

## DisCharge Decompiler

DisCharge is a decompiler for SuperCharged & TurboCharged programs. It is still in development, and features are added, and changed when needed.

DisCharge cannot just convert an executable file back into a ready to run SuperBASIC program. It may require some manual intervention during decompiling, 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 task, back into a SuperBASIC program.

DisCharge comes as five programs. A program to process the executable program, trying to identify the various sub-routines used in the compiling stage to perform the various SuperBASIC commands. And four programs to produce the SuperBASIC code. Two, for SuperCharged programs. And two, for TurboCharged programs. For each of 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.

Note, That while there is a program to deal with SuperCharged programs without line numbers. I don't know how to generate a SuperCharged program without numbers. The only example I have seen is the SuperCharge Parser program. This may have been produced by a special version of SuperCharge created by the author.

SuperCharged and TurboCharged programs share many similarities, but there are enough differences to create complications, by trying to have one program deal with all four combinations.

It's a good idea to copy the executable program to another drive, or sub directory to work on. As the decompiler will generate a number of working files.

There are 3 steps involved in converting a compiled program back to a SuperBASIC program.

The first is to identify the compiler used. Locate and identify the machine code routines used to replace the SuperBASIC commands. And create a **\_codes** file for the next stage.

The second stage is the run the decompiler that produces the SuperBASIC.

The third sage is to tidy the resultant SuperBASIC so that it can be loaded and run.

## Step 1 - Disassemble the executable program, and process the dump file.

Load and run the **DisCharge\_bas** program.

Enter the name of the program to decompile.

```
Enter the filename of the Executable file - dos3_trek_trek_exe
```

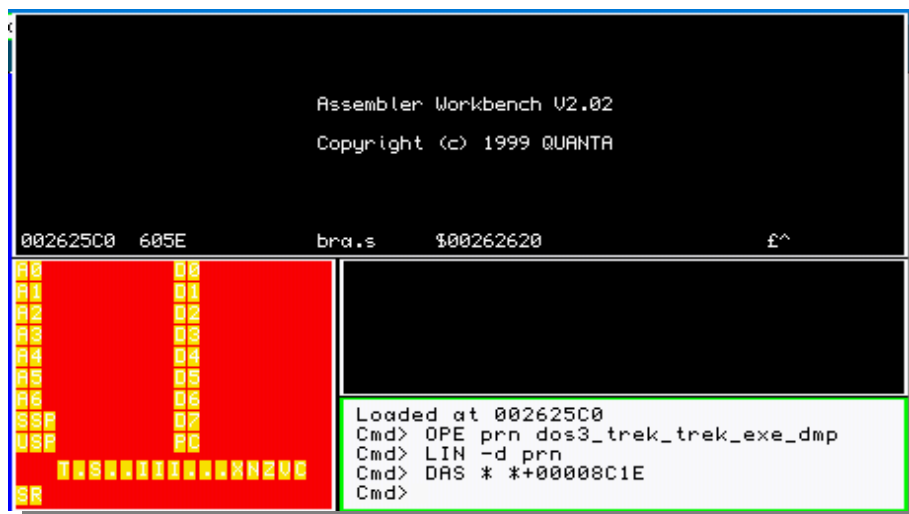
The program will then tell you the names of the files it will create, and where it thinks the library files are for DisCharge.

```
Enter the filename of the Executable file - dos3_trek_trek_exe
Setting dump file to use as      - dos3_trek_trek_exe_dmp
Setting codes file to create as - dos3_trek_trek_exe_codes
File path to Discharge library files - dos3_discharge_
OK?
```

After pressing 'Y'. If the Talent, Assembler Workbench is not in the home directory. You will be asked for it's location.

```
Enter the filename of the Executable file - dos3_trek_trek_exe
Setting dump file to use as      - dos3_trek_trek_exe_dmp
Setting codes file to create as - dos3_trek_trek_exe_codes
File path to Discharge library files - dos3_discharge_
OK?
Disassembler not found in dos3_discharge_
Enter path to disassembler win1_workbench_
```

