Topic 6

Software Design for Digital Power – Programming 101 for Analog Designers

Texas Instruments

Software Design for Digital Power Programming 101 for Analog Designers

David Figoli, Texas Instruments

ABSTRACT

Unlike their analog counterparts, digital power supply controllers can benefit from a technique known as Time Division Multiplexing (TDM) to perform more tasks with less resources. TDM permits a single processor or "executable" resource to be shared amongst several independent and often unrelated tasks, e.g. software control loops, diagnostics, fault monitoring, etc. In the analog domain, adding additional functions or tasks requires adding additional components. Resource sharing via TDM is possible because digital power supply controllers operate as time sampled systems. Designing software to take advantage of TDM for a fully digital controlled power supply can be a challenge, but by understanding some key principles and following appropriate guidelines this job can be greatly simplified. In this power seminar module we will examine how to structure software code to take advantage of a class of low cost digital controllers which offer the right performance to get the job done.

I. TIME SAMPLED SYSTEMS

Digital control of power supplies is part of the broader engineering discipline of "Time Sampled Systems". Here continuous time signals are represented as series of data points in time and manipulated via various mathematical "functions", generally know as digital signal processing.

More specifically closed loop control systems process incoming data streams (or series) and act upon it to control or regulate a real process. In its most basic form a single loop control system consists of three key blocks,

- Analog-to-digital conversion (A-D)
- Digital signal processing (or mathematical manipulation of discrete data)
- Digital-to-analog conversion (D-A), the reconstruction of continuous time signal from a discrete series

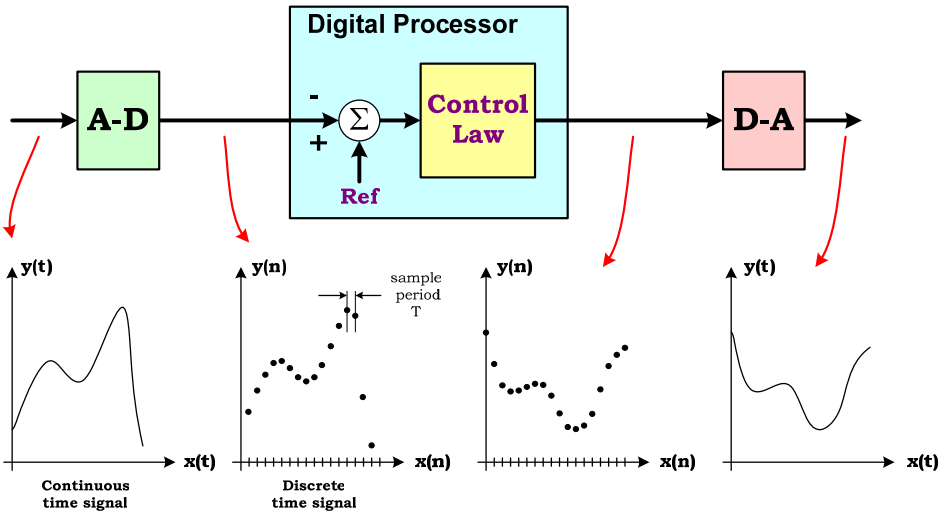


Fig. 1. Key blocks for a digital signal processing system.

In addition to closing one or several control loops, a modern power supply is also required to have a certain level of "General Intelligence" (GI), this can come in many forms, like diagnostics, reporting, start-up, shut-down, supervisory, fault management, communications, etc. With the increasing complexity and performance of modern silicon integration, it makes sense to integrate the functionality of both the control portion and GI portion of this processing into a single high performance digital controller.

II. Time Division Multiplexing for Software – An Overview

Unlike analog control, processing digital or discrete time data allows for multiple task execution to be performed from a single resource, e.g. a digital processor can control multiple loops from a single processing unit. In the analog case extra "processing" units need to be added, e.g. op-amps, comparators and passives to control multiple loops. The technique or mechanism which allows a digital system to perform multiple tasks is called Time Division Multiplexing (TDM), this concept is a very powerful one, but requires that the digital resources (CPU, ADC, PWM, communications, and other peripherals) be carefully and efficiently managed via well written and executed software.

A key parameter governing the application of TDM techniques is the data sampling rate, i.e. T_{SAMPLE}, the period or time between converted data points. Once a data point is sampled a digital processor has only this time to "act" upon this data before the next sample arrives. Failing to do so can lead to severe phase lag or even control loop failure. Fig. 2 shows an example where the processor is capable of executing a single set of control code before the next sample. In this example, a small portion of spare time remains prior to next sample; this is too small to execute another control loop but, as will be discussed later, serves as a good resource for slower background activities.

If the sample rate is made slower or processor throughput is increased then TDM can be effectively utilized. More input signals can be sampled and independently processed. In Fig. 3, 3 independent data streams or samples are shown (red, blue, green), each requiring independent control processing. If the processor of Fig. 2 (yellow above) is utilized, it would need to be duplicated 3 times to achieve given throughput requirements. However a single CPU of sufficient performance, with well structured software can be utilized to perform all three control tasks, C1, C2 and C3. Additionally, some spare time still exists before next sample, allowing for further TDM with slower loops and GI code.

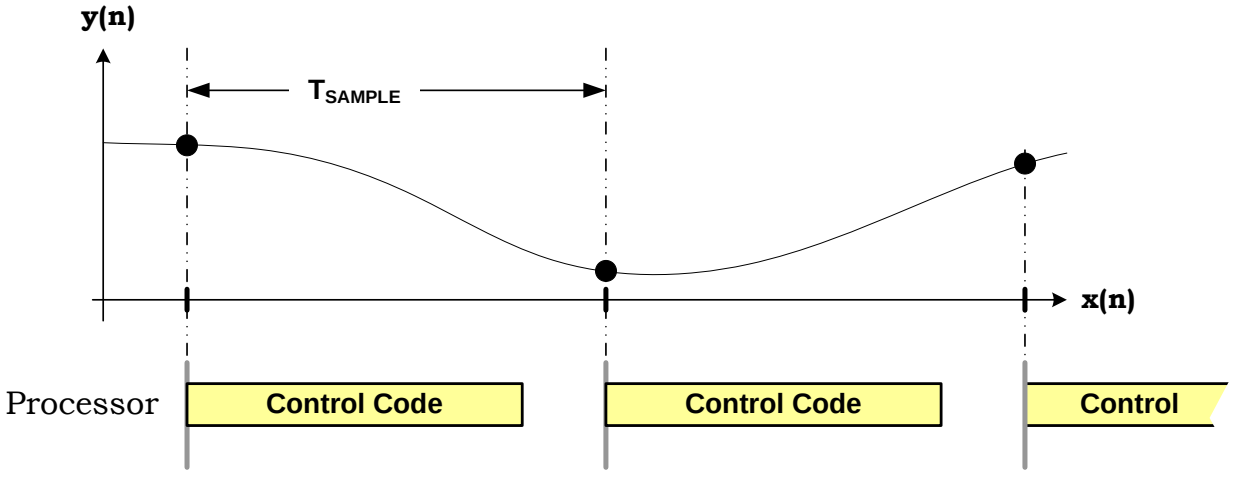


Fig. 2. Sampling interval between converted data points.

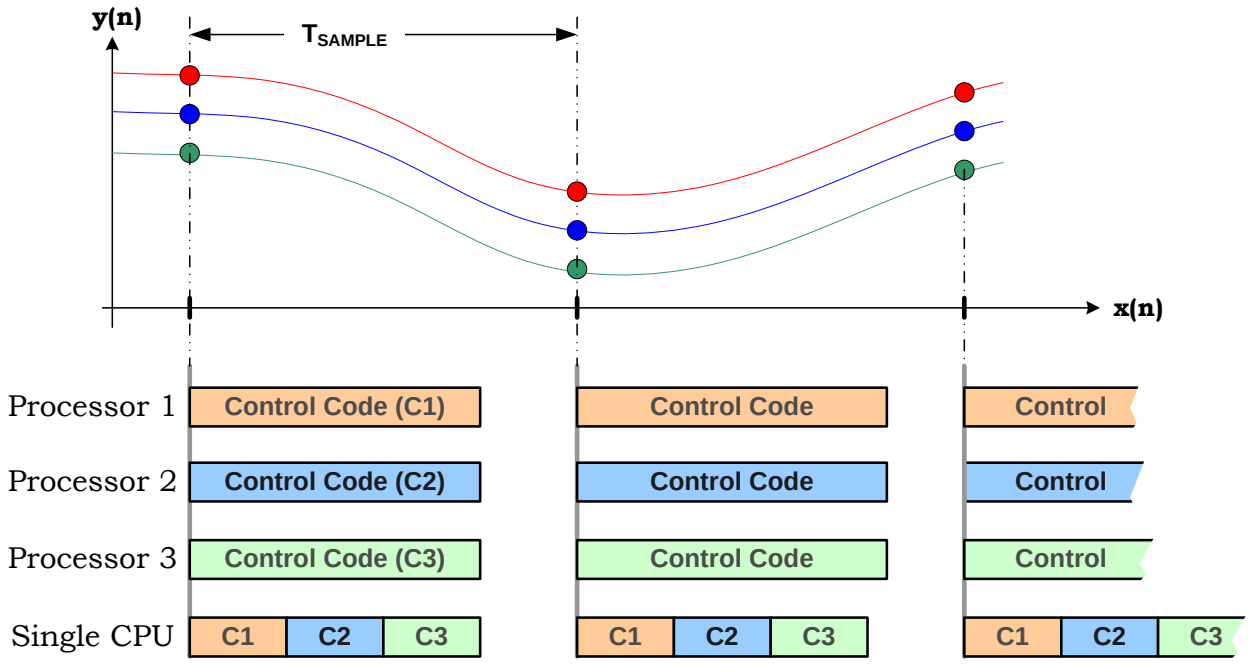


Fig. 3. Three data streams or samples requiring independent control processing.

