מרכז ההדרכה 2000
תמיכה ועדכונים
עדכון מס' 32
מאי 2001

תכנות בשפת C במערכות זמן אמת

מבוא

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

מרבית מערכות זמן אמת הקיימות היום ממומשות בשפת C, כלומר C היא שפת התוכנה שבה נכתבת תוכנת המערכת. בשפת C קיימת תמיכה מלאה בתכנות ברמת מכונה (machine level) ע"י הוראות כגון: volatile, register ו- asm וכן ע"י תמיכה בטיפול בשדות סיביות במבנים.

במאמר זה נכיר את מנגנוני שפת C התומכים בתכנות מערכות זמן אמת. תוכן המאמר מתבסס על הפרק "תכנות במערכות זמן אמת" שבספר "***הנדסת תוכנה בשפת C***" בהוצאת **מרכז ההדרכה 2000**. במאמר הבא נעסוק בתכנות מונחה עצמים במערכות זמן אמת בשפת C++.

מנגנונים ב- C לתמיכה בתכנות 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 מעוניינים למנוע את האופטימיזציה. לדוגמא, כאשר משתנה מסוים עלול להשתנות ע"י מערכת ההפעלה, 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 במבנים

ניתן להגדיר מבנים ששדותיהם אינם מילים שלמות, אלא חלקי מילים בגודל של מספר משתנה של סיביות. לדוגמא, הגדרת רשומה המייצגת רגיסטר סטטוס של החומרה בעל אורך של 7 סיביות:

```
typedef struct
{
        volatile unsigned int mode:2;
        volatile unsigned int BIT_result:3;
        volatile unsigned int fifo_overflow:1;
        volatile unsigned int fifo_underflow:1;
} status_reg;
```

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

לדוגמא, נגדיר משתנה מסוג הרגיסטר הנ"ל:

```
status_reg        sr;
```

וכעת ניתן להתייחס אל שדותיו כאילו היו שדות במבנה:

```
unsigned   int  is_overflow = sr.fifo_overflow;

if(is_overflow)
        /* overflow error... */
```

המהדר דואג להציב את הסיבית המתאימה המייצגת את fifo_overflow למשתנה השלם is_overflow תוך ביצוע פעולות הזזה, מיסוך והצבה מתאימות על הסיביות.

דוגמא נוספת: נגדיר טיפוס רגיסטר בקרה

```
typedef struct
{
        volatile unsigned int mode:2;
        volatile unsigned int init:1;
} control_reg ;
```

כעת נגדיר מצביעים לרגיסטרים הנ"ל המצביעים לכתובת מוחלטת בזיכרון שאליה ממופה הרגיסטר במערכת משובצת מחשב:

```
status_reg  *p_status    = (status_reg*)        0xff210980;
control_reg *p_control   = (control_reg*)       0xff210984;
```

לדוגמא, הצבת ערך לשדה ברגיסטר:

```
p_control->init = 1;
```

הערה : בפועל המהדר מקצה לכל רשומה כנ"ל מספר שלם של מילים (integers). כאשר סה"כ השדות קטן ממספר מילים שלם המהדר משלים את החסר ע"י ריפוד באפסים.

enum של ערכי סיביות

השדה mode המוגדר בשתי הרשומות הנ"ל הוא בעל 2 סיביות ולכן יכול לקבל 4 ערכים בינריים שונים :

00, 01, 10, 11

הגדרה נוחה של הערכים אפשרית ע"י enum בדרך הבאה :

```
typedef enum
{
        MODE_NORMAL  = 0,    /* normal mode */
        MODE_DEBUG= 1,       /* debugging mode */
        MODE_LOOP_BACK= 2,  /* loop back mode (no radio transmission) */
        MODE_BIT  = 3,       /* Built In Test mode */
} MODE;
```

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

```
p_control->mode = MODE_BIT;
```

בהוראה זו כתבנו לרגיסטר הבקרה את מצב הבדיקה MODE_BIT, מה שגורם לחומרה לעבור למצב זה. כעת ממתינים לסיום הבדיקה ע"י דגימת הסיבית המתאימה ברגיסטר הסטטוס :

```
while(p_status->mode == MODE_BIT)
        ;
```

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.

