JC64dis

by
Ice Team

JC64Dis
Next Generation Disassembler

1 Introduction

JC64dis is an iterative program designed to disassemble 8-bit machine code[1]. It specifically targets the **MOS M6502**, **Zilog Z80** and Intel 8048 processors.

Notably, it directly supports the following machines:

- Commodore 64 (C64)
- Commodore 128[2] (C128)
- Commodore VIC 20
- Commodore Plus4
- Commodore 1541 disk drive
- Atari
- Odyssey/Videopack

[1]At the actual 2.9 version, the program has over 101.000 lines of code and has an extimated COCOMO cost of $1.369.000 with 25 person/year of realization effort.
[2]Both M6502 and Z80 variants.

The program can read the following file formats:

- **PRG**: Standard programs containing machine code and the starting address for execution.
- **SID**: PSID/RSID image format for C64 music programs.
- **MUS**: Musical data related to the SIDPLAYER.
- **MPR**: Proprietary format for embedding multiple PRG files to be fully disassembled.
- **CRT**: Format containing cartridge dumps for C64 expansion cartridges (specific page analysis is necessary due to memory blocks).
- **VSF**: Snapshot of the VICE emulator containing C64 memory and state.
- **AY**: AY music program image format for ZX Spectrum, Amstrad CPC, and Atari ST, containing the Texas Instruments AY chip.
- **NSF**: Music program image format for the NES system.
- **SAP**: Music program image format for Atari using the POKEY chip.

Currently, *JC64dis* can directly generate code for the following assemblers[3]:

- **M6502**:
  - Dasm
  - TMPx
  - CA65
  - Acme
  - Kick Assembler
  - 64 Tass
  - Macro Assembler (AS)
- **Z80**:
  - Glass
- **I8048**:
  - Macro Assembler (AS)

Given its extensive configurability, *JC64dis* provides over 330 different configuration options that allow fine-tuning of its behavior. For a detailed understanding of each parameter's meaning, refer to the program's built-in help.

In this manual, we will emphasize its underlying logic and functionality.

[3]In this manual almost all code example provided are shown in Dasm format.

2 History

JC64dis was born in 2003 as a simple command-line utility associated with the *JC64* emulator written in Java by the *Ice Team*.

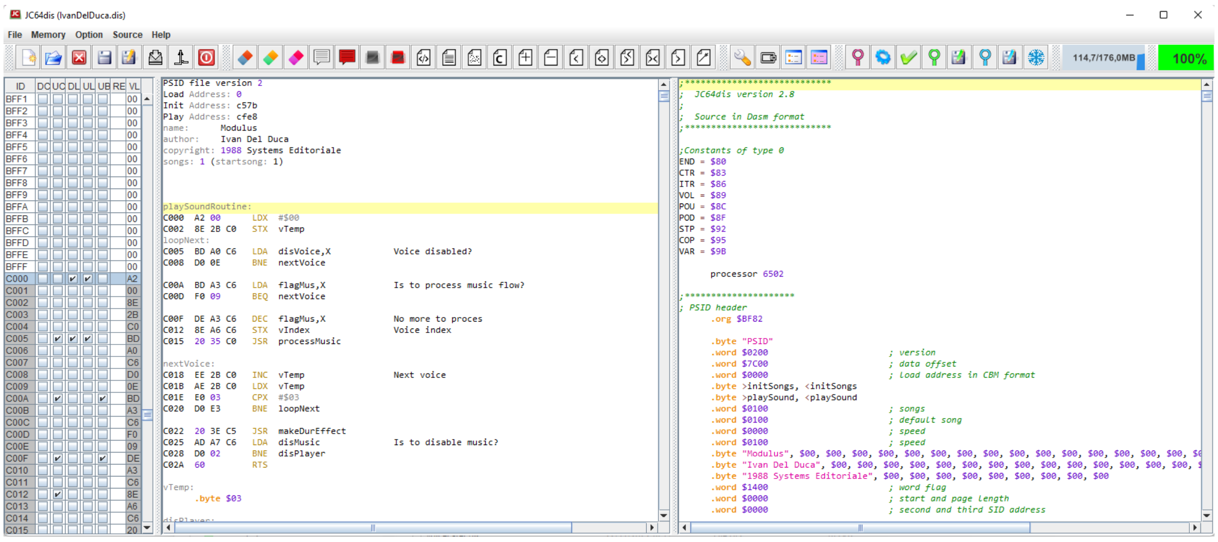
Initially, its purpose was to disassemble the C64's Kernal in order to verify the correctness of the emulator itself, as the first code executed at startup resides in the Kernal.

From the outset, this tool allowed for automatic commenting of memory locations for C64 instructions, making it easier to understand the machine code. Thanks to *JC64dis*, several music players were disassembled and later featured in the *SIDin magazine*, produced by the *Ice Team*.

The experience gained during this phase led to a complete overhaul of the program, with the goal of reducing the time required to analyze and fully understand machine code by a factor of 10. In 2020, version 0.7 of the program was released, marking the first version with a graphical interface.

The program distinguishes itself in the following three work areas:

- 64K Memory Cells (with corresponding states) Preview of Disassembled Code, presented in a mixed format of hexadecimal data and assembler instructions.
- Source Code of the Disassembled Program, available in one of the formats supported by assemblers.
- The program's logic revolves around having a "dumb" disassembler to which we can provide step-by-step information about each memory cell. This ultimately results in a thoroughly commented source code that can be reassembled to obtain the original machine code.



ID	DC	UC	DL	UL	UB	RE	VL
BFF1	☐	☐	☐	☐	☐		00
BFF2	☐	☐	☐	☐	☐		00
BFF3	☐	☐	☐	☐	☐		00
BFF4	☐	☐	☐	☐	☐		00
BFF5	☐	☐	☐	☐	☐		00
BFF6	☐	☐	☐	☐	☐		00
BFF7	☐	☐	☐	☐	☐		00
BFF8	☐	☐	☐	☐	☐		00
BFF9	☐	☐	☐	☐	☐		00
BFFA	☐	☐	☐	☐	☐		00
BFFB	☐	☐	☐	☐	☐		00
BFFC	☐	☐	☐	☐	☐		00
BFFD	☐	☐	☐	☐	☐		00
BFFE	☐	☐	☐	☐	☐		00
BFFF	☐	☐	☐	☐	☐		00
C000	☐	☐	✓	✓	☐		A2
C001	☐	☐	☐	☐	☐		00
C002	☐	☐	☐	☐	☐		8E
C003	☐	☐	☐	☐	☐		2B
C004	☐	☐	☐	☐	☐		C0
C005	☐	✓	✓	✓	☐		BD
C006	☐	☐	☐	☐	☐		A0
C007	☐	☐	☐	☐	☐		C6
C008	☐	☐	☐	☐	☐		D0
C009	☐	☐	☐	☐	☐		0E
C00A	☐	✓	☐	☐	✓		BD
C00B	☐	☐	☐	☐	☐		A3
C00C	☐	☐	☐	☐	☐		C6
C00D	☐	☐	☐	☐	☐		F0
C00E	☐	☐	☐	☐	☐		09
C00F	☐	✓	☐	☐	✓		DE
C010	☐	☐	☐	☐	☐		A3
C011	☐	☐	☐	☐	☐		C6
C012	☐	✓	☐	☐	☐		8E
C013	☐	☐	☐	☐	☐		A6
C014	☐	☐	☐	☐	☐		C6
C015	☐	☐	☐	☐	☐		20

```
PSID file version 2
Load Address: 0
Init Address: c57b
Play Address: cfe8
name:       Modulus
author:     Ivan Del Duca
copyright: 1988 Systems Editoriale
songs: 1 (startsong: 1)



playSoundRoutine:
C000  A2 00     LDX  #$00
C002  8E 2B C0  STX  vTemp
loopNext:
C005  BD A0 C6  LDA  disVoice,X          Voice disabled?
C008  D0 0E     BNE  nextVoice

C00A  BD A3 C6  LDA  flagMus,X           Is to process music flow?
C00D  F0 09     BEQ  nextVoice

C00F  DE A3 C6  DEC  flagMus,X           No more to proces
C012  8E A6 C6  STX  vIndex              Voice index
C015  20 35 C0  JSR  processMusic

nextVoice:
C018  EE 2B C0  INC  vTemp               Next voice
C01B  AE 2B C0  LDX  vTemp
C01E  E0 03     CPX  #$03
C020  D0 E3     BNE  loopNext

C022  20 3E C5  JSR  makeDurEffect
C025  AD A7 C6  LDA  disMusic            Is to disable music?
C028  D0 02     BNE  disPlayer
C02A  60        RTS

vTemp:
      .byte $03
```

```
;****************************
;  JC64dis version 2.8
;
;  Source in Dasm format
;****************************

;Constants of type 0
END = $80
CTR = $83
ITR = $86
VOL = $89
POU = $8C
POD = $8F
STP = $92
COP = $95
VAR = $9B

      processor 6502

;*********************
; PSID header
      .org $BF82

      .byte "PSID"
      .word $0200                       ; version
      .word $7C00                       ; data offset
      .word $0000                       ; Load address in CBM format
      .byte >initSongs, <initSongs
      .byte >playSound, <playSound
      .word $0100                       ; songs
      .word $0100                       ; default song
      .word $0000                       ; speed
      .word $0100                       ; speed
      .byte "Modulus", $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
      .byte "Ivan Del Duca", $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
      .byte "1988 Systems Editoriale", $00, $00, $00, $00, $00, $00, $00, $00, $00
      .word $1400                       ; word flag
      .word $0000                       ; start and page length
      .word $0000                       ; second and third SID address
```

JC64dis main screen

3 Memory

Each individual memory cell visible in the left table of the program internally contains several fields, some of which are later synthesized as symbols or flags in the columns of the same table.

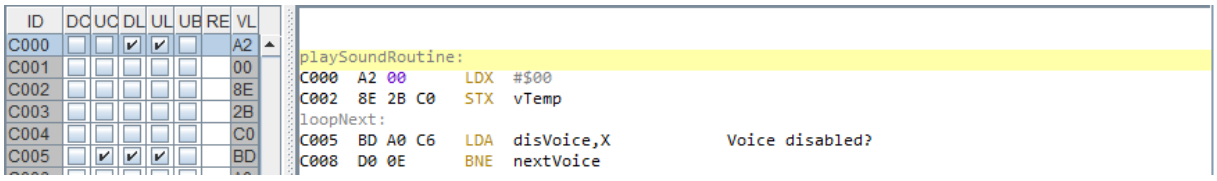
The fields present (the meaning of many of which will be discussed in detail below) are as follows:

- **Address**: The address from 0 to 65535 associated with the memory cell.
- **Value**: The value that the memory cell holds, derived from the loaded file or operations (such as relocation) performed on other cells.
- **Automatic Comment**: Automatically generated comment for the instruction whose last operand is this cell.
- **Manual Comment**: A comment manually written by the user for this cell (this comment takes priority over the automatic one, which is hidden in the presence of manual comments).
- **Automatic Comment Block**: A multi-line comment automatically generated by the program and associated with this memory cell.
- **Manual Comment Block**: A multi-line comment manually inserted by the user using the program (in this case, the manual block takes priority over the automatic one).
- **Automatic Label**: A label automatically associated by the program with this memory cell.
- **Manual Label**: A label manually created by the user of the program (this label takes priority over any automatically generated one).
- **Internal State**: Indicates whether the cell is part of the memory area associated with the program for disassembly.
- **Code State**: Indicates whether the cell should be evaluated as machine code.
- **Data State**: Indicates whether the cell should be evaluated as data area.
- **Trash State**: Indicates whether the cell contains data that should not be disassembled by the program.
- **Constant Index**: Indicates which of the 20 constant value tables to use for providing a non-numeric representation for this cell.
- **Related Value**: Indicates which memory cell this one is directly linked to.
- **Type of Relationship**: Specifies the type of relationship this cell has with the related one.
- **Base Relocation Address**: Represents the starting base for relocating this memory cell.
- **Destination Relocation Address**: Indicates the destination for relocating this memory cell.
- **Data Type**: Specifies how to represent the data contained in this memory cell.
- **BASIC Type**: For data-type cells, further specifies the type of BASIC to which this value belongs.

4 Data Types

The choice of data type for a memory cell is a fundamental consideration when using the program. By default, every memory area associated with loaded data is displayed as “code” type. Therefore, the primary task is to mark areas as “data” (or, more rarely, as “garbage”)[4].

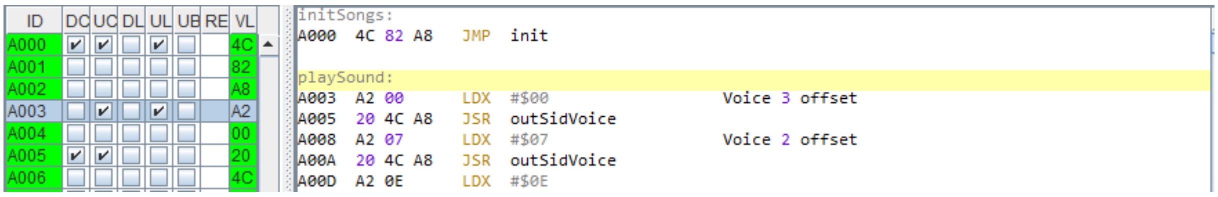
Visually, an unassociated memory area (and thus interpreted as “code”) is marked in gray:



ID	DC	UC	DL	UL	UB	RE	VL
C000	☐	☐	✓	✓	☐		A2
C001	☐	☐	☐	☐	☐		00
C002	☐	☐	☐	☐	☐		8E
C003	☐	☐	☐	☐	☐		2B
C004	☐	☐	☐	☐	☐		C0
C005	☐	✓	✓	✓	☐		BD

```
playSoundRoutine:
C000  A2 00       LDX  #$00
C002  8E 2B C0    STX  vTemp
loopNext:
C005  BD A0 C6    LDA  disVoice,X              Voice disabled?
C008  D0 0E       BNE  nextVoice
```

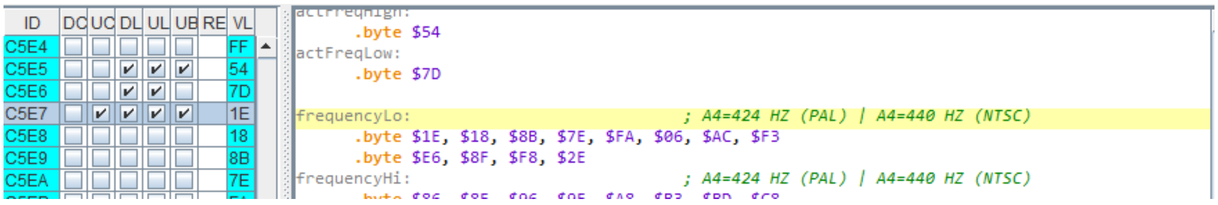
An area associated as “code” is instead marked in green:



