ELEKTRONIK TIDNINGEN

Anders Lundgren och Lotta Frimanson
Product Marketing, IAR Systems

Energidebugga din inbyggda programvara

Hitta strömtjuvarna och optimera energiförbrukningen i ett inbyggt system.

Redaktör
Jan Tångring
jan@etn.se
0734-17 13 09

EMBEDDED EXPERT

Energidebugga din inbyggda programvara

Hitta strömtjuvarna och optimera energiförbrukningen i ett inbyggt system.

Anders Lundgren och Lotta Fridmanson, Product Marketing, IAR Systems.

Energidebuggning (engelska ”power debugging”) är en arbetsmetod som förser programutvecklare med information om hur programvaran i ett inbyggt system påverkar systemets strömförbrukning.

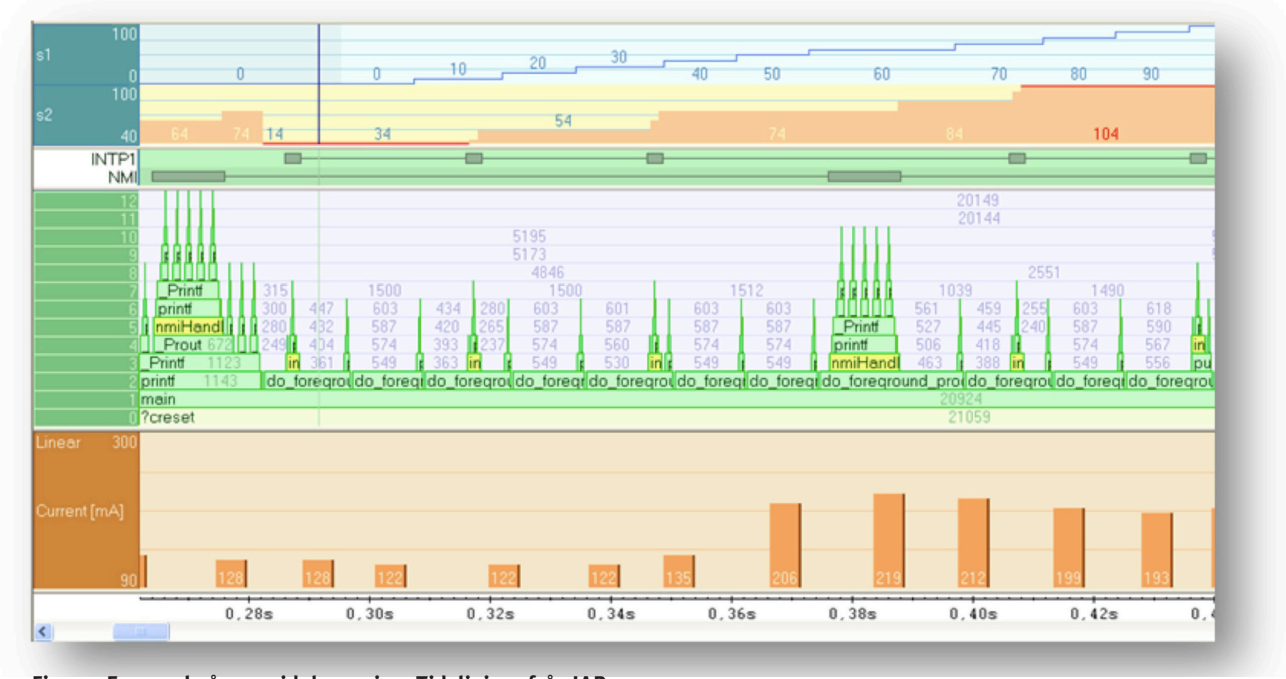
Genom att klargöra kopplingen mellan källkoden och strömförbrukningen blir det möjligt att testa och fintrimma för att minimera energiförbrukningen – det är vad vi har valt att kalla energidebuggning.

Att minska strömförbrukningen har länge varit ett designmål som bara hårdvaruutvecklare har haft något inflytande över. Men i ett aktivt system beror strömförbrukningen inte bara på hårdvarudesignen utan även på hur den används. Hur den används styrs i sin tur av systemets programvara.