III. Software Framework Considerations for a Digital Controller

Software framework is the term used to describe the "infrastructure" which supports (like the frame of a house) the application code. It defines the overall code flow and task scheduling, and how the CPU will be shared amongst the various application tasks. Key framework considerations are:

- How many ISRs (Interrupt Service Routines)
- Are ISRs synchronous or asynchronous?
- CPU % utilization balance between ISRs and Background (BG)
- High Level Language (HLL), e.g. "C/C++" or Assembly (ASM) or a mixture of both.
- Need to employ an operating system?
- Interrupt driven communications?

Although there are no wrong or right frameworks, certain choices will directly impact code efficiency/speed, CPU utilization, complexity, ease of debug and ease of development. In general a very good guideline to follow is: use the simplest framework that will get the job done. Even though modern power supplies are becoming more intelligent the embedded software used to run them can employ quite a simple framework consisting of a single ISR and BG structure with TDM management via periodic time slicing techniques.

It is good practice in power systems to have control code execution synchronized with PWM switching events; this is especially true for multi-stage power systems in which asynchronous events can lead to beat frequencies and noise generation. A single ISR synchronous with the PWM switching events has multiple advantages, some of which are:

- Better CPU utilization. An ISR has overhead (context save and restore), hence multi-ISRs increase overhead.
- Multi-ISRs which are asynchronous can lead to non-deterministic code execution. In a time sampled control system ensuring periodic code execution within a real-time deadline is essential.
- With many ISRs triggering, often simultaneously, code debug and development can be more complicated.

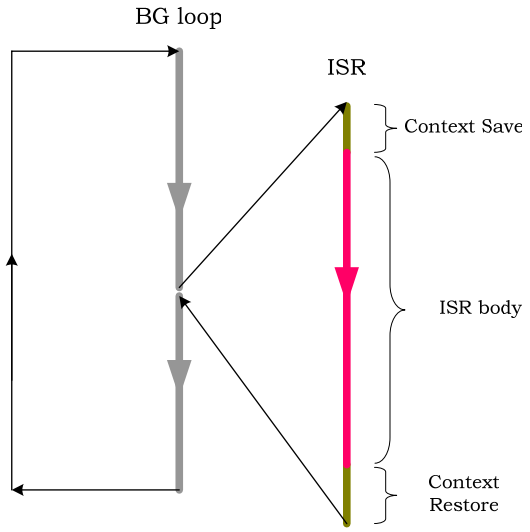


Fig. 5. Flow for a simple framework with single ISR and BG.

In this power seminar module we will focus on the single ISR/BG framework approach. This approach is also suitable for the hardware accelerated CPU controller.

In this scheme the ISR has the highest priority from the CPU and interrupts the BG code synchronous to the PWM switching events. Here the BG code executes only when the ISR relinquishes the CPU, i.e. during ISR idle time. The ISR consists of base code which executes every time and time slices (TS1, TS2, etc) which execute in a round-robin basis. The time-slice execution rate is the ISR rate divided by the number of time slices used. Whenever an ISR executes it carries an overhead, known as context save and restore, this is essentially wasted CPU cycles, but need to be accounted for when estimating CPU bandwidth utilization, discussed later. When a given code flow, e.g. Background C code, is interrupted, the context or state of all key CPU registers and status bits must be temporarily saved by the ISR such that on return, the BG code can continue seamlessly from point of interruption. The amount of ISR overhead depends mainly on the CPU type, and the amount of CPU resources the ISR will utilize during execution. The ISR need only save and restore the CPU registers that it uses, hence "lean" overhead ISRs are possible if care is taken. Often however, to reduce risk and help with software expansion and maintenance a full context save/restore is recommended.

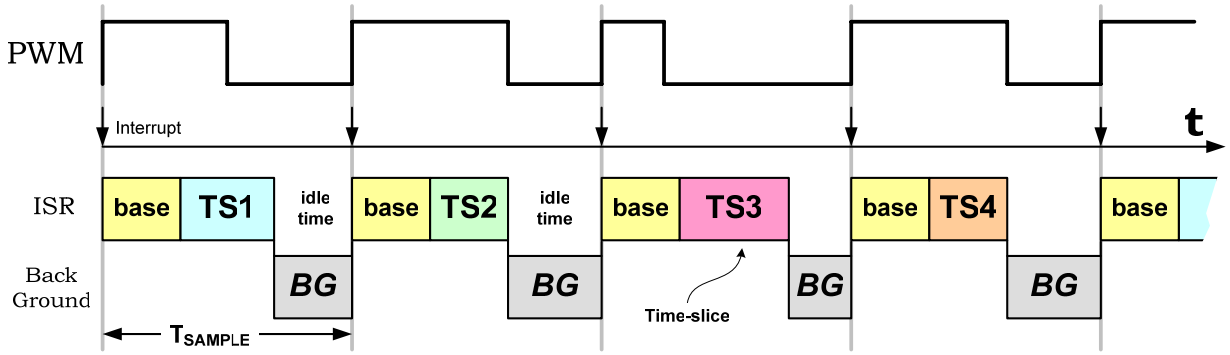


Fig. 4. Time sliced ISR synchronized to the PWM switching events.

A. Hardware Accelerated Digital Controllers

In cases where the processing requirement is more extreme, e.g. multiple very high frequency control loops with PWM frequencies in the 1~2 MHz range, more digital resources can be added in parallel, for example, hardware accelerators can be employed to assist the CPU in performing the compensation, (control law difference equations), of high speed loops while it handles low speed loops and GI functions. The industry trend is for both approaches, and in this seminar module we will explore software control of both single CPU based controllers and hardware assisted (accelerated) CPU based controllers.

Typically single CPU based controllers employ high performance DSP engines capable of executing millions of complex math instructions/operations per second. Hardware assisted controllers use a balance between a moderate performance MCU engine coupled with a very narrowly focused "hardwired" compute engine which performs a given type of difference equation (e.g. IIR filter) at very high speeds. The choice between the controller choices really depends on the end application and PWM frequencies to be targeted. In general a hardware accelerated CPU controller can achieve a greater throughput and hence greater control bandwidth, but this at the expense of control loop flexibility due to a fixed control law i.e. compensation scheme. For simpler multiple output high frequency voltage mode buck stages, for example, this is a great choice. Alternatively with a single DSP engine controller, a designer has greater flexibility in the choice of control strategies chosen, for example taking advantage of adaptive schemes or average current mode control with inner/outer loop strategies. Additionally a high performance DSP compute engine can push TDM techniques to greater limits permitting a large number (e.g. 4 ~10) of "soft loops" to be deployed without additional resources.

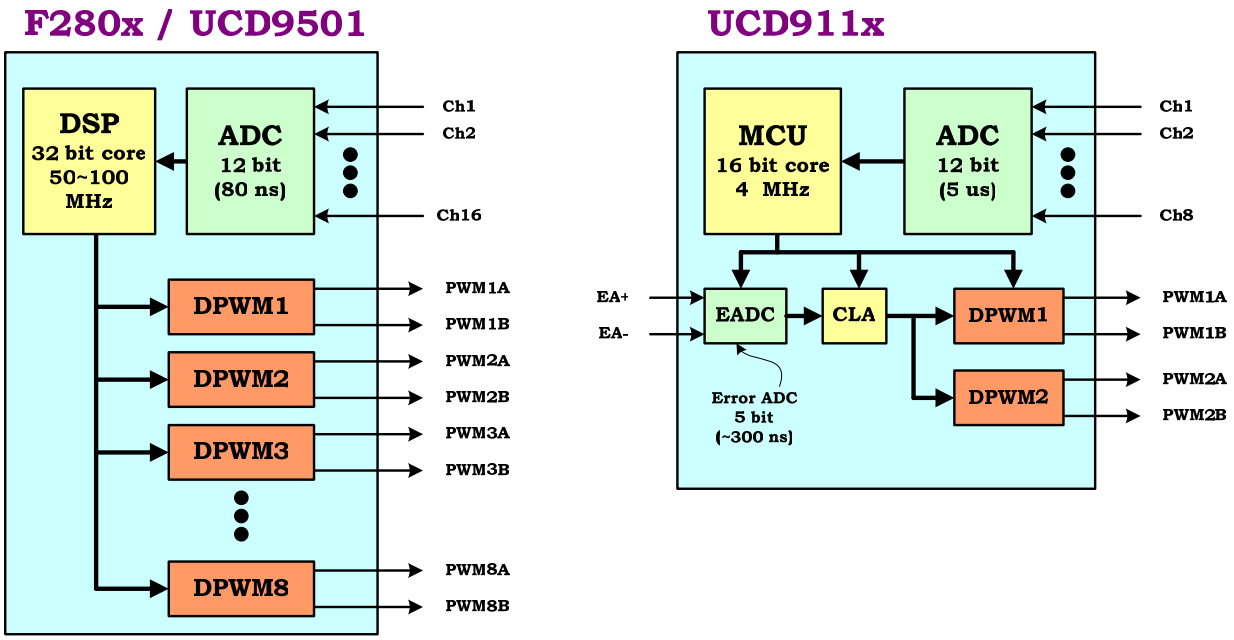


Fig. 6. Example of a non-accelerated (F280x) and H/W accelerated (UCD911x) digital controllers.

B. *Software Strategy for Hardware Accelerated Digital Controllers*

Whether a single CPU based controller is used or a hardware accelerated one, well structured and efficient software is vital in extracting the most out of a given digital controller, regardless if it is hardware accelerated or not. The two cases are outlined below showing the split between processing tasks. Since the CLAs are hardwired controllers, software considerations need only be targeted to the CPU code execution, and the balance between Interrupt Service Routine (ISR) code and Back-Ground (BG) code. In this respect from a CPU perspective, both cases need to deal with multi loop ISR code and multi function/loop BG code, the main difference being that in case 2 the CPU ISR needs to handle loops up to the "MHz" type range while in case 1 this range is in the "10s of kHz". As has been mentioned previously (and will be explored more later) this is really only a function of sampling rate and CPU throughput, software techniques to extract best performance are the same.