ID	DC	UC	DL	UL	UB	RE	VL
A000	✓	✓	☐	✓	☐		4C
A001	☐	☐	☐	☐	☐		82
A002	☐	☐	☐	☐	☐		A8
A003	☐	✓	☐	✓	☐		A2
A004	☐	☐	☐	☐	☐		00
A005	✓	✓	☐	☐	☐		20
A006	☐	☐	☐	☐	☐		4C

```
initSongs:
A000  4C 82 A8    JMP  init

playSound:
A003  A2 00       LDX  #$00                   Voice 3 offset
A005  20 4C A8    JSR  outSidVoice
A008  A2 07       LDX  #$07                   Voice 2 offset
A00A  20 4C A8    JSR  outSidVoice
A00D  A2 0E       LDX  #$0E
```

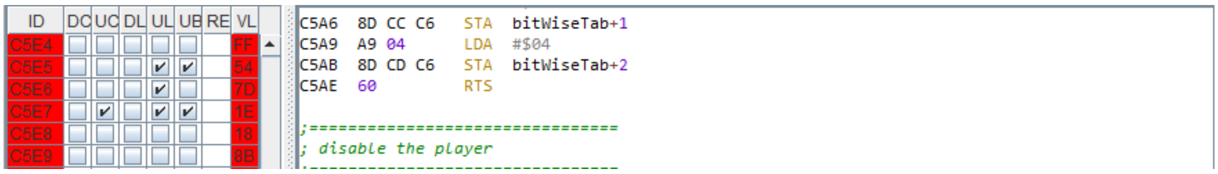
An area associated as “data” is marked in cyan:



ID	DC	UC	DL	UL	UB	RE	VL
C5E4	☐	☐	☐	☐	☐		FF
C5E5	☐	☐	✓	✓	✓		54
C5E6	☐	☐	✓	✓	☐		7D
C5E7	☐	✓	✓	✓	✓		1E
C5E8	☐	☐	☐	☐	☐		18
C5E9	☐	☐	☐	☐	☐		8B
C5EA	☐	☐	☐	☐	☐		7E

```
      .byte $54
actFreqLow:
      .byte $7D

frequencyLo:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
      .byte $1E, $18, $8B, $7E, $FA, $06, $AC, $F3
      .byte $E6, $8F, $F8, $2E
frequencyHi:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
```

Finally[5], an area associated as “garbage” is marked in red:



ID	DC	UC	DL	UL	UB	RE	VL
C5E4	☐	☐	☐	☐	☐		FF
C5E5	☐	☐	☐	✓	✓		54
C5E6	☐	☐	☐	✓	☐		7D
C5E7	☐	✓	☐	✓	✓		1E
C5E8	☐	☐	☐	☐	☐		18
C5E9	☐	☐	☐	☐	☐		8B

```
C5A6  8D CC C6    STA  bitWiseTab+1
C5A9  A9 04       LDA  #$04
C5AB  8D CD C6    STA  bitWiseTab+2
C5AE  60          RTS

;================================
; disable the player
```

The choice made affects what is displayed both in the preview and in the area designated for the data source:

- **Code Area**: Displays the disassembled bytes that are marked.
- **Data Area**: Shows the data in one of the chosen formats, with the default set to “byte” type.
- **Garbage Area**: The entire data area is ignored and not used for creating the source[6].

[4]However, labeling an area as “code” can be useful to highlight parts that we have analyzed and deemed correct within this category.
[5]The memory cells outside the program bytes are highlighted in white
[6]It’s worth noting that labels created in this area are still utilized by the program.

5 Data Areas

Data areas allow further characterization into various subtypes, making the source code more readable and manageable. Since not all assemblers provide support for certain data types, *JC64dis* addresses this issue by automatically creating appropriate macros to compensate for the lack, thus standardizing their treatment.

The following formats are indeed anticipated:

- **Hexadecimal Byte**: Undoubtedly the most commonly used format, which represents a sequence of bytes encoded in hexadecimal using the prefix $.

```
.byte $E0, $35, $02, $44, $E0, $29, $02, $44
.byte $E0, $45, $02, $44, $E0, $41, $02, $44
.byte $E0, $35, $02, $44, $E2, $3C, $02, $44
.byte $E2, $41, $02, $44, $E2, $45, $02, $44
```

- **Decimal Byte**: Represents a sequence of bytes in numerical decimal format.

```
.byte 224, 53, 2, 68, 224, 41, 2, 68
.byte 224, 69, 2, 68, 224, 65, 2, 68
.byte 224, 53, 2, 68, 226, 60, 2, 68
.byte 226, 65, 2, 68, 226, 69, 2, 68
```

- **Binary Byte**: Represents a sequence of bytes in binary format, using the prefix %.

```
.byte %11100000, %00110101, %00000010, %01000100, %11100000, %00101001, %00000010, %0100
.byte %11100000, %01000101, %00000010, %01000100, %11100000, %01000001, %00000010, %0100
.byte %11100000, %00110101, %00000010, %01000100, %11100010, %00111100, %00000010, %0100
.byte %11100010, %01000001, %00000010, %01000100, %11100010, %01000101, %00000010, %0100
```

- **Character Byte**: Represents a sequence of bytes in character format[7], enclosed in single quotes.

```
.byte $E0, '5, '⊡, 'D, $E0, '), '⊡, 'D
.byte $E0, 'E, '⊡, 'D, $E0, 'A, '⊡, 'D
.byte $E0, '5, '⊡, 'D, $E2, '<, '⊡, 'D
.byte $E2, 'A, '⊡, 'D, $E2, 'E, '⊡, 'D
```

- **Word**: A sequence of two bytes in little-endian[8] format that is represented as a 16-bit hexadecimal value.

```
.word $43E0, $4402, $40E0, $4402
.word $3CE0, $4402, $43E0, $4402
.word $40E0, $4402, $3CE0, $4402
.word $43E0, $4402, $40E0, $4402
```

- **Inverted Word**: A sequence of two bytes in big-endian[9] format that is represented as a 16-bit hexadecimal value.

```
dc.s $E043, $0244, $E040, $0244
dc.s $E03C, $0244, $E043, $0244
dc.s $E040, $0244, $E03C, $0244
dc.s $E043, $0244, $E040, $0244
```

[7]Depending on the assembler, any characters not convertible to ASCII are forced to be represented as hexadecimal bytes.
[8]In little-endian format, the first byte is the least significant, while the second byte is the most significant.
[9]In big-endian format, the first byte is the most significant, while the second byte is the least significant.

- **Tri-byte**: A sequence of 3 bytes in little-endian format that is displayed as a 24-bit. hexadecimal value.

```
Tribyte3 $334141, $414102, $334141
Tribyte3 $414108, $414141, $410D41
Tribyte3 $414141, $124111, $174141
Tribyte3 $414141, $414141, $414141
```

- **Long**: A sequence of 4 bytes in little-endian format that is shown as a 32-bit hexadecimal value.

```
.long $33414141, $41023341
.long $41414108, $41414141
.long $0D414141, $41124111
.long $17414141, $41414141
```

- **Address**: An address[10] (in little-endian format) is a word for which we provide the compiler with the knowledge that we will use it to store memory locations (pointers). In any case, we can create addresses as either two bytes or a word. There is no difference in program execution, but the use of the address attribute is merely a representation convenience employed by certain compilers.

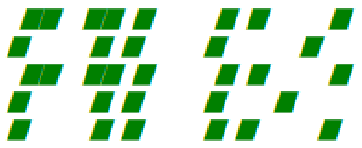
```
.word $4133, $4141, $0241, $4133
.word $4141, $0841, $4141, $4141
.word $410D, $4141, $1241, $1141
.word $4117, $4141, $4141, $4141
```

- **Stack Address**: It is a word decreased by 1. It is used to place the address that must be executed after a subroutine return instruction (RTS).

```
Stack4 $4134, $4142, $0242, $4134
Stack4 $4142, $0842, $4142, $4142
Stack4 $410E, $4142, $1242, $1142
Stack4 $4118, $4142, $4142, $4142
```

- **Monochromatic Sprite Definition**: A sequence of 64 bytes, logically grouped into blocks of 3x21 bytes[11], in which bit 0 represents transparency, while bit 1 corresponds to the sprite's color.

```
MonoSpriteLine %001100110100000101000001   ;
MonoSpriteLine %010000010100000100000010   ;
MonoSpriteLine %001100110100000101000001   ;
MonoSpriteLine %010000010100000100001000   ;
MonoSpriteLine %010000010100000101000001   ;
```



- **Multicolored Sprite Definition**: A sequence of 64 bytes, grouped into blocks of 3x21 bytes. In this case, the horizontal bits are taken in pairs to associate:
  - 00 with transparency
  - 01 with common color 1
  - 10 with common color 2
  - 11 with the sprite's color

[10]The method of inserting addresses into JC64dis will be discussed further.
[11]The last byte is ignored but serves to achieve a multiple-of-2 size.

```
MultiSpriteLine %0011001101000000101000001   ;
MultiSpriteLine %0100000101000000100000010   ;
MultiSpriteLine %0011001101000000101000001   ;
MultiSpriteLine %0100000101000000100001000   ;
```

- **Text**: A sequence of consecutive bytes treated as text, enclosed in quotation marks where characters can be represented as ASCII.

```
.byte "(PLAYER V1.4 BY GLOVER)-ZAK BY K"
.byte "ORDIAUKIS OF KRECIKI <-"
```

- **Text with a specified character count**: it is text of predetermined length determined by the first byte in the group that makes up the text.

```
.byte $37, "(PLAYER V1.4 BY GLOVER)-ZAK BY KORDIAUKIS OF KRECIKI <-"
```

- **Text terminated with zero**: it is text whose length ends when a zero (0) is encountered.

```
.byte "(PLAYER V1.4 BY GLOVER)-ZAK BY KORDIAUKIS OF KRECIKI <-", $00
```

- **Text terminated with a high bit of 1**: text whose length ends as soon as the most significant bit is 1.

```
.byte "(PLAYER V1.4 BY GLOVER)-ZAK BY KORDIAUKIS OF KRECIKI <", $AD
```

- **Left-shifted text**: text where the least significant bit is always 0, because the character code is shifted to the left. Few assemblers support this mode.

```
.byte "P", $A0, $98, $82, $B2, $8A, $A4, "@Vb\h@", $84, $B2, "@", $8E, $98, $9E, $AC, $8
```

- **Text converted to screen code**: text containing screen characters from the C64.

```
dc "(PLAYER V1.4 BY GLOVER)-ZAK BY K"
dc "ORDIAUKIS OF KRECIKI <-"
```

- **Text converted to PetAscii characters**: text containing PetAscii characters from the C64.

```
dc.b "(PLAYER V1.4 BY GLOVER)-ZAK BY K"
dc.b "ORDIAUKIS OF KRECIKI <-"
```

Remember that the way various data types are grouped on a single line to generate the source code is configurable in the options.

6 Sub-data area

For byte data types, an additional attribute[12] can be associated to declare the type of BASIC program represented by those sequences of values, drawing from this list:

- Basic v2.0 (C64/VIC20/PET)
- Basic v3.5 (C16)
- Basic v4.0 (PET/CBM2)
- Basic v7.0 (C128)
- Basic v2.0 with Simons' Basic (C64)
- Basic v2.0 with @Basic (C64)
- Basic v2.0 with Speech Basic v2.7 (C64)
- Basic v2.0 with Final Cartridge III (C64)
- Basic v2.0 with Ultrabasic-64 (C64)
- Basic v2.0 with Graphics Basic (C64)
- Basic v2.0 with WS Basic (C64)
- Basic v2.0 with Pegasus Basic v4.0 (C64)
- Basic v2.0 with Xbasic (C64)
- Basic v2.0 with Drago Basic v2.2 (C64)
- Basic v2.0 with REU-Basic (C64)
- Basic v2.0 with Basic Lightning (C64)
- Basic v2.0 with Magic Basic (C64)
- Basic v2.0 with Blarg (C64)
- Basic v2.0 with WS Basic final (C64)
- Basic v2.0 with Game Basic (C64)
- Basic v2.0 with Basex (C64)
- Basic v2.0 with Super Basic (C64)
- Basic v2.0 with Expanded Basic (C64)
- Basic v2.0 with Super Expander Chip (C64)
- Basic v2.0 with Warsaw Basic (C64)
- Basic v2.0 with Supergrafik 64 (C64)
- Basic v2.0 with Kipper Basic (C64)
- Basic v2.0 with Basic on Bails (C64)

[12]The attribute is used to automatically comment the data area, as will be seen later.

- Basic v2.0 with Eve Basic (C64)
- Basic v2.0 with The Tool 64 (C64)
- Basic v2.0 with Super Expander (VIC20)
- Basic v2.0 with Turtle Basic v1.0 (VIC20)
- Basic v2.0 with Easy Basic (VIC20)
- Basic v2.0 with Basic v4.0 extension (VIC20)
- Basic v2.0 with Basic v4.5 extension (VIC20)
- Basic v2.0 with Expanded Basic (VIC20)
- Basic v2.0 with Handy Basic v1.0 (VIC20)
- Basic v8 Walrusoft' (C128)

A BASIC instruction for the Commodore system is, in fact, formed by these values:

- Memory address (16-bit) in little-endian format of the next BASIC instruction after the current.
- Line number (16-bit) in little-endian format of this BASIC instruction.
- Sequences of:
  - Token of the instruction to be executed (1 or 2 bytes depending on the type of BASIC and its command)
  - parameters of the instruction terminating with a 0 as token.
- A 0 as the next BASIC instruction denotes the end of the program itself.

Some examples of data with BASIC code:

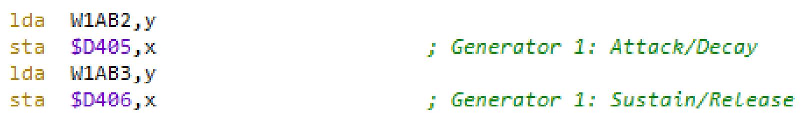
```
.byte $17, $08, $0A, $00, $9E, $20, $28, $32
.byte $30, $37, $33, $29, $20, $50, $45, $54
.byte $53, $50, $45, $45, $44, $00          ; 10 SYS (2073) PETSPEED
.byte $00, $00                              ; <- end of BASIC program ->
```

```
.byte $29, $08, $00, $00, $20, $8F, $20, $54
.byte $48, $49, $53, $20, $49, $53, $20, $41
.byte $4E, $20, $45, $58, $41, $4D, $50, $4C
.byte $45, $20, $42, $41, $53, $49, $43, $20
.byte $50, $52, $4F, $47, $52, $41, $4D, $00 ; 0  REM THIS IS AN EXAMPLE BASIC PROGRAM
.byte $3D, $08, $0A, $00, $20, $99, $22, $48
.byte $45, $4C, $4C, $4F, $20, $57, $4F, $52
.byte $4C, $44, $22, $00                     ; 10  PRINT"HELLO WORLD"
.byte $46, $08, $14, $00, $20, $89, $31, $30
.byte $00                                    ; 20  GOTO10
.byte $00, $00                               ; <- end of BASIC program ->
```

7 Comments

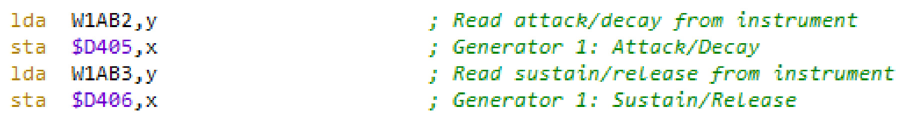
Comments inserted both in the code and in the data are essential for understanding the functioning of the program being analyzed. Therefore, *JC64dis* is capable of automatically commenting certain instructions or memory locations.

