מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 42
מרץ 2002

יעילות C++ ותמיכתה במערכות זמן אמת

שפת C++ מאפשרת לכתוב קוד יעיל במיוחד - הן בהשוואה לשפות תכנות מונחות עצמים כמו Java, והן ביחס לשפות מבניות כמו C. יתרון זה הופך אותה לבעלת עדיפות ראשונה כשפת פיתוח למערכות זמן אמת.

מערכות זמן אמת הן מערכות הפועלות במסגרות זמן ומקום - קבועות או חסומות - ידועות מראש. מערכות אלה הן בדרך כלל משובצות מחשב, כלומר, מכילות רכיבי חומרה שונים כגון: מעבד, זיכרונות, רגיסטרים, FIFOs, שעונים ועוד.

כיום, מרבית מערכות זמן אמת ממומשות בשפות C/C++, הכוללות תמיכה מלאה בתכנות ברמת מכונה (machine level) ע"י הוראות כגון: volatile, register ו- asm וכן ע"י תמיכה בשדות סיבית במחלקה. לאופרטורי הקצאת הזיכרון והשחרור של C++, new ו- delete, קיימת גרסה להקצאת זיכרון בכתובת נתונה (placement).

במאמר זה נתמקד במנגנוני שפת C++ התומכים בתכנות מערכות זמן אמת. כמו כן נבצע השוואה של יעילות תכנות מונחה עצמים ב- C++ הן ביחס לתכנות ב- Java והן ביחס לתכנות מבני ב- C.

המאמר מבוסס על הפרק האחרון (פרק 18) בספר "***C++ - מדריך מקצועי***" היוצא לאור בימים אלו.

יעילות

ליעילות שני פרמטרים עיקריים: **מקום** ו**זמן**.

- **מקום**: אנו מעונינים לחסוך בכמות הזיכרון שצורכת התכנית. לפעמים משאבי הזיכרון של התכנית מוגבלים ממש (למשל במערכות משובצות).

  מרכיבי פרמטר המקום:

  - קטע הנתונים (DATA) - גדלו קבוע, מושפע מהעצמים הגלובליים והסטטיים.
  - קטע הקוד (CODE) - גדלו קבוע, מורכב מקוד הפונקציות בלבד. חסכון בקוד משפיע גם על משך הפיתוח של התכנית: בכל שורת קוד מושקע זמן פיתוח מסוים: תיכון, מימוש, בדיקה, תיעוד, תחזוקה וכו'.
  - מחסנית הקריאות (STACK) - גודל משתנה, נקבע באופן דינמי ע"י הקריאות לפונקציות. מכיוון שכאשר פונקציה מסתיימת היא יוצאת מהמחסנית, לעצמים המקומיים השפעה שולית על צריכת הזיכרון.
  - הערימה (HEAP) - גודל משתנה, נקבע באופן דינמי ע"י ההקצאות הדינמיות ושחרורי הזיכרון במהלך ריצת התכנית.

- **זמן**: משך הזמן שבו מבוצע קטע קוד מסוים הוא לפעמים קריטי - המטרה היא לייעל את הקוד כך שירוץ במינימום זמן.

  מרכיבי פרמטר הזמן:

  - אתחול עצמים: משך הזמן שצורכים ה- constructors, משך הזמן הנדרש להקצאות זיכרון על הערמה.
  - ביצוע קוד פונקציות: התקורה שבקריאה לפונקציה, קוד הפונקציה.
  - שחרור עצמים: משך הזמן הצורכים destructors, שחרור עצמים בערמה.

יעילות: ++C ו- Java

Java נחשבת כשפה פופולרית מאוד עקב פשטות התחביר שלה, טבעיות הפיתוח לסביבה מבוזרת בכלל ולאינטרנט בפרט, ואי-התלות שלה במכונה. עם זאת, מבחינת יעילות, ל- ++C יתרון גדול על פני Java: מערכות הכתובות ב- ++C רצות ביעילות הגבוהה בסדר גודל מאלו הכתובות ב- Java.

במערכות מסוימות, ליעילות המקומית השפעה קטנה: לדוגמא, בפיתוח לאינטרנט, לרוב צוואר הבקבוק מבחינת זמן הריצה הוא תשתית האינטרנט, וגם אם תוכנה ב- Java רצה פי 10 לאט יותר מתוכנה מקבילה ב- ++C, המשתמש לא ירגיש בהבדל.

בקצה השני, במערכות זמן אמת ליעילות השפעה קריטית, ולכן לרוב ++C/C מועדפות כשפות הפיתוח (קיימים מספר מאפיינים נוספים של Java המונעים את בחירתה כשפה לפיתוח מערכות משובצות).

