

DisCharge Decompiler Technical Notes

This document contains notes for my use on the main Super/TurboCharge decompiler programs, **SuperDisCharge1/2_bas** & **TurboDisCharge1/2_bas**. But it may be of help for anyone trying to understand how the decompiler programs work.

Notes on the **DisCharge_bas** program can be found at the back of this document.

First a few notes on the internal make up of a compiled executable.

The compiled program can be split into four main parts.

The first is the initialization of the compiled program, error handling, and some common utility routines.

The second part is a group of routines that are called from the encoded version of the original SuperBASIC program. These routines perform many of the SuperBASIC commands and functions. Which of these routines that are included, and the order that they appear in, is dependent upon the original SuperBASIC program when it was compiled. The **Discharge_bas** program will attempt to identify these routines, and assign them to codes, in the **_codes** file, that are used by the main decompilers.

The third part is the encoded version of the original SuperBASIC program.

This part is also subdivided into a program/variables initialization area, and the program itself. In Turbo compiled programs the initialization area may appear both before, and after the program area.

The fourth part is a list of SuperBASIC commands and functions that are not handled by the compiled program, but are called from the operating system. These are commands like WINDOW, PAPER, BEEP, SIN(), etc.

This list may also contain externally loaded commands and functions like DEVICE_STATUS.

SuperCharge has a default set of these command whether used or not, however Turbo only includes commands that are actually used by the compiled program.

Register usage in Turbo (taken from Turbo source code)

Reg	Description	Name

A1	Maths stack pointer	maths
A4	Heap pointer	data
A5	Pseudo program pointer	thread
A6	Task base pointer	base
D0	QDOS Request code register	request
D5	Turbo error number	error
D6	(low word) BASIC line number	lineno

Data areas in compiled programs

Offsets from the start of the compiled program

CodeGen V5.00

\$00	W	Branch always to the start of program code.
\$02	L	CodeGen version - 5.00
\$06	W	\$4AFB marker word
\$08	W	string length of job name
\$0A		Bytes of job name
\$18		Copyright notice - 1986 Simon N Goodwin
.		
.		
.		
\$40	L	WHEN_ERROR 1 address
\$44	L	WHEN_ERROR 0 address
.		
.		
.		
\$5C		Start of program code

CodeGen V5.37

\$00	W	Branch always to another branch to the start of program code (offset \$48)
\$02	L	CodeGen version - 5.37
\$06	W	\$4AFB marker word
\$08	W	string length of job name
\$0A		Bytes of job name
\$18		Copyright notice - 2000 The Turbo Team
.		
.		
.		
\$48	W	Branch to the start of program code.
\$4A	L	WHEN_ERROR 1 address
\$4E	L	WHEN_ERROR 0 address
.		
.		
.		
\$88		Start of program code

Checksums in Library files

In the library files, there is a checksum to uniquely identify the code routines from the second part of the compiled program. The checksum is comprised of 40 bits. Split into two parts.

A 16 bit byte count of the routine. And a 24 bit checksum of the bytes of the routine. And is shown as a 10 character hexadecimal number.

The checksum is not a simple sum of the bytes in the routine, but a sum of the sum of the bytes in the routine. The double checksum is to get around the problem I encountered on more than one occasion, where two different routines happened to produce the same simple checksum.

example -

00100033C7

This routine contains 16 bytes (\$0010), and the 16 bytes make a checksum of (\$0033C7)

A6 offset calculations

Many addresses used in the compiled program are offsets to the A6 register. At the start of the compiled program, The A6 register is set to a value about 32K away from the start of the program.

This value is then used as a base address for jumps and other things in the compiled program.

To determine the end addresses, or find the offset for a particular address, use the following methods.

Offset to address

If offset is less than \$8000 then
 $\text{address} = \text{A6} + \text{offset}$

If offset is greater than, or equal to \$8000 then
 $\text{address} = \text{A6} - (\$10000 - \text{offset})$

Address to offset

If address is greater than A6 then
 $\text{offset} = \text{address} - \text{A6}$

If address is less than A6 then
 $\text{offset} = \$10000 - (\text{A6} - \text{address})$

Library code sub routine terminations

Depending on setting of the 'code size control' in the Turbo compiler, The way the code subroutines end differs. Which determines which Library _id files to use

-\$xxxx(a6)	when <64K is selected.	Use TCLibrary xxx_id	Type 1
\$00(a6,d0.w)	when >64K is selected.	Use TCLibrary xxx_5_id	Type 2

Procedures and Functions

While Procedures and Functions with the same names appear in all the decompilers. They may differ in their coding to allow for differences in the decompilers.

