

# c/360 Guide *(draft 1)*

Robert W Senser, PhD

January 20, 2024

## Abstract

Here we describe our c/360 C compiler and run-time system that can be used to execute C programs under the legacy IBM mainframe “Multiprogramming with a Variable number of Tasks” (MVT) operating system. MVT supports batch processing, some TSO and does NOT have virtual memory. Supplied is a PC-based, full tool chain and included are scripts to utilize the toolchain, installation verification programs (IVPs) and sample programs. c/360 forms a non-commercial C compiler and supporting libraries, which execute C code under MVT.

## 1 Introduction

The IBM mainframe line of computers has been available in many variations. One of the early variations was the S/360 line. This line was commonly used from the middle 1960s through the mid-late 1970s. The goal of this project is to provide a light-weight C compiler tool chain that generates mainframe assembler code suitable for use on an emulated 360 computer, running the MVT operating system[1].<sup>1</sup>

## 2 Why C for a 50-year-old Mainframe?

*Why go through the effort to form this C compiler tool chain?* Answer: To be able to code new programs for MVT using something more efficient than FORTRAN, COBOL or Assembler.<sup>2</sup> This need is similar to legacy MVS 3.8 using “KICKS” to provide CICS-like functionality.

---

<sup>1</sup>“light-weight” implies a compiler that is not intended for commercial use but instead for “hobby” and educational use.

<sup>2</sup>And, perhaps, just because we can!

### 3 Overview of Approach

Our approach uses an existing PC-based MVS 370 cross compiler (**gccmvs.exe**), which supports the C99 standard, to generate 370 assembler code[2]. Following the compile a utility program, called the “Expander” (**c360expand.py**), converts the 370 Assembler code to 360 “Assembler F” code and adds our run-time support. This run-time support is light-weight and not all the usual C99 run-time features are supported. Note that support is provided for a simplified **printf()** function and for the common **strxxx()** functions.

We use an “embedded” C-compiler style. This means that the generated program code does not depend on the dynamic finding and loading of C modules. Said differently, all C run-time requirements are provided in source form on the PC and are compiled into the resulting mainframe assembler file. The various headers files provided contain the C run-time function headers AND the function’s C code implementation. The c/360 “.h” files include the code normally associated with the “.c” files! As an example, the **strlen.h** file includes the C code that would normally be in a **strlen.c** file. See Section 13.2 for more details.

### 4 Installation

Listed below are the general installation steps. Notice that nothing is pre-installed on the S/360. Once the assembler code from the Expander step is produced, it is copied to the S/360 (emulator) and executed.<sup>3</sup>

1. Download Github Project (See Section 6)
2. Decide which IBM mainframe emulator(s) to use  
We tend to first test with z390 and then with a Hercules-based system.
3. Customize Scripts (bat files) for PC use  
This normally amounts to changing PATH in the **setPath.bat** file.
4. Run the Installation Verification Programs  
These are discussed in Section 13.4, it shows how to use the scripts customized above.
5. Enjoy!<sup>4</sup>

### 5 c/360 Environment

Listed here are the components that makeup the C compiler environment. Note that the Expander, light-weight Run-Time Library, Scripts and Programs were

---

<sup>3</sup>Knowing both the C language and mainframe basics, especially Assembler, is very helpful.

<sup>4</sup>We have produced several small C applications with this tool chain and they function as expected. In one case, a 64-line C program resulted in 2,309 lines of Assembler!

generated specifically for this environment. The Installation Verification Programs (IVPs) can be executed to show that the C compiler is functioning correctly.

1. Compiler: Existing **gccmvs.exe** Cross Compiler(Windows version)[3]
2. Expander: **c360expand.py** (see Section 13.1)
3. C Run Time: Run-Time Library for S/360 and MVT (see Section 13.2)
4. Assembler(s): MVT Assembler and optional z390 Emulator Tool
5. Specialized “Build” Scripts (Windows “doxxxxxx.bat” files)
6. Test and IVP Programs (see Section 13.4)

## 6 c/360 GitHub repository

The c/360 GitHub URL is <https://github.com/rwsenser/c360b1>, with c360b1 being the “beta1” version. The release version will be <https://github.com/rwsenser/c360>. See Section 13.3 for details. This repository is closed to external updates. If you have suggested code changes, please contact {0x72 0x77 0x73 0x65 0x6e 0x73 0x65 0x72}@gmail.com, hex values between { and } are ASCII chars.

