MikuASM User’s Handbook

# Introduction

MikuASM is an assembly-like programming language for DSC development, implemented in the eponymous compiler as well as the MeekStudio development environment.

This guide shows how to use MikuASM or MeekStudio for creation of DSC bytecode data and integrating with other tools in your chart building workflow.

# Understanding the DSC interpreter

*Notice: this was written upon FT and other versions might differ.*

The DSC interpreter inside the game behaves fairly much like an instance of a Turing machine — similar to a real computer, albeit a very limited one. Imagine the DSC file as a very long “tape” of commands and data, where every Int32 (4 bytes) is either of those. The interpreter reads them one by one, checks with it’s internals about what it should do, and then proceeds to read further until it reaches the end or is told to stop by an appropriate command.



Fig. 1 — Commands being read out from a stream of DSC data

For more detail about Turing machines, computers in general and bytecode interpreters, please consult a Computer Science guide or course of your choice, as it gets severely out of scope for this guide.

Right now, it’s important to establish the key characteristics of our DSC interpreter as a machine. First of all, it has three registers more or less accessible to our program — one that holds the current time (let’s call it *rTime*), one that holds the branch we’re executing (*rCurBranch*), and one that holds the current branch that’s being read from the script (*rBranch*). They can control the flow of our script by delaying execution until a certain timestamp, or skipping commands outside of the current branch.

In addition, we know that the current time is constantly rising from 0, where 0 is when our script starts executing.

The current branch register can only have two values — 1, which is when the user misses the success note, and 2, when the user has hit it. The executed branch however can also be 0, which is considered “common branch”.

From that knowledge we can find out the 3 only possible flow control commands: **END**, **TIME** and **PV\_BRANCH\_MODE**.

The **TIME** command will halt the interpreter until it’s time register is more than or equal to the command’s parameter. The **PV\_BRANCH\_MODE** will skip all of the commands, except for **TIME**, as long as the argument is *not equal* to the interpreter’s branch register and not 0 (which effectively means “no matter what branch”). The **END** command will stop the engine and shut down the interpreter, as if the song has ended.

Judging from that, we can finally determine the basic machine cycle of the DSC interpreter.



Fig. 2 — DSC engine machine cycle

This flowchart shows the basic logic for every tick of the DSC interpreter.

# MikuASM as a DSC Generator

MikuASM builds upon the concept of the aforementioned “tape of commands”. Essentially, MikuASM should probably be considered a meta-programming language, as issuing any command such as **TIME** or **MIKU\_MOVE** only adds it onto the current “tape”, or, as we shall call it, current buffer. After building up the desired DSC on the buffer, you can ask MikuASM to output the content of the buffer into a binary file, generating DSC data for further use in the game.

Consider the following example program:

TIME 10

MIKU\_ROT 0, 120

TIME 20

MIKU\_ROT 0, 240

TIME 30

END

If you input this program into MikuASM, then these commands will end up on the current buffer. However, the DSC file is not being generated. How come?

Essentially, by inputting this program into MikuASM, you only told it what commands you *have*, but never *what you want to do* with them. Imagine coming up to your friend and saying “fish cake”. Your friend understands what a fish cake is, but doesn’t get your point anyway. Do you want to eat some? Or do you want to make some?

This is where the *processor directives* come into play. And oh boy there is a ton of them in MikuASM. Let’s go over them one by one.

## MikuASM Data Formats

Before we begin, let’s settle with the data formats used in MikuASM. First and foremost, all TIMEs can be written in the following forms:

* Milliseconds, unlike other editors operating in microseconds, because such precision was deemed impractical. Example: TIME 1500
* Fractional seconds. Example: TIME 1.5
* Fractional time. Example: TIME 00:01.5
* Frames, at about 59.8 FPS, not really useful, but anyway. Example: F1234.

Any decimal values are written through a dot being the decimal point, example: 0.783.

Boolean values, for example in MIKU\_DISP directives, can be written both as binary (example: 0 or 1) as well as boolean (example: True or False).

Certain values which have an internal scale, such as CLOTH\_WET wetness values, can also be written as decimal parts (example: 0.5) or percentage (example: 50%).

