# תוכן

חבילה זו מכילה קוד, הגדרות וכו' עבור:

1. גרסה של האמולטור QEMU עם DroidScope, אשר תומכת בטעינת plugins בזמן ריצה.
2. Plugin בשם "bgu" עבור QEMU אשר מוסיף את הפקודה "pmemload", שבעזרתה ניתן לכתוב לזיכרון הפיזי של המכשיר שהאמולטור מריץ.
3. סביבת עבודה של Eclipse – יותר נכון, כמה פרויקטים של Eclipse, אשר כבר מקונפגים באופן שמאפשר לעשות build ל-Kernel Module ולקוד ++C של אפליקציית Android מתוך Eclipse. כמו-כן, זה מאפשר לטעון Kernel Modules לאמולטור שרץ כרגע (ולהעיף אותו מהזיכרון אם הוא כבר היה שם).
4. Kernel Module עבור Android (עבור ה-Kernel אשר גם נוכח; לא בטוח שזה יעבוד בגרסאות אחרות) בשם "MemAccess", אשר יוצר התקן במערכת בשם "/dev/MemAccess" שדרכו ניתן לקרוא ב-User Space מכתובת זיכרון פיזית קבועה מראש.
5. אפליקציה ל-Android בשם "WekaWrapper" אשר מכילה את Weka כמודול (כרגע היא לא עושה איתו כלום, אבל זה מתקמפל), ואשר יכולה להפעיל קוד ב-++C שנמצא בה, שמשתמש ב-Kernel Module הנ"ל ע"מ לקרוא מידע מהזיכרון הפיזי.

**בקיצור** – זו אפליקציית Java שמצליחה לגשת לכתובת קבועה בזיכרון ולקרוא ממנה, כאשר היא נמצאת באזור הזיכרון של ה-Kernel.

1. גרסה של ה-Goldfish Android Kernel שמאפשרת טעינה ופריקה של Kernel Modules.

גרסה זו מכילה גם כמה משתנים שבהם השתמשנו ע"מ לבדוק את היכולות שהפעלנו (הכתיבה לזיכרון מבחוץ וקריאתו מבפנים).

1. כמה סקריפטים שמאפשרים ומקלים על כל הנ"ל.

לצערי, אינני זוכר את כל חבילות התוכנה שהייתי צריך להתקין ב-Linux כדי לגרום לכל לעבוד, אז אם אתם נתקלים בספריה שחסרה לכם, פשוט תחפשו באינטרנט איך מתקינים אותה (זה בד"כ ע"י פקודת "sudo apt-get install <PACKAGE\_NAME>").

# 1. הפעלת האמולטור עם DroidScope

התבססתי על ההוראות שכתובות באתר הבא:

<https://code.google.com/p/decaf-platform/wiki/DroidScope>

ההוראות מסבירות כיצד לקמפל את האמולטור יחד עם DroidScope, אשר פועל בעזרת תוסף בשם "DECAF". לצערי, ההוראות מאוד לא ברורות, וה-patch שאמור להוסיף את הקוד הרלוונטי לאמולטור מתוכנן עבור מערכת הקבצים של מי שכתב את המדריך. לכן במקום לקמפל את האמולטור בעצמי בסביבה שלי, הורדתי את ה-VirtualBox Image שהם מספקים, אשר מכילה סביבה מתאימה (עם הקוד של DroidScope) שבה אפשר להשתמש ע"מ לקמפל את האמולטור.

קישור ל-Image ניתן למצוא באתר הנ"ל (הוא לוקח כ-7GB).  
בעת כתיבת הסבר זה, הקישור היה לכתובת הבאה:

<http://sycurelab.ecs.syr.edu/image/DroidScope.tar.gz>

**לפני שאתם מורידים אותו** ומנסים להפעיל את הקומפילציה, קחו בחשבון שהסיבה היחידה שהייתי צריך אותו בסוף הייתה לשם לקמפל את האמולטור **פעם אחת**.  
אז לקחתי את הקבצים הבינאריים, אשר גם כלולים בתיקיה המצורפת, השתמשתי בהם, ולא הייתי צריך יותר את ה-VirtualBox Image שלהם. אלה הקבצים הנמצאים בתיקייה "**DroidScope/bin**".