PROCedure SaveMe

Saves a copy of the program

FuNction GetString\$(ptr)

Returns a string of a standard QDOS string (word length then bytes of string) stored in memory at ptr

FuNction GetInt(ptr)

Returns a signed integer stored in memory at ptr

FuNction GetWord(address)

Returns an unsigned integer stored in memory at address

FuNction FLT(string\$)

Converts a 12 character (six byte) hex string into a floating point number

FuNction FLT\$(n)

Convert a Floating point number into a 12 character (six byte) hex string
by PJW & Steven Pool V0.02 03/12/17

FuNction NameListLen(start)

Returns the number of the highest name list entry in the keywords section at the end of the compiled program
start is the address in memory of the keyword table

PROCedure EmptyPipe (channel)

Makes sure the pipe is empty. If it is not empty, it is reported to channel, the pipe is emptied displaying the contents of the pipe.

FuNction GetTOS\$

Remove and return the item at the rear end of the pipe.

PROCedure FlagLine (LNum,rStart,rEnd)

If it does not already exist, add the supplied line number and addresses to the procFun array.

PROCedure CheckProcFun(num)**FuNction CheckProcFun(num)**

Checks to see if the line number, num is in the Proc/Fun list. If it is, then create a DEFine PROCedure/FuNction line.

Also checks for unresolved SElect's and IF's.

This routine is a procedure when line numbers have been omitted.

Returns when line numbers are present

0 If the line number is not in the Proc/Fun list.

1 If its a multiline DEFine PROCedure/FuNction.

2 If its a single line DEFine PROCedure/FuNction.

PROCedure AssignArray(varNum,noElements)

Adds an array entry into the arrayElements array

FuNction GetElements(varNum)

Returns the number of elements for an array. varNum is the SuperCharge reference for the array.

PROCedure ListProcs (ch)

List the Procedure/Function line numbers table to the supplied channel number.

PROCedure ReadCodeArray (file\$)

Read in the code array from file\$. The Format is index number,hex word index value
E.g. 23,90FA - codeArray(23)=\$90FA

PROCedure DoDataLines

Create DATA statements. DATAarray(0) holds the number of RESTORE commands.
Other indexes are addresses of data starts.

PROCedure CheckELSE (ptr,type)

Check to see if an ELSE is needed. Ptr is the current program position.

Type, is the size of the GO TO used for the ELSE. 0 = word sized. 1 = long sized.

FuNction CheckENDIF

Check for pending END IF

PROCedure StripIFStack

Strips the top item off the IFStack and ELSEStack string

PROCedure LoadFile (file\$)

Load the file (file\$) and set base addresses initialBase, base, and filelength.

PROCedure CheckValues

Check values in file, and set various pointers.

PROCedure GetVersion

Get program version information and set marker variables.

PROCedure FindArrayElements

(sc)

Scan variable initialization area looking for arrays, and filling in arrayElements.

PROCedure StartNewLine (linenumber)

Start a new listing line starting with the supplied line number.

PROCedure InitNewLine

Initialize line decompose variables.

PROCedure InitMain

Set up variables for decompiling.

PROCedure StatementSeperator

If line numbers are suppressed, start a new line. Otherwise output a colon (:) separator.

PROCedure CheckSElects (nl)

Checks for any outstanding SElect processing to deal with. If nl is 1, then we are at the start of a new line

PROCedure CheckLastSElect

Checks to see if this is the last selection. If so output SElect ON instruction. Otherwise add to current SElection and continue.

PROCedure ScanProcFun (keycode,progstart,progEnd)

Scans the program for Procedure and Function calls, and adds them to a list.

FuNction CountParameters (ptr)

Try to count the number of parameters in a Proc/Fun definition.

PROCedure ExternalGlobal (ptr)

Deal with EXTERNAL/GLOBAL constructs. Ptr, points at the start of the definitions.

FuNction FindProcFunType(progAddress)

Try to identify a procedure/function.

Returns 1=Proc, 2=Numeric Fun.

PROCedure SetProcFunParm (LNum,num)

Set the procFun array's number of parameters.

PROCedure SetProcFunType (LNum,num)

Set the procFun array's type number.

PROCedure SetProcFunRet (LNum,num)

Set the procFun array's number of RETurn's.

FuNction GetProcFunType (LNum)

Return the procFun array's type number.

FuNction GetProcFunParm (LNum)

Return the procFun array's parameter number.

FuNction GetProcFunRet (LNum)

Return the procFun array's returns number.

FuNction GetProcFunStart (LNum)

Return the procFun array's parameter number.

FuNction GetProcFunEnd (LNum)

Return the procFun array's returns number.

PROCedure SetProcFunEnds

For a structured program set all the end addresses in the procFun array.

PROCedure AmmendProcFun

Display and optionally amend the Procedure and Function definitions.

FuNction IndentDef

Adds an indent inside Proc/Fun definitions.

Returns 1 during a Proc/Fun definition, otherwise 0.

PROCedure ShowCompileOptions

Display the believed compile time options.

PROCedure CheckLineNumbers

Check the line number integrity of the just decompiled program.

And reports to the log file.

