Vážení zákazníci,

dovolujeme si Vás upozornit, že na tuto ukázku knihy se vztahují autorská práva, tzv. copyright.

To znamená, že ukázka má sloužit výhradně pro osobní potřebu potenciálního kupujícího (aby čtenář viděl, jakým způsobem je titul zpracován a mohl se také podle tohoto, jako jednoho z parametrů, rozhodnout, zda titul koupí či ne).

Z toho vyplývá, že není dovoleno tuto ukázku jakýmkoliv způsobem dále šířit, veřejně či neveřejně např. umisťováním na datová média, na jiné internetové stránky (ani prostřednictvím odkazů) apod.

redakce nakladatelství BEN – technická literatura
redakce@ben.cz

4

LADĚNÍ APLIKACÍ

Mikroprocesory ATMEL řady AT90 nemají integrované prostředky pro ladění vytvářeného programu, které by umožňovaly pozastavit běh programu na zvolené adrese a kontrolovat nebo modifikovat obsah vnitřní paměti. (V době vydání knihy byl k dispozici pouze jediný typ AT90S323, který má integrován JTAG interface. Princip a další informace jsou uvedeny v závěru této kapitoly.)

Pro základní ladění aplikace se používají různé simulační programy, např. volně dostupné StudioAVR. Simulátory umožňují především ověření algoritmů aplikace a běh těch částí programu, které nejsou časově kritické. Po ověření základní funkce se aplikace naprogramuje a ověří se chování v reálných podmínkách. I v této fázi vývoje je žádoucí co největší množství informací o běhu programu, hodnotách proměnných v registrech a v paměti. Tento úkol lze řešit prostřednictvím monitoru. Monitor je v podstatě část programového kódu, která se připojuje k laděné aplikaci a umožňuje přenést požadované údaje do hostitelského počítače. Získaná data jsou interpretována uživateli na hostitelském počítači v co nejsrozumitelnější formě. V následující části kapitoly budou podrobně rozebrány dva monitory. Jeden poměrně obsáhlý monitor, napsaný z větší části v programu C, druhý jednodušší s omezenými službami, ale také kratší délkou kódu. První typ monitoru lze libovolně používat v aplikaci v návaznosti na program hostitelského počítače „DebuggerAVR“.

4.1 PRINCIP MONITORU

Úkolem monitoru je přijímání příkazů z hostitelského počítače, zpracování a odeslání požadované odezvy zpět do hostitelského počítače. Monitor tedy vyžaduje komunikační kanál a zpravidla ještě další vstupně/výstupní signály mikropočítače.

Jako komunikační kanál byl v základní implementaci použit sériový kanál 19 200 Bd, 8 bitů bez parity, 1 start a 1 stop bit. Program „DebuggerAVR“ má implementovány některé služby umožňující přijmout a v okně zobrazit libovolný

text. Tato služba zmenšuje omezení, pokud využívá sériový kanál i vlastní aplikace. Existují ale aplikace, které potřebují sériový kanál „čistý“. V takovém případě je třeba komunikační kanál realizovat jiným způsobem. Je možné použít sériový kanál SPI nebo simulovat přenos dat přes některé I/O signály mikroprocesoru. Lze také použít další sériový obvod. Všechna tato řešení vyžadují doplňující hardware, který je popsán v kapitole *Doplňky monitoru*.

4.2 KOMUNIKAČNÍ PROTOKOL

Mezi monitorem a hostitelským počítačem je definován komunikační protokol. Vstupní (příkazové) pakety mají pevnou délku 8 bytů. Záhlaví tvoří řídicí znak „$“ a písmeno, poté následuje 6 bytů parametru. Písmeno určuje typ příkazu, obsah parametrů závisí na typu příkazu. Výstupní pakety mají proměnnou délku, záhlaví tvoří opět kombinace „$“ a písmeno, poté následuje soubor dat a kombinace <CR><LF>.

