Writing Games in Assembly Language

Introduction

At the end of SD-8516 Stellar BASIC we wrote a game called ROBOTS.BAS. This was a fun take on the classic CHASE or 'robots' game from the bsd games collection. Now, for our study of Assembly Language, let's write a similar game: 'robots.asm'. Don't worry if you're not familiar with the BASIC version of this program; everything will be explained here.

However, if this is your first time writing an assembly language program, you might want to familiarize yourself with the architecture of the SD-8516 first:

- SD-8516 Assembly Language

Onwards and upwards.

First Steps: Assembling and Running a Program

Just like BASIC programs have line numbers, or C programs have #include statements, an assembly language program should start with an `.address` statement. This will tell the assembler where to start assembling the program. This enables us to use labels like `start:` and `message:`. The assembler will calculate label addresses relative to this value while assembling the program.

```
    .address $000100
start:
    LDELM @message
    LDAH $18
    INT $10
    RET
message:
    .bytes "It works!", 13, 10, 0
```

If you enter this code into ed and save it as robots.asm, then you can do:

```
as robots.asm robots
```

To run the program, just DLOAD (or LOAD) it and type “SYS 256”. This can be a bit confusing; why 256? Without instructions, people might not know what to do. There are three options.

- 1. Tell people they have to type SYS and a number. Mysterious.
- 2. use `.address $030100` (the default SYS location). People can run it with SYS.
- 3. Include a BASIC stub so people can type RUN.

For this program I will demonstrate option #3, although #2 is probably just as common. Our program now becomes:

```
    .address $000100

    ; BASIC stub: 10 SYS 269
    .bytes $FB, $0A, $00
    .bytes "SYS 269", $00
    .bytes $00, $00

; Entry point at $00010D (decimal 269)
start:
    LDELM @message
    LDAH $18
    INT $10
    RET

message:
    .bytes "It works!", 13, 10, 0
```

Using the editor

Enter this into ed by typing:

```
  f robots.asm
  a
```

Enter the program then type `.` on a line by itself. Then type w to save and q to quit. Now you can assemble the program.

Assembly

To assemble a program named `robots.asm` in d-tank, type:

```
  as robots.asm r
```

LOAD and RUN

Finally, enter

```
  dload r
```

After you `dload r` you can type LIST. It will show:

```
  10 SYS 269
```

Now you can RUN the program. This is a fun and convenient way to let users launch your assembly program.

Game 1: Robot

Our first game will be a version of `robots` called “Robot”, since there's only one robot.

Part 1. Program Structure

We begin any project by mapping out the structure of the program and talking about the subroutines, variables, and main ideas behind the code.

For `robots` we need:

- A game loop
- A subroutine that draws the map
- A subroutine to get player input
- A subroutine to move the robot
- Collision detection

Initialization

Just like in the BASIC version, we can start by drawing the map. First, let's define our variables and then the map drawing function. We will include a simple main loop whose only job is to call draw_map once.

```
; Define variables.
; This section uses memory-location variables. $00 is memory location $00.
.equ PX $00        ; player x
.equ PY $01        ; player y
.equ RX $02        ; robot x
.equ RY $03        ; robot y

    .address $000100

    ; BASIC stub: 10 SYS 269
    .bytes $FB, $0A, $00
    .bytes "SYS 269", $00
    .bytes $00, $00

start:
    ; First, let's initialize our variables.
    LDAL #19
    STAL [@PX]         ; Store 19 as initial player X location.
    LDAL #11
    STAL [@PY]         ; Store 11 as player's initial Y
    LDAL #1
    STAL [@RX]
    STAL [@RY]         ; The robot starts at 1,1 for now,
```

```
main_loop:
    CALL @draw_map
    RET
```

Now that we have initialized our main variables and set up a program loop we are ready for the `draw_map` subroutine.

Part 2. Draw the Map

Like in the BASIC version, the `draw_map` subroutine will begin by clearing the screen. So the first two things we do are we replicate VSTOP and CLS from BASIC:

```
    .equ VIDEO_MODE $01EF00 ; This defines VIDEO_MODE as a memory location
variable.

draw_map:
    ; VSTOP
    LDAL $81
    STAL [@VIDEO_MODE]
    ; CLS
    LDAH $10        ; CLS
    INT $10
```

In assembly, we can't easily call a BASIC command, so we do things slightly differently. Setting video mode to $81 sets us up in mode 1 but stops video updates (because bit 7 is set). So $81 means “mode 1, but set bit 7 so we don't update the screen”. This is normally done to stop screen tearing caused by drawing during an update. But here we're just doing it for academic reasons since assembly is so fast the screen won't have time to update until we're done.

Secondly, INT $10 AH=$10 is the clear screen function. We can call that as an interrupt helper here.

Drawing the border

To draw the border we need to set up a loop that draws stars on the top and bottom, left and right, just like in BASIC. Here we will examine the top and bottom loop:

```
    ; Draw top/bottom border: '*' at (0..39, 0) and (0..39, 24)
    LDBL #'*'       ; Load character into BL
    LDXL #0         ; set X location

dm_top_bottom:
    LDYL #0         ; set Y location
    LDAH $11        ; write char AL at XL, YL
    INT $10
```

This simply loads the character, sets the x and y locations, and calls write_char_at_xy via INT $10.

```
    LDYL #24
    LDAH $11        ; write char (on the bottom of the screen this time)
    INT $10
    INC XL
    CMP XL, #40
    JNC @dm_top_bottom
```

Here's the loop: After the top and bottom characters are drawn, we INC XL (which is the column marker). Then we CMP XL, #40. This means “check if it's under 40”– i.e. 0-39. If we are on 39, we need to draw again. If we just drew on column 39 and then we INC XL to 40, that means don't continue the loop, and we fall-through to the next function; drawing on the left and right sides.

```
    ; Draw left/right border: '*' at (0, 0..24) and (39, 0..24)
    LDBL #'*'
    LDYL #0
dm_left_right:
    LDXL #0
    LDAH $11        ; write char
    INT $10

    LDXL #39
    LDAH $11        ; write char
    INT $10

    INC YL
    CMP YL, #25
    JNC @dm_left_right
```

This code works just like the top and bottom but on the left and right sides. Instead of looping from 0 to 39 on XL, we will loop from 0 to 24 on YL. As before, we CMP to see if YL is 25. If it is, we fall through. A simple loop, in essence the exact same loop technique we used to draw the map in BASIC.

Draw the player and robots

```
    ; Draw player '@'
    LDBL #'@'
    LDXL [@PX]
    LDYL [@PY]
    LDAH $11        ; write char
    INT $10

    ; Draw robot 'r'
    LDBL #'r'
```

```
    LDXL [@RX]
    LDYL [@RY]
    LDAH $11        ; write char
    INT $10

    ; VSTART
    LDAL #1
    STAL [@VIDEO_MODE]
    RET
```

Finally we draw the player and the robot, and turn video updates back on (Set mode to 1, but without bit 7 set). As we are done drawing the map, we can now RET. If you save, assemble and run the program now you will see it draws the border, player and robot before exiting.

It's working!

Part 3: Player Input

Let's begin part 3 by adding a new subroutine to our main loop. Our main loop now becomes:

```
main_loop:
    CALL @draw_map
    CALL @get_input
    RET
```

get_input

The primary goal of this section is to get input from the player and deal with it. We can do this by using the getkey function:

```
get_input:
    LDAH $02        ; getkey()
    INT $10
```

In this case, the function specification for AH=$02, INT 0x10 is:

```
  ;
================================================================================
  ; AH=02h - Read Character (Blocking)
  ; Input:  None
  ; Output: AL  = ASCII character
  ;         B  = number of keys that were in buffer before this call
  ;         K  = keyboard flags at time of press
  ; Note:   X, Y preserved for cursor position
```

```
  ;         This function BLOCKS until a key is pressed
  ;
================================================================================
```

As we can see, this function returns the ASCII code of the key the player will press in AL. Thus, we can deal with the player's commands.

uppercase or lowercase?

The player might type an uppercase command or a lowercase command. Let's normalize the keys to uppercase before processing them:

```
    ; Uppercase: if AL >= 'a', it must be an uppercase character (or an
invalid command).
    CMP AL, #97
    JNC @gi_check_keys      ; AL < 'a', skip

    CMP AL, #123
    JC @gi_check_keys       ; AL >= '{', skip
    SUB AL, #32
gi_check_keys:
    ...
```

CMP works as follows:

```
  CMP A, B        ; if A >= B, then set carry
```

Therefore,

```
  CMP AL, #97     ; if AL is greater or equal to #97 ('a') then set carry
```

So therefore if carry is NOT set, skip past the SUB command. Or else (if it's a lowercase ASCII) subtract 32 to convert from lowercase to uppercase.

The second compare has a protective function.

```
  CMP AL, #123             ; if AL >= 123, set carry.
  ; If it's '{' or anything after, skip:
  JC @gi_check_keys        ; Jump if Carry = Jump if AL > 'z' (al >= '}')
```

So after these two checks, the only values that reach SUB AL, #32 are exactly the bytes from 97 to 122 inclusive; i.e. 'a' to 'z'.

Visual summary:

AL Value	Meaning	What Happens
< 97	before a	Skipped.
97-122	a to z	SUB #32 becomes 'A' to 'Z'.

AL Value	Meaning	What Happens
>= 123	after 'z' '*}' and up*	Skipped.

Processing input

We will use the HJKL convention:

```
gi_check_keys:
    CMP AL, #'H'
    JZ @move_left

    CMP AL, #'J'
    JZ @move_down

    CMP AL, #'K'
    JZ @move_up

    CMP AL, #'L'
    JZ @move_right

    CMP AL, #'Q'
    JZ @quit_game

    RET
```

This is easy enough; after a CMP, if the values are equal then the zero flag is set. We will use JZ (jump if zero flag is set) to go to the correct routine.

Moving the Player

```
move_left:
    LDAL [@PX]        ; load variable PX into register AL.
    DEC AL            ; X = X - 1 (move to the left!)
    CMP AL, #1        ; bounds check. Is the player on the border?
    JC @ml_ok         ; AL >= 1, so it's ok. skip to ml_ok,
    LDAL #1           ; If  we reach here the player was on the border; set
position to 1.
ml_ok:
    STAL [@PX]        ; pass! Save the player's location.
    RET
```

This is the move_left command. We begin by loading the player's X location into AL and executing the move. After a bounds check, we save the variable. All of the other moves operate this way; we

attempt to move the player and then check for bounds:

```
move_down:
    LDAL [@PY]
    INC AL
    CMP AL, #23
    JNC @md_ok              ; AL < 23 (i.e. <= 22), keep
    LDAL #23
md_ok:
    STAL [@PY]
    RET

move_up:
    LDAL [@PY]
    DEC AL
    CMP AL, #1
    JC @mu_ok               ; AL >= 1, keep
    LDAL #1
mu_ok:
    STAL [@PY]
    RET

move_right:
    LDAL [@PX]
    INC AL
    CMP AL, #39
    JNC @mr_ok              ; AL < 39 (i.e. <= 38), keep
    LDAL #38
mr_ok:
    STAL [@PX]
    RET
```

Quit Command

The player can also quit the game by pressing q:

```
quit_game:
    ; CLS
    LDAH $10
    INT $10

    ; Print "QUIT GAME"
    LDBLX @msg_quit
    LDAH $66
    INT $05
    LDAH $64
    INT $05
```

```
    RET

msg_quit:
    .bytes "QUIT GAME", 0
```

Moving the Robot

Moving the robot is similar to the player, except that we do not have commands from the robot (it has no keyboard to type), so we must simply try to move it towards the player.

We will begin by determining if the player is to the left or the right, and then move left or right towards the player. Then we will check if the player is above or below us and then move up or down towards the player.

```
move_robot:
    ; X axis: move RX toward PX
    LDAL [@RX]
    LDBL [@PX]
    CMP AL, BL
    JZ @mr_check_y          ; RX == PX, skip
    ;; If carry is set, then AL is bigger than BL and we should decrease
RX's position.
    JC @mr_x_dec            ; RX >= PX (but not equal), so RX > PX


    ; fall-through RX < PX
    ; In this case since the RX is less than PX, we should increase it to
move towards the player.
    INC AL
    JMP @mr_x_done
mr_x_dec:
    DEC AL
mr_x_done:
    STAL [@RX]              ; save the robot's X position for draw_map.

mr_check_y:
    ; Y axis: move RY toward PY
    LDAL [@RY]
    LDBL [@PY]
    CMP AL, BL
    JZ @mr_done             ; RY == PY, skip
    JC @mr_y_dec            ; RY > PY
    ; RY < PY
    INC AL
    JMP @mr_y_done
mr_y_dec:
    DEC AL
mr_y_done:
```

```
    STAL [@RY]              ; save the robot's y position for draw_map.

mr_done:
    RET
```

After checking the relative location of the player vs. the robot, the robot attempts to move towards the player.

After this, you can update the main loop:

```
main_loop:
    CALL @draw_map
    CALL @get_input
    CALL @move_robot

    RET
```

Part 4. Win condition

The game is over when the robots collide with you. Here's the new main loop:

```
main_loop:
    CALL @draw_map
    CALL @get_input
    CALL @move_robot

    ; Check collision: PX==RX && PY==RY
    ; The collision check falls through to game_over only if both axes
match, otherwise loops back.
    LDAL [@PX]
    LDBL [@RX]
    CMP AL, BL
    JNZ @main_loop
    LDAL [@PY]
    LDBL [@RY]
    CMP AL, BL
    JNZ @main_loop
    JMP @game_over

    RET
```

That little bit at the end checks for collision. As an exercise to the reader, you can try to move it into a subroutine.

Next let's write the game-over ending:

```
game_over:
    ; CLS
    LDAH $10
    INT $10

    ; Print "OH NO! THE ROBOT CATCHES YOU."
    LDBLX @msg_game_over1
    LDAH $66
    INT $05
    LDAH $64                ; newline
    INT $05

    ; Print "GAME OVER."
    LDBLX @msg_game_over2
    LDAH $66
    INT $05
    LDAH $64
    INT $05

    RET

;
================================================================================
; String data
;
================================================================================
msg_game_over1:
    .bytes "OH NO! THE ROBOT CATCHES YOU.", 0
msg_game_over2:
    .bytes "GAME OVER.", 0
msg_quit:
    .bytes "QUIT GAME", 0
```

At this time, your `robots.asm` has the same functionality as BASIC's `ROBOTS.BAS`. In fact it's a little bit cleaner; instead of having to press ESC to quit the BASIC version, you have cleanly implemented a quit command.