## MikuASM Command Shortcuts

Currently there is just one command shortcut available, which is “@” for “TIME”. This means in any situation, for example “@ 1500” and “TIME 1500” are equivalent.

This shortcut was added because TIME is the most used command and is also very different from all others, because it controls the flow of the script.

## MikuASM Processor Directives

Processor directives are commands that are not supposed to end up inside the DSC, but rather, to be completed by the MikuASM compiler instead. In order to tell MikuASM it’s a command for it and not something to go onto the buffer, we start them all with a *#* mark.

### #unko

This is the simplest one you could possibly imagine. What it does is force MikuASM to produce a syntax error and stop. This could be useful when you don’t want someone to compile your unfinished script by accident or blindly compile it without changing something important inside.

### #include path/to.file

Now it’s getting a bit harder. You might guess that any relatively large DSC written in MikuASM will take up hundreds, if not thousands of lines. Navigating such a huge file will be a major pain, and you might get lost pretty quickly.

What *include* does is pretty obvious — it replaces itself with the contents of the file you provided as the parameter. Consider you have a file called “rotating\_miku.mia” with the content:

TIME 10

MIKU\_ROT 0, 120

TIME 20

MIKU\_ROT 0, 240

And a file “my\_script.mia” with the content:

**#include rotating\_miku.mia**

TIME 30

END

Let’s try to do what *include* does on our own here, in this case — replace the first line with what we have in our *rotating\_miku* script. We end up with this:

TIME 10

MIKU\_ROT 0, 120

TIME 20

MIKU\_ROT 0, 240

TIME 30

END

Which is what will effectively be interpreted by MikuASM. Pretty simple, huh?

### #incbin path/to.dsc

Alright, we have learned how to combine multiple text files into a single one. But what if we want to read out the commands from an already existing DSC file? Say, to alter how a PV works, or use a chart prepared in another editor like UPDC or Comfy. MikuASM got you covered with the *incbin* command, which is absolutely the same as *include* — but for DSC files instead of source code. How cool is that?



### #binflt FILTERMODE

So now that we can also use a DSC as part of our source code, we obviously want to only use just some pieces of it, so as not to make a mess.

For that we can use the *binflt* command. It tells MikuASM: all of the DSCs imported from now on should be treated with this or that mode. Valid modes are:

* *NO\_FILTER*: includes the DSC data as-is without processing
* *ONLY\_CHART*: includes only the TIME, END and TARGET commands from the DSC data
* *WITHOUT\_CHART*: includes everything but the END and TARGET commands from the DSC data

Of course, this is a simplified explanation, but those modes can be very useful when using a DSC of a chart prepared in an external editor.

### #clear

Removes all of the DSC commands currently on the buffer. This may be useful for some advanced use-cases.

### #const name=definition string

The *const* command pretty much defines a keyword to be used as a replacement in all of the lines encountered further. Think of it as an advanced Find-Replace kind of thing.

Consider the following example program:

#const EndOfSongT=1500

TIME EndOfSongT

END

After processing the *const* command, MikuASM will reach the **TIME** line and do a find-replace upon it, according to your definition, essentially transforming your program into:

TIME 1500

END

### #undef name

Reverts the action of *const*. If you don’t need a definition anymore or want to change it, you can delete it using the *undef* command.

### #sort!

This is, probably, the killer feature of MikuASM which makes it so powerful. Remember how we mentioned the **TIME** command before and how it works? That means, if you try to write the **TIME** commands out of order, e.g.

TIME 30

LYRIC 2 // “world!”

TIME 10

LYRIC 1 // “Hello”

The DSC interpreter will first wait until the time is 30 and display lyric #2. Then, because 30 is already past 10, it will show lyric #1 instantly, resulting in a mess. If your source code is text-only, it’s easy to fix by just writing in order, and even then it’s a pain in the neck once the source is big enough. And if your source contains a DSC binary through an *incbin*, what do?

This is the case when *sort* comes into play. It will split your current buffer into chunks marked with timestamps and branches, and then reposition them with the correct order.



### #ctxstart/#ctxend

These two are basically a sophisticated undo/redo kind of thing. The *ctxstart* command will save the current buffer and *ctxend* will restore the last saved one. This could be useful when combining the same project into multiple difficulty chart DSCs, like so:

#include my\_lyrics.mia

#binflt ONLY\_CHART

// easy chart

#ctxstart

#incbin easy\_chart.dsc

#sort!