נתבונן בשני קטעי הקוד הבאים - יצירת מלבן המוגדר ע"י שתי נקודות, פעם ע"י הכלת עצמים, כפי שמאפשרת ++C, ופעם ע"י הכלת מצביעים - האפשרות היחידה הקיימת ב- Java:

הכלת עצמים - ++C	**הכלת מצביעים - סגנון Java**
class Rect1 *{* *Point m_p1;* *Point m_p2;* *public:* *Rect1(int x1, int y1, int x2, int y2)* *: m_p1(x1,y1), m_p2(x2,y2) {}* *};* *Rect1 r1(10,10,50, 50);*	*class Rect2* *{* *Point m_p1;* *Point m_p2;* *public:* *Rect2(int x1, int y1, int x2, int y2)* *{* *m_p1 = new Point(x1,y1);* *m_p2 = new Point(x2,y2);* *}* *}* *Rect2 r2 = new Rect2(10,10,50, 50);*

<u>הסבר</u>: מצד ימין, Rect1 מוגדר כמכיל שני עצמי Point, בתחביר הכלה של ++C. מצד שמאל, מובא הקוד המקביל ב- Java: כל העצמים ב- Java מיוצגים ע"י מצביע, וכולם מוקצים על ה- HEAP (ולכן אין צורך בסימן "*" לציון מצביע).

ננתח את יעילות שני קטעי הקוד, עפ"י המקום ומשך הזמן שהם צורכים:

- **מקום**:
  - המקום שצורך העצם r1 הוא המקום שצורכים שני עצמי Point, סה"כ 4*sizeof(int).
  - r2 לעומת זאת צורך בנוסף 3 מצביעים: מצביע ל- r2 ושני מצביעים לשני עצמי ה- Point שהוא מכיל. כמו כן, על ה- HEAP יש צורך להחזיק לפחות מצביע אחד לכל הקצאה דינמית שמבוצעת, ולכן קיימים עוד 3 מצביעים.

    כך שבסך הכל r2 צורך 4*sizeof(int) ועוד 6 מצביעים.
- **זמן**:
  - ההקצאה של r1 היא מיידית וניתן לומר שהיא צורכת "אפס זמן". גם שחרור הזיכרון הוא מיידי.
  - הקצאה של r2 כוללת הקצאה של 3 עצמים על הערימה, פעולה איטית בהרבה בהשוואה להקצאה על ה- STACK או על ה- DATA.

    גם פעולת השחרור של r2 איטית בהרבה בהשוואה לזו של r1: השחרור מבוצע ע"י ה- Garbage Collector, שצריך לעבור על כלל הזיכרון מידי זמן מה, לסמן את כל העצמים שבשימוש, ולשחרר את אלו שאינם מוצבעים. לכן באופן ממוצע, לכל מצביע לעצם מוקדשים מספר "ביקורים" של ה- Garbage Collector במהלך חייו.

הדוגמא ממחישה היטב את הגורמים ליעילות הנמוכה של Java ביחס לזו של ++C. יש לשים לב בעיקר לפרמטר הזמן, שהוא החשוב מהשניים: מכיוון שהקצאות ושחרורים של עצמים מתרחשות בחלק נכבד של התכנית, הן המשפיעות העיקריות על איטיותה של Java ביחס ל- ++C.

כיצד Java מתמודדת עם חוסר יעילות זו? ב- Java קיימים מספר מנגנונים שנועדו לשפר את יעילותה: כלי אופטימיזציה חזקים שעקב פשטות השפה מייעלים משמעותית את קוד התכנית, טכניקות הקצאה ושחרור יעילות (Thread-local heap, cache), כתיבת ספריות שימושיות בקוד מכונה JNI, טכניקת JIT Compiling - הידור של קטעי קוד עפ"י צורך, ועוד.

יעילות: ++C ו- C

++C כוללת מספר מנגנונים המשפרים את יעילותה ביחס ל- C. למעשה, באמצעות תכנות נכון ומחושב ב- ++C ניתן לכתוב תכניות יעילות בהרבה מכאלו הנכתבות ב- C.

הטבלה הבאה מסכמת את המנגנונים המשפיעים על יעילות התכנות, ואת מימושם ב- ++C לעומת C:

מנגנון	C	++C
קריאה לפונקציות	פונקציות מאקרו	פונקציות inline
יצירת עצמים	-	ניתן להגדיר עצמים בכל מקום בתכנית, ולכן כדאי להגדיר עצמים לפני השימוש בהם
העברת עצמים גדולים כפרמטרים לפונקציות	העברת מצביע	העברת מצביע או reference
מבני נתונים ואלגוריתמים	המתכנת נדרש לממש בעצמו את מבני הנתונים, היעילות המושגת היא עפ"י יכולתו	שימוש בספרייה התקנית **STL**: מבני נתונים ואלגוריתמים כלליים ויעילים מאוד
שיתוף נתונים	שימוש במצביעים	שימוש ב- Design Patterns כגון: Proxy,Reference Count
מכנה משותף בין מספר מבנים	שימוש בהכלה	שימוש בירושה
ביצוע קוד כתלות במצב משתנים	משפטי switch-case	פולימורפיזם (פונקציות וירטואליות), שימוש ב- State Pattern
טיפול בשגיאות ובחריגות	בקרת שגיאות ע"י החזרת ערך שגיאה או שימוש במשתנים גלובליים.	בקרת שגיאות ע"י מנגנון החריגות (Exceptions): טיפול עפ"י צורך (On Demand)

כפי שניתן לראות, ++C מספקת מנגנונים ומרכיבי ספרייה המאפשרים כתיבת תוכניות יעילות מאלו הנכתבות ב- C.