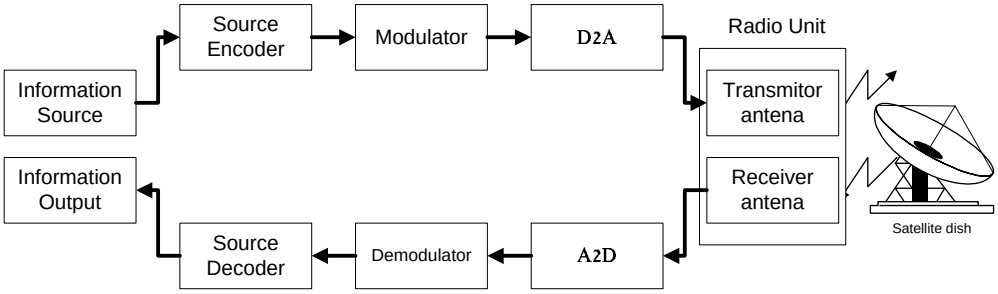
לדוגמא, בסביבת הפיתוח Visual C++ ניתן לבצע הוראות אסמבלר באמצעות ההוראה asm_ על בלוק שלם או כהוראות נפרדות :

```
__asm                   /* __asm block */
{
   mov eax, 01h
   int 10h
}
__asm mov eax, 01h       /* Separate __asm lines */
__asm int 10h
```

דוגמא: מערכת תקשורת אלחוטית

מבנה לוגי של המערכת

מערכת תקשורת אלחוטית משובצת מחשב כוללת שתי תת-מערכות עיקריות - משדר ומקלט. כל אחת מתת המערכות מכילה סדרת פעולות המבוצעות על המידע החל מקבלתו ועד לשידורו - במסלול השידור - ובמסלול ההפוך מקליטתו ועד למסירתו ליעד הפלט:



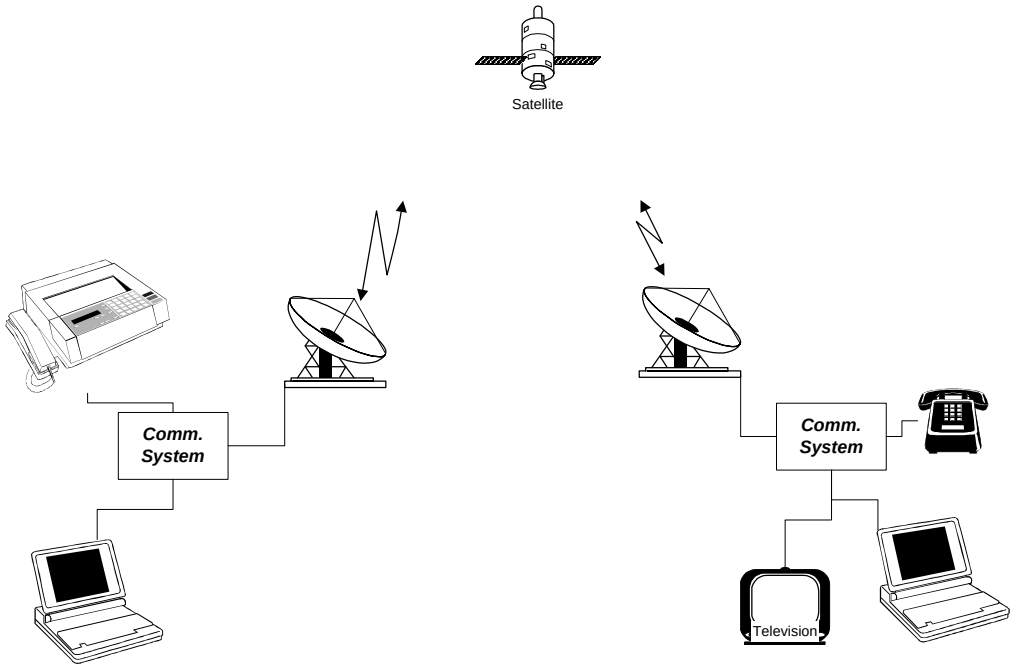
משדר - תת-מערכת הכוללת את המרכיבים הבסיסיים הבאים:

- **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 הספרתי לייצוג אנלוגי.
- **אנטנת שידור** (Transmitter antenna) - לשידור האות לאוויר

מקלט - תת-מערכת הכוללת את המרכיבים ההפוכים למשדר:

- **אנטנת קליטה** (Receiver antenna) - לקליטת האות שבאוויר
- **ממיר אנלוגי-לספרתי** (**A2D**) - המרת האות הנקלט מייצוג אנלוגי לספרתי
- **דה-אפנון** (Demodulation) - קידוד הפוך לאפנון שבוצע בשידור
- **קידוד מקור הפוך** (source decoder) - קידוד הפוך לקידוד המקור שבוצע בשידור
- **יעד הפלט** (Information output) - היחידה אליה נשלח המידע הנקלט

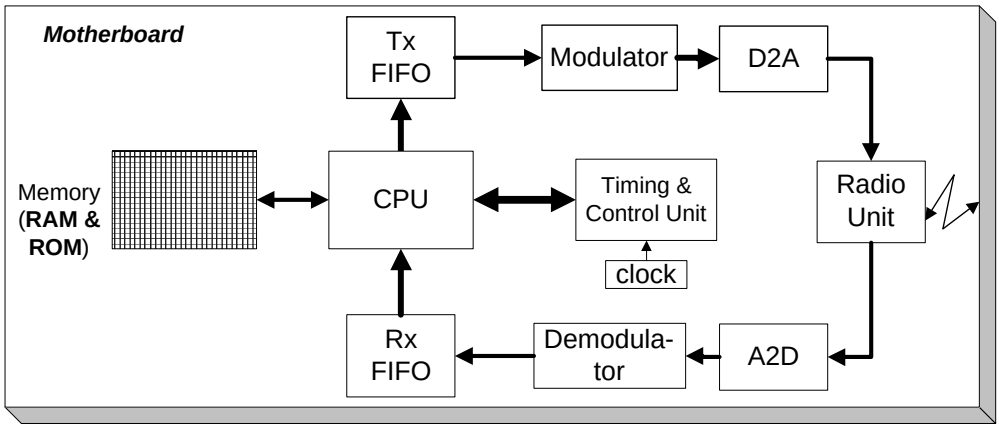
דוגמאות למערכות תקשורת אלחוטיות משובצות מחשב: טלפון סלולרי, מכשירי קשר ניידים, משדרי ומקלטי רדיו, טלפון אלחוטי. לדוגמא, קופסת קשר לוויני המיועדת להעברת מידע בין מחשבים, טלפונים, פקסימיליות וטלוויזיות ע"י תקשורת לווינים:



מערכת הקשר שבתרשים (Comm. System) מקבלת מידע לשידור ממכשיר מסוים להעברה ומשדרת אותו למערכת אחרת דרך הלוויין. בערוץ הקליטה המערכת מעבירה את המידע הנקלט למכשיר המתאים.

מבנה פיזי של המערכת

התרשים הבא מתאר מבנה פיזי של רכיבי המערכת:



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

הזיכרון (**memory**) מכיל שני חלקים עיקריים: ה- **ROM** (Read Only Memory) הוא מקום האחסון של תוכנת המערכת, כלומר של קוד הביצוע (Code Segment). בהפעלת המכשיר, התוכנה מתחילה להתבצע

מכתובת מסוימת ב- ROM. ה- **RAM** (Raw Access Memory) הוא זיכרון המיועד לאיזורי הזיכרון הדינמיים של התכנית - כלומר מקטע הנתונים (Data Segment), מחסנית הקריאות והערימה (Heap).

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.

יחידות ה**איפנון**, ההמרה **אנלוגי-דיגיטלי** ו**הרדיו** ממוקמות ביחד עם היחידות העיקריות הנ"ל על לוח האם (Motherboard), ודרכו מתבצעת העברת המידע והבקרה בשני הכוונים - בשידור ובקליטה.

הערה: הממשק בין המעבד ורכיבי החומרה ובין רכיבי החומרה לבין עצמם ממומש בדרך כלל ע"י רגיסטרים. לדוגמא, הממשק ליחידת הבקרה יהיו רגיסטרי בקרה להעברת פקודות מהתוכנה לחומרה וכן רכיבי סטטוס לקבלת סטטוסי ביצוע.

שאלת הבנה: עפ"י תרשים המערכת הפיזי, כיצד ממומש **מקודד המקור (modulator)** - בתוכנה או בחומרה?