1. In the hardware accelerated case (e.g. UCD911x):
   - 1 × "MHz" loop is possible via a Control Law Accelerator (CLA)
   - Several slower (<~10 kHz) loops via CPU ISR
   - General intelligence via spare CPU cycles in a BG loop

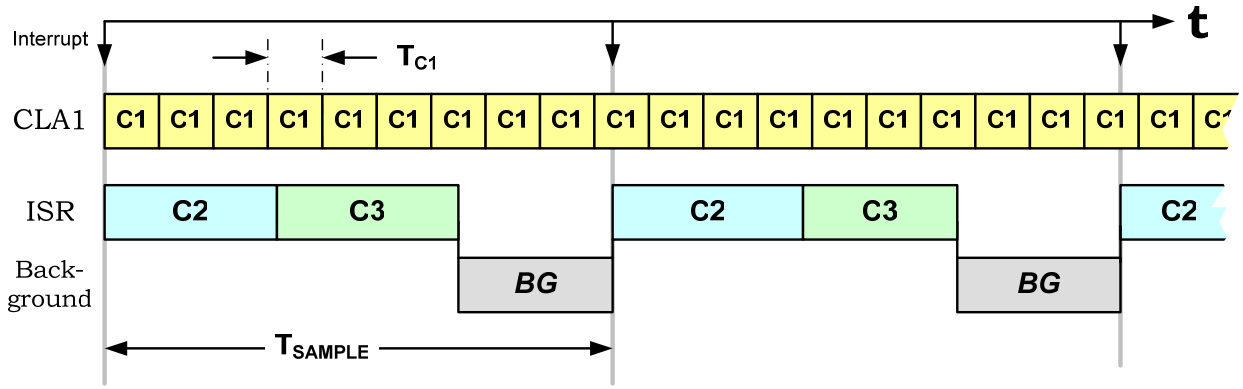


Fig. 7. Execution flow for hardware accelerated case.

2. In the single DSP engine case (e.g. F280x):
   - 2~3 × “MHz” loops are possible via CPU ISR
   - Several slower (<10~40 kHz) loops via spare CPU cycles in BG
   - General intelligence via spare CPU cycles in BG

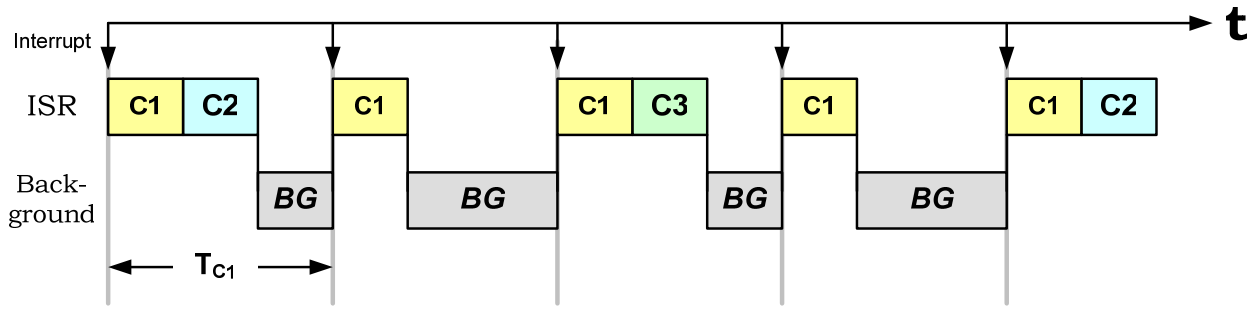


Fig. 8. Non-accelerated case with time-sliced ISR.

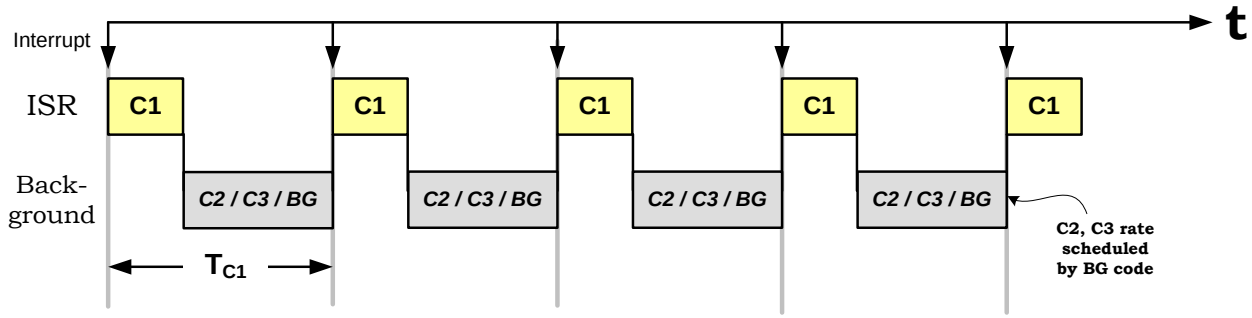


Fig. 9. Non-accelerated case with time-sliced BG loop.

IV. Estimating CPU Performance Needs for a Given Application

For any given power supply application it is important to determine if sufficient CPU performance is available, this will ensure the code will keep up with the sampled data throughput. It is useful to examine some concepts which will help with this determination:

- Available processing time
- MIPS (Millions of Instructions Per Second)
- Effective MIPS (also called "quality of MIPS")
- ISR bandwidth utilization
- BG bandwidth (average background code bandwidth)

A. Available Processing Time

As discussed previously, the available processing time is determined by the sampled data rate. The time between samples, T_{SAMPLE}, is simply calculated as the inverse of the sampling rate. Often this is the same as the PWM frequency. The table shows in ns (nano-seconds) the time available between samples for various PWM frequencies.

Table I. Time Available Between Samples

Sample Freq (=PWM) (kHz)	Sample Period (ns)
100	10000
300	3333
500	2000
700	1429
1000	1000
1500	667
2000	500

B. MIPS (Millions of Instructions Per Second)

If used properly, this "figure of merit" can be used to calculate the amount of work (processing) that can be done between data samples. For most GP (general purpose) CPU engines:

$$\text{clock rate} \neq \text{MIPS} \quad (1)$$

This is because instructions are not executed in a single cycle. Instructions are usually multi-cycle, and can vary, for example, from 2 ~ 6 instructions depending on the instruction complexity. To calculate MIPS in this case, a "de-rating" factor needs to be applied, this can be as simple as the median of the instruction count distribution, for example (1) above, this may be equal to 4 cycles/instruction. Hence the CPU clock rate needs to be de-rated (divided) by 4, so a 50-MHz CPU has an equivalent MIPS = 12.5 caution must be paid to multi-cycle multiply instructions, these are often 16 or more cycles, and can "average down" considerably the throughput. Often the most accurate method is to simply measure the actual number of cycles, i.e. manually count or use a profiling feature in the code debug environment to automatically evaluate this.

DSP based CPU engines on the other hand execute most (~95%) instructions in a single cycle and hence a de-rating of 1 can be used, i.e. clock rate = MIPS. In addition an extra performance boost can come from a single cycle multiply/accumulate combination, i.e. product and previous sum; this is especially useful because most mathematical processing can be broken down into "sum of product" type structures.

C. *Effective MIPS, Also Called "Quality of MIPS"*

"Not all MIPS are created equal" is a comment often made by experienced software designers. Examining and comparing MIPS quality among different CPUs can be quite complex, and beyond the scope of this introductory module, however as an illustration of the concept, let's examine data width, i.e. 8-bit, 16-bit, and 32-bit machines.

"8-bit MIPS" ≠ "16-bit MIPS" ≠ "32-bit MIPS"

Although this may seem obvious, and it may appear that a 16-bit machine is 2×"8-bit machine", this is true for simple data movement, however if multiplication is heavily used, e.g. filtering, or compensation where sum of products are calculated, the effective MIPS of a narrow data width machine (e.g. 8 bits) may be drastically reduced. The simple multiply example below illustrates this.

Let's assume each digit represents an 8-bit quantity (mathematically valid for illustration purposes). A 16×16 instruction (left) executes in a single cycle resulting in a 32-bit product (5561). On an 8-bit machine, this operation needs to be done as 4 separate multiplications and a summation of 4 terms with correct decimal point weighting (i.e. shifting). At a minimum and with appropriate hardware in place it would take 5 cycles, hence a MIPS quality de-rating of 1:5 results, hence a MIPS quality de-rating of 5 results, meaning that the effective MIPS is reduced by a factor of 5 in a multiply intensive algorithm.

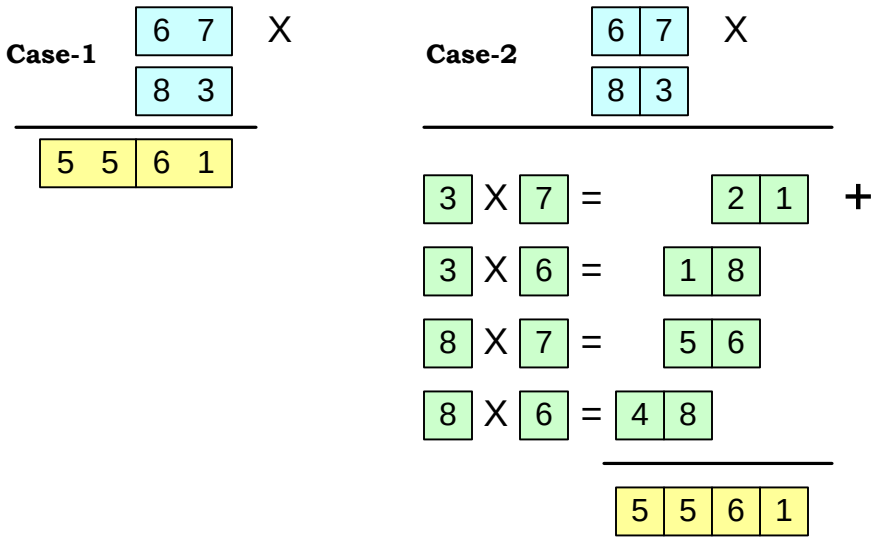


Fig. 10. Reducing MIPS of a narrow data width machine.

