15th January 2019 Martin Head

DisCharge Decompiler

DisCharge is a decompiler for SuperCharged & TurboCharged programs. It is still very much a work in progress, and as it is still in development, anything, and everything is libel to change at any time.

DisCharge cannot just convert an executable file back into a ready to run SuperBASIC program. It may require some manual intervention, and the resulting SuperBASIC program will need some tidying up before it will load and run correctly.

This document is an introduction to the steps required to convert a SuperCharged, or TurboCharged executable back into a SuperBASIC program.

DisCharge comes in two parts. A SuperCharge, and a TurboCharge decompiler. The TurboDischarge compiler is further divided into two programs. One for programs that were compiled with the ‘Include Nos’ option, and one for programs with the ‘Omit Nos’ option. Where line numbers are included, or omitted from the compiled executable.

SuperCharged and TurboCharged programs share many similarities, but there are enough differences to create complications, by trying to have one program deal with both types of compiled programs. Likewise for Turbo’ed programs with and without line numbers.

The two parts of Discharge can be identified by either starting with **Turbo/TC** or **Super/SC**, So the ProcessDump program for a SuperCharge program will be **SuperProcessDump_bas**, and for a TurboCharge program **TurboProcessDump_bas**.

In these instructions add Turbo/TC, or Super/SC to the start of any filenames as required.

There are 5 steps involved in converting a compiled program back to a SuperBASIC program.

Step 1 - Disassemble the executable program.

Disassemble the executable file to a text file. I use the Assembler Workbench by Talent/Quanta. If you use another disassembler, you may need to make changes to the **ProcessDump_bas** program to accommodate it.

When the SuperBASIC program was compiled, The compiler creates a group of machine routines that the compiled program uses to replace SuperBASIC instructions and operations. Which of these routines that are included in the final compiled program, and the order in which they appear depends on the original SuperBASIC program that was compiled.

This disassembly is used by the ProcessDump program to identify theses routines.

Step 2 - Process the dump file.

Load the **ProcessDump_bas** program, Edit the filenames near the start of the program to suit your system.

dumpFile$	The disassembly made in step 1
exeFile$	The executable file from step 1
doFile$	This will be the generated **_codes** file
dcPath$	Path to Discharge library files used for routine identification

Then **RUN** the program.

This will create two new files a **_lib** file, and a **_codes** file.

The **_lib** file is a copy of the disassembly with the various machine code routines for the SuperBASIC program separated out. It may miss the start of the first routine, which I will come back to later.

It also checks a **Library_id** file to see if it can identify any of the routines for you, and calculates a checksum for each of routines. Creating something like this for an identified routine

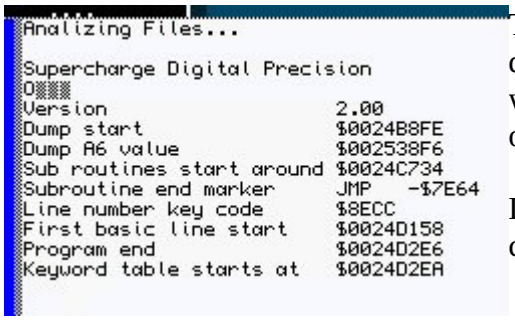
```
Prefix - 967E
Version 5.35 - Checksum = 000A0017D7
Matches Version 5.35 - READ (integer)
Code Index = 162
```

or this for an unidentified routine

```
Prefix - 9F2A
Version 5.37 - Checksum = 002E01471C
```

The version number refers to the compiler, and the checksum is an identifier for the routine found. The first four characters of the checksum is a Hex count of the number of bytes in the routine, and the following six characters is a checksum of the routine.

The **_codes** file is a list of these routines code numbers. Any unidentified routines will be given an index code of -1. This file is used later in the **DisCharge_bas** program to create an array used in the decoding the executable program.



The program will display some information on the disassembled program. A couple of pieces of information will come in useful later. So it may be worth making a note of them.

Dump A6 value, This value is used extensively by the compiled program.

Line number key code, This is the code used to identify the start of a SuperBASIC line.