## קימפול האמולטור עם DroidScope

אם אתם בוחרים שכן לקמפל את האמולטור, אני ממליץ לכם להוריד את ה-VirtualBox Image ואז בגדול רק לעקוב אחרי ההוראות שבקובץ "**manual.pdf**" שנמצא באותו הארכיון.

אני אומר "בגדול," כי הוא לא לגמרי מסונכרן עם תוכן הארכיון.  
להלן השינויים שיש לקחת בחשבון במעקב אחרי המדריך (המדריך עצמו וסקריפט לשם הרצת ה-VM נמצאים בתיקיה DroidScope/Archive בחבילה זו; ה-VM עצמו, כמובן, לא שם):

* [1.1] פקודת ההרצה של ה-VM צריכה להיות עם שם ה-Image, שהוא "**DroidScope.vdi**" ולא "DroidScope.qcow2".
* [1.1.2] שוב יש לשנות את שם ה-Image לשם הנכון. כמו-כן, מומלץ בפקודה מס' 3 לכתוב במקום "mnt" נתיב של תיקיה אליה אתם רוצים למפות את מערכת הקבצים שנמצאת בתוך ה-VM.

אני הייתי צריך את זה לשם העתקת הקבצים הבינאריים מה-VM לסביבה שלי.

יש בתיקיית ה-Archive שציינתי לעיל קובץ בשם "**Notes.txt**" המכיל את הפקודות הנחוצות לשם מיפוי מערכת הקבצים ולשם הסרתה.

* [4] אני ממליץ שלא לנסות לכתוב plugin לפי ההוראות של "First Plugin" במסמך ה-PDF; נראה שהוא לא מדויק במיוחד (אני חושב שבמקום "TEMU" בכל מקום צריך להיות כתוב "DECAF\_shared," למשל, ושמות קבצי ה-include לא לגמרי נכונים).

במקום זאת, עדיף להתבסס על plugin שעובד, כגון ה-DalvikInstructionTracer שלהם, או ה-HelloWorld שלי (אגב, אני חושב שהשארתי כמה include-ים מיותרים שם).

## הפעלת האמולטור החדש במקום הקיים

אני יוצא מנקודת הנחה שכבר יש לכם סביבת עבודה עם ה-Android SDK, ושכבר הפעלתם אמולטור של Android, ושעכשיו אתם רוצים להפעיל את אותו הדבר, אבל עם היכולת לטעון plugins וכל זה.

אתאר כיצד לעשות זאת בצורה שבה זה עבד לי:

1. בתיקיית ה-Android SDK, איפה שיש תיקייה בשם "**tools**" ובה יש את הקבצים "**emulator**" וְ-"**emulator-arm**", תיצרו תיקייה חדשה (אני קראתי לה "**DroidScope**").
2. שימו בתיקיה החדשה את האמולטור החדש.
   * אם אתם משתמשים במה שאני קימפלתי מראש, אז זה פשוט התוכן של התיקייה "**DroidScope/bin**".
   * אם קימפלתם את האמולטור עם DroidScope בעצמכם – אז קודם כל כל הכבוד לכם – והקבצים שאתם צריכים הם "**emulator**" וְ-"**emulator-arm**".
3. תריצו את האמולטור.
   * אם יש לכם כבר סקריפט שעושה זאת, אז פשוט תחליפו את הקובץ "**emulator**" שהוא מפעיל בזה שנמצא בתיקיה החדשה שיצרתם.
   * אם עוד לא יצא לכם להפעיל אמולטור מה-command line, אתם מוזמנים להשתמש בסקריפט שלי (אחרי שתשנו אותו בהתאם לסביבה שלכם). הוא נקרא "**wekaEmulatorScript**", וגם הוא נמצא בחבילה (מחוץ לשאר התיקיות).
     + תצטרכו לשנות בו את השם של ה-AVD (שלי הוא "Weka") ואת הנתיב של ה-image של ה-kernel שלכם. השם של ה-AVD הוא השם של המכשיר שהגדרתם כשיצרתם אותו (הוא מופיע במנהל ה-devices שמגיע עם ה-Android SDK).
   * **שימו לב**: אם שני הקבצים של האמולטור ("**emulator**" וְ-"**emulator-arm**") לא נמצאים בתת-תיקייה של ה-AndroidSDK, הוא לא יצליח להפעיל אותו עם ה-AVD שלכם (לפחות, אצלי הוא לא הצליח).