D. ISR Bandwidth Utilization

Determining the percentage of CPU bandwidth required to execute the ISR code is one of the most important considerations in designing a digitally controlled power supply system. Calculating this with a good confidence level will determine if the target control loop performance can be realized, how much spare CPU capacity can be allocated to the BG loop, and finally it indicates how much room for future software expansion is available. The following example, based on a voltage mode control loop, illustrates in some detail the process used to make this calculation. The control loop components and synchronous buck power stage are shown in Fig. 11.

The software control loop in this example was implemented in assembly language to achieve the lowest number of CPU cycles. The 2-pole/2-zero compensation filter is an IIR second order and is available as a software library module from TI. The control loop equations implemented here are shown below. The control loop equations implemented here are shown in Fig. 12.

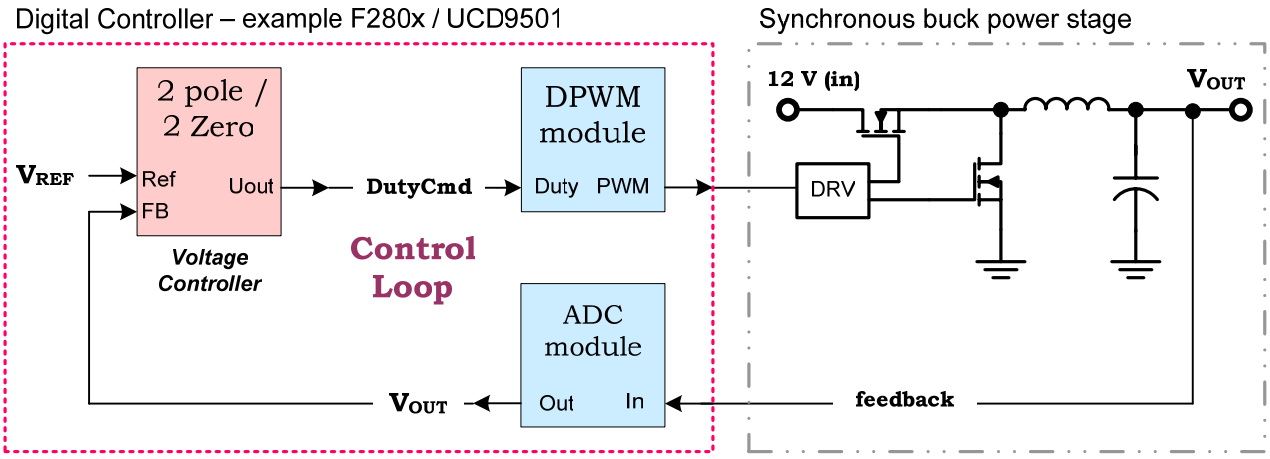


Fig. 11. Synchronous buck power stage with the software control loop.

$$\frac{U(Z)}{E(Z)} = \frac{B_0 + B_1Z^{-1} + B_2Z^{-2}}{1 + A_1Z^{-1} + A_2Z^{-2}} \quad \ldots\ldots (1)$$

$$U(n) = B0*E(n) + B1*E(n-1) + B2*E(n-2) + A1*U(n-1) + A2*U(n-2) \quad \ldots\ldots (2)$$

$$E(n) = V_{OUT} - V_{REF} \quad \ldots\ldots (3)$$

$$U_{Q0}(n) = PrdSF * U(n) \quad \ldots\ldots (4)$$

Fig. 12. Implemented control loop equations.

Equation (1) in Fig. 12 is the transfer function in the Z domain, its difference equation (2) is the code implemented in the 2-pole/2-zero controller block. Equation (3) is the error term calculation. Equation (4) is the scaling required to map the Q15 fixed point fractional representation of U(n) to a Q0 integer form based on the PWM period value used. PrdSF is the scaling factor. The number of cycles required to execute the above control equations is very accurately known (from TI library) and is summarized in Table II. Also listed in the table are other cycle numbers related to ISR overhead, namely context save/restore and ADC servicing and interrupt acknowledge. In this example the ADC logic causes an interrupt trigger at End of Conversion (EOC).

Table II also shows total clock cycles for 2 and 3 loops. Note, the context save/restore overhead remains the same, also interrupt acknowledge is done only once per interrupt. We now have all the information we need to accurately estimate the CPU utilization within the ISR. Fig. 13 summarizes the sampling and processing operation in terms of a time-line. Note, for a given ISR clock cycle count, the balance between ISR utilization and BG bandwidth (discussed later) is governed by the sample period time T_{SAMPLE} (inverse of sampling rate or frequency F_{SAMPLE}).

TABLE II. KNOWN CYCLES REQUIRED TO EXECUTE CONTROL EQUATIONS

Operation	# Clock Cycles (1 loop)	# Clock Cycles (2 loops)	# Clock Cycles (3 loops)
Context Save + Int. latency	16	16	16
ADC servicing + Ack	4	5	6
2P/2Z controller	25	47	69
DPWM access	4	8	12
Context Restore + Int. Return	16	16	16
Total	**65**	**92**	**119**

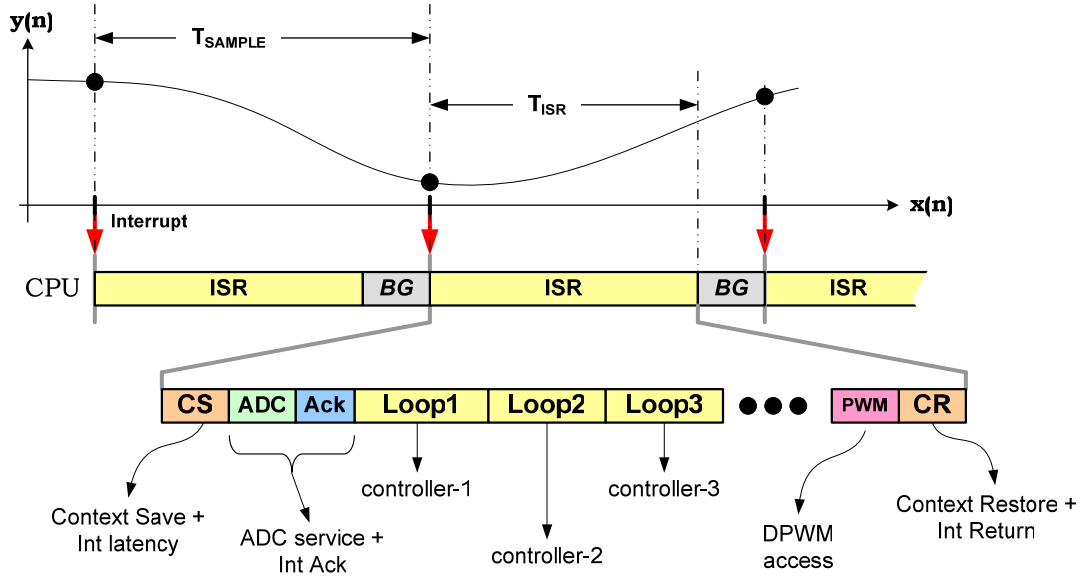


Fig. 13.Time-line summary of sampling and processing operation.

The calculation for % ISR utilization is simple and given by:

$$ISR\ utilization = \frac{T_{ISR}}{T_{SAMPLE}} \times 100\% \quad (2)$$

Examples of TSAMPLE values for various PWM frequencies were given earlier, to calculate TISR, we need to know the CPU clock speed of the controller used in the application. For the F280x/UCD9501 this clock speed is 100 MHz, which gives a clock period of 10 ns. So for the two control loops example, TISR = 92×10 ns = 920 ns. Assuming the sampling rate = PWM frequency, then at PWM = 1 MHz, i.e. PWM period = 1000 ns, then we would have 80 ns spare and an ISR utilization = 92%. Although the two control loops would run fine, the BG loop would not receive much bandwidth, so we need to make some trade-off, perhaps lowering the PWM frequency to 800 kHz. For convenience, Table III gives calculation results (percent of ISR utilization) for various combinations of PWM frequencies versus number of control loops, this is useful in making design trade-offs. Note, red entries exceed 100% utilization and are not possible under given assumptions.

E. BG Bandwidth, Average Background Code Bandwidth

As discussed earlier, the ISR has the highest CPU priority, this means the BG code or loop has access to any remaining CPU bandwidth, i.e. BG BW = 100% - ISR utilization. For example, in the case of PWM = 600 kHz and 4 loops, ISR utilization = 88%, hence BG BW = 12%. Although BG activities are not as time critical, it is useful however to estimate loop rates of any slower periodic functions running in the background. These activities may include slow voltage loops, current monitoring and balancing, temperature monitoring, and communications.

Unlike the ISR, the BG loop does not have precise deterministic timing, it is typically built around decision ("if then else") type code and usually written in high level language such as C/C++. Even so, it is still possible to estimate an average or aggregate loop time, i.e. the average time that it takes for the code to return to the same point again. It is important to know this especially if the periodic tasks mentioned previously need to be performed.