Note - TurboCharge has a compile option to omit line numbers. If this option was used during compilation, the line number key will either be blank, or FFFF. As there will not be any line number markings in the compiled program.

TurboProcessDump_bas will also scan the executable file for any embedded SuperBASIC extensions, and give you the option of extracting them to a separate file.

Step 3 - Identifying unknown the routines.

The **ProcessDump_bas** program will have hopefully identified most of the routines. However you may have to make some manual identifications.

View the **_codes** file in a text editor, and look for any lines that start with -1. These will need to be resolved, or removed, before trying to do the decompilation.

Any entries in the **_codes** file, starting with -1, that you cannot identify, can be ignored. When you run the DisCharge program it will report and ignore any unidentified codes.

Note - There used to be a problem if two routines in the library files had the same checksum. I have changed the way the checksums have been generated, and hopefully this problem has been fixed.

Identifying routines

There are various ways to identify codes.

It may be that, that routine has been changed in the version of the compiler that compiled the program, compared to the known routines in the Library file. Try picking an instruction in the unknown routine, and search for it in the **Library_lib** files, Looking for a similar routine.

There are two reference library files supplied for SuperDisCharge, **SCLibrary1_lib** and **SCLibrary2_lib**. SCLibrary1 is for SuperCharge versions under 2.00, and SCLibrary2 is for SuperCharge version 2.00
TurboCharge has **SCLibrary5xx_lib** files. These versions refer to the version of the Code Generator in Turbo, not the version of Turbo used to compile the program.

These files contain identified sample routines to compare with your **_dmp_lib** file.

If the routine is short, you may be able to figure out what it does by studying it.

Look for system Trap and Vector calls, which may give a clue to what it does.

Hand decompile (see later) the program line in the **_dmp_lib** file looking for clues, like what parameters it receives, and the expected result (if any)

If you have a copy of SuperCharge or TurboCharge. And you have some ideas of what you think the code may be. Write a short program using your ideas, then compile it, and then Decompile it. And compare the unknown routine with your ideas. (This is the main method I use in development of the decompiler)

When you have identified what the routine does, You can look at the **CodeArrayKeys** document to identify the correct index code. And then amend the index number, or re-insert it into the **_codes** file.

Step 4 - Do the decompilation.

Load the program **DisCharge_bas**, For a Turbo'ed program with line numbers use **TurboDisCharge1_bas**, and for a program without line numbers use **TurboDisCharge2_bas**.

If you pick the wrong turbo program, you should be informed, and the program will halt.

Edit the filenames near the start of the program to suit your system.

filename$	The filename of the executable program
codeArrayData$	The filename of the **_codes** file

Then **RUN** the program.

You only need to **RUN** the program once, after that use **GO TO 1000**.

There is a program line, just past line 1000 - **pauseLine=40000**

This line allows you to pause the decompilation at a BASIC line number. And then continue one line, or step at a time, by pressing any key. Using a number greater than 33000 will disable the pause, as there should not be any line numbers greater than 32767.

Use **GOTO 1000** to start decompiling.

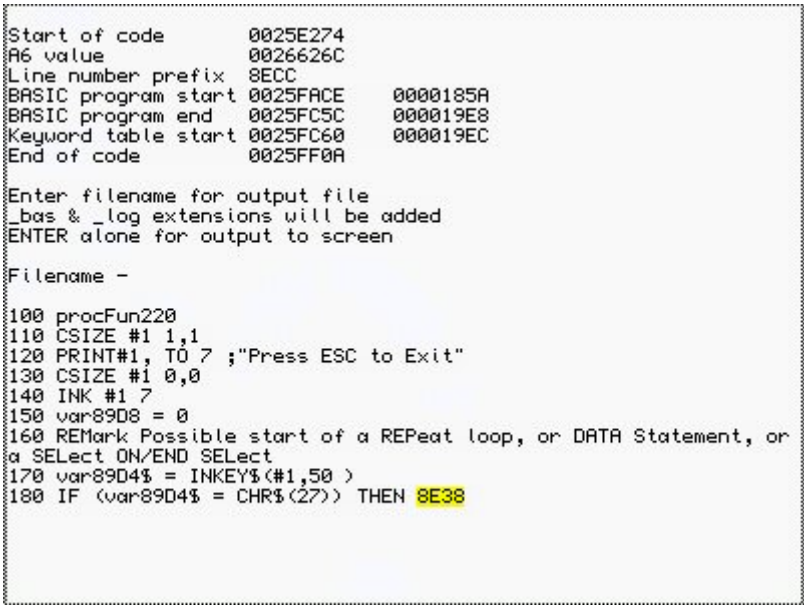
DisCharge may display warnings that it has found problems the _codes file.
You may receive "-1 index found" errors, or "Code Array index already in use" errors.
-1 errors, indicate entries starting with -1, and will be ignored.
Code Array index already in use errors, indicate that it has found duplicated routines in the dump file. The previous one will be ignored, and the current one is used.
These duplications may, or may not need to be sorted out. At the time of writing, there are known duplications of codes 55 and 105, Which need to addressed. However duplications of code 191, does not seem to cause a problem.

