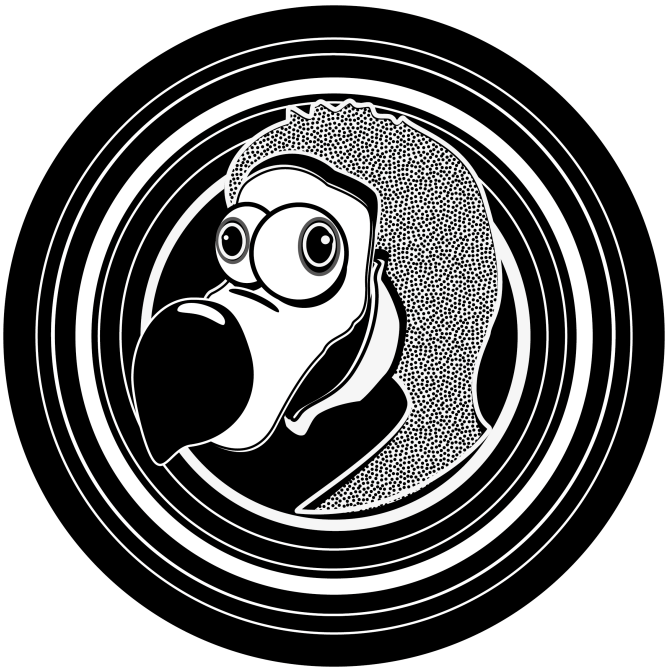
Dodo v 0.9



A BASIC Engine for Text Adventures on the Commodore C 64

Elmar Vogt
https://dodoif.wordpress.com

August 26, 2016

Contents

Part I.

Manual

1. Introduction

1.1. Overview: The Concept

Dodo is an engine for writing and running text adventures, or »interactive fiction«, a kind of computer games which consists of textual descriptions of a player's environment, to which the player responds with commands entered via the command line. This response will be analyzed by a parser, and evaluated by an engine which is basically a simulation of the game environment, the »world« expericed by the player, which may be realistic or fantasy-oriented.[1]

Most text adventures are written with an authoring system, which can either come in the shape of a programming language like Inform 6/7 or TADS, or as a graphic configurator like ADRIFT.[2]. The resulting scenarios are compiled and then played with a separate interpreter/engine. (By far the most popular example of this is the »Z-machine«[3].)

Dodo goes down a different lane. Dodo is a program written in the C 64's[4] notorious BASIC V2.0, a programming language which only timidly raises its head above assembler.[5]. As is, the program provides a framework and engine which can prompt for user input, parse the commands, and executing them in a reasonably straightforward manner to arrive at standard responses.

To arrive at a complete game, an author must extend the »naked« Dodo program with two more components:

- A `DATA` section which contains the definitions of the scenario – names and descriptions of rooms and items as well as their respective properties; connections between rooms, verbs beyond the standard vocabulary, doors, keys, etc.
- BASIC code sections which describe the particular non-standard responses to particular player actions, which are »out of the ordinary«.

For example, while the ordinary response of the program to the command »`TAKE BOTTLE`« would be a simple »`TAKEN`«, it would be boring if *only* those default reactions

[1] The term »text adventures« has fallen out of use lately, probably since nowadays all kinds of RPGs are called »adventures«. I'll stick with it, though, since that is the term that was used during the C 64's heydays. But »interactive fiction« is where Dodo's web address got it's »if« from, in case you wondered.

[2] `http://inform7.com/`, `http://www.tads.org/`, `http://www.adrift.co/`

[3] `https://en.wikipedia.org/wiki/Z-machine`

[4] I'm aware that in most english-speaking regions this machine is better known as the »CBM 64«, but »C 64« or »Commodore 64« was the term I grew up with, and thus will stick with.

[5] Unless explicitly noted, all further mentions of »BASIC« will refer to the Commodore BASIC V2.0

would happen. If, for example, you want to make »`TAKE JACKINTHEBOX`« result in the box opening and a contained item flying out of it, you would have to provide entries into the `DATA` section which describe the combination of the verb »`TAKE`« and the item »`JACKINTHEBOX`« (plus a few more lines), along with a code section which results in the box opening, and the item inside flying around.

In other words, while most adventure authoring systems consist of a compiler, a scenario file written by the author, and an engine which interprets the compiled scenario, in Dodo all three components come rolled into one, for better or worse.

Note:	As you probably have gathered, you will have to have a working knowledge of the C 64's »BASIC V2.0« to be able to do anything with Dodo. If you're a novice to that, check out the C 64 Programmer's Reference Guide to see if it's for you.

1.2. Limitations

Due to the limitations of the C 64, the resulting engine is also fairly rudimentary. Execution speed and memory mandate a »bare bones« machine, if you want to be left with enough calculating power and RAM to develop a scenario worth while.[6] For example, the parser is extremely simple (to avoid the word »dumb«), and will accept as input only strings with a maximum of three words – a verb, a subject, and an object. Thus, »`Put the silver coin in the red box`« will have to be reduced to »`put coin box`« by the player.[7]

On the other hand, Dodo itself imposes very few limitations, and I think any configuration the C 64 can reasonably be burdened with will not be foiled by arbitrary limitations of the engine.

Half the fun of working with Dodo – beside the nostalgia and geek factor – lies in getting around these limitations. If you want to write more complex scenarios, you're probably better off with one of the more powerful, modern authoring systems (see above).

1.3. But, why the name?

Text adventures are for the better part obsolete. BASIC is mostly obsolete. The C 64 has *definitely* been obsolete for more than 30 years.

The Dodo is an obsolete bird.

What better name would there be . . . ?

[6] Not to mention the fact that the C 64's BASIC doesn't exactly render itself to large programming projects.

[7] No capitalization there either, BTW.

2. Dodo: The Works

2.1. Notes about Wording

- Dodo is the *engine*, or framework to write games in.
- A particular game (ie the combination of the BASIC program, `DATA` section and code written by you) is called a *scenario*, and it is put together by the *author*. (That is *you*.)
- Someone actually playing the game is unsurprisingly referred to as the *player*, and he plays a scenario within one or more *sessions*.
- The term *object* is always used in the grammar sense – ie, a thing upon which a verb acts, or more concretely, the third word in a command line. Actual things which are parts of a scenario are generally referred to as *items*.

2.2. Making it Run

2.2.1. Downloading Dodo

Download the Dodo archive from the website, unzip it in a convenient location, and you're set.

2.2.2. Creating, Distributing, and Running Scenarios with Dodo

Note:	The recommended way to work with Dodo is installing the *CBM prg studio* on your machine. It is a full IDE for assembler and BASIC programs for the various flavours of Commodore 8-bit machines, and the tool I programmed Dodo with. Start it, open the project file »`DODO.CBMPRJ`« with it, and you're set.

Alternatively, you may want to use a different IDE (see 2.2.2.1). Open »`DODO.BAS`« in that IDE, and start from there to develop your own scenarios.

In theory, you can also edit `DODO.BAS` with any text editor and feed it to the emulator of your choice (see 2.2.2.2), and finally you can also edit Dodo *inside* the editor with the C 64's own full-screen editor – if you feel really adventurous.

Here are a few more tools which may help you with Dodo:

2.2.2.1. *C 64 studio*: Another IDE for the C 64

One alternative to the *CBM prg studio* is the (unfortunately very similarly named) *C 64 studio*[1]. Like the *CBM prg studio*, it is very powerful, well-integrated with Commodore emulators, reasonably bug-free, and maintained by very active and responsive developers.

2.2.2.2. Emulators

I'm not sure how many people out there still have a functioning C 64, and of course it would be the ultimate in geekdom to run DODO scenarios on one of those.[2]

For those less lucky off, you will have to use one of the emulators out there which do a very good job in re-creating the C 64 experience. Personally I use *VICE*, which is a superb piece of software, able to run on a number of platforms, even on Android. VICE integrates nicely with the *CBM prg studio*. It also allows you to »customize« the emulated C 64 down to the monitor picture output.

Another popular option is *CCS64*, though personally I haven't worked with it. Several other emulators are available, you even have the option to use an online emulator[3], if you don't want to install anything on your machine.