The average BG loop rate is calculated as follows:

$$BG\ LR = \frac{(BG\ MIPS)}{\#\ Instructions\ in\ longest\ path} \quad (3)$$

TABLE III. PERCENT OF ISR UTILIZATION FOR PWM FREQUENCY VS NUMBER OF CONTROL LOOPS

CPU clk = 100 MHz 10 ns

PWM		Number of LOOPS and Number of Cycles				
(kHz)	(µs)	1 65	2 92	3 119	4 146	5 173
200	5.00	13%	18%	24%	29%	35%
300	3.33	20%	28%	36%	44%	52%
400	2.50	26%	37%	48%	58%	69%
500	2.00	33%	46%	60%	73%	87%
600	1.67	39%	55%	71%	88%	**104%**
700	1.43	46%	64%	83%	**102%**	**121%**
800	1.25	52%	74%	95%	**117%**	**138%**
900	1.11	59%	83%	**107%**	**131%**	**156%**
1000	1.00	65%	92%	**119%**	**146%**	**173%**
1100	0.91	72%	**101%**	**131%**	**161%**	**190%**

BG MIPS = BG BW×total CPU MIPS, in this example BG MIPS = 12%×100 MIPS = 12 MIPS. At this point let's make an assumption for the number of instructions in the longest path, is say 300. Later we will see that by following some simple guidelines and structuring recommendations that we can minimize this path and make it's timing more accurate. We can now calculate the loop rate as follows:

$$BG\ LR = \frac{12{,}000{,}000}{300} = 40\ kHz \qquad (4)$$

V. Practical Guidelines to Software Design

Section III. (Software Framework Considerations) discussed the benefits of using a simple single ISR/BG loop combination. This strategy is very applicable for most digitally controlled power supplies. Its greatest advantage is simplicity; the entire software scope is limited to two main flows or loops. In flow chart form a software system may look something like the example below.

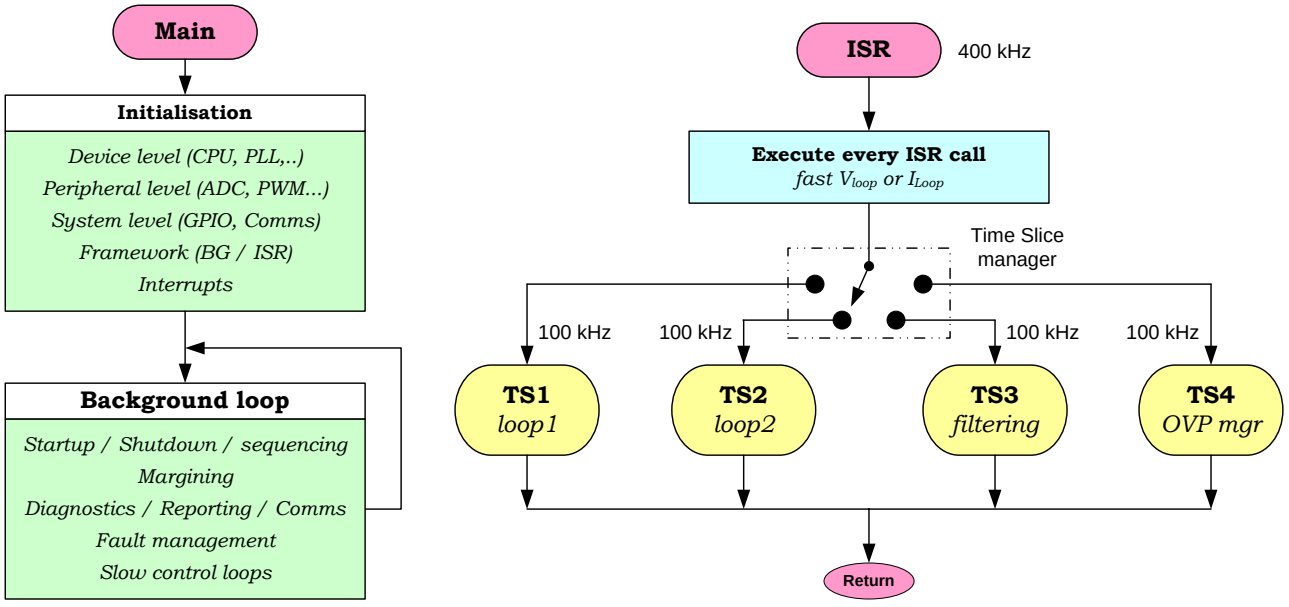


Fig. 14. Flow chart example of a typical software system.

A. *Do We Use "C" Code, Assembly or a Mix of Both?*

This is a question many software designers struggle with. As usual there is no wrong or right answer here, it really depends on the desired outcome. The following discussion will give some perspective in making this decision. Fortunately in the single ISR/BG approach only two choices need to be made, what to use for the ISR and what to use for the BG? Let's start with the BG, this is a straight forward choice.

The BG loop typically contains more than 90% of the total code, it is what gives the power supply it's "personality" i.e. differentiated features, intelligence, fault management, "smart" diagnostics, etc. Here C code (or C++) makes the most sense, it is a powerful language, and has great flexibility in performing the hundreds of "if then else" type decisions typical of background code. However, good coding structure is still very important in the BG if high loop rates and minimal latency is required, this will be discussed in detail in sub section C.

The ISR on the other hand can go either way, C or ASM. It really depends on how much performance a designer wants to extract out of a given CPU. Generally, because C code is so standardized (meets ANSI standards) it is somewhat abstracted from the underlying CPU architecture, i.e. it often does not take advantage of the various hardware resources sufficiently well, for example shift registers, multiplying modes, arithmetic saturation, addressing modes, etc. Many C compilers (e.g. TI C28xx/UCD95xx compiler) do however offer Intrinsics, which can get around this limitation in some cases. ASM (assembly language) is a custom fit to the CPU architecture, it has complete control of every feature, mode and Bit a CPU has to offer, and hence can extract the best performance possible from a given architecture. In general if a designer wants the best performance at a given price point, i.e. “do more for less”, for example by choosing a lower MIPS device, or pushing performance on a given device beyond competitors using the same device, then investment in ASM coding can pay off.

The thought of coding in ASM worries many software designers, they are usually concerned about:

- Coding complexity (writing large amounts of hard to understand ASM instructions)
- Maintenance of large amounts of ASM code
- What is the tangible benefit from using ASM code

Coding Complexity

Let's examine this in more detail, to see if we can alleviate much of this concern to the point where using ASM code becomes an advantage. Much of the concern is centered on the amount of ASM code an application may require. As mentioned previously, the bulk of the code resides in the BG loop which is in C, and only a small portion resides in the ISR. Recall that the primary reason for using an ISR is to implement a high priority periodic task which processes time sampled data. In fact if we examine the time available between samples in real applications, we see that it is actually physically impossible to "fit" (execute) a large number of instructions within a given sample period. The amount of code is physically bounded, hence the concern that code amounts like 5 K, 10 K, or 20 K bytes need to be developed/written is in reality not true. To illustrate this let's examine some figures. Table IV shows the maximum number of instructions physically able to execute within a sample period, assuming straight line coding, i.e. no loops or branches. For example a 60 MIPS CPU processing data at 200 KSPS (kHz) can execute at best 300 instructions (or lines of code). This implies 100% ISR utilization, more realistically if we apply the 75%/25% rule (i.e. 25% for BG) and take into account ISR overhead (this is one time code re-used throughout all applications) then only ~200 lines of ASM code is the actual software designer's burden in this example. Moreover, if a CPU with multi-cycle instructions is used the MIPS de-rating factor reduces the amount of ASM code even further.

Maintenance Considerations for ASM Code

On the question of ASM code maintenance, two points need be made, first, as discussed in A) the amount of code is not really large to begin with and second, the code that generally executes in an ISR consists predominantly of mathematical functions and interface to ADC and PWM peripherals. This code once developed and debugged seldom changes. It is analogous to wanting to modify or maintain C library math functions like sine, cosine, square-root, etc. There is no need, a designer trusts the functions work and keeps re-using them. As an example, in Fig. 15, below, is a 2-Pole/2-Zero compensator (i.e. IIR filter difference equation) used in many closed loop systems. Although the ASM code appears complex it is either obtained as part of an existing library, example from TI, or coded once by a knowledgeable ASM designer. This is the code snippet which implements the difference equation previously shown as equation (2).

TABLE IV. NUMBER OF INSTRUCTIONS POSSIBLE IN A SAMPLE PERIOD FOR A GIVEN CPU MIPS RATING

F_{SAMPLE} (=PWM) (kHz)	MIPS		
	25	60	100
200	125	300	500
250	100	240	400
300	83	200	333
350	71	171	286
400	63	150	250
500	50	120	200

```
;Calculate input (Ref - Fdbk):
 MOV        ACC, *XAR0++<<11              ; ACC = Ref1  (Q15 to Q26)
 SUB        ACC, *XAR1++<<14              ; ACC = Ref1 - Fdbk1 (Q12 to Q26)
 MOVL @CNTL_2P2Z_DBUFF1+4, ACC      ; e(n) = ACC = error (Q26)

;Calculate 2p-2z filter:      Note: based on Q25 coefficients
 MOVL XT, @CNTL_2P2Z_DBUFF1+8       ; XT = e(n-2)
 QMPYL      P, XT, *XAR3++                ; P  = e(n-2)*B2  (Q20 = Q26*Q26)
 MOVB ACC, #0                       ; ACC = 0
 MOVDL      XT, @CNTL_2P2Z_DBUFF1+6       ; XT = e(n-1), e(n-2) = e(n-1)
 QMPYAL     P, XT, *XAR3++                ; P  = e(n-2)*B1 (Q20 = Q26*Q26)
 MOVDL      XT, @CNTL_2P2Z_DBUFF1+4       ; XT = e(n), e(n-1) = e(n)
 QMPYAL     P, XT, *XAR3++                ; P  = e(n)*B0    (Q20 = Q26*Q26)
 MOVL XT, @CNTL_2P2Z_DBUFF1+2       ; XT = u(n-2)
 QMPYAL     P, XT, *XAR3++                ; P  = u(n-2)*A2 (Q20 = Q26*Q26)
 MOVDL      XT, @CNTL_2P2Z_DBUFF1+0       ; XT = u(n-1), u(n-2) = u(n-1)
 QMPYAL     P, XT, *XAR3++                ; P = Y1*A1 (Q20 = Q26*Q26)
 ADDL ACC, @P                       ; ACC = u(n-1)*A1+u(n-2)*A2+
                                           ; e(n)*B0+e(n-1)*B1+e(n-2)*B2 (Q20)
; Scale u(n) Q20 to Q26, and save it
 LSL  ACC, #6                       ; ACC = Q26, based on Q26 coef & e(n)
 MOVL @CNTL_2P2Z_DBUFF1, ACC        ; u(n-1) = u(n) = ACC (Q26)

; Saturate the result [0,1] & move to Uout as a Q15
 MINL ACC, *XAR3++                  ; Saturate to < 0.999999.. in (Q26)
 MAXL ACC, *XAR3++                  ; Saturate to > 0.000000.. in (Q26)
 LSL  ACC, #5                       ; Convert from Q26 to Q31 (Duty in Q31)

; output Uout
 MOV        *XAR4++, AH                 ; Output Duty (Q15) to terminal net
```

Fig. 15. An ASM code sample for a 2-pole/2-zero filter function.