It may also display the number of arrays it has found in the program.

When asked for a filename, just press Enter to start a decompilation to the screen.

The decompilation will begin up to the **pauseLine** variable, Press any key to continue one line or instruction at a time.

If the program encounters a Prefix code it does not understand, you will see the code highlighted.



From the example above, To find the start of this routine in the **dmp_lib** file, use the formula ‘A6 value-($10000-code)’ in this case $2538F6-($10000-$8E38) = $24C72E which in this example points to this code which would need to be identified.

0024C72E	3E1D	move.w	(a5)+,d7
0024C730	4BF67000	lea	$00(a6,d7.w),a5
0024C734	4EEE819C	jmp	-$7E64(a6)

You may be able to skip over this error and carry on by pressing any key. But you may be stuck until it is sorted out.

When you get a complete decompilation. You can change the **pauseLine=** to 40000.

Decompile again, this time entering a filename when asked. Two files will be created with your filename, and extensions of **_bas** and **_log**

The **_bas** file will be the BASIC program, and the **_log** file will contain any warning, or errors, and a list of Procedure/Function line numbers.

When decompiling a Turbo program with omitted line numbers, any “At line xxxx Pipe not empty” reports will only appear when a DEFine PROCedure/FuNction is started. And the xxxx will only mean the problem occurred before line xxxx, not on, or around it. This is due to there not being any markers in the program for the start of program lines.

After breaking into the decompiler, **PRINT pcount** will tell you how many items are left on the programs ‘stack’, and **PRINT getTOS$** will show the top item on the stack.

Step 5 - Tidy the SuperBASIC program

The SuperBASIC program produced is unlikely to Load without errors, So load the produced BASIC program into a text editor. And check for obvious problems.

Here are some of the things to look out for.

With TurboDisCharge2_bas, you may see a first line of `100 RETRY_HERE` This may be a ‘false’ program line due to there not being any line number markers in the program for the decompiler to know exactly where to start decompiling from.

If an empty program line is found in the code (where line numbers are not omitted) a ‘:’ (colon) will be displayed, possibly followed by a END IF/SELect.
The ‘:’ line may be the start of a REPeat loop, a SELect ON, or a DATA statement.

Procedures and Functions
DisCharge cannot tell the difference between a DEFine PROCedure, and a DEFine FuNction, and will produce code like.

```
DEFine PROCedure/FuNction procFun680
```

search for calls to procFun680 to establish if it is a Procedure or a Function, and edit the line accordingly.

Super/TurboCharge treats RETurn and a END DEFine the same. Discharge will try to differentiate and fill in the RETurns and END DEFine’s for you. If it has problems, it will just use.

```
RETurn/END DEFine
```

In a Turbo program compiled in Structured mode, DisCharge cannot keep track of the Procedure/Function names for the END DEFines.

In a Turbo program compiled with omitted line numbers, DisCharge may have problems with Procedure and Function parameters confusing them for LOCal variables. The same identification codes are used for parameters and LOCal variables. And as there is no line number markers to act as a seperator, DisCharge can get confused, you may see something like

```
DEFine PROCedure name
LOCal var922Cvar922C
```

Where var922C is a parameter, and not a LOCal variable.

