Máš vytvořený program v mbedu, nahrál jsi jej do Nuclea a… a nedělá, co by dělat měl? Pak je dost dobře možné, že máš v programu nějakou chybu. A k ladění chyb slouží debuggování (ladění). Zatímco pro ladění počítačových aplikací máme k dispozici monitory pro vizuální kontrolu, pevné disky pro ukládání chybových protokolů a další komponenty a periferie, u mikrokontrolerů (dále jen MCU) je to složitější, protože (až na výjimky typu blikání diody) není vidět, co program v MCU dělá a zda se chová tak, jak očekáváme.

Takže **jak debugovat**? Existují dvě hlavní možnosti, jak ladit program v MCU.

1) **ICD** - Většina vývojových desek typu ST Discovery nebo ST Nucleo mají **takzvaný in-circuit debugger** (ICD – u ST procesorů konkrétně ST-Link), který poskytuje jakési „okno“ do veškerého dění v mikrokontroléru a který je velmi schopným společníkem.

   Umí například zobrazit instrukce nahrané v MCU (**program**), hodnoty proměnných (**watch**), hodnoty **registrů** procesoru (vyhrazená místa v paměti MCU, která uchovávají především informace o aktuálním nastavení a stavu MCU a umožňují nastavení MCU), je schopen program zastavit na požadovaném místě (**breakpoint**) nebo vykonávat program krok po kroku před zrakem uživatele.

   Tento způsob debugování mbed bohužel nepodporuje, podpora debugu byla v mbedu vyměněna za absolutní uživatelskou jednoduchost a přístupnost. Debugger totiž vyžaduje další ovladače a navíc ne všechny desky, které mbed podporuje, zvládají debug. Desky ST-Nucleo však podporují debug všechny, takže pokud se časem rozhodneš přejít na „velké“ IDE, o in-circuit debugování nebudeš ochuzen [5].

2) **Debugování bez ICD** – zde je třeba trocha kreativity a mít nějaké volné periferie procesoru, které by se daly využít k jakési primitivní signalizaci uživateli, například sériovou linku (nebo alespoň LED diodu). Pro základní debugování tato metoda bohatě postačí, ale obskurnější chyby se takto budou hledat špatně. Návod na tuto metodu následuje.

Debuggování bez debuggeru

Pokud už jsi ve svém MCU zprovoznil nějaký prográmek napsaný v mbed online IDE, už je Ti asi jasné, že mbed bohužel in-circuit debugging nepodporuje. Pokud chceš tedy svou aplikaci ladit, je nutné využít alternativní přístup k debugování.

Takto lze využít například některý z dostupných vstupně-výstupních kanálů, jako například sériovou linku (která je pravděpodobně nejužitečnější), nebo i prostou LED diodu. A máš-li doma například nějaký osciloskop nebo logický analyzátor, můžeš využít libovolných nevyužitých výstupních pinů procesoru pro zobrazování požadovaných informací. Sériovou linku lze použít například takto:

```
Serial pc(SERIAL_TX, SERIAL_RX);
```

Jsou tři hlavní věci, které se dají sledovat debuggrem i bez něj:

a) **Watch** - Chceš-li zobrazovat například hodnotu nějaké proměnné, není nic jednoduššího, než si tuto proměnnou průběžně vypisovat na sériovou linku a zobrazovat si ji v počítači s pomocí hyperterminálu.

```
pc.write("%d\n", variable);
```

syntax příkazu write() je totožná se syntaxí funkce printf() v klasickém jazyce C. Escape sekvence "\n" navíc po vypsání proměnné zalomí řádek.

b) **Breakpoint** - Chceš-li program zastavit na konkrétním řádku programu, aby sis mohl(a) v klidu prohlédnout proměnné nebo registry, můžeš program zastavit například funkcí

```
while(!pc.readable());
pc.read();
```

která program zastaví, dokud se na vstupu sériové linky neobjeví nějaký znak. Nezapomeň nakonec zavolat pc.read(), nebo se příznak pc.readable nikdy nenastaví zpět na nulu a breakpoint se již nikdy nevykoná. Volitelně si pak můžeš posláním konkrétního znaku (textu) z PC a využitím například konstrukce switch-case vybrat, která proměnná Tě zajímá.

Ještě jednodušší řešení by bylo pouze čekat na stisknutí tlačítka.

c) **SFR - (Special) FunctionRegister**– Registry jsou součástí mikrokontroléru a zajišťují především konfiguraci celého mikrokontroléru.

Tyto registry lze číst prakticky stejným způsobem jako proměnné, byť přečtené číslo často příliš smysl nedává. Registry totiž debugujeme většinou v souvislosti se špatným nastavením mikrokontroléru – a konfigurační registry (na rozdíl od datových registrů) často představují soubor hodnot typu „zapnuto/vypnuto“.