Benefits of ASM Code

How much benefit will going to assembly language bring? The best way to gauge this is to get some perspective on the code performance boost obtained. This can be readily seen from the ISR utilization factor, and the impact that saving some instructions have on this value. For example if an application is estimated to have an ISR utilization factor of say 10% if coded in C, then coding the ISR in ASM has little to no impact, since the BG loop gets 90% BW and plenty of margin for overhead exists. On the other hand if performance is critical (high sampling rates) and code execution is tight, then saving a few cycles can make all the difference. A useful way to look at this is in Table V. This shows (in red) the percent of impact that 10 instructions can make on ISR utilization for various sample rates and CPU MIPS. For example, at an ISR loop rate of 400 kHz, while running a 60 MIPS CPU, saving 10 instructions can reduce the ISR utilization by 5%. In practice usually more than 10 instructions can be saved by adopting an ASM based ISR, and hence 10~30% reduction in utilization can often be realized.

Another equally important way to look at this is to see the impact on the BG loop rate. The BG loop often has slower tasks running but which still must maintain a certain periodicity, i.e. execution rate, for example an over-voltage or over current manager, or simply a slower control loop which must rely on a given sample processing rate to be valid.

Recall the example where:

$$BG\ LR = \frac{12{,}000{,}000}{300} = 40\ kHz \tag{5}$$

This was achieved based on an ISR utilization of 88%, i.e. BG BW = 12%. Now let's assume a saving of 9% is achieved by saving 15 instructions (15 inst = 10+5 =6%+3%, see table, 600 KHz and 100 MIPS), hence the ISR Utilization is now 79% and BG BW = 21%. Recalculating the BG loop rate and assuming the same BG code length, this gives:

$$BG\ LR = \frac{21{,}000{,}000}{300} = 70\ kHz \tag{6}$$

This is a significant boost in execution rate. Note a 9% saving (with just 15 instructions) in the ISR, has resulted in a 75% boost in BG loop rate. Note, the closer to the "edge" (i.e. ISR → ~100%) the more dramatic this boost will be.

TABLE V. PERCENT OF IMPACT OF 10 INSTRUCTIONS ON ISR UTILIZATION FOR PWM FREQUENCY VS CPU MIPS

F_{SAMPLE} (=PWM) (kHz)	MIPS		
	25	60	100
200	8.0%	3.3%	2.0%
250	10.0%	4.2%	2.5%
300	12.0%	5.0%	3.0%
350	14.0%	5.8%	3.5%
400	16.0%	6.7%	4.0%
600	24.0%	10.0%	6.0%

B. ISR Structure Details – Using Time Slicing

A single ISR for the control loop makes a lot of sense, it incurs the context save/restore overhead only once, and being the only ISR means it has the highest CPU priority and hence its execution is precisely periodic and deterministic. The price we pay for this is somewhat less flexibility when implementing multi-loop/multi-stage systems. In these cases we do not have the flexibility of choosing arbitrary combinations of sample rates and PWM frequencies. Our choices must be limited to frequencies (and sampling rates) having an integer multiple relationships with each other or to a fundamental frequency. Actually in power systems this restriction to integer multiple frequencies turns out to be a requirement rather than a limitation, since it is undesirable to have mismatched frequencies due to beating effects and non deterministic switching noise, making it very difficult to schedule ADC start of conversion triggers during non-switching time windows.

Implementing a time slice manager within an ISR is quite simple and very efficient on cycles. Any number of time slices can be chosen, whereby effectively dividing the ISR frequency by the chosen number. Below is a simple example of an 8-slot time-slicer.

Here four separate cases are shown, where the code module C1 is shown to execute at different rates, depending on which Time Slice (TS) it occupies. Four choices are possible, same frequency as the ISR, 1/2 rate, 1/4 rate and 1/8 rate.

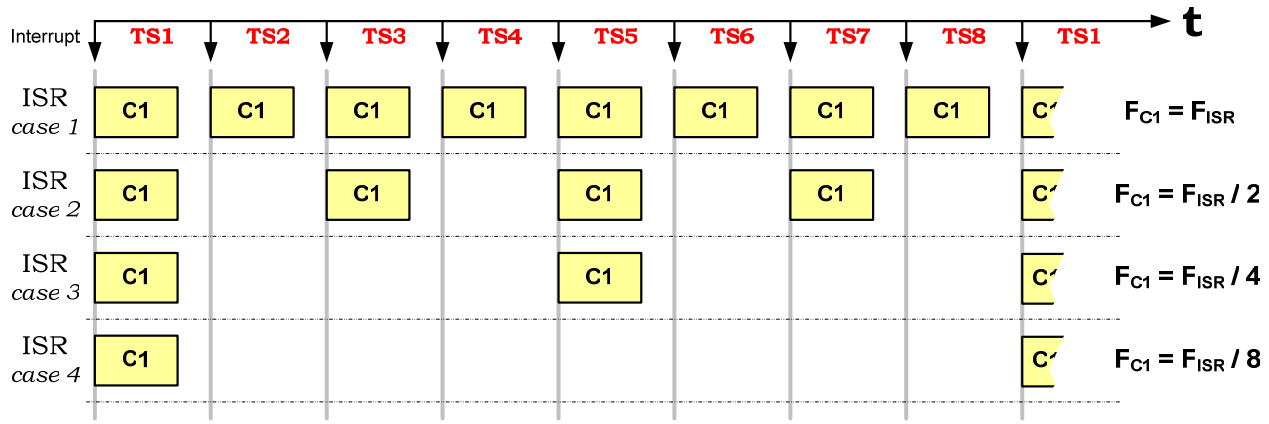


Fig. 16. 8-slot time-slicer.

To show how the time slice technique is used in practice, below are two examples of multi-stage/multi-output power systems, requiring multiple independent control loops all executed from a single CPU and ISR. The first example is control of multiple DC/DC bucks which might be used in a POL application. Two single phase bucks (independent outputs) are running at full ISR speed, i.e. 600 kHz, while three independent multi-phase interleaved bucks are running at 150 kHz and are run during only one time-slot, where by reducing the execution rate to 1/4 of the ISR rate.

Note, the PWM frequency of bucks 3, 4 and 5 is 300 kHz in this example, it could have been made 150 kHz, or 600 kHz, depending on magnetics. It does not need to be the same as the code execution rate, only in integer multiple of it.

TABLE VI. SUMMARY OF FREQUENCIES FOR EACH BUCK STAGE

Code Function	PWM rate (kHz)	Code execution rate (kHz)	Identifier
Buck 1 - single phase V Loop	600	600	B1
Buck 2 - single phase V Loop	600	600	B2
Buck 3 - 4-phase IL V Loop	300/phase (90° apart)	150	B3
Buck 4 - 3-phase IL V Loop	300/phase (120° apart)	150	B4
Buck 5 - 3-phase IL V Loop	300/phase (120° apart)	150	B5

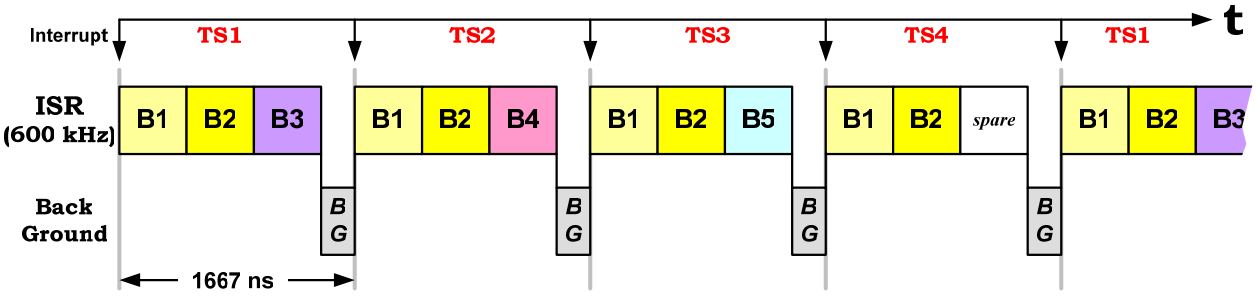


Fig. 17. Time-slice timing for the 5-buck system.

The second example is for an AC/DC rectifier implementation. This particular rectifier has two DC/DC outputs and in addition to PFC and Boost, has many code modules running. Here also it is convenient to choose a four time-slice ISR approach. Table VII summarizes the multiple code activities that need to execute and at what rate.