חלוקת משימות בין התוכנה והחומרה במערכת

משימות המערכת הנ"ל מבוצעות במתואם ע"י התוכנה והחומרה במערכת. ההחלטה מי מבצע מה תלויה במספר פרמטרים:

- מהירות הביצוע (התלויה בקצב המידע): אם המהירות גבוהה מדיי לביצוע ע"י התוכנה נדרש רכיב חומרה ייעודי לביצוע המשימה. יש לשים לב שמהירות הביצוע ע"י התוכנה תלויה במעבד שבמערכת.
- עלות רכיבי החומרה: רכיב חומרה יקר עלול לייקר את המערכת - מימוש המשימה בתוכנה (אם אפשרי) יחסוך את הצורך ברכיב.
- זמן הפיתוח בתוכנה: פיתוח של מודול תוכנה המבצע משימה מורכבת עלול לצרוך זמן (וכסף) רב.
- גמישות הפעלה ויכולת ביצוע שינויים: מודול תוכנה בדרך כלל גמיש יותר הן להפעלה במודים שונים והן לביצוע שינויים במודול.

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

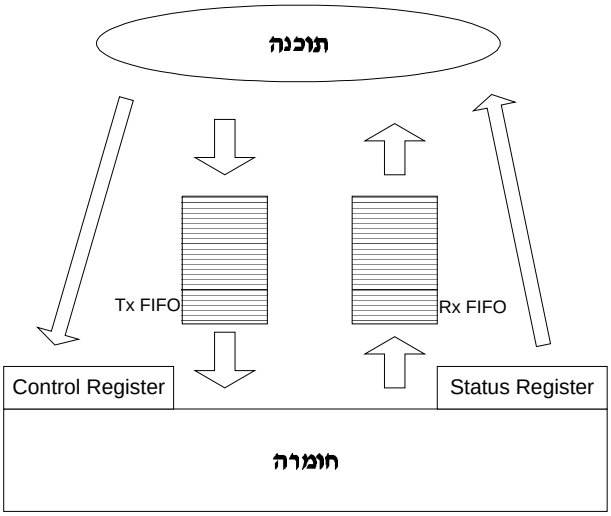
כיצד התוכנה והחומרה "מדברות" ביניהן? קיימים מספר מנגנונים המשמשים בהעברת מידע ובקרה בין התוכנה והחומרה:

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 ופסיקות - 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, והם סימנים מאוד שכיחים בעולם התקשורת.

תוכנת המערכת

נראה כעת את התוכנה המפעילה את המערכת. ראשית נגדיר את הטיפוסים עבור רגיסטרי הסטטוס והבקרה וכן עבור ערכי השדות. יש לשים לב שהשדות ברגיסטרים מוגדרים כ- **volatile**.

- הגדרת קבועים עבור גדלי ההודעות הנשלחות בערוץ:

```
#define MSG_SIZE      64
#define MSGS_NUM      4
```

– רגיסטר הבקרה :

```
typedef struct
{
        volatile unsigned int mode:2;
        volatile unsigned int init:1;
} control_reg ;
```

– קבועי המצבים :

```
typedef enum
{
        MODE_NORMAL  = 0,    /* normal mode */
        MODE_DEBUG= 1,       /* debugging mode */
        MODE_LOOP_BACK= 2,  /* loop back mode (no radio transmission) */
        MODE_BIT  = 3,       /* Built In Test mode */
} MODE;
```

– רגיסטר הסטטוס :

```
typedef struct
{
        volatile unsigned int mode:2;
        volatile unsigned int BIT_result:3;
        volatile unsigned int fifo_overflow:1;
        volatile unsigned int fifo_underflow:1;
} status_reg;
```

– קבועי תוצאות הבדיקות (BIT) :

```
typedef enum    /* Built In Test Results */
{
        BIT_RESULT_OK           = 0,
        BIT_RESULT_RADIO_ERROR    = 1,    /* radio unit test error */
        BIT_RESULT_MEM_ERROR      = 2,    /* memory unit test error */
        BIT_RESULT_FIFO_ERROR     = 4     /* FIFO unit test error */
} BIT_RESULT ;
```

– טיפוס רגיסטר של FIFO באורך 8 סיביות :

```
typedef struct
{
        volatile unsigned int data:8;
} fifo_reg;
```

– הגדרת כתובות המצביעים לרגיסטרים השונים :

```
status_reg  *p_status  = (status_reg*)      0xff210980;
control_reg *p_control = (control_reg*)     0xff210984;
fifo_reg    *p_tx_fifo = (fifo_reg*)        0xff210988;
fifo_reg    *p_rx_fifo = (fifo_reg*)        0xff21098c;
```

הגדרות אלו תלויות במיפוי כתובות מסוים שבוצע לרגיסטרי המערכת. במקרה זה, ממופים הרגיסטרים למרחב הזיכרון 0xFF210980..0xFF21098C.