Finally, there is also *CBMBASIC*[4], which is the original BASIC interpreter ported to various platforms. It can be invoked from Windows in a DOS box, for example. Since the »hardware part« of the emulation is missing, you lose all graphics and sound capabilities (not much of a loss for a text adventure system), and while you can enjoy the full execution speed of your PC, you're still stuck with »38911 bytes free«, due to the C 64's 8-bit system. Unfortunately, this interpreter seems to be no longer maintained.

Note: I haven't tried the *CBMBASIC* option yet.

2.2.3. Programming Hints

»Out-of-the-box«, DODO will leave you with approx. 25 kByte of memory, and the execution of a player's standard move will take from 2 to 3 seconds in a small scenario. Developing your own scenario you can quickly run into memory limitations and less-than-acceptable execution times.[5] To alleviate your pain, here are a few tips which help *all* BASIC programs to run faster and with less RAM on the C 64:

- Use as few code *lines* as possible, but rather put as many statements into each line as is reasonable. This is not so much a problem during step-by-step execution of

[1] http://www.georg-rottensteiner.de/de/C~64.html

[2] Actually, if you really own such a machine and have DODO running on it, I'd *love* to see a foto of this! Pretty please . . . ?

[3] http://www.64er-online.de/emulator/

[4] https://github.com/mist64/cbmbasic

[5] Okay, this is a 64 kByte machine with a 1 MHz CPU!

a program, but jumps to a different line require the interpreter to search line by line through the whole program for the target line.

- Since this search always starts at the beginning, often-used code sequences should rest at the beginning of the program. (Dodo leaves you with a fairly limited choice here.)
- Likewise, often-used variables should be declared early during program execution (not necessarily at the physical program start), because variables in the lookup table will be entered in the sequence in which they were encountered. (Dodo tries to help here.)
- Whitespace and the contents of `REM` instructions are meaningless to the interpreter, but add to the memory and time requirements of the program. (IDE's like CBM prg studio give you the option to automatically strip them from your code upon compilation, which comes very handy.)
- Integer variables are converted to floats for *all* operations and then reconverted before storage, so they come with a performance penalty. *Individual* integers also take up as much space as a float, so there is no reason to use them at all. Integer *arrays* OTOH require less space per member than a float, you may save some space here, sacrificing speed in return.

A note particular to Dodo: The engine will allow you to set limits to the number of rooms, verbs, etc., minimizing the load for the C 64. See 4.1.1 for details.

2.3. How Scenarios Work

Dodo starts with initialization of the game engine and the scenario. This is followed by turns, which involve cycles of player command input/parsing/processing. At the end of each cycle, Dodo tests for scenario end conditions.

2.3.1. Initialization of the Engine

This is the sequence of events for starting up Dodo:

1. Upon starting the Dodo program, line number 9000 is called to perform a number of initialization steps.
2. A number of often-used variables are declared to make sure they appear at the top of Dodo's variable list, and are quickly found when the interpreter searches for them.
3. Array boundaries are set to a default value, then line 20000 is called. This section is used to override the defaults, and allocate only as much memory as is required for the scenario. See 4.1.1.

4. The various scenario arrays are set up.
5. Line 14000 is called to make sure all user-defined functions are known to the interpreter.
6. The names of the standard directions (»up«, »north«, and so on) are read.
7. Read standard and user-defined verbs, see 6.3,
8. Read item configuration, see 6.4.
9. Read user hooks, see 6.5.
10. Call line 21000 (see 4.1.2) to set up demons and fuses, and to write an initial banner

After this, control is passed on to line 1000 for the processing of regular turns.

2.3.2. Processing Moves: Input, Parsing, and Stages

See also appendix 3.

DODO is strictly turn-oriented, which means that each input of the player leads to a parse-evaluate-respond sequence on the part of the engine. This also means that there is no option to introduce real-time components into a scenario.[6]

The player input string »co$« is then parsed into a maximum of three words, separated by spaces: A *command verb* »v$« (first word on the line), a *subject* »s$« (second word), and an *object* »o$« (third word). Subject and object are optional, leading and trailing spaces are ignored.

Importantly, immediately after that DODO tests whether the conditions for »darkness« are met (see 2.4.3). If so, the execution of a number of verbs is blocked.

Note:	This happens *before* any user hooks are evaluted which could override the standard behaviour.

Then, DODO checks whether any conditions for user hooks are met (see 2.4.8) and executes the respective hooks.

Unless the variable »ok« is set, the standard processes for verbs are subsequently executed (see 2.5).

Next, demons and fuses (see 2.4.9) are processed, before finally Line 22000 is called to test for game-end conditions (see 4.1.3).

Then, move results are printed, and DODO prompts the player for the next command.

Note:	Each stage is processed if and only if the previous stages didn't result in an abort condition, ie while »ok« is FALSE.

[6] The reason for this is the fact that the player input is read through an INPUT command which allows no interruption between the point it's started and the endpoint. A character-by-character read with the chance of interruption through a timer would be conceivable in the future.

2.4. Scenario Concepts

2.4.1. Items and their Attributes

With DODO, all items are created equal, and there is no fundamental distinction between rooms, ordinary items and NPC's.[7]

To each item, an attribute string »`at$()`« is assigned. Each letter in this attribute string flags the presence or absence of a certain attribute for this item. For example, if `at$(i)` contains the letter »`d`«, then it is considered to be a door, if it contains a »`c`«, it is a container in which other items can be stored. For a full list of attribute codes, see 5.

The attributes of an item are determined at startup by the respective string read from the `DATA` section. At any later time, the string can be manipulated to raise or clear the individual attribute flags:

- To set attribute flag »`x`« with item `i`, simply add the letter to the attribute string: `at$(i)=at$(i)+"x"`
- To check whether an attribute flag is raised, call line 10000 (see 4.3.1)
- To clear a flag, call line 10100 (see 4.3.2 for details)

You can introduce more flags than the ones described in the appendix to your heart's delight and use them for your own purposes.

2.4.2. Location and »Presence«

Location

The location of any item is encoded in its »`lc%()`« variable: if `lc%(i)=10`, then item `i` is at location 10: 10 is the index of either a room or a container. »`lc`« (without an array index) is the location of the player.

There are two special values for locations:

»`lp`«: (»location player«, which is set to -1) A location value of `lp` means »in possession of the player«. Note that this is *not* the same thing as being in the same room as the player (»`lc`«).

0: Items with location zero are considered to be »nowhere«, ie not currently taking part in the game. An eaten item will for example receive location zero. You can »revive« an item from this state of limbo by giving its location a different value, but in general items with location 0 are disregarded by DODO.

You may assign a location to a room, too, but this is meaningless: Rooms don't *have* locations, they *are* locations. And it doesn't really make sense to place a room for example in a container or in the player's possession, unless you want to make for a *really* esoteric piece of gameplay.

[7] »Non-player characters«. Actually, currently there is absolutely no support for NPC's at all.

Presence

An item's location naturally determines whether it is » present« within the player's scope – ie whether the player can `TAKE`, `DROP` or `EXAMINE` the item, and whether the item can affect him. In general, interaction is possible when the item is either in the same room as the player, *or* in his possession, ie `lc%(i)=lp`.

Note:	This has consequences for items which are carried within containers which are held by the player. Assume you put `SAUSAGE` in `BOX` and `TAKE BOX`. If you now try to `EXAMINE SAUSAGE`, you will be told off: »`You don't have a sausage.`« To be able to do anything with the sausage again, you must `TAKE SAUSAGE BOX`, ie take the sausage from the box. For more detail, refer to 2.4.4

For some scenarios, this makes sense, for some it may come across as a little odd. For example, if the player has a cigarette in a box of cigarettes, why should he not be able to smoke one of them, when he so desires?

On the other hand, look at a more complicated example – namely a lockable wooden box inside which are chocolates. Clearly, the player should be able to eat the chocolates if and only if the box is unlocked, or locked while the player holds the key. In the latter case, eating the chocolate would also entail unlocking the box and possibly leaving it in an unlocked state afterwards with all its consequences. Hence, the player must go through the individual steps, rather than having immediate access to a container's contents.

2.4.3. The Darkness

If the variable »`d`« is set to `TRUE`, the current location is considered »dark«.

