ReSource65

User's Guide

v. 1.00

A MOS Technology/Commodore 65xx Source Generator
for Microsoft DOS Platforms

Charles R. Bond
http://www.crbond.com

September 11, 2009

Chapter 1

Introduction

`ReSource65` is a programmer's aid which can be of great value in converting a binary object file to a form which can be hand edited and assembled. Whether the original source of the binary image is a computer file or ROM, as long is it can be read by `ReSource65` an assembler file can be produced.

Some of the major features of `ReSource65` are:

- Identifies and names target memory locations.
- Supports an auxiliary symbol file which resolves code vs. data issues.
- Allows input of known labels and addresses.
- Handles `.text`, `.byte`, `.word` and `.code` blocks.
- Generates a source file conforming to the syntax of many assemblers.

Even if nothing whatsoever is known about the object file at first, a viable assembler file can be generated. In the event that some information is available about calls to known addresses, this information can be provided to improve the process. Recursive application of `ReSource65` and expansion of the symbol file can lead to a complete sectioning of the assembler file into code and data blocks.

Chapter 2

Getting Started

2.1 Installation

To install ReSource65 simply unzip the archive into an appropriate folder. The files included in resource65.zip are:

ReSource65.exe – the executable program file.

ReSource65.pdf – this manual.

sample.bin – an example binary file.

ReSource65 can operate on raw binary files, such as those created when a ROM image is dumped to a file, or on files generated in the H65 format. The H65 file format is a printable format suitable for hand editing. It contains all the information in a pure binary file, but includes address location information. No checksums are used, so editing can be done without recalculating checksum values.

A portion of a typical H65 file is shown below.

```
B180 4C E9 DC 20 52 B0 A0 01 B1 21 D0 06 A9 F9 A2 FF
B190 D0 E4 AA 88 B1 21 86 22 85 21 4C 63 B1 86 87 A6
B1A0 2A A4 2B 86 55 84 56 E8 D0 01 C8 38 8A E5 34 98
B1B0 E5 35 90 03 4C 55 C3 84 2B 86 2A A0 01 A2 00 A1
B1C0 55 91 55 A5 55 D0 02 C6 56 C6 55 38 A5 55 E5 77
```

If no H65 file exists for the program under consideration, ReSource65 will create one from the raw binary file.

2.2 Running ReSource65

Generally, the first run of ReSource65 on a new program will operate on a raw binary file. This file must be a ROM image file with a .bin extender. The sample file provided with this implementation is named sample.bin.

The source generator built into ReSource65 actually operates on an H65 file, if one is not called it will be created from the binary file.

To run ReSource65 on a binary file simply type:

```
C:[path] resource65 sample.bin
```

at the DOS prompt. Note that [path] is the path to your chosen folder.

You will be prompted to provide a starting address for the ROM code. This is the only critical information that *must* be known for every program. Fortunately, the ROM start location is usually accessible information.[1]

When this run completes, a file named sample.h65 will have been created. Subsequent runs of ReSource65 should use this file to reduce the processing overhead.

In the event that a file in H65 format is available, invoke Resource65 with:

```
C:[path] resource65 sample.h65
```

2.2.1 Parameter Option

ReSource65 supports a single option on the command line. This option, which is simply an n, determines whether the original code will be appended to source lines as comments. For example, the following command line will cause the output file (*.asm) to omit adding the original code as comments.

```
C:[path] resource65 sample.h65 n
```

This command line switch *must* follow the filename.

[1] Actually, it is possible to guess at the starting location and recover the correct start by troubleshooting the result.

2.2.2 Source Generation

A first attempt at reconstructing the source file will be made on the first run of `ReSource65`. It will be named `sample.asm`. This file should compile with slight modifications on many 6502 assemblers. It will compile without modification on the `cbA65` assembler available as a companion to `ReSource65`.

The sample files also include `sample.sym`. This file is an example of the auxiliary file which must be generated by the programmer to aid in analysing the object file. Details are provide in Section2.3.

A typical sequence of events to regenerate a viable source code version of a ROM file will consists of these steps.

1. Run `ReSource65` on a ROM image file with a `*.bin` extender.
2. Examine the reconstructed source `*.asm` to identify code and data sections, where possible.
3. Create a `*.sym` specifying the start of code or data sections. (See Section 2.3 for instructions on manipulating `*.sym` files.)
4. Rerun `ReSource65` using the `*.h65` file.
5. Update `*.sym` if needed.
6. Repeat steps 4 and 5 until satisfied.

When the `*.asm` file is in acceptable form, it is now time to carefully analyze the symbols created by `ReSource65`.

