# Formální metody návrhu software aplikované na embedded systémy
## Platformně nezávislý zdrojový kód (1)

**Michal Bližňák, Dušan Kolář**

### Úvodem

Pod pojmem embedded systémy (často označované také jako vestavěné, či vestavné systémy), rozumíme systémy s integrovaným výpočetním subsystémem a příslušnými I/O periferiemi. Lze je v současné době nalézt v širokém spektru spotřební elektroniky či průmyslových zařízeních; ať už se jedná o digitální kamery, audio a DVD přehrávače, zdravotnické diagnostické přístroje, řídící a monitorovací průmyslové technologie, či řídící systémy v osobní dopravě, letectví, či vojenské technice [1].

Navzdory tomuto, relativně širokému, rozšíření embedded systémů je jejich programování i v současné době stále mnohem méně komfortní a efektivní, než je tomu např. u programování softwarových aplikací určených pro běžné desktop počítače. To je způsobeno především faktem, že programátor embedded systémů musí zvládnout nejen samotný programovací jazyk, ale musí se také podobně seznámit s hardwarovou strukturou a možnostmi cílového systému pro který je aplikace vytvářena a jím vytvářený produkt musí efektivně využívat všech prostředků, které mu daná HW platforma poskytuje [1]. Je nutno podotknout, že systémové prostředky embedded systémů jsou ve většině případů značně omezené, zejména pak výkon MCU (Micro Controller Unit; což je obdoba CPU) či velikost volné paměti, a proto je žádoucí vytvářet optimalizované aplikace splňující specifická omezení a podmínky dané cílovou platformou [2].

Aby bylo možno dosáhnout co nejvyšší efektivity využití HW prostředků cílové platformy, je často nutné používat pouze striktně vymezené programovací jazyky (často jen Assembler a ANSI C/C++) a technologie, umožňující přímé řízení interních a externích periferií MCU. Je zřejmé, že tento způsob programování je časově náročný a náchylný na vznik programových chyb. Navíc, jak vyplývá z prostého faktu, že vytvářená aplikace využívá přímo zařízení nabízené cílovou platformou, bude tato aplikace s velkou pravděpodobností nepřenosná na jiné embedded systémy, a to i v případě, že se tyto systémy budou lišit pouze minimálně.

Otázka tedy zní, jakým způsobem by bylo možné zrychlit a zefektivnit vývojový proces aplikací určených (nejen) pro embedded systémy?

Jedena z možných cest spočívá ve využití některé z dostupných formálních metod návrhu/vývoje softwarových aplikací. Tyto moderní vývojové metody zaručují rychlý, intuitivní a bezpečný (z hlediska minimalizace chybovosti zdrojového kódu) návrh aplikací a co je důležité, formální popis aplikační logiky je platformně nezávislý. V současné době jsou tyto, nebo podobné, techniky návrhu SW implementovány v mnoha integrovaných vývojových prostředích podporujících zrychlený vývojový proces aplikací (RAD IDE – Rapid Application Development Integrated Development Environment) pro klasické počítače, ve vývojových prostředích určených pro embedded systémy však prozatím většinou chybí.

Je však potřeba podotknout, že ne všechny techniky formálního návrhu aplikací jsou vhodné pro použití u embedded systémů. Je tomu tak zejména z důvodu již zmiňovaných nároků na optimalizaci zdrojového kódu aplikace a to zejména z hlediska objemu výsledného kódu a efektivního a šetrného nakládání s dostupnými systémovými prostředky. V tomto článku se zaměříme zejména ta techniky využitelné společně s procedurálně orientovanými programovacími jazyky, jakými jsou např ANSI C a Pascal. Za tohoto předpokladu lze pro popis aplikační logiky vytvářeného programu použít obecně dobře známých konečných stavových automatů/diagramů (Finite State Machines/Charts – FSM), z jejichž struktury lze za předpokladu použití vhodných nástrojů generovat zdrojový kód vytvářené aplikace. Jak si ukážeme dále, tento aplikační kód může být zapsán pomocí různých programovacích jazyků a navíc lze v rámci zvoleného výstupního jazyka použít rozličné generující algoritmy ovlivňující specifické vlastnosti výsledného zdrojového kódu.

Využití FSM tedy zajišťuje nejen přehledný způsob vývoje programu a jeho deterministické chování, ale navíc umožňuje aplikaci rozličných ověřujících a optimalizačních algoritmů, které mohou dále zkvalitňovat generovaný kód a potažmo i výslednou softwarovou aplikaci.