Global variables

(sc) SuperDisCharge If one of these two are shown, it means that this variable is
(tc) TurboDisCharge specific to one of the decompilers.

filelength		Length of executable file.
initalBase		Loaded base address, may be different to 'base' if Qemulator header found.
base		Start of the executable.
finish		End of the executable.
progOffset		Offset from start of executable to start of BASIC program area.
prog		Running pointer for the BASIC program.
basicProgStart		Start of BASIC program area.
varInitOffset		Offset from start of executable to start of variable init area.
varInitStart		Start of variable init area.
keywords		Start of keyword table.
progEnd		End of BASIC program area.
TCver\$		Library file version from executable.
TCtitle\$		TurboCharge copyright message.
jobName\$		Executables job name.
DATAendMarker		Word that marks the end of any DATA in the initialization area.
DATAexist		Flag to indicate if any DATA statements need to processed 0 if no READ instructions encountered 1 if READ instructions encountered
lineStart		Word that marks 'move.w (a5)+,d6', the start of a BASIC program line.
progStartAdj	(tc)	An adjustment for the start of the BASIC program in compiler versions that have a RETRY_HERE inserted before the start of the BASIC program
A6		Address in loaded executable that represents the A6 register.
keywordOffset		Address that is a base for accessing Keywords. Not a pointer to the keyword list itself.
lineNoExist		1 if compiled program has embedded line numbers 0 if compiled program does not have line numbers.
freeform	(tc)	1 if the FREEFORM option was used during compiling. 0 if the STRUCTURED option was used during compiling. The decompiler will assume FREEFORM until it encounters a STRUCTURED style Procedure of Function definition.
lineStartPrefix		Prefix code for BASIC program line start, or -1 if no embedded line numbers.
lineCount		Number of program lines decompiled.
lineNo		Line number of line currently de compiling.
pseudoLine	(tc)	Address of a reference point when line numbers are suppressed
ch		Channel to send decompiled output to.
er (sc) errCh	(tc)	Channel to send Errors & log to.

pcount		Number of items in the pipe.
InIF		Number of levels of nested IF's.
IFStack\$		List of 8 character HEX destination addresses of END IF's or ELSE's.
ELSEStack\$		
FORselect\$	(tc)	Used to hold the selection for FOR statement e.g. "1 TO 5,8,10 TO 12"
FORstep\$	(tc)	Used to hold the STEP value
FORvar\$	(tc)	Used to hold the FOR variable reference
newLine		0 = Not at the start of a new line.
InPrint		0 = Not in a PRINT/INPUT.
SELselect\$	(tc)	Used to hold the selection for SElect statement e.g. "1 TO 5,8,10 TO 12"
InProcFun		0 = Not in a Procedure or Function
InSel		Number of levels of nested SElect's
Indef		0 = Not in any DEFine's. Once in a DEFinition, it points to just past the end of the definition block. Unless the STRUCTED option was used during compiling, Then Indef is set to -1.
channelOpenCheck		0 = No pending channel operations.
currentChannel\$		Current channel number, or channel variable/expression.
key		Holds the current operation prefix code.
maxArrayEle		Number of array elements in arrayElements().
maxData		Number of array elements in DATAarray().
maxSel		Number of array elements in selStack().
maxProcFun		Number of array elements in procFun().
maxCodes		Number of array elements in codeArray().
disChargeVer\$		Version number of the decompiler.
dataSpace		Compiled programs set data space.
structProcStart		Structured program start. If not zero, line number of 1st Proc/Fun in a structured program.
retCount		Number of RETurn's/END DEF's in the proc/fun being decompiled.
outputFiles\$		The filename of the decompile file.
sbExts		Address of the header of any installed SuperBASIC extensions.

Arrays

nameList()	Array of offsets for keywords at end of code.
arrayElements(maxArrayEle,1)	Holds the number of elements in arrays, entries of 2
DATAarray(maxData)	DATA statement RESTORE points
codeArray(maxCodes)	The code array
selStack(maxSel,2)	Holds information on nested SELECT's
	Second index -
	0 Pointer to possible END SELECT
	1 Pointer to next test/REMAINDER/END SELECT
	2 Select variable
procFun(maxProcFun,2)	Holds information on GO SUBs, Procedures and Functions
	Second index -
	0 Proc/Fun line number
	1 Type
	0 undefined
	1 procedure
	2 numeric function
	3 string function
	4 gosub
	2 Number of expected parameters
	-1 indicates not yet defined
	3 Number of returns in proc/fun, including the
	END DEFINE
	0 = undefined
	4 Start address
	5 End address