Per default, all rooms are lit. But if the attribute »`D`« (note the capitalization!) for the room in which the player currently is, is set, then the room is supposed to be »dark«. This severely reduces his options, and leaves only a handful of actions possible:

- He still can move like before. (This isn't very realistic, but is the only chance he has to get out of a dark room if he's stepped into it.)
- Verbs with indices > 17 can still be attempted.

The latter group are verbs which are mostly »passive« (like `WAIT`) or refer to the game status, but they also include all scenario-defined particular verbs.[8]

The darkness can be »overridden« by any item in scope (see 2.4.2) which has the attribute »`*`« set. Such items are supposed to emit light (like a torch or a flashlight)

[8] It may or may not be the author's intent that these particular actions can still be employed, so it's your responsibility to write appropriate behaviour for them.

lighting an otherwise dark room. (Which is why the variable `d` is not equivalent to the `D` attribute of a room.)

Note:	This only refers to items in the same room or »immediately held« by the player. If the torch is inside a box held by the player, it won't be able to penetrate the darkness, even if the box is transparent or open. Again, that was a compromise between realism and the C 64's calculating power.

The routine at line 10400 ff (see 4.3.5) determines the state of light or darkness in the room.

2.4.4. Containers and Containing

The standard concept of locations is »flat«, ie, each locations holds a number of items, and that's about it. But, of course, things get much more interesting when containers come into play, and items can be held within other items and you develop a »tree« of items.

In DODO, this is relatively easy to do, because there is no general distinction between rooms and regular items – all of them do have an index, and if you give an item's location `lc%()` the value `i`, then that item is at location `i` (ie the room with index `i`).

DODO also supports a number of mechanisms to provide a more realistic behaviour of things. This is regulated with the attributes »`c`« (»container«), »`o`« (»openable/closable«), »`O`« (»open«), »`l`« (»lockable«, ie can be locked) and »`L`« (»is locked«)

2.4.4.1. Containers

Attributes »`c`«, »`o`« and »`O`« provide complementing features an item must have for you to be able to place anything else inside it (or take from it): The item must be a container, and it must be openable, and openend.[9]

`OPEN`ing and `CLOSE`ing will set and clear the item's »`O`« flag, respectively. `PUT` provided with both a subject and an object (ie »`PUT BANANA BOX`« rather than simply »`PUT BANANA`«) will place the subject item inside the object item.[10]

`EXAMINE`ing or taking an `INVENTORY` of an item will provide information about any other item contained inside it, provided the container is open or transparent (»`t`« attribute is set). This only works for one level: Assume that `BOX` holds `BOTTLE` inside which is the item `MESSAGE`. Then »`EXAMINE BOX`« will only report about the bottle, but not about the message. `EXAMINE BOTTLE` will display information about the message, though.[11]

[9] The concepts of »container« and »openable« may look redundant at first, but they are not: It is conceivable that a thing is openable but unable to contain anything – a book, for example. Likewise it could be a container, but it's impossible to close it, like a bowl.

[10] `PUT` is simply an alias for `DROP`, so using `PUT` without an object will result in the subject item being dropped to the floor of the current room.

[11] This was done for performance reasons.

2.4.4.2. Locking and Unlocking

Items with the attribute »`l`« can be locked (and unlocked), provided the player has the fitting key – keys are items with the »`k`« attribute set.

The value of a key's `sp%()` »special« variable determines which item can be locked/unlocked with that particular key. If `BOX`' item number is 12, then `SILVERKEY`'s `sp%()` must be set to 12 to enable it to lock or unlock `BOX` with it. Thus it's possible that several keys may be used to lock or unlock a specific item, but any one key is useful only with one particular item.

In »regular use«, the idea would be to make an item lockable, openable and a container, and then provide a key to lock or unlock it. That would provide the behaviour one would expect for a box or a closet, but other arrangements are conceivable: Medieval books and modern journals, for example, are known to have locks to make it impossible for the uninitiated to read them, and they can be opened, but nothing can be put inside. Motorbikes or other devices may have a key used to start their ignition, or otherwise arm or activate them (a bomb's timer?), while it wouldn't make sense to »open« them.[12]

2.4.5. Rooms, Movements, and Exits

Any room (this includes places outdoors) is supposed to have a roughly cubic shape with six faces. These six faces are oriented towards north, east, south, west, up, and down, respectively, and these directions are assigned the numbers 1 ... 6. For any room `i` possible exits correspond to the values in the array `ex%(i, k)`, `k` being the direction: Any value `ex%(i, k)` other than 0 will be considered a valid exit in direction `k` to the room with that index.[13]

Exits are defined only for rooms, ie for items which have the attribute flag »`r`« set. Although it might make sense to step into containers like closets or coffins, DODO doesn't currently provide for that.

There is no check that the exits of the rooms mutually match. In other words, it's the author's responsibility to ensure that, if room `i` has an exit to the east leading to room `j`, room `j` will have a corresponding exit to the west leading back to room `i`. This is a design decision, since it may be the author's intent to create »one-way« doors (like chutes or trapdoors), or a confusing geometry.

2.4.6. Doors

Doors provide a mechanism for a more dynamic way of dealing with exits.

Doors are regular items placed in rooms, and have their location »`lc%()`« set to the room index. They need to have both their »`d`« and »`o`« attribute set to be recognized as doors, so that the player can open and close the door.[14]

[12] In this case it would probably be best to provide synonyms for `LOCK` and `UNLOCK`, like `IGNITE/ARM` and `STOP/DISARM` etc., depending on context.

[13] Resist the temptation of setting any of the exits to -1.

[14] Opening and closing the door will actually »create« and »delete« the room's exits. It is up to you whether you make the door »lockable« with a key, or not.

Furthermore, like rooms, doors also require a set of six »`ex%(i, k)`« exits,[15] and a »`sp%(i)`« »special« value in their `DATA` section.

Note:	I must check out whether there is a possible bug, namely, that you can place anything »inside« a door – after all, it is a container . . .

In general, doors perform innocuously like any other item,[16] unless you `open` or `close` them. When you do, the door's set of `ex%()` values will overwrite the exit values of the room it's in, for all exits which are unequal to zero.[17]

So, let's assume we have for a door and a room we have sets of exits like this:

```
(dir:)  n  e  s  w  u  p
DOOR:   0  0  0 10  0  0
ROOM:   5  4  0  0  0  0
```

We see, per default the room has exits to the north and the east only.

After `open door`, the configuration for `room` will be:

```
ROOM:   5 4 0 10 0 0 0
```

The room now has an additional exit to the west the player can now employ by simply calling »`west`« without explicitly referring to the door.

Once you `close` the door all the door exits which are unequal to 0 will be set to 0 for the room.[18]

This behaviour is not yet realistic, because usually a door connects *two* rooms rather than just manipulating a single room's exits. This is where the door's `sp%()` variable comes into play: `sp%()` is supposed to point to a *second* door item which also contains a set of `ex%()` exits. Upon opening the first door, two things will happen:

- The exits of the room of the opened (first) door will be created,
- Also the exits of the room in which the second door sits will be created.

This may require a bit of explanation. Let's take the following scenario with two door which have the indices »`10`« and »`11`«, respectively:

```
                n  e  s  w  u  d
lc%(10)= 5
ex%(10, i)= 0, 0, 9, 0, 0, 0
sp%(10)= 11

lc%(11)= 9
ex%(11, i)= 5, 0, 0, 0, 0, 0
```

[15] Confusingly, a DODO door can seemingly lead to *several* rooms – but please read on!

[16] As Graham Nelson noted in his standard work on the *Inform* authoring system, it's a good idea to always make doors non-portable.

[17] Hence, to make the door's behaviour more intuitive, only *one* of the `ex%()` values should be <> 0.

[18] This holds also true for exits which were <> 0 *before* the door was opened.

```
sp%(11)= 10
```

Let's also look at the exits of the rooms in which the doors are located, namely »5« and »9«:

```
                n  e  s  w  u  d
ex%(5, i)=      0, 6, 0, 0, 0, 0
ex%(19, i)= 0, 0, 0, 7, 0, 0
```

Thus, in the beginning room 5 would have an exit to the east, leading to room 6, and room 19 has a single exit to the west to room 7.