## 7 Helpful Tools

### 7.1 z390

This tool has worked well on our Windows 10 PC. We usually test a newly compiled program first with z390 before moving the code to a Hercules-based emulator. z390 is not perfect but we find it very useful[4].

### 7.2 webMVT

This is a web-based S360 MVT emulator, which utilizes the Hercules emulator, created by the author[5]. This MVT runs entirely in the browser and permits the uploading of new programs.

### 7.3 Notepad++

Just a very useful editor, especially when working with a mixture of source-code types.

## 8 c/360 Known Issues and Limits

1. Only supports the C99 Standard
2. Limited C run-time system
3. Does not link together separate C compiles
4. Only the first 8 characters of MVT Assembler and Loader symbols are meaningful<sup>5</sup>
5. The **puts()** and **printf()** library functions are limited to 120 characters of output and they ignore special characters like “newline”
6. The tool chain works only with Windows
7. The C run-time stack size is set in the PDPXXX.cpy modules (and is not trivial to change)<sup>6</sup>
8. The compiler optimization option -O2 causes issues<sup>7</sup>
9. The **gccmvs.exe** compiler is not bug free<sup>8</sup>
10. Optional tool z390’s DASD (file) support can be awkward to use<sup>9</sup>

## 9 Frequently Asked Questions

1. Why does the C Run Time have limited functionality?  
It takes time and effort to produce a C Run Time and as this is a non-commercial compiler, compromises were made.
2. How can additional functions be added to c/360 Run Time?  
See Section 13.2.2.
3. What C Run Time options are provided?  
The **PDPOPTS.cpy** file provides options for setting the **puts()** destination and use of the stack frame marker.
4. Are there any known coding limits with c/360?  
The C99 standard does not support the **for (int i = ....** style; use **int i; for (i = ....** instead. Only the first 8 characters of labels and variable names are used to in processing. For example, using variables named **xxxxxxxxa** and **xxxxxxxxb** in a program will result in an error.

---

<sup>5</sup>This means that only the first 8 characters of C labels are significant!

<sup>6</sup>Default stack size is around 5,120 bytes. The size is set in PDPMAIN.cpy AND PDP-TOP.cpy.

<sup>7</sup>We have not researched the cause of these issues.

<sup>8</sup>We encountered a compiler bug while building the **nanoprintf()** function and had to omit its floating-point support to avoid the bug. We have not yet attempted to diagnose this compiler issue.

<sup>9</sup>Having data files output in ASCII involves using a non-standard DCB.

5. What is a good approach to debugging with c/360?  
 In the core dump, look for the “STCK” literal that marks the stack control block (possibly pointed to by R2), look for stack frame markers (counts singled-digit X’F0’ thru X’F9’) in each stack frame and treat debugging as an Assembler, not C, task. When using z390, the **dorun** script accepts the “dump” option, for example **dorun str dump**. The “dump” option will cause a core dump to be created by z390.
6. How is c/360 licensed?  
 Some components are licensed separately, such as nanoprintf[[6] – see their respective licenses. Otherwise, c/360 follows the MIT License for open-source software.

## 10 Potential Enhancements

Potential future enhancements listed in no particular order:

- Support C labels over 8 characters in length via “name mangling,” performed in the Expander.
- Enable multiple C compiles to form one assembler module.
- Add **printf()** “%f” support.
- Provide a more complete set of C run-time libraries.
- Automatically “close” the **gets()** and **puts()** OS files at termination of **main()** execution.

## 11 Acknowledgments

The following people and tools are recognized in the creation of c/360:

- **gccmvs.exe**[3]  
 The basic compiler making all of this possible.
- z390[4]  
 Mainframe emulator making debugging much quicker and easier.
- webMVT[5]  
 Convenient way to run MVT without installing Hercules locally.
- Bruce Ray, president of Wild Hare Computer Systems[7]  
 For rekindling the author’s interest in legacy computing.

## 12 Conclusions

This darn thing works!

