Martin Malý

Kódy, buildy, firmwary

Základy vývojářského řemesla pro tvůrce hobby elektroniky

Edice CZ.NIC

KÓDY, BUILDY, FIRMWARY
Základy vývojářského řemesla pro tvůrce hobby elektroniky

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2026
Kniha vyšla jako 36. publikace v Edici CZ.NIC.
ISBN 978-80-88168-89-8

ISBN 978-80-88168-89-8

— Martin Malý

Kódy, buildy, firmwary

Základy vývojářského řemesla pro tvůrce hobby elektroniky

— Edice CZ.NIC

Předmluva vydavatele

Předmluva vydavatele

Vážená čtenářko, vážený čtenáři,

software už dávno nejsou jen zdrojové kódy uložené někde v repozitáři. Mezi nápadem a fungujícím zařízením dnes stojí build systémy, automatizace, firmware, CI/CD pipeline, aktualizace, nástroje pro správu závislostí a spousta dalších vrstev, bez kterých by moderní vývoj prakticky nemohl existovat. A právě o těchto méně viditelných, ale důležitých částech technologického světa je nová kniha Martina Malého.

Pokud jste už některou z jeho předchozích knih četli, asi vás nepřekvapí, že ani tentokrát nejde jen o přehled příkazů, postupů nebo technologií. Autor má vzácnou schopnost vysvětlovat složitější věci způsobem, který neztrácí technickou přesnost, ale přitom zůstává srozumitelný a lidský. Možná i proto jde už o jeho sedmou knihu vydanou v Edici CZ.NIC.

Máme radost, že se v českém prostředí stále daří vydávat technické knihy, které neslouží jen jako rychlá dokumentace k jednomu konkrétnímu nástroji, ale pomáhají čtenářům pochopit širší souvislosti. Technologie se mění rychle, názvy frameworků přicházejí a mizí, ale principy dobrého návrhu, automatizace a spolehlivého vývoje zůstávají.

Kniha *Kódy, buildy, firmwary* míří ke čtenářům, kteří chtějí vědět nejen co použít, ale také proč věci fungují tak, jak fungují. Ať už pracujete s embedded zařízeními, automatizujete buildy, bastlíte vlastní hardware nebo vás prostě zajímá, co se děje mezi napsáním kódu a výsledným firmwarem v zařízení, věříme, že si v této knize najdete to své.

A možná při čtení zjistíte, že i věci, které bývají schované hluboko pod povrchem moderního vývoje, mohou být překvapivě zajímavé.

Pro mě osobně má tato kniha ještě jeden rozměr. Předchozí publikace Martina Malého vznikaly v Edici CZ.NIC v době, kdy jsem měla možnost být jejich součástí od prvních nápadů až po finální vydání. Tentokrát už pomyslnou štafetu převzala moje kolegyně, ale o to větší radost mám z toho, že knihy Martina Malého v Edici CZ.NIC dál vycházejí – se stejnou zvídavostí, nadhledem a schopností vysvětlovat technické věci bez zbytečné složitosti.

Kateřina Slavíková, CZ.NIC
Praha, 19. května 2026

Předmluva

Předmluva

Když jsem psal svou první knihu o elektronice, uvědomil jsem si, jak často se svět bastlířů a svět programátorů míjejí. Na jedné straně stojí ti, kdo si s páječkou a osciloskopem rozumějí beze slov, ale jakmile se před nimi objeví terminál, začnou být nejistí. Na druhé straně lidé, kteří dokonale ovládají vývojové prostředí, ale nepoznají rozdíl mezi rezistorem a kondenzátorem. Mezi těmito světy se rozkládá široká šedá zóna. A právě tam leží téma této knihy.

Zatímco první knihy se snažily přiblížit svět elektroniky programátorům, tato se snaží přiblížit svět programování elektronikům a konstruktérům. Zkušený programátor, který je jako doma v příkazovém řádku, nebojí se skriptování, umí překládat programy pomocí Makefile a rutinně řeší CI pomocí GitHub Actions, v ní nenajde nic, co by nevěděl. Ale pro kutilské vývojáře elektronických zařízení přináší, troufám si říct, spoustu užitečných tipů a námětů.

Mnoho domácích konstruktérů dnes používá PlatformIO nebo Arduino. Je to pohodlné, přímočaré, funkční. Jenže pohodlí má i svou cenu: odvádí pozornost od toho, co se vlastně děje. Když vývojové prostředí dělá vše za vás, snadno podlehnete dojmu, že to složité pod povrchem ani nechcete znát. Že hlavní je, aby to blikalo, měřilo, připojilo se k Wi-Fi a poslalo data. A dokud se daří, není důvod o tom pochybovat.

Jenže přijde chvíle, kdy se věci přestanou chovat podle očekávání. Kód se nezkompiluje, program se zacyklí, nebo se celý systém prostě odmlčí. V tu chvíli už nestačí hledat rady na fórech. Je potřeba rozumět tomu, co se děje uvnitř: jak se kód překládá, co dělá kompilátor, jak funguje make, jak se ladí firmware, co znamenají ty zvláštní soubory v knihovnách, proč se mezi nimi pohybuje nějaký ninja a proč mají názvy jako `CMakeLists.txt` a `actions.yml`.

Možná si říkáte, že „na takové to domácí bastlení" to není potřeba. A máte pravdu: **nutné** to není. Ale je to **výhodné**. Jakmile znáte postupy a nástroje, které stojí v pozadí, uvolní se vám ruce, které vám svazuje IDE, aniž si toho jste vědomi. Možná to po prvním přečtení bude znít složitě a jako „spousta práce navíc". Mým úkolem je ukázat, že to sice je práce navíc, ale není jí spousta, a že čas, který na začátku věnujete přípravě, se vám stokrát vrátí v pozdějších fázích.

Spousta domácích kutilských projektů zůstane ve fázi jediného kusu finálního zařízení, ale v současnosti je běžné, že mnoho autorů svůj výtvor publikuje ve formě schémat, návodů a kódu. Když se najdou další uživatelé a projekt začne „žít", tak je nejvyšší čas ho začít dělat pořádně, postavit ho na pevný základ. Mnoho kutilů se tohoto kroku bojí – nikdy takové nástroje a postupy nepoužívali, nepotřebovali je, ale jakmile vstoupí do celé věci další lidé, začne to být nutnost.

Nemusíte být ani přímo autoři. Může se vám stát, že použijete něčí konstrukci a kód, napadne vás vylepšení, zkusíte to, funguje a bylo by fajn ho nabídnout ostatním. Jak na to? V dokumentaci je

napsáno něco o *pull requestu* a *CI* a podobných věcech, což je přesně ten bod, kdy se na myšlenku „poslat to ostatním“ spousta lidí vykašle.