- פונקציות ה- **inline** הן אחד הגורמים העיקריים ליעילות הגבוהה של קוד ++C.
- תכנות מונחה עצמים ע"י **פולימורפיזם** הוא במקורו גישה יעילה יותר מביצוע משפטי תנאי או switch-case. כמו כן קיימים Design Patterns שונים (ראה/י פרק 16 בספר) כפתרונות יעילים במיוחד לבעיות תכנות שכיחות כגון: **State**, **Observer**, **Command**, **Composite** ועוד.
- הספרייה התקנית ממומשת בגישת התכנות הגנרי - התבססות על מנגנון ה- templates - ומספקת מבני נתונים ולאגוריתמים יעילים במיוחד.
- מנגנון ה- **Exceptions** מאפשר להגדיר טיפול עפ"י צורך (On-Demand) בחריגות המתגלות בתכנית. בניגוד לסיגנון בקרת השגיאות בשפות פורצדורליות כגון C, המחייבות בדיקת שגיאות חוזרת בכל קריאה לפונקציה (צריכת זמן+מקום), ב- ++C הטיפול מבוצע במקום אחד ובזמן אירוע החריגה, כך שהזמן והמקום שצורך מנגנון זה הוא מינימלי.

תמיכת C++ במערכות זמן-אמת

volatile

אחד השלבים בהידור תכנית הוא אופטימיזציה. בשלב זה המהדר מנסה לבצע פעולות שונות בכדי לזרז ולייעל את פעולת התכנית. אחת מפעולות הייעול היא שמירת ערכו של משתנה שניגשים אליו פעמים רבות בקטע קוד מסוים ברגיסטר מהיר גישה.

במקרים מסוימים אנו מעונינים למנוע את האופטימיזציה. לדוגמא, כאשר משתנה מסוים עלול להשתנות ע"י מערכת ההפעלה, ע"י החומרה או ע"י תהליך אחר עליו להיות מאוחסן בזיכרון ולא ברגיסטר.

volatile הוא מציין בהגדרת משתנה המורה למהדר להימנע מלבצע עליו אופטימיזציה, ולא לשמור אותו ברגיסטר בזמן השימוש בו. דוגמא:

```
volatile int hw_flag;
```

register

register מצוין בהקשר להגדרת משתנה המעורב באופן מסיבי בקוד, ולכן ממליץ למהדר לשמור אותו ברגיסטר כדי לייעל את פעולת התכנית. למעשה register הוא הוראה הפוכה ל- volatile. דוגמא:

```
register int counter;
```

יש לשים לב לכך שהמהדר לא מחויב להישמע להמלצה. יתרה מזאת, עקב האופטימיזציות המתקדמות של מהדרים מודרניים, מרביתם יתעלמו מהמלצת register של המתכנת.

שדות סיביות במחלקה

ניתן להגדיר מחלקות ששדותיהן אינן מכילות מילים שלמות, אלא חלקי מילים בגודל של מספר משתנה של סיביות. לדוגמא:

```
class X
{
public:
    unsigned int d1:1 ;        // 1 bit field
    unsigned int d2:2 ;        // 2 bit field
};

X x1;
x1.d1 = 1;
x1.d2 = 3;
```

הסבר: שדות סיביות מוגדרים בצירוף אורך השדה לאחר הנקודתיים. כל פעולות ההתאמה של משתנים שלמים לשדות הסיבית (פעולות & ו- | על סיביות) מבוצעות ע"י המהדר.

כמובן, רצוי להגדיר את שדות הסיביות כ- private, ולספק פונקציות גישה public. כמו כן, את ערכי המשתנים ניתן להגדיר ע"י enum. לדוגמא, הגדרת מחלקה המייצגת רגיסטר סטטוס של החומרה בעל אורך של 7 סיביות:

```
class StatusReg
{
public:
    enum Mode {                 // mode of work
        NORMAL=0,               // normal work: transmit and receive
            DEBUG=1,            // debug mode
            LOOP_BACK=2,        // loop-back mode
            BIT=3               // Built-In Test mode
```

```
    };
    enum BITResult {            // Built-In Test (BIT) result
        BIT_OK=0,               // BIT result OK
        RADIO_ERROR=1,          // radio unit test error
        MEM_ERROR=2,            // memory unit test error
        FIFO_ERROR=3            // FIFO unit test error
    };
    int     getMode()           const { return m_mode; }
    int     getBITResult()      const { return m_BITResult; }
    bool    isOverflow()            const { return m_fifoOverflow; }
    bool    isUnderflow()       const { return m_fifoUnderflow; }
private:
    volatile    unsigned char   m_mode:2;
    volatile    unsigned char   m_BITResult:3;
    volatile    bool            m_fifoOverflow:1;
    volatile    bool            m_fifoUnderflow:1;
};
```