# 2. The BGU Plugin

את קוד המקור של כל ה-plugins ניתן למצוא בתיקיה:

DroidScope/src/DECAF\_plugins

בקוד של ה-plugin יש מספר קבצים מעניינים:

* **plugin\_cmds.h** – מכיל את ההגדרות של הפקודות שה-plugin מוסיף לאמולטור.
* **BGU.c** – מכיל את קוד המקור של ה-plugin.
* **configure** – קובץ ריצה שמקנפג את הקבצים בתיקיה לקראת קומפילציה. יש להריצו לפני שמקמפלים את ה-plugin, ע"מ לקבוע שה-target שלו הוא Android ולספק לו את הנתיב של תיקיית קוד המקור של DECAF. ניתן לראות דוגמה של כיצד יש להריצו בקובץ **configure.sh** (אגב, נדמה לי שלא הצלחתי להריץ את **configure.sh** כקובץ ריצה, אלא רק בעזרת הפקודה "bash **configure.sh**").

## תיאור הפקודות החדשות לאמולטור

בקובץ **plugin\_cmds.h** מתוארות מספר פקודות חדשות שה-plugin מוסיף לאמולטור.

מבנה תיאור של פקודה:

{

.name = "<COMMAND\_NAME>",

.args\_type = "<ARGUMENT\_TYPES>",

.\_QEMU\_MON\_HANDLER\_CMD = <NAME\_OF\_HANDLER\_FUNCTION>,

.params = "<NAMES\_OF\_ARGUMENTS\_FOR\_HELP>",

.help = "<HELP\_DESCRIPTION>"

},

הסבר על כל שדה:

* name – שם הפקודה.

זהו השם בו משתמשים לשם הפעלת הפקודה באמולטור.

דוגמה: pmemload

* args\_type – טיפוסי הפרמטרים של הפקודה.

זוהי מחרוזת המתארת את טיפוסי המשתנים המועברים לפונקציה שמטפלת בפקודה.

כל תו מייצג טיפוס של משתנה אחד או יותר.

למשל התו 's' מייצג מחרוזת, התו 'i' מייצג int והתו 'l' ('L' קטנה) מייצג long, אשר מועבר לפונקציה כשני int-ים.

דוגמה: lis

(עם מחרוזת כזו, הפונקציה צריכה לקבל זוג של **unsigned int** עבור האות הראשונה, **int** אחד עבור האות השנייה ו-**const char\*** עבור האות השלישית.)

אני הסקתי אילו משתנים אני צריך לקבל מהתבוננות בקובץ "**qemu-monitor.hx**" שנמצא בקוד של האמולטור (תיקיית **DroidScope/src**), אשר מתאר פקודות קיימות. התבססתי על pmemsave כי רציתי ליצור pmemload שעושה משהו מאוד דומה.

במידת הצורך, אתם יכולים להתבונן בקוד של האמולטור – יש שם קובץ בשם "**monitor.c**" שבו יש פונקציה בשם "**monitor\_handle\_command**" שמטפלת בפקודות שהאמולטור מקבל. שם נעשה כל הטיפול של פירוש האותיות השונות לטיפוסי משתנים.

* \_QEMU\_MON\_HANDLER\_CMD – מצביע לפונקציה המטפלת בפקודה.

זה צריך להיות פשוט השם של הפונקציה שכתבתם שאמורה להיקרא כשמפעילים את הפקודה.

דוגמה: bgu\_do\_physical\_memory\_load

חתימת הפונקציה צריכה להיות בסגנון הבא:

void bgu\_do\_physical\_memory\_load(Monitor \*mon, <PARAMETERS>)

כאשר מספר הפרמטרים שמופיעים במקום <PARAMETERS> וסוגיהם תלויים במחרוזת ה-args\_type הנ"ל של הפקודה.

* params – תיאור בסטייל command-line של הארגומנטים של הפקודה.

שמות הארגומנטים לא צריכים להתאים לחתימת הפונקציה; מחרוזת זו משמשת בכדי להסביר למשתמש כיצד להשתמש בפקודה.