## 13 Appendix

### 13.1 Expander Description

The “Expander”, python program **c360expand.py**, performs these tasks:

1. Inputs the type “.s” source file created by **gccmvs.exe**.
2. Expands the “copybook”s referenced by the COPY instructions in the input file.
3. Trims trailing spaces from source lines.
4. Re-codes some Extended Mnemonic Instructions (ie., “BHR” to “BCR 2,”).<sup>10</sup>
5. Creates warnings for labels over 8 characters is width.
6. Outputs a new source file of type “.mlc”.

The Expander is executed via **doexpand.bat** and **dobuild.bat**.

Sample Expander Output:

Note: The ‘-b’ option provides the PATH to the copybooks.

```
>c360expand.py -i str.s -o str.mlc -b cpy\.
```

```
c360expand:
663 source lines read
5 COPYbooks processed
0 instructions recoded
0 warnings (long labels)
970 source lines written
processing completed
```

### 13.2 C Run-Time Description

#### 13.2.1 Run-Time Library Functions

Here is a partial list of C library functions provided with c/360:

1. **atoi()** (rwsatoi.h)
2. **gets()** (rwsgets.h)
3. **inttypes** (rwsinttypes.h)
4. **itoa()** (rwsitoa.h)

---

<sup>10</sup>We only support this re-coding for the actual extended instruction that we have encountered.

5. **memcmp()** (rwsmemcmp.h)
6. **memcpy()** (rwsmemcpy.h)
7. **puts()** (rwsputs.h)
8. **stdarg** (rwsstdarg.h)
9. **stddef** (rwsstddef.h)
10. **stdint** (rwsstdint.h)
11. **stdio** (rwsstdio.h)
12. **strcat()** (rwsstrcat.h)
13. **strcpy()** (rwsstrcpy.h)
14. **strlen()** (rwsstrlen.h)
15. **printf** (nanoprintf.h)

The first column above contains the C function name, or facility, and the second contains the c/360 header file name. These functions may contain different or less functionality than their standard C counterparts, hence the “rws” prefix. These differences are especially true with I/O functions and “deep internal” C definitions.<sup>11</sup>

### 13.2.2 Thoughts on creating addition C library functions

There is a saying, “Imitation is the sincerest form of flattery.” And, when I was a new COBOL programmer, there was a saying, “Only 2 new COBOL programs have ever been written, all the others were created via *cut and paste* from these first two.” On the other hand, copying someone’s work without attribution is plagiarism (theft). My suggestion is to look for an existing version of a given library function and then, licensing permitting and with attribution given, port it to c/360. Another possibility is to enhance an existing c/360 library function to provide what is needed. For example, create a desired **strncpy()** from the existing **strcpy()**.

## 13.3 Git Directory Structure

The most significant c/360 subdirectories are:

- **bat**: Windows “.bat” files
- **bin**: Executables (**gccmvs.exe**)
- **cpy**: Copybooks

---

<sup>11</sup>Perhaps it is wise at this point to reiterate that this is not intended as a commercial-grade C compiler.

- **ivp**: Installation Verification Programs
- **lib**: Library of C include files
- **pytuil**: Python code (**c360expand.py**)
- **smp**: Assortment of sample c360 programs
- **z390**: Empty directory to hold optional z390

### 13.4 Installation Verification Programs

Four Installation Verification Programs (IVPs) are provided in the c/360 Git repository, **ivp** directory. They are built and executed from the directory containing the IVP's source code by executing the following scripts. For "ivp1" (program **ivp1.c**) the following would be entered:

```
>dogcc ivp1
>dobuild ivp1
>dorun ivp1      <-- if using z390
```

Note: The **dobuild** script generates a ".jcl" file that can be moved to MVT. Submitting this JCL, which includes the Assembler code, will assemble, link and execute the program.

#### 13.4.1 ivp1

Here is a sample z390 output from **dorun ivp1**:

```
>dorun ivp1

** lines omitted **

12:12:30 ivp1      MZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:12:31 ivp1      MZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 76 IO=3836
12:12:31 ivp1      LZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:12:31 ivp1      LZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 14 IO=319
12:12:31 ivp1      EZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:12:31 ivp1      EZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 20 IO=50 INS=89
```