<u>הסבר</u>:

מכיוון שזהו רגיסטר המיועד לקבלת סטטוס מהחומרה, שדות המבנה מוגדרים כ- **volatile**. כל שדה במחלקה כולל מספר סיביות. אנו מתייחסים לכל אחד מהשדות כאילו היה משתנה שלם רגיל. המהדר דואג לביצוע ההזזות ופעולות על הסיביות המתאימות.

> *הערה: בפועל המהדר מקצה לכל רשומה כנ"ל מספר שלם של מילים (integers). כאשר סה"כ השדות קטן ממספר מילים שלם המהדר מבצע ריפוד באפסים.*

השדה m_mode הוא בעל 2 סיביות ולכן יכול לקבל 4 ערכים בינריים שונים:

```
00, 01, 10, 11
```

הגדרה נוחה של הערכים היא לכן ע"י enum:

```
enum Mode {             // mode of work
    NORMAL=0,           // normal work: transmit and receive
    DEBUG=1,            // debug mode
    LOOP_BACK=2,        // loop-back mode
    BIT=3               // Built-In Test mode
};
```

באופן דומה מוגדר הטיפוס **BITResult**.

הקבועים מציינים מצבים שונים של מערכת תקשורת: מצב נורמלי, מצב ניפוי, מצב "לולאה" או מצב בדיקה (BIT). כעת ניתן לקרוא את ערכי השדה כך:

```
StatusReg  sr;
if(sr.getMode()==StatusReg::BIT)
    // system is in BIT state
```

הוראת asm

הוראת asm היא הוראה המתחילה בלוק הכתוב בשפת אסמבלר, כלומר שפת המחשב עליו עובדים. הוראה זו אינה חלק משפת C אך מרבית המהדרים הקיימים תומכים בה.

בדוגמא הבאה, מופעל קוד אסמבלר של מעבדי אינטל XXX86, פעם במסגרת בלוק ופעם כהוראות נפרדות:

```
asm                     /* asm block */
{
```

```
   mov eax, 01h
   int 10h
}

asm mov eax, 01h      /* Separate asm lines */
asm int 10h
```

הקצאת עצם בכתובת נתונה

פרמטר מיקום (Placement) לאופרטורים new ו- delete

לאופרטורים new ו- delete קיימת גרסה המקבלת פרמטר נוסף: גרסה זו נקראת גרסת placement, והיא מאפשרת לקבוע את המקום בזיכרון בו יוקצה העצם (להרחבה, עיין/י בפרק 6, **Operator Overloading**, בספר "**C++ - מדריך מקצועי**").

ביישומי זמן אמת רבים מעוניינים לייצר עצמים בכתובת אבסולוטית בזיכרון. נניח שנתונה מחלקה בשם Register המייצגת רגיסטר זיכרון:

```
class Register
{
        char m_data;
public:
        Register(char data) { m_data = data;}
        write(char c);
        char read();
};
```

כעת נניח שאנו מעוניינים ליצור רגיסטר בכתובת האבסולוטית 0xE5F0 בזיכרון של המערכת המשובצת. ראשית נגדיר מצביע מסוג Register ונאתחלו באופן הבא:

```
Register *p = reinterpret_cast<Register*> (0xE5F0);
```

האופרטור reinterpret_cast פועל בדומה ל- casting המוכר מסגנון C:

```
Register *p = (Register*) 0xE5F0;
```

וכעת ניתן לייצר עצם מסוג Register בכתובת המאוחסנת ב- p ע"י:

```
Register *pr = new (p) Register(0x12); // pr=p=0xE5F0,   pr->m_data=0x12
```

כלומר, במקום לייצר את המקום בזיכרון על הערימה, הזיכרון מיוצר בכתובת הניתנת כפרמטר לאופרטור new.

באופן מקוצר, ניתן להקצות עצם המוצבע ע"י pr בכתובת נתונה ישירות כך:

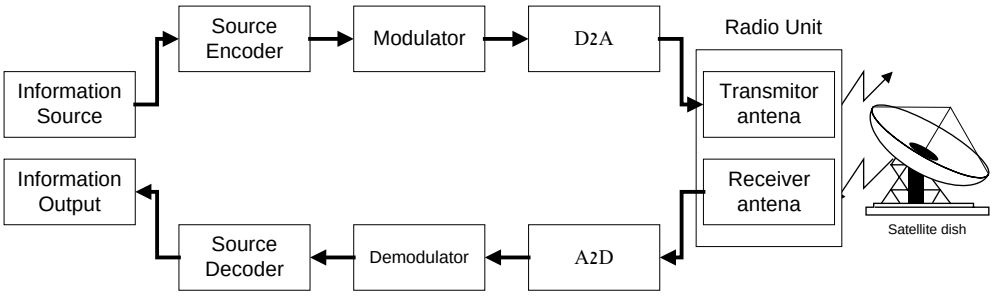
```
Register *pr = new ((Register*) 0xE5F0) Register(0x12);
```

דוגמא: מודם אלחוטי- מימוש ב- C++

בעדכון מספר 32 סקרנו מערכת מודם אלחוטי שמומשה בשפת C. כדוגמא מסכת למאמר זה נראה כיצד ממשים מערכת זו ב- C++.

