EMBEDDED

л.2
Засоби та інструменти для проектування. Чому інструменти це дуже важливо.

Палій Святослав

Український Католицький Університет

APPLIED SCIENCES FACULTY

Програмні засоби

Інтегроване середовище розробки (IDE)

- Інтегроване середовище розробки (IDE) - це програмне забезпечення, яке надає розробникам інструменти для написання, тестування та налагодження коду в одному місці. Основні компоненти IDE включають:

1. **Редактор коду**: Зручний текстовий редактор з підсвічуванням синтаксису, автодоповненням, форматуванням та іншими функціями для полегшення написання коду.
2. **Компілятор/Інтерпретатор**: Інструмент для перетворення вихідного коду в машинний код або виконання коду безпосередньо.
3. **Налагоджувач (дебагер)**: Інструмент для виявлення та виправлення помилок у коді, дозволяє встановлювати точки зупинки, переглядати змінні та виконувати код покроково.
4. **Інтеграція з системами контролю версій**: Підтримка роботи з системами контролю версій, наприклад Git, для управління змінами в коді.
5. **Інструменти для тестування та управління дефектами**: Засоби для автоматизованого тестування коду та перевірки його коректності.
6. **Графічний інтерфейс користувача (GUI)**: Зручний інтерфейс для взаємодії з усіма інструментами IDE.

Приклади: Arduino, Eclipse, Visual Studio, uVision, PSoC Creator ...

Компілятор та лінкер

Компілятор

Компілятори перетворюють код написаний на мові програмування в машинний код.

- Етап аналізу

На цьому етапі компілятор аналізує вихідний код, перевіряючи його на синтаксичні та семантичні помилки.

- Етап оптимізації

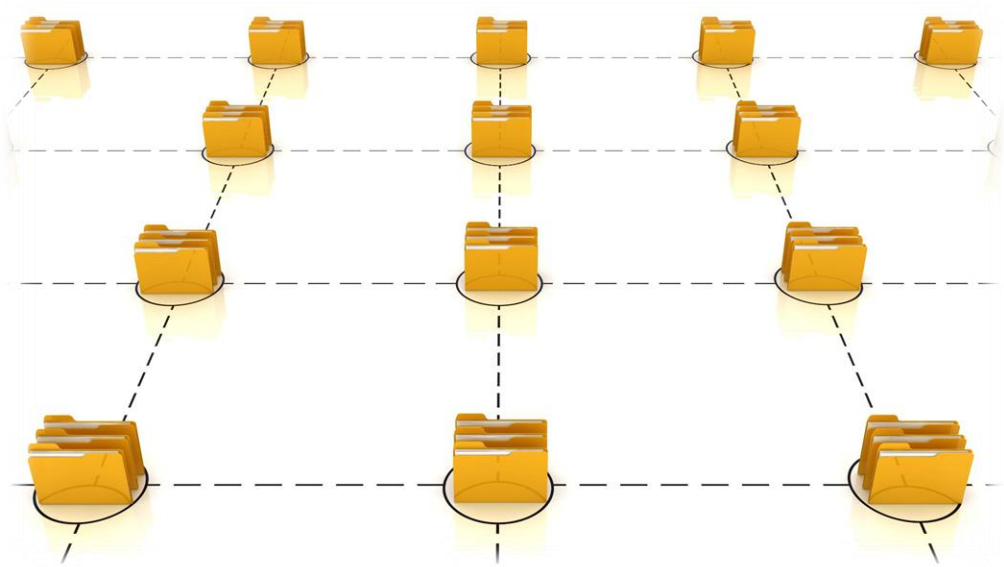
Під час оптимізації компілятор покращує код, зменшуючи його обсяг і підвищуючи ефективність виконання програми.

- Етап генерації коду

На заключному етапі компілятор генерує машинний код, який може бути виконаний комп'ютером або іншим пристроєм.

Лінкер

Лінкер об'єднує різні об'єктні файли чи іншими словами модулі програми (як правило отримані в результаті компіляції) в єдине програмне забезпечення, яке можна виконувати.



Ефективність програмістів

Розуміння компіляторів та лінкерів допомагає програмістам підвищувати свою ефективність та покращувати якість коду.

Як компілятор може впливати на вихідний код

Найперше слід відрізняти код скомпільований для кінцевого використання (**release**) та код скомпільований для налагодження (**debug**).