For instance, knowing the architecture for which the program was written (a choice made when creating a new project) allows us to associate a known location with its meaning and add it as a comment (in the options, you can enable or disable comments for homogeneous memory address banks):



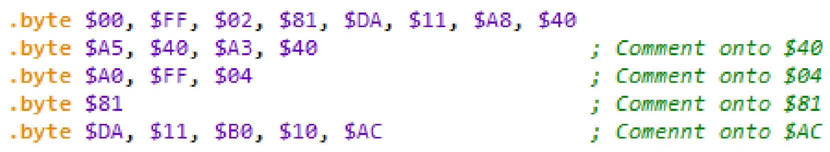
```
lda  W1AB2,y
sta  $D405,x                    ; Generator 1: Attack/Decay
lda  W1AB3,y
sta  $D406,x                    ; Generator 1: Sustain/Release
```

Of course, in *JC64dis*, we can manually add comments or suppress an automatic comment if it is not deemed useful, all to enhance text comprehension:



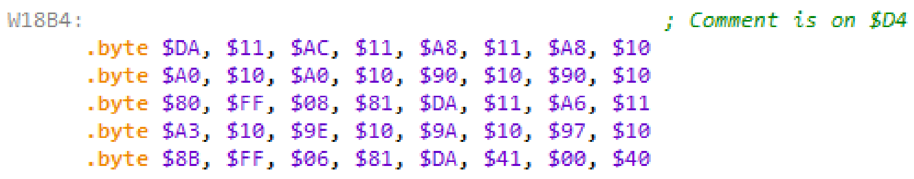
```
lda  W1AB2,y                    ; Read attack/decay from instrument
sta  $D405,x                    ; Generator 1: Attack/Decay
lda  W1AB3,y                    ; Read sustain/release from instrument
sta  $D406,x                    ; Generator 1: Sustain/Release
```

As evident from the images just above, comments on instructions appear at the end of the instruction itself, while comments on data locations always appear near where they are inserted:



```
.byte $00, $FF, $02, $81, $DA, $11, $A8, $40
.byte $A5, $40, $A3, $40                         ; Comment onto $40
.byte $A0, $FF, $04                              ; Comment onto $04
.byte $81                                        ; Comment onto $81
.byte $DA, $11, $B0, $10, $AC                    ; Comennt onto $AC
```

The only variation, made exclusively to improve the readability of the source code, is that in the presence of a label and a comment in the first byte, the comment is placed at the label level.



```
W18B4:                                           ; Comment is on $D4
     .byte $DA, $11, $AC, $11, $A8, $11, $A8, $10
     .byte $A0, $10, $A0, $10, $90, $10, $90, $10
     .byte $80, $FF, $08, $81, $DA, $11, $A6, $11
     .byte $A3, $10, $9E, $10, $9A, $10, $97, $10
     .byte $8B, $FF, $06, $81, $DA, $41, $00, $40
```

Currently, automatic comments are associated with the following events

- Memory locations known in processor instructions.
- Graphical comment for mono or multi-color sprite data.
- Comment on the data with BASIC code.
- Comment on the frequency of A4 note for a musical frequency table[13].

[13]This will be discussed later.

8 Comment block

A comment block is a method for inserting comments across multiple lines. These comments are placed starting from the first column and, if supported by the compiler[14], can be written within specific separation tags that further enhance the readability of the resulting source code.

Although JC64dis utilizes block management to automatically insert comments into the code, these comments are not tied to a memory address. Therefore, they always appear at the top of the source and can be customized or excluded through program options:

```
;****************************
;  JC64dis version 2.9
;
;  Source in Dasm format
;****************************
```

Each comment block inserted into a memory cell is manually executed by the user who utilizes it to document various points in the source code:

```
;===============================
; Track format
;
; FC xx transpose down of xx
; FD xx transpose up of xx
; FE    stop play of track
; FF xx restart from position xx
;================================
trackV1:
      .byte $0F, $10, $02, $02, $02, $02, $02, $02
      .byte $02, $02, $01, $07, $01, $07, $0C, $0C
      .byte $0A, $0A, $0A, $0A, $FF, $00
```

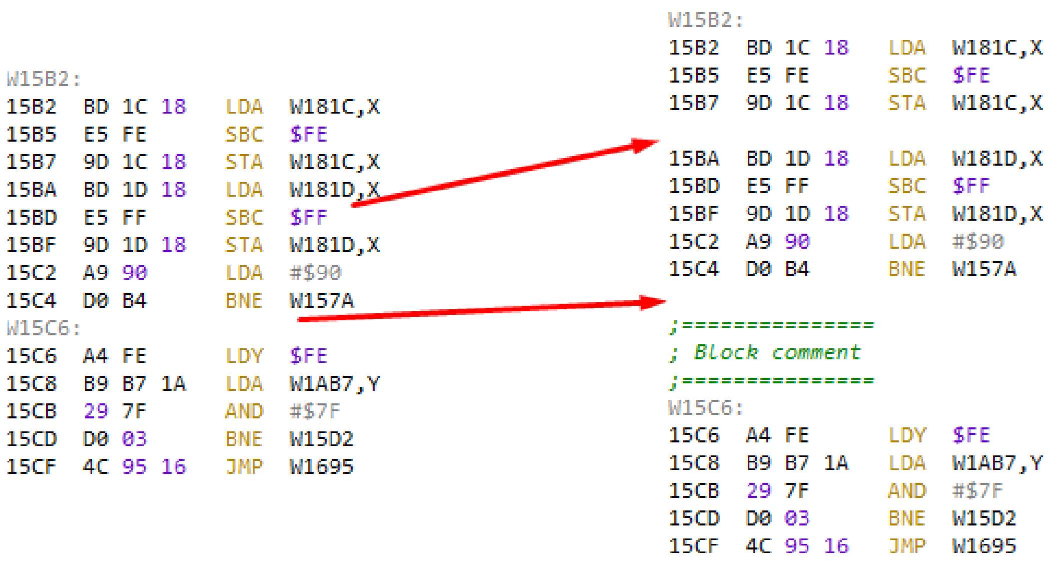
However, a comment block can also be used to create spacing within the text, separating different parts of the code. This action significantly improves readability.

By default, JC64dis adds a line break after each code line where the instruction causes a sudden flow change, such as after a JMP (Jump), RTS (Return from Subroutine), or RTI (Return from Interrupt):

```
W15A1:
      lda  W185F,x
      sta  W1833,x
      lda  #$00
      sta  W181E,x
      sta  W1887,x
      jmp  W1695

W15B2:
      lda  W181C,x
      sbc  $FE
      sta  W181C,x
      lda  W181D,x
      sbc  $FF
      sta  W181D,x
      lda  #$90
      bne  W157A
```

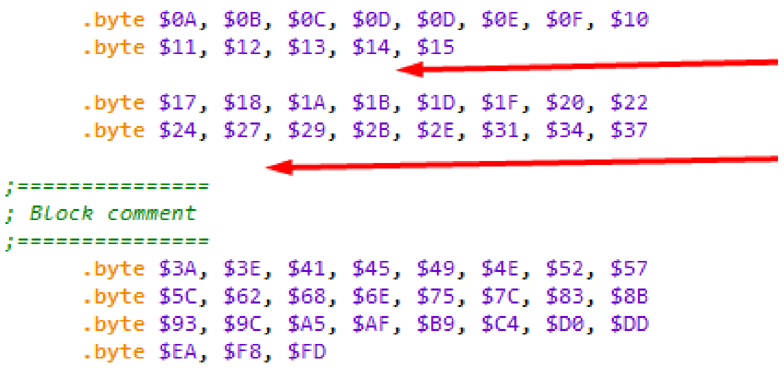
The method to activate the separation of program or data flow into two parts is to insert a space followed by a line break. This will insert an empty line in its place.

[14]Obviously in the options of each compiler it will be possible to select whether it supports it.



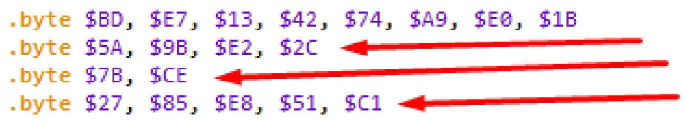
```
W15B2:
15B2  BD 1C 18    LDA  W181C,X
15B5  E5 FE       SBC  $FE
15B7  9D 1C 18    STA  W181C,X
15BA  BD 1D 18    LDA  W181D,X
15BD  E5 FF       SBC  $FF
15BF  9D 1D 18    STA  W181D,X
15C2  A9 90       LDA  #$90
15C4  D0 B4       BNE  W157A
W15C6:
15C6  A4 FE       LDY  $FE
15C8  B9 B7 1A    LDA  W1AB7,Y
15CB  29 7F       AND  #$7F
15CD  D0 03       BNE  W15D2
15CF  4C 95 16    JMP  W1695
```

```
W15B2:
15B2  BD 1C 18    LDA  W181C,X
15B5  E5 FE       SBC  $FE
15B7  9D 1C 18    STA  W181C,X

15BA  BD 1D 18    LDA  W181D,X
15BD  E5 FF       SBC  $FF
15BF  9D 1D 18    STA  W181D,X
15C2  A9 90       LDA  #$90
15C4  D0 B4       BNE  W157A

;===============
; Block comment
;===============
W15C6:
15C6  A4 FE       LDY  $FE
15C8  B9 B7 1A    LDA  W1AB7,Y
15CB  29 7F       AND  #$7F
15CD  D0 03       BNE  W15D2
15CF  4C 95 16    JMP  W1695
```



```
      .byte $0A, $0B, $0C, $0D, $0D, $0E, $0F, $10
      .byte $11, $12, $13, $14, $15

      .byte $17, $18, $1A, $1B, $1D, $1F, $20, $22
      .byte $24, $27, $29, $2B, $2E, $31, $34, $37

;===============
; Block comment
;===============
      .byte $3A, $3E, $41, $45, $49, $4E, $52, $57
      .byte $5C, $62, $68, $6E, $75, $7C, $83, $8B
      .byte $93, $9C, $A5, $AF, $B9, $C4, $D0, $DD
      .byte $EA, $F8, $FD
```

In the case of a data sequence, the comment block can be used to alter the normal grouping of the data itself. For example, if we have an 8-byte grouping and want each line to contain less bytes, we can simply insert a comment block containing only a line break (without spaces).



```
      .byte $BD, $E7, $13, $42, $74, $A9, $E0, $1B
      .byte $5A, $9B, $E2, $2C
      .byte $7B, $CE
      .byte $27, $85, $E8, $51, $C1
```

There is an additional feature related to block comments worth mentioning: macro expansion for automatically creating graphical, parametric comments based on data values in the following cells.

The data relevant to graphics, as we've seen in the case of sprite definitions, but also for characters, is organized in rows by columns, and the colors to be used are determined by appropriate bit combinations.

When analyzing code, being able to visually inspect these objects makes understanding the program much easier, but manually inserting a graphical comment block becomes impractical.

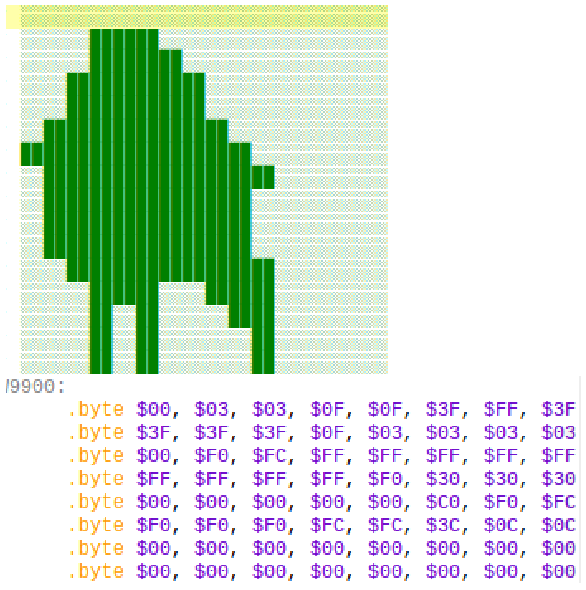
With macros, we simply specify how the subsequent n x m blocks (distinguishing between sprites and character definitions), arranged from bottom to top or vice versa, should be represented graphically.

9 Block comment macro

A macro in a block comment has this syntax: [<***TYPE*#*MODE*#NxM#*DIR***>] where *type*, *mode* and *dir* are the values to insert according to this table:

- [<**CHAR#MONO#NxM#UPDN**>] Shows a grid of NxM monochromatic chars arranged in row that goes from up/sx to up/dx and then go below of one line and so on.
- [<**CHAR#MONO#NxM#DNUP**>] Shows a grid of NxM monochromatic chars arranged in row that goes from up/sx to dn/sx and then go rigth of one line and so on.
- [<**CHAR#MULTI#NxM#UPDN**>] Shows a grid of NxM multicolor chars arranged in row that goes from up/sx to up/dx and then go below of one line and so on.
- [<**CHAR#MULTI#NxM#DNUP**>] Shows a grid of NxM multicolor chars arranged in row that goes from up/sx to dn/sx and then go rigth of one line and so on.
- [<**SPRITE#MONO#NxM#UPDN**>] Shows a grid of NxM monochromatic sprites arranged in row that goes from up/sx to up/dx and then go below of one line and so on.
- [<**SPRITE#MONO#NxM#DNUP**>] Shows a grid of NxM monochromatic sprites arranged in row that goes from up/sx to dn/sx and then go rigth of one line and so on.
- [<**SPRITE#MULTI#NxM#UPDN**>] Shows a grid of NxM multicolor sprites arranged in row that goes from up/sx to up/dx and then go below of one line and so on.
- [<**SPRITE#MULTI#NxM#DNUP**>] Shows a grid of NxM multicolor sprites arranged in row that goes from up/sx to dn/sx and then go rigth of one line and so on.