Tato kniha vznikla proto, aby se hobby elektronici těchto věcí přestali bát. Aby si mohli zefektivnit práci se softwarem, pochopili nástroje, které používají, a přitom se neutopili v detailech.

Nechci z nikoho dělat programátora, který žije v příkazové řádce. Chci ukázat, že i bez toho se dá získat jistota, že člověk ví, co dělá a proč. Že dokáže porozumět procesům, které se skrývají za tlačítkem „Build“.

V předchozích knihách jsem se zabýval hardwarem, mikrořadiči a tím, jak se chovají. Tady se zaměřím na prostředí, ve kterém s nimi pracujeme. Na nástroje, které nám umožňují řídit, skládat a ladit software. Na to, co se děje ve chvíli, kdy firmware vzniká, testuje se, nahrává a ladí. Nejde o teoretickou příručku, ale o pokus ukázat cestu k hlubšímu porozumění, bez nutnosti stát se vývojářem se znalostmi moderních vývojových procesů na plný úvazek.

Pokud jste někdy měli pocit, že programátorské věci jsou příliš složité, že ty zvláštní textové soubory, terminály a skripty nejsou pro vás, možná právě tato kniha vás přesvědčí o opaku. Není důvod se jich bát. Stačí se na ně podívat klidně, krok za krokem, a zjistíte, že dávají smysl a že vám dokážou ušetřit hodiny práce i nejistoty.

A navíc vám pochopení těchto nástrojů a postupů otevře cestu dál. Něco, co začalo jako garážový projekt jednoho nadšence, se právě díky tomu bude moct s minimem námahy proměnit ve skutečný produkt, u kterého se bude na vývoji podílet spousta lidí po celém světě. Protože rozdíl mezi hobby projektem a něčím, co je obecně použitelné, spočívá právě ve znalosti těchto *základů vývojářského řemesla*.

Poděkování a věnování

Poděkování a věnování

Knihu věnuji všem bastlířům a kutilům, kteří se nebojí jít, obrazně řečeno „s vlastní kůží na trh“, i když jde v tomto případě o kůži z křemíku a firmware; všem těm, kteří sebrali odvahu a svoje výtvory dávají k dispozici dalším lidem a starají se o ně. Věnuji ji i těm, kteří se k takovému kroku teprve chystají, a držím jim palce. Ta zkušenost je velmi silná a nepřenosná, nikdo a nic vás na ni nepřipraví, a ne každý je na to ustrojený. Nedivím se mnoha šikovným lidem, že do toho nikdy nepůjdou – i když je to velká škoda.

Děkuji všem bastlířům z „BastlGrupáče“ i Twitteru (který je teď X) za inspiraci, za zpětnou vazbu a za přímou i nepřímou podporu. Díky Petře, Michale, Martine, Martine, Martine, Martine, Milane, Ondřeji, Vlasto, Lukáši, Romane a další!

Děkuji všem čtenářům. Nebýt vás, nikdy bych nenapsal tolik knih, kolik jsem jich napsal.

Děkuji vydavateli, Edici CZ.NIC, protože nebýt Edice, a jmenovitě paní Kateřiny a paní Anny, nenapsal bych knihu žádnou!

A děkuji nejbližším, hlavně partnerce Lence, za trpělivost a toleranci, protože když píšu knihu, jsem celé dny zavřený ve svém světě a vycházím jen jíst a spát.

Obsah

Předmluva vydavatele **7**

Předmluva **11**

Poděkování a věnování **15**

1 Rychlokurz efektivního vývoje **23**
1.1 Příběh jednoho projektu 23
1.2 Jak vzniká firmware 27
1.3 Linux? Proč vyvíjet na Linuxu? 31
1.4 Od Arduino IDE k PlatformIO… a dál 34
1.5 Odbočka: Linux ve Windows 53
1.6 První blik 70
1.7 Skriptování a automatizace 90
1.8 Make a Makefile 103
1.9 Přecházíme na CMake 115
1.10 Orientace v SDK 141
1.11 Ninja 157
1.12 Kconfig a konfigurace projektu 170
1.13 Knihovny, závislosti a verze 185
1.14 Ladění a měření 228
1.15 Testování 273
1.16 Jeden repozitář, víc cílů 289
1.17 CI a GitHub Actions 298
1.18 Udržitelný projekt 311
1.19 Úpravy cizích knihoven 346

2 Nástroje podrobněji **361**
2.1 Linux BASH 361
2.2 OpenOCD 367
2.3 GDB do hloubky 376
2.4 Několik vývojových prostředí v jednom počítači 388

3 Péče o firmware **409**
3.1 Správa knihovny a spolupráce s přispěvateli 409
3.2 Distribuce firmware 420

4 Dodatky **431**
4.1 Od Arduina k C 431
4.2 Picoprobe 441

5 Doslov **447**
Další knihy tohoto autora: 452

— Obsah

1 Rychlokurz efektivního vývoje

1 Rychlokurz efektivního vývoje

1.1 Příběh jednoho projektu

Bylo, nebylo… Možná nebylo přesně takto, ale mohlo být.

Jeden bastlíř, říkejme mu třeba Tomáš, si chtěl postavit jednoduchý projekt na víkend: pásek s RGB LED, který reaguje na hudbu. Tedy nic složitého. Koupil si Raspberry Pi Pico 2 s RP2350, pár pásků s WS2812B a mikrofon. V Arduino IDE napsal základní smyčku, která snímala hodnotu z mikrofonu a podle hlasitosti měnila barvu všech LED najednou. Červená pro ticho, zelená při průměrném hluku, modrá při hlasité hudbě. Za dvě hodiny měl funkční prototyp a večer poslouchal hudbu se světelným doprovodem.

Fungovalo to dobře, ale po týdnu ho napadlo, že by bylo zajímavé mít efekty, které reagují nejen na hlasitost, ale i na frekvence. Aby basy rozsvěcely spodek pásku červeně, výšky nahoře modře a středy zeleně uprostřed. Pustil se do programování, použil knihovnu pro FFT, ale nějak to začalo dřít. Snažil se použít obě jádra v RP2350, jedno pro obsluhu LED pásku s přesným časováním, druhý pro vzorkování zvuku a spektrální analýzu, ale objevily se nějaké podivné chyby. Při pokusu o spuštění kódu na druhém jádře začala aplikace občas zamrzat a aktualizace pásku někdy neproběhla, jak by měla. Podle všeho při intenzivních výpočtech FFT docházelo k přetečení zásobníku a softwarová komunikace mezi jádry nebyla dostatečně spolehlivá. Ani StackOverflow nenabízel řešení. Pak na to přišel v dokumentaci: knihovna pro Arduino má problémy při použití více jader. Navíc potřeboval nějak organizovat kód, protože „vše v jedné složce“ začalo být chaotické.