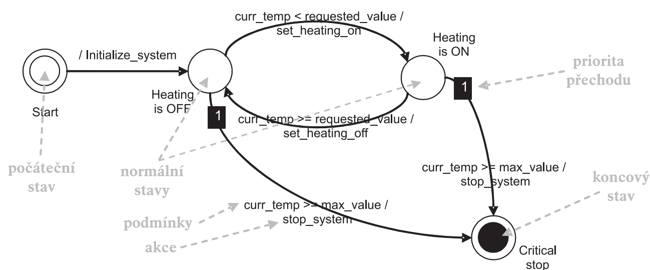
### Co je to FSM?

Je možné, že ne všichni čtenáři jsou alespoň zběžně seznámeni s problematikou stavových automatů, a proto se zde krátce zmíníme o základních vlastnostech tohoto formálního popisu systému a o jeho spojitosti s aplikační logikou.

Stavový automat (FSM) je, z našeho pohledu, model chování skládající se ze tří základních stavebních prvků: stavů systému, přechodů mezi těmito stavy a akcemi, které se provádějí, v době přechodu z jednoho stavu systému do stavu jiného. Stavy v podstatě uchovávají informaci o celé minulosti systému, tzn. reflektují jakými změnami systém prošel od počátku jeho činnosti do současnosti. Přechody indikují změnu jednotlivých stavů a jsou popsány podmínkami, které je nutné splnit, aby k danému přechodu došlo. Akce je pak činnost prováděná v době přechodu mezi stavy, případně, při vstupu do určitého stavu, nebo při jeho opuštění [3], [4].

Obecně rozlišujeme dva základní typy stavových automatů: rozpoznávající (Acceptors/Recognizers), jejichž výstupem je binární informace říkající, zda je daná posloupnost vstupů přijata daným automatem a existuje na ni adekvátní odpověď a převodové (Transducers), jejichž výstup závisí na přijatém vstupu a aktuálním stavu systému. Zde navíc rozlišujeme mezi tzv. Mealyho a Mooreovými stavovými automaty (výstup Mooreova stavového automatu závisí pouze na aktuálním stavu, kdežto výstup Mealyho automatu závisí na aktuálním stavu a aktuálním vstupu).

Dalším možným dělením stavových automatů by mohlo být členění na deterministické a nedeterministické. U deterministických stavových automatů existuje pro jednu uspořádanou dvojici stav-vstup vždy pouze jeden možný přechod, u nedeterministických automatů může existovat více možných přechodů z daného stavu za podmínky jednoho konkrétního vstupu.




**Obr.1 Jedna z možných grafických reprezentací FSM**

Obr. 1 ilustruje jednu z možných grafických reprezentací stavového automatu, tzv. stavový diagram (state chart). Obrázek představuje popis aplikační logiky programu, který zajišťuje dvoupolohovou regulaci tepelného systému s ověřováním překročení kritické hodnoty teploty.

Význam použitých symbolů a celého diagramu je následující: Vstupní bod algoritmu (programu) je definován tzv. počátečním stavem (initial state) z něhož program bezprostředně po své spuštění přechází do stavu „Heating is OFF“. Přechod mezi těmito dvěma stavy není ničím podmíněn (proběhne bezprostředně po startu programu) a vyvolá akci s názvem „initialize_system“. Přechod mezi stavy „Heating is OFF“ a „Heating is ON“ je podmíněn poklesem aktuální teploty pod definovanou žádanou hodnotu a při jeho provedení je vyvolána akce s názvem „set_heating_on“, jejímž výsledkem je aktivace topného akčního

| Podminky / Aktuální stav | N/A | Curr_Temp >= Max_Val | Curr_Temp < Requested_Val | Curr_Temp >= Requested_Val |
|---|---|---|---|---|
| Start | Heating_is_OFF | N/A | N/A | N/A |
| Heating_is_OFF | N/A | Critical_Stop | Heating_is_ON | Heating_is_OFF |
| Heating_is_ON | N/A | Critical_Stop | Heating_is_OFF | Heating_is_ON |
| Critical_Stop | N/A | N/A | N/A | N/A |