As an example, this [<***CHAR*#*MONO*#3x2#DNUP**>] macro produces this result:



```
I9900:
        .byte $00, $03, $03, $0F, $0F, $3F, $FF, $3F
        .byte $3F, $3F, $3F, $0F, $03, $03, $03, $03
        .byte $00, $F0, $FC, $FF, $FF, $FF, $FF, $FF
        .byte $FF, $FF, $FF, $FF, $F0, $30, $30, $30
        .byte $00, $00, $00, $00, $00, $C0, $F0, $FC
        .byte $F0, $F0, $F0, $FC, $FC, $3C, $0C, $0C
        .byte $00, $00, $00, $00, $00, $00, $00, $00
        .byte $00, $00, $00, $00, $00, $00, $00, $00
```

10 Labels

A label is the method used in assembly source code to reference a memory location, whether it's for code or data, at any point in the program. For this reason, *JC64dis* automatically creates labels[15] called **W** followed by their **16-bit hexadecimal address** whenever it encounters an instruction that refers to that memory area.

```
        ldx   #$00
        txa
W110B:
        sta   W1046,x
        inx
        cpx   #$BD
        bne   W110B
```

In the example above, two labels have been defined: **W110B**, which points to the jump loop, and **W1046**, which points to the write instruction."

Just like comments, labels can also be created manually[16]. Importantly, this allows for renaming (since a manual label takes precedence over an automatically generated label in the same location) those automatic labels by giving them meaningful names, as dictated by good programming practices.

```
        ldx   #$00
        txa
loopClear:
        sta   noteDuration,x
        inx
        cpx   #$BD
        bne   loopClear
```

An important consideration is that labels are not automatically created for memory areas external to the program itself (or are in garbage area):

```
        lda   #$09                          ; Test bit, gate on
        sta   $D404,x                       ; Voice 1: Control registers
        sta   stopTrack,x                   ; Stop track if 0
```

In the example above, $D404 (the address of the C64 sound chip) remains unassigned. This choice may seem unconventional, but it serves a purpose: since the label is external to the program area, it can only be assembled if declared as a constant, which is typically done at the beginning of the program. Therefore, users are allowed to create their own labels for external areas that become constants.

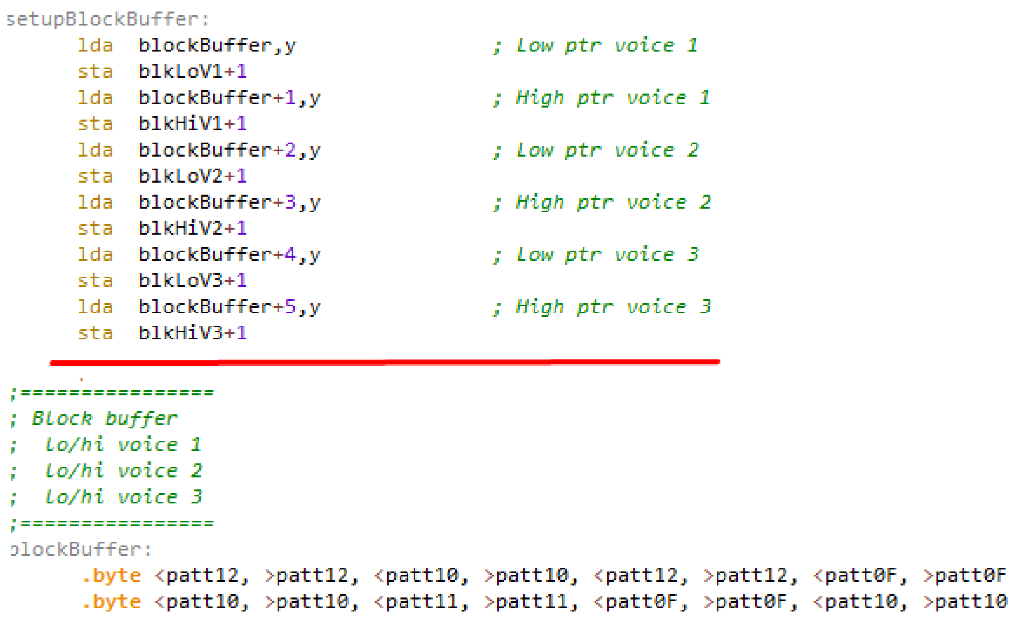
```
SID_CTRL1 = $D404

        lda   #$09                          ; Test bit, gate on
        sta   SID_CTRL1,x                   ; Voice 1: Control registers
        sta   stopTrack,x                   ; Stop track if 0
```

[15]However, even though it's possible to delete an automatically created label, this operation is effective only when that memory location is no longer actively referenced (which happens when we convert a code area into data, for example). Otherwise, the label will be recreated during the next disassembly execution.

[16]In any case, the program does not allow the creation of a duplicate label or potentially similar to those automatically generated (Wxxx) to prevent ambiguities that could lead to source code compilation errors.

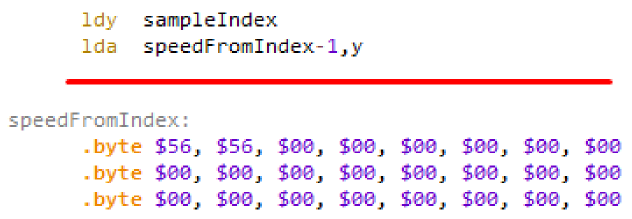
11 Relative labels

JC64dis allows you to create a reference to an existing label for a memory location, relative to the distance (+/-) from that label. This technique is often used in assembler programs when dealing with a data buffer and referring to various subsequent positions starting from the buffer itself[17]:



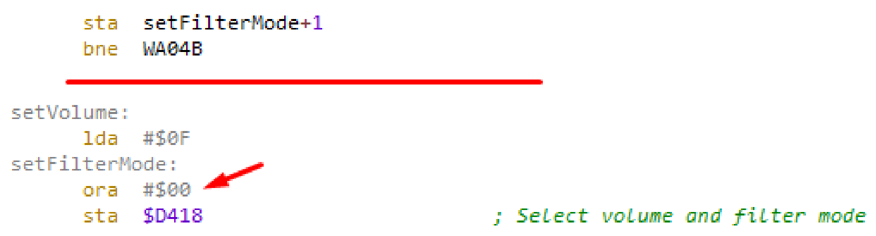
```
setupBlockBuffer:
      lda  blockBuffer,y              ; Low ptr voice 1
      sta  blkLoV1+1
      lda  blockBuffer+1,y            ; High ptr voice 1
      sta  blkHiV1+1
      lda  blockBuffer+2,y            ; Low ptr voice 2
      sta  blkLoV2+1
      lda  blockBuffer+3,y            ; High ptr voice 2
      sta  blkHiV2+1
      lda  blockBuffer+4,y            ; Low ptr voice 3
      sta  blkLoV3+1
      lda  blockBuffer+5,y            ; High ptr voice 3
      sta  blkHiV3+1

;================
; Block buffer
;  lo/hi voice 1
;  lo/hi voice 2
;  lo/hi voice 3
;================
lockBuffer:
      .byte <patt12, >patt12, <patt10, >patt10, <patt12, >patt12, <patt0F, >patt0F
      .byte <patt10, >patt10, <patt11, >patt11, <patt0F, >patt0F, <patt10, >patt10
```

Even though it is used less frequently, backward references (negative ones) can also be very useful for keeping the code clean[18]:



```
      ldy  sampleIndex
      lda  speedFromIndex-1,y

speedFromIndex:
      .byte $56, $56, $00, $00, $00, $00, $00, $00
      .byte $00, $00, $00, $00, $00, $00, $00, $00
      .byte $00, $00, $00, $00, $00, $00, $00, $00
```

Relative labels are also automatically created by the program when an instruction operates on a memory address that belongs to an operand of another instruction. This is the only system for identifying that location[19]:



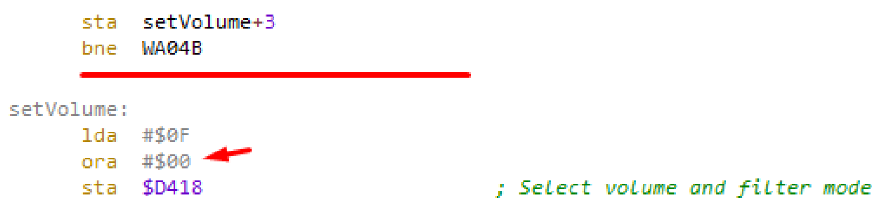
```
      sta  setFilterMode+1
      bne  WA04B

 setVolume:
      lda  #$0F
 setFilterMode:
      ora  #$00
      sta  $D418                  ; Select volume and filter mode
```

[17]In the example, *blockBuffer* contains a table of low/high addresses, one for each of the 3 entries. Creating a separate label for each byte would have significantly cluttered the code.

[18]In the example, *sampleIndex* holds values starting from 1. Therefore, referencing backward by one unit ensures alignment with table *speedFromIndex*.

[19]In the example, the byte of the ORA operation is dynamically changed by the program flow.

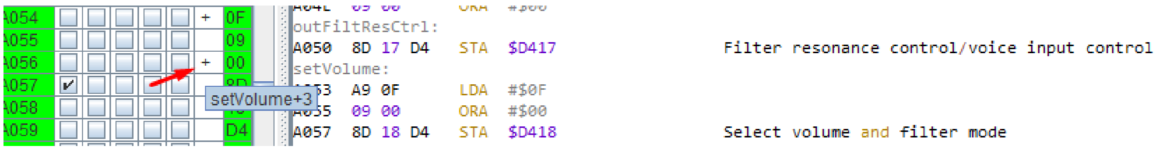
Automatically, the label is always created by referring to the same instruction. However, manually, we can force an address (even to be negative related) with respect to another location that we prefer to relate to[20]:



```
        sta   setVolume+3
        bne   WA04B

setVolume:
        lda   #$0F
        ora   #$00
        sta   $D418                          ; Select volume and filter mode
```

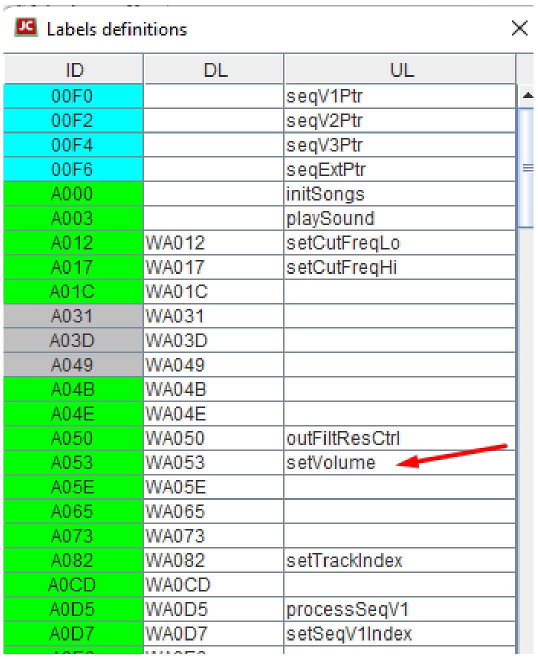
Relative labels are limited to a maximum distance of +/- 255 positions (1 byte) from the associated address to prevent potential compilation issues.

Furthermore, visually, relative labels are indicated by their respective sign (+/-) within the memory table to facilitate their identification (in fact, hovering over that cell displays the associated relative label)



```
A054  ☐ ☐ ☐ ☐ ☐ +  0F    outFiltResCtrl:
A055  ☐ ☐ ☐ ☐ ☐    09    A050   8D 17 D4    STA   $D417            Filter resonance control/voice input control
A056  ☐ ☐ ☐ ☐ ☐ +  00    setVolume:
A057  ☑ ☐ ☐ ☐ ☐    8D         A9 0F       LDA   #$0F
A058  ☐ ☐ ☐ ☐ ☐               09 00       ORA   #$00
A059  ☐ ☐ ☐ ☐ ☐    D4    A057   8D 18 D4    STA   $D418            Select volume and filter mode
```

In the program window that displays the list of all created labels (both automatically and manually), relative labels are not shown, only the regular ones, to keep the display clear.”



Labels definitions

ID	DL	UL
00F0		seqV1Ptr
00F2		seqV2Ptr
00F4		seqV3Ptr
00F6		seqExtPtr
A000		initSongs
A003		playSound
A012	WA012	setCutFreqLo
A017	WA017	setCutFreqHi
A01C	WA01C	
A031	WA031	
A03D	WA03D	
A049	WA049	
A04B	WA04B	
A04E	WA04E	
A050	WA050	outFiltResCtrl
A053	WA053	setVolume
A05E	WA05E	
A065	WA065	
A073	WA073	
A082	WA082	setTrackIndex
A0CD	WA0CD	
A0D5	WA0D5	processSeqV1
A0D7	WA0D7	setSeqV1Index

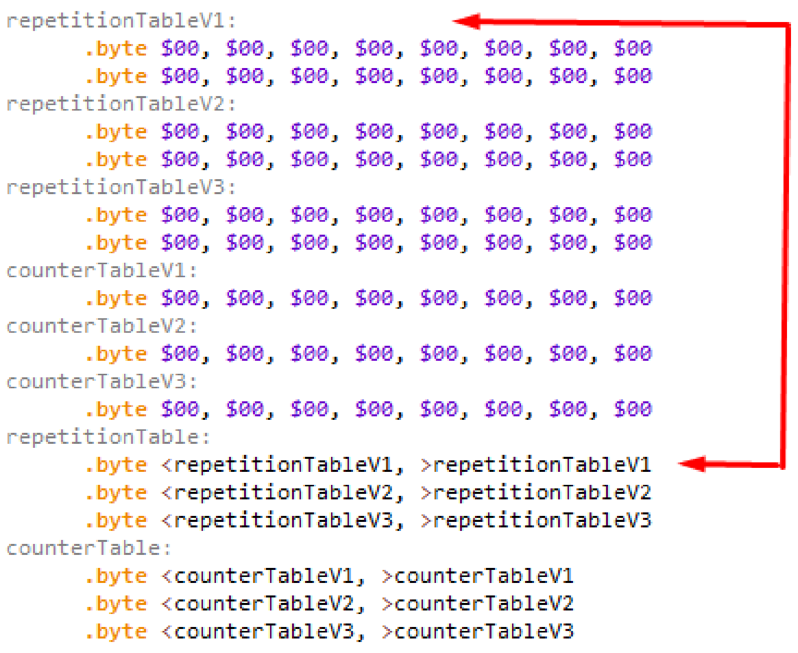
Now the presence of the relative value and the type of relationship in the definition of a memory cell in *JC64dis* becomes clear: they represent precisely the relative labels.

[20]In the example, now the relative reference is forced to be on the *SetVolume* label.

12 References to Labels

When a 16-bit address is written in a data area, it is split into the low byte and the high byte. To achieve relocatable code, it is necessary to associate those values with the address that generated them during its decomposition. In assembly language, there are specific operators for the low part (<) and the high part (>) that, given an address, return its low and high values.