Now, let's open door 5 (or 11 for that purpose). This is the exit configuration of both rooms afterwards (the exits of the doors never change):

```
                n  e  s  w  u  d
ex%(5, i)=      0, 6, 9, 0, 0, 0
ex%(19, i)= 5, 0, 0, 7, 0, 0
```

In addition to the previous exits, room 5 now also has an exit to the south to room 9, while room 9 has an exit to the north back to room 5 – as you would intuitively expect.

This technique allows you to configure either regular »pairs of doors«, or a »single door« behaviour (like a trapdoor), which will not automatically provide a way back.

Since all item names need to be unique, the two connected doors must have different names. Thus, while the two doors are to the player for all intents and purposes one and the same thing, he will have to address them by different names. This is a bit unfortunate.

2.4.7. »Extended« Verbs

Per default, DODO understands about two dozen different verbs, with a maximum of three synonyms each. You can extend the vocabulary by writing a `data` section which contains the names of any additional verbs (see 6.3), and a code section which contains the BASIC code to be executed whenever the player enters the verb.

All of this is pretty simple: Add the indices (24 or higher) and names for your extended verbs in the program between line 15240 (the last default verb, `STATUS`) and line 20000. Make sure that you finish the `data` section with a single entry of -1. (This is for DODO to understand that the verb section ends here.)

Subsequently, the extended verbs will be treated indistinguishably from the default verbs.

The code associated with the verbs – standard or extended – starts at line 4000, moving up in increments of 100 with the verb index. Thus, if your new verb »`flabberghast`« has index 35, the code section to deal with the default response will start at line $4000 + 35 * 100 = 7500$. You must be finished with your code before line 7600 (to avoid interfering with the subsequent verb). The code section must consists at a minimum of

a `RETURN` command and should also end with a `RETURN` to avoid fallthrough to the next verb.

Note:	If you want to change the behaviour of standard verbs, this is simple to do as well: The line numbers for their code follow the same rule, and you can simply alter the BASIC there. But due to the slightly arcane nature of Commodore's BASIC, this should be done with dilligence.

2.4.8. Hooks

The mechanism of »hooks« provides the author with a means to configure non-standard behaviour in a scenario: Whenever a certain combination of conditions becomes true, the code asssigned to a hook will be performed, replacing or supplementing the regular game responses. This works with both standard and extended verbs.

DODO can handle up to 16 different hooks.[19] Their actual number is stored in the variable »`nh`«, the default is 8. Much like verbs, the configuration for each hook consists of –

- a `data` section, which contains the conditions under which the hook is engaged (see also 6.5), and
- the code section which is performed when the hook is triggered.

Upon scenario startup, the `data` section for all hooks is fed into the four arrays »`hv()`«, »`hs()`«, »`ho()`«, and »`hl()`«, which contain the indices for the verb, subject, object and location, respectively, connected with the hook.

During gameplay, after each command entered by the player, the parser will process all hooks *before* executing any other actions.

Each hook can individually be activated and deactivated. Internally, this is done by setting and clearing the 16 bits in the variable »`ho%`«), one bit per hook. For hook number `i`, its state is determined through the function »`fn hh(i)`« which will return `TRUE`, if the hook is »live«, or active.

If this is the case, then in the next step the parser will see if the hook is »triggered«, ie if the verb, subject and object of the current command all match the hook definition.[20] Finally the »hook location« is matched with the current location.[21]

[19] Though I'm under the vague feeling that some more would be required for any useful game.

[20] Or, more precisely, if the *indices* of the words match. This is important in the case of a verb with several synonyms. For example, verb 11 has the synonyms »`D`«, »`DROP`« and »`PUT`«. Regardless which of the three verbs the player entered, it will always be parsed as »verb 11«, and the hook will correspondingly be triggered if `hv(i)` equals 11, and not otherwise. In other words, in this example it's impossible for the hook to be triggered by the command »`DROP`«, but not by »`PUT`«.

[21] It may be tempting to try to use `hl(i)=-1`, ie the »player location« for a hook, meaning »trigger the hook whenever the subject is in the playes's possession«. But this won't work since the current location `lc` is always the index of a room. Otherwise, the player would be »inside himself«, and the philosophical consequences of this were too deep for me to fathom.

Note:	Per default, hooks do *not* check for the presence of either subject or object: The will be triggered regardless of whether either of them is present.

Matches are determined by the `DATA` values read for the corresponding hook: If for example `ho%(i)` is 5, then hook `i` can be triggered only if the current object has an index of 5 as well. If `ho%(i)` is 0, then this hook can be triggered with any object, or even if none at all is given (and correspondingly with verbs/subjects/locations).

Note:	It's not directly possible to create more complex triggers, for example to realize an `OR`-condition. A workaround would be to define different hooks for different conditions, but to redirect them with `GOSUB`s to the same code section.

If all of the conditions given are met, then is the corresponding code sequence called: Similarly to verbs, hooks have separate procedures assigned. They start at line 40000 and increase in intervals of 1000 (to allow for more elaborate treatment of the hooks). Thus, for hook 7, code would begin at line $40000 + 7 * 1000 = 47000$. The procedure must not extend beyond line 48000, and must finish with a `RETURN` command.

If the hook's code does set the variable `ok` to `TRUE`, then the move is finished now. Otherwise, the regular verb responses will be processed subsequently.

See 6.5 for an example of the configuration of a hook.

Note:	Note that the code for the hooks is placed at the very end of your BASIC program with the highest line numbers. This means that their execution will be particularly slow.

2.4.9. Demons and Fuses

DODO provides two standard mechanisms to perform »automatic« actions outside the realm of standard responses: These are »demons« and »fuses«, respectively.

Demons

Demons are background actions which are performed *every* move, regardless of the player's actions. Demons can be used to keep track of the movement of a vehicle, or to allow actions of NPC's.

The total number of demons is stored in the variable »`nd`«. DODO will support a maximum of 16 demons, the default is 8. Any combination of demons can be active at any time. Similar to hooks, the variable »`de%`« stores information about the currently active demons: If demon `i` is active, then `de% AND 2^i` will be `TRUE`.[22] (Function `FN`

[22] The C 64's PETSCII character set didn't feature a caret »^«, but used an upward-pointing arrow »↑« instead. In this documentation we'll use both characters interchangeably.

`do(i)` will provide you with this information for demon `i`, see 4.2.3) Start or stop individual demons by setting or clearing their respective bits.

The code for the demons is expected to lie starting at lines 30000, growing in intervals of 100: The code for demon 4 would thus start at line 30400 and will be executed every move as long as `de% AND 2^4` is `TRUE`. It is the author's responsibility to provide the code, at minimum a `RETURN` instruction.

Fuses

Fuses are similar to demons, but are different insofar as they are only executed *once*, namely after a previously set number of moves. You may consider them as timers which, counting down, will trigger an action upon reaching 0.

The number of fuses can be determined at startup by setting »`nf`«, the default is 8. The state of the fuses is stored in the array »`fu%()`«. To set fuse `i` in motion, set `fu%(i)=x`, and after `x` moves, the corresponding action will be triggered. (The values for all `fu%()` entries are counted down each move, so you can easily read how much time is left before each fuse will go off.)

The code for the fuses is expected to lie starting at lines 32000, growing in intervals of 100: The code for fuse 4 would thus start at line 32400. It is the author's responsibility to provide the code, at minimum a lone (pointless) `RETURN` instruction.

Properties of Demons and Fuses

As opposed to hooks, which are employed *before* the regular verb processing takes place, both demons and fuses are evaluated *after* the regular verbs. This also means that setting the value of `ok` inside a demon/fuse has no effect on the move processing.

Any combination of demons can be active at one move, and any number of fuses may simultaneously »go off« within one move.

2.4.10. The »Special« Variable

For each item `i`, there is a »special« variable »`sp%(i)`«. This variable has a predefined function only for two classes of items:

- For doors, it holds the index of the »partner« door item. (See 2.4.6 for details about the working of doors in DODO).
- For keys, it holds the index of the item the key can `LOCK/UNLOCK`.

For all items which are neither doors nor keys, you're free to use the `sp%()` variable for your own purposes.[23]

[23] Oddly enough this stipulates that no item can succesfully be a door *and* a key at the same time.

2.5. Standard Processing of Verbs

This section explains how the standard verbs which are provided with DODO per default work, and what they do to items. The list is in alphabetical order with the exception of the movement verbs, which are grouped together in 2.5.1.