Some points to consider:

- The PFC voltage loop is typically a very slow loop (a few kHz) and can be run in the BG loop, providing a periodic execution rate, (say 1 kHz) can be implemented. More details on this later in the BG structure discussion.
- Not all the software tasks shown need to run at an execution rate of 50 kHz or greater, some can run just fine at 25 kHz or even 12.5 kHz, for example the current balancing loop (I_{BAL}), however it may be more convenient to just allow it to run at the time-slot rate especially if enough CPU bandwidth exists. If cycles are getting tight, a given time-slice can be further subdivided if necessary, for example in TS4, execution of I_{CMD} and RA could be done on a interleaved basis, i.e. run each one every other time, this effectively reduces the rate to 25 kHz and opens up more cycles for other processing if required.

TABLE VII. – SUMMARY OF TASKS AND FREQUENCIES FOR THE AC/DC RECTIFIER SYSTEM

Code Function	PWM rate (kHz)	Code execution rate (kHz)	Identifier
DC/DC-1 V Loop	200	200	V1
DC/DC-1 I Loop	200	200	I1
DC/DC-2 V Loop	200	100	V2
DC/DC-2 I Loop	200	100	I2
PFC I loop	100	50	IPfc
PFC V loop (done in BG)		50	VPfc
PFC $1/X^2$ (X=Vac Rect & Avg)		50	1/X2
PFC I-cmd (V1*V2*V3)		50	Icmd
PFC Vac Rect. and Average		50	RA
PFC I-balance		50	Ibal

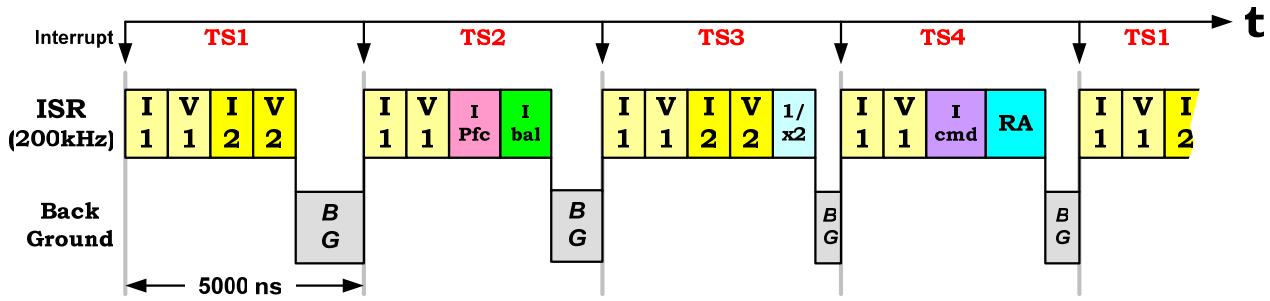


Fig. 18. Time-slice timing for the AC/DC rectifier.

Sample code (in assembly) showing a practical implementation of a 4 slot time-slicer.

```
;=========================================================
_ISR_Entry:    CONTEXT_SAVE   ; call save macro

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Code executed every Interrupt
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       User code....
       User code....
       User code....
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; Time Slice Manager (1:4)
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       MOVW    DP,#tsPtr
       MOVL    XAR7,@tsPtr    ; fetch current TS address
       LB      *XAR7          ; Jump there (indirectly)
;=========================================================
; TIME SLICE 1 - Code executed only on TS1
;=========================================================
TS1:   MOVL    XAR7,#TS2      ; Load TS2 address for
       MOVL    @tsPtr,XAR7    ; next time through
;=========================================================
       User code....
       User code....
       User code....
       LB      EXIT_ISR       ; Branch to ISR exit

;=========================================================
; TIME SLICE 2 - Code executed only on TS2
;=========================================================
TS2:   MOVL    XAR7,#TS3      ; Load TS3 address for
       MOVL    @tsPtr,XAR7    ; next time through
;=========================================================
       User code....
       User code....
       User code....
       LB      EXIT_ISR       ; Branch to ISR exit

;=========================================================
; TIME SLICE 3 - Code executed only on TS3
;=========================================================
TS3:   MOVL    XAR7,#TS4      ; Load TS4 address for
       MOVL    @tsPtr,XAR7    ; next time through
;=========================================================
```

```
        User code....
        User code....
        User code....

        LB      EXIT_ISR        ; Branch to ISR exit

;=============================================================
; TIME SLICE 4 - Code executed only on TS4
;=============================================================
TS4:    MOVL    XAR7,#TS1       ; Load TS1 address to
        MOVL    @tsPtr,XAR7     ; start over again
;=============================================================
        User code....
        User code....
        User code....

;=============================================================
EXIT_ISR        CONTEXT_REST    ; Restore context & return
                IRET
;=============================================================
```

Fig. 19. Practical time-slicer implementation using C28xx assembly.

C. *BG Structure Details Using State-Machine Techniques*

The BG loop (BGL) is responsible for many tasks (as many as 50~100 in some cases) and typically contains more than 90% of an applications software. As already discussed the best choice for the BGL is high level language such as C or C++.

The BGL needs to manage two main categories of tasks or functions:

- Local decision based/request (e.g. communications) based execution
- Periodic based execution

It is critical to run all tasks in a well structured and timely manner so the BGL rate is maximized and the periodic functions can execute at rates required by the application. This can be achieved in several ways, including the use of an operating system. In keeping with the "keep it simple" approach, an approach using state-machine based task scheduling will be used. With this approach tasks can execute on a time schedule (i.e. periodic) or based on decisions.

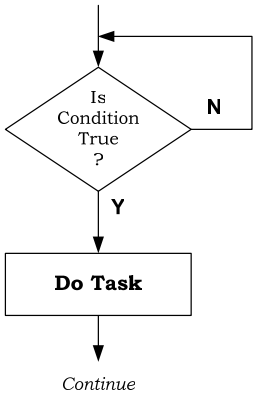
Decision Based Execution

One of the key concepts of this approach is to totally avoid using any type of "closed loop constructs", i.e. code that loops on itself waiting for a condition or state to be true before executing a task and the passing control to other tasks. Fig. 20 shows a flow chart and equivalent C example of what needs to be avoided.

Avoiding this is critical because of two main reasons:

- It is very wasteful on cycles.
- It can be very difficult or almost impossible to put a measure on the time of such a "wait" loop. Worst case, the condition may never happen, hence wait is "infinite"

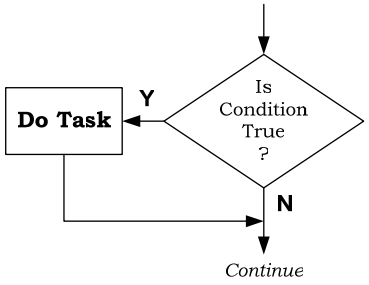
The key in a TDM system is to continually share the CPU cycles amongst all tasks as often as possible, this cannot be guaranteed using the above coding. A better approach is to check for the condition once every BGL time, if the condition is not true then control must pass over to next task/s. If all tasks or functions are designed to execute in this manner, then the BGL rate is maximized. The recommended approach is shown in Fig. 21 below.




```
while( condition != 1 )
{
      // wait
}

DoTask();
```

Fig. 20. Avoiding closed loop constructs.




```
if( condition = 1 )
{
      DoTask();
}
```

Fig. 21. Recommended approach for decision based software.

Often a task consists of multiple subtasks (or components) which can them selves be decision based or simply just "go execute". Keeping track of which subtask to execute and in what order, can get quite complex, especially if many main tasks have subtasks at various stages of completion. One very simple yet powerful technique to address this is by utilizing state-machine based execution. As an example let's examine the implementation of a soft-start function using this technique.

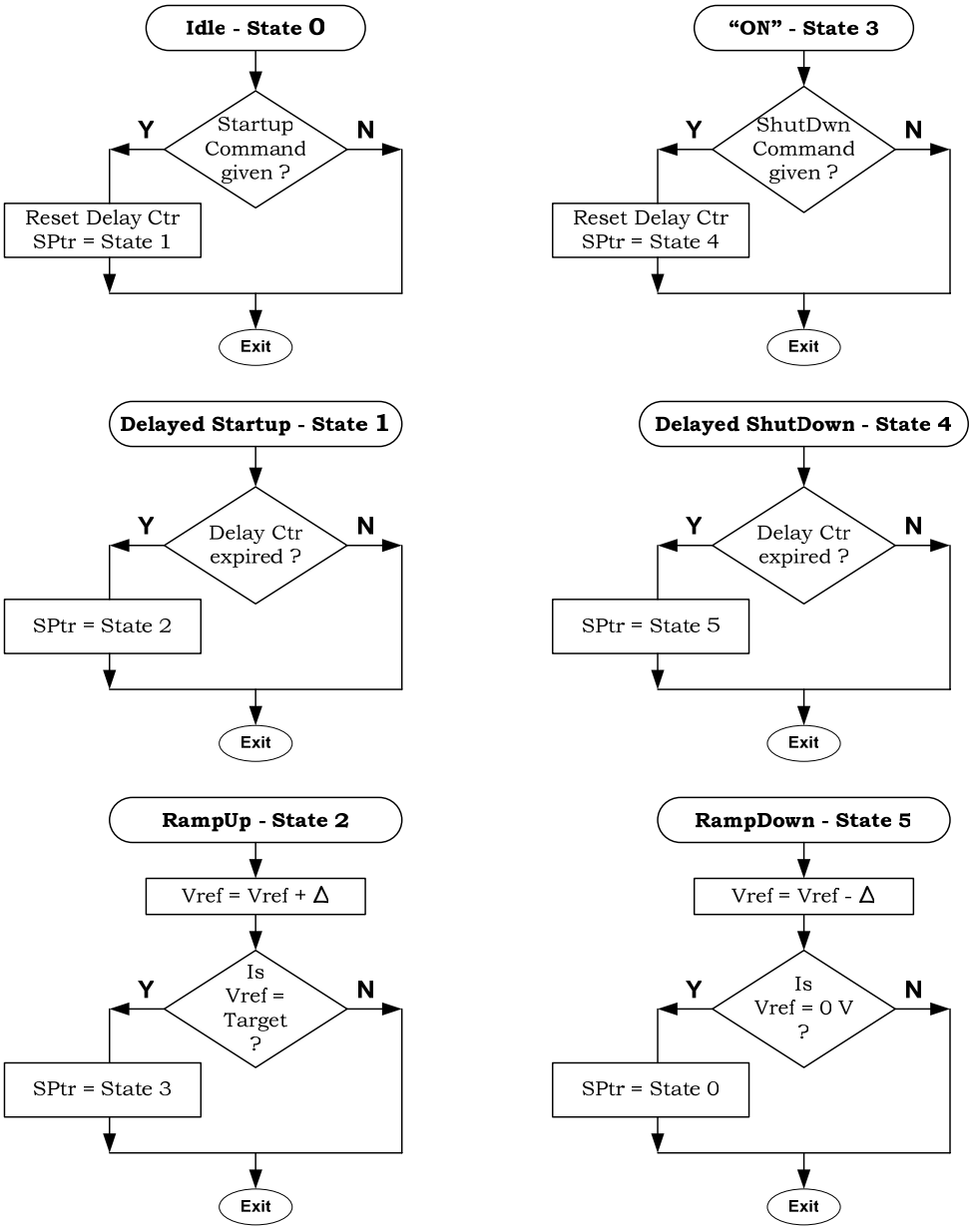


Fig. 22. Flow diagram utilizing state-machine based execution.