Each symbol has the form `LONNNN`, where the `N`s stand for a 4-digit hexadecimal memory address preceded by a leading zero. The symbol naming convention preserves uniqueness of every address and makes identification of `ReSource65` labels easy, even with editor search functions.

The assembler source file begins with a list of symbolic labels which are not found in the body of the program. All labels used within the program represent accessible target memory locations inside the ROM[2] memory space.

If any labels are found which belong in the ROM memory space, look for undetected data areas or locations which were accessed using offsets. This is an opportunity to replace

[2]Note that the binary file could have come from a tape or disc file which loads into RAM.

the `ReSource65` generated labels with more meaningful ones which represent the original programmer's methods.

Auxiliary documents and materials which might identify common external memory storage locations or utility routines can be matched to the appropriate labels and used to replace them. If this step takes place during the iterative creation of the source file using `ReSource65` the symbol file can be updated for further processing. Otherwise, you are now at the point of fine-tuning the source file for use by the assembler.

2.3 Symbol Files

Symbols files contain user provided labels for important memory locations including starting address for subroutines, messages, tables, ports, etc.

In some cases the programmer will have access to no prior information about these locations. On occasion, a limited number of entry points or other locations may be known. In the best cases, comprehensive memory maps or access to other versions of the same software could be available. The basic rule is: *the less you know about the code internals, the more work you have to do to reconstruct a viable source code.*

2.3.1 Syntax

Symbol files support the following file block control commands:

`.code` – the following address starts a code block.

`.byte` – the following address starts a block discrete bytes.

`.word` – the following address starts a block of word values.

`.text` – the following address starts a block of ASCII text.

A typical symbol file will alternate control commands as in the following example.

```
.code $c000
.byte $c433
.code $c480
.word $c624
.code $c626
```

In addition block control commands, you may add symbolic labels for any discrete memory address as in the following example.

```
ptr = $80
temp = $92
scrn = 32768
errmsg = $c080
porta = $d840
prtchr = $ffef
```

These addresses may refer to storage locations, subroutine entry points, string start addresses, ports, etc. The addresses may be entered in decimal or in hex with a `$` prefix.

2.3.2 Sample File

Probably the easiest way to explain the evolution of a symbol file is to go through an example. The `sample.bin` file provided with this application will serve as a vehicle. It represents a small amount of code which occupies the lower part of a 256-byte memory block.

The first step is to run `ReSource65` on the sample file to produce `sample.h65` and `sample.asm`.

This can be with the following command line:

```
C:[path]>resource65 sample.bin
```

When prompted for the starting location, enter `c000`.

The resulting `sample.asm` file can be assembled 'as is' with `cbA65`,[3] or with minor modifications to suit other assemblers. But we would like to improve the readability of this preliminary source code version.

The next step is to create a `sample.sym` file which guides the process through code and data. Examine the file `sample.asm` and notice that there is one labeled address which is within the ROM memory space. This address bears the label `L0C047`.

Looking in the assembler file around this location we see that there are several invalid opcodes detected. The first, at `L0C03A` is clearly the start of a jump table or message block or some other data.

[3]Note that `ReSource65` retains the original code and data bytes as comments unless the command line switch `-n` was added.

Using your favorite text editor, create a new file named `sample.sym` and enter the following lines:

```
.code $c000
.byte $c03a
```

Save this file in the same folder as the other files, and run `ReSource65` on `sample.h65`. This time you will *not* be prompted for the starting location. The reason is that the `H65` file has that information already placed in the file.

An examination of the new `sample.asm` file reveals that the data starting at `L0C03A` is consistent with the ASCII character set. If you are familiar with ASCII codes you can verify this by eye, but in any case you can replace the `.byte` designator in the `sample.sym` file with `.text` to confirm it.

It is also clear that the ROM has been filled with zeros following the data. You can label this fill block by revising the `sample.sym` file as follow:

```
filler = $c054
.code $c000
.text $c03a
.byte $c054
```

After a little more sleuthing we conclude that location `$80` in page zero holds a pointer and, from previous knowledge of the system in which the ROM is be place, location `$8000` is the start of screen RAM. We can add labels for these to the `sample.sym` file so it looks like this,

```
ptr1 = $80
scrn = $8000
rdymsg = $c03a
scrnmsg = $c047
filler = $c054
.code $c000
.text $c03a
.byte $c054
```

where I have also added labels for the two strings.

Running `ReSource65` in `sample.h65` incorporates these improvements in the resulting `sample.asm` file. At this point, `ReSource65` has done about all it can, leaving the ad-

ditional improvements in the hands of the programmer. It is now time to focus on the directly modifying the assembler source file.