For the concept of »being present«, see 2.4.2.

- A mandatory parameter to the verb is indicated in square brackets »`[]`«. An optional parameter is given in curly braces »`{}`«: »`verb [mandatory] {optional}`«.
- The first parameter to the verb (and the second word on the command line) is called the »subject«, the second parameter (or third word) is the »object«.
- If there are possible alternative forms for a verb (eg »`sleep`« for »`wait`«), these are given in parentheses.
- Parentheses in the section header also give the index for the respective verb in the »`vb$()`« table.

To add your own verbs to DODO's vocabulary, see 2.4.7 and 6.3. To change the standard behaviour of any combination of verb/item/location, use the »hook« mechanism (see 2.4.8).

2.5.1. Movement: »`north`«, »`east`«, »`south`«, »`west`«, »`up`«, »`down`« (1 ... 6)

Upon one of these commands, DODO will check if the room you're located in has an exit in that direction.

For example, if the player location is `lc=12` and the command is »`east`« (with index 2), then DODO will consult the value of »`ex%(12, 2)`«: If it is `FALSE`, then the exit is blocked, otherwise the new player location is `lc= ex%(12, 2)`.

Items held by the player will automatically move along with him. (This is easily done, since the location of an item held in the inventory is not the index of the current room, but -1, ie the »player position« `lp`)

See also the section on rooms (2.4.5) and doors (2.4.6) for more information on movement and exits.

Note:	Movements are almost the only commands which succeed in a dark room with the »`D`« attribute set. (Verbs like `help` which don't change the game status, and scenario-defined verbs with indices above 23 also work per default in darkness). For more information about the handling of darkness, see 2.4.3.

2.5.2. »again« (»g«, 0)

Repeat the last move.

If `again` is chosen, DODO restores the last move's values for verb, subject and object, which are held in the variables `vo`, `so`, and `oo`, respectively, before performing the previous command once more.

2.5.3. »close [x]« (»c«, 13)

Will close the item `x` (clear flag »O«), provided –

- It's present,
- It's opened (flag »O« raised),
- It's openable (flag »o« raised).

If `x` is a door (»d« flag raised), then this will close all associated exits from the current room. (See 2.4.6 for details.)

`Close` will ignore the »L« flag, ie the item can still be closed even if it is locked in the opened position.

2.5.4. »down« (»d«, 6)

See 2.5.1.

2.5.5. »drop [x] {y}« (»d«, »put«, 11)

Drop the item »[x]« either to the current location, or, if »{y}« is given, put `x` into the container »{y}«,[24] provided `x` is in the player's possession (but not within a container held by the player).

If `y` is given as well, then –

- `y` must be present,
- `y` must not be locked (the »L« flag is cleared), and
- `y` must be a container (the »c« flag is set).

Note that, to make it quicker for the player, he doesn't have to explicitly open the recipient `y` to place `x` inside, as long as `y` is a reachable unlocked container – these actions are tacitly skipped.[25] This means that when putting `x` into `y`, any hooks possibly connected with `OPEN`ing or `CLOSE`ing `y` will *not* be triggered.

[24] The two operations – dropping something to the ground, or putting something away in a container – are very similar in terms of game logic, but hard to describe with the same verb, hence the dissimilar names of `drop` and `put`.

[25] Due to sloppy programming on my part, it is even possible to place something inside another item when this is a container, but not openable. OTOH, the scenario author could also be blamed for configuring a non-openable container, which makes little sense.
(Opposed to that, an openable item which is no container is conceivable, eg a book.)

2.5.6. »east« (»e«, 2)

See 2.5.1.

2.5.7. »eat [x]« (»drink«, 16)

If »x« is present and it is marked as »edible« (flag »e« is set), then it will be »eaten« or »drunk«.[26]

This has no further effects, except to remove x from the game (ie, assign its location `lc%(x)= 0`), but it can serve as trigger for a hook for special actions, to reduce »fatigue points« from the player or such.

2.5.8. »examine [x]« (»x«, 8)

This command returns the long description `lo$()` of x if

- it is present,
- or it *is* the room the player is in,
- and the invisibility flag »i« is *not* set.

If the item is a container (»c« flag raised) and open (»O« flag raised) and *not* locked (»L« flag is not raised), then DODO will in addition give the short name (»`nm$()`« of its contents.[27]

Warning:	Todo: Check for transparency flag!

Warning:	There is a bug in DODO (or, let's call it a »counter-intuitive behaviour«): While an item is inside a container which is held by the player, it's impossible to `examine` it. The player must take it out of the container first (eg, `take sausage box`, see 2.5.20).

2.5.9. »free« (»memory«, »ram«, 21)

Explicitly initiates a garbage collection (see 2.6 for details) to free up RAM by using the BASIC's `fre(0)` function.

The command may take from a second or two to several minutes to execute.

[26] In a trade-off between programming effort and simulation realism, DODO does not discriminate between the two actions.

[27] For performance reasons, like the other »descriptive commands«, this does not recurse: If there is a `box` with an `envelope` inside, inside which is a `letter`, then DODO will report the long description of `box`, followed by »`... Inside is the envelope`«, and stop there. The player will have to try »`examine envelope`« to find out about the letter.

2.5.10. »help« (»?«, 19)

Gives a list of *all* verbs understood by DODO. (This includes both the built-in vocabulary and scenario-defined verbs.)

No explanations are given.

2.5.11. »inventory« (»inv«, »i«, 9)

This command lists the names of all the items the player is currently carrying, ie »nm$(i)« for all i for which lc%(i)=lp.

If the item is transparent or an open container (flags »t« and/or »O« are raised), then DODO will also describe the contents, ie write *their* short description . See also footnote 27.

2.5.12. »lock [x] [y]« (14)

Will lock item »x« (ie, raise flag »L« and make it impossible to open it), using key »y«.

The following conditions apply for a successful operation:

- Both the item x and the key y are present,
- x is »lockable« (»l« flag set),
- y has the »key« attribute (»k«) set, and
- y's »special« variable sp%(y) is equal to x (see 2.4.10)

So, y must not just be *any* key, but it has to be a key fitting x.

It's not possible to lock (»chain«) two items to each other.

2.5.13. »look« (»l«, 7)

Look around to get a more detailled description of your surrounding. DODO gives you the following information, supposed you're in room x:

- The long description of the current room, as given in lo$(x),
- A list of all possible exits as given in ex%(x, 1 ... 6),
- A list of all items in the same room as you, provided their invisibility attribute »i« is *not* set. See also footnote 27.

2.5.14. »north« (»n«, 1)

See 2.5.1.

2.5.15. »open [x]« (»o«, 12)

Will open the item x (raise flag »0«), provided –

- It's present,
- It's not already open (flag »0« is clear),
- It's not locked (flag »L« clear), and
- It's openable (flag »o« raised).

If x is a door (»d« flag raised), then it will open all associated exits from the current room. (See 2.4.6 for details.)

2.5.16. »quit« (»exit«, »stop«, 22)

This terminates the current game session without prompting for a confirmation and will return control to the C 64 interpreter.

Note: There is currently no option to save or restore a session.

2.5.17. »read« (»s«, 17)

Currently, this verb is a non-functional dummy (DODO will always reply »There is nothing to read here.«), but it could conceivably be used with hooks to provide »reading matter« (like books, signs, etc.). In a pinch, it can simply be re-routed to the examine code.

2.5.18. »south« (»s«, 3)

See 2.5.1.

2.5.19. »status {x}« (23)

This command is mostly meant for scenario debugging purposes, and you may consider disabling it before you release your scenario to the public.

Status will display the following scenario information:

- Number of moves played, number of items defined,
- Status of all demons and fuses,

Note: Suspiciously, hooks are missing

- Free memory,[28]
- If x is given, then this additional information about the item is printed:
  - The index of x, it's name `nm$(x)`, »first« and »long« descriptions `fi$(x)` and `lo$(x), resp.`,
  - Its attribute string `at$(x)` and location `lc%(x)`,
  - All its exits `ex%(1 ... 6)`, [29] and
  - x's »special« variable `sp%(x)`.