דוגמה: addr size file

כאשר המשתמש של האמולטור יפעיל את פקודת help (לאחר טעינת ה-plugin שלכם), תוצגנה עבורו הפקודות החדשות, ועמן הפרמטרים שלהן – מה שתכתבו כאן יוצג בתור רשימת הפרמטרים. למשל:

pmemload addr size file

* help – הסבר על הפקודה.

מה שתכתבו כאן יוצג למשתמש במידה והוא יבקש עזרה לגבי הפקודה בעזרת הפקודה "help <COMMAND\_NAME>".

דוגמה: 'load 'size' bytes from file 'file' to physical memory at address 'addr

## תיאור קובץ המקור של המימוש

בקובץ המקור של ה-plugin יש את הקוד של כל הפקודות החדשות (אני יצרתי את פקודת pmemload ופקודת hello, ועוד פקודה שבסוף לא עבדה אז עזבתי אותה).  
אני מאמין שהקוד של אתחול ה-plugin, שהוא פונקציית ה-**init\_plugin**, די ברור כפי שהוא.  
יש לשים לב לשני דברים בנוגע לכך:

1. אם לא קובעים פונקציית cleanup (כלומר אם שמים NULL), אז האמולטור קורס אם וכשמנסים להוריד את ה-plugin מהזיכרון.
2. את המערך BGU\_term\_cmds, אשר מכיל את התיאורים של הפקודות מהקובץ **plugin\_cmds.h**, יש להגדיר אחרי שמוגדרות הפונקציות המטפלות בפקודות (או לפחות אחרי הצהרה שלהן).

בנוסף, שימו לב שלאחר ה-#include של קובץ הפקודות יש להוסיף לאתחול עוד איבר אחד שערכיו הם NULL.

הקוד של **bgu\_do\_physical\_memory\_load**, שמטפל בפקודת pmemload שהוספתי, מבוסס על הקוד של **do\_physical\_memory\_save**, שמטפל בפקודת pmemsave, אשר מובנה באמולטור (הקוד שלו גם הוא נמצא בקובץ "**monitor.c**").

## קימפול ה-plugin והפעלתו באמולטור

האתגר הגדול ביותר בקימפול ה-plugin הוא לגרום לו לקמפל גרסה עבור מעבד של 32-bit (כי נראה שהאמולטור הוא תוכנה של 32-bit).  
לשם כך היה עליי לשנות כמה דברים ב-Makefile.  
כמובן, כיוון שכבר שיניתי אותם, אתם יכולים פשוט להתבסס על ה-Makefile שלי.  
בכל מקרה, השינויים שעשיתי היו הוספת הדגל "-m32" לדגלי הקומפיילר (המשתנה "CFLAGS" ב-Makefile) והוספת הדגל " -Xlinker -melf\_i386" לדגלי הלינקר (המשתנה "LDFLAGS" ב-Makefile).  
באופן קצת מוזר, הדגלים של הלינקר מועברים בכלל לקומפיילר, ולכן צריך את התוספת של  
"-Xlinker", אשר מורה לקומפיילר להעביר את הדגל הבא ללינקר.

לאחר שקימפלתם את ה-plugin (או אם אתם לוקחים את ה-plugin המקומפל שלי), וסידרתם את הפעלת האמולטור החדש, אתם יכולים לטעון את ה-plugin לאמולטור בזמן ריצה דרך ה-shell שלו.

הפקודה לטעינת ה-plugin היא:

load\_plugin <PATH\_OF\_PLUGIN\_FILE>

כאשר <PATH\_OF\_PLUGIN\_FILE> הוא הנתיב ל-plugin, כמובן.  
ה-shell של האמולטור יודע לזהות את מערכת הקבצים (לפחות עבור הפקודה load\_plugin), לכן מטעמי נוחות סידרתי שה-build של ה-plugin שלי יעתיק את הקובץ הסופי לתיקיית "**plugins/**" שפתחתי לצד האמולטור.