Missing comma’s after channel numbers
You may find that some commands with channel numbers have the comma after the channel number missing.

```
CSIZE #1 1,0
```

This is not a problem with the decompiler itself, but the compiler not recording that a comma is required.

REPeat loops
Super/TurboCharge converts REPeat loops into GO TO's.
The TurboDisCharge program tries to identify the **NEXT**s and **EXIT**s for you and supplies line numbers.

Look out for GO TO's which point back to a line right after a line containing just a colon (:)

Note - In a TurboCharged program with line numbers omitted, you may not get this line.

This GO TO is probably the END REPeat.

Within this loop, If you see a GO TO back to the start of the loop, It is probably an NEXT loop. And a GO TO to just past the END REPeat, is probably a EXIT loop.

You don't always get the REMark and GO TO's that neatly give line numbers. For example this code for an actual decompile, has negative GO TO's. This is caused by there not being an actual line number to EXIT to, as the END REPeat is in the middle of a statement.

```
15040 var8970% = 0
15045 INPUT#9,var8818$ : IF EOF(#9) THEN GO TO -29220 : END IF
15060 PRINT#3,var8818$ : var8970% = ((var8970% + 1) MOD 4) : IF
      (var8970% = 0) THEN procFun3000(" ") : END IF
15070 IF (var8858% = 27) THEN GO TO -29220 : END IF
15075 GO TO 15045 : CLOSE #9  : DELETE var880C$ & var8828$ &
      "_zxzx"  : RETurn/END DEFine
```

After hand decompiling the GO TO's the code becomes.

```
15040  var8970% = 0 : REPeat loop15040
15045  INPUT#9,var8818$ : IF EOF(#9) THEN EXIT loop15040 : END IF
15060  PRINT#3,var8818$ : var8970% = ((var8970% + 1) MOD 4) : IF
       (var8970% = 0) THEN procFun3000(" ") : END IF
15070  IF (var8858% = 27) THEN EXIT loop15040 : END IF
15075  END REPeat loop15040 : CLOSE #9  : DELETE var880C$ &
       var8828$ & "_zxzx"  : END DEFine procFun15000
```

SELect ON
SuperTurboCharge converts SELect's into GO TO's. DisCharge will try to convert them for you. But a **SELect ON y** line is not recorded as such in the Charged program, and the decompiler outputs a program line just containing a ':' (colon).

Note - In a TurboCharged program with line numbers omitted, you may not get this line.

and the **SELect** lines look like

```
[SELect] ON var8914 = 8 : procFun2940
```

If you have a line with just a colon line, just before the SELect lines. Then it's probably a long form SELect. And the colon line would be **SELect ON var8914**, and the select line would be **ON var8914 = 8 : procFun2940**

If there is no colon line. Then it's probably a short form SELect. And the SELect line would be **SELect ON var8914 = 8 : procFun2940**

IF..THEN
If you see something like, **IF 1 THEN**... this is probably **IF COMPILED THEN**...
Likewise, **IF 0 THEN**... probably means **IF NOT COMPILED THEN**...

DATA statements
The data in **DATA** statements are not stored within the encoded version of the BASIC program in SuperTurboCharge. They decompile as a line with just a colon.

Note - In a TurboCharged program with line numbers omitted, you may not get this line.

and at the end of the decompiled program there is a list of the DATA values

```
DATA statements

00274828  DATA  "Alter","Compile","Edit","Invert","Load"
```

Search the listing for line like

```
25904 RESTORE ** Line number to be determined ** data00274828
```

When you determine where to put the DATA lines in the program, you can amend the RESTORE line.

FOR..NEXT..END FOR statements
SuperCharge uses the same code for a **NEXT** and a **END FOR** in **FOR** loops.

So you may need to change a **END FOR** to a **NEXT** for a **FOR..NEXT..END FOR** loop.

If the programmer of the original SuperBASIC program used **FOR..NEXT** as a loop, instead of a **FOR..END FOR** loop. Then SuperCharge will add the missing **END FOR**. I am not exactly sure how SuperCharge decides where to put them. So you may find **END FOR**'s in odd looking places.