The disassembler will then be started.



```
Assembler Workbench V2.02
Copyright (c) 1999 QUANTA

002625C0 605E bra.s $00262620 f^

00 01 02 03 04 05 06 07 08
09 0A 0B 0C 0D 0E 0F 10 11
12 13 14 15 16 17 18 19 1A
1B 1C 1D 1E 1F 20 21 22 23
24 25 26 27 28 29 2A 2B 2C
2D 2E 2F 30 31 32 33 34 35
36 37 38 39 3A 3B 3C 3D 3E
3F 40 41 42 43 44 45 46 47
48 49 4A 4B 4C 4D 4E 4F 50
51 52 53 54 55 56 57 58 59
5A 5B 5C 5D 5E 5F 60 61 62
63 64 65 66 67 68 69 6A 6B
6C 6D 6E 6F 70 71 72 73 74
75 76 77 78 79 7A 7B 7C 7D
7E 7F 80 81 82 83 84 85 86
87 88 89 8A 8B 8C 8D 8E 8F
90 91 92 93 94 95 96 97 98
99 9A 9B 9C 9D 9E 9F A0 A1
A2 A3 A4 A5 A6 A7 A8 A9 AA
AB AC AD AE AF B0 B1 B2 B3
B4 B5 B6 B7 B8 B9 BA BB BC
BD BE BF C0 C1 C2 C3 C4 C5
C6 C7 C8 C9 CA CB CC CD CE
CF D0 D1 D2 D3 D4 D5 D6 D7
D8 D9 DA DB DC DD DE DF E0
E1 E2 E3 E4 E5 E6 E7 E8 E9
EA EB EC ED EE EF F0 F1 F2
F3 F4 F5 F6 F7 F8 F9 FA FB
FC FD FE FF

Loaded at 002625C0
Cmd> OPE prn dos3_trek_trek_exe_dmp
Cmd> LIN -d prn
Cmd> DAS * *+00008C1E
Cmd>
```

Press 'F1' to start the disassembly. When the disassembly has finished. Type 'unlink', 'close prn' and 'quit'.

```

0026B1B2 4253          cld.w          (a3)          BS
0026B1B4 006806494E4B ori.w          #$0649,$4E4B(a0)      .h.INK
0026B1BA 4559          ILLEGAL          EY
0026B1BC 2434FFFF      move.l          -$01(a4,a7.l),d2      $4...
0026B1C0 00000000      ori.b          #$00,d0          ....
0026B1C4 00B60000      bclr          d5,$00(a6,d0.w)      ....
0026B1C8 000000B8      ori.b          #$B8,d0          ....
0026B1CC 177400000000 move.b          $00(a4,d0.w), $0000(a3) .t....
0026B1D2 00000000      ori.b          #$00,d0          ....
0026B1D6 00000000      ori.b          #$00,d0          ....
0026B1DA 00000000      ori.b          #$00,d0          ....
0026B1DE 00000001      ori.b          #$01,d0          ....

```

```

A0 00
A1 01
A2 02
A3 03
A4 04
A5 05
A6 06
SSP 07
USP 08
T 09
SR 0A

```

```

Cmd> OPE prn dos3_trek_trek_exe_dmp
Cmd> LIN -d prn
Cmd> DAS * *+00008C1E
Cmd> unlnk
Cmd> close prn

```

The executable will then be processed. Creating a modified version of the disassembly, and a **\_codes** file, which is used in the next phase to identify the purpose of the subroutines used in the compiled version of the original SuperBASIC program.

```
Analizing Files...

Job name   : trek5_exec
Copyright  : 1986 Simon N Goodwin.

Version                5.05
Dump start             $002625C0
Dump A6 value         $0026A5B8
Sub routines start around $002627B8
Subroutine end marker  JMP  $00(a6,d0.u)
Line number key code   $9350
First basic line start $00264AE4
Program end            $0026B06C
Keyword table starts at $0026B070

Searching for imbedded SuperBASIC extensions

No SuperBASIC extensions found

There are 2 potential problems in the codes file

Processing complete. _lib and codes file created

Do you want to start the main decompiler? (y/n)
```

At this point you can press 'N' to stop, or press 'Y' to load the appropriate program for the next phase.

The program will display some information on the disassembled program. Some of it may be useful later. Like the 'Dump A6 value'. This value is used extensively by the compiled program as a reference point for various offsets. And the 'Line number key code'. This is the code used to identify the start of a SuperBASIC line. It is followed by a word containing the line number.