Turbo Toolkit commands handled in the Keywords section

ALLOCATION, DEALLOCATE, SEARCH_MEMORY, PEEK_F, BASIC_F, FREE_MEMORY, BASIC_B%, BASIC_W%, BASIC_L, BASIC_POINTER, BASIC_NAME\$, BASIC_TYPE%, BASIC_INDEX%, TYPE_IN, SET_PRIORITY, COMMAND_LINE, END_CMD, CONNECT, CHANNEL_ID, SET_CHANNEL, EXECUTE, EXECUTE_W, EXECUTE_A, DEFAULT_DEVICE, LIST_TASKS, REMOVE_TASK, SUSPEND_TASK, RELEASE_TASK, DATASPACE, CURSOR_ON, CURSOR_OFF, EDIT\$, EDITF, EDIT%, SET_FONT, POSITION, SET_POSITION, INTEGER\$, FLOAT\$, STRING\$, STRING%, STRINGF, GET%, GETF, GET\$, INPUT\$, DEVICE_STATUS, DEVICE_SPACE

Turbo toolkit commands handled by the compiler internally

PEEK\$, POKE\$, COMPILED, OPTION_CMD\$, DATA_AREA (makes an empty line), RETRY_HERE, MOVE_MEMORY, SNOOZE, LEN(variable), ERLIN%, ERNUM%

Known SuperCharge versions

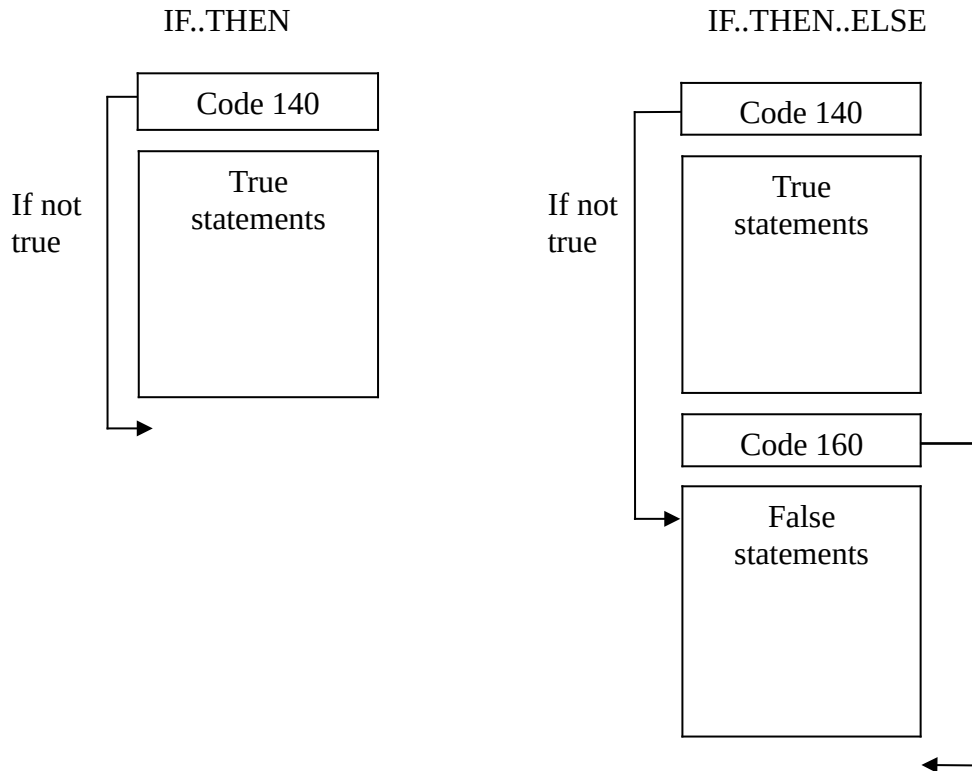
Version	Copyright message	Known by DisCharge
0.14	1985 Simon N Goodwin	yes
1.09		no
1.11	1985 Simon N Goodwin	yes
1.15	1985 Simon N Goodwin	yes
1.16	1985 Simon N Goodwin	yes
1.17	1985 Simon N Goodwin	yes
1.18		yes
1.19	1985 Simon N Goodwin	yes
2.00	SuperCharge Digital Precision	yes

Known TurboCharge versions

Date	Version	Parser?	Codegen	Copyright message	Known
			/5.00	1986 Simon N Goodwin	yes
			/5.05	1986 Simon N Goodwin	yes
			/5.09	1987 The Turbo Team	yes
			/5.10	1987 The T Team (like 5.09)	yes
1987 DP	3.24	2.00/2.04	/5.10	1987 The Turbo Team	yes
2000 GG	4.4	4.4	3.03/5.15	2000 The Turbo Team	yes
2002 GG	4.15	4.15	3.09/5.20	2000 The Turbo Team	yes
2006 GG	5.02	3.24	3.23/5.35	2000 The Turbo Team	yes
2007 GG	5.06	3.44	3.26/5.36	2000 The Turbo Team	yes
2007 GG	5.07	3.44	3.26/5.36	2000 The Turbo Team	yes
2007 GG	5.08	3.44	3.27/5.37	2000 The Turbo Team	yes
2007 GG	5.09	3.44	3.27/5.37	2000 The Turbo Team	yes

Format of program storage in the compiled program

Where there are differences between SuperCharge and TurboCharge, and the different versions. It will be made, as so - (TC 5.10) meaning the format for Turbo with a code generator of Version 5.10 is different.