4.3 ZÁKLADNÍ FUNKCE

Protože je monitor v podstatě část programu, bývá tendence služby rozšiřovat a získat co nejvíce informací. V případě podstatného omezení paměťového prostoru mikroprocesoru je třeba nalézt optimální řešení. Jako výchozí procesor pro použití popisovaného monitoru byl stanoven AT90S8515, tedy 8 kB programové paměti a přes 200 bytů paměti dat. Většina služeb monitoru je ve zdrojových souborech podmíněně přeložitelná, což vede k úspoře kódu. Nejmenší délka kódu monitoru je zhruba 500 bytů, největší asi 1,1 kB. Monitor poskytuje následující soubor služeb:

1. Skenování vnitřní paměti mikroprocesoru za běhu programu aplikace.
2. Zastavení běhu programu a jeho opětovné spuštění.
3. Reset.
4. Modifikaci obsahu registrů a vnitřní i vnější paměti RAM mikroprocesoru.
5. Plný běh programu do zvoleného místa v programu /*UserBreak*, *InterruptBreak*/.
6. Krokování programu po instrukcích.
7. Kontinuální krokování programu až do stanoveného místa */Break/*.
8. Příkazem zadat a libovolně měnit místo zastavení v programu.

V další části budou jednotlivé funkce a jejich realizace podrobně rozebrány přímo na zdrojovém kódu. Čtenář tak bude mít možnost v případě potřeby libovolně upravit monitor tak, aby co nejlépe vyhověl jeho potřebám.

4.3.1 Realizace funkcí monitoru

Realizace uvedených funkcí monitoru spočívá v naprogramování potřebných podprogramů (jsou obsahem modulu *debugger.c*) a v jejich aktivování. Z uvedených požadavků vyplývá, že monitor musí aktivovat jednak hostitelský počítač (např. požadavek uživatele na zastavení běhu programu), jednak sám kód aplikace (např. zastavení programu na stanovené adrese). Oba typy aktivace jsou začleněny do přerušovacího systému mikroprocesoru. Počítač PC spouští monitor aktivitou na komunikačním kanálu (v základní koncepci tedy na sériovém kanálu RS232), přičemž obsluha sériového kanálu je prováděna v rámci přerušení. Aktivuje-li monitor přímo aplikace, provádí to v rámci některého např. vnějšího přerušení INT0 nebo INT1 eventuálně v rámci přerušení od časovače. Tata variabilita umožňuje uživateli zvolit zdroj, který bude co nejméně omezovat systémové požadavky aplikace. Začlenění obsluhy monitoru pod jednotlivá přerušení je obsaženo v souboru *dbg_gcrt.c*. Základní je funkce *ServiceInterruptPending*, která je volána v rámci přerušení podle nastavení konfiguračních podmínek podmíněného překladu. Detailní využití je uvedeno při popisu realizace jednotlivých funkcí monitoru.

```
void  ServicePendingInterrupt(void)
{
     asm
     (
       "sts ContextDebuggerRegister,r31\n"
     );

#if  T0_INTERRUPT_PENDING
     asm
     (
       "ldi    r31,0x00\n"
       "out    0x33,r31\n"
     );
#endif

     asm
     (
```

```
    "in    r31,__SP_L__\n"
    "sts (ContextDebuggerRegister + 1),r31\n"
    "in    r31,__SP_H__\n"
    "sts (ContextDebuggerRegister + 2),r31\n"
    "in    r31,__SREG__\n"
    "sts (ContextDebuggerRegister + 3),r31\n"

    "ldi   r31,lo8(DebuggerStack + 79)\n"
    "out __SP_L__,r31\n"
    "ldi   r31,hi8(DebuggerStack + 79)\n"
    "out __SP_H__,r31\n"
);

asm
(
    "push  r0\n"
    "push  r1\n"
    ……..
    ……..
    "push  r29\n"
    "push  r30\n"
);

ScanDebuggerCommandPacket();
```

