the programmer's convenience. Anything that is done via a directive could be translated into raw assembly, it's just not as readable.

Text Directive	API Call	Description
!LABEL name	.AIL(name)	Provide a symbolic label to the current memory address. Primarily used for loops, if-then logic, etc. You can use a label and hand-roll a procedure, but you probably want to use the !PROC directive instead.
!PROC name[type]	.AP(name, ** type)**	Begin a procedure. This will emit the boilerplate code to start a procedure. Arguments and Local variables can be declared with !ARG and !LOCAL directives listed below. These declarations must occur before any instruction statements or an error will occur. This will generate the boilerplate function startup code, which consists of PUSHing the EBP register, copying the current location of ESP, and translating arguments and local variables into references via the offset of the EBP pointer. If the previous sentence didn't make any sense to you, just remember that the EBP register shouldn't be manipulated in your code here or things will get screwed up.
!ARG argname [size]	.AddArg(name)	An argument passed to a procedure via the stack. By default, we assume the size is 4 bytes although you can specify if you need to.
!LOCAL varname [size]	.AddLocal(name)	A local variable maintained on the procedure's stack frame.
!ENDPROC	.EP()	End a procedure. Emit the cleanup code as the caller's responsibility.]
!CONST name value	.AC(name,val)	Just declares a constant that is replaced in subsequent occurrences. Keep in mind that this is resolved at compile time, so the values should really only be numbers. !CONST hello_world "hello world\n\0" is invalid.
!CHARS name value	.AddStr()	Create a character array (aka a string)
!COMMENT text	n/a	Ignore this line.

Typical Usage

Typically, usage is as simple as the hello world example listed above. Import the pyasm function from the pyasm package and call it. globals blah blah blah.

Assembly via the API

calling pyasm is fine if you're just trying to inline some assembly functions, but if you're trying to dynamically generate assembly (such as writing a python compiler) you're better off accessing the api directly. This involves a few steps:

- 1) import the assembler class from x86 asm and instantiate.
- 2) Add instructions either as strings that need to be preprocessed or via the api.
- 3) generate an intermediate 'codepackage' by calling the .Compile() method
- 4) transform the codePackage to runtime memory via CpToMemory.

Command-line assembler

NOT IMPLEMENTED YET

If you really want to, a command-line assembler is available for usage. Usage is straightforward:

```
python pyassemble.py asmfile.asm
```

This will generate an object file asmfile.o that can be used by your linker.