You may also encounter a problem where an integer is used as the control variable in a FOR loop. It may not have the % on the end in the **FOR** line, but in the loop, and the **END FOR** it will be there.

Turbo Compiled programs with omitted line numbers
Complied programs without embedded line numbers are somewhat trickier to decompile.
Embedded line numbers make good anchor points to keep track of where you are.

Without line numbers, DisCharge does not know how many instructions appear on on one program line, so it gives each instruction it's own line.

It calculates this line number from its current position in the compiled program.

This can cause **GO TO**'s to not point at line numbers that exist, so you may need check that the next line after the non existent line number looks correct.

It's possible that DisCharge may generate line numbers greater that 32767 which will cause problems for the interpreter.
You may also find that DisCharge has gets confused when it tries to determine the parameters of a Procedure or Function with **LOCal** variables. It may think that the **LOCal** variable is a parameter, as there are no 'line number' breaks in the program.

Threaded and Inline code
Compiled programs are usually generated in Threaded code. However sections of the code can be generated in Inline mode, where instead of having the usual coded version of the original SuperBASIC program stored, you have machine code routines buried inside the coded version of the original SuperBASIC program.

The original SuperBASIC has these lines surrounded by a **REMark +**, and a **REMark -**

At the time of writing this, I have only come across one SuperCharged program that uses Inline code. And at this time I have not thought of a method of automatically decoding this Inline code back into SuperBASIC.

I have added to DisCharge, limited support for Inline code. When you decompile a program using Inline code, it will generate code something like.

```
44 REMark +
46 REMark ** Inline code line **
47 REMark ** Inline code line **
48 REMark ** Inline code line **
49 REMark ** Inline code line **
50 REMark ** Inline code line **
51 REMark -
```

Where **** Inline code line ****** means that the decompiler found a program line of Inline code.

This code is made from joining up the code routines that are normally called separately. However there is nothing separating the code segments, so the decompiler does not know how to identify, and separate the routines.

Hence this code will need to be decompiled by hand. See the Technical Notes document for more details.

As I have not seen Inline code used often, you may need to change the value of the variable **inlineEndMarker** in the procedure **GetVersion** in the **DisCharge_bas** program for versions of SuperCharge below V1.19, and Turbo versions below 5.09

LOADing the generated program
After tidying the generated BASIC into something that looks like it may load and run. On loading into SMSQ/E, you may get syntax errors for things that you have missed. Once you get the program to load without error. When you try to RUN it you get errors like.

At line 277:2 incomplete SELect clause

```
243   ON var8978 = 99 :
247    IF ((var8938 = 0) OR var88F8%) THEN procFun1953 : ELSE
249     ....
275     ...
277     procFun1727 : END IF
279    procFun1259
```

Although SMSQ/E complains about a SELect clause, In this case it's the IF..THEN..ELSE it's upset about.

If you change the code to something like this -

```
243   ON var8978 = 99 :
247    IF ((var8938 = 0) OR var88F8%) THEN
248     procFun1953
249    ELSE
250     ...
275     ...
277     procFun1727
278    END IF
279    procFun1259
```

SMSQ/E will then stop complaining about it.

Once you have the decompiled program running, You can set up the original compiled program running in Qemulator, and the decompiled program running in QPC2 next to it. Then compare the operation of the two programs.

One of the problems that is seen, is where the wrong separators have been used in PRINT statements causing layout problems. It's difficult for DisCharge to keep track of the current print position, and which channel is currently being used.

Hand decompiling

If you are trying to identify what a particular routine does, or you think the decompiler may be decompiling incorrectly. Then you may want to decompile the program manually.

These are a few notes to help you get started decompiling by hand.

The encoded SuperBASIC program in the disassembly listing of the executable file, is located after the end of the routines marked with ‘Prefix’, and before the list of SuperBASIC keywords at the end of the listing.

The encoded BASIC program in Super/TurboCharge makes extensive use of a stack, referenced by the A1 register. Any machine code programmers who have written SuperBASIC extensions will find this familiar. The Super/TurboCharge stack is the equivalent of SuperBASICs maths stack.