#write easy.dsc

#ctxend

// normal chart

#ctxstart

#incbin normal\_chart.dsc

#sort!

#write normal.dsc

#ctxend

In this example, the same lyric commands from my\_lyrics.mia will be applied into both EASY and NORMAL charts.

### #write path/to.dsc

We’ve seen it in the example before, so what does it do? Essentially, this tells MikuASM to dump the current command buffer into a DSC file. That file can then be used inside the game, assuming your code was valid and correct.

### #bintime STARTTIME ENDTIME

This is another thing similar to *binflt* that affects the *incbin* command. It will tell *incbin* to only take the commands that are after **TIME** STARTTIME and before **TIME** ENDTIME. Pretty much a way to chop up a DSC without any external tools.

### #slidetime OFFSET

Offsets all of the TIMEs inside the current buffer by a specified amount. This may be useful when testing a script if you want to get right to a certain point in the project.

### #for PARAM START END STEP PARAM… / #endfor

Repeats the commands between those two directives while changing the variable PARAM from START to END in steps of STEP. Useful when you’re trying to animate something that’s not really animatable by the DSC engine. Consider the following example:

#for RTime 0 200 100 RVal 0 60 30

TIME RTime

MIKU\_ROT 0, RVal

#endfor

Once compiled, this essentially turns into:

TIME 0

MIKU\_ROT 0, 0

TIME 100

MIKU\_ROT 0, 30

TIME 200

MIKU\_ROT 0, 60

But saves you the headache of calculating and writing all this manually.

# Using the MikuASM Compiler

The MikuASM Compiler (mikuasm.exe or dscedit.exe, depending on the version) is a command-line utility that interprets MikuASM commands. It is the simplest implementation of MikuASM which you can use inside your build environment of choice, such as bat-files or Makefile.

In order to compile a MikuASM source file, just provide it as the argument, like so: *mikuasm.exe my\_source.mia*. The provided source file will be considered user input, and all *include* or *incbin* paths within it will be considered relative to the source file’s location.

To display more detailed compilation logs, you can use the -v flag, e.g. *mikuasm.exe -v my\_source.mia*.

If you would like to use MikuASM Compiler as an interactive help tool, use *mikuasm.exe --interactive*. That will bring up an interactive prompt which will respond to all commands as if they were input inside a source file. You can also start typing a command and hit TAB to see suggestions.

If you would like to use MikuASM Compiler as a REPL to see how commands work in real time, set the game on the song list screen, then execute *mikuasm.exe --interactive --attach <gameprocessname.exe>* and run the song of your choice. The chart will not appear and the PV will not animate. Inputting any DSC command into the interactive prompt in this mode will execute it inside the game right away, so you can see how it affects the PV display.

For more detail, execute *mikuasm.exe --help* to see other available flags.

# Using the MeekStudio IDE

MeekStudio IDE is the more user-friendly implementation of MikuASM inside a development environment for Windows.

## Interface overview

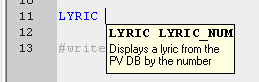
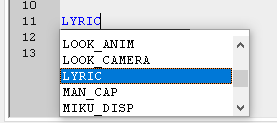
Most of the work in MeekStudio is done inside the main window as shown below.



You can use keyboard shortcuts, the toolbar or menu bar to operate MeekStudio.

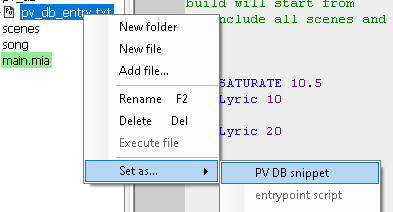


When possible, command suggestions and hints will be displayed in the source code editor.



## Setting up a project

After creating a project, you need to tell MeekStudio where some of your project files are located. Some will be preconfigured, others you will have to select on your own. This is done by right-clicking the file in the Project Hierarchy view, then choosing “Set As” and the appropriate file disposition.