Part 5. Music and Sound

Let's add a little sound effect to the game (just like in BASIC!)

```
robot_beep:
    LDELM @robot_sfx
    LDAH $52
    INT $11
    RET
```

```
robot_sfx:
    .bytes "T120 W1 V5 O2 L24 B", 0
```

Now add `CALL @robot_beep` right after `CALL @move_robot` in the main loop:

```
main_loop:
    CALL @draw_map
    CALL @get_input
    CALL @move_robot
    CALL @robot_beep
    ...
```

There, now you can PLAY music strings just like in BASIC. This is good!

robots.asm Complete Listing

```
// robots.asm

; Variables
.equ PX $00    ; player X
.equ PY $01    ; player Y
.equ RX $02    ; robot X
.equ RY $03    ; robot Y

.equ VIDEO_MODE           $01EF00   ; Current video mode (1 byte)

    ; Address statement.
    .address $000100

    ; BASIC stub: 10 SYS 269
    .bytes $FB, $0A, $00
    .bytes "SYS 269", $00
    .bytes $00, $00

; Entry point at $00010D (decimal 269)
start:
    ; Initialize variables
    LDAL #19
    STAL [@PX]
    LDAL #11
    STAL [@PY]
    LDAL #1
    STAL [@RX]
    STAL [@RY]

main_loop:
```

```
    CALL @draw_map
    CALL @get_input
    CALL @move_robot
    CALL @robot_beep

    ; Check collision: PX==RX && PY==RY
    ; The collision check falls through to game_over only if both axes
match, otherwise loops back.
    LDAL [@PX]
    LDBL [@RX]
    CMP AL, BL
    JNZ @main_loop
    LDAL [@PY]
    LDBL [@RY]
    CMP AL, BL
    JNZ @main_loop
    JMP @game_over

    RET

;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Draw the map.
;;;;;;;;;;;;;;;;;;;;;;;;;;
draw_map:
    ; VSTOP
    LDAL $81
    STAL [@VIDEO_MODE]

    ; CLS
    LDAH $10        ; CLS
    INT $10

    ; Draw top/bottom border: '*' at (0..39, 0) and (0..39, 24)
    LDBL #'*'
    LDXL #0

dm_top_bottom:
    LDYL #0
    LDAH $11        ; write char AL at XL, YL
    INT $10

    LDYL #24
    LDAH $11        ; write char
    INT $10
    INC XL
    CMP XL, #40
    JNC @dm_top_bottom

    ; Draw left/right border: '*' at (0, 0..24) and (39, 0..24)
    LDBL #'*'
```

```
    LDYL #0
dm_left_right:
    LDXL #0
    LDAH $11        ; write char
    INT $10

    LDXL #39
    LDAH $11        ; write char
    INT $10

    INC YL
    CMP YL, #25
    JNC @dm_left_right

    ; Draw player '@'
    LDBL #'@'
    LDXL [@PX]
    LDYL [@PY]
    LDAH $11        ; write char
    INT $10

    ; Draw robot 'r'
    LDBL #'r'
    LDXL [@RX]
    LDYL [@RY]
    LDAH $11        ; write char
    INT $10

    ; VSTART
    LDAL #1
    STAL [@VIDEO_MODE]

    RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Get input from the player.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
get_input:
    ; Blocking GETKEY -> AL
    LDAH $02
    INT $10

    ; Uppercase: if AL >= 'a' and AL < 'z'+1, subtract 32
    CMP AL, #97
    JNC @gi_check_keys      ; AL < 'a', skip
    CMP AL, #123
    JC @gi_check_keys       ; AL >= '{', skip
    SUB AL, #32

gi_check_keys:
    CMP AL, #'H'
```

```
    JZ @move_left
    CMP AL, #'J'
    JZ @move_down
    CMP AL, #'K'
    JZ @move_up
    CMP AL, #'L'
    JZ @move_right
    CMP AL, #'Q'
    JZ @quit_game

    RET

move_left:
    LDAL [@PX]
    DEC AL
    CMP AL, #1
    JC @ml_ok               ; AL >= 1, keep
    LDAL #1
ml_ok:
    STAL [@PX]
    RET

move_down:
    LDAL [@PY]
    INC AL
    CMP AL, #23
    JNC @md_ok              ; AL < 23 (i.e. <= 22), keep
    LDAL #23
md_ok:
    STAL [@PY]
    RET

move_up:
    LDAL [@PY]
    DEC AL
    CMP AL, #1
    JC @mu_ok               ; AL >= 1, keep
    LDAL #1
mu_ok:
    STAL [@PY]
    RET

move_right:
    LDAL [@PX]
    INC AL
    CMP AL, #39
    JNC @mr_ok              ; AL < 39 (i.e. <= 38), keep
    LDAL #38
mr_ok:
    STAL [@PX]
```

```
    RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Move the robot.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
move_robot:
    ; X axis: move RX toward PX
    LDAL [@RX]
    LDBL [@PX]
    CMP AL, BL
    JZ @mr_check_y          ; RX == PX, skip
    JC @mr_x_dec            ; RX >= PX (but not equal), so RX > PX
    ; RX < PX
    INC AL
    JMP @mr_x_done
mr_x_dec:
    DEC AL
mr_x_done:
    STAL [@RX]

mr_check_y:
    ; Y axis: move RY toward PY
    LDAL [@RY]
    LDBL [@PY]
    CMP AL, BL
    JZ @mr_done             ; RY == PY, skip
    JC @mr_y_dec            ; RY > PY
    ; RY < PY
    INC AL
    JMP @mr_y_done
mr_y_dec:
    DEC AL
mr_y_done:
    STAL [@RY]

mr_done:
    RET

;;;;;;;;;;;;;;;;;;;;;;;;
;; Game Over.
;;;;;;;;;;;;;;;;;;;;;;;;
game_over:
    ; CLS
    LDAH $10
    INT $10

    ; Print "OH NO! THE ROBOT CATCHES YOU."
    LDBLX @msg_game_over1
    LDAH $66
    INT $05
    LDAH $64                ; newline
```

```
    INT $05

    ; Print "GAME OVER."
    LDBLX @msg_game_over2
    LDAH $66
    INT $05
    LDAH $64
    INT $05

    RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Player quit game
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
quit_game:
    ; CLS
    LDAH $10
    INT $10

    ; Print "QUIT GAME"
    LDBLX @msg_quit
    LDAH $66
    INT $05
    LDAH $64
    INT $05

    RET

robot_beep:
    LDELM @robot_sfx
    LDAH $52
    INT $11
    RET

;
================================================================================
; String data
;
================================================================================
msg_game_over1:
    .bytes "OH NO! THE ROBOT CATCHES YOU.", 0
msg_game_over2:
    .bytes "GAME OVER.", 0
msg_quit:
    .bytes "QUIT GAME", 0
robot_sfx:
    .bytes "T120 W1 V5 O2 L24 B", 0
```

Game 2: Rogueima

Game programming is not that difficult, as long as you establish the scope of your game and understand all the individual parts, it can be considered a robotic, mechanical exercise of coding. Even if there are some parts you don't understand, you can learn-as-you-go so long as you have a conception of what it should be like.

But it's important not to bite off more than you can chew. Let's continue on our journey of learning Assembly for the SD-8516 by moving from `Robot` towards `Rogue.`

Scope and Planning

As always, the first step is scope and planning. What exactly are we trying to accomplish with this game? And, why Assembly, why not BASIC? The simple answer is that in a typical roguelike game, although it is character based, it requires data structures and computations that are difficult to perform in BASIC. In it's current form. Stellar BASIC is not set up to really make a big, professional-quality game. That will change in the future as it gets improved over time, but for now, if you want to make something 'big' like a clone of one of the great classic games, you are probably better off with Assembly. Let's hope I can get a C compiler up someday and that might change for good.

Not that there's anything wrong with assembly, but it takes twice as long to really get anything done. It ends up twice as fast and half the size, though, so, it's not all bad!

Ok, here's what we need.

- Everything in the `robot.asm` game
- A status display showing our character information
- A map that we can explore
- Some form of 'fog of war' or 'lighting' – not required, but nice.
- The ability for monsters to randomly appear
- The ability to fight and defeat monsters
- The ability to pick things up and use them
- The ability to view an inventory
- The ability to save your game

Once we have all these things, the game will be considered finished. Beyond the above, there is actually quite a lot of work that would need to be done to make a game like NetHack (or even, like bsdgames' rogue), but that's okay. Baby steps.

Step 1. Drawing the Screen

The easiest first step is to draw a status bar. In classic rogue, the status bar is on the bottom of the screen. But this means the map is now 80 x 23 (or 80×22 on some displays) characters, and that is incredibly unbalanced. Let's try a different approach and put the status on the right. This is how Temple of Apshai did it, not to mention Ultima 4, and many other games. The Bard's Tale series, technically, is an example of the other style; having the status on the bottom.

Second, let's use Mode 6 for this game, since it is the classic terminal interface you would expect from

a game like Rogue or NetHack.

```
; ************************************
; *                          *       *
; *                          *       *
; *                          *       *
; *                          ***********
; *                          *       *
; *                          *       *
; *                          *       *
; *                          *       *
; *                          *       *
; ************************************
```

The above is not to scale, but is the idea. The left side will hold the map, and the right side will hold the status. Status, for now, will include Name, Strength, and Hit Points. That's all we need to start. Therefore we will need new variables to hold Name, Strength, and Hit Points, and we will need to write that into the status bar after we draw the map border.

To the front of the code we will add variable space for the player's stats:

```
.equ PX  $00    ; player X
.equ PY  $01    ; player Y
.equ RX  $02    ; robot X
.equ RY  $03    ; robot Y
.equ player_name    $04     ; max 16 characters.
.equ pname_zero     $15     ; zero-terminated
.equ player_hp      $16     ; max HP 255, 0=dead.
.equ player_str     $17     ; one byte to hold strength
```

An interesting question is what things like “strength” and “hit points” mean. We could just as well call them “hearts” or “health” or “attack power” or “skill”. We could use the old D&D idea of 3d6 (3 to 18) but maybe for a different game. Here we just set strength and health to 10, and worry what that means in game terms later on.

Therefore, our initialization becomes:

```
init_game:
    LDA #10
    STAL [@player_str]
    STAL [@player_hp]
```

You may notice we defined the variables differently here. Above, we added a .bytes imperative to reserve space. The label above the .bytes becomes the label of the memory address where those bytes are assembled. Instead, we could have defined the variables like this:

```
.equ player_str    $00       ; zero page address 0
.equ player_hp     $01       ; zero page address 1
.equ player_name   $02       ; 16 bytes + 1 zero starting at $02
.equ pname_zero    $12       ; this must always be a zero even if there
```

```
are earlier zeroes.
  .equ free_space    $13       ; free space starts at $13
```

Next, when drawing the map, we will use the following modified routine:

```
draw_map:
    ; VSTOP
    LDAL $86                  ; Mode 6 VSTOP
    STAL [@VIDEO_MODE]

    ; CLS
    LDAH $10        ; CLS
    INT $10

    ; Draw top/bottom border: '*' at (0..79, 0) and (0..79, 24)
    LDBL #'*'
    LDXL #0

dm_top_bottom:
    LDYL #0
    LDAH $11        ; write char AL at XL, YL
    INT $10

    LDYL #24
    LDAH $11        ; write char
    INT $10

    CMP XL, #59      ; if XL >= 10, then set carry.
    JNC @dm_skip_ms    ; So if it's not, then don't draw the middle
separator char.

    LDYL #10        ; status/text window separator
    LDAH $11        ; write char
    INT $10

dm_skip_ms:
    INC XL
    CMP XL, #80     ; Mode 6 is 80 chars wide.
    JNC @dm_top_bottom

    ; Draw left/right borders: '*' at (0, 0..24) and (59, 0..24) and (79,
0..24)
    LDBL #'*'
    LDYL #0
dm_left_right:
    LDXL #0         ; left border
    LDAH $11        ; write char
    INT $10

    LDXL #59        ; middle border
```

```
    LDAH $11        ; write char
    INT $10

    LDXL #79        ; right side border
    LDAH $11        ; write char
    INT $10


    INC YL
    CMP YL, #25
    JNC @dm_left_right

    ; Draw player '@'
    LDBL #'@'
    LDXL [@PX]
    LDYL [@PY]
    LDAH $11        ; write char
    INT $10

    ; Draw robot 'r'
    LDBL #'r'
    LDXL [@RX]
    LDYL [@RY]
    LDAH $11        ; write char
    INT $10

    ; VSTART
    LDAL #6               ; Turn off VSTOP
    STAL [@VIDEO_MODE]

    RET
```

The essential changes are, we section off the map differently, and we VSTOP/VSTART based on Mode 6.

Because we're using Mode 6, we need to initialize into Mode 6. Start becomes:

```
start:
    ; Set mode 6.
    LDAH $40    ; Set video mode
    LDAL #6     ; to 6
    INT 0x10

    CALL @init_game
    JMP @main_loop    ; start game

init_game:
    ; Initialize variables
```

```
    LDAL #19
    STAL [@PX]
    LDAL #11
    STAL [@PY]
    LDAL #1
    STAL [@RX]
    STAL [@RY]

    LDAL #10
    STAL [@player_str]
    STAL [@player_hp]
    CALL @get_player_name
    RET
```

Next, the game won't be any fun unless we can name our hero. Let's ask the player their name. To do this, we will need an INPUT function. Let's use IO_INPUT:

```
  ; ------------------------------------------------------------------------
----
  ; AH=$68: IO_INPUT - Read line of input from user
  ; Input:  None
  ; Output: ELM = pointer to input string (null-terminated, in @PATB_TBUF)
  ;         B = numeric value (parsed via atoi)
  ;         CF = 0 on success, 1 on empty input
  ; Notes:  Reads characters until ENTER (13).
  ;         Supports backspace for editing.
  ;         Echoes characters to screen.
  ;         ENTER is not echoed.
  ;         Max input length limited by PATB_TBUF size.
  ; ------------------------------------------------------------------------
----
```

```
get_player_name:
    LDELM @what_is_your_name
    LDAH $10      ; write string
    INT 0x10
    LDAH $68      ; IO_INPUT
    INT 0x05
    ; player name is now at ELM.

    ; Ensure player name is no longer than 16 characters.
    MOV FLD, ELM
    ADD FLD, #17
    LDAL #0
    STAL [FLD]

    ; Copy player name into variable space.
    LDFLD @player_name
name_copy_loop:
```

```
    LDAL [ELM]
    STAL [FLD]
    INC ELM
    INC FLD
    JNZ @name_copy_loop

    RET ;; return.

what_is_your_name:
    .bytes "What is your name? ", 0
```

Drawing the Status Area

When a function gets too big I like to break it up into smaller functions, that way it's easier to think about the program. It seems draw_map is getting too big, so I will break it into `draw_border`, `draw_player` and `draw_mobs`. Then we will add 'draw_world', 'draw_stats' and 'draw_msg'. These individual functions will all be called by draw_map. This makes it easier to co-ordinate things.

Secondly, let's move all these functions into a file called draw.asm, and the main file can be changed from robots.asm to rogueima.asm. You can keep an old copy of robot.asm if you want, but we aren't going to use it anymore.

> What is a “mob” you ask? A mob is a “mobile object”. In the game, each tile is represented by a letter; monsters, as well as walls, doors, items and the player, will all be represented by characters. This means a moving object like a monster (or the player) is the same kind of thing as a wall or a sword; an object in the game.

So let's get all the drawing functions done.

```
draw_map:
    CALL @draw_borders
    CALL @draw_world
    CALL @draw_mobs
    CALL @draw_player
    CALL @draw_stats
    CALL @draw_msgs
    RET

draw_borders:
    ; Put the old draw_map code in here.
    RET

draw_player:
    ; Put the old draw-player code in here.
    RET
```

```
draw_mobs:
    ; put the old draw robot code in here.
draw_stats:
    ; We will work on this next.
    RET

draw_msgs:
    ; We will work on this after.
    RET

draw_world:
    ; We will work on this last.
    RET
```

Our plan of attack is to write draw stats, then msgs, then world. For each one, we will need to define some sort of variable to hold the data. For STATUS, we already have everything we need; name, strength, and health. We'll throw in “steps” and “score”. You can add the steps and score variables yourself, or we will include them in the intermission program listing shortly.