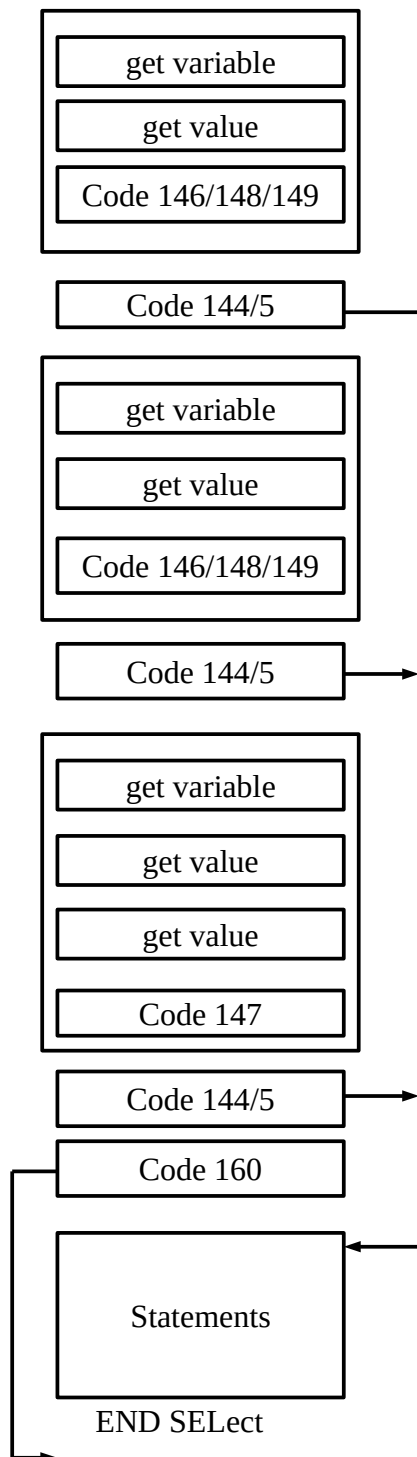
Note - In Turbo, the code 160 (word sized GO TO) may be a code 240 (long sized GO TO) in large programs.

REPEAT loops in SuperCharge V2.00

Code 135 (NEXT/ END FOR) is also used in REPEAT loops for NEXT

Usual SElects

Select ON x 1,7,14,2 TO 10

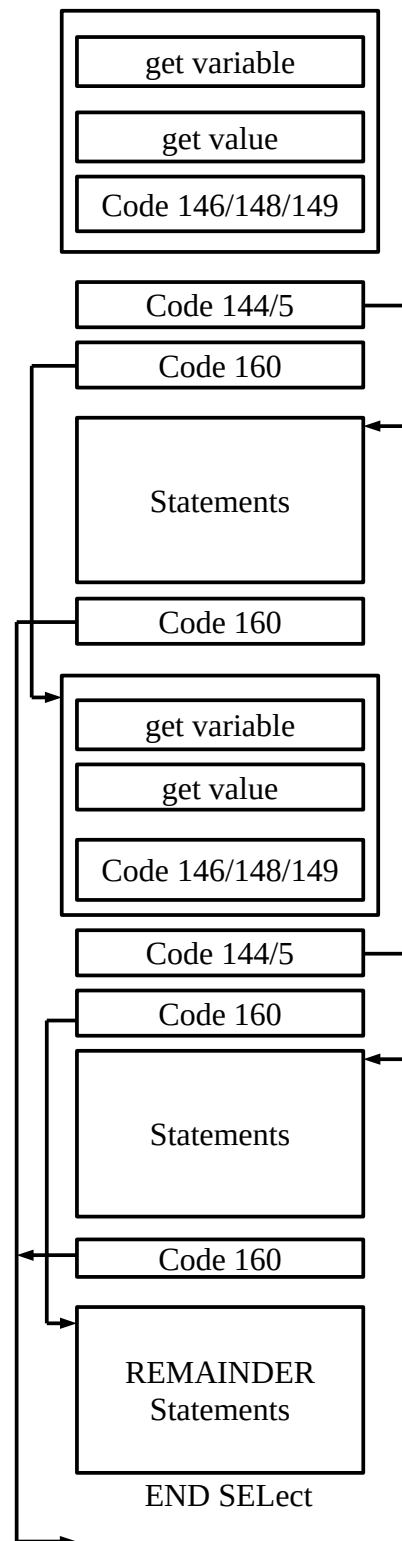


SElect ON x

ON x=..

ON x=..