This information will also be saved in a **`dmp_info`** file.

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

For TurboCharged programs, there will be a scan of the executable file for any embedded SuperBASIC extensions, and give you the option of extracting them to a separate file.

In the above screen dump you will see that it mentions that there are 2 potential problems in the codes file. It is not unusual for Turbo'ed programs to have a couple of problems with unidentified codes that don't cause any problems later on. But more than two, may well cause problems later when DisCharge tries to generate the SuperBASIC code. See later in this document for ways to identify unknown and duplicated codes.

The **\_codes** file produced by this step is very important for the decompilation stage. Unidentified codes are likely to stop it in it's tracks.

Once the **\_codes** file has been produced, it can be corrected in a text editor. So you don't need to re-run the first step again.

During the first stage, The decompiler will generate some files. With the same name as the executable, plus an extension.

<b>_codes</b>	This is a reference table of decompiler codes to the code routines.
<b>_dmp</b>	This the disassembly of the executable program.
<b>_dmp_lib</b>	This the disassembly above after code routine identification.
<b>_dmp_info</b>	This is the information displayed on screen during stage 1.

## Step 2 - Do the decompilation.

This stage will be started automatically from the last stage. If the last stage was stopped, or you want to restart decompilation at a later time. You can start this stage manually by loading **SuperDisCharge1\_bas** for a SuperCharged program, **TurboDisCharge1\_bas** for a TurboCharged program with line numbers, or **TurboDisCharge2\_bas** for a program without line numbers.

```
Title 1986 Simon N Goodwin.  
Version 5.05  
Job Name trek5_exec
```

A copyright message, and version number will be displayed. Note that the version number displayed for Turbo programs is not the Turbo version, but the Turbo code generator version, Which is usually different.

```
-1 index found for 8204, ignoring entry  
-1 index found for 8216, ignoring entry  
  
Found 7 Array(s)  
  
Scanning for Procedure and Function calls, code 99  
Found 3 Procedures and Function calls  
  
Scanning for Procedure and Function calls, code 100  
Found 42 Procedures and Function calls  
  
Start of code 00251094  
A6 value 0025908C  
Line number prefix 9350  
Variable Init start 00253534 000024A0  
BASIC program start 002535B8 00002524  
BASIC program end 00259B3E 00008AAA  
Keyword table start 00259C42 00008BAE  
End of code 00259CA2  
  
Enter filename for output file  
_bas & _log extensions will be added  
ENTER alone for output to screen  
  
Filename -
```

Code file problems will be reported, along with the number of Array declarations and Procedure/Function calls.

Code file errors will be “-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 found will be ignored, and the current one used instead.

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.

When asked for a filename, If you just press Enter, then the de compilation will be sent to the screen. This is best thing to do at first to check to see if there are any problems while decompiling.

```

9112 : END SElect
9114 GO TO 9210 : END IF
9120 :
9130 : [SElect] ON var91F2 = 1 : : var9206% = "SIRIUS"
9140 : [SElect] ON var91F2 = 2 : : var9206% = "DENEb"
9150 : [SElect] ON var91F2 = 3 : : var9206% = "CAPELLA"
9160 : [SElect] ON var91F2 = 4 : : var9206% = "BETELGEUSE"
9170 : [SElect] ON var91F2 = 5 : : var9206% = "ALDEBARAN"
9180 : [SElect] ON var91F2 = 6 : : var9206% = "REGULUS"
9190 : [SElect] ON var91F2 = 7 : : var9206% = "ARCTURUS"
9200 : [SElect] ON var91F2 = 8 : : var9206% = "SPICA"
9202 : END SElect
9204 :
9210 IF (var91FE <> 1) THEN
9220 :
9230 : [SElect] ON var91F6 = 1,5 : : var9206% = var9206% & " I"
9240 : [SElect] ON var91F6 = 2,6 : : var9206% = var9206% & " I"
9250 : [SElect] ON var91F6 = 3,7 : : var9206% = var9206% & " I"
9260 : [SElect] ON var91F6 = 4,8 : : var9206% = var9206% & " I"
9270 : END SElect
9280 END IF : :
9290 RETURN/END DEFine : STOP : GO TO ** somewhere in line 9290 **

Reached end of program

```

When the program finishes decompilation, you can restart it by typing **GO TO 1000**

You are libel to see some rubbish at the end of the decompile, as the decompiler may over-run the end of the program before it notices that it's got there.