– הגדרת חוצצי השידור והקליטה:

```
char  tx_buffer[MSGS_NUM][MSG_SIZE];
char  rx_buffer[MSGS_NUM][MSG_SIZE];
```

חוצצים אלה הם בעלי מספר קבוע (MSGS_NUM) של הודעות בגודל קבוע (MSG_SIZE). ההנחה היא שקיים תהליך הכותב לחוצץ השידור מידע מהמשתמש ומחזיר לו את המידע הנקלט מהצד השני. עבודה עם מספר הודעות הכרחית כדי לאפשר "זרימה" רצופה של מידע בשני הכוונים.

– אתחול החומרה מבוצע ע"י הצבה לשדה init שברגיסטר הבקרה:

```
p_control->init = 1;
```

– בדיקת החומרה (BIT) מבוצעת ע"י פונקצית בדיקה:

```
BIT_RESULT do_BIT()
{
        /* Built In Test */
        p_control->mode = MODE_BIT;   /* start Built In test */

        /* wait for BIT end */
        while(p_status->mode == MODE_BIT)
                ;

        return p_status->BIT_result;
}
```

במהלך הפונקציה מחכים לסיום ה- BIT של החומרה ע"י בחינת שדה ה- mode שברגיסטר הסטטוס. צורת המתנה זו נקראת **polling**.

– את תוצאות הבדיקה קוראים מהפונקציה הראשית:

```
if(do_BIT() != BIT_RESULT_OK)
        error("Hardware BIT error");
```

– לאחר מכן משנים את מצב העבודה של החומרה למצב עבודה רגיל:

```
p_control->mode = MODE_NORMAL;
```

– כעת אנחנו מוכנים לביצוע הלולאה הראשית של התכנית - כתיבת מידע ל- FIFO השידור וקריאת מידע נקלט מ- FIFO הקליטה:

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

        /* write message to tx FIFO */
        write_to_FIFO(tx_buffer[index], MSG_SIZE);

        /* read message from rx FIFO */
        read_from_FIFO(rx_buffer[index], MSG_SIZE);
}
```

בכל חזרה כותבים את ההודעה הבאה מחוצץ שידור ציקלי (שתוכנו מגיע מיחידת הממשק למשתמש) ל- FIFO השידור, וקוראים מ- FIFO הקליטה הודעה לחוצץ קליטה ציקלי. גדלי ההודעות הן 64 בתים.

– הפונקציה write_to_FIFO מוגדרת כך :

```
void write_to_FIFO(char *msg, int length)
{
        int i;

        /* write to FIFO data register */
        for(i=0; i<length; i++)
                p_tx_fifo->data = msg[i];

        /* check for overflow */
        if(p_status->fifo_overflow)
        {
                puts("Error: Overflow in FIFO");
                exit(1);
        }
}
```

– הפונקציה read_from_FIFO מוגדרת כך :

```
void read_from_FIFO(char *msg, int length)
{
        int i;

        /* read from FIFO data register */
        for(i=0; i<length; i++)
                msg[i] = (char) p_rx_fifo->data;

        /* check for underflow */
        if(p_status->fifo_underflow)
        {
                puts("Error: Underflow in FIFO");
                exit(1);
        }
}
```

להלן קוד המודול העיקרי ביישום בכללותו :

```
/* file: comm_main.c */
#include <stdio.h>
#include <stdlib.h>

/************* CONSTANTS ***************************/
#define MSG_SIZE        64
#define MSGS_NUM        4

/************* TYPES ******************************/
/* control register type */
typedef struct
{
```

```
        volatile unsigned int mode:2;
        volatile unsigned int init:1;
} control_reg ;

typedef enum
{
        MODE_NORMAL  = 0,   /* normal mode */
        MODE_DEBUG   = 1,   /* debugging mode */
        MODE_LOOP_BACK= 2,/* loop back mode (no radio transmission) */
        MODE_BIT     = 3,   /* Built In Test mode */
} MODE;