התוצר שה-build של ה-plugin אמור ליצור הוא קובץ עם סיומת "**.so**" (למשל "**bgu.so**").  
בהנחה שהעתקנו אותו לתיקיית "**plugins/**" שנמצאת לצד האמולטור, פקודת הטעינה לאמולטור תהיה:

load\_plugin plugins/bgu.so

הוא אמור לכתוב לכם שהטעינה הצליחה.  
אם אתם מקבלים שגיאה, יתכן שזה משום שה-plugin שלכם קומפל עבור 64-bit; אתם יכולים לבדוק זאת ע"י הרצת הפקודה:

objdump –f <PATH\_OF\_PLUGIN\_FILE>

ע"מ לפרוק plugins מהזיכרון, יש להשתמש בפקודה דומה (נדמה לי ששמה "unload\_plugin", או משהו כזה).  
היא לא שואלת אתכם מה היא מורידה (את כולם? את האחרון? לא בדקתי).  
כמובן, אם אתם מבצעים זאת עם plugin שעבורו הגדרתם פונקציית cleanup שהיא NULL, אז פה האמולטור כאמור יקרוס. ככה זה כשלא מבצעים null checks...

# 3. הכנת Eclipse לעבודה עם ה-Kernel ועם NDK

קודם כל, עליכם להשתמש בגרסת Eclipse שעובדת עם ה-SDK של Android ע"מ להפעיל את הפרויקטים של האפליקציות השונות.  
בנוסף, על Eclipse שלכם לתמוך בפרויקטים של C/C++ (ניתן להוריד עדכון לכך מהאינטרנט).

ע"מ ש-Eclipse יוכל למצוא את קבצי ה-h בפרוייקטי ה-Kernel Module ובאפליקציות המשלבות ++C עם Java, יש להגדיר עבורם את ה-Include Paths המתאימים.  
כמו-כן, עבור Kernel Modules יש להוסיף למאפייני הפרוייקט את הגדרת הסימבול \_\_KERNEL\_\_, שכן חלק מקבצי ה-h בודקים את קיום סימבול זה (ע"י #ifdef), ואם Eclipse לא יודע שבעת הבניה זה צריך להיות מוגדר, הוא חושב שקוד זה לא נכנס, ואז יתכן שהוא לא יכיר בכל ההגדרות.

כיוון שאנו משתמשים ב-Makefiles משלנו, ו-Eclipse לא באמת עושה את הבנייה, אם אין את ההגדרות הללו, עדיין יהיה אפשר לקמפל את הקוד, אבל לא נוח לקרוא ולערוך אותו מ-Eclipse (גם כי אי-אפשר לקפוץ להגדרות של מבנים ומשתנים וגם כי הוא מסמן הרבה מהקוד באדום, וזה מציק למדי – במיוחד עבור מתכנת שהוא שור, אני מתאר לעצמי).

## הגדרת משתני סביבה

אני קינפגתי מראש את כל הפרוייקטים לעבוד עם משתני סביבה של Eclipse במקום עם absolute paths, כך שלמעשה מספיק שתגדירו מספר משתנים בהגדרות של Eclipse, ואז בתקווה הפרויקטים כבר יעבדו לכם כפי שהם.

כשעשיתי זאת בעצמי, לא שמתי לב לכך שיש שני מקומות שבהם ניתן להגדיר משתנים, והגדרתי את חלקם באחד (בנתיב הקונפיגורציה *Preferences🡪C/C++🡪Build🡪Build Variables*) ואת חלקם בשני (*Preferences🡪C/C++🡪Build🡪Environment*).  
לפי שמותיהם, אני חושב שהראשון מוסיף משתנים רק בעת קומפילציה והשני ניתן לשימוש בכל רחבי ה-workspace. בכל מקרה, העיקר שהם יוגדרו במקום כלשהו ושזה יעבוד.

להלן הערכים שיש להגדיר בסביבה:

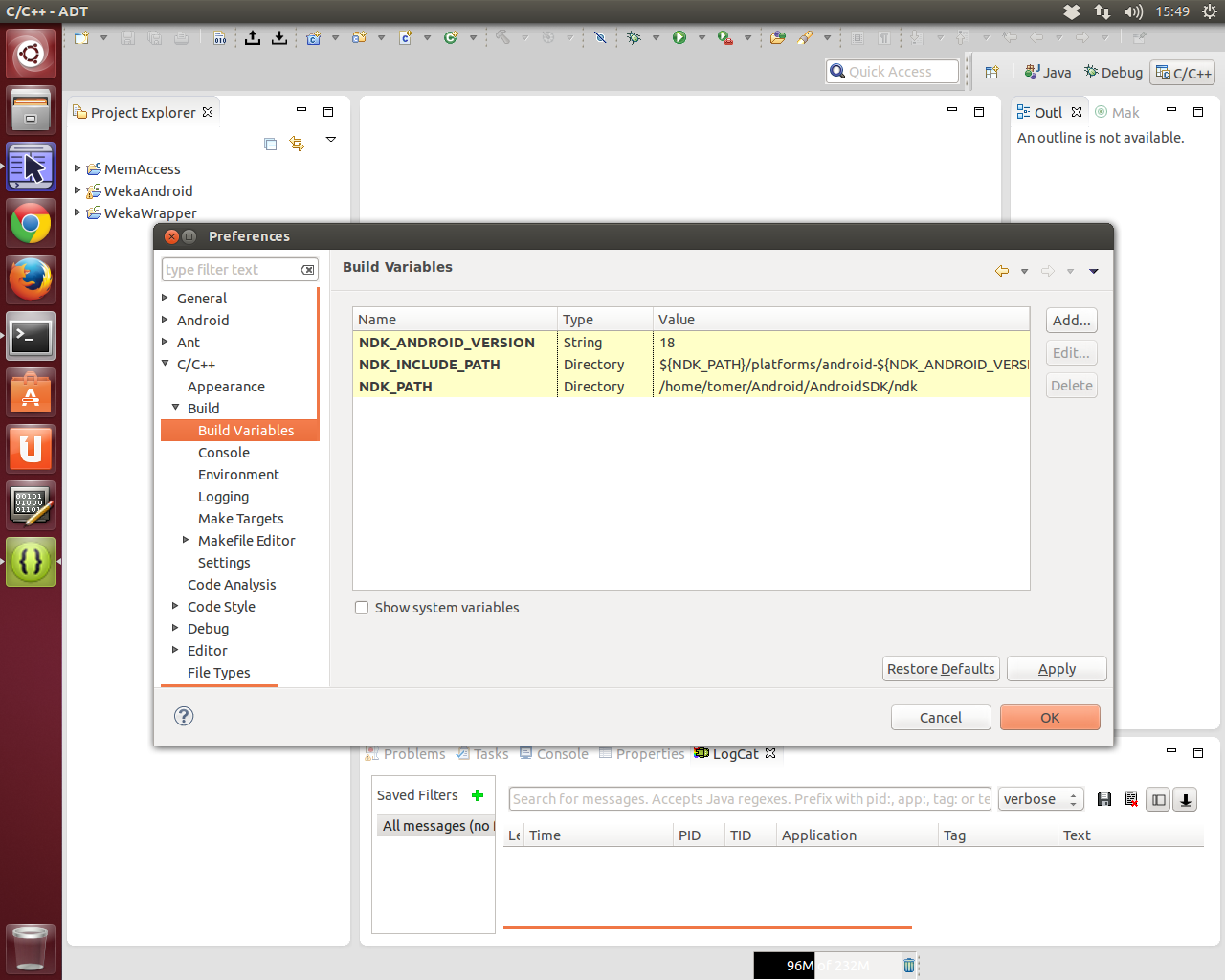
|  |  |
| --- | --- |
| ערך | שם משתנה |
| <LOCATION\_OF\_GOLDFISH\_WEKA> | WEKA\_KERNEL\_DIR |
| ${WEKA\_KERNEL\_DIR}/include | WEKA\_KERNEL\_INCLUDE\_DIR |
| ${WEKA\_KERNEL\_DIR}/arch/arm/include | WEKA\_ARM\_INCLUDE\_DIR |
| <LOCATION\_OF\_NDK> | NDK\_PATH |
| <SELECTED\_ANDROID\_VERSION> | NDK\_ANDROID\_VERSION |
| ${NDK\_PATH}/platforms/android-${NDK\_ANDROID\_VERSION}/arch-arm/usr/include | NDK\_INCLUDE\_PATH |