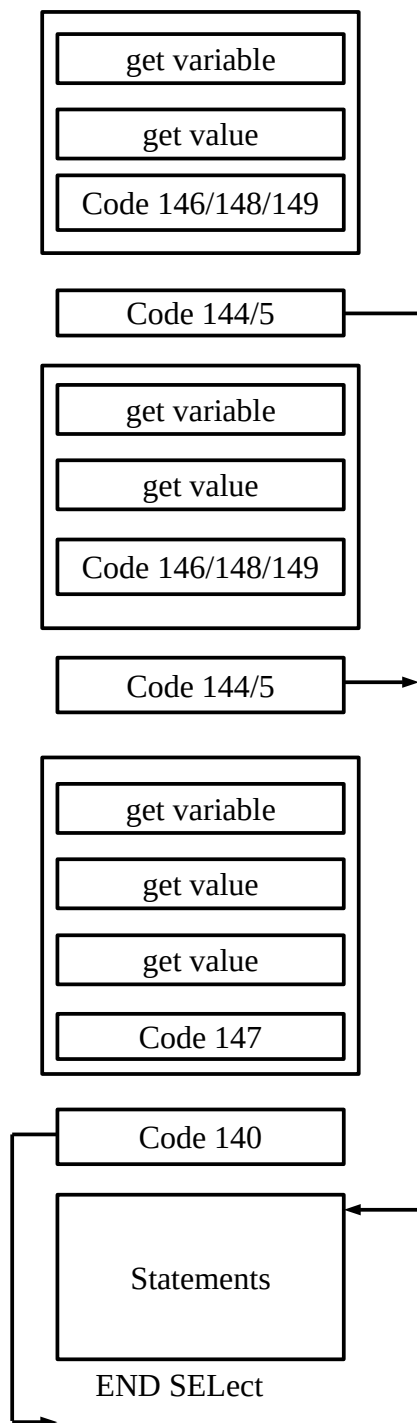
ON x=REMAINDER..



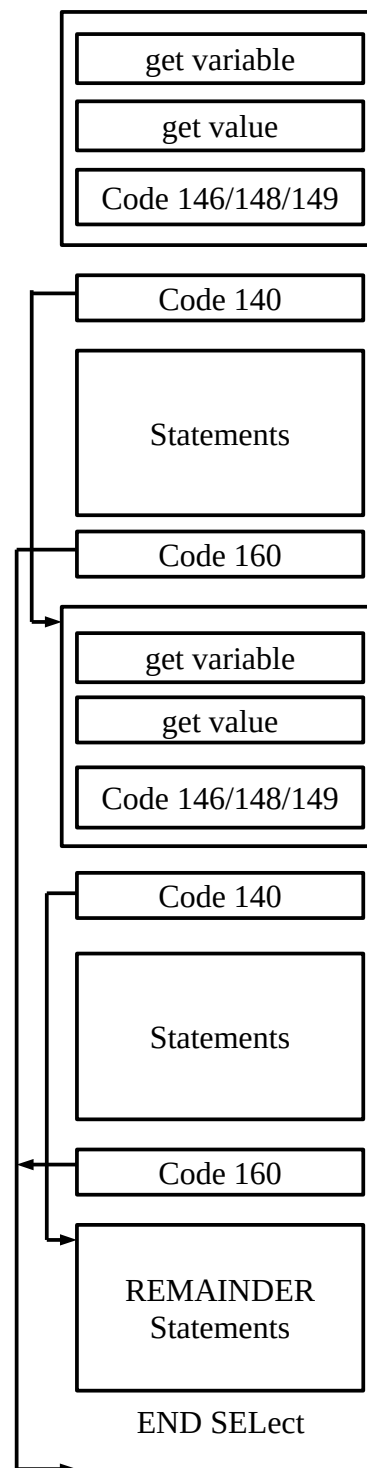
Note - In Turbo, the code 160 (word sized GO TO) may be a code 240 (long sized GO TO) in large programs. The 145 codes may be 144 in TurboCharge

SElect with Turbo versions 5.00 to 5.10

Select ON x 1,7,14,2 TO 10



SElect ON x
ON x=..
ON x=..
ON x=REMAINDER..



Note - In Turbo, the code 160 (word sized GO TO) may be a code 240 (long sized GO TO) in large programs. The 145 codes may be 144 in TurboCharge. Code 140 is equal to IF..THEN

Procedure and Function definitions

SuperCharge

Start	2 words	Go To (160),	A6 offset to end of proc/fun definition
-------	---------	--------------	---

Followed by the parameters, if any. Note that the parameters are in reverse order

Integer	4 Words	Code 101,	variable reference,
		Code 62,	variable reference
Float	4 words	Code 102,	variable reference,
		Code 63,	variable reference
String	6 words	Code 85	
		Code 103,	variable reference
		Code 190?	
		Code 64	variable reference

Turbo is as above except for

Start	2 words	Go To (160),	A6 offset, or may be -
Start	3 words	Go To (240),	A6 offset, in long programs. End of proc/fun definition
String	2 Words	Code 103 or 107	variable reference
Float Array	2 words	Code 125	variable reference

If the Structured option was used in compilation, then there is no GOTO (code 160/240) and offset at the start of the Procedure/Function definition. Except for the first definition, which has a STOP (code 161).