Monitor pracuje s vlastním zásobníkem (symbolická proměnná *DebuggerStack*), na který ukládá při přerušení obsah registrů, dále požaduje prostor v paměti SRAM, na který ukládá návratovou adresu a obsah stavového registru (symbolická proměnná *ContextDebuggerRegister*). Je třeba si uvědomit, že právě návratová adresa přerušení je zároveň aktuální adresou vykonávané aplikace. Velikost zásobníku monitoru určuje uživatel nastavením konstanty *nDeepDebuggerStack.* Velikost zásobníku, která je závislá na použitém překladači, je odesílána v rámci základního informačního paketu do hostitelského počítače. Celá činnost monitoru je řízena proměnnou *DebuggerStatus*. Jednotlivé bity mají následující význam:

```
#defineINTERRUPTBREAK_STATUS     0x08
```

aplikační program se zastavil na místě makra *InterruptBreak*

```
#defineUSERBREAK_STATUS          0x10
```

aplikační program se zastavil na místě makra *UserBreak*

```
#defineGOTOBREAK_STATUS          0x20
```

monitor zajišťuje kontinuální krokování instrukce po instrukci až do dosažení nastaveného bodu přerušení

```
#defineSTEP_STATUS          0x40
```

monitor zajišťuje krokování po instrukci v rámci základní úrovně

```
#defineORDER_STATUS               0x80
```

monitor setrvává ve smyčce v přerušení až do přijetí dalšího příkazu.

Skenování pamětí mikroprocesoru

Nejprve bude rozebrán mechanizmus nejjednodušší funkce monitoru. Příkazem „*$S*“, který odešle hostitelský počítač, se aktivuje požadavek na odeslání obsahu vnitřní paměti mikroprocesoru, tedy registrů a paměti SRAM. Monitor po indikaci tohoto příkazu funkcí

```
  ServiceScanSramProcessor((unsigned char *)0x00, (unsigned
int)SRAM_SIZE, 'S')
```

odešle obsah vnitřní paměti do sériového kanálu. Poté se vrátí z přerušovací rutiny a následuje provádění kódu aplikace. Pro skenování obsahu externí paměti RAM je implementován příkaz „*$M<adresa>*“, který aktivuje odeslání 256 bytů externí paměti od zvolené adresy. Pro skenování obsahu paměti EEPROM obsahuje monitor příkaz „*$F*“, který aktivuje odeslání celého obsahu této paměti. Je třeba si uvědomit, že přenos dat na nejnižší přenosové rychlosti podporované hostitelským programem „DebuggerAVR“ trvá zhruba 1 s. Po tuto dobu je systém v přerušení, což ovlivňuje chování celé aplikace.

Zastavení běhu programu a jeho opětovné spuštění

Zastavení provádění programu aplikace a kontrola proměnných je v podstatě nejvíce používaná funkce každého monitoru.

```
if(DebuggerPacket.Data[1] == 'P')
{
     DebuggerStatus = DebuggerStatus | ORDER_STATUS;
```

```
    ServiceScanSramProcessor((unsigned char *)0x00, (unsigned
int)SRAM_SIZE, 'S');
}
```

Příkazem „*$P*“ se aktivuje odeslání obsahu vnitřní paměti mikroprocesoru a stavová proměnná se nastaví do režimu smyčky v rámci přerušovací rutiny. V této smyčce monitor přijímá všechny implementované příkazy kontinuálním testováním sériového kanálu.

```
    if((DebuggerStatus & (STEP_STATUS | ORDER_STATUS)))
  {
    DebuggerStatus = DebuggerStatus | ORDER_STATUS;
    do
    {
#if (PORT_COMPILER == GCC_AVR)
          while(!(inp(USR) & 0x80));
          DebuggerPacket.Data[DebuggerPacket.nIndex++] = inp(UDR);
          if(DebuggerPacket.Data[0] != '$') DebuggerPacket.nIndex
= 0;
#endif
#if (PORT_COMPILER == ICC_AVR)
          while(!(USR & 0x80));
          DebuggerPacket.Data[DebuggerPacket.nIndex++] = UDR;
          if(DebuggerPacket.Data[0] != '$') DebuggerPacket.nIndex
= 0;
#endif
          if(DebuggerPacket.nIndex >= nMAX_LEN_DEBUGGER_PACKET)
          {
            ParseDebuggerComandPacket();
          }
    } while((DebuggerStatus & ORDER_STATUS));
    }
```

Je možné modifikovat obsah registrů a vnitřní nebo vnější paměť RAM. Ve smyčce monitor setrvá až do přijetí příkazu *Go* nebo *Go_to_Break*, kterým se stavová proměnná *StatusBreak* přestaví.