To write at a specific locatoin, we will need to use the set_cursor routine:

```
  ;
================================================================================
  ; INT 0x10, AH=15h - Set Cursor Position
  ; Input:  Y = row (0-based)
  ;         X = column (0-based)
  ; Output: Clamps X and Y to valid range.
  ; Notes:  Works with current video mode (reads VIDEO_ROWS, VIDEO_COLUMNS)
  ;
================================================================================
```

We will also use INT 0x05 IO_PUTNUM, to draw numbers up to 65,535:

```
  ; ------------------------------------------------------------------------
----
  ; INT 0x05, AH=$63: IO_PUTNUM - Output number to screen
  ; Input:  B = number to output (16-bit, signed)
  ;         AL = 0 unsigned, 1 signed (we will let the caller decide)
  ; Output: Number written to screen as ASCII digits
  ; ------------------------------------------------------------------------
----
```

```
draw_stats:
    ; Set cursor
    LDA $1500
    LDX #61
    LDY #2
    INT 0x10
```

```
    LDELM @str_name  ; Draw 'Name: '
    LDA $1800        ; write string
    INT 0x10

    ; Draw player's name.
    ; This will draw right after the 'Name: ' above.
    ; That's why we added spaces to the strings (below).
    LDELM @player_name
    LDA $1800   ; write string
    INT 0x10

;;;;;;;;;;;;;;;;;;; Strength
    LDA $1500        ; set cursor
    LDX #61
    LDY #4
    INT 0x10

    LDELM @str_str     ; Draw 'Str: '
    LDA $1800          ; write string
    INT 0x10

    ; Draw player's strength score
    LDA $6300          ; print unsigned word (will print after string
above).
    LDB [@player_str]
    INT 0x10

;;;;;;;;;;;;;;;;;;; Health
    LDA $1500          ; set cursor
    LDX #61
    LDY #5
    INT 0x10

    LDELM @str_hp      ; Draw 'Health: '
    LDA $1800          ; Write string
    INT 0x10

    LDA $6300          ; Write number (unsigned)
    LDB [@player_hp]
    INT 0x05


;;;;;;;;;;;;;;;;;;; Score
    LDA $1500          ; Set cursor
    LDX #61
    LDY #7
    INT 0x10

    LDELM @str_score   ; Draw 'Score: '
    LDA $1800          ; write string
```

```
    INT 0x10

    LDA $6300          ; print number
    LDB [@player_score]
    INT 0x05


;;;;;;;;;;;;;;;;;;; Time
    LDA $1500          ; set cursor
    LDX #61
    LDY #8
    INT 0x10

    LDELM @str_time    ; Draw 'Time: '
    LDA $1800          ; write string
    INT 0x10

    LDA $6300          ; print number
    LDB [@game_time]
    INT 0x05

    RET

str_name:
    .bytes "Name: ", 0

str_str:
    .bytes "STR: ", 0

str_hp:
    .bytes "HP: ", 0

str_time:
    .bytes "T: ",0

str_score:
    .bytes "Score: ", 0
```

Finally, there's one more little change we need to make. Remember that the old `robot` game was 40×25? Let's head over to the `get_input:` function and just change where the player is able to move.

```
move_right:
    LDAL [@PX]
    INC AL
    CMP AL, #59
    JNC @mr_ok              ; AL < 59 (since we're in mode 6 for this game).
    LDAL #58
mr_ok:
```

```
    STAL [@PX]
    RET
```

MVP 1: Base game

Now that we have status drawing nicely, let's remove the draw_robot function. We will keep the stub, but we won't draw the robot or check for collision. You can also remove move_robots and change it to move_mobs (add a stub; you can make all these changes to your own copy). What's left is a simple game where you can move the player around. But nothing interesting happens, so let's increment game time every tick; add a CALL @inc_game_time (or similar) to the game loop (at the end):

```
inc_game_time:
    LDA [@game_time]
    INC A
    STA [@game_time]
    RET
```

We'll also add a way for the player to quit out, since we changed the game loop. Change quit_game to this:

```
quit_game:
    ; CLS
    LDAH $10
    INT $10

    ; Print "QUIT GAME"
    LDBLX @msg_quit
    LDAH $66
    INT $05

    POP ELM ; destroy return address of CALL from main loop
    RET     ; exit program.
```

We now have, essentially, a 'game'; the longer you play it, the higher your score. The top score is 65,535 – after that, it rolls over.

- Source code so far: Rogueima I MVP 1

Variables

To proceed from this point we need to talk about variables and data structres. First we will talk a bit about NVAR, and why it's no good here. Then we will introduce structs and INSQUE.

NVAR stands for named variable system. It is the system BASIC uses to store variables. It stores data

starting at $FFFF and grows downwards towards program space. This lets you efficiently use the space in bank 0 to separate code and data. Note that there are no memory protections here; it's up to you not to accidentally over-write your code or data. To do this, check PATB_VARTOP ($01EEE4; 3 bytes). This points to the top of variable storage. If you want to reserve 4k of memory for variables for example, don't add more variables if PATB_VARTOP falls below $F000 ($F000 to $FFFF is 4k). For our purposes we will ignore this, since our code and data is not expected to be more than a few kilobytes. We should have plenty of room to spare.

> **So,** Using NVAR is very cumbersome in assembly because it requires the use of strings to set and get variable names and data.

Should you use NVAR?

In a previous version of this tutorial, I wrote several examples that outlined how to do parallel arrays in assembly language using NVAR. It took hundreds of lines of code and was very clunky. After writing the section on structs and INSQUE, I realized that the code had no value, not even for educational purposes. NVAR is just not useful for assembly programming because it only understands strings, even when storing integers.

To be honest I don't even know if the old code worked. It was getting huge and was nowhere near finished. Ultimately, there's nothing “wrong” with NVAR, but it's designed to be a part of Stellar BASIC, and not to be used as a general purpose variable storage library. For assembly, we should focus on structs and INSQUE instead.

Game Objects using Structures

Structures are C's `struct` done by hand. They are fixed (or variable length) data *structures,* hence the name. Structures are easy because they are mechanical and quick to program, but they are also difficult because you need to re-write the CRUD (create-read-update-destroy) functions for every structure. This is where programming in a language like C makes things faster. Yet, a little work never hurt anyone, and at runtime, nothing is faster than a fixed width assembly struct.

Let's dive right in. First, we define the structure itself, which is the location of the data in the record. What we'll do here is just define every kind of thing we need to know about an object:

```
    ; Object structure
.equ OBJ_ID     0       ; 2 bytes, max 64k objects
.equ OBJ_TYPE   2       ; 2 bytes; ex. 0 = nothing, 1 = wall, 2=gold, etc.
.equ OBJ_X      4       ; 2 bytes; x position on map
.equ OBJ_Y      6       ; 2 bytes; x position on map
.equ OBJ_GLYPH  8       ; 1 byte; what it looks like (char)
.equ OBJ_VIS    9       ; 1 byte; what it looks like on the map (char)
.equ OBJ_NAME   10      ; 16 bytes + 1 zero (16 max len str)
.equ OBJ_DATA1  27      ; 2 bytes (data 1, ex. gold value)
.equ OBJ_DATA2  29      ; 2 bytes (data 1, ex. gold value)
.equ OBJ_LEN    31      ; record length for an OBJ.
```

This gives us everything we need to know to access an object once we find it.

While we're at it, we can temporarily use the same structure for monster data. Health and Strength can be kept in DATA1 and level/other can be kept in DATA2.

Since we are combining objects and monsters in one structure (this is where the term `MOB` or `mobile object` comes from), we can calculate how many monsters/objects we will be able to have in the game. If we use Bank 3 for this data, we have 64k, enough to hold over 2,000 objects at 31 bytes per record.

CRUD

For any struct we need Create, Read, Update and Destroy accessor functions. However, we will not design functions like that just yet; instead we will look at an example first, to understand the concept.

Example Create

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Create object
;;
;; Returns the ID of a free object in A.
;; A = 0 if no free space.
;;
;;
.equ OBJ_STRUCT_BASE   $F000 ; memory location of data structure
.equ OBJ_STRUCT_END    $FFFF ; End of valid memory

create_obj:
    PUSH X
    PUSH FLD

    ; Variable set-up: 3 register variables
    :   X: stores current record ID
    ; ELM: Address of current record
    : FLD: stores last valid memory address for an object
    LDX #0               ; object ID counter
    LDELM @OBJ_STRUCT_BASE  ; object data base
    LDFLD @OBJ_STRUCT_END
    SUB FLD, @OBJ_LEN

create_obj_loop:
    ADD ELM, @OBJ_TYPE   ; Point to object type
    LDA [ELM]            ; Get object type
    SUB ELM, @OBJ_TYPE   ; Return to start-of-struct
    CMP A, #0            ; Is it an unused object (TYPE=0)?
    JZ @create_obj_found ; Yes, we found an empty object.
```

```
    INC X                ; Point ID to next record
    ADD ELM, @OBJ_LEN    ; Point to next record's memory address
    CMP ELM, FLD         ; Do we have space to write a record here?
    JC @create_obj_not_found  ; Yes. That means we didn't find a free ID.
    JMP @create_obj_loop ; Try again with new ELM/ID no.

create_obj_found:
    ; Return free ID in A
    ; ELM already points to the free object address.
    MOV A, X
    POP FLD
    POP X
    RET

create_obj_not_found:
    MOV A, #0    ; 0 = no free space
    POP FLD
    POP X
    RET
```

This code is very simple. It walks the struct and if it finds an object with no type, it returns it's ID in A and the address of the free struct memory in ELM.

Here is an example of what the initialization subroutine might look like:

```
;;;;;;;;;;;;;;;;;;;;;;;;
;; Init object structure
;; input/output: none
;;
;; Clears all objects and ensures their ID is set.
;;
;;;;;;;;;;;;;;;;;;;;;;;;
init_struct_obj:
    PUSH A
    PUSH X
    PUSH ELM
    PUSH FLD
    ; Variable set-up: 4 register variables
    ; ELM: Address of Record
    ; A: stores default type (will be 0)
    : X: stores current record ID
    : FLD: stores last valid memory address for an object
    LDELM @OBJ_STRUCT_BASE  ; object data base
    LDA #0                  ; type will be set to 0
    LDX #0                  ; object ID counter
    LDFLD @OBJ_STRUCT_END   ; load end of memory
    SUB FLD, @OBJ_LEN       ; reserve space for at least one object

init_struct_obj_loop:
    ADD ELM, @OBJ_ID     ; point to ID
    STX [ELM]            ; Store ID
```

```
    SUB ELM, @OBJ_ID     ; return to start-of-record
    ADD ELM, @OBJ_TYPE   ; point to TYPE
    STA [ELM]            ; Store TYPE
    SUB ELM, @OBJ_TYPE   ; return to start-of-record
    INC X                ; Point ID to next record
    ADD ELM, @OBJ_LEN    ; Point to next record's memory address

    ; check if next record is valid
    CMP ELM, FLD         ; Does this record have enough space (is it valid?)
    JNC @init_struct_obj_loop    ; Yes, set it up (ELM >= FLD = set carry).
    ; finished initializing.
    POP FLD
    POP ELM
    POP X
    POP A
    RET
```

NVAR or Struct?

Each approach has benefits and drawbacks. The NVAR system is useful for dynamic allocation. This can be a huge win. It's an obvious choice for string-based hash tables. But since you have to write your own accessors to use NVAR anyways, you might as well use structs. A custom struct will always be smaller and faster than using NVAR.

But there is another big reason to use structs; it allows you to make lists with INSQUE.

INSQUE, REMQUE, SCANQUE

INSQUE deals with linked lists. It is a simple enough concept; each node has a forward link and a backward link, and the head is a data-less sentinel node at the start of variable memory. As it turns out, this makes programming data structures very easy, so we will use this final method for our game.

To start, we must create a HEAD node. This is just a six byte sentinel node that points to itself:

```
  LDBLX $A000
  LDELM $A000
  STBLX [ELM]   ; Set FLINK to point to itself.
  ADD BLX, 3
  STBLX [ELM]   ; Set BLINK to point to itself.
```

```
  $A000: 00 0A 00 00 0A 00 <user data follows>
```

Setting FLINK (forward link) and BLINK (backward link) to the same address creates the HEAD node. The data at +6 is considered user data. When used with a struct this adds six bytes to the total struct. We now define our struct like this:

```
; Object structure (offset includes 6 byte INSQUE header)
.equ OBJ_FLINK  0        ; forward-link (managed by INSQUE)
.equ OBJ_BLINK  3        ; back-link (managed by INSQUE)
.equ OBJ_ID     6        ; 2 bytes, max 64k objects
.equ OBJ_TYPE   8        ; 2 bytes; ex. 0 = nothing, 1 = wall, 2=gold, etc.
.equ OBJ_X      10       ; 2 bytes; x position on map
.equ OBJ_Y      12       ; 2 bytes; x position on map
.equ OBJ_GLYPH  14       ; 1 byte; what it looks like (char)
.equ OBJ_VIS    15       ; 1 byte; what it looks like on the map (char)
.equ OBJ_NAME   16       ; 16 bytes + 1 zero (16 max len str)
.equ OBJ_DATA1  33       ; 2 bytes (data 1, ex. gold value)
.equ OBJ_DATA2  35       ; 2 bytes (data 2, reserved)

.equ OBJS_BASE  $030000  ; objects data starts here
.equ OBJS_END   $03FFFF  ; use all of bank 3 (over 1,700 objects at 37
bytes/obj.)
.equ OBJ_LEN    37       ; record length for an object record including 6
byte header
```

The original 31 byte STRUCT is now 37 bytes in total length. What do we get for using INSQUE instead of a flat structure? There are many useful advantages.

Adding and Deleting nodes

You can easily add a new node anywhere in the list without moving memory by calling INSQUE to insert a new node after an existing node. Inserting after the head node inserts at the front of the list:

```
  .equ HEAD_NODE $A000
  .equ HEAD_BLINK $A003
  ; Insert as first node (after head node).
  INSQUE ELM, @HEAD_NODE     ; Insert after head node
```

```
  ; Append as last node (after last node)
  LDBLX @HEAD_NODE
  LDBLX [BLX+3]    ; get HEAD.BLINK (last node)
  INSQUE ELM, BLX  ; Insert after last node (append to list)
```

INSQUE modifies the HEAD node and any other nodes it needs to in order to link all the nodes together in a circular list. These records now form a linked list which you can traverse by looking at the first six bytes of each node.