**Obr.2 Přechodová tabulka**

```
TYPE_STATE Regulator(void)
{
  TYPE_STATE state=ID_Start;
  for(;;)
  {
    /* Main loop */
    switch( state )
    {
      /* State: Start */
      case ID_Start:
        initialize_system();
        state=ID_Heating_is_OFF;
      /* State: Heating is OFF */
      case ID_Heating_is_OFF:
        if( curr_temp >= MAX_VAL )
        {
          set_heating_off();
          state=ID_Critical_stop;
        }
        else if( curr_temp < requested_value )
        {
          set_heating_on();
          state=ID_Heating_is_ON;
        }
        break;
      /* State: Critical stop */
      case ID_Critical_stop:
        return ID_Critical_stop;
      /* State: Heating is ON */
      case ID_Heating_is_ON:
        if( curr_temp >= MAX_VAL )
        {
          set_heating_off();
          state=ID_Critical_stop;
        }
        else if( curr_temp >= requested_value )
        {
          set_heating_off();
          state=ID_Heating_is_OFF;
        }
        break;
    }
  }
}
```

**Obr.3**

členu. Analogicky, přechod mezi stavy „Heating_is ON“ a „Heating_is_OFF“ je strážen podmínkou testujících překročení žádané hodnoty řízené teploty a jeho akcí je vypnutí topného akčního členu. Kromě dvou výše zmiňovaných stavů realizujících vlastní dvoupolohový regulační algoritmus pak popisovaná aplikace obsahuje také speciální koncový stav „Critical stop“, který může být dosažen tehdy, překročí-li aktuální hodnota řízené tepelné veličiny předem definovanou kritickou hranici. Vstup do tohoto koncového stavu je zajištěn dvěma přechody vedoucími z obou standardních stavů programu. U těchto přechodů si povšimněte zejména definovaných priorit které zajišťují, že test překročení kritické hodnoty je proveden přednostně před porovnáním žádané a aktuální hodnoty regulované veličiny.

Struktura stavového automatu může být popsána nejen pomocí stavového diagramu, ale také pomocí tzv. přechodové tabulky (obr. 2).

V tomto článku se budeme dále věnovat pouze deterministickým Mealyho stavovým automatům, protože právě tyto jsou schopny nejlépe popisovat chování většiny základních typů softwarových aplikací. Jaký je tedy vztah mezi aplikační logikou a stavovými automaty?

## Stavové automaty vs. aplikační logika

Každý stav, ve kterém se může popisovaná aplikace nalézat, lze definovat jako jeden stav obsažený ve stavovém automatu. Přechody mezi těmito stavy lze realizovat pomocí klasického řízení toku programu s využitím vhodných programových příkazů (if, switch, goto, …), kde testované logické výrazy jsou implementacemi podmínek strážících jednotlivé přechody stavového automatu. Akce spojené s těmito přechody pak mohou být realizovány programovým kódem volaným za předpokladu splnění definované podmínky.

Je zřejmé, že tímto způsobem lze popisovat aplikační logiku na všech úrovních vytvářeného programu – jak chování hlavní funkce programu, tak i dalších dílčích funkcí a procedur. Pro ilustraci zde uveďme zdrojový kód v programovacím jazyce ANSI C, který by odpovídal struktuře stavového automatu uvedené na obr. 1.

## Literatura


[1] QING, L.: Real-Time Concepts for Embedded Systems, CMPBooks, 2003, ISBN 1-57820-124-1

[2] BARR, M.: Programming Embedded Systems, O'Reilly, 1999, ISBN: 1-56592-354-5

[3] CARROLL, J., LONG, D.: Theory of Finite Automata with an Introduction to Formal Languages, Prentice Hall, Englewood Cliffs, 1989.

[4] HOPCROFT, J. E., ULLMAN, J. D.: Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[5] ČERNÝ, S., KOLÁŘ, D., STRUŽKA, P.: Processor Expert, Component Application Builder for Embedded Systems, In: Proceedings of 5th IEEE Design and Diagnostics of Electronics Circuits and Systems Workshop, Brno, CZ, FIT VUT, 2002, s. 393-397, ISBN 80-214-2094-4


*Pokračovanie v budúcom čísle.*


**Ing. Michal Bližňák**
**doc. Dr. Ing. Dušan Kolář**

**Univerzita Tomáše Bati ve Zlíně, ČR**
**Fakulta aplikované informatiky**
**Ústav Aplikované Informatiky**
**e-mail: bliznak@fai.utb.cz**