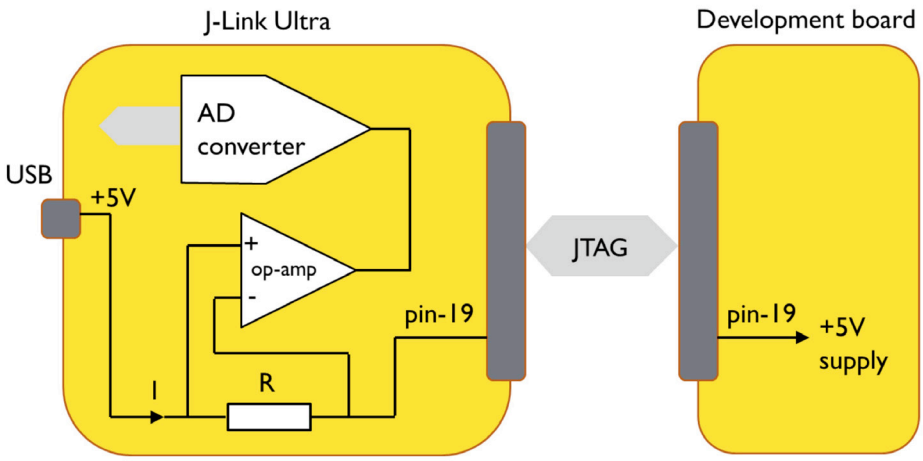
Med IAR C-SPY Debugger från IAR Systems kan man visualisera data om strömförbrukning både statiskt och dynamiskt i olika vyer, och därefter både profilera och debugga programvaran utifrån detta (**figur 1**).

Embedded Workbench

Energidebuggningstekniken bygger på förmågan att kunna mäta strömför-



Figur 1: Exempel på energidebuggning: Tidslinjevy från IAR



Figur 2 - Effektmätning

brukningen och koppla varje mätvärde till programmets instruktionsföljd och därigenom till källkoden.

En svårighet är att lyckas mäta med hög precision. Egentligen behöver man mäta strömförbrukningen vid samma klockfrekvens som systemklockans, men systemets kapacitans minskar pålitligheten hos sådana mätningar. Ur en programutvecklares perspektiv är det därför mer intressant att koppla strömförbrukningen till källkoden och olika händelser i programkörningen än till enskilda instruktioner, så den finkornighet som krävs är mycket lägre än ett mätvärde per instruktion.

Effekten mäts av debuggningsproben. För IAR Embedded Workbench används exempelvis proben IAR J-Link Ultra. Den mäter spänningsfallet över ett litet motstånd seriekopplat med processorkretsens strömkälla (**figur 2**) Spänningsfallet mäts med en differentialförstärkare och samplas sedan av en AD-omvandlare.

Hemligheten bakom tillförlitlig energidebuggning är att ha en god korrelation mellan instruktionstrace och strömmätning. Bäst korrelation får man om man har tillgång till fullständig instruktionstrace. Nackdelen med det är att den inte är tillgänglig för alla processorvarianter, och om den är det så kräver det oftast en särskild debuggningsprob.

Mindre exakt, men fortfarande användbart, är att använda programräknarsamplingen hos en del moderna arkitekturer med debuggningsstöd på kretsen. Den samplar programräknaren (PC) med jämna mellanrum och ger varje sampling en tidsstämpel.

Debuggningsproben mäter hårdvarans strömförbrukning med hjälp av en AD-omvandlare. Genom att tidsstämpla de uppmätta strömvärdena och programräknarsamplingarna kan debuggern visa strömförbrukningsdata på samma tidsaxel som sådana grafer som exempelvis avbrottsloggen och variabelplottningar, och den kan koppla strömförbrukningsdata till källkod (**figur 3**).

I allmänhet är optimering för låg strömförbrukning väldigt likt hastighetsoptimering. Ju snabbare en process exekveras, desto mer tid kan tillbringas i strömsparläge. Alltså minskar vi strömförbrukningen genom att maximera tiden i viloläge.

Det kan vara svårt att identifiera hur ett system förbrukar ström i onödan och var systemets strömkrav kan optimeras. Typiskt sett är det inte fel i programkoden man hittar, utan snarare möjligheter att justera hur hårdvaran används.