```
  Byte 0: 24 bit pointer to next node
  Byte 3: 24 bit pointer to previous node
  Byte 6: data
```

To delete such a node,

```
  REMQUE ELM             ; Where ELM points to any node.
```

This will delete the node at ELM by unlinking it from the list; the node at it's FLINK will be connected to the node at it's BLINK. The data isn't changed, but it will no longer appear in the list's chain of links.

Sorting a list

You can easily sort a list by unlinking an 'B' object in ELM and adding it after an 'A' object with:

```
  REMQUE ELM      ; unlink node A (at ELM) from the list.
  INSQUE ELM, FLD ; insert node A (at ELM) after B (at FLD).
```

Memory Management

INSQUE offers several advantages in terms of memory management. In any struct-based system system you can traverse the list of objects and check for some field that indicates the object is unused. Such as id=0 or an `alive` flag. But with INSQUE you can use SCANQUE to find this space very quickly.

```
  LDA $0000                ; Object ID, for example
  SCANQUE ELM, A, @OBJ_ID  ; Find first $0000 at index OBJ_ID
```

Pointing to the start of the list is just `LDELM [@OBJS_BASE]`. This loads the address of the first object on the list.

Initialization is just like a flat struct; point to the field you need to configure to show an object is empty and set that field for every object. Then finding free space is a matter of looking for an object which is not in use.

```
init_objects:
    ; Write blank HEAD node at @OBJS_BASE (this clears the whole list)
    LDELM @OBJS_BASE        ; ELM = node ptr
    LDFLD @OBJS_BASE        ; FLD = FLINK/BLINK ptr
    STELM [FLD]             ; write FLINK and advance to BLINK
    ADD ELM, 3
    STELM [FLD]             ; write BLINK (and leave FLD pointed at
HEAD.BLINK)

    ADD ELM, #6             ; advance past head to first data node space.

    LDGLK @OBJS_END
    SUB GLK, @OBJ_LEN
    INC GLK                 ; GLK now equals first invalid address

init_obj_loop:
    ; 1. test if there's enough memory for another object
    CMP GLK, ELM            ; Will we overflow memory if we initialize an
object here?
```

```
    JNC @init_obj_done      ; Yes, so exit (ELM >= GLK == CF).

    ; 2. initialize this object space
    LDI @OBJ_ID
    STB [ELM + I]           ; Write id = 0 for this record
    INSQUE ELM, [FLD]       ; FLD is a ptr to HEAD.BLINK; Append to list

    ; 3. loop
    ADD ELM, @OBJ_LEN       ; point to next object
    JMP @init_obj_loop      ; loop

init_obj_done:
    LDELM [@OBJS_BASE]      ; Load HEAD.FLINK (Point to first object on
list)
    RET
```

This is just 17 lines of code; vastly smaller than the NVAR init, get and set code – and those functions didn't have any memory overflow checks.

If your records are not of equal length (or if you just prefer to unlink them when you remove them) you can perform garbage collection by copying each record to a new space, using INSQUE to create new records. Then either change the OBJ_BASE pointer to the new object data, or copy the objects back to the original location.

Representing Items and Monsters in the Game World

Adding objects and monsters to the map becomes easy. We simply traverse the list of all objects and display anything with an X and a Y coordinate.

Here is the new create_gold function using structs and INSQUE:

```
create_gold:
    ; Find a free space in the object list
    LDA #0
    LDELM [@OBJS_BASE]  ; start scanning at HEAD.FLINK (i.e. the first
object)
    LDI @OBJ_ID
    SCANQUE ELM, A, I
    JNZ @create_gold_oom

    ; Since we're using a sentinel node, if the sentinel is a match it means
there was no valid node.
    CMP ELM, @OBJS_BASE
    JZ @create_gold_oom

    LDA [@obj_ids]
    LDI @OBJ_ID
    STA [ELM+I]
```

```
    INC A
    STA [@obj_ids]

    LDA #1              ; obj type 1 = gold
    LDI @OBJ_TYPE
    STA [ELM+I]

    ;; Now that we'e saved type, lets get its X, Y and amount data.
    LDAH $00     ; 16 bit random number
    INT 0x13     ; math servics
    MOD B, #100
    INC B       ; amount is now 1 to 100.
    LDI @OBJ_DATA1
    STB [ELM+I]

    LDAH $00     ; 16 bit random number
    INT 0x13
    MOD B, #58
    INC B        ; random number 1 to 58
    LDI @OBJ_X
    STB [ELM+I]

    LDAH $00     ; 16 bit random number
    INT 0x13
    MOD B, #23
    INC B        ; random number 1-23
    LDI @OBJ_Y
    STB [ELM+I]

    ; Add GLYPH and VIS.
    LDAL #'$'
    LDI @OBJ_GLYPH
    STAL [ELM+I]
    LDI @OBJ_VIS
    STAL [ELM+I]

    CLC         ; no error
    RET         ; return

create_gold_oom:
    SEC     ; set carry on error
    RET
```

Putting an object on the map becomes a simple matter of adding it to the list of objects.

Testing item generation

Let's test our new item and object system by adding `create_gold` as a command (bound to g). We'll

also add arrow key movement.

```
gi_check_keys:
    CMP AL, #'H'
    JZ @move_left
    CMP AL, #'J'
    JZ @move_down
    CMP AL, #'K'
    JZ @move_up
    CMP AL, #'L'
    JZ @move_right

    CMP AL, #128
    JZ @move_left
    CMP AL, #129
    JZ @move_down
    CMP AL, #130
    JZ @move_up
    CMP AL, #131
    JZ @move_right

    CMP AL, #'G'
    JZ @create_gold
    CMP AL, #'Q'
    JZ @quit_game

    RET
```

In addition, add a call in `draw_map:`

```
  ...
  CALL @draw_world
  CALL @draw_items    ; <-- add this
  CALL @draw_player
  ...
```

Now, we need to create the draw_items function. All it does is loop through every item, find it's X, Y and glyph, and draw it:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Draw all the items that are placed on the map.
;;
draw_items:
    LDELM @OBJS_BASE       ; Start at first object

draw_items_loop:
    LDELM [ELM]            ; Traverse to next object
    CMP ELM, @OBJS_BASE
```

```
    JZ @draw_items_done    ; End loop if we returned to HEAD node.

    LDI @OBJ_ID
    LDA [ELM+I]      ; get object ID.
    JZ @draw_items_loop    ; Skip non-initialized/empty objects

    LDI @OBJ_TYPE
    LDA [ELM+I]
    CMP A, #1
    JNZ @draw_items_loop         ; not amn item. skip
    ; Does this item have an X and Y position?
    LDI @OBJ_X
    LDX [ELM+I]
    JZ @draw_items_loop
    LDI @OBJ_Y
    LDY [ELM+I]
    JZ @draw_items_loop

    ; Get the VIS (visible glyph) of the object.
    LDI @OBJ_VIS
    LDBL [ELM+I]

    LDAH $11        ; write char BL at XL, YL
    INT $10
    JMP @draw_items_loop ; Next!

draw_items_done:
    RET
```

Finally, add a call to `init_items` at the end of `init_game` in rogueima.sda:

```
init_game:
    ...
    LDA #0
    STA [@game_time]
    STA [@player_score]

    CALL @init_objects      ; Initialize the game objects list.

    RET
```

With all this, pressing g will create piles of gold which appear on the map!

Picking up gold

It makes sense that the player is able to pick up gold. First, let's detect if there is any object on the

square the player is standing on. If there is, we can print a small message.

First let's add a pick up command to the command list:

```
gi_check_keys:

    ....
        CMP AL, #'G'
    JZ @create_gold
    CMP AL, #'Q'
    JZ @quit_game
    CMP AL, @','        // pick things up using comma
    JZ @pick_up_items

    RET
```

Next, let's add gold that the player picks up to his inventory and remove such gold from the objects list.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Pick up items
;; Picks up the first pile of gold where the player is standing.
;;
pick_up_items:
    LDELM @OBJS_BASE         ; Start at first object

pick_up_items_loop:
    LDELM [ELM]              ; Traverse to next object
    CMP ELM, @OBJS_BASE
    JZ @pick_up_items_done   ; End loop if we returned to HEAD node.

    LDI @OBJ_ID
    LDA [ELM+I]              ; get object ID.
    JZ @pick_up_items_loop   ; Skip non-initialized/empty objects

    ; Does this item match the player's location?
    LDI @OBJ_X
    LDX [ELM+I]              ; object x

    LDIL [@PX]               ; player x
    CMP IL, XL
    JNZ @pick_up_items_loop  ; No match, keep looking.

    LDI @OBJ_Y
    LDY [ELM+I]              ; objecy y

    LDJL [@PY]               ; player y
    CMP JL, YL
    JNZ @pick_up_items_loop  ; No match, keep looking.
```

```
    ; Oh, there's an item here!
    LDI @OBJ_TYPE
    LDA [ELM+I]
    CMP A, #1                ; is it gold?
    JNZ @pick_up_items_loop  ; No, keep looking.

    ; How much gold is it?
    LDI @OBJ_DATA1
    LDA [ELM+I]
    LDB [@player_score]
    ADD B, A                 ; Add one point for every gold.
    STB [@player_score]      ; Save score.

    LDA #0
    LDI @OBJ_ID              ; Remove the gold object from the game
    STA [ELM+I]              ; by zeroing it's ID (thats all!)

pick_up_items_done:
    RET
```

Now you can have fun running around picking up treasure. We have already come so far!

Designing an 8 bit rogue

It's worth taking a moment to pause here and look, really look, at the data struct-ures we have so far and think carefully about how we want to proceed. Ideally what we **want** is an easy pointer from where we stand to a list of items. That's easy to do in a flat structure or with an insque. Let's say we want to have a map as data. Each square takes up space. But at 37 bytes a record, a 256×256 world map will take more space than we have total RAM. How did games like Ultima 4, 5 and 6 have such large world maps and so many objects? There are a few answers.

First, be realistic. For Ultima 6, it was intended for computers with more memory. If you stored the Ultima 6 world map as an array of bytes, it would be 1 megabyte in size. The dungeon areas were just as large. Ultima 4 and 5 had a 256×256 world map – just large enough to be byte-addressable. But since it couldn't fit in memory how did they do it? The answer is that they didn't load it into memory all at once. They kept it on disk or on the cartridge (or CD) and read only the parts they needed to display.

A roguelike game has around as many tiles and data, more or less, or at least must be able to support a similar amount of data. It is worth noting however that ROGUE, HACK as they were written are much more suited to a 16 bit environment. They didn't really exist in the 8 bit world; although there were some notable ports, no full version of the game ever made it to systems like the C64. Telengard was a decent try, but the microcomputer era was much more well known for games like Sword of Fargoal and the Temple of Apshai.

That being said, an 8-bit style ROGUE is possible, with three main constraints:

- 1. The map must be byte-addressable.

- 2. The map data and all objects must fit in one bank to be plausable.
- 3. 80 column mode was not common on some microcomputers; we use it, but in monochrome.

Four, even; We should take care not to allow more data than would feasably be able to be stored on a floppy disk to exist in the system as a total (about 160k). The game should be designed to fit in that space as a total package. If we want more data, we need to come up with an excuse, such as we're bit-packing the map into 4 bits (and only allow 32 different tile types). Or we're using compression.

At any rate, we will not need to worry about larger data sets until we do a world map (which I will probably skip until Rogueima II).

Rendering the Maps

Now let's work on displaying a map. To display a map, we will need three important things:

1. A source of truth for the map
2. A projection of that map onto the play area
3. Game logic that provides map interaction

The first and most important thing we need is the map itself. Here is our first world map:

```
map1_id:
    .bytes 1
map1_name:
    .bytes "START   ", 0       ; 8 characters and zero terminated
map1_up:
    .bytes 5, 5
map1_down:
    .bytes 22, 15
map1_dim:
    .bytes 80, 40
map1_data:
    .bytes
"################################################################################
#####" ; 0
    .bytes "#         #
#" ; 1
    .bytes "#         #
#" ; 2
    .bytes "#         #
#" ; 3
    .bytes "#         #
#" ; 4
    .bytes "#    @    +
#" ; 5
    .bytes "#         #
#" ; 6
    .bytes "#         #                    #########
#" ; 7
```

```
    .bytes "#         #                         #       #
#" ; 8
    .bytes "#         #                         #       +
#" ; 9
    .bytes "#####+#####                         #       #
#" ; 10
    .bytes "#                                   #       #
#" ; 11
    .bytes "#                                   #########
#" ; 12
    .bytes "#
#" ; 13
    .bytes "#
#" ; 14
    .bytes "#                     >
#" ; 15
    .bytes "#
#" ; 16
    .bytes "#
#" ; 17
    .bytes "#
#" ; 18
    .bytes "#
#" ; 19
    .bytes "#
#" ; 20
    .bytes "#
#" ; 21
    .bytes "#
#" ; 22
    .bytes "#
#" ; 23
    .bytes "#
#" ; 24
    .bytes "#
#" ; 25
    .bytes "#
#" ; 26
    .bytes "#
#" ; 27
    .bytes "#
#" ; 28
    .bytes "#
#" ; 29
    .bytes "#
#" ; 30
    .bytes "#
#######+#############+#############+#######" ; 31
    .bytes "#                                     #             #
#            #" ; 32
```

```
    .bytes "#                                          #             #
#             #" ; 33
    .bytes "#                                          #             #
#             #" ; 34
    .bytes "#                                          #             #
#             #" ; 35
    .bytes "#                                          #             #
#             #" ; 36
    .bytes "#                                          #             #
#             #" ; 37
    .bytes "#                                          #             #
#             #" ; 38
    .bytes
"##############################################################################
#####" ; 39
```

This map was constructed to be very simple and allow us to test:

- Having a map in the first place (it works!)
- Reading and displaying the map
- Map interaction (Opening and closing doors, walls blocking movement and vision)

(Note, the @ and > are added for visual reference. They will be ignored by the game).

Projecting the Map

To project the map we need to calculate the area of the map around the player that will fit in the play area. We will draw that box in the play area. The first step is to gather all the data. We have `map1_dim`, now we need `disp_dim` (display dimensions).

```
  disp_dim:
      .bytes #58, #23
```

This represents the play area which is 58×23 wide.

Second, we need to re-bound the player's movement to the map, and not to the play area. Change the movement code to use `@map1_dim:`

```
move_left:
    LDAL [@PX]
    JZ @ml_done
    DEC AL ; was not zero; can move left.
    STAL [@PX]
ml_done:
    RET

move_down:
    LDAL [@PY]
```

```
    INC AL
    LDB [@map1_dim]
    DEC BH
    CMP AL, BH              ; should be set to MAP_HEIGHT
    JNC @md_ok              ; AL < map_height (not <=!!)
    MOV AL, BH
md_ok:
    STAL [@PY]
    RET

move_up:
    LDAL [@PY]
    JZ @mu_done
    DEC AL
    STAL [@PY]
mu_done:
    RET

move_right:
    LDAL [@PX]
    INC AL
    LDB [@map1_dim]
    DEC BL
    CMP AL, BL              ; Should be set to MAP WIDTH.
    JNC @mr_ok              ; AL < map width
    MOV AL, BL
mr_ok:
    STAL [@PX]
    RET
```

This will bound the player's PX and PY to a co-ordinate which also exists on the map. Next, we need to *project* the area around the player to the play area.

Java method from NetWhack

This algorithm is something I came up with many years ago for the Java game NetWhack. I'll show you the render function in Java first, then show the assembly version.