נתחיל בתזכורת לגבי מבנה המודם, התוכנה והחומרה:

המבנה הלוגי של המודם:



משדר - תת-מערכת הכוללת את המרכיבים הבסיסיים הבאים:

מקור מידע (Information source) - המקור ממנו מגיע המידע לשידור

קידוד מקור (source coding) - קידוד המידע ברמת המקור. מיועד להתגברות על שגיאות מידע בקלט. דוגמאות לשיטות קידוד: קידוד ויטרבי, קידוד ריד-סולומון.

אפנון (Modulation) - קידוד המידע עפ"י שיטת השידור. שיטות אפנון נפוצות: אפנון תדר, אפנון פאזה, אפנון משרעת, CDMA, GSM.

ממיר ספרתי-לאנלוגי (**D2A**) - המרת המידע מייצוג ספרתי לייצוג אנלוגי.

אנטנת שידור- לשידור האות לאוויר.

מקלט - תת-מערכת הכוללת את המרכיבים ההפוכים למשדר:

אנטנת קליטה - לקליטת האות שבאוויר.

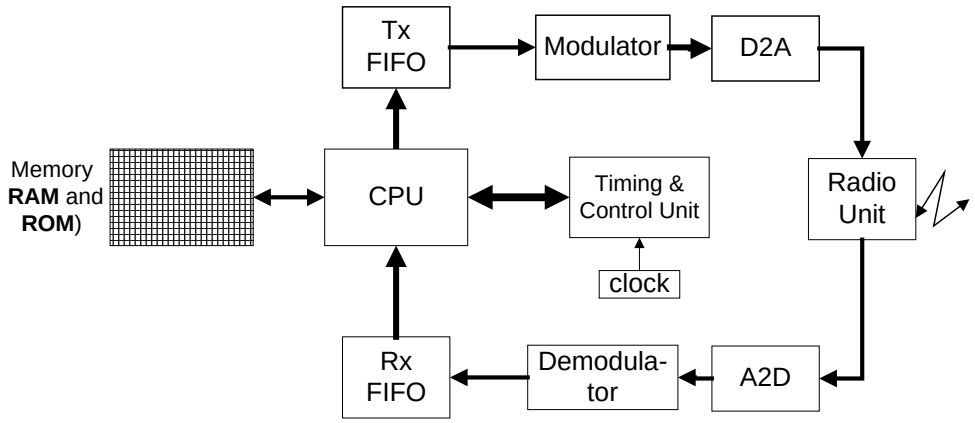
ממיר אנלוגי-לספרתי (**A2D**) - המרת האות הנקלט מייצוג אנלוגי לייצוג ספרתי.

דה-אפנון (Demodulation) - קידוד הפוך (פענוח) לאפנון שבוצע בשידור.

קידוד מקור הפוך (source decoder) - קידוד הפוך לקידוד המקור שבוצע בשידור.

יעד הפלט (Information output) - היחידה אליה נשלח המידע הנקלט.

המבנה פיזי של המודם :



קווי הבקרה והתזמון מיחידת הבקרה לכל רכיבי המערכת הושמטו בכדי לפשט את השרטוט.

ה**זיכרון** (**memory**) מכיל שני חלקים עיקריים: ה- **ROM** (Read Only Memory) הוא מקום האחסון של תוכנת המערכת, כלומר של קובץ הביצוע. בהפעלת המכשיר, התוכנה מתחילה להתבצע מכתובת מסוימת ב- ROM. ה- **RAM** (Raw Access Memory) הוא זיכרון המיועד לפעולות חישוב ולאחסון תוצאות הביניים.

המעבד (CPU) היא יחידת החישוב של המערכת. בכל רגע נתון מתבצעת במעבד הוראה אחת מתוך קוד התכנית לביצוע. המעבד קורא את ההוראה הבאה מקטע הקוד (Code segment) ומבצע אותה.

יחידת הבקרה והתזמון (Timing & Control unit) היא מרכיב ראשי בתכנון המערכת: היא מספקת את אותות הבקרה והתזמון לרכיבי המערכת השונים. בין היתר היא גם מכניסה פסיקות למעבד עפ"י דרישות התוכנה. יחידה זו ממומשת בדרך כלל ע"י רכיב חומרה מתוכנת.