Číslo přečtené z (nebo zapsané do) registru přestavuje zpravidla 32bitovou hodnotu, ale nás zajímají převážně hodnoty jednotlivých bitů (až na pár výjimek - některé hodnoty mohou být vícebitové).

U LED diody můžeš zase využít několik rozdílných frekvencí blikání, nebo prosté svítí-nesvítí. Využití ostatních pinů s pomocí osciloskopu již necháme na Tvé kreativitě.

Jak měřit rychlost kódu

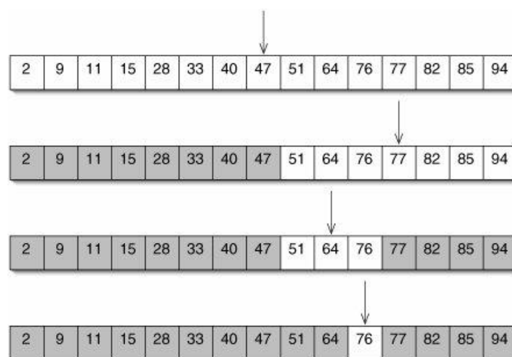
Máš napsaný kousek kódu, ale procesoru jeho vykonání trvá věčnost? Pak můžeš k metodám debugování přidat ještě **měření rychlosti kódu.** To uděláš snadno, například využitím Timeru, což je periferie procesoru přímo určená pro měření času. V mbed by pak implementace takového měření mohla vypadat například následovně:

```
Timer timer();
timer.start();
<Tvůj kód>
Int ubehlyCas = Timer.read_us(); (nebo Timer.read_ms(); Timer.read())
Timer.reset();
```

Jak objevit problematické místo?

Dalším dobrým nápadem je umisťování jakési „**debugovací informace**“ do kódu. To může vypadat například tak, že si za každou důležitější část kódu dáš výpis do sériové linky ve stylu „Prošel jsem bodem 1“. Pokud máš třeba nějaký cyklus (třeba for cyklus), tak si můžeš vypisovat „Prošel jsem 153. iterací cyklu“. Pokud si pak program spustíš a zároveň si otevřeš sériovou linku s počítačem, hned uvidíš, který příkaz byl poslední a kde se pravděpodobně stala chyba. Je jasné, že psát si každý pátý řádek nějakou stavovou zprávu je otravné. Proto klidně můžeš programovat i bez těchto zpráv a začít tohle řešit až ve chvíli, kdy se v programu něco pokazí. Pak stačí použít metodu půlení intervalů:

Dáš si nějakou stavovou hlášku do půlky kódu. Pokud se program zasekne ještě před touto hláškou, dám si novou hlášku opět do půlky té **první půlky kódu.** Pokud se program zasekne až za ní, pak si dáš novou hlášku do půlky té **druhé půlky kódu**. Zopakuješ celý postup tolikrát, dokud takto nevytipuješ dostatečně malý kousek kódu, ve kterém půjde chyba odhalit relativně snadno. Zkus si na Internetu vyhledat „půlení intervalů“, „binární hledání“, „binary search“ (doporučuji například [4]). Princip je stejný.



Obrázek 1 - hledáme číslo 76 v seřazeném seznamu půlením intervalů

Nevýhodou debugování bez debuggeru je, že pokud se rozhodneš uprostřed ladění debugovat jinou část programu, musíš si program **přepsat, znova zkompilovat a nahrát do mikrokontroléru**. Někdy se sice dá například měnit debugovanou proměnnou (registr) za běhu, ale je potřeba si promyslet a naprogramovat způsob komunikace, kterým svému mikrokontroléru řekneš, co chceš přesně debugovat, a i potom je to dost neflexibilní a náchylné na chyby.

A nakonec, pokud by Ti to nestačilo a chtěl(a) (či potřeboval(a)) bys využít funkcionalit plnohodnotného debuggeru, můžeš si z webu soutěže stáhnout návod, jak zprovoznit plnohodnotné desktopové vývojové prostředí pro mikrokontroléry s debuggerem, jako je například Keil MDK-ARM nebo Em::Blocks a využít in-circuit debuggingu.

Debugging ve velkém IDE

Je podrobněji rozebráno v návodu na desktopové IDE, který je dostupný z webu soutěže [5].

Odkazy

[1] https://developer.mbed.org/platforms/ST-Nucleo-F303RE/

[2] http://www.st.com/st-web-ui/static/active/en/resource/technical/document/reference_manual/DM00043574.pdf

[3] http://www.hw.cz/navrh-obvodu/software/programovani-v-jazyce-c-3-operatory-a-vyrazy.html

[4] http://www.algoritmy.net/article/21/Binarni-vyhledavani

[5] http://measure.feld.cvut.cz/system/files/files/cs/vyuka/predmety/A3B38MMP/CortexChallenge - Export do desktopoveho IDE.pdf