```
// Pre: lvl and pc are not null.
public void redraw_screen(Level lvl, Player pc) {
    // Take the position of the PC on the map.
    // Draw a screen-sized box around the PC with the PC in the middle.
    int from_x = pc.xpos - vs.screen.maxcols / 2;
    int from_y = pc.ypos + 2 - vs.screen.maxrows / 2; // Add 2 for message
line.
    int to_x = pc.xpos + vs.screen.maxcols / 2;
    int to_y = pc.ypos - 3 + vs.screen.maxrows / 2;  // Subtract 3 for stats
```

```
line.

    // These two if statements are a kind of kludge.
    // the problem with largemaps could be fixed another way.
    if ((to_x - from_x) >= vs.screen.maxcols) {
        to_x--;
        from_x++;
    }
    if ((to_y - from_y) >= vs.screen.maxrows) {
        to_y--;
        from_y++;
    }

    // Move the box so that the left edge of the box is within map bounds.
    while (from_x < 0) {
        from_x++;
        if (to_x < lvl.xmax) {
            to_x++; // dont increase to_x beyond map boundaries.
        }
    }

    // Move the box so that the right edge of the box is within map bounds.
    while (to_x >= lvl.xmax) {
        to_x--;
        if (from_x > 0) {
            from_x--; // dont decrease to_x beyond map boundaries.
        }
    }

    // Move the box so that the top edge of the box is within map bounds.
    while (from_y < 0) {
        from_y++;
        if (to_y < lvl.ymax) {
            to_y++; // dont increase to_y beyond map boundaries.
        }
    }

    // Move the box so that the bottom edge of the box is within map bounds.
    while (to_y >= lvl.ymax) {
        to_y--;
        if (from_y > 0) {
            from_y--; // dont decrease to_y beyond map boundaries.
        }
    }

    // At this point we have a box which is guranteed to be equal to or
    // smaller than the size of the screen, and also guranteed to be within
    // map bounds.

    // So, we should center this box on the drawable area of the screen.
    int at_x, at_y;
```

```
    if ((to_x - from_x) < vs.screen.maxcols) {
        at_x = (vs.screen.maxcols - Math.abs(to_x - from_x)) / 2;
    } else {
        at_x = 0;
    }

    if ((to_y - from_y) < (vs.screen.maxrows - 3 - 3)) {
        at_y = (vs.screen.maxrows - Math.abs(to_y - from_y)) / 2;
    } else {
        at_y = 2;
    }

    // double check screen heights...
    //if ((to_y-from_y)>=(vs.maxcols-3)) to_y--;

    bufrenderlevel(lvl, at_x, at_y, from_x, from_y, to_x, to_y);
}
```

The above Java code uses an *iterative* approach. The function we will use below is a *closed calculation*. They execute on the same basic idea and should produce identical results. I am undecided which algorithm I like better.

draw_world

```
draw_world:
    LDT [@disp_dim]    ; TL = play_width,  TH = play_height
    LDK [@map1_dim]    ; KL = map_width,   KH = map_height

    LDX #0
    LDY #0
    LDXL [@PX]
    LDYL [@PY]
    MOV I, X
    MOV J, Y

    ; Clamp I to [0, KL - TL]
    MOV AL, TL
    DEC AL
    SHR AL, #1          ; AL = (TL-1)/2 = centered offset
    CMP IL, AL
    JC @dw_i_lo_ok
    MOV IL, AL
dw_i_lo_ok:
    SUB IL, AL

    MOV AL, KL
    SUB AL, TL          ; AL = max_I_start
```

```
    CMP IL, AL
    JNC @dw_i_hi_ok
    MOV IL, AL
dw_i_hi_ok:

    ; Clamp J to [0, KH - TH]
    MOV AL, TH
    DEC AL
    SHR AL, #1
    CMP JL, AL
    JC @dw_j_lo_ok
    MOV JL, AL
dw_j_lo_ok:
    SUB JL, AL

    MOV AL, KH
    SUB AL, TH
    CMP JL, AL
    JNC @dw_j_hi_ok
    MOV JL, AL
dw_j_hi_ok:

    ; put player screen position into C, D
    LDAL [@PX]
    SUB AL, IL
    INC AL
    LDAH #0
    MOV C, A

    LDAL [@PY]
    SUB AL, JL
    INC AL
    LDAH #0
    MOV D, A

    ; render loop
    LDX #1
    LDY #1
    MOV G, I

dw_x_loop:
    LDELM @map1_data
    ADD ELM, I
    MOV Z, J
    MUL Z, KL
    ADD ELM, Z
    LDBL [ELM]

    LDAH $11
    INT 0x10
```

```
    INC I
    INC X
    CMP TL, XL          ; continue while TL >= X
    JC @dw_x_loop

    LDX #1
    MOV I, G
    INC J
    INC Y
    CMP TH, YL
    JC @dw_x_loop       ; same trick on the outer

    ; Draw player on top
    LDAH $11
    LDBL #'@'
    MOV X, C
    MOV Y, D
    INT 0x10
    RET
```

draw_map

Finally let's change draw_map: to this:

```
draw_map:
    LDAH $51        ; VSTOP
    INT 0x18

    CALL @draw_borders
    CALL @draw_world
    ;CALL @draw_items
    ;CALL @draw_mobs
    ;CALL @draw_player
    CALL @draw_stats
    CALL @draw_msgs

    LDAH $50        ; VSTART
    INT 0x18

    RET
```

The draw_item, draw_player and draw_monster funtions need to be commented out until we calculate their location on the map. This adjustment number will be saved during draw_world and used to calculate the projection offset for items, monsters and so on.

Projecting Items

We also need to change items so they draw based on the map location. The solution is to save the camera origin during draw_world so we can use it later to draw anything else.

Create the variables:

```
  cam_origin_x:
      .bytes 0,0
  cam_origin_y:
      .bytes 0,0
```

Save them in draw_world after clamping I and J:

```
dw_j_lo_ok:
    SUB JL, AL

    MOV AL, KH
    SUB AL, TH
    CMP JL, AL
    JNC @dw_j_hi_ok
    MOV JL, AL

dw_j_hi_ok:
    STIL [@cam_origin_x]        ; Save computed camera origin
    STJL [@cam_origin_y]
```

Now, we can use these coordinates to rework the draw_items function to make items appear on the map in their proper location:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Draw all the items that are placed on the map.
;;
draw_items:
    LDD [@cam_origin]       ; DL = I_start, DH = J_start
    LDT [@disp_dim]         ; TL, TH for upper-bound additions
    LDELM @OBJS_BASE
draw_items_loop:
    LDELM [ELM]
    CMP ELM, @OBJS_BASE
    JZ @draw_items_done

    LDI @OBJ_ID
    LDA [ELM+I]
    JZ @draw_items_loop

    LDI @OBJ_X
```

```
    LDX [ELM+I]
    JZ @draw_items_loop
    LDI @OBJ_Y
    LDY [ELM+I]
    JZ @draw_items_loop

    ; is (XL, YL) inside the viewport?
    CMP XL, DL
    JNC @draw_items_loop    ; XL < I_start (off-screen left)
    MOV AL, DL
    ADD AL, TL
    CMP XL, AL
    JC  @draw_items_loop    ; XL >= I_start + TL (off-screen right)

    CMP YL, DH
    JNC @draw_items_loop    ; YL < J_start (off-screen up)
    MOV AL, DH
    ADD AL, TH
    CMP YL, AL
    JC  @draw_items_loop    ; YL >= J_start + TH (off-screen down)

    ; xlate map to screen
    SUB XL, DL
    ADD XL, #1              ; play_x_offset
    SUB YL, DH
    ADD YL, #1              ; play_y_offset

    ; draw
    LDI @OBJ_VIS
    LDBL [ELM+I]
    LDAH $11
    INT $10
    JMP @draw_items_loop
draw_items_done:
    RET
```

Now, finally, you will notice that when the player steps over an item the item draws over the player. This is because the player is being drawn in draw_world! Refactor the player drawing code back into draw_player and uncomment the CALL @draw_player line in the game loop. You can now remove the section “player screen locations into C and D” from draw_world:

```
draw_player:
    LDBL #'@'
    LDXL [@PX]
    LDYL [@PY]
    LDIL [@cam_origin_x]
    LDJL [@cam_origin_y]
    SUB XL, IL              ; XL = PX - I_start
```

```
    ADD XL, #1              ; play area x offset (same as draw_world)
    SUB YL, JL              ; YL = PY - J_start
    ADD YL, #1              ; play area y offset (it's because of the screen
border).
    LDAH $11
    INT $10
    RET
```

Map Interaction

Next let's add a little map interaction. Players cannot step onto walls (=) or closed doors (+). Furthermore, we can let players can open or close doors with the O key (which then allows them to choose a direction).

We will need a few functions;

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; get_glyph(X, Y) -> AL
;;
;; Returns the map glyph at column X, row Y.
;; Caller is responsible for X, Y being within map bounds.
;;
;; Inputs:   X = column, Y = row
;; Output:   AL = glyph byte (e.g. '#', '.', '+', ' ')
;; Clobbers: ELM, K, Z
;;
get_glyph:
    PUSH ELM
    PUSH K
    PUSH Z
    LDK [@map1_dim]      ; KL = map_width
    MOV Z, Y             ; Z = Y
    MUL Z, KL            ; Z = Y * map_width
    ADD Z, X             ; Z = Y * map_width + X
    LDELM @map1_data
    ADD ELM, Z           ; ELM -> map[Y][X]
    LDAL [ELM]           ; AL = glyph
    POP Z
    POP K
    POP ELM
    RET

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; is_walkable(X, Y)
;;
;; Sets CARRY if this tile is walkable.
;;
;; Non-walkable tiles: = and +
```

```
;;

is_walkable:
    PUSHA
    CALL @get_glyph     ; AL now has glyph.
    CMP AL, #'#'
    JZ @is_walkable_no
    CMP AL, #'+'
    JZ @is_walkable_no

    ;; default: yes.
is_walkable_yes:
    POPA
    SEC
    RET
is_walkable_no:
    POPA
    CLC
    RET
```

These two functions will now let us check to see if the tile is walkable. Inside the player movement section, we can now check to see if the player has landed on a walkable tile. If no, we don't save the player's move:

```
move_left:
    LDAL [@PX]
    JZ @ml_done
    DEC AL                  ; AL = new PX
    LDX #0
    LDY #0
    MOV XL, AL              ; XL = new PX (survives CALL)
    LDYL [@PY]
    CALL @is_walkable
    JNC @ml_done
    STXL [@PX]
ml_done:
    RET

move_right:
    LDAL [@PX]
    INC AL                  ; AL = new PX
    LDB [@map1_dim]         ; BL = map_width
    CMP AL, BL
    JC @mr_done             ; AL >= map_width — out of bounds, abort
    LDX #0
    LDY #0
    MOV XL, AL              ; XL = new PX
    LDYL [@PY]
```

```
    CALL @is_walkable
    JNC @mr_done
    STXL [@PX]
mr_done:
    RET

move_up:
    LDAL [@PY]
    JZ @mu_done
    DEC AL                  ; AL = new PY
    LDX #0
    LDY #0
    LDXL [@PX]
    MOV YL, AL              ; YL = new PY (survives CALL)
    CALL @is_walkable
    JNC @mu_done
    STYL [@PY]
mu_done:
    RET

move_down:
    LDAL [@PY]
    INC AL                  ; AL = new PY
    LDB [@map1_dim]         ; BH = map_height
    CMP AL, BH
    JC @md_done             ; AL >= map_height — out of bounds, abort
    LDX #0
    LDY #0
    LDXL [@PX]
    MOV YL, AL              ; YL = new PY
    CALL @is_walkable
    JNC @md_done
    STYL [@PY]
md_done:
    RET
```

Now you will notice that the move function rejects moves that would put the player on an 'unwalkable' tile. Things are getting spicy!

is_walkable in object creation

Make sure you modify the create_gold function to use is_walkable, so we don't accidentally spawn gold on a wall or closed door. Change the random x,y position from the play screen to a map-aware x,y finder:

```
    LDK [@map_dim]
create_gold_xy:
```

```
    LDAH $00     ; 16 bit random number
    INT 0x13
    MOD B, KL
    MOV X, B

    LDAH $00     ; 16 bit random number
    INT 0x13
    MOD B, KH
    MOV Y, B

    CALL @is_walkable
    JNC @create_gold_xy

    ; store final x,y location
    LDI @OBJ_X
    STX [ELM+I]

    LDI @OBJ_Y
    STY [ELM+I]

    ; Add GLYPH and VIS.
    LDAL #'$'
    LDI @OBJ_GLYPH
    STAL [ELM+I]
    LDI @OBJ_VIS
    STAL [ELM+I]
```

Opening doors

When the player presses O, we have a secondary key input that directionally searches for a door symbol (+ or -). If it finds one, it changes the symbol. This is the method by which doors can be opened and closed.

First change `gi_check_keys:` to add the new command:

```
    ....

    CMP AL, #'O'
    JZ @open_door

    CMP AL, #'G'
    JZ @create_gold
    CMP AL, #'Q'
    JZ @quit_game
    CMP AL, #','        // pick things up using comma
    JZ @pick_up_items
```

```
    RET
```

Next, we need the `open_door:` function:

open_door

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; open_door()
;; This replicates the gi_get_keys pattern
;; but is used to directionally change a door's status.
;;
open_door:
    ; Ask player for key
    LDAH $02    ; Blocking GETKEY -> AL
    INT $10

    ;; Add the player's key as entropy.
    LDAH $02
    INT 0x13

    ; Uppercase: if AL >= 'a' and AL < 'z'+1, subtract 32
    CMP AL, #97
    JNC @od_check_keys      ; AL < 'a', skip
    CMP AL, #123
    JC @od_check_keys       ; AL >= '{', skip
    SUB AL, #32

od_check_keys:
    ; Check direction given
    CMP AL, #'H'
    JZ @open_door_left
    CMP AL, #'J'
    JZ @open_door_down
    CMP AL, #'K'
    JZ @open_door_up
    CMP AL, #'L'
    JZ @open_door_right

    CMP AL, #128
    JZ @open_door_left
    CMP AL, #129
    JZ @open_door_down
    CMP AL, #130
    JZ @open_door_up
    CMP AL, #131
    JZ @open_door_right

    ; Bad direction -- ignore.
```

```
    RET

open_door_up:
    LDI #0
    LDJ #0
    LDK #0
    LDL #1
    JMP @open_door_go

open_door_down:
    LDI #0
    LDJ #1
    LDK #0
    LDL #0
    JMP @open_door_go

open_door_left:
    LDI #0
    LDJ #0
    LDK #1
    LDL #0
    JMP @open_door_go

open_door_right:
    LDI #1
    LDJ #0
    LDK #0
    LDL #0
    JMP @open_door_go

open_door_go:
    LDX #0
    LDY #0
    LDXL [@PX]
    LDYL [@PY]
    ADD XL, IL
    ADD YL, JL
    SUB XL, KL
    SUB YL, LL
    CALL @get_glyph
    CMP AL, #'+'
    JZ @open_door_open_it
    CMP AL, #'-'
    JZ @open_door_close_it

    ;; bad command? ignore.
    RET

open_door_open_it:
    LDAL #'-'
```

```
    CALL @put_glyph
    RET

open_door_close_it:
    LDAL #'+'
    CALL @put_glyph
    RET
```