- Компілятори як правило викликаються з командної стрічки і ми передаємо їм опції командної стрічки для визначення режиму компіляції. Як приклад покликання на список опцій компілятора gcc: https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Invoking-GCC.html

Деякі з них особливо варті уваги:

Рівень оптимізації:

- -O2: Ця опція вмикає хороший рівень оптимізації без значного збільшення часу компіляції.
- -O3: Ця опція вмикає більш агресивні оптимізації, що може призвести до кращої продуктивності, але може збільшити час компіляції.
- -Os: Ця опція оптимізує для розміру, зменшуючи розмір виконуваного файлу.
- -Og / -O0: Оптимізація практично вимкнена на полегшення налагодження.

Інформація для налагодження:

- -g0: Ця опція вимикає генерацію інформації для налагодження, яка зазвичай не потрібна в релізній збірці.

Оптимізація під час лінкування (Link Time Optimization, LTO):

- -flto: Ця опція вмикає оптимізацію під час лінкування, що може додатково оптимізувати код.

Попередження як помилки:

- -Werror: Ця опція трактує всі попередження як помилки, забезпечуючи відсутність попереджень у коді.
- -Wall: Ця опція вмикає всі попередження котрі вміє аналізувати компілятор

Implementation Defined Behavior

```c
int func(int a, int b)
{
    return a + 2*b;
}

int main()
{
    int x = 1;
    int y = func(x++, x++); // Результат може бути різним залежно від компілятора
    int z = (2*x++) + (3*x++); // тут теж
    printf("%d\n", y);
    return 0;
}
```

Implementation Defined Behavior

```c
int main()
{
    int x = INT_MAX;
    x = x + 1; // Поведінка не визначена стандартом (переповнення знакових чисел)
    x = INT_MAX;
    x = x << 1; // Поведінка не визначена стандартом (зсув знакових чисел)
    printf("%d\n", x);
    return
}
```

https://uk.wikipedia.org/wiki/Неспецифікована_поведінка
https://uk.wikipedia.org/wiki/Невизначена_поведінка

Як лінкер впливає на вихідний код

Основні функції лінкера:

1. **Об'єднання об'єктних файлів**: Лінкер об'єднує кілька об'єктних файлів (.o або .obj), створених компілятором, в один виконуваний файл або бібліотеку. Це дозволяє розділяти програму на модулі та компілювати їх окремо.
2. **Розв'язання символів**: Лінкер розв'язує символи, тобто знаходить визначення змінних та функцій, які використовуються в різних об'єктних файлах. Якщо символ не знайдено, лінкер видає помилку.
3. **Розміщення коду та даних**: Лінкер визначає, де в пам'яті будуть розташовані різні частини програми, такі як код, дані та стек. Це включає визначення адрес для функцій та змінних. Також лінкер виділяє секцію доступної пам'яті для функцій динамічного виділення пам'яті.
4. **Оптимізація**: Лінкер може виконувати оптимізації, такі як видалення невикористовуваних функцій та змінних (dead code elimination), а також об'єднання однакових функцій та змінних (common subexpression elimination).
5. **Створення таблиць символів**: Лінкер створює таблиці символів, які використовуються для налагодження та аналізу виконуваного файлу. Ці таблиці містять інформацію про адреси функцій та змінних.

Приклад роботи лінкера

```c
#include <stdio.h>

void helperFunction();

int main() {
    printf("Hello from main!\n");
    helperFunction();
    return 0;
}
```

```c
#include <stdio.h>

void helperFunction() {
    printf("Hello from helper!\n");
}
```

>gcc -c main.c -o main.o

>gcc -c helper.c -o helper.o

>gcc main.o helper.o -o my_program.hex

Що ще не так з лінкером?

- Лінкер керується лінкер-скриптом, який не подібний на мову С.
- Компілятори різних виробників мають тотально різний синтаксис лінкер-скриптів
- І тільки одне добре: як правило початковий лінкер-скрипт як правило генерується IDE на початку проекту, але це тільки початок…

```
SECTIONS
{
    /* The bootloader location */
    .cybootloader 0x0 : { KEEP(*(.cybootloader)) } >rom

    /* Calculate where the loadables should start */
    appl1_start   = CY_APPL_ORIGIN ? CY_APPL_ORIGIN : ALIGN(CY_FLASH_ROW_SIZE);
    appl2_start   = appl1_start + ALIGN((LENGTH(rom) - appl1_start - 2 * CY_FLASH_ROW_SIZE) / 2, CY_FLASH_ROW_SIZE);
    appl_start    = (CY_APPL_NUM == 1) ? appl1_start : appl2_start;
    ecc_offset    = (appl_start / CY_FLASH_ROW_SIZE) * CY_ECC_ROW_SIZE;
    ee_offset     = (CY_APPL_LOADABLE && !CY_EE_IN_BTLDR) ? ((CY_EE_SIZE / CY_APPL_MAX) * (CY_APPL_NUM - 1)) : 0;
    ee_size       = (CY_APPL_LOADABLE && !CY_EE_IN_BTLDR) ? (CY_EE_SIZE / CY_APPL_MAX) : CY_EE_SIZE;
    PROVIDE(CY_ECC_OFFSET = ecc_offset);

    .text appl_start :
    {
        CREATE_OBJECT_SYMBOLS
        PROVIDE(__cy_interrupt_vector = RomVectors);

        KEEP(*(.romvectors))

        /* Make sure we pulled in an interrupt vector.  */
        ASSERT (. != __cy_interrupt_vector, "No interrupt vector");

        ASSERT (CY_APPL_ORIGIN ? (SIZEOF(.cybootloader) <= CY_APPL_ORIGIN) : 1, "Wrong image location");

        PROVIDE(__cy_reset = Reset);
        *(.text.Reset)
        /* Make sure we pulled in some reset code.  */
        ASSERT (. != __cy_reset, "No reset code");
```