Identifying these bytes from a machine code program requires some experience. However, once recognized, *JC64dis* allows you to easily[21] create the association for the low and high bytes to the memory location it pertains to:



```
repetitionTableV1:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
      .byte $00, $00, $00, $00, $00, $00, $00, $00
repetitionTableV2:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
      .byte $00, $00, $00, $00, $00, $00, $00, $00
repetitionTableV3:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
      .byte $00, $00, $00, $00, $00, $00, $00, $00
counterTableV1:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
counterTableV2:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
counterTableV3:
      .byte $00, $00, $00, $00, $00, $00, $00, $00
repetitionTable:
      .byte <repetitionTableV1, >repetitionTableV1
      .byte <repetitionTableV2, >repetitionTableV2
      .byte <repetitionTableV3, >repetitionTableV3
counterTable:
      .byte <counterTableV1, >counterTableV1
      .byte <counterTableV2, >counterTableV2
      .byte <counterTableV3, >counterTableV3
```

In the cases that low and high part are in little-endian order, you can use the world data type to obtain the address directly without operators:

```
repetitionTable:
      .word repetitionTableV1, repetitionTableV2, repetitionTableV3
counterTable:
      .word counterTableV1, counterTableV2, counterTableV3
```

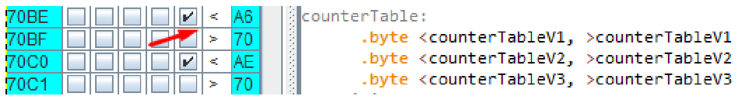
References can also be associated with the memory cells of a CPU instruction[22], as happens for example with this code:

```
initCopy:
      lda  #<buffer
      sta  $FB
      lda  #>buffer
      sta  $FC

      lda  #<tuneDataBuffer
      sta  $FD
      lda  #>tuneDataBuffer
      sta  $FE
```

[21] There are various specific utilities, including a wizard with real-time preview of the changes that will be made to the memory cells, complete with association of progressively numbered labels.

[22] In this case the association must be created in the cell in which the operand is contained, not in the cell with the opcode of the instruction.

As in the case of the relative labels, symbols are also shown in the memory table for the references to easily understand the status of that byte (in fact it is still used the type and relationsip relation of the memory cell defined before).

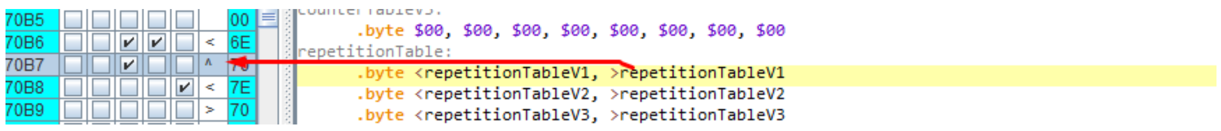


70BE	☐	☐	☐	☐	✓	<	A6
70BF	☐	☐	☐	☐	☐	>	70
70C0	☐	☐	☐	☐	✓	<	AE
70C1	☐	☐	☐	☐	☐	>	70

```
counterTable:
        .byte <counterTableV1, >counterTableV1
        .byte <counterTableV2, >counterTableV2
        .byte <counterTableV3, >counterTableV3
```

What happens when there are both relative labels and references to such values? The answer is that the program handles them simultaneously, appropriately mapping these values to the previously defined type and relationship fields.

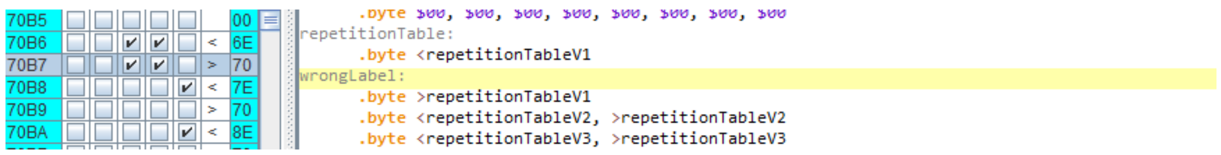
Let's consider a scenario where we have an address that serves as both a reference to a memory cell (*repetitionTableV1*) and a relative label (+1) to *repetitionTable*:



70B5	☐	☐	☐	☐	☐		00
70B6	☐	☐	✓	✓	☐	<	6E
70B7	☐	☐	✓	☐	☐	^	70
70B8	☐	☐	☐	☐	✓	<	7E
70B9	☐	☐	☐	☐	☐	>	70

```
        .byte $00, $00, $00, $00, $00, $00, $00, $00
repetitionTable:
        .byte <repetitionTableV1, >repetitionTableV1
        .byte <repetitionTableV2, >repetitionTableV2
        .byte <repetitionTableV3, >repetitionTableV3
```

As you can see, the relationship is now indicated by a new symbol, and the program automatically manages both cases. Otherwise, I would have encountered what follows, namely the use of the label *wrongLabel* because it is referenced elsewhere in the program:



70B5	☐	☐	☐	☐	☐		00
70B6	☐	☐	✓	✓	☐	<	6E
70B7	☐	☐	✓	✓	☐	>	70
70B8	☐	☐	☐	☐	✓	<	7E
70B9	☐	☐	☐	☐	☐	>	70
70BA	☐	☐	☐	☐	✓	<	8E

```
        .byte $00, $00, $00, $00, $00, $00, $00, $00
repetitionTable:
        .byte <repetitionTableV1
wrongLabel:
        .byte >repetitionTableV1
        .byte <repetitionTableV2, >repetitionTableV2
        .byte <repetitionTableV3, >repetitionTableV3
```

Considering this legend:

- *Plus*: reference +
- *Minus*: reference -
- *Major*: reference #<
- *Minor*: reference #>

The correspondence between the code on the screen and the meaning for the program becomes:

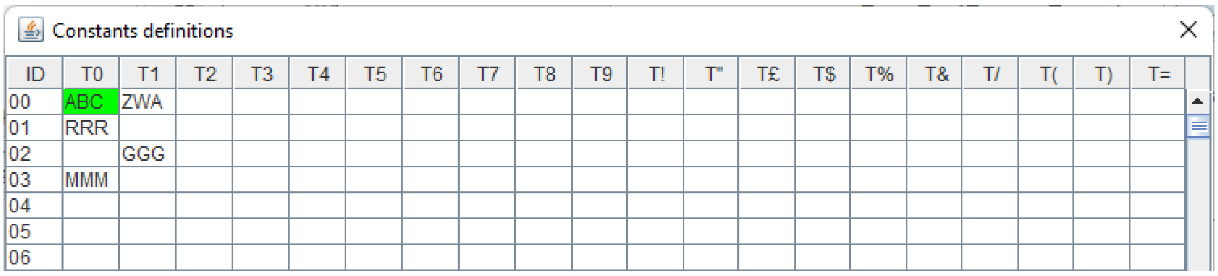
- + Plus
- - Minus
- > Major
- < Minor
- ˆ Plus Major
- \ Plus Minor
- , Minus Major
- ; Minus Minor

13 Constants

Constants, which were previously explained as being possible to create for labels external to the program area, can actually be generated in a dedicated editor and associated with memory cells as attributes of the same cell. This allows us to have different constants for the same value if they are used in different parts of the program and for different purposes (thus making the code more readable and easily maintainable).

For example, the value $ff could represent the end of a music player pattern, but it could also represent the end of a track. Therefore, we would like to be able to name them both END_PATTERN and END_TRACK, both with the value $ff, and insert these constants within instruction operands or data area bytes.

In the editor, there are 20 columns numbered from T0 to T9 and corresponding to the characters T+shift+0 to T+shift+9. In these columns, we can insert both the label associated with the constant and its optional description.

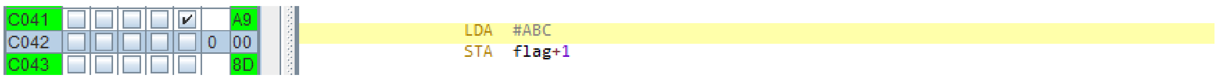

Constants definitions

ID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T!	T"	T£	T$	T%	T&	T/	T(	T)	T=
00	ABC	ZWA																		
01	RRR																			
02		GGG																		
03	MMM																			
04																				
05																				
06																				

In the example above, the constant of type T0 for the value 0, labeled as ABC, has a comment that will appear next to its declaration in the assembly source.

```
;Constants of type 0
ABC = $00                              ; Test description
RRR = $01
MMM = $03

;Constants of type 1
ZWA = $00
GGG = $02
```

When a constant is associated with a memory cell, the value of that cell in the source code is replaced by the constant, whether it is used as an operand in an instruction or within a data area. Additionally, the symbol (without the 'T') from the chosen column appears inside the memory table, visually indicating our selection.


```
C041 ☐☐☐☐☑   A9              LDA  #ABC
C042 ☐☐☐☐☐ 0 00              STA  flag+1
C043 ☐☐☐☐☐   8D
```

It is undeniable that a source code in which constants appear becomes easier to understand. For instance, consider this example where certain commands are represented as constants within the data area:

```
patt01:
      .byte SUS, $0E                   ; sustain (0x)
      .byte DUR, $00                   ; duration
      .byte FRM, $F3                   ; filter resonance/mode
      .byte IST, $0B                   ; instrument
      .byte CHF, $55                   ; filter cut off high
```

In addition to being able to define a constant between 0 and 255, it is also possible to define constants to be assigned to numbers from 256 up to 32767, which corresponds to a word. Therefore, when dealing with 2-byte operands or data areas formatted as words, assigning these constants results in their substitution. Consider the following example:

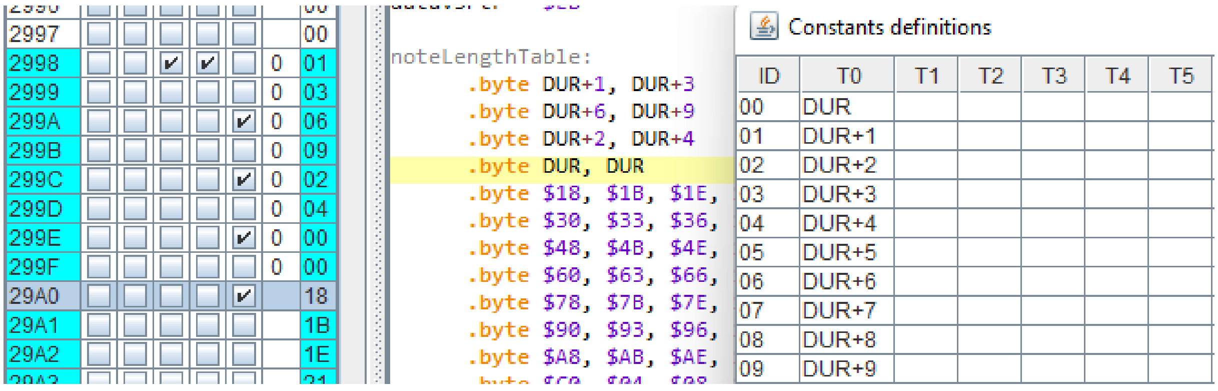
```
tune2:
      .word $01E8, $072C, $013E         ; Length of tune 1,2,3
      .word TEMPO450, VOL15, DEF0, DEF1
      .word WAVNOI, ATK0, DCY0, SUS15
      .word FLT_THROUGH_NO, RLS0, C3_32ND, C7_32ND
      .word C5_32ND, DCY9, SUS0, WAVSAW
      .word END1, HED4, C5_32ND_TIE, Eb5_32ND_TIE
      .word G5_32ND_TIE, TAL1, REST_32ND, CAL1
      .word HED4, Bb4_32ND_TIE, D5_32ND_TIE, F5_32ND_TIE
      .word TAL1, REST_32ND, CAL1, HED4
```

Constants of this type are used in the following context:

```
VOL1 = $1E01
POR30 = $1E03
HED30 = $1E36
PUS30 = $1E56
VDP30 = $1E76
TPS01 = $1EA6
HED31 = $1F36
DCY2 = $2001
TEMPO450 = $2006
```

Even for constants, it is verified that the entered name is unique. However, this check is omitted when dealing with free constants. By 'free constants', we mean constants where there is a mathematical formula[23] that can depend on other constants or memory labels. But what utility do they serve?

For instance, if we denote the duration value of a musical note with the constant DUR, then DUR+1 represents the next duration step, DUR+2 represents the one after that, and so on. Instead of having N constants named DUR_0, DUR_1, etc., we have a single constant DUR, and additional values are expressed as DUR+x.



2997	☐	☐	☐	☐	☐		00
2998	☐	☐	✓	✓	☐	0	01
2999	☐	☐	☐	☐	☐	0	03
299A	☐	☐	☐	☐	✓	0	06
299B	☐	☐	☐	☐	☐	0	09
299C	☐	☐	☐	☐	✓	0	02
299D	☐	☐	☐	☐	☐	0	04
299E	☐	☐	☐	☐	✓	0	00
299F	☐	☐	☐	☐	☐	0	00
29A0	☐	☐	☐	☐	✓		18
29A1	☐	☐	☐	☐	☐		1B
29A2	☐	☐	☐	☐	☐		1E

```
noteLengthTable:
      .byte DUR+1, DUR+3
      .byte DUR+6, DUR+9
      .byte DUR+2, DUR+4
      .byte DUR, DUR
      .byte $18, $1B, $1E,
      .byte $30, $33, $36,
      .byte $48, $4B, $4E,
      .byte $60, $63, $66,
      .byte $78, $7B, $7E,
      .byte $90, $93, $96,
      .byte $A8, $AB, $AE,
```

Constants definitions

ID	T0	T1	T2	T3	T4	T5
00	DUR					
01	DUR+1					
02	DUR+2					
03	DUR+3					
04	DUR+4					
05	DUR+5					
06	DUR+6					
07	DUR+7					
08	DUR+8					
09	DUR+9					

[23] Attention, the program does not verify that the formulas entered have a correct value, but simply replaces in the source the occurrence of the value to which it is associated with the formula itself.

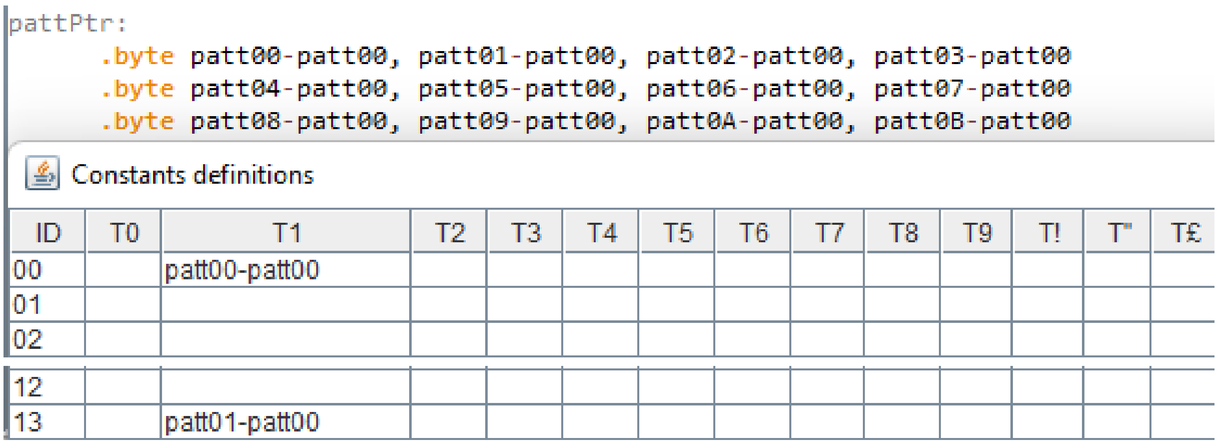
14 Related relative address

The constants we have just seen can also be used to address data areas that contain relative addresses within a table. These addresses are placed as offsets relative to the base address. In the source code, these offsets are determined by the difference between the position of the relevant location and the base address. Therefore, it suffices to create constants representing this difference value and insert the appropriate extraction symbol (if the offset is 16 bit) for either the high or low part (otherwise just the byte).