Note that this function relies on `put_glyph:` – this function is exactly the same as get_glyph except you change the LDAL to a STAL.

put_glyph

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; put_glyph(X, Y, AL)
;;
;; Places the glyph AL at x, y
;; Caller is responsible for X, Y being within map bounds.
;;
;; Inputs:   X = column, Y = row, AL = chr (ex. 'A')
;;
put_glyph:
    PUSH ELM
    PUSH K
    PUSH Z

    LDK [@map1_dim]      ; KL = map_width
    MOV Z, Y             ; Z = Y
    MUL Z, KL            ; Z = Y * map_width
    ADD Z, X             ; Z = Y * map_width + X
    LDELM @map1_data
    ADD ELM, Z           ; ELM -> map[Y][X]
    STAL [ELM]           ; AL = glyph

    POP Z
    POP K
    POP ELM
    RET
```

In the same way we interact with items and the map itself, we can now programatically interact with anything on the map. Our game is almost complete! There's just one more thing we “should” add, a message window.

The Message Window

A message window is an important part of any game that uses text to communicate with the player. In our case, we need to display messages in a small area of the screen and not have them spill over. We can do this by creating a message buffer and manually printing into that buffer as we go.

First let's create the data structures we will need; a message buffer and all the data to work with it:

```
msg_dim_x:
    .bytes #19, #0
msg_dim_y:
    .bytes #13, #0  ; width and height

msg_home_x:
    .bytes #60, #0
msg_home_y:
    .bytes #11, #0  ; X and Y where to start drawing

msg_cursor_x:
    .bytes #0, #0    ; X internal cursor location (start: left side)
msg_cursor_y:
    .bytes #12, #0   ; Y internal cursor location (start: lower row)

msg_data:
    ; 13 rows of 19 bytes (must match msg_dim)
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Let's target movement and the opening of doors as a test of the system:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; String data                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
str_open_where:
    .bytes "> Open where?", 13, 10, 0
str_NORTH:
    .bytes "NORTH", 0
str_EAST:
```

```
    .bytes "EAST", 0
str_SOUTH:
    .bytes "SOUTH", 0
str_WEST:
    .bytes "WEST", 0
str_OPEN_DOOR:
    .bytes " OPEN DOOR ", 0
str_crlf:
    .bytes #13, #10, #0
</armasm>

=== Message window functions
Writing to the message window is very straightforward. You have a print
function, and the window is printed during the draw cycle.

Replace ''CALL @draw_msgs'' with ''CALL @msg_display'' in ''draw_map:' in
''draw.sda''. That will print the messages; now we just have to get from
//print_message// to //msg_display//!

==== print_msg:
Our first stop is the intended use case: print_msg.

<codify armasm>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; print_msg(ELM)
;;
;; ELM points to a string. This function
;; simplifies a "puts" type function by
;; calling the "putc" type function.

print_msg:
    PUSH A                  ; We over-write AL so save register A.
    PUSH ELM                ; Save pointer to string

print_msg_loop:
    LDAL [ELM]              ; Load char and advance ELM.
    INC ELM
    JZ @print_msg_done      ; If we hit a zero, (end of string), then exit.
    CALL @msg_putchar       ; Writes char al at position in msg_cursor
    JMP @print_msg_loop

print_msg_done:
    POP ELM                 ; restore original string pointer
    POP A
    RET
```

This function delegates actually printing to a msg_putchar function. This is a fair divvy of work.

msg_putchar:

The putchar function simply calculates the correct position in the char-buffer and writes the ASCII code. That would be the end of it, but we need to handle special characters. Here we handle CR and LF; you could add other special characters here too.

The divvy here is that this is a dispatcher; if it doesn't print the character, it delegates any special work to a helper function. These are:

- msg_putchar_advance: responsible for advancing the cursor (and all that comes with that)
- msg_do_cr: a simple function, just moves x.
- msg_do_lf: a more in-depth function, linked to msg_scroll:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; msg_putchar(AL)
;; looks at msg_cursor, msg_dim, etc. and writes the character.
;;
msg_putchar:
    ;; We are called with character in AL.
    ;; Find the current cursor position:
    LDX [@msg_cursor_x]
    LDY [@msg_cursor_y]

    ; Is AL a special character?
    CMP AL, #10     ; line feed
    JZ @msg_putchar_lf
    CMP AL, #13     ; carriage return
    JZ @msg_putchar_cr

mpc_check_done:
    ; Not a special character. Print it.
    LDFLD @msg_data
    ADD FLD, X
    LDI [@msg_dim_x]
    MUL Y, I        ; multiply Y by row length to get ptr
    ADD FLD, Y
    STAL [FLD]      ; Store this character in the buffer.

    ; Advance the cursor properly.
    CALL @msg_putchar_advance

msg_putchar_done:
    RET

msg_putchar_lf:
    CALL @msg_do_lf
    JMP @msg_putchar_done

msg_putchar_cr:
```

```
    CALL @msg_do_cr
    JMP @msg_putchar_done
```

msg_putchar_advance:

This is a cursor advance function but it is (mis)-named *putchar_advance* because it is intended only to be called by `msg_putchar`.

The divvy here is that if CR/LF needs to be performed, it calls those functions as helper functions.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Advance the cursor properly.
;; This is a big responsibility!
;; We may need to linefeed the msg window.
;;
msg_putchar_advance:
    PUSHA
    LDX [@msg_cursor_x]
    LDY [@msg_cursor_y]

    INC X     ; Attempt to move cursor.

    ;; Check if we moved past the X-width
    LDI [@msg_dim_x]
    CMP X, I        ; Since X is 0 based, if it is equal to I it's out of
bounds.
    JNZ @mpc_done   ; X is fine, so we're done.

    ; We've moved past the right side, do a CR/LF.
    CALL @msg_do_cr        ; Carriage return
    CALL @msg_do_lf        ; Do a linefeed, will scroll if necessary.

mpc_done:
    STX [@msg_cursor_x]
    STY [@msg_cursor_y]
    POPA
    RET
```

msg_do_cr: and msg_do_lf:

These are the carriage_return and line_feed functions. Essentially, we just increment the cursor location; but in the case of line_feed we may need to scroll the screen.

```
;;;;;;;;;;;;;;;;
;; msg_do_cr
;; this doesn't affect the buffer in any way, just the cursor.
;;
```

```
msg_do_cr:
    LDX #0
    STX [@msg_cursor_x]
    RET

;;;;;;;;;;;;;;
;; msg_do_lf
;; Just lf.
;; Adds a line only if we need to.
msg_do_lf:
    PUSH J
    INC Y
    LDJ [@msg_dim_y]
    CMP Y, J
    JNC @msg_do_lf_done
    CALL @msg_scroll    ; manually scroll window.

msg_do_lf_done:
    STY [@msg_cursor_y]
    POP J
    RET
```

msg_scroll:

If the message buffer needs to be fed a new line, this is the function you would call.

```
;;;;;;;;;;;;;;;;;;;;;;;;
;; msg_scroll
;; Called by msg_do_lf.
;; You probably don't want to call this by itself.
;; Note that it modifies Y, and the caller may depend on this.
msg_scroll:
    PUSH ELM
    PUSH FLD
    CMP Y, 0                ; if Y is zero,
    JZ @msg_scroll_nodec     ; don't dec Y.

    DEC Y       ; We keep the cursor at the same position it was at in the
buffer.
    STY [@msg_cursor_y]

msg_scroll_nodec:
    LDI [@msg_dim_x]    ; get row size
    LDJ [@msg_dim_y]    ; get height
    LDELM @msg_data     ; get start of msg area
    LDFLD @msg_data     ; copy start into FLD
    ADD ELM, I          ; skip one row in ELM (start)
    DEC J               ; don't copy last row (# of bytes)
```

```
    MUL J, I            ; find total bytes to move except last row

msg_scroll_loop:
    LDAL [ELM]
    STAL [FLD]
    INC ELM
    INC FLD
    DEC J
    CMP J, #0
    JNZ @msg_scroll_loop

    LDFLD @msg_data
    LDI [@msg_dim_x]
    LDJ [@msg_dim_y]
    DEC J                ; AL = last row index
    MUL J, I             ; AL = byte offset of last row
    ADD FLD, J

    LDAL $20             ; space
msg_scroll_clear:
    STAL [FLD]
    INC FLD
    DEC I
    JNZ @msg_scroll_clear  ; write a blank line

    POP FLD
    POP ELM
    RET                  ; Buffer has been updated!
```

msg_display:

This is the round trip now, we're done! This is the function that will copy the message buffer onto the screen.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; msg_display
;; Copies the message buffer to the screen at (msg_home_x, msg_home_y).
;; Treats zero bytes as spaces so the all-zeros init looks blank.
;;
msg_display:
    PUSHA
    LDELM @msg_data         ; ELM walks through the buffer

    LDY #0
    LDYL [@msg_home_y]      ; Y = current screen row
    LDJL [@msg_dim_y]       ; JL = rows remaining

mdpy_row:
    LDX #0
```

```
    LDXL [@msg_home_x]      ; X = current screen col (start of row)
    LDIL [@msg_dim_x]       ; IL = cols remaining in this row

mdpy_col:
    LDBL [ELM]              ; BL = char, advance ELM
    INC ELM
    JNZ @mdpy_print
    LDBL #' '               ; map 0 -> space for the renderer
mdpy_print:
    LDAH $11
    INT $10                 ; write char BL at XL, YL

    INC X
    DEC IL
    JNZ @mdpy_col

    INC Y
    DEC JL
    JNZ @mdpy_row

    POPA
    RET
```

Rogueima I: Final Thoughts

There are a lot of things we could add now.

- Random level generation
- The ability to go deeper into the dungeon
- Loading and saving levels
- A win condition (such as getting a certain sword from deep in the dungeon)
- Monsters
- Combat
- Ranged weapons

Each of these things presents an interesting challenge to the programmer, but, these are not strictly game design concerns that depend on SDA-8516 Assembly Language, they are mechanical algorithmic processes that build almost entirely on what you already know. The one interesting thing would be random levels; I will include some level generation functions below taken from Java NetWhack; you can feel free to use them, if you can translate them into assembly!

Improvements and optimizations

Looking at the direction the code took, I would rather have some way of accessing an inventory (a list) on a per-tile basis to determine things about that tile. However, no matter how you try and arrange the data, you will never be able to have an index into a large map; it just requires too much memory. A head node for a world map (256×256) would require 6 banks of memory. That just isn't going to

work on a machine with less memory!

The first solution is to only store the visible tiles in memory. Since the play window is 58×23, what if we only stored the current screen (and a small buffer zone) in memory? What if all we did was store a glyph and two bytes that pointed to an in-bank HEAD node? If we used 64*28, that's 5,376 bytes. On top of this, 100 objects and 100 monsters would equal 7,400 bytes. That seems about right for such a game. If you can get the data to load and save and can spare the 12k RAM, you could have maps of unusual size.

There's also the zelda technique, where you just load a new map chunk and scrol into it when you move past the edge; this is a fun technique too, and allows us to deal solely with the play area.

Another technique is shrinking the play area. Compared to rogue, etc. this is the Bard's Tale and Ultima 4 solution; The Bard's tale had a small play window in the upper left of the screen; Ultima 4 had an 11×11 play window. 11×11! Even if you stored the 33×33 tiles around the player with HEAD pointers, that's just 3,267 bytes! Storing maps in 11×11 chunks is very interesting. When you think back to how Ultima 4 and 5 worked, isn't it true that monsters more than 2 screens away would tend to disappear? That is a clue to the method they used to cache the world map. Additionally, most dungeon combat and dungeon rooms were one-screen. This saved a lot of memory, and greatly contributed to the game's look and feel. You didn't notice it so much, since the towns and world map were much larger.

A third solution is splitting the difference. if the top 16 tiles on the map account for ~95% of the data, you can save space by bit packing those tiles into half a byte, and then including a list of special objects (ex. a bridge, a moongate, a town/abbey) in a separate list of objects to be drawn on the map. This allows you to store a huge world map (ex. 256*256) in just 32k. You could even get it down to 24k if you only had 8 main terrain types. How many terrain types did Ultima 4 use? Glancing at a *shapes* file there appeaer to be only 22-23 overworld tiles, possibly a few more; It would be easy to split some tiles off into a special list (like towns, bridges and docks, shrines, and such). In fact, it seems as if Ultima 4's world map was designed with this as a fallback:

- Basic terrain (tiles 0–8): Deep Water, Medium Water, Shallow Water, Swamp, Grasslands, Scrubland/Brush, Forest, Hills, Mountains.
- Special location/overlay tiles (commonly placed on the map): Dungeon Entrance (9), Town (10), Castle (11), Village (12), Ruins (29), Shrine (30), various moongates, bridges, Lord British’s castle pieces, etc.

Having 16 basic tiles means you can bit pack a 256×256 map into 32k. Next, you can have a small list of extra overlay tiles. If you cut the map into 4×4 (16 chunks), each chunk would fit into 2k memory. Then an additional list of over 200 monsters and items could exist in this chunk, using *just* another 2k of memory. The object struct for this would be 10 bytes; 2 bytes for ID, 2 bytes for NEXT_ID (for containers), 2 bytes for X and Y in the chunk, 1 byte type, 3 bytes data.

Large, active worlds

If you absolutely want to make a very large, active world (civilization clone?) think about what you must absolutely do with each thing. For example if you have to keep track of a crowd of 20,000 people, and they have four behavior states (ex. chopping wood, mining gold, farming, bulding) then each person's behavior strategy can be stored in 2 bits. This means you need 40,000 bits, or 5k. If you only have 16 types of object in your game (or 8) consider bit packing for that as well. Bit packing is cheap; computing it (decoding it) is also cheap, in general.

How far can you push the machine to make it play the kind of games you want to make?

Moving the monsters

It would be very easy to move the monsters. Simply, just calculate the difference between each monster's position and the player's position and have the monster try to move towards the player. That's all.

Combat occurrs when a monster or player tries to occupy the other's space. Monsters will naturally attack players by trying to move onto their square. It's a simple function, just check if the monster is moving onto the player's co-ordinates, or if there is a monster where the player is moving. A function like `tile_hasmon` would be very useful here; if the tile has a monster it returns a pointer to the monster (ex. in FLD). Then you can write a combat routine.

Levels

In the same way you would save data for larger maps you can save and load entire levels. To do this you would want to rewrite parts of the game to use map_ map1_ and then make map_ a pointer to an area where you load and save maps. Or do this with map1 and then make a map2 that you can load from disk. Either way, adding a load and save map function is vital for traversing up and down dungeon levels.

A save game function is not far away from a save level/save map function.

Go forth and code great games!

Good artists copy, great artists steal. This means if you want to be good, write a game similar to an existing popular game. But if you want to be great, you must take the ideas you get from that game and make them your own ideas – creatively! Part of the challenge of writing a game is being creative and coming up with your own stuff. But to do this you need inspiration. Play the games of old; hack, rogue, adventure, zork, The Bard's Tale, Ultima – The Legend of Zelda – and you will understand what you must do!

Rogueima I MVP-4 Source Code

Here is the complete source code for the tutorial (including the extensions in Appendix I).