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

Then enter a filename to save it to. A **\_bas** extension will be added automatically, and a **\_log** file will also be created.

If you have some problem and want to pause the decompilation. 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. However programs compiled without line numbers, can end up with line numbers above 32767 and would need to be edited in the next stage.

The decompilation will begin, and run up to the **pauseLine** variable, Press any key to continue one line or instruction at a time. Depending on whether line numbers are omitted or not.

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

```
Start of code      0025E274
A6 value          0026626C
Line number prefix 8ECC
BASIC program start 0025FACE      0000185A
BASIC program end   0025FC5C      000019E8
Keyword table start 0025FC60      000019EC
End of code        0025FF0A

Enter filename for output file
_bas & _log extensions will be added
ENTER alone for output to screen

Filename -

100 procFun220
110 CSIZE #1 1,1
120 PRINT#1, T0 7 ; "Press ESC to Exit"
130 CSIZE #1 0,0
140 INK #1 7
150 var89D8 = 0
160 REMark Possible start of a REPEAT loop, or DATA Statement, or
a SElect ON/END SElect
170 var89D4$ = INKEY$(#1,50 )
180 IF (var89D4$ = CHR$(27)) THEN 8E38
```

If you see something like this, then the decompiler has encountered a code it does not know how to deal with. In this case 8E38. At this point you may be able to continue by pressing any key.

However you may find that the program will not. But you can always **BREAK** out of the program, and restart with **GO TO 1000**

You may also see warning messages, and other reports going to #0 during the decompilation, most of which will also appear in the **\_log** file.

See later in this document for help in identifying unknown codes.

### Step 3 - 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 `REMark`, 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 `RETurn`'s 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 are no line number markers to act as a separator, DisCharge can get confused, you may see something like

```
DEFine PROCedure name  
LOCAL var922Cvar922C
```

Where `var922C` is a parameter, and not a `LOCAL` variable.

If you look at the procedure call, you will know how many parameters there should be.

Note, DisCharge may get confused with `GO SUB` routines. Treating them as Procedures. The compiler treats `GO SUB` as a Procedure without parameters.



### 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$ &
      "_zxxz" : 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$ & "_zxxz" : END DEFINE procFun15000
```

## **SElect ON**

Super/TurboCharge converts SElect's into tests and 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 on it, 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 is not stored within the encoded version of the SuperBASIC program in Super/TurboCharge. They are stored all together in one block outside of the SuperBASIC area.

Within the SuperBASIC area, there are empty lines where they should be. When decompiled they appear as a line with just a colon.

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

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 when 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**

Compiled programs without embedded line numbers are somewhat trickier to decompile. The 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 one program line, so it gives each instruction it's own line. And 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 to be correct.

It's possible that DisCharge may generate line numbers greater than 32767 which will cause problems for the interpreter if you try to **LOAD** it.

You can get around this problem by splitting the program into two parts in a text editor and renumbering the second parts line numbers to be below 32000. Then **LOAD** each part, **RENUM**ber, and **MERGE** back together.

You may also find that DisCharge 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** programs for versions of SuperCharge below V1.19, and Turbo versions below 5.09

#### **\ Print separator**

The \ (backslash) print separator may not appear in the decompiled program as the same code is used at the end of every PRINT line. If I make it appear you get backslash characters at the end of every PRINT line. So a line like **100 PRINT "hello"** would decompile as **100 PRINT#1,"hello"**

#### **\*\* stack empty\*\***

If you see this, then the decompiler had difficulty earlier identifying a procedure/function. And assumed it to be Procedure, where it was in fact a Function.

It later expected something to be on the stack (the Function), And it was not there. So it replaces the missing stack entry with **\*\* stack empty \*\***.

Somewhere before the **\*\* stack empty \*\*** you should find what looks like a Procedure call, but if you look at the definition, you should find it's a Function.

If you see something like

```
5100 procFun31204
5110 procFun31256
5120 var8F90=(** stack empty ** +PEEK_W(**stack empty **))
```

Then first procFun should be placed as the last stack empty, and the second procFun as the first stack empty. To give this