The following dispositions need to be set in order to fully use the functionality of MeekStudio:

* PV DB snippet — the PV DB text file containing **only the song you are currently working on** inside the project. This shall be used in all of the debug and preview modes.
* Entrypoint Script — the MIA source code file containing the beginning of your program. This would be what you’d usually pass as the argument to mikuasm.exe as user input.
* Movie Output — the DSC file produced by your project that contains no chart. This shall be used when using the “Boot and Play Movie” function to preview your PV.
* Movie Audio — the OGG file to be used as your project’s audio. This shall be used when using the “Boot and Play Movie” function to preview your PV.

### Default Project Hierarchy

When a project is created in MeekStudio, it already comes with a few files and folders. This is not a strict requirement, but just a recommended way of organizing your project.

* Charts — the directory for you to place your chart DSCs prepared externally.
* Output — the directory to place the resulting DSC files from building your PV.
* PV\_DB — the directory for your PV DB snippets.
* Scenes — the directory for scene code that you’ll include later into your entrypoint script.
* Song — the directory for your audio files.
* Main.mia — the entrypoint source code file.

## Interactive Debugging Features

MeekStudio includes a variety of interactive features to assist in developing PV Scripts. Prior to using any of them, specify the vanilla game executable path inside a clean installation by selecting Debugger → Set game exe path…

**NB: Acquiring a legal, working copy of the game is the user’s responsibility.**

### Connecting to Game Process

In order to use interactive features, select the appropriate connection mode from the Debugger menu or the toolbar.

* **Boot Game to Debug**: starts up the game software into a new debug session with the current project’s PV DB entry. External plugins are not loaded in this mode, thus any kind of manual control of the game interface is not supported. *This is the recommended debug mode*.
* **Attach to Game…**: attaches to an existing game process. Set the game on the song list screen before selecting this item. Once invoked, select the song you are debugging. The PV will not animate, the chart won’t play, and interactive features will become available.
* **Reattach by UDP**: use this if MeekStudio crashed or was accidentally closed, and an instance of the game inside a debug state is still running.
* **Boot and Play Movie**: this will build the project and start up the game to play the resulting PV. Other interactive functions are not available in this mode. Close the game software after invoking this mode to continue using MeekStudio.

### Evaluation and Editing

Once MeekStudio is connected to the game software and has entered the interactive mode, you can use evaluation commands to execute code and see how it affects the PV scene or display.

To run a whole source file, right-click on an entry inside the *Project Hierarchy* view and select *Evaluate File*. This will execute all of the commands inside the selected file, except for TIME and PV\_BRANCH\_MODE.

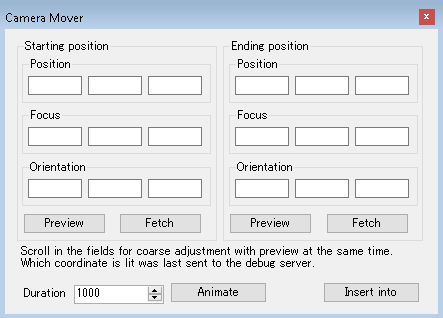
The following commands are also available from the Debugger menu for quick execution of code:

* **Evaluate current line**: send the line where the text cursor currently resides to the game and execute it.
* **Evaluate until current line**: send all of the commands in the lines above the one where the text cursor currently resides.
* **Evaluate current file**: execute all of the commands inside the active code editor (same as *Evaluate File* in *Project Hierarchy*).

### Moving and Animating the Camera

Camera motion and framing is key to most of the PV production. In order to bring up the camera motion tool, select *Debugger → Interactive → Camera move wizard* or press F10.

In case the text cursor is currently on a *MOVE\_CAMERA* command, coordinates from the command will be loaded into the tool. Otherwise, if the tool was closed, coordinates from the game will be loaded. If the tool was just in the background, the coordinates are not changed.



All coordinate fields are aligned as X, Y, Z. *Position* and *Focus* designate the camera location within the PV field, and the point it’s looking at respectively. *Orientation* designates a normal vector for the camera rotation. The *Starting Position* frame encloses the coordinates where the animation will begin, and *Ending Position* where the animation will end.

Pressing *Preview* in either of the frames will set the camera inside the game software to the respective coordinates instantly. Pressing *Fetch* will retrieve the current coordinates from the game software into the respective position frame.