הפעלת החומרה ע"י התוכנה :

מערכת משובצת מחשב מופעלת במשולב ע"י התוכנה והחומרה. התוכנה היא השולטת ומפעילה את כלל המערכת. פעולות עיקריות שמבצעת התוכנה במסגרת זו:

- אתחול החומרה וביצוע בדיקות עצמיות (**BIT**=Built In Test)
- הפעלת ממשק המשתמש
- תזמון מודולי התוכנה והחומרה לעבודה בזמנים מתאימים
- קבלת מידע מהמשתמש והעברתו לחומרה לשידור
- קבלת המידע הנקלט מהחומרה והעברתו למשתמש
- בדיקת סטטוס החומרה לזיהוי תקלות ולטיפול בהן

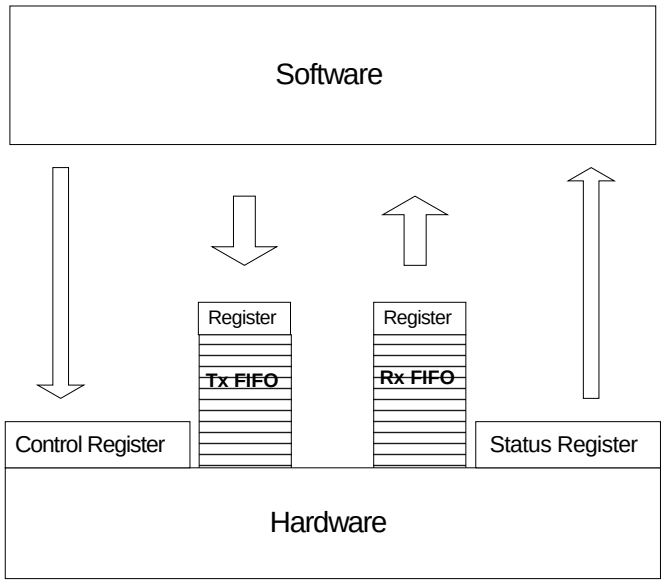
כיצד התוכנה והחומרה "מדברות" ביניהן? קיימים מספר רכיבים המשמשים בהעברת מידע ובקרה בין התוכנה והחומרה:

רגיסטרים - יחידות זיכרון המשמשות להעברת הוראות מהתוכנה לחומרה ולקבלת סטטוסים חוזרים מהחומרה. גדלים טיפוסיים: 8, 16 ו- 32 סיביות.

רכיבי **FIFO** - יחידות שניתן לכתוב אליהן ולקרוא מהן בו זמנית, כאשר המידע המוכנס ראשון בכתיבה יוצא ראשון בקריאה. הן שימושיות בדרך כלל בהעברת המידע לשידור מהתוכנה לחומרה (FIFO שידור) והעברת המידע הנקלט מהחומרה לתוכנה (FIFO קליטה).

שעונים - יחידות המתזמנות את המודולים השונים במערכת. התוכנה מאתחלת את שעוני המערכת לקבלת פסיקות בזמנים או במרווחי זמן מסוימים, ובקבלת הפסיקות התוכנה מתזמנת את המודולים המתאימים לביצוע.

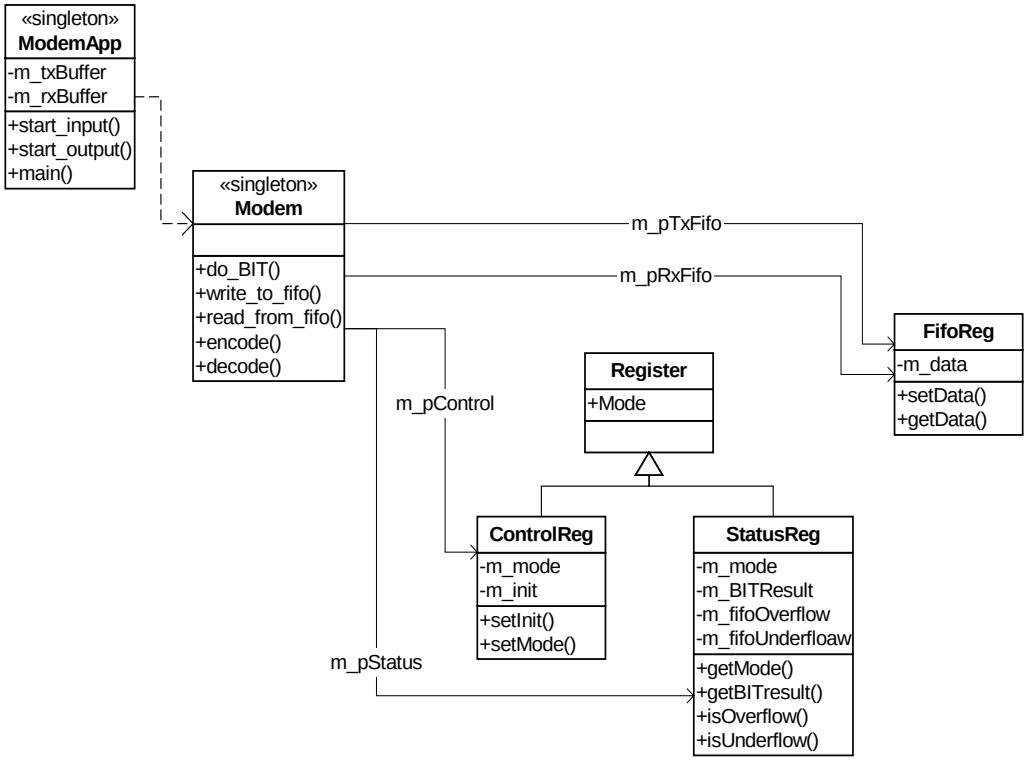
התרשים להלן ממחיש את אופן הפעלת החומרה ע"י התוכנה:



הערה: Tx ו- Rx הם קיצורים מתאימים ל- transmit ו- receive (שידור וקליטה).

תוכנת המודם

נראה כעת את התוכנה ב- ++C המפעילה את המערכת. תרשים המחלקות:



הסבר: תרשים זה כולל מספר מצומצם וחיוני של מחלקות בתוכנת המודם.

- **Modem** היא המחלקה הראשית המפעילה את המודם. היא מוגדרת כ- ***Singleton***, ולמעשה היא גם משמשת כ- ***Facade*** עבור כלל מערכת המודם (ראה/י פרק 16).
- **Register** היא מחלקת הבסיס לרגיסטרי הבקרה (**ControlReg**) והסטטוס (**StatusReg**), המספקת הגדרת Mode. הירושה כאן אינה כוללת פונקציה וירטואלית כלשהי, וזאת בכדי שעצם מהרגיסטר לא יתפוס יותר מבית בודד בזיכרון (תוספת פונקציה וירטואלית אחת תגרום להוספת המצביע vptr לעצם).
- **FifoReg** הוא רגיסטר הממשק ל- FIFO: המחלקה הראשית, Modem, מחזיקה שני מופעים שלו - אחד עבור הכתיבה בשידור, והאחר עבור הקריאה בקליטה.
- **ModemApp** היא מחלקת היישום המשתמשת בתוכנת המודם. גם היא מוגדרת כ- Singleton. היא כוללת את הפונקציה ()main הכוללת את הלולאה הראשית (אינסופית) של היישום, וכן תהליכים לקריאת מידע לשידור מהקלט ולמשלוח המידע הנקלט לפלט.

קוד המחלקה Register:

```
class Register
{
public:
        enum Mode {                 // mode of work
                NORMAL=0,           // normal work: transmit and receive
                DEBUG=1,            // debug mode
                LOOP_BACK=2,        // loop-back mode
                BIT=3               // Built-In Test mode
        };
};
```

קוד המחלקות ControlReg ו- StatusReg:

```
class ControlReg : public Register
{
public:
        void   setInit()              { m_init = true; }
        void   setMode(Mode mode) { m_mode = mode; }
private:
        unsigned char     m_mode:2;
        unsigned char     m_init:1;       // hardware initialize, atuo-reset
};

class StatusReg : public Register
{
public:
        enum BITResult { // Built-In Test (BIT) result
                BIT_OK=0,           // BIT result OK
                RADIO_ERROR=1,          // radio unit test error
                MEM_ERROR=2,  // memory unit test error
                FIFO_ERROR=3   // FIFO unit test error
        };
        int    getMode() const { return m_mode; }
        int    getBITResult() const { return m_BITResult; }
        bool   isOverflow() const { return m_fifoOverflow; }
        bool   isUnderflow() const { return m_fifoUnderflow; }
private:
        volatile unsigned char   m_mode:2;
        volatile unsigned char   m_BITResult:3;
        volatile bool            m_fifoOverflow:1;
        volatile bool            m_fifoUnderflow:1;
};
```

קוד המחלקה FifoReg:

```
class FifoReg
{
        volatile unsigned char   m_data:8;
public:
        void   setData(int data) { m_data = data; }
        int    getData() const { return m_data; }
};
```

וקוד המחלקה הראשית, **Modem**:

```
class Modem
{
```

- הגדרת מצביעים לרגיסטרים:

```
        StatusReg   *m_pStatus;
        ControlReg  *m_pControl;
        FifoReg     *m_pTxFifo;
        FifoReg     *m_pRxFifo;
```

איתחול ב-constructor: כתובות הרגיסטרים נקבעות באמצעות הטכניקה שלעיל, "placement new":

```
        Modem()
        {
                // set addresses of registers
                m_pStatus  = reinterpret_cast<StatusReg*> (0xff210980);
                m_pControl = reinterpret_cast<ControlReg*> (0xff210984);
                m_pTxFifo  = reinterpret_cast<FifoReg*> (0xff210988);
                m_pRxFifo  = reinterpret_cast<FifoReg*> (0xff21098c);

                // allocate with placement new
                m_pStatus = new (m_pStatus) StatusReg;
                m_pControl = new (m_pControl) ControlReg;
                m_pTxFifo = new (m_pTxFifo) FifoReg;
                m_pRxFifo = new (m_pRxFifo) FifoReg;

                m_pControl->setInit();              // init hardware, auto-reset
        }

public:
```