Tomáš tedy musel přejít na „čisté“ `pico-sdk`, které mu nabídlo lepší podporu pro více jader. Ostatně většina dotazů na fórech končila tímtéž závěrem: *Přejděte na SDK*.

Přechod znamenal opustit pohodlí Arduino IDE a naučit se překládat projekty ručně. Pico-sdk vyžaduje správné nastavení cest pro `#include`, linkování knihoven a specifikaci cílového čipu. První kompilace se mu podařila až po půl hodině googlení správného postupu pro `arm-none-eabi-gcc`. Ale výsledek stál za to! Měl konečně stabilní dvoujádrovou aplikaci, LED pásek běžel hladce na jednom jádře, zatímco druhé počítalo FFT v reálném čase.

Po měsíci víkendového ladění měl Tomáš šest různých vizuálních efektů a kód organizovaný do osmi souborů. Ruční kompilace s dlouhými cestami pro `#include` byla únavná. Každý build by vyžadoval zadat osm příkazů `arm-none-eabi-gcc` s komplexními parametry. Už na začátku se mu stalo, že zapomněl přeložit jeden soubor, nevšiml si toho a celý firmware přestal fungovat. Musel si napsat skript, který všechny kroky automatizoval. Teď stačilo napsat `./build.sh` a za půl minuty měl nový firmware připravený k nahrání.

Skript Tomášovi fungoval skvěle, dokud nepřidával nové funkce a jen upravoval to, co už měl. Ale pak ho napadlo, že by si mohl přidat `FatFS` pro ukládání audio vzorků na SD kartu a `tinyusb` pro komunikaci s počítačem. Každá knihovna ale měla své požadavky a překládací skript začal být nepřehledný. Horší bylo, že překládal všechno pokaždé znovu, i soubory které se nezměnily. Znovu a znovu se při každém testu překládaly obří knihovny pro FFT a pro hardware, a z pár sekund překladu byly najednou skoro pauzy na kávu.

Logický krok, který musel přijít, se jmenoval `make`. Ten automaticky sledoval změny v souborech a překládal jen ty, které se skutečně změnily. Sledování změn fungovalo spolehlivě a Tomáš si konečně mohl dovolit experimentovat s kódem bez dlouhého čekání na každý build. Tím si výrazně zrychlil vývojový cyklus a za týden přidal tři nové efekty včetně spektrogramu, běžícího po celé délce pásku.

Jenže pak se dostal k pokročilým matematickým funkcím. Detekce beatů vyžadovala analýzy, navíc si k pásku pořídil i matici a chtěl udělat 2D efekty. Ty ale potřebovaly další výpočty, které chtěl optimalizovat předpočítanými tabulkami, generovanými při kompilaci. `Makefile` s těmito složitými závislostmi začal být nepohodlný a čas překladu se opět protáhl na neúnosnou dobu.

Tak přišel CMake, který elegantně vyřešil problémy s komplexním procesem sestavování a překladu. Pico-sdk má oficiální podporu CMake s připravenými definicemi, takže přidání `FatFS` nebo `tinyusb` bylo otázkou pár řádků. Správu závislostí mezi generovanými tabulkami a zdrojovými soubory zvládal CMake automaticky. Build byl opět čistý a přehledný.

Výkon při překladu ale stále nebyl ideální. Tomáš objevil generátor Ninja a časy opakovaných překladů se propadly z desítek sekund na jednotky. Ninja optimalizoval paralelní kompilaci tak dobře, že Tomáš mohl experimentovat bez přerušení. To mu umožnilo rychle vyladit spektrální analýzu do stavu, kdy přesně rozeznávala bicí, basové linky a melodické nástroje.

Jenže pak se někde pochlubil se svým výtvorem a pár lidí řeklo, že by to chtěli taky. Ale různí lidé chtěli různé varianty, někdo jen základní analyzátor spektra, jiný všechny efekty. Jeden měl třicet LED, druhý sto. Někdo chtěl USB komunikaci, jiný potřeboval podporu SD karty. Jak se říká – sto lidí, sto chutí. Tady jich sice nebylo sto, ale i tak už bylo neúnosné řešit množství různých konfigurací změnou hlavičkových souborů.

Nakonec problém vyřešil `KConfig` a `menuconfig`. Uživatel si jednoduše vybral požadované funkce přes přehledné menu a systém automaticky vygeneroval odpovídající konfiguraci. Tomáš mohl udržovat jeden zdrojový kód, který se choval různě podle toho, co uživatelé chtěli. Měl pak vývojovou verzi s detailním laděním pro sebe, a k ní optimalizovanou verzi bez ladicích výstupů pro zájemce.

Jak ale jeho kód začali používat různí lidé s různým hardware, začaly se objevovat i zákeřné chyby. Při některých kombinacích efektů se LED občas rozblikaly špatnými barvami. S více než dvěma

sty LED se objevovala podivná poškození dat. Nepravidelné problémy časování v dual-core komunikaci způsobovaly výpadky několikrát za hodinu. Ladění přes `printf` už nestačilo; chyby se objevovaly nepravidelně a volání `printf` zase ovlivňovalo časování tak, že problém zmizel.

Nezbylo než sáhnout po skutečných nástrojích pro ladění. Tomáš zvolil debugger GDB, připojený přes SWD, aby našel příčinu. A viděl zajímavé věci, třeba poškození paměti kvůli překročení velikosti bufferů v DMA komunikaci s LED. Problémy při předávání dat z FFT, které kvůli časování v ladicí verzi nebyly. Objevil chyby v indexování tabulek. Díky breakpointům a sledování paměti v reálném čase mohl problémy nejen najít, ale i opravit. Stabilita se tím výrazně zlepšila.

Ale růst projektu znamenal i rostoucí složitost algoritmů, hlavně ve chvíli, kdy jeden z uživatelů začal psát rozšiřující algoritmy a posílat je Tomášovi. Algoritmy ale bylo těžké testovat na reálném zařízení. Když někdo z uživatelů nahlásil, že beat detection má falešné poplachy při určitých žánrech hudby, Tomáš potřeboval způsob, jak algoritmy systematicky otestovat před implementací.