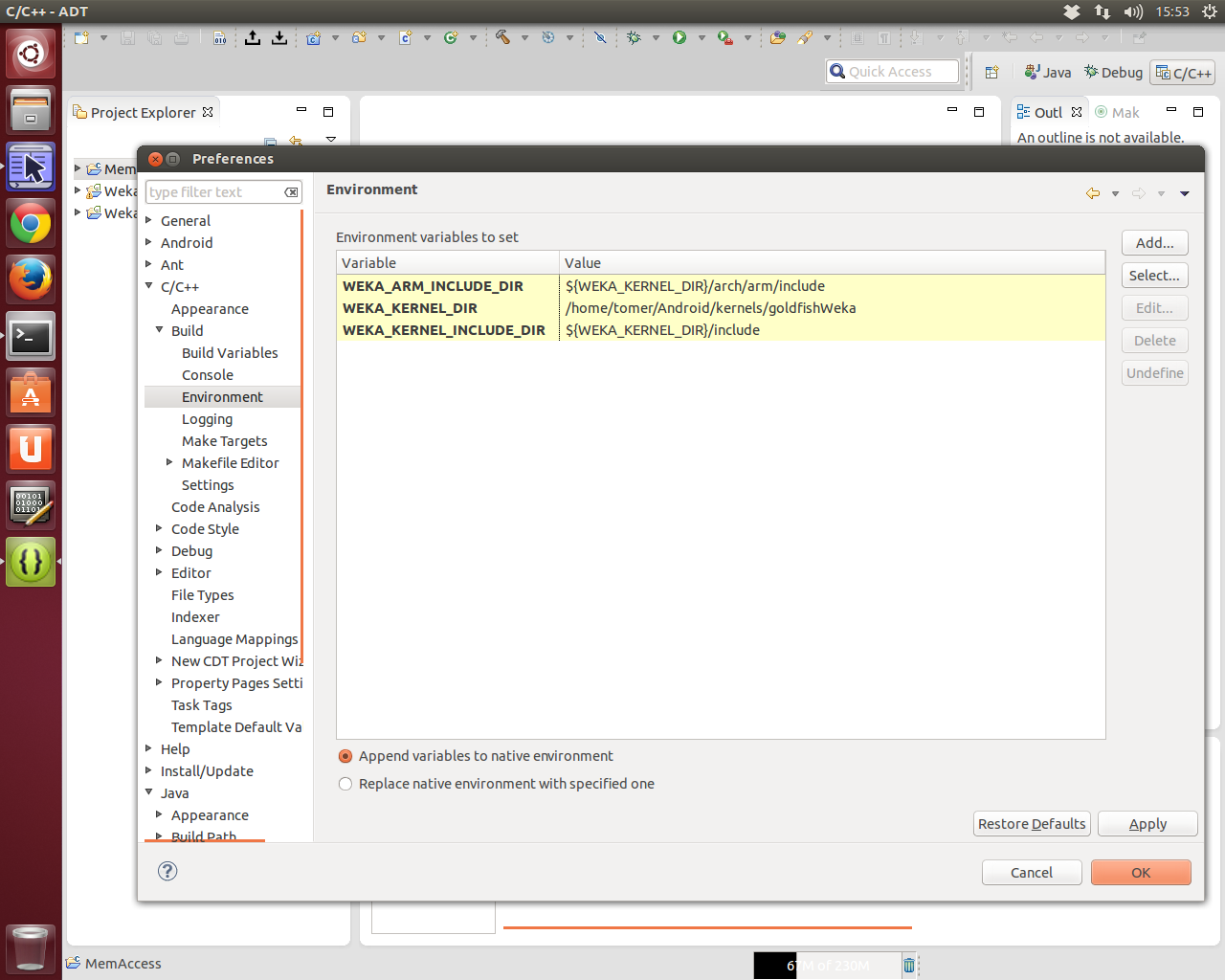
כאשר "<LOCATION\_OF\_GOLDFISH\_WEKA>" הוא המיקום בו נמצאת אצלכם התיקייה "goldfishWeka" (כלומר ה-Kernel של Android; אסביר עליו בהמשך), וְ-"<LOCATION\_OF\_NDK>" הוא מיקום תקיית ה-NDK (התיקייה בה נמצא הקובץ "**ndk-build**").  
את ה-Kernel שבו אני השתמשתי ניתן למצוא בחבילה, בתיקיה "**goldfishWeka**".  
את ה-NDK ניתן להוריד מהאינטרנט; חיפוש קצר ב-Google הניב עבורי את הכתובת הבאה:

<http://developer.android.com/tools/sdk/ndk/index.html>

לגבי "<SELECTED\_ANDROID\_VERSION>": אני חושד שכדאי שזו תהיה גרסת ה-Android שבה הפרויקט המשולב של Java/C++ שלכם משתמש, אבל אני פשוט הלכתי לתיקייה "**${NDK\_PATH}/platforms/** " אצלי ובחרתי את המספר הגבוה ביותר מבין התיקיות "**android-#**" שהיו שם. זכרו שזה לא באמת משפיע על ה-build, אלא רק על מה Eclipse מזהה.