Інтерпретатор

Інтерпретатори порівняно рідкісні в embedded

Хоча...

MicroPython - був створений у 2013 році австралійським програмістом і фізиком-теоретиком Джорджем Демієном після успішної кампанії, яку підтримали на Kickstarter. Оригінальний реліз MicroPython було випущено для «pyboard» плат на основі чіпу STM32F4. Пізніше реалізували підтримку процесорів на основі архітектури ARM Cortex-M, ESP8266, ESP32, 16-бітних PIC.

Lua - швидка і компактна скриптова мова програмування, розроблена підрозділом Tecgraf Католицького університету Ріо-де-Жанейро на початку 90-х. Інтерпретатор мови є вільно поширюваним, з відкритим початковим кодом на мові C. В embedded її популяризувала підтримка ESP8266, ESP32.

Інструменти статичного аналізу коду

Зазвичай не входять до IDE, хоча останнім часом компілятори все більше і краще починають виконувати цю роль.

Static checker для мови програмування C - це інструмент, який аналізує вихідний код без його виконання, з метою виявити потенційні помилки, недоліки та проблеми з безпекою. Цей інструмент допомагає розробникам знаходити помилки на ранніх етапах розробки, що дозволяє зменшити кількість помилок у кінцевому продукті.

- **Виявлення потенційних помилок:** Виявлення можливих помилок, таких як використання неініціалізованих змінних, переповнення буфера, витоки пам’яті, використання undefined/implementation defined behavior та інші.
- **Аналіз потоку даних:** Відстеження потоку даних у програмі для виявлення потенційних проблем з безпекою та логікою.
- **Перевірка відповідності галузевим стандартам.**
- **Перевірка стилю коду (bonus):** Перевірка відповідності коду встановленим стандартам стилю та рекомендаціям.