Reset

Restart aplikace bývá při ladění požadován velmi často. Je možné provést buď hardwarový reset generovaný doplňujícím obvodem připojeným na sériovou linku hostitelského počítače (viz kapitolu *Doplňky*). Příkazu „*$R*“ se realizuje softwarový reset.

```
#if ENABLE_SWRESET
          if(DebuggerPacket.Data[1] == 'R')
          {
             asm
             (
              "ldi  r30, 0x00\n"
              "ldi  r31, 0x00\n"
              "ijmp\n"
             );

          }
#endif
```

Softwarový reset neprovede nastavení systémových registrů a registrů periferií do výchozí hodnoty.

Modifikaci obsahu paměti SRAM mikroprocesoru

Příkazem <*$W adresa, data* > se modifikuje obsah registrů nebo paměti SRAM mikroprocesoru. Parametr adresa určuje místo v paměti SRAM nebo registr, data požadovaný obsah.

Pokud se provádí modifikace obsahu registrů, provádí se nad obsahem buď proměnné *ContextDebuggerRegister[0]*, jedná-li se o registr *r31*, nebo nad zásobníkem monitoru *DebuggerStack*, jedná-li se o obsah ostatních registrů.

Při modifikaci paměti SRAM je třeba brát zřetel na systémové proměnné monitoru a v nadřízeném systému přijmout taková opatření aby tyto obsahy nebylo možné přepsat.

Plný běh programu do zvoleného místa v programu

Při ladění aplikace lze nalézt určitá klíčová místa, kde je třeba ověřit obsah proměnných, přičemž je žádoucí aby až do jejich dosažení běžel program plnou

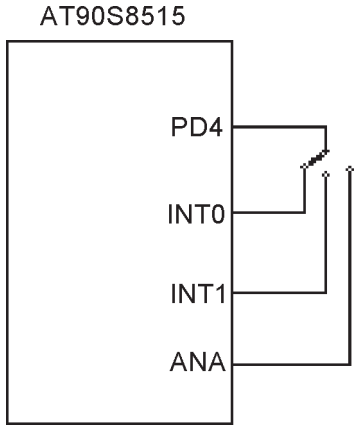
rychlostí. V monitoru je tento požadavek realizován prostřednictvím makra *UserBreak*. Makro vkládá do aplikačního programu na požadované místo programátor aplikace. Makro je definováno v souboru *debugger.h* a je závislé na typu zvoleného přerušení, které je jeho zpracování přiřazeno.

```
#if(INT0_INTERRUPT_PENDING | INT1_INTERRUPT_PENDING |
ACOMP_INTERRUPT_PENDING)

#define UserBreak { DebuggerStatus = DebuggerStatus |
USERBREAK_STATUS; cbi(PORTD, PD4);
         while(DebuggerStatus & USERBREAK_STATUS);}

#else
#define UserBreak { DebuggerStatus = DebuggerStatus |
USERBREAK_STATUS;
         while(DebuggerStatus & USERBREAK_STATUS);}
#endif
```

Makro představuje smyčku, ve které program setrvá až do přerušení. Pokud se zvolí zpracování v rámci vnějšího přerušení, musí být toto přerušení v makru aktivováno. To provádí instrukce *cbi(PORTD, PD4).* Výstup 4 portu D je zvolen jako generátor vnějšího přerušení. Výstup se přivede na vstup zvoleného vnějšího přerušení podle *obr. 4.1*.



Obr. 4.1

Nulováním tohoto výstupu se generuje požadavek přerušení. Další zpracování je již známé, v rámci přerušení po uložení kontextu a registrů se volá obslužná funkce *ServiceInterruptPending.* Ta informuje nadřízený systém o dosažení bodu

UserBreak odesláním paketu „*$U*“. Vnější přerušení může být generováno i jiným signálem, tak aby byla laděná aplikace co nejméně omezena. Uživatel musí příslušným způsobem upravit zdrojové soubory.