Let's consider this data area (pattPtr), which may seem inconspicuous at first glance. However, upon closer observation, those numbers are nothing but offsets of patterns relative to the first one (patt00):

```
pattPtr:
      .byte $00, $13, $35, $48, $4D, $58, $5B, $6D
      .byte $79, $82, $89, $B6
;=============================
; Pattern format:
;
; 01..5F  note
; 60..7F  INx instrument
; 80      GTF Gate off
; 81..EF  Dxx Duration xx
; F0..FF: FL0..FLF filter table x
;=============================
patt00:
      .byte FL5, IN0, D40, $09, FL9, $10, FL2, $09
      .byte FL8, $90, $10, GTF, GTF, FL3, $10, IN6
      .byte D01, FL4, $00
patt01:
      .byte D01, IN1, $2D, IN2, $2D, IN1, $34, IN2
      .byte $34, IN1, $2F, IN2, $2F, IN1, $30, IN2
      .byte $30, IN1, $39, IN2, $39, IN1, $40, IN2
      .byte $40, IN1, $3B, IN2, $3B, IN1, $3C, IN2
      .byte $3C, $00
patt02:
      .byte IN3, D08, $21, GTF, $2D, GTF, $2B, GTF
      .byte $2A, GTF, $28, GTF, $34, GTF, $32, GTF
      .byte $31, GTF, $00
```

All we need to do is define suitable constants to make the result more meaningful:



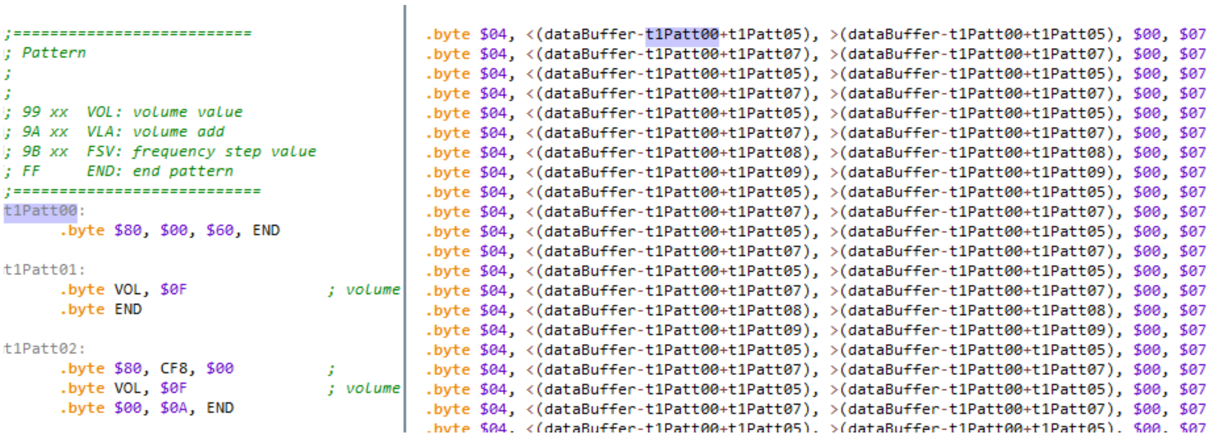
```
pattPtr:
      .byte patt00-patt00, patt01-patt00, patt02-patt00, patt03-patt00
      .byte patt04-patt00, patt05-patt00, patt06-patt00, patt07-patt00
      .byte patt08-patt00, patt09-patt00, patt0A-patt00, patt0B-patt00
```

Constants definitions

ID	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T!	T"	T£
00		patt00-patt00											
01													
02													
12													
13		patt01-patt00											

This works well, but in the presence of offsets that are later relocated to different memory positions (as often happens to optimize its usage), the formula we need to write becomes even more complex.

In this case, the system allows us to define references to memory addresses[24], prompting us to input the base and destination[25]. After that, the address (in composite formula) is automatically generated by the system:



```
;==========================
; Pattern
;
;
; 99 xx  VOL: volume value
; 9A xx  VLA: volume add
; 9B xx  FSV: frequency step value
; FF     END: end pattern
;===========================
t1Patt00:
       .byte $80, $00, $60, END

t1Patt01:
       .byte VOL, $0F                ; volume
       .byte END

t1Patt02:
       .byte $80, CF8, $00           ;
       .byte VOL, $0F                ; volume
       .byte $00, $0A, END
```

```
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt08), >(dataBuffer-t1Patt00+t1Patt08), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt09), >(dataBuffer-t1Patt00+t1Patt09), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt08), >(dataBuffer-t1Patt00+t1Patt08), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt09), >(dataBuffer-t1Patt00+t1Patt09), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt07), >(dataBuffer-t1Patt00+t1Patt07), $00, $07
.byte $04, <(dataBuffer-t1Patt00+t1Patt05), >(dataBuffer-t1Patt00+t1Patt05), $00, $07
```

This is, in fact, the relocation code that moves the patterns into the *dataBuffer*, starting from pattern position *t1Patt00*:

```
tune1:
       ldx   #$00
loopInitTune1:
       lda   instr0Tune1,x
       sta   instr0,x
       lda   instr4Tune1,x
       sta   instr4,x
       lda   instr8Tune1,x
       sta   instr8,x
       dex
       bne   loopInitTune1

       lda   #<t1Patt00
       sta   $FC
       lda   #>t1Patt00
       sta   $FD

       ldx   #$0F
       lda   #<dataBuffer
       sta   $FA
       lda   #>dataBuffer
       sta   $FB
copyBlock:
       ldy   #$00
loopCopy:
       lda   ($FC),y
       sta   ($FA),y
       dey
       bne   loopCopy

       inc   $FB
       inc   $FD
       dex
       bne   copyBlock
```

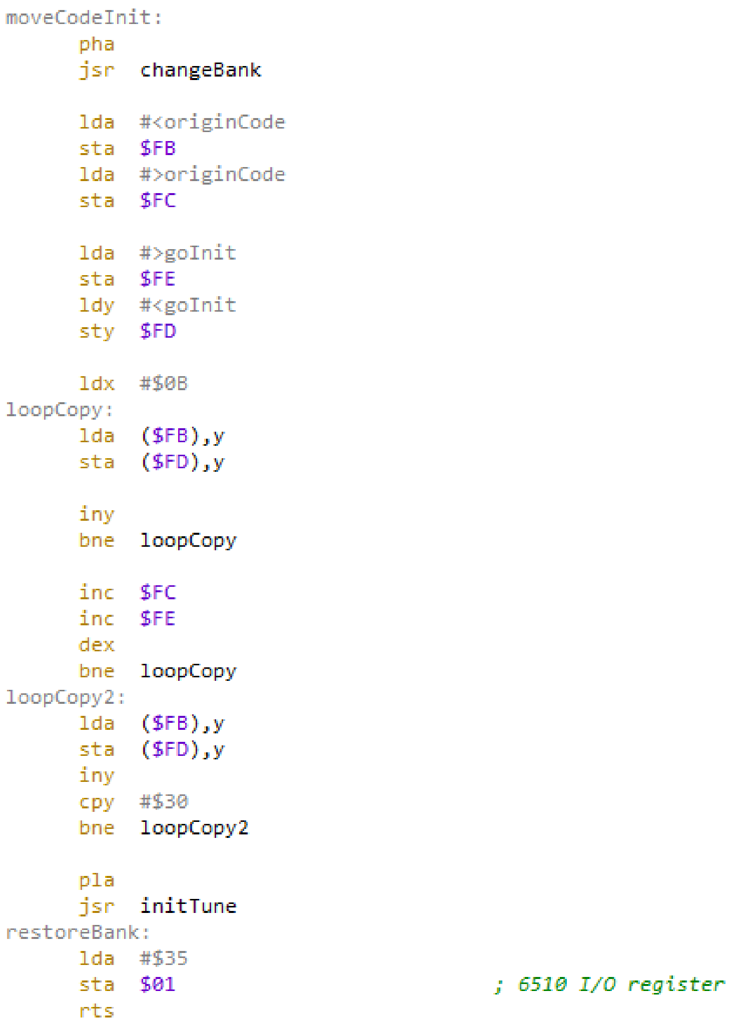
[24]The assignment occurs through a specific mask that applies the addresses to the selection made.
[25]These are two fields we had seen in the memory cell definition for the program.

15 Code Relocation and Memory Patches

There are two features of a program that alter the original file (thus preventing an identical binary code from being reconstructed by assembling the obtained source). However, these features serve a purpose.

1. **Value Variation in Binary Files**: The first feature allows for modifying the values present in the binary file being analyzed. The new value replaces the existing one and becomes the effective value disassembled during execution. This enables us to address bugs in the code through "old-school" patches. For instance, we can alter the opcodes of instructions (e.g., to achieve infinite lives in a game) or increment other values (such as time for completing a game level).

2. **Code Relocation**: The second feature involves relocating a piece of code to a different memory area. This adjustment accommodates the program's behavior during its initial phases, allowing us to study the source code cleanly. Practically, relocation involves copying the relevant code section to the destination area. Consequently, the original area remains visible, making it advisable to later declare it as garbage.



```
moveCodeInit:
      pha
      jsr   changeBank

      lda   #<originCode
      sta   $FB
      lda   #>originCode
      sta   $FC

      lda   #>goInit
      sta   $FE
      ldy   #<goInit
      sty   $FD

      ldx   #$0B
loopCopy:
      lda   ($FB),y
      sta   ($FD),y

      iny
      bne   loopCopy

      inc   $FC
      inc   $FE
      dex
      bne   loopCopy
loopCopy2:
      lda   ($FB),y
      sta   ($FD),y
      iny
      cpy   #$30
      bne   loopCopy2

      pla
      jsr   initTune
restoreBank:
      lda   #$35
      sta   $01                          ; 6510 I/O register
      rts
```

In the example above the code is relocated (by the program) from *originCode* to *goInit* and the call to *initTune* occurs in the new area. Without the relocation applied in *JC64dis* the instruction that jumps to *initTune* would be on an empty memory area and therefore meaningless.

16 Automatic Type Detection

JC64dis has the capability to automatically determine whether a memory area is code or data type when analyzing a file in the C64 SID format[26].

It features an integrated SID player (CRSID) that, while not able to perfectly emulate all files and may not provide the most faithful audio quality, allows you to track the executed/read/written state for each memory cell while listening to music.

To use this functionality, follow these steps:

- Listen to each individual song contained in the file in its entirety (at normal speed or accelerated to save time).
- Load the files located in the temporary area (as set in the options) into your project. These files should have the same name as the SID file, followed by an underscore, the tune number, and the .bin extension.
- Apply the flags contained in the newly loaded files.

Based on these flags, the areas that were “touched” by the program during emulation are now categorized as either code or data type, making our initial task easier.

The flags (SIDLN) for a memory cell contain the following information:

- Read Memory: The memory was read by the program.
- Written Memory: The memory was written by the program.
- Executed Memory: The memory was executed by the program.
- Initially Read Memory: The first operation was reading the memory.
- Initially Written Memory: The first operation was writing to the memory.
- Initially Executed Memory: The first operation was executing an instruction in memory.

Given these flags, the algorithm that marks the memory becomes:

- Assigns memory as data if it was read or written (whether or not it is the first operation)
- Assigns memory as code if it has been executed (whether or not it is the first operation): this assignment takes priority for the program over the previous one.

This feature is however experimental (since CRSID has been patched to generate SDLN flags) and the results, while good, are not 100% complete.

[26]So the file emulation format that contains the player code and the music data that, if executed, produce the music onto the C64.

17 A4 Frequency

Previously, it was mentioned that the program is capable of determining the frequency of the A4 note used in a musical composition and using it as a comment. In reality, it does more than that—it also marks the area of this table as a data type.

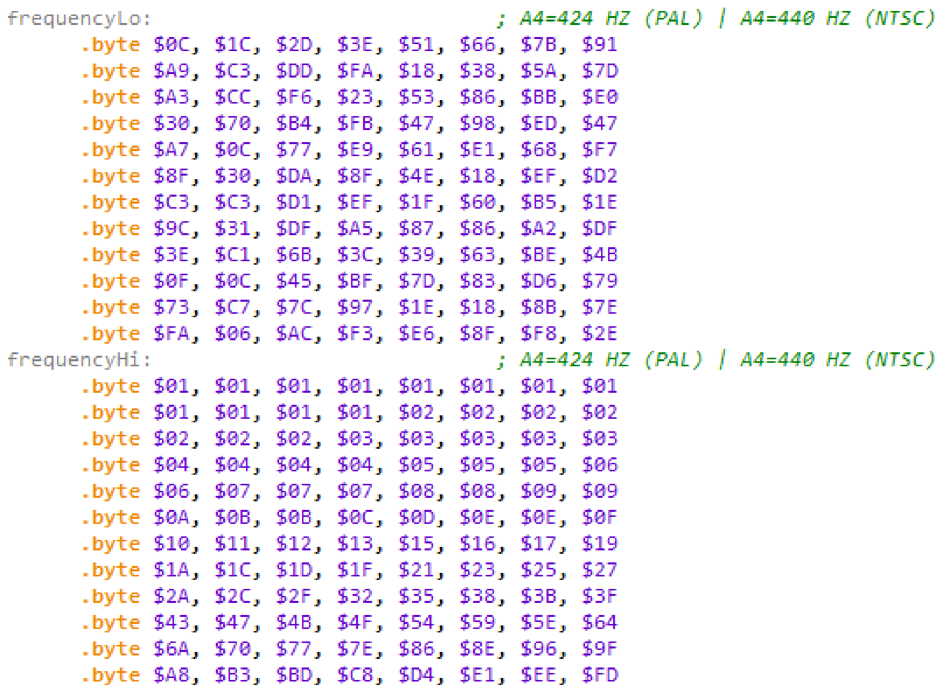
When programming the SID sound chip, you have 8 octaves of note range available. Therefore, a frequency is chosen to serve as the reference for all other notes, conventionally set as the A4 note. The program can determine this frequency by leveraging certain "statistical" analyses of the data[27], not just the code itself[28].