Here is one result of further analysis and improvement of the reconstructed source file.

```
;   sample.h65,  ReSource65: v 0.91a,  Mar  7 2008
ptr         .equ $0080
scrn        .equ $8000
dostr       .equ $ffe1

            .org $C000
            ldx #$77            ; C000 A2 77
            lda #$00            ; C002 A9 00
clrscr      sta scrn,x          ; C004 9D 00 80
            sta scrn+120,x      ; C007 9D 78 80
            sta scrn+240,x      ; C00A 9D F0 80
            sta scrn+360,x      ; C00D 9D 68 81
            sta scrn+480,x      ; C010 9D E0 81
            sta scrn+600,x      ; C013 9D 58 82
            sta scrn+720,x      ; C016 9D D0 82
            sta scrn+840,x      ; C019 9D 48 83
            dex                 ; C01C CA
            bpl clrscr          ; C01D 10 E5
            ldx <scrn           ; C01F A2 00
            stx ptr             ; C021 86 80
            ldx >scrn           ; C023 A2 80
            stx ptr+1           ; C025 86 81
            ldy #$0C            ; C027 A0 0C
            lda scrnmsg,y       ; C029 B9 47 C0
@           sta (ptr),y         ; C02C 91 80
            dey                 ; C02E 88
            bpl @B              ; C02F 10 FB
            lda rdymsg          ; C031 AD 3A C0
            ldy #$0C            ; C034 A0 0C
            jsr dostr           ; C036 20 E1 FF
            rts                 ; C039 60
rdymsg      .text "System ready."
scrnmsg     .text "Screen clear."
filler      .align 256,0
            .end
```

This will assemble with `cbA65` and produce a binary file identical to the original.

Although no comments have been placed in this source file, minimal commenting would not take much additional analysis or effort. Clearly, access to any source or system documentation would be helpful.

Chapter 3

Theory of Operation

`ReSource65` uses simple strategies to translate the binary file into readable form, but the implementation details deserve some clarification and comment.

3.1 The Reconstruction Process

In the first pass through a binary file, `ReSource65` assumes that the entire memory block consists of code and no data. Generally, this is poor assumption, but it yields information which is valuable in partioning the memory space into reasonable sections.

On examination of the first attempt at reconstructing a source file, i.e., the `*.asm` file, there are likely to be a number of invalid opcodes detected. These are indicated by `***` where a processor mnemonic would normally be found. Since the byte values at these locations are not valid opcodes, the must represent some kind of data.

`ReSource65` provides assistance to the programmer in the following ways:

- Disassemble as code blocks,
- Add the original code values as comments,
- Assign symbolic labels to all reference memory addresses,

The task of analyzing the data in the neighborhood of these invalid opcodes is part of the programmer's burden. An educated guess as to where the code block ends, the data block starts, and the code restarts are all that is needed to uncover the next level of detail.

Creating a symbol file, which must have the same name as the binary file but with a `.sym` extender, is the next step in the process.

The symbol file guides `ReSource65` through the binary file by designating the start of regions containing code or data. Data can be further specified as text, bytes or words. The directives corresponding to the recognized types are `.code`, `.text`, `.byte` and `.word`. An internal mode switching algorithm is used to assure that the text sent to the assembler source file is appropriate.

The symbol file also allows the programmer to provide meaningful names to memory locations. These user supplied names and locations will override any internally generated labels.

Running `ReSource65` and improving the symbol file constitute the interative process used to generate an acceptable assembler language source file.

3.2 Source File Structure

3.2.1 External Address List

`ReSource` outputs a list of all external addresses identified by disassembling code regions or by processing any user provided labels. The `*.asm` file associates the name with its address using the `.equ` directive. Most assemblers accept this convention.

The list of external addresses is sorted by `ReSource65` in order of increasing memory address.

Once the programmer identifies the role of any of these addresses, he can replace the automatically generated label with a more meaningful one by simply including a line such as `prtline = $ff2e` in the symbol file. For these equates `ReSource65` uses the equal sign.

3.2.2 ROM Contents

After the list of external names and addresses, `ReSource65` place an `.org` statement in the output file. This statement is followed by the address which was given during the first pass for the start of the ROM code.

Although not every binary object file can be completely analyzed and understood without external support documents, the work product is often useable for making modifications, corrections and additions to the original code.

Following the `.org` statement `ReSource65` will interleave blocks of disassembled code with data, as specified in the symbol file.

3.3 Conclusion

Like other programmer's tools, `ReSource65` has capabilities and limitations. There is certainly no substitute for the kind of intelligent analysis which an experienced programmer can bring to a reconstruction problem. The extent to which complex mental processes can be emulated by mechanical devices is a subject for Artificial Intelligence, and progress has been dismally slow.

Nevertheless well designed tools can make the programmer's job much easier. It is hoped that `ReSource65` will find its place as one of those tools.