To compile this code, concatenate the files in any order (rogueima.sda must be first). Then INCOPY the file and type:

```
  as allfiles.sda rogueima
```

This will assemble the game. Then you can play it by typing:

```
  SYS 49152
```

I may return to soup it up a little with some easy monster AI and a quest (“Thou must kill... a viper!”) but this is really only intended as an instructional game to show you how to program in assembly. If you'd like to continue from here, you may wish to check out another game in the “Writing Games in Assembly Language” series: The Rogue's Tale I. Coming soon!

- Rogueima I MVP-4

Appendix I Extending the Game

As a bonus lesson, i'll show you how easy it is to extend the game. First, if you don't know how to do something, just think about it. For a couple of days, I knew that the key to adding monsters was in the create_gold function; because monsters are `mobs` in programmer lingo; that is, `mobile objects`. And I knew the key to extending the game to include a TALK command would then require a version of the `open_door` command. So let's get started.

Spawning Monsters

By this time you know how to add a new command, so add two new commands: m will call `spawn_monster:` and `t` will call `do_talk:`.

Writing spawn_monster is easy.

1. Copy the create_gold function into a file called mon.sda and rename it to spawn_monster.

2. Change the object type (in spawn_monster) to be #2 (monster). Why 2? Gold is 1, monster is 2. Everything has to be different so we know what kind of object it is.

3. Now, rip out all the random number generation and put it in a function called `random_monster:`. We will now change the random stuff slightly so it is for a monster and not a pile of gold.

Now, draw_items will also draw monsters automatically! To make the visuals right (monsters always walk on top of items) add a check like this to draw_items:

And we will make a draw_mobs function too (that only draws item type #2). This draw_mobs will be called after items, and before player:

```
draw_mobs:
    LDD [@cam_origin_x]    ; DL = I_start, DH = J_start
    LDT [@disp_dim]        ; TL = play_width, TH = play_height
    LDELM @OBJS_BASE
draw_mobs_loop:
    LDELM [ELM]
    CMP ELM, @OBJS_BASE
    JZ @draw_mobs_done

    LDI @OBJ_ID
```

```
    LDA [ELM+I]
    JZ @draw_mobs_loop          ; uninitialized slot

    LDI @OBJ_TYPE
    LDA [ELM+I]
    CMP A, #2                   ; OBJ_TYPE_MONSTER
    JNZ @draw_mobs_loop         ; only draw monsters

    LDI @OBJ_X
    LDX [ELM+I]
    LDI @OBJ_Y
    LDY [ELM+I]

    ; can see the monster?
    CMP XL, DL
    JNC @draw_mobs_loop
    MOV AL, DL
    ADD AL, TL
    CMP XL, AL
    JC @draw_mobs_loop

    CMP YL, DH
    JNC @draw_mobs_loop
    MOV AL, DH
    ADD AL, TH
    CMP YL, AL
    JC @draw_mobs_loop

    ; map to screen
    SUB XL, DL
    ADD XL, #1
    SUB YL, DH
    ADD YL, #1
    ; draw
    LDI @OBJ_VIS
    LDBL [ELM+I]
    LDAH $11
    INT $10
    JMP @draw_mobs_loop
draw_mobs_done:
    RET
```

Extending and refactoring

Notice that draw_mobs is essentially a cut and paste of draw_items. This suggests that it is possible to refactor some of the code into helper functions. Same thing with create_items vs. spawn_monsters. Same idea.

Next up we will do the talk command. The same idea will apply; talk is essentially a copy of the open

command.

The 'Talk' command

Did you add the talk command to the key dispatcher yet?

```
gi_check_keys:
    CMP AL, #'H'
    JZ @move_left
    CMP AL, #'J'
    JZ @move_down
    CMP AL, #'K'
    JZ @move_up
    CMP AL, #'L'
    JZ @move_right

    CMP AL, #128
    JZ @move_left
    CMP AL, #129
    JZ @move_down
    CMP AL, #130
    JZ @move_up
    CMP AL, #131
    JZ @move_right

    CMP AL, #'M'
    JZ @spawn_monster
    CMP AL, #'T'
    JZ @do_talk
    CMP AL, #'O'
    JZ @open_door
    CMP AL, #'G'
    JZ @create_gold
    CMP AL, #'Q'
    JZ @quit_game
    CMP AL, #','        // pick things up using comma
    JZ @pick_up_items

    RET
```

Now, here's the whole talk.asm file. We'll examine this a bit more quickly since it's just a copy of open doors (essentially). It works the same way – ask the player for a direction and then print a message about what happens when you talk to any particular thing.

```
; Talk.sda
; handle talking.
; Right now you can only talk to a monster, unfortunately.
```

```
; Or an item?

;;;;;;;;;;;;;;;;;;;;;;;;;;
;; do_talk()
;; This started as a copy of open_door
;;

do_talk:
    ; Show "Talk where?"
    LDELM @str_talk_where
    CALL @print_msg
    CALL @msg_display

    ; Ask player for key
    LDAH $02    ; Blocking GETKEY -> AL
    INT $10

    ;; Add the player's key as entropy.
    LDAH $02
    INT 0x13

    ; Uppercase: if AL >= 'a' and AL < 'z'+1, subtract 32
    CMP AL, #97
    JNC @talk_check_keys      ; AL < 'a', skip
    CMP AL, #123
    JC @talk_check_keys       ; AL >= '{', skip
    SUB AL, #32

talk_check_keys:
    ; Check direction given
    CMP AL, #'H'
    JZ @talk_west
    CMP AL, #'J'
    JZ @talk_south
    CMP AL, #'K'
    JZ @talk_north
    CMP AL, #'L'
    JZ @talk_east

    CMP AL, #128
    JZ @talk_west
    CMP AL, #129
    JZ @talk_south
    CMP AL, #130
    JZ @talk_north
    CMP AL, #131
    JZ @talk_east

    ; Bad direction -- ignore.
```

```
    RET

talk_north:
    ; Show "talk north"
    LDELM @str_TALK
    CALL @print_msg
    LDELM @str_NORTH
    CALL @print_msg
    LDELM @str_crlf
    CALL @print_msg

    LDI #0
    LDJ #0
    LDK #0
    LDL #1
    JMP @talk_go

talk_south:
    LDELM @str_TALK
    CALL @print_msg
    LDELM @str_SOUTH
    CALL @print_msg
    LDELM @str_crlf
    CALL @print_msg

    LDI #0
    LDJ #1
    LDK #0
    LDL #0
    JMP @talk_go

talk_west:
    LDELM @str_TALK
    CALL @print_msg
    LDELM @str_WEST
    CALL @print_msg
    LDELM @str_crlf
    CALL @print_msg

    LDI #0
    LDJ #0
    LDK #1
    LDL #0
    JMP @talk_go

talk_east:
    LDELM @str_TALK
    CALL @print_msg
    LDELM @str_EAST
    CALL @print_msg
    LDELM @str_crlf
```

```
    CALL @print_msg

    LDI #1
    LDJ #0
    LDK #0
    LDL #0
    JMP @talk_go

talk_go:
    LDX #0
    LDY #0
    LDXL [@PX]
    LDYL [@PY]
    ADD XL, IL
    ADD YL, JL
    SUB XL, KL
    SUB YL, LL
    CALL @get_glyph

    CMP AL, #' '
    JZ @talk_air
    CMP AL, #'#'
    JZ @talk_wall
    CMP AL, #'+'
    JZ @talk_cdoor
    CMP AL, #'-'
    JZ @talk_odoor
    CMP AL, #'r'
    JZ @talk_rat
    CMP AL, #'s'
    JZ @talk_snake
    CMP AL, #'x'
    JZ @talk_spider

    ;; bad command? ignore.
    RET

talk_air:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_air
    CALL @print_msg
    RET

talk_wall:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_wall
    CALL @print_msg
    RET
```

```
talk_cdoor:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_cdoor
    CALL @print_msg
    RET

talk_odoor:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_odoor
    CALL @print_msg
    RET

talk_rat:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_rat
    CALL @print_msg
    RET

talk_snake:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_snake
    CALL @print_msg
    RET

talk_spider:
    LDELM @str_ytt
    CALL @print_msg
    LDELM @str_talk_spider
    CALL @print_msg
    RET



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; talk strings

str_talk_where:
    .bytes " > Talk where? ", 13, 10, 0

str_TALK:
    .bytes "  TALK ", 0

str_ytt:
    .bytes " You talk to ", 0

str_talk_air:
    .bytes "the ", 13, 10, " air. But no-one", 13, 10, " is there!", 13, 10,
0
```

```
str_talk_wall:
    .bytes "the ", 13, 10, " wall. It's not ", 13, 10, " listening.", 13,
10, 0

str_talk_cdoor:
    .bytes "the ", 13, 10, " closed door. ", 13, 10, 32, 34,"Open sesame!",
34, 13, 10, 0

str_talk_odoor:
    .bytes "the ", 13, 10, " open door. Nothing happens.... YET!", 13, 10, 0

str_talk_rat:
    .bytes "the ", 13, 10, " rat. It squeaks", 13, 10, " in anger!", 13, 10,
0

str_talk_snake:
    .bytes "the ", 13, 10, " snake. It hisses", 13, 10, " dangerously!", 13,
10, 0

str_talk_spider:
    .bytes "the ", 13, 10, " spider. It moves", 13, 10, " closer!", 13, 10,
0
```

Fixing a bug

Notice how when you try to talk to a monster, it doesn't work? That's because get_glyph only looks at the map. Let's add a check for any item or monster.

```
get_glyph:
    PUSH ELM
    PUSH FLD
    PUSH K
    PUSH Z
    PUSH I
    PUSH B

    ; map tile glyph
    LDK [@map1_dim]
    MOV Z, Y
    MUL Z, KL
    ADD Z, X
    LDELM @map1_data
    ADD ELM, Z
    LDAL [ELM]              ; AL = map glyph (default return)

    ; Override if any object sits at (X, Y)
    LDFLD @OBJS_BASE
get_glyph_obj_loop:
```

```
    LDFLD [FLD]
    CMP FLD, @OBJS_BASE
    JZ @get_glyph_done      ; wrapped to head, no match, keep map glyph

    LDI @OBJ_ID
    LDB [FLD+I]
    JZ @get_glyph_obj_loop  ; uninitialized slot

    LDI @OBJ_X
    LDBL [FLD+I]
    CMP BL, XL
    JNZ @get_glyph_obj_loop

    LDI @OBJ_Y
    LDBL [FLD+I]
    CMP BL, YL
    JNZ @get_glyph_obj_loop

    ; Match will override AL with the object's VIS, then exit loop
    LDI @OBJ_VIS
    LDAL [FLD+I]

get_glyph_done:
    POP B
    POP I
    POP Z
    POP K
    POP FLD
    POP ELM
    RET
```

Appendix II: Monster Combat

Just one more thing! Here's how to make monsters move around and attack you.

First we need to know if a square has a monster or not. This means, when we try to move onto a square, we test if that square has a monster. If it does, we attack that monster.

has_mon

This function tells us if there is a monster on the tile.

```
;;;;;;;;;;;;;;;;;;
;; has_mon(x,y)
;;
;; Returns CARRY SET if monster.
;; Returns monster pointer in FLD.
```

```
;;
has_mon:
    PUSH I
    PUSH B

    LDFLD @OBJS_BASE
has_mon_loop:
    LDFLD [FLD]
    CMP FLD, @OBJS_BASE
    JZ @has_mon_no

    LDI @OBJ_ID
    LDB [FLD+I]
    JZ @has_mon_loop           ; uninitialized slot

    LDI @OBJ_TYPE
    LDB [FLD+I]
    CMP B, #2                   ; OBJ_TYPE_MONSTER
    JNZ @has_mon_loop

    LDI @OBJ_X
    LDBL [FLD+I]
    CMP BL, XL
    JNZ @has_mon_loop

    LDI @OBJ_Y
    LDBL [FLD+I]
    CMP BL, YL
    JNZ @has_mon_loop

    ; Match returns pointer in FLD, CF=1
    SEC
    JMP @has_mon_done

has_mon_no:
    CLC                         ; no monster; FLD unchanged.

has_mon_done:
    POP B
    POP I
    RET
```

move_mon

This is where we attempt to move the monsters. Don't forget to move them *before* you draw them in the game loop:

draw_map:

```
  LDAH $51        ; VSTOP
  INT 0x18
```

```
  CALL @draw_borders
  CALL @draw_world
  CALL @draw_items
  CALL @move_mon
  CALL @draw_mobs
  CALL @draw_player
  CALL @draw_stats
  CALL @draw_msgs
  CALL @msg_display
```

```
  LDAH $50        ; VSTART
  INT 0x18
```

```
  RET
```

Add this in, and add `CALL @move_mon` before `CALL @draw_mon` in `draw_map:`.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; move_mon
;;
;; Attempt to move every live monster
;; towards the player
;;

move_mon:
    LDELM @OBJS_BASE
move_mon_loop:
    LDELM [ELM]
    CMP ELM, @OBJS_BASE
    JZ @move_mon_done

    ; live monsters only
    LDI @OBJ_ID
    LDA [ELM+I]
    JZ @move_mon_loop
    LDI @OBJ_TYPE
    LDA [ELM+I]
    CMP A, #2                   ; OBJ_TYPE_MONSTER
    JNZ @move_mon_loop

    LDI @OBJ_X
    LDXL [ELM+I]
    LDI @OBJ_Y
    LDYL [ELM+I]

    ; sign(PX - mx) to CL
    LDAL [@PX]
```

```
    CMP AL, XL
    JZ  @mm_dx_zero
    JC  @mm_dx_pos              ; PX > mx
    LDCL #$FF                   ; PX < mx, step left
    JMP @mm_dx_done
mm_dx_pos:
    LDCL #$01
    JMP @mm_dx_done
mm_dx_zero:
    LDCL #$00
mm_dx_done:

    ; sign(PY - my) into DL
    LDAL [@PY]
    CMP AL, YL
    JZ  @mm_dy_zero
    JC  @mm_dy_pos
    LDDL #$FF
    JMP @mm_dy_done
mm_dy_pos:
    LDDL #$01
    JMP @mm_dy_done
mm_dy_zero:
    LDDL #$00
mm_dy_done:

    ; Pick axis
    CMP CL, #0
    JNZ @mm_dx_nz
    CMP DL, #0
    JZ  @move_mon_loop          ; both zero, monster sits on player, skip
    JMP @mm_use_y

mm_dx_nz:
    CMP DL, #0
    JZ  @mm_use_x

    ; Both non-zero (coin flip)
    LDAH $00
    INT 0x13
    MOD B, #2
    CMP B, #0
    JZ  @mm_use_x
    ; fall through to mm_use_y

mm_use_y:
    ADD YL, DL                  ; YL = my + sign(dy)
    JMP @mm_check
mm_use_x:
    ADD XL, CL                  ; XL = mx + sign(dx)
```

```
mm_check:
    ; Don't step on the player
    LDAL [@PX]
    CMP AL, XL
    JNZ @mm_check_obj
    LDAL [@PY]
    CMP AL, YL
    JZ  @move_mon_loop

mm_check_obj:
    PUSH ELM
    CALL @has_mon
    JC  @mm_pop_skip            ; another monster occupies target
    CALL @is_walkable
    JNC @mm_pop_skip            ; wall or other non-walkable