When a procedure or function is called by a code 99/100, Then the entry point is just after the 'start' Go To (160/250)

Local variables

SuperCharge

Integer	2 words	Code 101,	LOCal variable reference
Float	2 words	Code 102,	LOCal variable reference
String	4 words	Code 55, Code 103,	256 (max string length?) LOCal variable reference
Integer array	2 words	Code 260,	LOCal array reference
Float array	2 words	Code 261,	LOCal array reference
String array	4 words	Code 262, Code 263,	LOCal 256 (max string length?) LOCal array reference

Turbo

Integer	2 words	Code 101 or 126,	LOCal variable reference
Float	2 words	Code 102,	LOCal variable reference
String	4 words	Code 55, Code 103 or 107	string length LOCal variable reference
Integer array 1 dim	5 words	Code 55, Code 260,	Dimension size \$0000, LOCal variable reference
Float array 1 dim	5 words	Code 55, Code 261,	Dimension size \$0000, LOCal variable reference
Integer array x dim		x values placed on stack Code 260,	no of index's -1, LOCal variable reference
Float array x dim		x values placed on stack Code 261,	no of index's -1, LOCal variable reference
String array x dim		x values placed on stack Code 263,	no of index's -2, LOCal variable reference

Calling Procedures and Functions

After the parameters (if any) have been placed on the stack.

SuperCharge 2 Words Code 100, A6 offset

Turbo 3 Words Code 99, Long A6 offset (don't know what's different to 100)

Turbo 3 Words Code 100, Long A6 offset

Note the offset is an odd number and needs 1 subtracting.

When a procedure or function is called by a code 99/100, Then the entry point is just after the 'start' Go To (160/250)

In-bedded DATA statements

The contents of DATA lines are not in-bedded in the codec BASIC program where they belong, But inserted in the initialization area before the main part of the BASIC program.

The end of the DATA being marked with a \$8300 for SuperCharge, and \$8001 for TurboCharge.

Decoding the DATA is as follows -

SuperCharge

If the first word is less than \$7FFF, Then it's a string in the normal QDOS fashion of a length word followed by the characters of the string.

If the first word is exactly \$7FFF, Then it's an integer in the next word.

If the first word is \$8000 and above, Then its a float.

Do a 2's complement (invert all bits and add 1) to the word.

This is the first two bytes of the float. And the next two words are the rest of the float.

TurboCharge

If the first word is less than \$7FFF, Then it's a string in the normal QDOS fashion of a length word followed by the characters of the string

If the first word is \$8000 and above, Then its a number.

Do a 2's complement (invert all bits and add 1) to the word

If the result is less than, or equal to \$FFF, Then it's a float. In which case, subtract \$1BB8 from this result. And this is the first two bytes of the float. And the next two words are the rest of the float.

If the result is greater than \$FFF, Then it's an integer.

If the first word is exactly \$7FFF, Then it's part of a T_CONFIG. In which case, read the next word, which is a normal integer.

GLOBAL and EXTERNAL variables

Global variable and procedures are defined in the variables initialization as normal.

External variables do not appear in the variable initialization (but function parameters do).

EXTERNAL/GLOBAL definition block in a compiled program

Start	2 words	Go To(160),	A6 offset to just past end of definition block
	1 word	code 191	Marks start of individual entries
For each Global or External definition	1 word	The definition number. Followed by either a word of zero, or a byte giving the length of a definition name, followed by the bytes of the name padded out to a word boundary.	
Then for each entry in the definition	2 words	1st word is a type code, 2nd word is a variable/proc reference	
	repeats....		
	1 word	A zero marks the end of the definition	
At the end of all the definitions	4 words	\$FFFF, \$0000, \$4EAE, \$8218 This marks the end of the definitions	

Decoding the type word

The type word defines the type of the entry in the definition

Bits 0 to 4	Entry type	1 = string 2 = float 4 = integer 8 = Procedure 16 = Function
Bit 5	Set for an array	
Bit 6	Set for a parameter to a proc/fun	
Bits 9 to 11	Number of array dimensions	
Example -	\$0422 (%0000 0100 0010 0010)	This is a float array with 2 dimensions y(0,0)
	\$0014(%0000 0000 0001 0100)	This is an integer function
	\$0261(%0000 0010 0110 0001)	and this is a string parameter for the function note, it is defined as a single dimension array which is a normal string in a compiled 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 +
45 REMark ** Inline code line **
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.

Format of the Inline code.

I only have the one program that uses Inline code, and this is how it's laid out in this compiled program, which has line numbers.

The **REMark +** line start with a normal code 0 word, followed by a line number word, then an A6 offset word pointing to the start of the inline machine code.