2.5.20. »take [x] {y}« (»t«, 10)

Brings »x« from the current room, or, if »y« is given, from container »y«, into the player's possession, subject to the following conditions for object »x« and subject »y«:

- x is not already in the player's possession, ie `lc%(x)<>lp`,[30]
- x (and y) are present in the same room as the player,
- Neither x (nor y) are invisible, ie the »i« flag in their attributes is not set,
- y is a container, ie its »c« flag is set,[31]
- y is open, ie its »O« flag is set.[32]

2.5.21. »talk« (»speak«, »ask«, 20)

This verb is meant for future expansion when there is a chance to interact with animated NPC's.

Currently, it serves only to frustrate the player (»`You don't know what to say.`«).

2.5.22. »unlock [x] [y]« (15)

Will unlock item »x« with key »y« provided the same conditions as for `lock`ing an item (see 2.5.12) are given: As a result, the flag »L« will be cleared, and it is subsequently possible to `open` x again.

---

[28] As reported by the function `FN rf()`, see 4.2.1. Note that this value is calculated without resorting to BASIC'S `fre(0)` function, and hence does *not* trigger a garbage collection immediately.

[29] if x is neither a room or a door, these values aren't particularly meaningful

[30] But taking an item a from another item b will succeed.

[31] This is admittedly a bit superfluous, because if y wasn't a container, there could be no item x located inside it.

[32] While you can `put` an item inside a container without explicitly `open`ing it, the same doesn't hold true for `take`ing it. This is admittedly an inconsistency.

2.5.23. »up« (»u«, 5)

See 2.5.1.

2.5.24. »wait« (»z«, »sleep«, 18)

»Do nothing«. During this move, only activated demons and fuses will be executed (and possible hooks linked to `wait`).

2.5.25. »west« (»w«, 4)

See 2.5.1.

2.6. The Issue of Garbage Collection

As a computer with limited capacities, the C 64 had problems with the administration of strings of variable length. Since at the time of creation of a string variable it wasn't clear what the later maximum length of the string would be, the C 64 would always only assign the minimum space required for the string upon creating. When later the string was manipulated, the C 64 simply dropped the old space and assigned new space for the resulting string from the heap. Of course, this led to ever-new assignments within the heap, and a constant reduction of free space, even though the total length of assigned string space might remain constant.

When at one point a string assignment wouldn't work anymore for the lack of free space, the C 64 would – suddenly, and without warning or notice – start a »garbage collection«, by squeezing all the gaps out of the heap which remained from previous string reassignments. On the up side, this resulted in reclaiming all the unused space and freeing up the C 64's memory for work again, but on the downside, the garbage collection might kick in at any time during the C 64's operation, it could literally take hours, and there would be no indication to the user why the machine froze. Naturally, garbage collections would be required more often when working with string-heavy software – like text adventures, for example.

Garbage collections can't be avoided with BASIC programs. But to keep the effect minimal, Dodo constantly surveys the free memory and gives a warning when the free RAM drops below 2 kByte. At this point the user can *manually* at his leisure start off a garbage collection, and then go have a coffee.[33]

An instant garbage collection is initiated with the command `FREE`, see 2.5.9.

[33] The alternative to picking a convenient point in time is to initiate the garbage collection more often, while there is still plenty of space. In this case, the operation usually is over within a few seconds. Dodo had an implementation where the garbage collection would be invoked after every move, but this added on average more than a second to an already large processing time, so this was dropped again.

2.7. Putting it all together: A Brief Example

To maybe make things a little clearer, let's put together a small example. Say, you want to introduce an item called »rose« in your game. Whenever the player gets a whiff of its fragrance by performing `smell rose`, something like his »life points« are supposed to go up by a value of 10.[34]

2.7.1. Creating the Rose

First of all, create a »rose« item by adding it to your program's `DATA` section, somewhere after line 50000:

```
...
52000 DATA 99, "rose", "You see a beautiful red rose", "It does looks
    pretty,@"
52010 DATA "and it has thorns. But will it smell as wonderfully...?"
52020 DATA "",20,10
...
```

Which means you have defined an item named »`rose`« with item number 99 and a proper »first« description (used when you first encounter it – »`You see a beautiful red rose`«), and a »long« description for examining it (»`It does looks pretty, and it has thorns. But will it smell as wonderfully...?`«) .[35]

The attribute string is empty, ie the `rose` can be taken, is visible, and doesn't act like a door, room, key, or container. It is located in room 20 (presuming you have set up a room with that number), and its »special« value will be initialized to 10, which we will use as the increment of the life energy achieved by smelling it.

2.7.2. Defining »Smelling«

Unfortunately, until now DODO has not the slightest idea what a »smell« is supposed to be, so trying »`smell rose`« will only lead to a curt response like »`How to 'smell'?`«

To teach DODO how to smell, enter a new verb in the `DATA` section after the existing verbs (which start at line 15000), and before the section marker −1 in line 18000:

```
...
16000 DATA 24,SMELL,WHIFF,
...
```

[34] Per default, DODO has no concept of life energy or such. But you can easily build something comparable into your own code.

[35] Bear in mind that the »`@`« at the end of the long description string will append the subsequent string in the data section to the description itself. See 6.4 for more details.

This will make Dodo aware that there *is* a verb with index 24 called `smell`, and it has a synonym called `whiff`.[36]

Before you stark working on the rose's smell, you need to provide some code for default responses of Dodo, namely if the player tries to smell anything *but* a rose.

The corresponding code section must begin at a line determined by the verb's index »`i`«: The value is $4000 + i * 100$, ie in this case line 6400. So, write something like:

```
...
6400 ? "You smell no particular aroma.":return
...
```

That will already do the trick and make all items beside the rose non-smelling.

2.7.3. The Hook

2.7.3.1. Setting up the Hook

Finally, you need to provide a hook which intercepts the »special« behaviour of the rose which is different from all other smelled items.[37]

So, first of all define a hook with a simple `DATA` line in the corresponding section:

```
...
55150 data 5, 24, 99, 0, 0
...
```

These five numbers are already all there is to it. The code line basically means, »Hook number »5« is to be triggered whenever the player types the verb »`smell`« (number »24«) followed by the subject »`rose`« (number »99«) and any object (»0«), regardless of the current location (number »0«).«

Upon initialization, the hook is automatically active and ready to be triggered.

2.7.3.2. Executing the Hook

Now, whenever the hook is triggered, Dodo will attempt to perform the corresponding code section – which, for hook number `i` begins at line $35000 + i * 1000$. Thus, assuming that the variable `zz` holds your current »life energy«, write something along the lines of –

```
...
40000 ? "The rose smells wonderful, and you feel refreshed and ";
40010 ? "invigourated."
40020 zz=zz+sp%(99):ok=1:return
```

[36] Note the terminating comma »`,`«: You *must* provide three synonyms for each verb. If a comma is the last character on a `DATA` line, then the C 64's BASIC interpreter will tacitly insert an invisible empty string (or the 0 value).

[37] Theoretically you could also make that distinction directly in the `smell` verb's code section, but this would be less elegant.

```
...
```

Line 40020 may require some explanation:

- As mentioned in the definition of `rose`, the variable `sp%(99)` holds the increment for the life energy which will be gained by smelling the rose.
- At the end of your code section, you should set `ok` to any true value. Not doing so will result in DODO performing the default action for the verb *in addition* to the hook after that is done.[38]
- Conclude the hook's code section with `RETURN` to avoid a fallthrough of code execution to the subequent hook's section.

Currently, the code doesn't yet work properly, because smelling the rose will *always* invigourate the player, regardless of whether he has the rose in possession or not. This can be remedied by checking the presence of the rose with the function `nh()` beforehand – see 4.2.5 for details how to handle this.

If you feel adventurous, you can expand this example easily. For example, you could set up a fuse when the rose is picked up (»plucked«) by the player for the first time. After a certain number of moves, the rose is considered »withered« and will lose the magic powers of its scent. To do so, disarm its hook in the fuse code (see below), and change the rose's long description to something less inviting, like »`The plant has shrivelled away. Its leaves are dead and colourless`«, or similar.

2.7.3.3. Disarming the Hook

When starting up a scenario, any hook for which the definition is read also is armed immediately, ie ready to be triggered. Due to game requirements, or simply for performance reasons, it may be desirable to disarm a hook later on.