/* status register type */
typedef struct
{
        volatile unsigned int mode:2;
        volatile unsigned int BIT_result:3;
        volatile unsigned int fifo_overflow:1;
        volatile unsigned int fifo_underflow:1;
} status_reg;

typedef enum    /* Built In Test Results */
{
        BIT_RESULT_OK          = 0,
        BIT_RESULT_RADIO_ERROR    = 1,     /* radio unit test error */
        BIT_RESULT_MEM_ERROR      = 2,     /* memory unit test error */
        BIT_RESULT_FIFO_ERROR     = 4      /* FIFO unit test error */
} BIT_RESULT ;

/* fifo data register */
typedef struct
{
        volatile unsigned int data:8;
} fifo_reg;


/************** GLOBAL VARIABLES **********************/
status_reg  *p_status    = (status_reg*)        0xff210980;
control_reg *p_control   = (control_reg*)       0xff210984;
fifo_reg    *p_tx_fifo   = (fifo_reg*)          0xff210988;
fifo_reg    *p_rx_fifo   = (fifo_reg*)          0xff21098c;

char tx_buffer[MSGS_NUM][MSG_SIZE];
char rx_buffer[MSGS_NUM][MSG_SIZE];


/************* FUNCTION BODIES ************************/
void write_to_FIFO(char *msg, int length)
{
        int i;
```

```
        /* write to FIFO data register */
        for(i=0; i<length; i++)
                p_tx_fifo->data = msg[i];

        /* check for overflow */
        if(p_status->fifo_overflow)
        {
                puts("Error: Overflow in FIFO");
                exit(1);
        }
}

void read_from_FIFO(char *msg, int length)
{
        int i;

        /* read from FIFO data register */
        for(i=0; i<length; i++)
                msg[i] = (char) p_rx_fifo->data;

        /* check for underflow */
        if(p_status->fifo_underflow)
        {
                puts("Error: Underflow in FIFO");
                exit(1);
        }
}

BIT_RESULT do_BIT()
{
        /* Built In Test */
        p_control->mode = MODE_BIT;   /* start Built In test */

        /* wait for BIT end */
        while(p_status->mode == MODE_BIT)
                ;

        return p_status->BIT_result;
}

void main()
{

        int i, index;

        /* init hardware */
        p_control->init = 1;

        /* do Built In Test */
        if(do_BIT() != BIT_RESULT_OK)
        {
```

```
            puts("Error: Hardware BIT error");
            exit(1);
        }

        /* set normal working mode */
        p_control->mode = MODE_NORMAL;

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

            /* write message to tx FIFO */
            write_to_FIFO(tx_buffer[index], MSG_SIZE);

            /* read message from rx FIFO */
            read_from_FIFO(rx_buffer[index], MSG_SIZE);
        }
}
```

סיכום

- מערכות זמן אמת הן מערכות הפועלות במסגרות זמן קבועות או חסומות הידועות מראש. הן נקראות גם מערכות משובצות מחשב, מכיוון שהן 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** וכן ע"י תמיכה בטיפול בשדות סיביות.
- מערכת משובצת מחשב כוללת בד"כ את המרכיבים העיקריים הבאים:
  - **זיכרון** (**memory**) מכיל את ה- **ROM** (Read Only Memory) - מקום האחסון של קוד הביצוע (Code Segment), ה- **RAM** (Raw Access Memory) - מיועד לאחסון קטע הנתונים (Data Segment), מחסנית הקריאות והערימה (Heap).
  - **המעבד** (**CPU**) היא יחידת החישוב של המערכת, המבצע את קוד התכנית בקטע הקוד. בכל שלב המעבד קורא את ההוראה הבאה מקטע הקוד (Code Segment) ומבצע אותה.
  - **יחידת הבקרה והתזמון** (**Timing & Control unit**) היא מרכיב ראשי המספק 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.
- מערכת תקשורת אלחוטית משובצת מחשב כוללת שתי תת-מערכות עיקריות - **משדר** ו**מקלט**. כל אחת מתת המערכות מכילה סדרת פעולות המבוצעות על המידע החל מקבלתו ועד לשידורו - במסלול השידור - ובמסלול ההפוך מקליטתו ועד למסירתו ליעד הפלט.

כל הזכויות שמורות © מאיר סלע

מרכז ההדרכה 2000