Vänta på periferistatus

Ett vanligt misstag som kan leda till onödig strömförbrukning är att använda en pollningsloop för att vänta på att exempelvis en periferienhet ska växla tillstånd. Kodkonstruktioner som i exemplet nedan exekverar utan avbrott tills statusvärdena ändras till det önskade tillståndet:

```
while (USBD_GetState() <
       USBD_STATE_CONFIGURED) {};
while ((BASE_PMC->PMC_SR & MC_MCKRDY) !=
       PMC_MCKRDY) {};
```

En besläktad kodkonstruktion är implementationen av en programvarufördröjning i form av en for- eller while-loop som i detta exempel,

```
i = 10000;          // SW Delay
do i--;
while (i != 0) {};
```

där kodsnutten håller CPU:n upptagen med en instruktion som inte uträttar någonting förutom att få tiden att gå.

I båda dessa situationer skulle koden kunna ändras för att minimera strömförbrukningen. För tidsfördröjningar är det mycket bättre att använda en hårdvarutimer. Timeravbrottet ställs in och sedan försätts CPU:n i strömsparläge tills avbrottet väcker den igen. Även pollning av en enhets tillstånd borde om möjligt lösas med hjälp av avbrott, eller timeravbrott så att CPU:n kan vara i viloläge mellan pollningarna.

DMA eller pollad IO?

DMA har traditionellt använts för att öka överföringshastigheten. På en del arkitekturer kan CPU:n till och med försättas i viloläge under DMA-överföringar. Med energidebuggning kan utvecklaren experimentera och med hjälp av debuggern se vilka effekter de här DMA-teknikerna har jämfört med ett traditionellt angreppssätt med CPU-driven pollning.

Strömspardiagnostik

Många inbyggda program tillbringar det mesta av sin tid med att vänta på att något ska hända. Om processorn fortfarande kör i full hastighet trots att den inte gör något, laddas batteriet ur utan att något uträttas. I många program är alltså mikroprocessorn aktiv bara en bråkdel av tiden – genom att försätta den i strömsparläge kan man förlänga batteritiden rejält.

Ett bra angreppssätt är att använda sig av en processorienterad design och ett RTOS. I en processorienterad design kan en process definieras med lägsta prioritet så att den bara körs när det inte finns några andra processer som behöver köras. Idle-processen är det perfekta stället att implementera strömbesparing på. Varje gång idle-processen aktiveras försätter den i praktiken processorn, eller delar av den, i ett av potentiellt flera strömsparlägen.

CPU-frekvens

Strömförbrukning i en CMOS-mikrokontroller fås i teorin ur formeln:

$$P = fU^2k,$$

där f är klockfrekvensen, U är spänningen och k är en konstant.

Med energidebuggning kan utvecklaren verifiera att strömförbrukningen är en faktor av klockfrekvensen. Ett system som sällan befinner sig i viloläge vid 50 MHz förväntas befinna sig i viloläge 50 procent av tiden när det körs vid 100 MHz. Genom att analysera strömförbrukningen i debuggern kan utvecklaren verifiera att beteendet är som förväntat och om sambandet mellan klockfrekvensen och strömförbrukningen inte är linjärt kan man välja den frekvens som resulterar i så låg strömförbrukning som möjligt.

Avbrottshantering

Figur 4 visar ett diagram över strömförbrukningen i ett händelsedrivet system

där systemet befinner sig i ett passivt läge vid t_0 och strömmen är I_0. Vid t_1 aktiveras systemet och strömmen ökar till I_1, vilket är systemets strömförbrukning i aktivt läge när en periferienhet används. Vid t_2 avbryts körningen av ett avbrott som hanteras med högre prioritet. Periferienheter som redan var aktiva stängs inte av trots att tråden med högre prioritet inte använder dem. Istället aktiveras fler periferienheter av den nya tråden, vilket resulterar i en ström I_2 mellan t_2 och t_3, då tråden med lägre prioritet återfår kontrollen.