The encoded SuperBASIC program is stored as a sequence of Word sized instruction codes, optionally, followed by a (even) number of bytes.

If you examine the CodeArraysKeys document, it will tell you how many additional bytes are required by each instruction.

Most of the instructions will manipulate the information that is on the stack. For example Code Index 20, + Add (float), will take two six byte floating point numbers off the stack, Add them together, Then place the resulting six byte floating point number back onto the stack.

If you know the line number, of the line you want to decompile, Then in the disassembly listing, search for the hexadecimal number of the line. For example to find the SuperBASIC line 100, search for 0064.

Program structures such as REPeat loops, IF..THEN..ELSE, and SELect are converted into GOTO’s

I will now work through a few lines from the sample decompile walk through.

```
110 CSIZE 1,1        This is the original SuperBASIC line
          8ECC 006E
          8F0E
          8F14 0001
          8F14 0001
          8F14 0001
          8F1A 8604 03831303
```

8ECC is code index 0, Start of a program line. 006E is 110 in hexadecimal.

8F0E is code index 96, Precedes actual parameters of a command.

8F14 is code index 55, Integer to place on stack, This is the channel number.

Two further 1’s are then placed onto the stack, which are the two parameters for CSIZE.

8F1A is code index 97, Keyword (procedure), 8604 is a reference to the CSIZE command.
03 is the number of parameters, 831303 are the Type Byte parameters.
The Type Byte parameters, are as in the SuperBASIC Name Table.
In this case it means #integer integer,integer

Note that the comma after the channel number is missing, as I mentioned above in **Tidying the SuperBASIC** program above.

```
160 REPeat loop
0024D1DC  8ECC 00A0
```

Note that REPeat loops start with an empty line. DATA lines, SELect ON, and REMarks are also empty lines

```
180  IF a$=CHR$(27) THEN EXIT loop
0024D1F8  8ECC 00B4
          91A8
          91A8
          951A 89D4
0024D204  8F14 001B
          95A2
0024D20A  95B0
          965A 991E
0024D210  8E38 9966
```

The two 91A8’s are code index 58, Put an integer 0 on the stack.

951A is code index 61, Fetch a string variable onto the stack, the two preceding zeros on the stack, mean the whole string. If there was say, a 1 and a 5 on the stack, then it would have been the string(1 TO 5). The 89D4 is a reference to the string variable (a$).

8F14 is code index 55, Integer to place on stack, 001B being 27.

95A2 is code index 151, CHR$(), Convert the 27 on the stack to a string character.

95B0 is code index 3, = Equals (string), Remove the two strings on the stack, compare them, and place back on the stack, a True, or a False.

965A is code index 140, IF..THEN, If there is a False on the stack, then jump over the next bit of code to the offset 991E, Which is an offset from the A6 register to jump to. The calculation is A6-($10000-offset) In this case A6 is $2538F6 and the calculation is $2538F6-($10000-$991E) = $24D214 which is the start of the next program line 190.

8E38 is code index 160, GOTO, Continue program execution at the offset 9966, Which is worked out as above $2538F6-($10000-$9966) = $24D25C. Which is the start of the program line 220.

```
200  x=x+1
0024D23E  8ECC 00C8
0024D242  9674 89D8
0024D246  97D2 080140000000
          97DC
0024D250  91B6 89D8
```

9674 is code index 60, Fetch a floating point variable onto the stack. The 89D8 is a reference to the variable (x).

97D2 is code index 56, Place a floating point number onto the stack, 080140000000 being 1.

97DC is code index 20, + Add(float), Take the two numbers on the stack, Add them together, and place the result back on the stack.

91B6 is code index 63, Assign a variable (float), Store the result back in the variable referenced by 89D8.

```
210 END REPeat loop
0024D254  8ECC 00D2
0024D258  8E38 98EA
```

8E38 is code index 160, GOTO, Continue program execution at the offset 98EA, Which is worked out as above $2538F6-($10000-$98EA) = $24D1E0. Which is the start of the program line after the REPeat loop, which in the case of the sample program is line 170.

```
220 DEFine PROCedure Setup
0024D25C  8ECC 00DC
0024D260  8E38 99F0
```