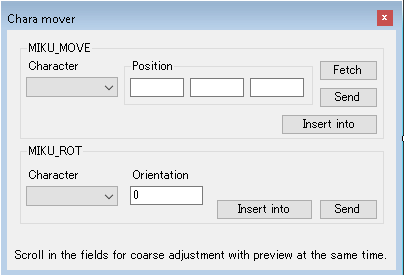
You can also enter the duration (in milliseconds) into the *Duration* field and press *Animate* to get a rough preview of the camera motion. When you’re done, press *Insert into script* to add the resulting animation command into the currently active code editor.

To move the camera easier, you can hover any of the coordinate fields with the mouse pointer and use the scroll wheel to move around.

### Moving and Rotating the Mikus

In order to move the characters, you can use the Chara Mover tool. Select *Debugger → Interactive → Chara move wizard* or press F11 to bring it up.

If your cursor is currently over a *MIKU\_MOVE* or *MIKU\_ROT* command, respective coordinates will be loaded into the tool. Otherwise, if the tool was not open, coordinates from the game software will be retrieved. If the tool was just in the background, coordinates are not changed when bringing up the tool.



The *Position* coordinates are laid out as X, Y, Z. Select the character number in the *Character* dropdown. Clicking *Fetch* will retrieve that character’s position from the game software.

The *Orientation* is specified in degrees.

Clicking *Send* will move that character to the entered coordinates or alter the character’s rotation. When you’re done, click *Insert into script* to add the resulting command into the active code editor.

# Examples of MikuASM in Practical Applications

## Adding lyrics to a chart

Currently, no publicly available chart editor has a lyric editing capability. MikuASM can be used to easily add lyrics into your chart.

First, place your audio data into the *song* folder of the project, and chart DSC into the *charts* folder.

Then, adjust the PV DB snippet text file to include your lyrics.

Create a new file called *lyrics.mia* in the project root and specify your lyrics numbers in the format like shown below:

@ 0:00.000

LYRIC 1 // sekai de ichiban ohimesama

@ 0:05.849

LYRIC 2 // ki ga tsuite nee nee

@ 0:09.288

LYRIC 3 // mataseru nante rongai yo

@ 0:12.089

LYRIC 4 // watasi wo dare da to omotteru no?

@ 0:17.771

LYRIC 5 // mou nandaka amai mono ga tabetai

@ 0:24.983

LYRIC 6 // ima sugu ni yo

@ 0:30.000

LYRIC 7 // ... continued

Finally, open *main.mia* and replace the contents with the following code:

@ 0

PV\_BRANCH\_MODE 0

/// In the beginning, load a chart DSC (FT) created in UPDC

#binflt NO\_FILTER

#incbin charts/common\_with\_success\_note.dsc

/// Then add your lyrics

#include lyrics.mia

/// Mix everything together

#sort!

/// And you're done!

#write output/test.dsc

In the end, press F5 in MeekStudio or invoke *mikuasm -v main.mia* to build your project. The resulting *output/test.dsc* file will have lyrics integrated into the chart at the correct timings.

### How does that work?

When you compile this project, *main.mia* is being read. First 2 operators tell MikuASM to load *charts/common\_with\_success\_note.dsc*, which you created in UPDC earlier, without any filtering onto the current buffer.

The next line *include*s *lyrics.mia*, effectively adding a sequence of *TIME* and *LYRIC* commands with your lyric timing onto the same buffer.

Right after it the *sort* operator tells MikuASM to sort all of the timings inside the current buffer and place them in the correct order.

Finally, the *write* operator asks MikuASM to output the new buffer, which includes both the lyric timings and the chart, and is in the correct order, into the *output/test.dsc* file. Simple as that!

## Adding a Success Event to your chart

Currently all of the available editors support creating Success Notes (the ones with a rainbow frame around them), however, none support altering the script flow just yet. MikuASM can be used to change the flow depending on the branch, for example, to display a visual effect on top of the PV.

First, place your audio data into the *song* folder of the project, and chart DSC created in UPDC into the *charts* folder. The chart must have a success note inside it, at no later than around 2s.