The last step of EZ390 that shows **RC = 0**, marks success. Just for reference, here is the **ivp1.c** source code:

```
//
// ivp1.c: Just verify that basic math is working (no functions called!)
//
int a = 1;
int b = 2;
int c;
```



```

float fa = 1.0;
float fb = 2.0;
float fc;
int errs = 0;
int main() {
    // do int math
    c = a + b;
    if (c != 3) errs++;
    // do float math
    fc = fa + fb;
    if (fc != 3.0) errs++;
    return errs;
}

```

### 13.4.2 ivp2

This second IVP is similar to the first but uses the **puts()** function to display output. Here is an example output. Again, an RC of zero marks success.

```
>dorun ivp2
```

```
** lines omitted **
```

```

12:33:36 ivp2      MZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:33:36 ivp2      MZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 76 IO=4120
12:33:37 ivp2      LZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:33:37 ivp2      LZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 15 IO=408
12:33:37 ivp2      EZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
integer add OK
float   add OK
12:33:37 ivp2      EZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 20 IO=68 INS=523

```

### 13.4.3 ivp3

This third IVP exercises more functions, such as **itoa()**, and it produces more results. Here is an example output. Again, an RC of zero marks success.

```
>dorun ivp3
```

```
** lines omitted **
```

```

12:45:17 ivp3      MZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:45:17 ivp3      MZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 76 IO=5369
12:45:17 ivp3      LZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
12:45:17 ivp3      LZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 19 IO=720
12:45:18 ivp3      EZ390 START USING z390 v1.8.0 ON J2SE 1.8.0_131 01/16/24
sum 1 + 2:

```

```

CPU0000 PSW=FFF4000EA0779532 24..... instcount=5
200241 ASP1 R= RWSTEST IVP4 RESULTS:
200241 ASP1 R= RWSTEST AIM: AIM
200241 ASP1 R= RWSTEST HIGH: HIGH
200241 ASP1 R= RWSTEST STRLEN() RESULT OK
200241 ASP1 R= RWSTEST INSERT ' '
200241 ASP1 R= RWSTEST BUFFER2 CONCAT RESULT IS:
200241 ASP1 R= RWSTEST AIM HIGH
200241 ASP1 R= RWSTEST STRCMP() RESULT OK
200241 ASP1 R= RWSTEST NUMBER OF ERRORS:
200241 ASP1 R= RWSTEST 0
RWSTEST STEP: RWSTEST GO ET=00.00.01 RC=0000
IEF429I INITIATOR 'ASPB01' WAITING FOR WORK
200242 ASP1 R= RWSTEST STEP: RWSTEST GO ET=00.00.01 RC=0000
200242 ASP1 R= RWSTEST IEF404I RWSTEST ENDED TIME=20.02.42
200244 AMSU11 OS/MVT ON ASP1 WAITING FOR WORK
200245 SPRO1 JOB 0030,RWSTEST IS ON PR1 (00E), LINES = 002076

```

Figure 1: MVT Output from `ivp4.c`

```

3
should display '1234'
1234
number of errors:
0
12:45:18 ivp3      EZ390 ENDED   RC= 0 SEC= 0 MEM(MB)= 20 IO=108 INS=2148

```

#### 13.4.4 ivp4

This fourth IVP uses the `strxxx()`, `atoi()` and `puts()` functions and has 52 lines of source code. Shown is an example output from webMVT, see Figure 1. Again, an RC of zero marks success.

### 13.5 Other Source Code Examples

The examples subdirectory (name: `smp`) contains addition source examples, including the `app.c` sample application.

## References

- [1] URL: [https://en.wikipedia.org/wiki/OS/360\\_and\\_successors#MVT](https://en.wikipedia.org/wiki/OS/360_and_successors#MVT).
- [2] URL: <https://gccmvs.sourceforge.net/>.
- [3] URL: [https://sourceforge.net/projects/gccmvs/files/GCCMVS/GCC%203.2.3%20MVS%209.0/gccmvs-3\\_2\\_3-9\\_0-win32.zip/download](https://sourceforge.net/projects/gccmvs/files/GCCMVS/GCC%203.2.3%20MVS%209.0/gccmvs-3_2_3-9_0-win32.zip/download).
- [4] URL: <https://z390.org/>.
- [5] URL: <http://www.rwsenser.com/ELS/beta1.htm>.

[6] URL: <https://github.com/charlesnicholson/nanoprintf>.

[7] URL: <https://www.wildharecomputers.com/>.