Napsal si jednotkové testy pro samotné algoritmy, které mohl spustit na svém počítači s připravenými audio vzorky a očekávanými výsledky. Rutiny pro FFT testoval proti referenční implementaci, mapování barev kontroloval s přesnými RGB hodnotami, detekci beatů ověřoval na skladbách s ručně označenými dobami. Testováním na PC si ověřil, že algoritmy jsou správné, ještě před tím, než je začal implementovat do firmware.

Po roce vývoje měl Tomáš hezký a funkční systém, ale uživatelé začali žádat pokročilé funkce: Wi-Fi ovládání přes mobilní aplikaci, integraci s chytrou domácností přes protokol Matter, možnost synchronizovat víc LED pásků současně. RP2350 ale neměl Wi-Fi a jeho výkon už pro komplexní síťové protokoly nestačil.

Rozhodl se vytvořit druhou verzi hardware, tentokrát založenou na ESP32. Zachoval ale i verzi 1 s RP2350 jako rychlou, energeticky úspornou variantu pro bateriové aplikace. Ve verzi 2 přidal síťové funkce a pokročilé zpracování dat. Jeden zdrojový kód, dva cíle, a uživatel si vybral podle potřeb. CMake naštěstí zvládá sestavování pro víc různých cílů s podmíněným překladem podle zvolené platformy.

Víc cílových platforem ale znamenalo nové riziko. Změna pro ESP32 mohla rozbít firmware pro RP2350 a naopak. Jeden vývojář mohl otestovat třeba jen jednu platformu a nezaregistrovat problémy na druhé. Manuální testování obou platforem se všemi konfiguracemi zabralo spoustu času a objevily se záludné zavlečené chyby.

Tomáš nakonec vyřešil i tento problém, a to průběžnou integrací (*Continuous Integration*) v GitHub Actions. Při každém commitu nové revize firmware se spustil překlad a test firmware na obou platformách se všemi populárními konfiguracemi. Pokud nějaká úprava rozbila jednu platformu, přišlo se na to během minut, ne až po týdnech, když si uživatelé stěžovali.

Do vývoje se už zapojili další lidé, což bylo fajn, ale růst týmu přinesl organizační problémy: „Tvůj kód nefunguje!“ – „Ale na mém počítači to funguje!“ Jeden patch fungoval jen s konkrétní verzí SDK, různí vývojáři měli různé verze toolchainů, CI používal jiné verze než lokální prostředí. Nekonzistentní výsledky překladů začaly být frustrující a některé bugy se objevovaly jen v určitých verzích nástrojů.

Tomáš proto zadal přesné verze všech nástrojů a knihoven a přenesl sestavování do Dockeru a jeho kontejnerů. Tím zajistil stejné prostředí pro všechny vývojáře.

Po čase měl Tomáš z víkendového bastlení profesionální projekt. Více než padesát vizuálních efektů, dva podporované čipy, Wi-Fi ovládání, integrace se systémy chytré domácnosti, automatizované testování a reprodukovatelné buildy. Projekt používaly stovky lidí po celém světě a měl aktivní komunitu přispěvatelů. Z jednoduchého světelného pásku reagujícího na hlasitost se stal pokročilý systém s profesionálním vývojovým workflow.

Dobře, ten příběh jsem si vymyslel, na GitHubu jeho kód nenajdete, přesto je ale založený na reálných událostech, které se dnes a denně stávají hobby bastlířům po celém světě. A něco z toho se určitě stalo i vám. A pokud ne, tak se vám to teprve stane.

Cesta od Arduina k plnohodnotné správě softwarového projektu nebyla v tom příběhu naplánovaná předem. Ani vy ji asi neplánujete. Ale každý evoluční krok měl opodstatnění, které vyplývalo z omezení předchozího řešení a rostoucích požadavků. Arduino IDE nestačilo pro dual-core; ruční kompilace byla pomalá; Makefile nezvládal složité závislosti; nedostatečná konfigurace vedla k nepohodlným překladům; chyby vyžadovaly systematické ladění; rostoucí komplexita potřebovala unit testy; požadavky uživatelů si vyžádaly více platforem; tým rostl a potřeboval automatizaci; různá prostředí způsobovala nekonzistentní buildy.

Nic z toho rozumný vývojář nepoužije jen proto, že to existuje a je *cool*. Každý další nástroj vždy řeší konkrétní problém, který už nejde obejít jednodušším způsobem. A právě to je podstata moderního vývoje (nejen) firmware – nástroje nejsou cíl, ale prostředek k řešení reálných problémů ve chvíli, kdy něco, co jste si postavili na koleni, začne přerůstat ve větší projekt.

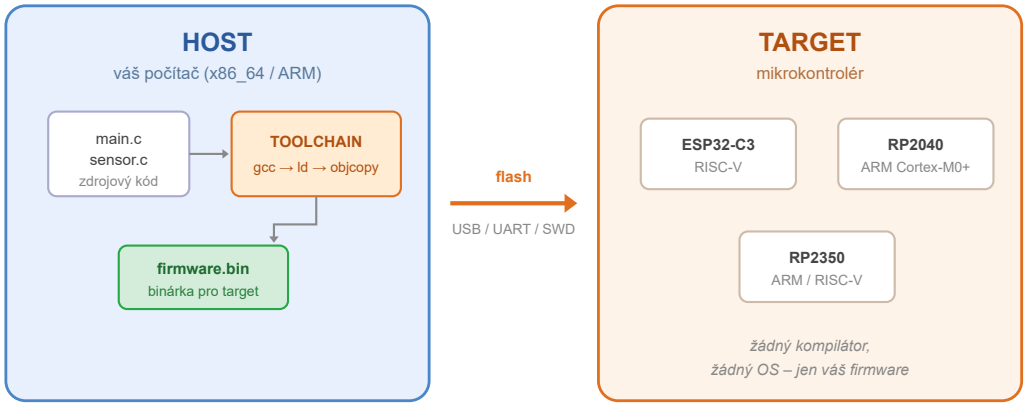
V následujících kapitolách si projdete podobnou cestu jako Tomáš z tohoto příběhu. Nebude to tak jednoduché, jak to ukazují promo videa, leckdy se natrápíte, budete muset přepsat kus kódu, změnit návyky, dělat věc „složitěji“ a někdy určitě s nostalgií zavzpomínáte, jak to bylo jednoduché v Arduinu. Bylo. Ale nerostlo to, a když to rostlo, tak to dřelo. Pokud chcete růst, musíte tomu něco obětovat a dát věcem, ale i sobě, kázeň a řád. Zvládnete to i sami, po svém, ale lepší je použít takové postupy, které používá výrazná většina kolegů z oboru.