For this reason:

- It might identify areas that coincidentally resemble musical frequencies, resulting in a note that doesn't match.
- It could identify an area of musical frequencies but fail to precisely locate the start and end, detecting an A4 note slightly shifted higher or lower.
- It might not identify the table at all due to bugs introduced by the programmer in the data, which could also affect the played notes if present in the composition.

The methods used are listed here[29]:

- **Linear table**: the 96 frequency notes are divided into two linear tables, with a lower part and an upper part. The order in which these two tables appear and their distance from each other can be entirely variable (depending on how the programmer has organized the memory). However, the program searches for them regardless until a match is found.



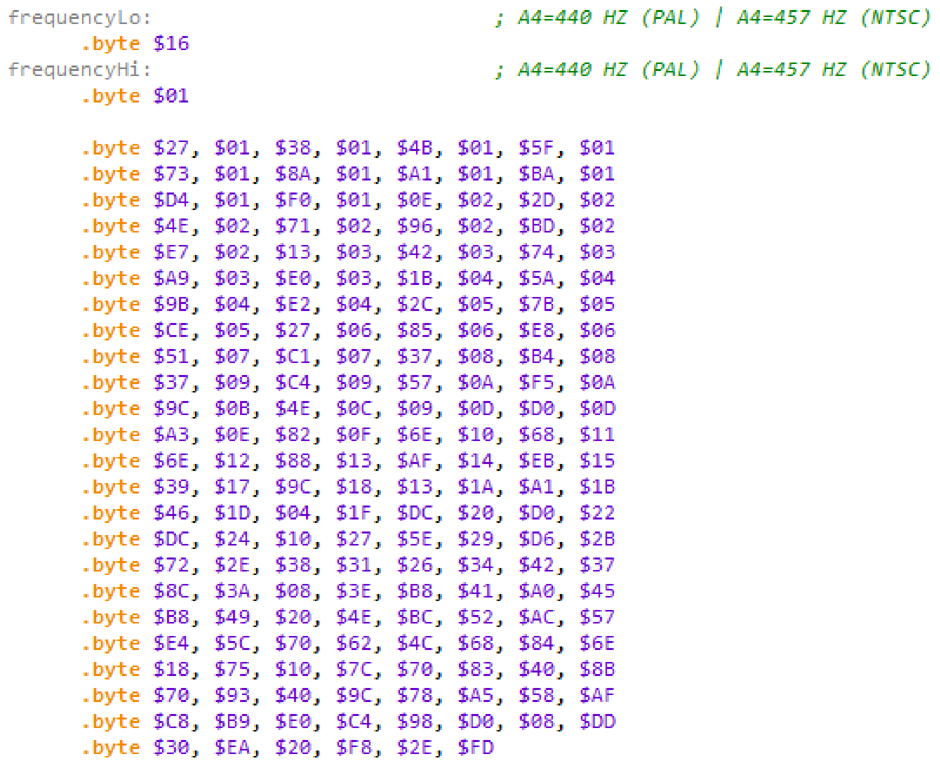
```
frequencyLo:                          ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $0C, $1C, $2D, $3E, $51, $66, $7B, $91
        .byte $A9, $C3, $DD, $FA, $18, $38, $5A, $7D
        .byte $A3, $CC, $F6, $23, $53, $86, $BB, $E0
        .byte $30, $70, $B4, $FB, $47, $98, $ED, $47
        .byte $A7, $0C, $77, $E9, $61, $E1, $68, $F7
        .byte $8F, $30, $DA, $8F, $4E, $18, $EF, $D2
        .byte $C3, $C3, $D1, $EF, $1F, $60, $B5, $1E
        .byte $9C, $31, $DF, $A5, $87, $86, $A2, $DF
        .byte $3E, $C1, $6B, $3C, $39, $63, $BE, $4B
        .byte $0F, $0C, $45, $BF, $7D, $83, $D6, $79
        .byte $73, $C7, $7C, $97, $1E, $18, $8B, $7E
        .byte $FA, $06, $AC, $F3, $E6, $8F, $F8, $2E
frequencyHi:                          ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $01, $01, $01, $01, $01, $01, $01, $01
        .byte $01, $01, $01, $01, $02, $02, $02, $02
        .byte $02, $02, $02, $03, $03, $03, $03, $03
        .byte $04, $04, $04, $04, $05, $05, $05, $06
        .byte $06, $07, $07, $07, $08, $08, $09, $09
        .byte $0A, $0B, $0B, $0C, $0D, $0E, $0E, $0F
        .byte $10, $11, $12, $13, $15, $16, $17, $19
        .byte $1A, $1C, $1D, $1F, $21, $23, $25, $27
        .byte $2A, $2C, $2F, $32, $35, $38, $3B, $3F
        .byte $43, $47, $4B, $4F, $54, $59, $5E, $64
        .byte $6A, $70, $77, $7E, $86, $8E, $96, $9F
        .byte $A8, $B3, $BD, $C8, $D4, $E1, $EE, $FD
```

[27]Rather than relying solely on statistics, this process involves mathematics. Within certain error limits, the program ensures that the values of frequency doubling every 12 notes are respected, evenly distributed across the octave.

[28]All the methods used, as well as their activation, are parameterizable through options. It's up to the user to determine whether the localization is indeed correct and useful.

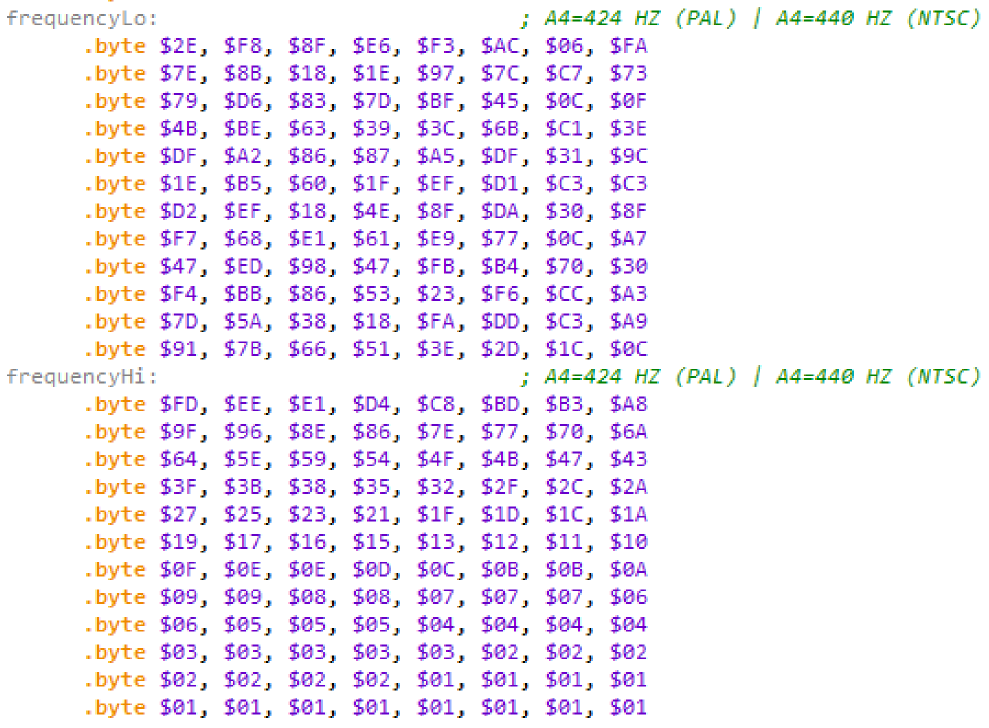
[29]The order is not random, but they are searched from the most probable to the least probable to reduce the margins of errors.

- **Combined table**: the 96 frequency notes are stored as 16-bit values with either little-endian or big-endian alignment, resulting in their contiguous placement in memory.



```
frequencyLo:                            ; A4=440 HZ (PAL) | A4=457 HZ (NTSC)
       .byte $16
frequencyHi:                            ; A4=440 HZ (PAL) | A4=457 HZ (NTSC)
       .byte $01

       .byte $27, $01, $38, $01, $4B, $01, $5F, $01
       .byte $73, $01, $8A, $01, $A1, $01, $BA, $01
       .byte $D4, $01, $F0, $01, $0E, $02, $2D, $02
       .byte $4E, $02, $71, $02, $96, $02, $BD, $02
       .byte $E7, $02, $13, $03, $42, $03, $74, $03
       .byte $A9, $03, $E0, $03, $1B, $04, $5A, $04
       .byte $9B, $04, $E2, $04, $2C, $05, $7B, $05
       .byte $CE, $05, $27, $06, $85, $06, $E8, $06
       .byte $51, $07, $C1, $07, $37, $08, $B4, $08
       .byte $37, $09, $C4, $09, $57, $0A, $F5, $0A
       .byte $9C, $0B, $4E, $0C, $09, $0D, $D0, $0D
       .byte $A3, $0E, $82, $0F, $6E, $10, $68, $11
       .byte $6E, $12, $88, $13, $AF, $14, $EB, $15
       .byte $39, $17, $9C, $18, $13, $1A, $A1, $1B
       .byte $46, $1D, $04, $1F, $DC, $20, $D0, $22
       .byte $DC, $24, $10, $27, $5E, $29, $D6, $2B
       .byte $72, $2E, $38, $31, $26, $34, $42, $37
       .byte $8C, $3A, $08, $3E, $B8, $41, $A0, $45
       .byte $B8, $49, $20, $4E, $BC, $52, $AC, $57
       .byte $E4, $5C, $70, $62, $4C, $68, $84, $6E
       .byte $18, $75, $10, $7C, $70, $83, $40, $8B
       .byte $70, $93, $40, $9C, $78, $A5, $58, $AF
       .byte $C8, $B9, $E0, $C4, $98, $D0, $08, $DD
       .byte $30, $EA, $20, $F8, $2E, $FD
```

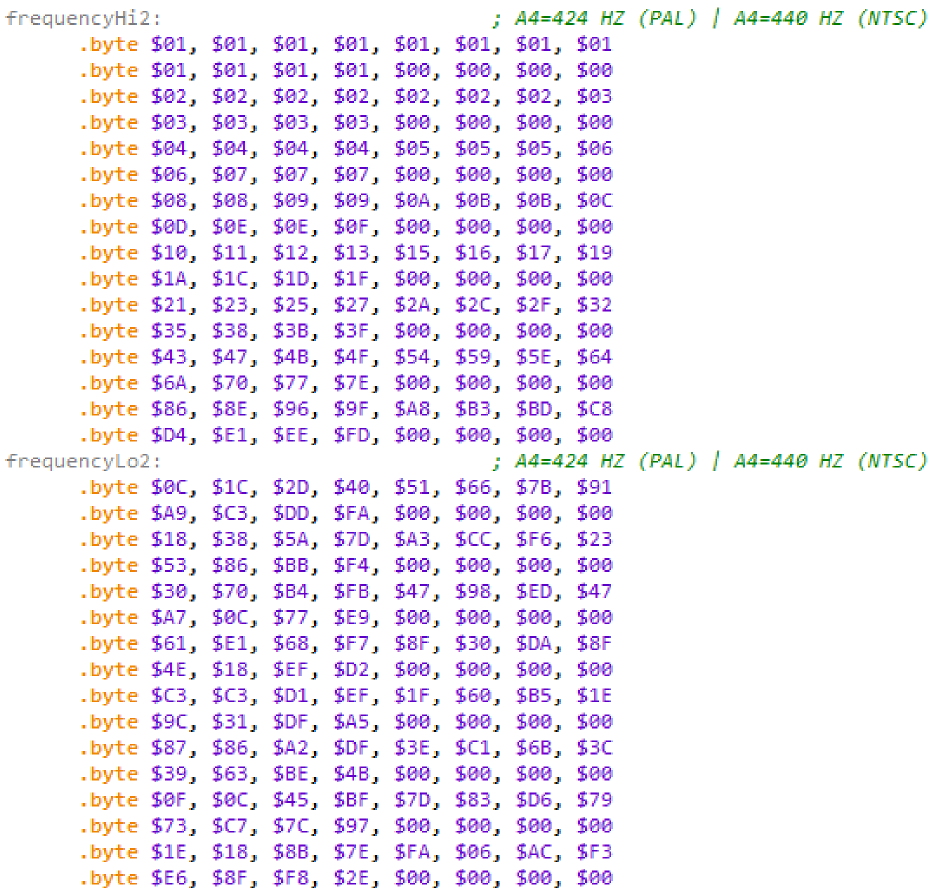
- **Inverse linear table**: this method is similar to the linear list approach, except that the notes start from the highest frequency and proceed to the lowest, effectively appearing inverted.



```
frequencyLo:                            ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
       .byte $2E, $F8, $8F, $E6, $F3, $AC, $06, $FA
       .byte $7E, $8B, $18, $1E, $97, $7C, $C7, $73
       .byte $79, $D6, $83, $7D, $BF, $45, $0C, $0F
       .byte $4B, $BE, $63, $39, $3C, $6B, $C1, $3E
       .byte $DF, $A2, $86, $87, $A5, $DF, $31, $9C
       .byte $1E, $B5, $60, $1F, $EF, $D1, $C3, $C3
       .byte $D2, $EF, $18, $4E, $8F, $DA, $30, $8F
       .byte $F7, $68, $E1, $61, $E9, $77, $0C, $A7
       .byte $47, $ED, $98, $47, $FB, $B4, $70, $30
       .byte $F4, $BB, $86, $53, $23, $F6, $CC, $A3
       .byte $7D, $5A, $38, $18, $FA, $DD, $C3, $A9
       .byte $91, $7B, $66, $51, $3E, $2D, $1C, $0C
frequencyHi:                            ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
       .byte $FD, $EE, $E1, $D4, $C8, $BD, $B3, $A8
       .byte $9F, $96, $8E, $86, $7E, $77, $70, $6A
       .byte $64, $5E, $59, $54, $4F, $4B, $47, $43
       .byte $3F, $3B, $38, $35, $32, $2F, $2C, $2A
       .byte $27, $25, $23, $21, $1F, $1D, $1C, $1A
       .byte $19, $17, $16, $15, $13, $12, $11, $10
       .byte $0F, $0E, $0E, $0D, $0C, $0B, $0B, $0A
       .byte $09, $09, $08, $08, $07, $07, $07, $06
       .byte $06, $05, $05, $05, $04, $04, $04, $04
       .byte $03, $03, $03, $03, $03, $02, $02, $02
       .byte $02, $02, $02, $02, $01, $01, $01, $01
       .byte $01, $01, $01, $01, $01, $01, $01, $01
```

- **Linear octave/note table**: conceptually, the notes are grouped octave/note-wise as nibbles within a byte. Since there are 12 notes, 16 values are used: the first 12 represent notes, while the remaining 4 are zeros.