Each program line of the 'inline' code starts with the word \$3C3C, followed by a line number word, then the machine code of the program line.

This continues until the **REMark -** line is reached, which starts with the word \$3C3C, followed by a line number word, then a long word 'jump to subroutine relative to A6' starting \$4EAExxxx. The second word may vary between different versions of the original compiler.

You then revert back to the normal compiled code 0 word, followed by a line number word.

The code routines that are joined together may not be exactly as as they appear in the library routines, but close enough to be able to identify them.

Variable references and data that were in the coded version of the SuperBASIC program are handled slightly differently.

Variable references are handled by setting a pointer to the variable, then moving the value either onto, or off of, the stack pointed to by A1.

```
246E88BC    movea.l  -$7744(a6),a2    point at 'var88BC'
```

Note that the SuperBASIC variable reference is the second word of the 'movea.l' instruction. In this case the '88BC'

```
3392E800    move.w   (a2),$00(a1,a6.l)    get value onto the stack
```

```
34B1E800    move.w   $00(a1,a6.l),(a2)    set variable from the stack
```

Note that the format of the instruction may change between different versions of the compiler used, but it always involves moving between the A1 stack, and A2.

Note- I have seen a Turbo program (QPuzzle_exe). Where the whole program seems to be in 'in line' code, without line numbers.

There are no \$3C3C codes to mark the start if line numbers.

The \$4EAExxxx instructions are undefined.

\$246E references variables

\$3392/\$34B1 Don't seem to exist.

DisCharge Decompiler Technical Notes

This document contains technical information on the 'DisCharge_bas' program.

This program performs the first phase of the decompiling process. It disassembles and examines the source compiled program, identifies the embedded code routines, and produces a 'codes' file that is passed to the next phase of the decompiling process.

Embedded code routines and checksums

The compiled program contains a group of machine code subroutines that perform various SuperBASIC operations and features. However this group of machine code subroutines, and the order they appear in, is different for every SuperCharged, or Turbo'ed program.

A checksum is produced for each of the found embedded machine code routines, and is compared with the appropriate library '_id' file for the version of SuperCharge or Turbo compiler originally used.

This then supplies a 'code' for the function of that machine code routine in the compiled program, to be supplied the main decompiler program.

The checksum is comprised of 40 bits. Split into two parts.

A 16 bit byte count of the routine. And a 24 bit checksum of the bytes of the routine. And is shown as a 10 character hexadecimal number.

The checksum is not a simple sum of the bytes in the routine, but a sum of the sum of the bytes in the routine. The double checksum is to get around the problem I encountered on more than one occasion, where two different routines happened to produce the same simple checksum.

example -

00100033C7

This routine contains 16 bytes (\$0010), and the 16 bytes make a checksum of (\$0033C7)

Procedures and Functions

PROCedure SaveMe

Saves a copy of the program.

PROCedure GotRoutine

Deals with an end of subroutine marker.

FuNction GotLineStart (strg\$)

Returns the address of the A6 referenced supplied parameter

Found a 'move.w (a5)+,d6' routine for the start of line.

PROCedure OpenFiles

Opens the dumpfile to #3, Loads the executable program and sets the variables
initalBase, base and fileLen.

FuNction DoChecksum\$ (ptr)

Returns the 24bit checksum for the code routine at 'ptr'.

FuNction CheckLibrary(c\$)

Scans the Library ID file looking for the supplied checksum. If found updates the '_lib' file, and
returns the code array index. Otherwise returns -1.

FuNction CheckLibrary2(c\$)

Scans the Library ID file looking for the supplied checksum. If found returns the code array index
Otherwise returns -1. Special version for first routine.

PROCedure BasicExtensions

Scan the executable looking for SuperBASIC extensions.

PROCedure GetFileNames

Get the filenames required and sets some global variables.

PROCedure SetVersionVariables

Set the version variables.

PROCedure CheckCodeFile(file\$)

Check the codes file for -1 entries. Meaning unidentified code routines.

PROCedure DoDisassembly

If the '_dmp' file has not been created. Make one by starting the disassembler.

FuNction CodeTerminator

Scan the executable file looking for 'jmp \$00(a6,d0.w)' instructions

To try to determine the code routines end markers. If there are more than 2 then this is the likely
terminator.

Returns 0 for 'jmp -\$7DD8(a6)' type terminators, or 0.5 for 'jmp \$00(a6,d0.w)' type terminators

These are different for programs compiled with a code control size of <64K, or >64K

Global variables

exeVer\$	String containing the version number of the code generator used in the compile process
version	exeVer\$ multiplied by 100
compileType	0 = SuperCharge, 1 = Turbo
jobName\$	Executable jobs, job name
title\$	Executable jobs copywrite message
base	Start of the program loaded in memory
exeA6	A6 value in executable file
dumpA6	A6 value in the dump
subStart\$	Start of sub routines
jmp\$	Subroutine end marker
lineno\$	Line number key code