1.2 Jak vzniká firmware

Máte před sebou možná Arduino IDE, možná PlatformIO. Spustíte nový projekt, napíšete pár řádků pro blikání LEDkou, připojíte vývojovou desku k počítači a v editoru stisknete tlačítko **Build**. Pak se „něco děje“, a na konci je program v paměti čipu a bliká dioda.

Jenže co se přesně stalo během té doby, kdy vývojové prostředí z uspořádaných řádků kódu stvořilo funkční program? Magie? Černá skříňka? Ve skutečnosti něco mezi tím: odehrál se promyšlený řetězec kroků. Jakmile mu porozumíte, přestanete jen psát kód a začnete ho ovládat.

Začnu u pojmu **firmware**. Není to obyčejný program, tedy *software*. Software je obecně jakýkoli program, který běží na operačním systému, například na počítači, a využívá jeho služeb, jako je správa paměti a přístup k souborům. Firmware je speciální druh software, který je spojen s *hardware*, na kterém běží. Je určený pro provoz na konkrétním zařízení, v konkrétním čipu, v přesně vymezené paměti. Je to základní, neměnný ovládací program, který oživuje mikrokontrolér (nebo jiný CPU, ale nejčastěji to bývá mikrokontrolér, tedy MCU, česky *jednočip*).



Cesta začíná u textového souboru, dnes typicky v jazyce C nebo C++, ale může to být i Rust, Zig, Nim a další. To je náš *zdrojový kód*, tomu musí rozumět hlavně člověk. Stroje ale lidské popisy nechápou; potřebují sled instrukcí, tedy *strojový kód*, vyjádřený jedničkami a nulami. Proměnu kódu *zdrojového* na kód *strojový* zařizuje dnes nejčastěji řetězec nástrojů, zvaný **toolchain**.

Nezbytná jazyková vsuvka: Ačkoli jsem velký milovník češtiny a je mi z duše protivná „češtingličtina“, kterou jsou zamořené české návody, literatura a dokumentace, tak jsem také pragmatik. Pro některé výrazy nemáme vhodný český ekvivalent, jako třeba pro „toolchain“. Je to „řetězec nástrojů“, ale s takovým po-

pisem je to jak s „mezistykem" (kdysi navrhované slovo pro interface) nebo „systémem vzájemné výměny" (handshake). Některá anglická slova si čeština ohnula, takže se v Gitu běžně „commituje" a „commitnutý" výsledek se „pushne" do „repozitáře" i s „buildem". Trošku mě to dráždí, ale dokážu s tím i v této knize nějak žít. Navíc jsem realista, a pokud lidé okolo vás budou flashovat, tak by bylo hloupé vám existenci takového termínu zatajovat a tvrdošíjně lpět na „nahrání binárního kódu do vnitřní paměti mikrořadiče". Dobře si pamatuji doby, kdy jsem sám luštil, co je v českých knihách „klepnutí" a co „poklepání"…

Téměř vždy v této sadě najdeme programy, pojmenované preprocesor, kompilátor, assembler, linker a flasher. V toolchainu na sebe navazují. Tato sada se liší pro každou cílovou architekturu, pro každý cílový procesor, a to je podstata její existence. Používáme `xtensa-esp32-elf-gcc` pro starší ESP32, nebo `riscv32-esp-elf-gcc` pro novější čipy s RISC-V, jako je ESP32-C3. Pro čipy RP2040 s jádrem ARM používáme `arm-none-eabi-gcc`.

První na řadě je **preprocesor**. Ten vezme náš zdrojový kód a připraví ho pro kompilátor. Je to čistě textová operace, kde se neřeší logika ani syntaxe, pouze doslovné nahrazení. Má tři hlavní úkoly: vloží obsah hlavičkových souborů, které definují funkce, vyřeší podmíněné překlady pomocí příkazů jako `#ifdef` a `#endif` a nakonec nahradí symbolické konstanty jejich skutečnými hodnotami. Například řádek `#define LED_PIN 2` je pokyn pro preprocesor: *V celém kódu nahraď LED_PIN číslem 2*.

Preprocesor vytvoří jeden obrovský, čistý textový soubor, ve kterém už nejsou žádné makra ani podmínky. Tím končí jeho role.

S tímto textem začíná pracovat **kompilátor**. Ten má za úkol převést kód vyšší úrovně, jako je jazyk C, do kódu nízkoúrovňového, který je bližší instrukcím procesoru. Kompilátor kontroluje syntaktickou správnost a optimalizuje kód, aby běžel co nejrychleji a zabíral co nejméně paměti. Na výstupu vzniká **objektový soubor**, nejčastěji s koncovkou `.o` nebo `.obj`. Tento soubor ještě není spustitelný. Je to jen útržek kódu, který sice obsahuje strojové instrukce a data z vašeho zdrojového kódu, ale neví, kde přesně v paměti bude ani kde najde funkce z jiných souborů nebo z knihoven.

Ještě jedna jazyková poznámka: Správně česky je „kompilátor", ale setkáte se s názvem „kompiler".

Typický krok, který kompilátor provádí, vypadá takto (pokud si jej pustíte v terminálu):

```
riscv32-esp-elf-gcc -c main.c -o main.o -Os -Wall
```

Příkaz říká kompilátoru, aby přeložil soubor `main.c` do objektového souboru `main.o`. Přepínač `-c` znamená *přeložit, ale nelinkovat*. Přepínač `-Os` optimalizuje kód pro velikost a `-Wall` zapíná všechna varování, což je dobrá praxe. Tento krok se opakuje pro každý zdrojový soubor v našem projektu,

včetně všech souborů ze všech knihoven, které používáte (a někdy o tom ani nevíte). Na konci máme spoustu objektových souborů, a teď je musíme spojit.

Často se během kompilace tiše odehraje ještě jeden krok, **assembler**. Kompilátor může převést kód z jazyka C do jazyka symbolických adres, kde je každá strojová instrukce zapsaná svým symbolickým jménem, a až poté do strojového kódu (a většinou to udělá v jednom kroku). Někdy, například u velmi rychlých nebo hardwarově specifických rutin, píšeme kód přímo pro assembler. Třeba samotný startovací kód mikrokontroléru je většinou napsaný v tomto strojovém jazyce, ale bývá ukrytý v hlubinách systémové knihovny. I ten musí být pomocí *assembleru* převeden do strojových instrukcí.