Detta system kanske fungerar utmärkt och kan tänkas vara optimerat för exekveringshastighet och kodstorlek, men när det gäller strömförbrukning finns det mer att hämta. Det gula området representerar den energi som hade kunnat sparas om periferienheterna som inte används mellan t_2 och t_3 hade stängts av, eller om prioriteten mellan trådarna hade kunnat ändras.

Energidebuggning hade gjort det enkelt att upptäcka den stora ökningen av strömförbrukningen som inträffade vid avbrottet, och flagga den som abnorm.

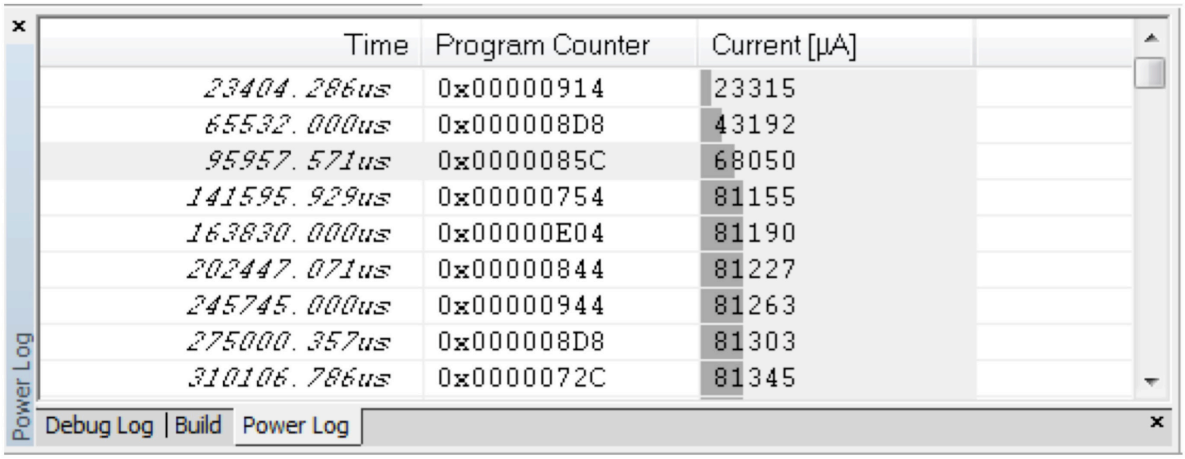
Lokalisera konflikter i hårdvaruinställningarna

För att undvika oanslutna ingångar är det brukligt att jorda oanvända IO-kontakter på mikrokontrollern. Om programvaran av misstag konfigurerar en av de jordade IO-kontakterna som en logisk 1-utgång, kan strömmen genom kontakten bli upp till 25 mA. En så oväntat hög ström ser man lätt genom att läsa värdet på strömmen från energimätningsfönstret, och man kan även lokalisera den initieringskod som är boven i dramat genom att titta på det här fönstret medan programmet startar.

Analog interferens kan också påverka. Att blanda analoga och digitala kretsar på ett och samma kretskort är inte oproblematiskt. Kortlayout och routing blir viktigt för att hålla nere den analoga brusnivån så att mätningen av analoga lågnivåsignaler blir korrekt. God design av blandade signaler kräver skicklighet och noggranna hårdvaruövervägande.

Slutsats

Energidebuggning ger utvecklare av inbyggda system möjligheten att titta inuti programvaran för att avgöra hur programkoden och programflödet påverkar strömförbrukningen. Med den kunskapen kan man fintrimma och optimera källkoden så att strömförbrukningen minimeras. På det viset kan programvaruingenjörer försäkra sig om att deras design är så strömsnål som möjligt utan att det påverkar programmets prestanda negativt.



Time	Program Counter	Current [µA]
23404.286us	0x00000914	23315
65532.000us	0x000008D8	43192
95957.571us	0x0000085C	68050
141595.929us	0x00000754	81155
163830.000us	0x00000E04	81190
202447.071us	0x00000844	81227
245745.000us	0x00000944	81263
275000.357us	0x000008D8	81303
310106.786us	0x0000072C	81345

Power Log

Debug Log | Build | Power Log

Figur 3 Korrelation mellan programräknaren och strömmätning.

Figur 4. Det gula området är energi som skulle kunna sparas.