Then, adjust the PV DB snippet text file to include the effect definitions:

pv\_420.edit=1

pv\_420.edit\_chara\_scale=1

# Effect 007: Hearts

pv\_420.edit\_effect.00=007

# Effect 004: Burning

pv\_420.edit\_effect.01=004

As well as the different lyrics for the purpose of this tutorial:

pv\_420.lyric.001=Success Event Test

pv\_420.lyric.002=You FAIL!!

pv\_420.lyric.003=You Succeed!!

Create the file *consts.mia* in the project root and add the following code inside it:

#const BRANCH\_COMMON=0

#const BRANCH\_SUCCESS=2

#const BRANCH\_FAILURE=1

#const BRANCH\_SPLIT\_TIME=3250

Then replace the contents of *main.mia* with the following code:

// Include constants definitions

#include consts.mia

// Set binary import to only allow chart related commands

#binflt ONLY\_CHART

// The first command cannot be a branch definition it seems

// So let's put a useless command in there

@ 0

CHANGE\_FIELD 1

/// Define the common branch (before success event)

PV\_BRANCH\_MODE BRANCH\_COMMON

// You must ALWAYS load the chart at the 0 time

@ 0

/// Show the lyric "Success Event Test"

LYRIC 1

/// Leave just one Meek

MIKU\_DISP 1, false

MIKU\_DISP 2, false

/// Set camera position

MOVE\_CAMERA 0, -200, 5920, 3160, 0, 6240, 0, 0, 1000, 0, -200, 5920, 3160, 0, 6240, 0, 0, 1000, 0, -1, -1

/// Load the actual chart including a success event

/// The chart was made with UPDC and exported as FT DSC

#incbin charts/common\_with\_success\_note.dsc

/// Define the failure branch (after success event)

// For ease of use I split at the same point here, using a

// constant for the time value specified in consts.mia

PV\_BRANCH\_MODE BRANCH\_FAILURE

@ BRANCH\_SPLIT\_TIME

/// Display the burning effect

EDIT\_EFFECT 1, 1000

/// Show the lyric "You fail!!"

LYRIC 2

/// Change face

SET\_CHARA 0

EDIT\_EXPRESSION 22, 100

EDIT\_MOUTH 4

/// Define the success branch (after success event)

PV\_BRANCH\_MODE BRANCH\_SUCCESS

@ BRANCH\_SPLIT\_TIME

/// Change to another stage for success

CHANGE\_FIELD 2

/// Show hearts effect

EDIT\_EFFECT 0, 1000

/// Show the lyric "You succeed!!"

LYRIC 3

/// Change face

SET\_CHARA 0

EDIT\_EXPRESSION 1, 100

EDIT\_MOUTH 23

/// Align all timings

#sort!

/// Write the final output chart file

#write output/test.dsc

Build the project and then play *output/test.dsc* as part of your extended data. The Miku’s face and displayed visual effects and lyrics will change depending if you hit the success note or not.

### How does that work?

First of all, we have created *consts.mia*. It only contains *const* definitions purely for convenience — because who’s gonna remember what 0 or 1 or 3250 means in a year or two‽

Then inside of *main.mia* we include that file to use the definitions inside our code.

After that we configure *binflt* to only load chart-related commands from the external DSC files.

Inside the *COMMON* branch we add a few commands: one that shows the first lyric from PV DB, another two that remove the extraneous Mikus from the stage, then move the camera in front of Miku’s face up close, and finally, load the chart from UPDC.

Despite the chart probably overflowing in time over the further commands residing inside other branches, since we have loaded it while being in *COMMON*, even after *sort*ing all of the chart commands — and thus, targets and things — stay common.

After that, at a certain point in time, specified by the *BRANCH\_SPLIT\_TIME* constant, we add a bunch of commands, but this time different commands for different branches. Under *FAILURE* we show a burning visual effect, show lyric #2 (which is “You Fail!!”) and set Miku’s face expression to be something weird.

However under *SUCCESS* we show a hearts effect, lyric #3 (“You Succeed!!”) and change Miku’s face to a smile.

In the end we execute *sort* to tell MikuASM to move the commands around into the right timing order. MikuASM also keeps track of which command was in what branch, so everything written as *COMMON* stays inside *COMMON* even if it has to be moved around, and so forth.

Finally we once again ask MikuASM to write the current buffer, which contains all of our commands and branches, into the *output/test.dsc* file, which produces a chart with a flow differing upon whether the success note was hit or not.