ПРИКЛАДИ: Clang static analyser, CPPCheck, Coverity, PCLint

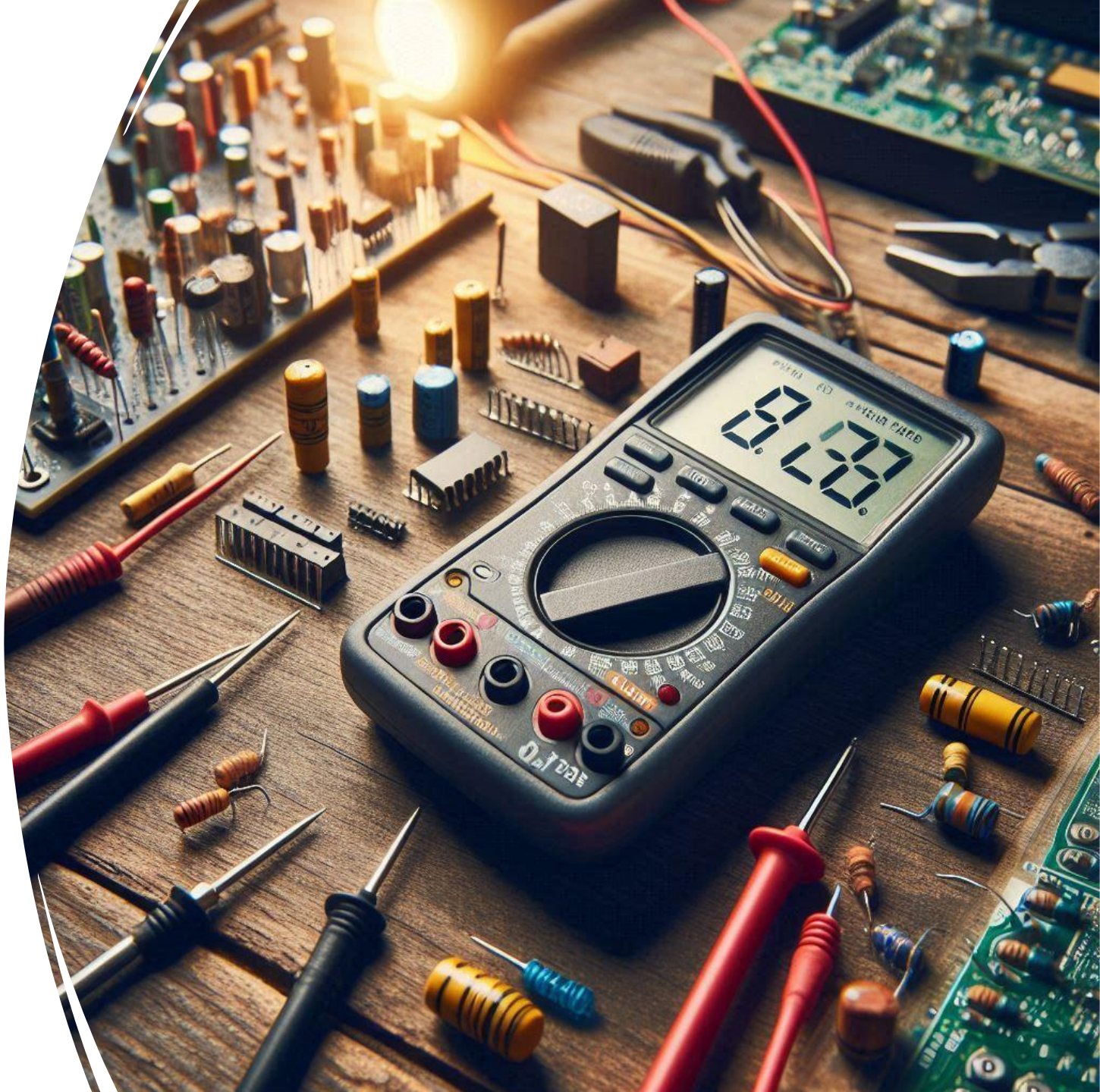
UML? Simulink?

Напевно не в рамках цього курсу!

Апаратні інструменти

Мультиметр

Вимірювання статичних або повільно змінних значень напруги, струму, опору, ємності, індуктивності, частоти, тощо.