```
5120 var8F90=(procFun31256 +PEEK_W(procFun31204))
```

## Procedures and Functions

QDOS or SMSQ/E may get upset about brackets with parameters in Procedure and Function calls.

### Brackets in Procedure calls

<code>procFun1234 (1,10)</code>	bad
<code>procFun1234 1,10</code>	good

### Empty parameters in Function calls

<code>x = procFun1236()</code>	bad
<code>x = procFun1236</code>	good

Functions that return a string may need a '\$' appended onto the end of the Function calls and the Function definition.

## LOADing the generated program

After you have tidied the generated BASIC into something that looks like it may load and run. You can try to LOAD it. It is best to try loading in SMSQ/E, as that will report the line numbers with syntax errors in #0. Once you get the program to load without error. When you try to RUN it you may 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 likely to be 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.

## What can possibly go wrong?

Well, there are a few things that can go wrong and stop decompiling. In the order that they may occur.

1- You try to decompile a program that was compiled with a version of SuperCharge/Turbo that the decompiler does not know about.

In this case you will need to add the new version in the **GetVersion** procedures in the Discharge programs to recognize the new version.

With a SuperCharged program, you will probably find that a lot of the code routines will match.

In the case of Turbo'ed programs, you will have to create both a **TCLibrary5xx\_lib** and a **TCLibrary5xx\_id** file. Then manually identify all the code routines found in the *executable\_dmp\_lib* file. (more on that later) Adding them to the *TCLibrary5xx\_lib* and *TCLibrary5xx\_id* files as you go. Expect there to be about a hundred routines to identify.

2 - At the end of stage 1, You find a few potential problems in the *executable\_codes* file.

You could ignore them for now and carry on. If it's unimportant code routine as far as the decompiler is concerned, you may get away with it. Otherwise you will hit problem 3 below.

The *executable\_codes* file is a list of the code routine numbers, preceded by an identification key for the second stage program. With -1 indicating that the routine is unidentified.

For example, say there is an entry in the *\_codes* file, -1,9F2A

If you look through the *executable\_dmp\_lib* file, you should find something like this

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

The version number refers to the compiler version, 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 bytes of the routine.

## Identifying routines

There are various ways to identify code routines.

Compare the code routine with routines in the other *library\_lib* files looking for matches. A lot of the code routines between different versions of the compilers remain basically unchanged. With just some JMP destinations different.

Use the generated checksums first 4 characters of routine length, to search for routines of the same length. And visually compare them. Note that some routines are almost exactly the same and just

differ in one instruction. For example, one may have a BEQ instruction, and another with a BNE instruction instead.

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

### 3 - Stage 2 decompiling stops with something like

```
Start of code      0025E274
A6 value          0026626C
Line number prefix 8ECC
BASIC program start 0025FACE      0000185A
BASIC program end   0025FC5C      000019E8
Keyword table start 0025FC60      000019EC
End of code        0025FF0A

Enter filename for output file
_bas & _log extensions will be added
ENTER alone for output to screen

Filename -

100 procFun220
110 CSIZE #1 1,1
120 PRINT#1, TO 7 ; "Press ESC to Exit"
130 CSIZE #1 0,0
140 INK #1 7
150 var89D8 = 0
160 REMark Possible start of a REPEAT loop, or DATA Statement, or
a SELECT ON/END SELECT
170 var89D4$ = INKEY$(#1,50)
180 IF (var89D4$ = CHR$(27)) THEN 8E38
```

This is telling you that the decompiler has encountered a code 8E38 that it does not know how to handle. The **\_codes** file will probably have a line -1,8E38 in it. You need to go back to problem 2, above and identify the code routine.

Once the routine has been identified, you can just edit the **\_codes** file, and do **GO TO 1000**

The point at which the decompiler stops in the SuperBASIC generation may give you a clue to the function of the problem code. Or you may need to hand decompile around the problem area to try to determine the function by 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** documents to identify the correct index code. And then amend the **\_codes** file.



#### 4 - "At line xxxx Pipe not empty" warnings

The decompiler uses a pipe to represent the stack used in the compiled program. This warning is telling you that the decompiler found some items on this stack, when it reached the end of a program line.

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 the decompiler not knowing when the original program line ended.

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.

## 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 SuperBASIC 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 documents, 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.

The first 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

```