Všechny tyto útržky kódu, objektové soubory, je nutné spojit do jednoho programu. To je práce pro **linker**. Linker je ten, kdo program skládá a umisťuje do paměti. Používá k tomu takzvaný **linker script**, což je přesný návod, jak má být výsledný program uspořádán. Je v něm definováno, která část kódu a dat má být na jaké adrese v paměti Flash, a která v paměti RAM. Linker bere v úvahu, jaké funkce a proměnné se volají napříč soubory, a propojí je. Vytvoří tabulku, kde je každá funkce a proměnná umístěna na své finální adrese. Tím se vyřeší všechna volání funkcí, které kompilátor v souborech `.o` označil jako externí, tedy takové, které zdrojový kód neobsahoval, ale je důvod se domnívat, že „někde budou“. Linker musí vědět, kde je najde, aby je, pokud je najde, připojil do výsledného programu.

Program je rozdělený na základní sekce. Sekce `text` obsahuje samotný spustitelný kód, tedy instrukce. Sekce `data` drží inicializované globální proměnné, které se musí z paměti Flash zkopírovat při startu do paměti RAM. Například pole konstant `int pole[3] = {1, 2, 3}`. A sekce `BSS` (Block Started by Symbol) obsahuje neinicializované globální proměnné, které se při spuštění vynulují. Tam přijde například pole, definované jako `int pole[10]`. Proč je toto dělení důležité? Protože určuje, kolik drahocenné paměti RAM váš program spotřebuje, i když zdánlivě nic nedělá. Právě BSS je to, co zabere většinu paměti RAM.

Když se program po spuštění chová divně, nebo dojde k pádu, často je to kvůli chybě v linkování: kód volá funkci z nesprávné adresy, nebo přistupuje k paměti, která mu nepatří.

Linker poskládá objekty do jednoho **spustitelného souboru**, nejčastěji s koncovkou `.elf`. Formát ELF (Executable and Linkable Format) obsahuje program samotný, ale i spoustu informací pro ladění, včetně jmen proměnných a přesných adres. Tyto informace jsou zcela zásadní pro práci s ladicími nástroji jako je GDB, o nichž si povíme později. Zároveň linker může vytvořit soubor `.map`, který obsahuje mapu paměti, tedy přehled o tom, jaké sekce jsou na jakých adresách a kde v nich sídlí jaká část programu.

Typický příkaz linkování je složitější, protože zahrnuje desítky knihoven, ale podstata vypadá takto:

```
riscv32-esp-elf-ld -T linker.ld main.o periferie.o -o firmware.elf
```

Přepínač `-T linker.ld` je klíčový. Říká linkeru, z jakého linker skriptu má načíst definice pro daný stroj.

Soubor ELF ale nelze jen tak poslat do mikrokontroléru. Je příliš velký a plný informací, které jsou pro běžný provoz nepotřebné. Proto se musí převést a vyčistit do formy binárního souboru, familiárně zvaného **binárka**. Binární soubor s koncovkou třeba `.bin` nebo `.uf2` obsahuje pouze čisté strojové instrukce a data, která čip potřebuje k běhu, a to v podobě, v jaké budou zapsány do Flash. Binárka je finální a nejmenší forma programu, ta podstatná část. Odstraněním ladicích informací šetříme čas i místo v paměti.

Promiňte mi to, milí čtenáři, ale místo opakování „výsledný soubor v binárním formátu" budu v knize používat právě tu slangovou „binárku". Ostatně vývojáři tomu mezi sebou neřeknou jinak.

Finální krok v počítači je **flashování**. Tím se program přenáší do mikrokontroléru. Používají se k tomu nástroje jako `esptool.py` pro ESP32 nebo `picotool` pro Raspberry Pi Pico. Flashování není nic jiného než sériová komunikace, která zapisuje náš binární soubor do paměti Flash, tedy do trvalé paměti v čipu. Program tam pak čeká na své spuštění.

Po úspěšném nahrání následuje **reset** mikrokontroléru. Když moderní čip připojíte k napájení nebo dáte příkaz k restartu, nezačne hned spouštět váš program. Nejprve běží malý kousek kódu, který je napevno v čipu, takzvaný *bootloader* nebo *ROM kód*. Bootloader, podle typu čipu, třeba zkontroluje paměť, nastaví základní registry, zkopíruje data, připraví běh na více jádrech, zkrátka udělá nezbytné věci, a pak předá řízení uživatelskému programu, a to na přesně definované adrese. Tam proběhne startovací kód aplikace (zmiňoval jsem se o něm u linkeru), který se postará o inicializaci paměťových struktur a periferií, a nakonec zavolá funkci `main()`. Teprve v tomto bodě běží váš program.

Stisknutí tlačítka *Build* spustilo celý tento proces, kde se textový soubor (zdrojový kód) nejprve připravil (preprocesor), přeložil do strojových instrukcí (kompilátor), poskládal dohromady a umístil do paměťové mapy (linker). Poté se odstranila zbytečná metadata, vznikla finální binárka, která se nahrála do čipu (flasher), a nakonec, po resetu, program běží.

Celému procesu od zdrojových kódů po výslednou binárku se anglicky říká „build", český termín pro to je sestavení, někdy (méně přesně) překlad. Do detailu vzato je překlad jen ta první část, preprocesor a kompilátor, zbytek je „linkování", česky spojování.

1.3 Linux? Proč vyvíjet na Linuxu?

Budu upřímný: Vyvíjím ve Windows. Důvod to má jednoduchý: Windows jsou na mém pracovním počítači od roku 1995. Několikrát jsem zkusil přejít na „Linux na desktopu“, ale nikdy to nebylo bezproblémové, vždycky to někde dřelo, prošel jsem si spoustou problémů s ovladači a podobnými věcmi, a v neposlední řadě dělám na pracovním PC věci, pro které v Linuxu neexistují srovnatelné alternativy.

Přešel bych na MacOS, který alespoň pro mě spojuje to nejlepší z obou světů. Ale tam jsem limitovaný hardwarem. Moje pracovní stanice má asi čtyři disky, čas od času vyměním procesor za novější, koupím novou výkonnější grafickou kartu a počítač se mnou roste. U Maců bych byl odkázaný na modely, které Apple navrhl a prodává, anebo bych platil spoustu peněz za počítače na míru. MacOS mám, na notebooku je skvělý, v práci jsem ho používal, ale doma mám Windows.

Tato kniha se bude věnovat primárně Linuxu. Bude vás provázet celou knihou a je lepší si to říct hned na začátku, než abyste se na každé druhé stránce ptali: „A to nejde udělat normálně ve Windows?“

Odpověď je: jde to, samozřejmě. Prakticky si ale někdy přiděláte práci, kterou byste nemuseli dělat.