The state of the hooks is stored in the variable `ho%`: If bit `i` is set, then hook number `i` is armed. So, to disarm hook `i`, all you need to do is clear `ho%`'s corresponding bit:

```
ho%= ho% and not (2^i)
```

To arm the hook again at a later stage, set the bit again:

```
ho%= ho% or (2^i)
```

Note:	This method is »safe«, ie arming or disarming the same hook several times in a row will *not* cause any undue trouble.

[38] There may be occassions where you want this kind of behaviour, but here this would lead to responses like `The rose smells wonderful, and you feel refreshed and invigourated. You smell no particular aroma.`, where the default reply is tacked onto the hook's reply.

Part II.

Reference

3. Processing Stages

Here is a brief list of the stages involved in preparing a scenario and processing moves. In italics, the scenario author's code comes into play.

1. Start DODO initialization
2. Call line 20000: *Set scenario boundaries (see 4.1.1)*
3. Scenario initialization: Read scenario data etc.
4. Call line 21000: *Set demons and fuses, print a scenario banner (see 4.1.2)*
5. For each move:
   a) Prompt Input
   b) Parse into verb, subject, object
   c) Check for Darkness
   d) *Evaluation of Hooks*
   e) if `ok` is `FALSE`, then:
      i. Standard behaviour for regular (*and extended*) verbs
      ii. *Demons and fuses*
   f) Call line 22000: *Check for game end conditions (see 4.1.3)*

You'll find more detail in 2.3.2.

4. User-defined Code, Functions and Variables

This chapter summarizes all functions and variables used/provided by DODO.

4.1. Scenario-specific Code Sections

This section details the »particulars«, ie the line numbers for code sections which must be provided by the author to define his scenario, along with the data sections of chapter 6. At minimum, each section must consist of a `RETURN` statement on the line which was the `GOSUB` target.

Note:	*All* jump targets *must* be valid code lines with at least one command. Jump target code should *end* in a `RETURN` statement, unless you want to create a »fall-through« condition where the subsequent code section will also be executed.

4.1.1. Line 20000: Setting the Scenario Size

This code is called at the *start* of DODO's initialization. Your code performs the following functions:

- Set the player's starting position »`lc`«
- Optionally change the default size of your scenario by setting the values of the variables `nd, nf, nh, no, nr, nv` (number of demons, fuses, hooks, items, rooms, and verbs, resp.)

Per default, DODO reserves memory for 32 items,[1] 32 verbs, as well as 8 hooks, demons and fuses each.

If you require more or less components in your game, you should adjust the corresponding variable accordingly: Reducing your scenario to the minimum will reduce memory requirements and significantly speed up the game.

Note that you there is an absolute maxmimum of 16 hooks and 16 demons for your game (see 2.4.8, 2.4.9).

[1] Note that this is the combined number of ordinary game objects, plus rooms

4.1.2. Line 21000: Set Demons and Fuses and Print a Banner

This code is called at the *end* of the scenario intiialization, but *before* the first move is processed. The code here should perform two functions:

- Set all demons and fuses which are supposed to be running at the start of the game (see 2.4.9)
- Print a »banner« or similar which introduces the player to the game.

4.1.3. Line 22000: Game-end Conditions

Line 22000 is called each time after *after* processing a move – basically, line 22000 is a demon which is definitely called at every move and can't be switched off.

Winning or losing conditions should be evaluated here (since per default DODO itself has no idea when a game is supposed to end, and will happily play forever), and you may place any code here which you will definitely want to have executed after each move.

4.1.4. Line 30000: Code for Demons

This section contains code to be performed when a demon (see 2.4.9) is active.

For demon number »`i`«, code starts at line $30000 + 100 \cdot$ `i`.

4.1.5. Line 32000: Code for Fuses

This section contains code to be performed when a fuse (see 2.4.9) is expired.

For fuse number »`i`«, code starts at line $32000 + 100 \cdot$ `i`.

4.1.6. Line 40000: Code for Hooks

This section contains code to be performed when a hook (see 2.4.8) is triggered.

For hook number »`i`«, code starts at line $40000 + 1000 \cdot$ `i`.

4.2. Pre-defined Functions

These functions are `DEF`'d in lines 14000ff and may be used in your hooks.

4.2.1. »`FN rf()`« – Real free memory

Returns the amount of free memory as calculated from the vectors at addresses 51/52 and 49/50, resp. This function is used to have an estimate for remaining memory without triggering a time-consuming `FRE()` operation.

See 2.6 for more details.

4.2.2. »FN te()« – Time elapsed (since move processing began)

Returns the number of seconds elapsed since the processing of the current move began, rounded to hundredths of seconds.

This is only used for diagnostic purposes.

4.2.3. »FN do(x)« – Demon »x« is on?

Returns TRUE, if demon x is currently active.[2]

4.2.4. »FN hh(x)« – Hook »x« is on?

Returns TRUE, if hook x is currently *active*.

Note:	»Being active« for a hook only means that DODO will check for the hook conditions in every move, not that these conditions actually were met.

4.2.5. »FN nh(x)« – Item »x« is not here?

Returns TRUE, if item x is *not* »in reach«, or »present« (see 2.4.2 for the concept of »presence«). »Being in reach« is defined as either being in the same room as the player, or in the player's possession (ie, inventory).

Note:	An item inside a container in the same room as the player is *not* considered to be »here«. The reasoning behind this is that if it's inside a locked box or such, it would not be »reachable«. Of course, different situations may require a different evaluation.

4.3. Pre-defined Code Sections

These code sections belong to routines which didn't fit into a single line, or which would require more than one parameter to work properly.

4.3.1. Routine 10000: Is Attribute p1$ Set?

- Parameters:

  p0$ Attribute string copied from some item

  p1$ Attribute letter

- Return value:

[2] There is no corresponding functions for fuses, because fuses are evaluated differently. See 2.4.9

`r0` $= -1$, if the attribute letter was found inside the attribute string, else 0.

Note:	To-do: Optimize this by not copying the attribute string, but using it *in situ*.

4.3.2. Routine 10100: Clear Attribute `p1$`

- Parameters:

  `p0$` Attribute string

  `p1$` Attribute letter

- Return value:

  `r0$` =`p0$` with all occurences of `p1$` inside it removed

This function works properly if `p1$` occurs several times, or not at all inside `p0$`.

4.3.3. Routine 10200: Print »It's not here.«

- Parameters: —
- Return value: —

Simply print a messagethat the current subject `s` is not currently present. (See also 2.4.2).

4.3.4. Routine 10300: *Pretty print* Long Strings

- Parameters:

  `p0$` Source string to be prettied up

- Return value:

  `r0$` Result string with line breaks

This routine »prettifies« strings of arbitrary lengths and introduces line breaks in the string's whitespace when neccessary, so there will be proper wordwraps on a 40 character screen. It works correctly under most circumstances (several subsequent space characters etc.). Only blanks » « are recognized as whitespace.

The routine really makes only sense for the `lo$()` long description strings of items, and this is the only place where it's currently used. (Called from line 9120).

Note:	This routine is quite time- and memory-consuming. You might want to consider one of these alternatives: • Introduce sufficient spaces or line breaks manually in your definition strings, or • Call routine 10300 »on occassion«, whenever you have to deal with a particularly long string. Note that with the latter option you'll slow down the output of move results.

4.3.5. Routine 10400: Is it Dark in Here?

- Parameters: —
- Return value:
  `d` $= 0$ if the room is »lit«, $= -1$ otherwise

This routine is used to determine whether the current location is »in darkness« (see 2.4.3 for the concept).

If the room has it's »darkness« attribute »`D`« set, *and* if none of the items present (see 2.4.2) in the room has the »emitting light« attribute »`*`« set, then the room is dark, otherwise it's lit. (The routine will not work properly if, for example, a light-emitting firefly is held inside a transparent bottle.)

Because this routine itself calls routine 10000, it can't use »`r0`« as a return value. »`d`« is used instead.

4.3.6. Routine 10500: Print »Progress Whirly«

- Parameters: —
- Return value: —

Simply prints a little square dancing in place while some background processing (game initialization, move evaluation ...) takes place. Each call of the routine will make the square move one position.

The routine is fairly well-behaved and will not wander off screen etc.