פונקצית הגישה ל- Singleton:

```
        static Modem& instance() // singleton access method
                { static Modem theModem; return theModem; }
```

פונקציה לביצוע הבדיקה הפנימית של החומרה (BIT):

```
        int    do_BIT() // Built In Test
        {
                m_pControl->setMode(ControlReg::BIT);   // start Built In Test

                // wait for BIT end (Polling)
                while(m_pStatus->getMode() == StatusReg::BIT)
                        ;

                // return BIT result
                return m_pStatus->getBITResult();
        }
```

לאחר קביעת המצב ברגיסטר הבקרה ל- BIT, הפונקציה בודקת בלולאה (Polling) האם החומרה סיימה את תהליך ה- BIT. בסיום, מוחזרת תוצאת ה- BIT.

פונקציה לביצוע קידוד מקור למידע לפני השידור:

```
void encode(const char *msg) const {/*...*/}
```

פונקציה לביצוע דה-קידוד (פענוח) למידע המקודד שנקלט:

```
void decode(const char *msg) const {/*...*/}
```

פונקציה לכתיבת חוצץ בעל אורך נתון ל- FIFO השידור:

```
void write_to_FIFO(const char *msg, int length)
{
	encode(msg);
	for(int i=0; i<length; i++)  // write to Tx FIFO register
		m_pTxFifo->setData(msg[i]);

	if(m_pStatus->isOverflow())  // check for overflow
		cerr << "Error: Overflow in Tx FIFO" << endl;
}
```

פונקציה לקריאת חוצץ בעל אורך נתון מ- FIFO הקליטה:

```
void read_from_FIFO(const char *msg, int length) const
{
	for(int i=0; i<length; i++)  // read from Rx FIFO register
		msg[i] = (char) m_pRxFifo->getData();

	if(m_pStatus->isUnderflow()) // check for underflow
		cerr << "Error: Underflow in Rx FIFO" << endl;
	else
		decode(msg);
}
};
```

מחלקת היישום, ModemApp:

```
class ModemApp
{
	enum { MSG_SIZE=64, MSGS_NUM=4};
	unsigned char	m_txBuffer[MSGS_NUM][MSG_SIZE];
	unsigned char	m_rxBuffer[MSGS_NUM][MSG_SIZE];
	ModemApp() {}
public:
	static ModemApp& instance()
	{ static ModemApp theModemApp;  return theModemApp; }
	void start_input() { /*...*/ }
	void start_output() { /*...*/ }

	// main() - demo uses of the Modem application
	int main()
	{
		// do Built In Test
		if(Modem::instance().do_BIT() != StatusReg::BIT_OK)
		{
			cerr << "Error: Hardware BIT error" << endl;
			return -1;
		}

		start_input(); // this process writes the tx buffer
		start_output(); // this process reads the rx buffer
```

```
            // main loop
            for(int i=0 ;  ; i++) // forever
            {
                int index = i % MSGS_NUM;

                // write tx buffer to tx FIFO
                Modem::instance().write_to_FIFO(m_txBuffer[index], MSG_SIZE);

                // read rx buffer from rx FIFO
                Modem::instance().read_from_FIFO(m_rxBuffer[index], MSG_SIZE);
            }
        }
};
```

והפונקציה הראשית, main(), פשוט קוראת ל- ModemApp::main():

```
int main()
{
    return ModemApp::instance().main();
}
```

סיכום

- **יעילות** התוכנה נמדדת עפ"י שני פרמטרים עיקריים: מקום וזמן.
- שפת C++ מאפשרת לכתוב קוד יעיל ביותר - הן בהשוואה לשפות מבניות יעילות כגון C, והן בהשוואה לשפות interpreter מונחות עצמים כגון Java - יכולת המשפיעה על בחירתה כשפת פיתוח למערכות זמן אמת.
- מערכות זמן אמת הן מערכות הפועלות במסגרות זמן ומקום ידועות מראש. הן נקראות גם מערכות משובצות מחשב, מכיוון שבדרך כלל הן מכילות רכיבי חומרה כגון: מעבד, זיכרונות, רגיסטרים וכו'.
- בשפת C++ קיימת תמיכה מלאה בתכנות ברמת מכונה (machine level) ע"י הוראות כגון: **volatile**, **register** ו- **asm**, ע"י תמיכה בטיפול בשדות סיביות, וע"י היכולת להקצות עצמים בכתובות נתונות (**placement new**).

מאיר סלע

כל הזכויות שמורות © מרכז ההדרכה 2000