Podívejte se na jakýkoli embedded projekt na GitHubu. Otevřete ESP-IDF, pico-sdk, Zephyr, nebo třeba NuttX. Podívejte se na jejich skripty, návody, příklady. Všude uvidíte bash, všude uvidíte cesty s lomítkem, všude se předpokládá, že máte make, cmake, python3, git, find, sed, grep a dalších dvacet utilit, které v Linuxu *prostě jsou*.

Ve Windows nejsou, nebo se jmenují jinak, nebo mají jiné parametry, nebo mají trošku jiný formát výstupu…

Jasně, můžete nainstalovat Git for Windows, který s sebou přinese Git Bash a s ním i minimální sadu unixových nástrojů. Můžete nainstalovat CMake pro Windows, Python pro Windows, Make pro Windows (a doufat, že se vám nepopere s tím Make, který přišel s něčím jiným, třeba z VS). Můžete si pohrát s MSYS2 nebo MinGW. Ale pořád budete lepit dohromady něco, co v Linuxu funguje samo od sebe.

A teď k tomu podstatnému: nejde jen o to, jestli ty nástroje *existují*. Jde o to, jestli fungují *stejně*.

Například Windows používají zpětné lomítko jako oddělovač cest. Linux používá dopředné lomítko. Zdánlivá drobnost, ale má dalekosáhlé důsledky.

Představte si build skript, který v sobě má:

```
TOOLCHAIN_PATH=/opt/gcc-arm/bin
SRC_DIR=$(dirname "$0")/src
$TOOLCHAIN_PATH/arm-none-eabi-gcc -I${SRC_DIR}/../include ...
```

Tato třířádková ukázka předpokládá tři věci, které na Windows nativně nefungují. Za prvé cestu s lomítkem. Za druhé příkaz `dirname`, který na Windows neexistuje. Za třetí expanzi `$()` a `${}`, které rozumí `bash`, ale `cmd.exe` ne, a ani `PowerShell` si s tím někdy neporadí.

Možná si říkáte: „Ale přece můžu použít PowerShell." Můžete. Ale ty skripty, které najdete u ESP-IDF, u pico-sdk, u každého druhého projektu na GitHubu, jsou psané pro `bash`. Ne pro PowerShell. Takže buď každý skript přepisujete, nebo si nainstalujete `bash`. A pokud si nainstalujete `bash`, tak jste vlastně… nainstalovali Linux. Jen trochu nepohodlně.

Nebo se podívejte na něco ještě jednoduššího. Soubor CMakeLists.txt v pico-sdk obsahuje řádky jako:

```
execute_process(COMMAND ${CMAKE_COMMAND} -E env
    "PICO_SDK_PATH=${PICO_SDK_PATH}"
    ${Python3_EXECUTABLE} ${PICO_SDK_PATH}/tools/pioasm/gen_sdk.py ...
)
```

Na Linuxu toto projde. Na Windows se vám může stát, že `Python3_EXECUTABLE` ukazuje na cestu s mezerou (`C:\Program Files\Python312\python.exe`), a najednou se vám rozpadne parsování příkazu. Nebo `PICO_SDK_PATH` obsahuje zpětná lomítka, a skript s tím nepočítá. Jsou to drobnosti, ale každá taková drobnost vás stojí půl hodiny googlování.

Překladové toolchainy pro ARM a RISC-V (tedy ty, kterými překládáte firmware pro mikrokontroléry) jsou primárně vyvíjené a testované na Linuxu. Verze pro Windows existují a většinou fungují shodně. Většinou. Občas ale narazíte na problém, který na Linuxu prostě nenastane, a proto autor nepovažuje za nutné ho řešit, pokud o něm vůbec ví…

Typický příklad: nainstalujete `arm-none-eabi-gcc` na Windows, přidáte ho do `PATH`, a překlad projde. Jenže pak chcete použít OpenOCD pro ladění přes SWD sondu, a zjistíte, že potřebujete Zadig pro přeřazení USB ovladače, protože Windows to samy neumí. Na Linuxu napíšete pravidlo `udev` (jeden řádek do jednoho souboru) a je hotovo.

Nebo chcete flashovat ESP32. Na Linuxu připojíte desku a v `/dev/` se objeví `ttyUSB0` nebo `ttyACM0`. Na Windows se objeví COM3 – a doufáte, že je to ten správný COM port, protože jinak si otevřete Správce zařízení a hledáte, který z deseti COM portů patří vaší desce, a pak doufáte, že flashovací utilitu dělal někdo, kdo myslel na to, že ne všechny porty jsou `/dev/něco`…

Nejde o to, že by to nešlo vyřešit. Všechno to jde vyřešit. Jde o to, že *řešíte problémy, které nemusíte mít*.

A co WSL?

Tady bych chtěl být upřímný. WSL (Windows Subsystem for Linux) je skvělý nápad. Celý Linux běžící uvnitř Windows, přístupný z terminálu, integrovaný s VS Code. Na papíře ideální řešení. A pro běžný vývoj vlastně taky. Ale při vývoji pro mikrokontroléry v praxi narazíte na háčky.

První je USB. Když připojíte vývojovou desku k počítači, Windows ji vidí. Linux ve WSL ji nevidí. Musíte nainstalovat `usbipd-win`, ručně předat zařízení z Windows do WSL příkazem `usbipd bind` a `usbipd attach` (jedno z toho musíte navíc volat jako administrátor) a doufat, že se vám po odpojení a připojení kabelu celý postup nezboří. Někdy se zboří. A vy místo ladění firmware hledáte, proč WSL nevidí vaši desku.

Druhým háčkem je souborový systém. Soubory uvnitř WSL (`/home/vase-jmeno/`) jsou rychlé. Soubory na disku Windows (`/mnt/c/Users/...`) jsou pomalé. Jak moc? Překlad většího projektu v ESP-IDF může trvat dvakrát nebo třikrát déle, když máte zdrojáky v `/mnt/c/`. Řešení existuje – prostě si projekt držte uvnitř WSL. Ale to znamená, že vaše soubory jsou v linuxovém souborovém systému, ke kterému se z Windows dostanete jen přes `\\wsl$\` a některé nástroje pro Windows s tím mají problém.

A třetím je prostě složitost celého řešení. Máte Windows, ve Windows máte WSL, ve WSL máte Ubuntu, v Ubuntu máte toolchain, z Windows předáváte USB do WSL, z VS Code se připojujete do WSL přes Remote extension. Funguje to. Ale když to nefunguje, tak hledáte problém v pěti vrstvách najednou, a nikdo přesně neví, jestli za chybu můžou Windows, WSL, usbipd, kernel WSL, nebo váš toolchain.