4.4. Variables

This are *all* variables used by Dodo. You may use any of them to your advantage (and, naturally, at your own risk), and you may safely assume that you can use any *other* variable name without Dodo interfering with it. The first letter of variable names (mostly) follows some conventions:

`h` : connected with a user »hook«

`l` : »location«

`n` : »number of«, the variable indicates the number of verbs, rooms etc. It represents the highest index of a certain array.

`p, r` : »parameter« and »return values« for a user routine[3]

`t` : »temporary« variables

`v, s, o` : related to »verbs«, »subjects« and »objects«

Following is a table of all variable names used by DODO. The third table column refers you to the manual section where the use is explained in more detail.

Note:	No variable names beginning with »`w`« through »`z`« are used by DODO. All of these variable names are safe for you to use in your own code unmolested by DODO.

[3]Of course, Commodore BASIC doesn't provide for named parameters to a routine, so this is just a convention but by no means safe

Name	Function	see
at$(no)	Attribute string of an item	5
br$	Shorthand for space$ " "	
co$	Player's current command line	2.3.2
cr$	Shorthand for CHR$(13)	
d	Darkness (it's dark here)	2.4.3
de%	Flag: Demon is activated?	2.4.9
di$(6)	Names of possible directions (north, etc.)	
ex%(nr, 5)	Exits of rooms	2.4.5
fi$(no)	First description of object	
fu%(nf)	Time left for fuse	2.4.9
ho%	Flag: Hook is activated?	2.4.8
hh(nh), hv(nh), hs(nh), ho(nh), hl(nh)	Hook configuration: Handle, verb, subject, object, location	2.4.8
i, k	Loop vars	
lc	Current location of player	2.4.2
lc%(no)	Location of item (can be a room or another item)	2.4.2
lo$(no)	Long description of object	
lp	"-1", this location means "item is player-held"	2.4.2
mv	Move number (in the game)	
nd, nf, nh, no, nr, nv	Number of demons / fuses / hooks (do not confuse with nh()!) / items (total) / rooms / verbs	
o, o$	Index and name of current command object	2.3.2
oo	"Object old", from previous move	2.5.2
ok	"Okay", move is done, no further rules to execute	
p0, p1, p2 p0$, p1$, p2$	General parameters for procedures (*Do not recurse!*)	
r0, r0$	Return values of procedures	
s, s$	Index and name of current command subject	2.3.2
so	"Subject old", from previous move	2.5.2
sp	sp= sp%(s), "special" value for current subject	
sp%(no)	"special":	
	for keys: which object to open with that	2.4.4.2
	for doors: what is their corresponding 2nd door	2.4.6
t0, t0$, ...	Temporary vars	
tc, tc$()	"Ticks", for progress whirlies	
te()	"Time elapsed" (in processing a move)	
ts, tv, tx	Temporary vars	
tz	Timer value at the beginning of processing (a move)	
v, v$	Index and name of current command verb	2.3.2
vo	"Verb old", from previous move	2.5.2
vb$(nv, 2)	Verb names (max 3 synonyms for each)	2.4.7
vi%(no)	Has this room/item been visited/seen?	

5. Attributes

This is a list of the pre-defined attributes for DODO. Only characters from this list will be used by DODO, so you are free to use any others with the `at$()` for your purposes.

DODO discriminates between upper and lower case letters: »`A`« and »`a`« are considered to be different flags.

- `a` Animated, item is an NPC (not used, but reserved)
- `c` Container, can hold other items inside
- `d` Door
- `D` Darkness (needs to be made explict, rooms are per default lit)
- `e` Edible/drinkable
- `f` Fixed, can't be `TAKE`n
- `i` Invisible, not listed upon `LOOK` or `EXAMINE`
- `k` Key, can `LOCK` and `UNLOCK` lockable items
- `l` Lockable, item can be `LOCK`ed/`UNLOCK`ed
- `L` Item *is* actually locked
- `m` Movable (not used, but reserved)
- `o` Openable/closeable
- `O` Item *is* currently open
- `r` Is a room
- `t` Is transparent
- `*` Emitting light, makes an otherwise dark room lit

6. »DATA« formats

This chapter outlines the format for the DATA sections which define the scenario, and describes to which variables the data are mapped.

6.1. Sequence of Data Sections

The sequence of data sections in the BASIC program must be as follows:

1. Standard verbs
2. User-defined verbs
3. Items (including rooms)
4. Hooks

Data sections are seperated from each other by a single data entry of -1.

6.2. Use of the »Index«

All data entries begin with an index. This index is used as the array index into which the subsequent data are written.

The index is provided to make it easier for the scenario author to keep track of his verbs and items: Internally, Dodo always refers to the index.

It is possible to leave »gaps« in the numbering of array elements, though this is wasteful on resources. It's also possible to assign an index twice which will result in the first definition being overwritten. (This will be useful in rare cases only.)

The highest index used must not exceed the default array sizes or the values set in line 20000 (see 4.1.1), resp.

6.3. Verb Definitions

A verb definition which extends the standard vocabulary (see 2.4.7) consists of four data:

1. `i`: An index number
2. `vb$(i, k)`: Three synonyms for the verb

The synonyms are there to make it easier for the player to achieve what he wants to do, or to adapt to different situations. (Sometimes it may appear more natural to use `ASK` instead of `SAY`, although both actions are evaluated in the same manner.) At least one of the synonyms must be a non-empty string.

After parsing for the verb, only the first entry `vb$(i, 0)` is used to describe it subsequently.

Example:

```
15100 data 9, I, INV, INVENTORY
```

6.4. Item Definitions

An item definition consists of the following data:

1. `i`: Index number
2. `nm$(i)`: Item name. This must be a single word with no spaces, and must not be an empty string. Each item's name *must* be unique.
3. `fi$(i)`: »First« description. This description is given when a room is first entered or an item is first encountered as determined by `vi%(i)`
4. `lo$(i)`: »Long« description. Given as response to `EXAMINE`. See 6.4.
5. `at$(i)`: »Attributes«. These are the attributes given to the item upon startup. (See 5 for details.)
6. If the item is a room or a door (ie if »`r`« or »`d`« are present in the corresponding attribute string `at$(i)`):
   - `ex%(i)`: Six »exits« the room/door provides. A value of 0 means »no exit in that direction.«
7. `lc%(i)`: »Location« of the item at the beginning of the game. For rooms, this may safely be 0. See 2.4.2.
8. `sp%(i)`. »Special« variable, used for doors and keys, see 2.4.10.

Example:

```
50030 data 2, "bar", "You enter the small bar of the opera house."
50040 data "The bar is pretty much deserted. Overall, it's not a very
   pretty @"
50050 data "place. You could leave back north to the foyer.", "r"
50060 data 1, 0, 0, 0, 0, 0,  ,10
```

In this case, `lc%()` of the room would be empty (it makes no sense to assign a location to a room), and the value of `sp%()` would be 10.

Long Descriptions

Long descriptions `lo$()` may in contrast to other strings be longer than a code line. If the last character of a `lo$()` is the At-Sign »`@`«, then another string from the data section will be read and appended to `lo$()`.

Still, the total length of `lo$()` must not exceed 255 characters.

6.5. Hook Definitions

A hook definition consists of the following data:

1. `i`: Index number
2. `hv(i), hs(i), ho(i), hl(i)`: The verb, subject, object and location index the hook will be triggered by.

For more details, see 2.4.8.

Example:

```
55110 data 5, 17, 0, 9, 0
```

This hook with index 5 would be triggered by the following verb/subject/object/location combination:

Verb:	17
Subject:	any
Object	9
Location:	any

Appendices

.1. License

.2. About the author

Elmar Vogt was born in 1966, studied physics and is now working as a tech writer for a major German electronics company. Dodo is his first open source software project.

His first programming experiences were a Perkin-Elmer multiuser system at school, the *C 64*, and the *Atari ST*. He took up an interest in interactive fiction and text adventures around 2011, first in writing games, now in developing an authoring system.

»Dud, the Dodo«

Dodo's logo character was created and drawn by Bryan Hillesheim.

Elmar Vogt
Ludwigstr. 57
90763 Fürth
elvogt@gmx.net
Tel.: (++49) 173/591 29 93

[1] Well, without that option you wouldn't really be able to do anything with Dodo ...