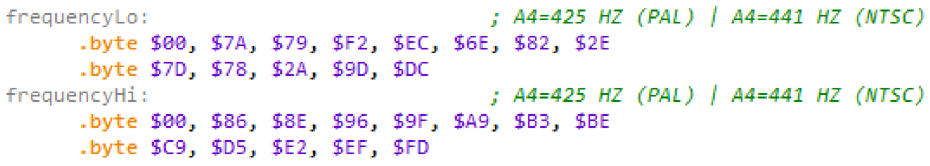
  These values are split across two tables, containing the low and high parts of the frequency value. The tables can be scattered in memory, with one preceding the other or vice versa.



```
frequencyHi2:                               ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $01, $01, $01, $01, $01, $01, $01, $01
        .byte $01, $01, $01, $01, $00, $00, $00, $00
        .byte $02, $02, $02, $02, $02, $02, $02, $03
        .byte $03, $03, $03, $03, $00, $00, $00, $00
        .byte $04, $04, $04, $04, $05, $05, $05, $06
        .byte $06, $07, $07, $07, $00, $00, $00, $00
        .byte $08, $08, $09, $09, $0A, $0B, $0B, $0C
        .byte $0D, $0E, $0E, $0F, $00, $00, $00, $00
        .byte $10, $11, $12, $13, $15, $16, $17, $19
        .byte $1A, $1C, $1D, $1F, $00, $00, $00, $00
        .byte $21, $23, $25, $27, $2A, $2C, $2F, $32
        .byte $35, $38, $3B, $3F, $00, $00, $00, $00
        .byte $43, $47, $4B, $4F, $54, $59, $5E, $64
        .byte $6A, $70, $77, $7E, $00, $00, $00, $00
        .byte $86, $8E, $96, $9F, $A8, $B3, $BD, $C8
        .byte $D4, $E1, $EE, $FD, $00, $00, $00, $00
frequencyLo2:                               ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $0C, $1C, $2D, $40, $51, $66, $7B, $91
        .byte $A9, $C3, $DD, $FA, $00, $00, $00, $00
        .byte $18, $38, $5A, $7D, $A3, $CC, $F6, $23
        .byte $53, $86, $BB, $F4, $00, $00, $00, $00
        .byte $30, $70, $B4, $FB, $47, $98, $ED, $47
        .byte $A7, $0C, $77, $E9, $00, $00, $00, $00
        .byte $61, $E1, $68, $F7, $8F, $30, $DA, $8F
        .byte $4E, $18, $EF, $D2, $00, $00, $00, $00
        .byte $C3, $C3, $D1, $EF, $1F, $60, $B5, $1E
        .byte $9C, $31, $DF, $A5, $00, $00, $00, $00
        .byte $87, $86, $A2, $DF, $3E, $C1, $6B, $3C
        .byte $39, $63, $BE, $4B, $00, $00, $00, $00
        .byte $0F, $0C, $45, $BF, $7D, $83, $D6, $79
        .byte $73, $C7, $7C, $97, $00, $00, $00, $00
        .byte $1E, $18, $8B, $7E, $FA, $06, $AC, $F3
        .byte $E6, $8F, $F8, $2E, $00, $00, $00, $00
```

- **High octave (13) notes**: only the frequencies of the last octave are reported (as the others can be derived by appropriately dividing their values).

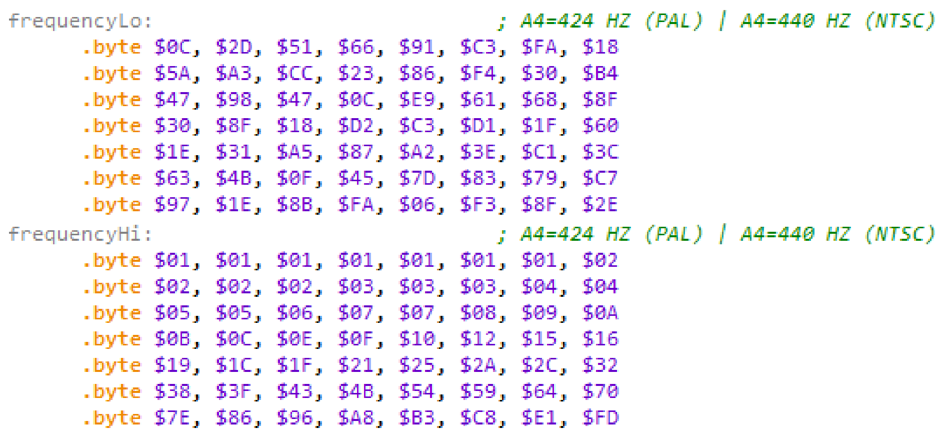
  These frequencies are presented in two tables as the low and high parts of the frequency, with the first note being a pause (no sound). Consequently, the table consists of 13 compressed words.



```
frequencyLo:                                ; A4=425 HZ (PAL) | A4=441 HZ (NTSC)
        .byte $00, $7A, $79, $F2, $EC, $6E, $82, $2E
        .byte $7D, $78, $2A, $9D, $DC
frequencyHi:                                ; A4=425 HZ (PAL) | A4=441 HZ (NTSC)
        .byte $00, $86, $8E, $96, $9F, $A9, $B3, $BE
        .byte $C9, $D5, $E2, $EF, $FD
```

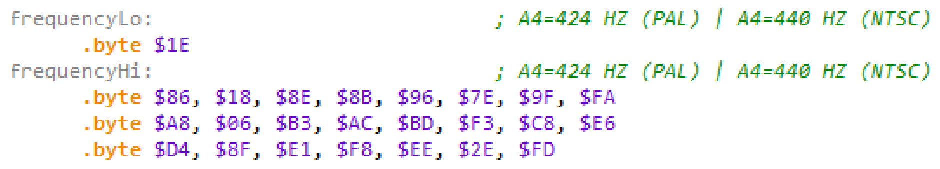
- **Linear scale (CDEFGAB) table**: in total, 56 notes are listed, divided into two tables for the high and low parts (spatially distributed in memory).

  These correspond to the 7 notes (across 8 octaves) of a C major scale.



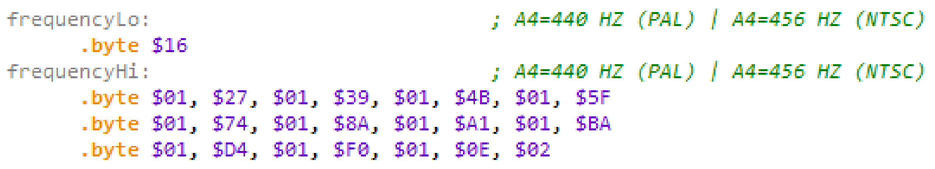
```
frequencyLo:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $0C, $2D, $51, $66, $91, $C3, $FA, $18
        .byte $5A, $A3, $CC, $23, $86, $F4, $30, $B4
        .byte $47, $98, $47, $0C, $E9, $61, $68, $8F
        .byte $30, $8F, $18, $D2, $C3, $D1, $1F, $60
        .byte $1E, $31, $A5, $87, $A2, $3E, $C1, $3C
        .byte $63, $4B, $0F, $45, $7D, $83, $79, $C7
        .byte $97, $1E, $8B, $FA, $06, $F3, $8F, $2E
frequencyHi:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $01, $01, $01, $01, $01, $01, $01, $02
        .byte $02, $02, $02, $03, $03, $03, $04, $04
        .byte $05, $05, $06, $07, $07, $08, $09, $0A
        .byte $0B, $0C, $0E, $0F, $10, $12, $15, $16
        .byte $19, $1C, $1F, $21, $25, $2A, $2C, $32
        .byte $38, $3F, $43, $4B, $54, $59, $64, $70
        .byte $7E, $86, $96, $A8, $B3, $C8, $E1, $FD
```

- **Short linear table**: as "Linear Table" but limited to first 7[30] octaves.
- **Short combined table**: as "Combined Table" but limited to first 7 octaves.
- **High octave combined table**: only the high octave is present, but with combined table of low/high pair of values.



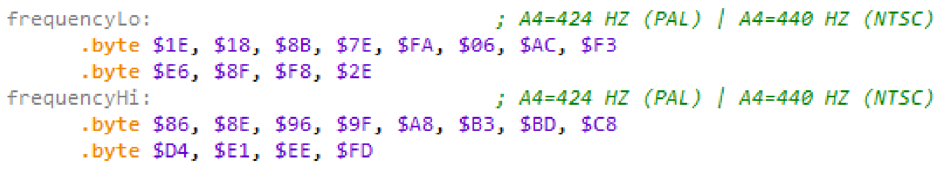
```
frequencyLo:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $1E
frequencyHi:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $86, $18, $8E, $8B, $96, $7E, $9F, $FA
        .byte $A8, $06, $B3, $AC, $BD, $F3, $C8, $E6
        .byte $D4, $8F, $E1, $F8, $EE, $2E, $FD
```

- **High octave combined inverted table**: as the previous, but with inverted value (so from high to low frequencies).
- **Low octave combined table**: only the first octave is present, but with combined table of low/high pair of values.



```
frequencyLo:                                ; A4=440 HZ (PAL) | A4=456 HZ (NTSC)
        .byte $16
frequencyHi:                                ; A4=440 HZ (PAL) | A4=456 HZ (NTSC)
        .byte $01, $27, $01, $39, $01, $4B, $01, $5F
        .byte $01, $74, $01, $8A, $01, $A1, $01, $BA
        .byte $01, $D4, $01, $F0, $01, $0E, $02
```

- **High octave (12) table**: this is like the high octave (13) table, but iwthout the rest (empty) note at beginning, so with 24 values.



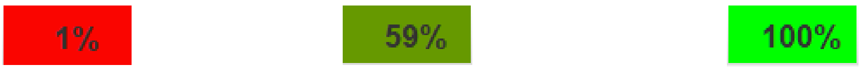
```
frequencyLo:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $1E, $18, $8B, $7E, $FA, $06, $AC, $F3
        .byte $E6, $8F, $F8, $2E
frequencyHi:                                ; A4=424 HZ (PAL) | A4=440 HZ (NTSC)
        .byte $86, $8E, $96, $9F, $A8, $B3, $BD, $C8
        .byte $D4, $E1, $EE, $FD
```

[30] Actually, in many players, the last notes are sometimes not encoded because, if the choice of A4 frequency to use is high, it exceeds the limit of 65535 for the chip's word, so the check is set to be less strict about the correctness of the values towards the end of the table. If other algorithms fail, this further reduced search (within 7 octaves) should still ensure finding a table.

18 Percentage and Freeze

Since the program serves to recreate the original source of a binary file, *JC64dis* proposes a virtual semaphore that indicates the degree of “understanding” we have managed to recreate with it:

- A 0% percentage, represented by red, means that we have not yet understood anything.
- A 50% percentage, represented by pastel yellow, indicates that we have already outlined the source.
- A 100% percentage, represented by green, signifies that everything is understood down to the finest details.



This percentage is derived by calculating how many labels we have renamed (thus giving them their true meaning in the source) compared to the total existing in the source. This is because labels like “W1234” provide no information, while a name like “playFlag” already identifies its function.

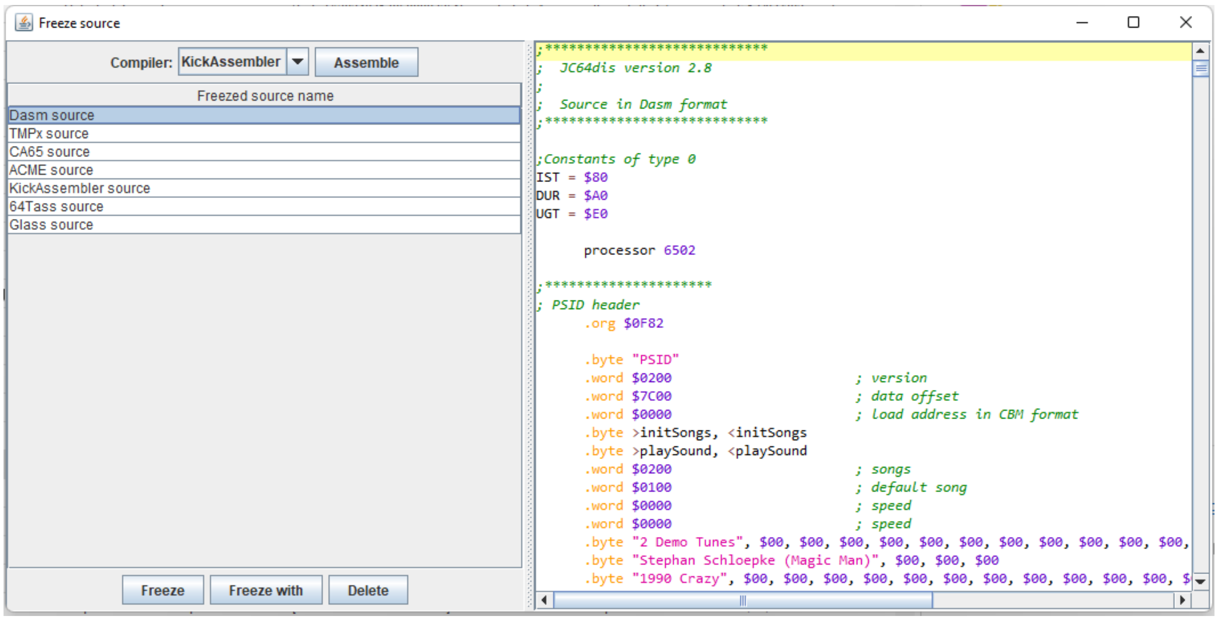
Another unique feature in JC64dis is freeze, which, as the name suggests, is used to store a snapshot of the current recreated source code.

Its usage can serve two purposes:

- Saving the work’s progress state (e.g., for a 15% step percentage) to document how we arrived at the final source.
- Presenting the final source correctly.

The latter point is not trivial. As we have seen, JC64dis has over 330 options that influence how the final result is generated. Therefore, if we open a file created by another user, we may find the source formatted aesthetically poorly because our options are entirely different.

However, by opening the source within freeze, we will find it in the correct (original) format.



```
;****************************
;  JC64dis version 2.8
;
;  Source in Dasm format
;****************************

;Constants of type 0
IST = $80
DUR = $A0
UGT = $E0

      processor 6502

;********************
; PSID header
      .org $0F82

      .byte "PSID"
      .word $0200                     ; version
      .word $7C00                     ; data offset
      .word $0000                     ; load address in CBM format
      .byte >initSongs, <initSongs
      .byte >playSound, <playSound
      .word $0200                     ; songs
      .word $0100                     ; default song
      .word $0000                     ; speed
      .word $0000                     ; speed
      .byte "2 Demo Tunes", $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00,
      .byte "Stephan Schloepke (Magic Man)", $00, $00, $00
      .byte "1990 Crazy", $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $
```