Moje doporučení je jednoduché a pragmatické. Nemusíte přeinstalovat počítač. Nemusíte se stát *„linuxákem“*. Ale **pro práci s nástroji, o kterých je tato kniha, prostě potřebujete linuxové prostředí**.

Jak říká klasik: „Se s tím smiř!“

Máte v zásadě tři cesty. WSL2 je ta nejrychlejší. Je to kompromis, má své mouchy, ale většina věcí funguje (i když třeba ne pohodlně) a nepotřebujete k tomu nic navíc. Pokud máte Windows 10 nebo novější, je to otázka jednoho příkazu. Spolehlivější variantou je virtuální stroj, třeba VirtualBox nebo VMware. Plnohodnotný Linux, žádné překvapení se souborovým systémem, USB se předávají přímo. Zabere víc místa na disku a trochu víc RAM, ale jinak funguje bez překvapení.

A pak je tu samozřejmě nativní Linux, buď jako druhý systém na disku, nebo na jiném počítači. Pokud vás to láká, je to nejčistší řešení. Ale rozhodně to není nutné.

Prošel jsem si všechny. Nakonec jsem si koupil malé PC, kde mám Linux, běží mi tam Docker s domácí automatizací, a protože se většinu času fláká a čeká, až se něco stane, tak jeho výkon používám na vývoj a překlad. Sedím u svého Windowsího počítače, kde si vyvíjím ve VS Code. Když dělám weby, mám vše, co potřebuji. Když jdu níž, k hardware nebo do C/C++, tak se ve VS Code připojím k vývojovému serveru, přepnu si prostředí (mám několik oddělených, jedno pro ESP, jedno pro Raspberry Pi Pico, jedno pro CH32, …) a vyvíjím vzdáleně. Mám pohodlí svého VS Code, vyladěného podle svých potřeb, a přitom překládám na Linuxu.

Celá tato kniha je psaná tak, aby příklady fungovaly v kterékoli z těchto variant. Příkazy jsou příkazy, cesty jsou cesty, nástroje jsou nástroje. Jestli pod tím běží WSL nebo Ubuntu na starém ThinkPadu, je jedno.

Neříkám vám „zahoďte Windows“. Sám je nezahazuji, protože mám spoustu dobrých důvodů, proč u nich zůstat. Říkám vám „připravte si linuxový kout, ve kterém budete pracovat“.

1.4 Od Arduino IDE k PlatformIO… a dál

Ale ještě na chvíli zůstaneme ve vašem domácím prostředí. Jak Arduino, tak PlatformIO existují pro všechny hlavní platformy.

1.4.1 Co před vámi Arduino tají

Arduino jistě znáte. Je skvělé, snížilo vstupní práh na velmi příjemné minimum a vydržíte s ním dlouho. I profesionálové ho používají, protože je to často nejrychlejší způsob, jak si osahat nový hardware.

V této kapitole ale nahlédneme do zákulisí Arduina a podíváme se na věci, co vás zbrzdí. Jak se píše na sociálních sítích: *Toto Arduino nechce, abyste věděli! Sdílejte, než to smažou!*

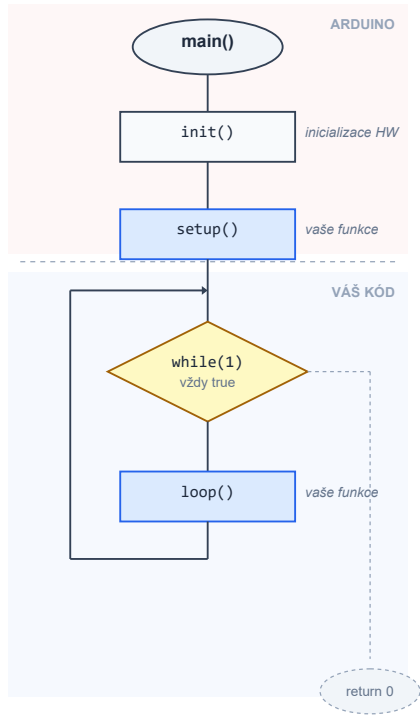
Pokud jste dosud programovali v Arduino IDE, měli jste pohodlí. Napsali jste `setup()`, napsali jste `loop()`, klikli na šipku a hotovo. Všechno fungovalo. Jenže teď jste otevřeli příklad z ESP-IDF nebo pico-sdk a koukáte na kód, který vypadá úplně jinak. Kde je `setup()`? Co je to `main()`? Proč tu nejsou žádné třídy? Co jsou ty hvězdičky za typem? A kam se poděl `Serial.println()`?

Nebojte se. Není to proto, že by SDK bylo složitější. Je to proto, že Arduino před vámi hodně věcí schovává.

V Arduinu napíšete toto.

```
void setup() {
  pinMode(13, OUTPUT);
}


void loop() {
  digitalWrite(13, HIGH);
  delay(500);
  digitalWrite(13, LOW);
  delay(500);
}
```



Možná si myslíte, že `setup()` a `loop()` jsou nějaké speciální věci v programování. Nejsou. Jsou to obyčejné funkce, které Arduino volá z hlavní funkce každého céčkového programu, z `main()`. Ten `main()` existuje, jen ho nevidíte. Schválně, tady je (zjednodušeně):

```
int main() {
  init();           // inicializace hardware
  setup();          // vaše funkce setup()
  while (1) {
    loop();         // vaše funkce loop(), dokola
  }
  return 0;         // sem se nikdy nedostane
}
```

Arduino IDE tento `main()` přidá za vás. Je to hezké gesto – začátečník nepotřebuje vědět, že existuje. Jenže ve chvíli, kdy přejdete na SDK, musíte `main()` napsat sami.

V pico-sdk to vypadá takto:

```
#include "pico/stdlib.h"
```

```
int main() {
    stdio_init_all();


    gpio_init(25);
    gpio_set_dir(25, GPIO_OUT);


    while (true) {
        gpio_put(25, 1);
        sleep_ms(500);
        gpio_put(25, 0);
        sleep_ms(500);
    }


    return 0;
}
```

A v ESP-IDF (kde se vstupní funkce jmenuje `app_main`) zase takto:

```
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"


void app_main(void) {
    gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);


    while (1) {
        gpio_set_level(GPIO_NUM_2, 1);
        vTaskDelay(pdMS_TO_TICKS(500));
        gpio_set_level(GPIO_NUM_2, 0);
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```