Lze také zvolit zpracování v rámci přerušení od časovače 0, v tomto případě setrvává aplikace ve smyčce makra až do přetečení časovače. Poté je generováno přerušení. V uvedeném příkladě „Hello, World“ se makro *UserBreak* vkládá na začátku programu aplikace. Toto použití umožňuje po restartu mikroprocesoru zastavit běh programu na začátku běhu aplikace a zkontrolovat veškeré inicializační hodnoty. Z praktického hlediska je vhodné používat makro *UserBreak* s podmíněným překladem.

Krokování programu po instrukcích

Krokování programu po instrukcích využívá vlastnosti mikroprocesoru „po návratu z přerušení provést alespoň jednu instrukci v základní úrovni před dalším přerušením“. Znamená to tedy, že po každé vykonané instrukci přechází aplikační program do obsluhy přerušení a spouští funkci *ServiceInterruptPending*. Do režimu krokování se monitor uvede nejprve zastavením běhu aplikace příkazem „*$P*“ následovaným příkazem „*$S*“. Tím se status monitoru nastaví do režimu STEP_STATUS a uvolní pozastavovací smyčku a před návratem z obsluhy přerušení nastaví buď zdroj vnějšího přerušení nebo přerušení od časovače.

```
if(DebuggerStatus & (STEP_STATUS | GOTOBREAK_STATUS))
{
#if T0_INTERRUPT_PENDING
#if(PORT_COMPILER == GCC_AVR)
          outp(0xfe, TCNT0);
#endif
#endif
#if (INT0_INTERRUPT_PENDING | INT1_INTERRUPT_PENDING)
#if(PORT_COMPILER == GCC_AVR)
     cbi(PORTD, PD4);
#endif
#endif
}
```

Interval časovače nastavuje tak, aby již po instrukci *reti* bylo přerušení od časovače aktivováno. Po instrukci *reti* se provede jedna instrukce v základní úrovni programu a opět se přejde do pozastavovacího režimu. Pouze tehdy, je-li

po provedení instrukce *reti* aktivováno jiné přerušení s vyšší prioritou, provede se celá obslužná funkce tohoto přerušení a po jejím ukončení se opět aktivuje monitor.

Kontinuální krokování programu až do stanoveného místa

Při ladění je třeba mít možnost zadat libovolné místo v paměti programu na kterém se vykonávání aplikace pozastaví a aktivuje se činnost monitoru. Lze nastavit až čtyři místa zastavení v programové paměti. Nastavení všech čtyř bodů zastavení se provádí jediným příkazem „*$B<br1><br2><br3><br4>*“, kde *br1..br4* obsahují adresu bodu zastavení. Pokud nemá být některý bod zastavení aktivní, musí být jeho obsah nulový. Body zastavení ukládá monitor do proměnné *BreakPoint*. Příkazem „*$C*“ se všechny body zastavení ruší.

Příkazem „*$I*“ se monitor uvede do stavu *GOTOBREAK_STATUS*. V tomto režimu mikroprocesor krokuje po jedné instrukci až do doby, kdy se shoduje obsah čítače programu s některou položkou ze seznamu *BreakPoint*.

```
#if ENABLE_BREAKPOINT
  if(DebuggerStatus == GOTOBREAK_STATUS)
  {
#if FOUR_BREAKPOINT
  if(PCBrk.PCBreak == BreakPoint[0] ||
    PCBrk.PCBreak == BreakPoint[1] ||
    PCBrk.PCBreak == BreakPoint[2] ||
    PCBrk.PCBreak == BreakPoint[3])
#else
  if(PCBrk.PCBreak == BreakPoint[0])
#endif
  {
    ServiceScanSramProcessor((unsigned char *)0x00, (unsigned
    int)SRAM_SIZE, 'R');
    DebuggerStatus = ORDER_STATUS;
  }
```

Při shodě se informuje hostitelský počítač odesláním paketu „*$R*“. Z důvodu úspory kódu je možné volit testování na všechny čtyři body zastavení, nebo pouze na jeden.