    ; commit the move
    POP ELM
    LDI @OBJ_X
    STXL [ELM+I]
    LDI @OBJ_Y
    STYL [ELM+I]
    JMP @move_mon_loop

mm_pop_skip:
    POP ELM
    JMP @move_mon_loop

move_mon_done:
    RET
```

Combat

When a monster attempts to move onto a player, this is a combat situation.

When a player attempts to move onto a monster, this is also a combat situation.

Ideally we want to use the same code, but this will never work out in practice unless we add the player as an object in the item list (so it has data like a monster). This is actually one way of running the game, but for now we will have a special case to set up player vs monster or monster vs player. What we will do is set up a general data structure and have the code for player/monster at the start and end. That will be good enough for now.

Let's change mon_move to call the combat function:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; move_mon
;;
;; Attempt to move every live monster
```

```
;; towards the player
;;
move_mon:
    LDELM @OBJS_BASE
move_mon_loop:
    LDELM [ELM]
    CMP ELM, @OBJS_BASE
    JZ @move_mon_done

    ; live monsters only
    LDI @OBJ_ID
    LDA [ELM+I]
    JZ @move_mon_loop
    LDI @OBJ_TYPE
    LDA [ELM+I]
    CMP A, #2                   ; OBJ_TYPE_MONSTER
    JNZ @move_mon_loop

    LDI @OBJ_X
    LDXL [ELM+I]
    LDI @OBJ_Y
    LDYL [ELM+I]

    ; sign(PX - mx) to CL
    LDAL [@PX]
    CMP AL, XL
    JZ  @mm_dx_zero
    JC  @mm_dx_pos              ; PX > mx
    LDCL #$FF                   ; PX < mx, step left
    JMP @mm_dx_done

mm_dx_pos:
    LDCL #$01
    JMP @mm_dx_done

mm_dx_zero:
    LDCL #$00

mm_dx_done:
    LDAL [@PY]
    CMP AL, YL
    JZ  @mm_dy_zero
    JC  @mm_dy_pos
    LDDL #$FF
    JMP @mm_dy_done

mm_dy_pos:
    LDDL #$01
    JMP @mm_dy_done
```

```
mm_dy_zero:
    LDDL #$00

mm_dy_done:
    ; Pick axis
    CMP CL, #0
    JNZ @mm_dx_nz
    CMP DL, #0
    JZ  @move_mon_loop         ; both zero, monster sits on player, skip
    JMP @mm_use_y

mm_dx_nz:
    CMP DL, #0
    JZ  @mm_use_x

    ; Both non-zero; coin flip
    LDAH $00
    INT 0x13
    MOD B, #2
    CMP B, #0
    JZ  @mm_use_x
    ; fall through to mm_use_y

mm_use_y:
    ADD YL, DL                 ; YL = my + sign(dy)
    JMP @mm_check

mm_use_x:
    ADD XL, CL                 ; XL = mx + sign(dx)

mm_check:
    LDAL [@PX]
    CMP AL, XL
    JNZ @mm_check_obj
    LDAL [@PY]
    CMP AL, YL
    JNZ @mm_check_obj

    ; Target tile IS the player, so attack instead of move
    LDFLD @player_obj           ; ELM attacks FLD

    CALL @do_combat
    JMP @move_mon_loop          ; monster's turn is spent attacking

mm_check_obj:
    PUSH ELM
    CALL @has_mon
    JC  @mm_pop_skip            ; another monster occupies target
    CALL @is_walkable
    JNC @mm_pop_skip            ; wall or other non-walkable
```

```
    ; commit the move
    POP ELM
    SED
    LDI @OBJ_X
    STXL [ELM+I]
    LDI @OBJ_Y
    STYL [ELM+I]
    CLD
    JMP @move_mon_loop

mm_pop_skip:
    POP ELM
    JMP @move_mon_loop

move_mon_done:
    RET
```

And now, the combat function. This is just an example; every attack does 1 damage to the defender. You could expand on this later.

combat.sda

```
player_obj:
    ; 40 bytes, about sizeof(OBJ)
    .bytes 0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0
    .bytes 0,0,0,0,0,0,0,0,0,0

str_you:
    .bytes "You ", 0
str_the:
    .bytes "The ", 0
str_attack:
    .bytes "attack for 1 damage!", 13, 10, 0   ; player verb
str_attacks:
    .bytes "attacks for 1 damage!", 13, 10, 0  ; monster verb
str_rat:
    .bytes "rat ", 0
str_snake:
    .bytes "snake ", 0
str_spider:
    .bytes "spider ", 0
str_unknown:
    .bytes "thing ", 0
str_dies:
    .bytes "dies!", 13, 10, 0
```

```
str_you_die:
    .bytes "You die!", 13, 10, 0

copy_player_to_obj:
    PUSH ELM
    PUSH A
    PUSH I

    LDELM @player_obj

    LDI @OBJ_DATA1
    LDA [@player_hp]
    STA [ELM+I]

    LDI @OBJ_DATA2
    LDA [@player_str]
    STA [ELM+I]

    POP I
    POP A
    POP ELM
    RET

copy_obj_to_player:
    PUSH ELM
    PUSH A
    PUSH I
    LDELM @player_obj

    LDI @OBJ_DATA1
    LDA [ELM+I]
    STA [@player_hp]

    LDI @OBJ_DATA2
    LDA [ELM+I]
    STA [@player_str]
    POP I
    POP A
    POP ELM
    RET


print_mon_name:
    PUSH ELM
    PUSH A
    PUSH I
    LDI @OBJ_VIS
    LDAL [ELM+I]
    CMP AL, #'r'
    JZ  @pmn_rat
    CMP AL, #'s'
```

```
    JZ  @pmn_snake
    CMP AL, #'x'
    JZ  @pmn_spider
    LDELM @str_unknown
    JMP @pmn_emit
pmn_rat:
    LDELM @str_rat
    JMP @pmn_emit
pmn_snake:
    LDELM @str_snake
    JMP @pmn_emit
pmn_spider:
    LDELM @str_spider
pmn_emit:
    CALL @print_msg
    POP I
    POP A
    POP ELM
    RET



;;;;;;;;;;;;;;;;;;;;;;;;;
;; Call with:
;; ELM is the attacker
;; FLD is the defender
;; Note: If player is attacking or defending,
;;       pass @player_obj in ELM or FLD as appropriate.
;;       If the player is not involved, syncing doesn't hurt.
;;
do_combat:
    ; sync
    CALL @copy_player_to_obj

combat_damage:
    ; part 2: damage
    LDI @OBJ_DATA1
    LDA [FLD+I]
    JZ  @combat_msg                 ; defender already 0 (defensive)
    DEC AL
    STAL [FLD+I]                    ; HP -= 1, stored (may be 0)

combat_msg:
    CMP ELM, @player_obj
    JZ  @combat_msg_player
    PUSH ELM
    LDELM @str_the
    CALL @print_msg
    POP ELM
```

```
    CALL @print_mon_name              ; ELM = attacker
    PUSH ELM
    LDELM @str_attacks
    CALL @print_msg
    POP ELM
    JMP @combat_check_dead

combat_msg_player:
    PUSH ELM
    LDELM @str_you
    CALL @print_msg
    LDELM @str_attack
    CALL @print_msg
    POP ELM

combat_check_dead:
    LDI @OBJ_DATA1
    LDA [FLD+I]
    JNZ @combat_post
    CALL @kill_obj

combat_post:
    CALL @copy_obj_to_player

combat_done:
    RET


kill_obj:
    ; Monster dies
    ; player will be handled in main loop
    PUSH ELM                          ; save caller's attacker
    MOV ELM, FLD                     ; ELM = victim for everything below

    PUSH ELM
    LDELM @str_the
    CALL @print_msg
    POP ELM
    CALL @print_mon_name              ; reads OBJ_VIS via ELM = victim
    PUSH ELM
    LDELM @str_dies
    CALL @print_msg
    POP ELM

    LDI @OBJ_ID
    LDA #0
    STA [ELM+I]                       ; free the slot

    POP ELM                           ; restore attacker
    RET
```

```
kill_player:
    PUSH ELM
    LDELM @str_you_die
    CALL @print_msg
    POP ELM

    ; CLS
    LDAH $10
    INT $10

    ; Print "You died."
    LDBLX @msg_died
    LDAH $66
    INT $05

    POP ELM ; get rid of return address from CALL @draw_map game loop
    RET     ; exit program.
```

Finally, if the player is dead, we end the game. in `draw_map:`

```
  ...
  CALL @draw_stats
  CALL @draw_msgs
  CALL @msg_display
  LDAH $50        ; VSTART
  INT 0x18
  LDA [@player_hp]
  CMP A, 0
  JZ @kill_player
```

```
  RET
```

Appendix III: DungeonMaker

Random dungeon maker used by NetWhack v0.7.2

I don't know where I got this from, but it is ancient code. It is likely older than I am. It's probably from the original hack or rogue games. Of course, it has been heavily modified, but the core is that old, ancient algorithm for blowing room bubbles and connecting them. I have removed some irreverent boilerplate to present a cleaner pseudocode-like version.

```
    // fill dungeon with solid rock.
    public void fill_with_rock() {
        amap = new char[xmax][ymax];
        for (int ix = 0; ix < xmax; ix++) {
            for (int iy = 0; iy < ymax; iy++) {
```

```
                amap[ix][iy] = '#';
            }
        }
        return;
    }

    public void bound_with_undiggable_walls() {
        // bound the map with undiggable walls
        for (int ix = 0; ix < xmax; ix++) {
            amap[ix][0] = 'X';    // X = undiggable wall
            amap[ix][ymax - 1] = 'X';    // X = undiggable wall
        }
        for (int iy = 0; iy < ymax; iy++) {
            amap[0][iy] = 'X';
            amap[xmax - 1][iy] = 'X';
        }
        return;
    }

    // returns true if area is filled with "diggable tiles".
    public boolean isWall(Coord j, Coord k) {
        for (int ix = j.x; ix <= k.x; ix++) {
            for (int iy = j.y; iy <= k.y; iy++) {
                if (amap[ix][iy] != '#') {
                    return false;
                }
            }
        }

        return true;
    }

    public boolean isWall(Coord a) {
        return (amap[a.x][a.y] == '#');
    }

    public boolean isWall(int x, int y) {
        return (amap[x][y] == '#');
    }

    // Finds a wall we can build on.
    public Coord find_build_spot() {
        // We must find a spot for this feature.
        boolean success = false;
        int maxtries = (xmax * ymax) / 2;
        Coord a = new Coord();

        while ((success == false) && (maxtries-- > 0)) {
            // 1. pick a random spot in the dungeon
            a.x = Dice.roll(1, xmax - 2);
            a.y = Dice.roll(1, ymax - 2);
```

```
            // 2. if it is a wall
            if (isWall(a)) {
                // 3. does it have *1* floor next to it?
                int floors = 0;
                int build_dir = 0;  // opposite from the floor

                if (amap[a.x + 1][a.y] == ' ') {
                    floors++;
                    build_dir = WEST;
                }

                if (amap[a.x - 1][a.y] == ' ') {
                    floors++;
                    build_dir = EAST;
                }

                if (amap[a.x][a.y + 1] == ' ') {
                    floors++;
                    build_dir = NORTH;
                }

                if (amap[a.x][a.y - 1] == ' ') {
                    floors++;
                    build_dir = SOUTH;
                }

                if (floors == 1) {
                    // 4. Yes, we found a wall we can attempt to build a
feature on.
                    return (a);
                }
            } // if it's a wall

        } // while loop

        // failure.
        return (new Coord(0, 0));
    }

    // Determines the direction we should build in.
    // returns -1 on failure.
    public int find_build_dir(Coord a) {
        int floors = 0;
        int build_dir = 0;  // opposite from the floor

        if (amap[a.x + 1][a.y] == ' ') {
            floors++;
            build_dir = WEST;
        }
```

```
        if (amap[a.x - 1][a.y] == ' ') {
            floors++;
            build_dir = EAST;
        }

        if (amap[a.x][a.y + 1] == ' ') {
            floors++;
            build_dir = NORTH;
        }

        if (amap[a.x][a.y - 1] == ' ') {
            floors++;
            build_dir = SOUTH;
        }

        if (floors == 1) {
            // 4. Yes, we found a wall we can attempt to build a feature on.
            return (build_dir);
        }

        // failure.
        return (-1);
    }

    // Adds a random feature to the dungeon.
    // Assumes there is already a feature in the dungeon.
    public boolean addfeature() {
        boolean success = false;
        int maxtries = (xmax * ymax) / 2;

        // Find a spot, a build direction, and try to add a feature.
        while ((success == false) && (maxtries-- > 0)) {
            Coord a = find_build_spot();
            int build_dir = find_build_dir(a);
            success = addfeature(a, build_dir);
        } // while loop

        return success;
    }

    // builds a random feature at coord a, direction dir.
    public boolean addfeature(Coord a, int bd) {
        return addfeature(a.x, a.y, bd);
    }

    public boolean addfeature(int atx, int aty, int dir) {
        boolean success = false;

        int feature_type = Dice.roll(1, 50);
        switch (feature_type) {
            case 1:
```

```
                success = makeshop(atx, aty, dir);
                break;
            case 2:
            default:
                success = makeroom(atx, aty, dir);
                break;
        }

        // connect the features.
        if (success) {
            amap[atx][aty] = '+';
        }

        return success;
    }

    public boolean makeroom(int xloc, int yloc, int dir) {
        // come up with a size for the room.
        int roomw = Dice.roll(2, 8);
        int roomh = Dice.roll(2, 6);

        Coord a = new Coord();

        // set up the directional modifiers
        switch (dir) {
            case NORTH:
                a.x = xloc - roomw / 2;
                a.y = yloc - roomh - 1;
                break;
            case EAST:
                a.x = xloc + 1;
                a.y = yloc - roomh / 2;
                break;
            case SOUTH:
                a.x = xloc - roomw / 2;
                a.y = yloc + 1;
                break;
            case WEST:
                a.x = xloc - roomw - 1;
                a.y = yloc - roomh / 2;
                break;
        }
        Coord b = new Coord(a.x + roomw, a.y + roomh);

        check_coords(a, b);

        if (!isWall(a, b)) {
            return false;
        }
```

```
        // Mine the room.
        for (int ix = a.x; ix <= b.x; ix++) {
            for (int iy = a.y; iy <= b.y; iy++) {
                amap[ix][iy] = ' ';
            }
        }

        // we're done here.
        return true;
    }
}
```

The idea here is you create a room on a map of diggable tiles, surrounded by undiggable (end of map) tiles. Then you create a room, and mark the floor as connected. Then you create another room randomly somewhere else, and then draw a passage from the new room to the old room. then mark every floor space as 'connected'.

Actually I think the code above is even simpler; it does not attempt to create corridors and always builds rooms beside each other. Ancient code indeed.

For every room or feature you add, repeat this process. It is actually very easy to create a rogue or hack-style dungeon! Get creative with your features. NetHack has a lot of amazing features, Slash'Em has even more. Those are two very good sources to draw inspiration from.

Or maybe you'd like to make your game more like ADOM? Or even ZZT? Include a pseudo-3d view like in Bard's Tale or Ultima 4?

The future holds many exciting possibilities. I wish you luck in your game programming adventures!