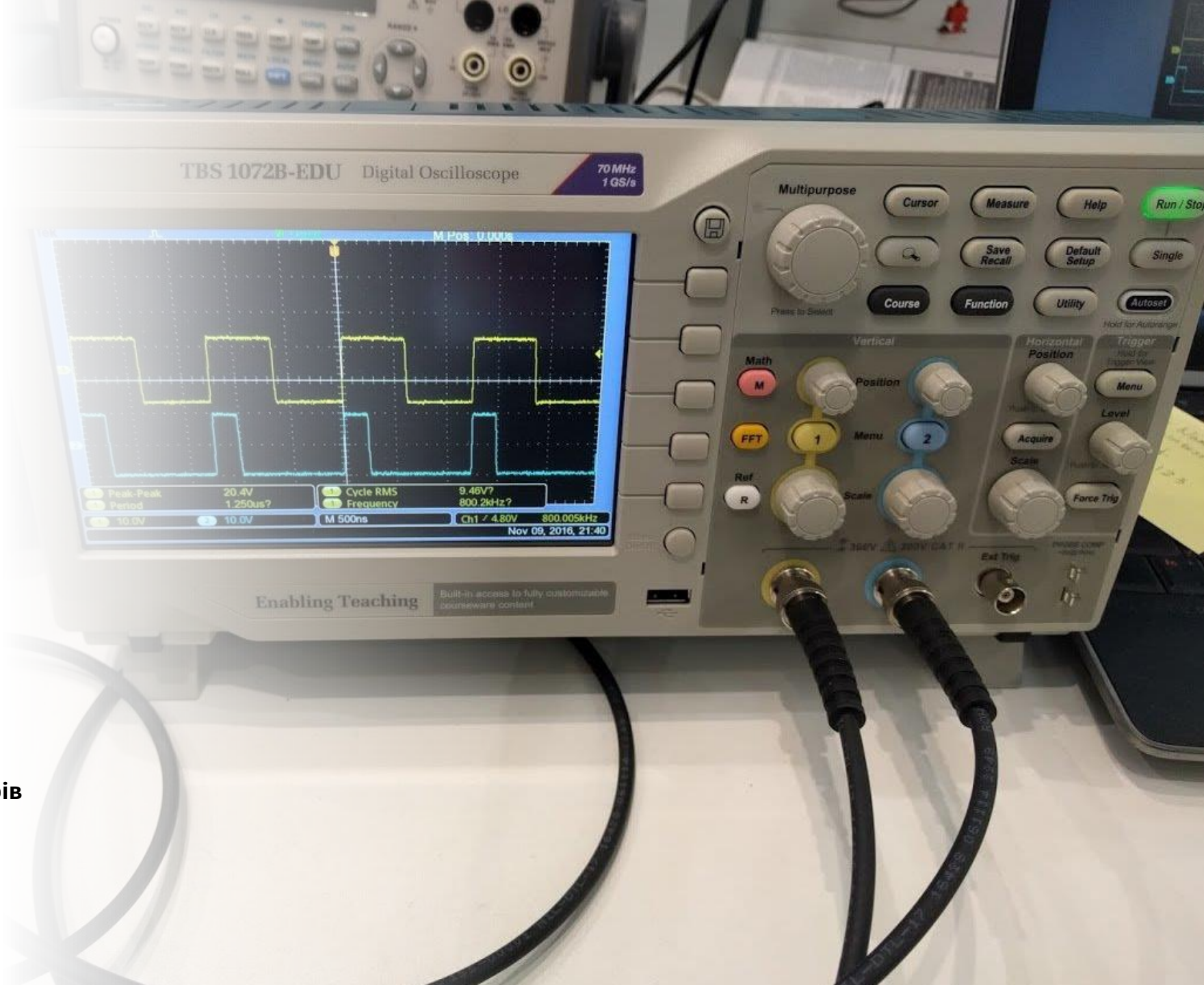


Логічний аналізатор

Запис та вимірювання параметрів динамічних цифрових сигналів. Декодування цифрових протоколів.

Осцилограф

Аналіз аналогових (при потребі і цифрових) сигналів та їх параметрів



Програматор

Завантаження програми в пам’ять мікроконтролера, налагодження

Лабораторний блок живлення

Подача необхідної напруги чи струму для живлення системи

Пірометр / тепловізійна камера

Детекція елементів системи котрі виходять за межі проектованих температур (як правило внаслідок помилок та збоїв)

Емуляція зовнішнього світу та відповідні інструменти

Підходи до емуляції зовнішнього світу

- **Емуляція зовнішнього світу не заміняє тестування в реальних умовах, швидше доповняє його. Тестування в реальних умовах може бути важкодоступним (наприклад для медичних приладів).**
- **Емуляція зовнішнього світу дозволяє перевірити умови яких важко добитися в реальному світі, але які очікувано можуть траплятися.**
- **Емуляція дозволяє суттєво пришвидшити тестування повільних процесів**
- **Емуляція також дозволяє перейти від тестування всього приладу до більш відокремленого тестування мікроконтролера та його програмного забезпечення.**

Так що ж ми збираємося емулювати?

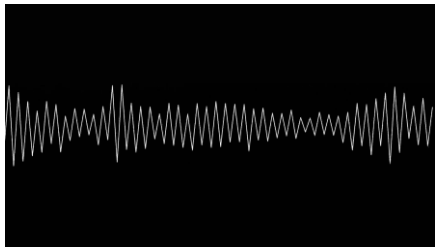
- **Беремо до уваги, що всі сигнали котрі потрапляють до мікроконтролера є перетвореними в електричну форму.**
- **Всі окрім часу...**
- **Емулюємо відповідні сигнали використовуючи зовнішні засоби.**
- **Перевіряємо як (unit level) функціонування мікроконтролера та його ПЗ.**
- **Розширяємо тести включаючи зовнішні компоненти для уникнення помилок управління зовнішніми компонентами.**

Що ми використовуємо для емуляції?

- **Для стандартних сигналів чи процесів використовуємо стандартні інструменти, наприклад генератор сигналів, чи кероване джерело напруги чи струму**

- **Для тестування комунікаційних протоколів на відповідність стандартів використовуються так звані compliance tester.**

- **При можливості записуємо реальні сигнали в електричній формі і тестуємо відтворюючи їх.**



Що ми використовуємо для емуляції?

- Нерідко нам приходиться також проектувати систему котра емулює зовнішній світ або його частину для нашого приладу.

- Коли ми проектуємо такий прилад ми як правило зосереджуємося на засобах та елементній базі котра забезпечує не ефективне використання ресурсу кінцевим приладом, а ефективне використання часу інженера проектанта.

Дякую за увагу!