להלן תמונות של הקונפיגורציה שאני קבעתי:





ניתן למצוא את התמונות הללו – יחד עם תמונות נוספות של כל השינויים שעשיתי לפרויקטים השונים ע"מ ליצור את הסביבה הנוחה שיצרתי – בתיקייה "**EclipseScreenshots**".

# 4. The MemAccess Kernel Module

את הקוד של מודול זה תוכלו למצוא בתיקייה "**Workspace/MemAccess**".  
כמו-כן, התיקייה כוללת קבצי פרויקט עבור Eclipse וקובץ Makefile בשביל הבנייה ובשביל טעינת המודול לאמולטור פועל (או למכשיר מחובר, אם הוא מאפשר זאת).

## תיאור פעולת המודול

המודול הוא למעשה driver.  
הוא מגדיר device בשם "MemAccess", ומכיל פונקציות המטפלות בקריאות open, close, ioctl וְ-read.  
מתוך כולן, הפונקציה היחידה עם תוכן מעניין היא read, שכן בכל קריאה מה-driver, המודול פותח את ה-device הקבוע במערכת "/dev/mem", אשר מאפשר גישה ישירה לזיכרון הפיזי וקורא ממנו את כמות הבתים המבוקשת החל מכתובת קבועה מראש (שנקבעת בעת הקומפילציה של המודול; זהו הקבוע **BASE\_ADDRESS** אשר מוגדר ב-"**MemAccess.h**") לתוך ה-buffer הנתון.

## גישה ל-/dev/mem

לשם קבלת הרשאות גישה ל-"/dev/mem" על ה-process להיות בעל הרשאות מסוימות.  
לא התעמקתי יותר מדי באם ניתן להשיג את ההרשאות הללו מתוך User Space מבלי גישת Root או עזרה מה-Kernel (כרגע ה-Kernel Module הופך כל מי שמשתמש בו לבעל הרשאה כזו) – אך כן ראיתי שלא דרוש לשם כך שה-process יהיה בעל UID שהוא 0 (כלומר שיחשב כ-Root).  
מה שיש לשנות הוא את ערכם של שני שדות במאפייני ה-process, אשר שמורים במערך **cap\_effective.cap** של מבנה ה-credentials שלו (זהו מערך עם שני תאים מסוג **int**).  
אם יש בשני התאים את הערך -1, אז ה-process יכול לגשת ל-"/dev/mem" מבלי לקבל שגיאה.

יתכן שמספיק לשנות רק ערך אחד, ויתכן שלא חייבים לשנותם ל--1; לא בדקתי זאת.  
בכל מקרה, הפונקציה **setcreds** בקובץ "**MemAccess.c**" משנה את ההרשאות הללו עבור ה-process הנוכחי.  
הקובץ מכיל גם קוד עבור שינוי ה-UID וה-GID (**setids**), וכן פונקציה הבודקת את ההרשאות של ה-process הנוכחי (**creds\_test**).

## קריאת קבצים מה-Kernel

באופן כללי, ה-Kernel של Linux (עליו מבוסס ה-Kernel של Android) מאוד לא אוהב את הרעיון של פתיחת קבצים מתוך ה-Kernel. לשם כך היה עליי לגשת לפונקציות ברמה קצת גבוהה מפונקציות הגישה עצמן (**vfs\_read** במקום **sys\_read**), והיה עליי גם לשכנע את ה-Kernel שזה בסדר שהכתובות שהוא מקבל לשם כך מגיעות מתוך Kernel Space ולא User Space (זה מה שהפונקציה **set\_fs** עושה, פחות או יותר).  
כל הקוד שעושה את הפתיחה, הסגירה והקריאה מקובץ נלקח מהמקור הבא:

<http://stackoverflow.com/questions/1184274/how-to-read-write-files-within-a-linux-kernel-module>