A State Pointer variable (SPtr) keeps track of the current state to be executed. Once per BG loop cycle, one of the states (0 – 6) is executed. The state-machine may keep it's state pointer set to one particular state for a long time, waiting for a certain condition to occur, for example a ShutDown command. If the required condition is met, the state pointer is set for the next state. Note, code always enters the state-machine only once per BGL, executes only one State and exits again, it does not loop waiting for any conditions to occur.

Implementing such state-machine control in C is very simple; one way is through the "case statement". Fig. 23 contains some sample code of the soft-start example.

```
//=====================================================================
void SoftStartFunction(void) // Soft Start via Vref control
//=====================================================================
{
	switch(StatePtr)
	{
		case 0: // Idle (wait for Start cmd) State
			if(StartUp == 1)
			{
				StatePtr = 1;
				DelayCtr = 0;
			} break;

		case 1: // Delayed Start-up state
			DelayCtr++;
			if(DelayCtr >= DelayUp)
			{
				StatePtr = 2;
			} break;

		case 2: // RampUp State
			Vref = Vref + RampUpRate;
			if(Vref >= VrefTarget)
			{
				Vref = VrefTarget;
				StatePtr = 3;
			} break;

		case 3: // "On" (at Target Voltage) state
			if(StartUp == 0)
			{
				DelayCtr = 0;
				StatePtr = 4;
			}
```

```
                case 4: // Delayed Shut-Down State
                        DelayCtr++;
                        if(DelayCtr >= DelayDown)
                        {
                                StatePtr = 5;
                        } break;

                case 5: // Ramp-down state
                        Vref = Vref - RampDownRate;
                        if(Vref <= 0)
                        {
                                Vref = 0;
                                StatePtr = 0;
                        } break;
        }
}
```

Fig. 23. Sample code of the soft-start example.

For a small number of state-machines and states with each machine, the case statement approach works fine. A typical power supply application however has a lot more going on in the BGL than just a single soft start controller.

For example a multi-output/multi-phase POL application may have the following BGL activities:

- Soft start and shutdown control for multiple outputs (V_{OUT1}, V_{OUT2}, V_{OUT3} and V_{OUT4})
- Phase current measurement scheduling
- Phase current balancing loop (slow 1~5 kHz)
- Temperature monitoring loop for each V_{OUT} and each phase (in multi-phase case)
- Reporting (V_{IN}, V_{OUT}, Temp, I_{PHASE}, etc) to host
- Self diagnostics for failure prediction
- Fault management – phase shedding (N→N-1) in case of phase fault/over-current
- Other etc.

Clearly what is needed is a method to support a multitude of state-machines each with possibly a large number of sub-states, all managed with minimal CPU cycles. The weakness with the case statement is when the number of cases is large, because every case needs to be evaluated up until the "true" case is found. If for example, the state-machine is positioned at a state towards the end, the number of evaluations can be large, and each needs to be calculated by the CPU every BGL time. A better method is to use a pointer to Function approach, where each function is a state or sub-state. In this approach the C compiler can use the indirect addressing capabilities of the CPU and hence the State function can be "jumped to" immediately based on a preloaded function address. In a complex system, this can help save a large amount of BGL cycles. A flow diagram of what this may look like is shown below.

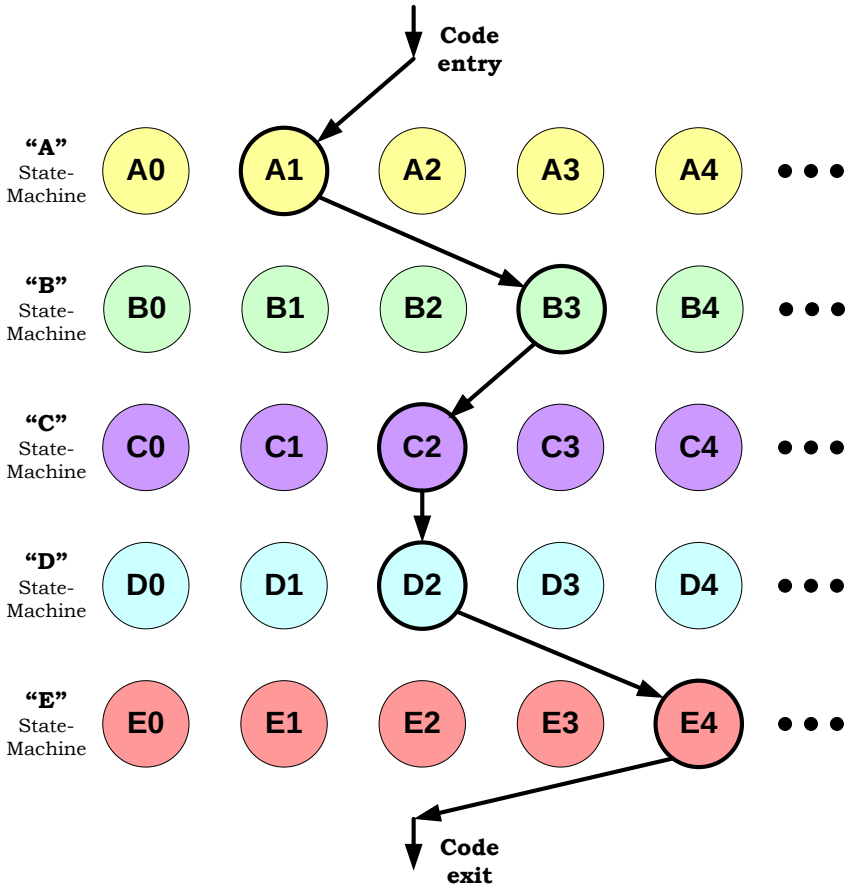


Fig. 24. Flow diagram for "pointer-to-function" based state-machine.

Here as many sub-states (1, 2, 3…) as needed can be added, each state-machine (A, B, C…) can have different number of sub-states. The number of state functions (A0, A1,…B0, B1,…) can be very large and a system could have a large amount of code, however, the path of execution each time round the BGL can be kept quite short.

To illustrate more clearly this method, the soft-start function has been implemented using a pointer-to-function based state-machine. In this case, the SS function could be any one of the 6 state-machines shown previously, i.e. A-tasks, B-tasks, etc. Here A-tasks have been used.

```
// State Machine function prototypes declarations
//=======================================================
// A task states
void A1(void); //state A1
void A2(void); //state A2
void A3(void); //state A3
void A4(void); //state A4
void A5(void); //state A5
void A6(void); //state A6

// Variable declarations
void (*A_Task_Ptr)(void);     // State pointer A tasks
void (*B_Task_Ptr)(void);     // State pointer B tasks

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Soft Start state-machine
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       (*A_Task_Ptr)();       // jump to an A Task (A1,A2,A3,...)

//=======================================================
void A1(void) // Idle (wait for Start cmd) - State 1
{
       if(StartUp == 1)
       { A_Task_Ptr = &A2; DelayCtr = 0; }
}
//=======================================================
void A2(void) // Delayed Start-up - State 2
{
       DelayCtr++;
       if(DelayCtr >= DelayUp)
       { A_Task_Ptr = &A3; }
}
```

```
//=========================================================
void A3(void) // RampUp - State 3
{
        Vref = Vref + RampUpRate;
        if(Vref >= VrefTarget)
        { Vref = VrefTarget; A_Task_Ptr = &A4; }
}
//=========================================================
void A4(void) // "On" (at Target Voltage) - State 4
{
        if(StartUp == 0)
        { DelayCtr = 0; A_Task_Ptr = &A5; }
}
//=========================================================
void A5(void) // Delayed Shut-Down - State 5
{
        DelayCtr++;
        if(DelayCtr >= DelayDown)
        { A_Task_Ptr = &A6;    }
}
//=========================================================
void A6(void) // Ramp-down - State 6
{
        Vref = Vref - RampDownRate;
        if(Vref <= 0)
        { Vref = 0; A_Task_Ptr = &A1; }
}
```

Fig. 25. Implementation of soft-start using a pointer-to-function based state-machine.

Periodic Based Execution in the BGL

In many cases it is very important to execute tasks periodically, and synchronous to a CPU time-base. Two examples of this are:

1. Slow control loop, such as a PFC voltage control. Here the sample rate (loop rate) can be quite slow, for example 1 kHz. Some timing jitter can be tolerated, but the sample/execution rate does need to be periodic.
2. "Tick time-base" for use in soft start. The state-machine driven soft-start function makes an assumption that it is executed at a fixed time interval, for example every 0.5 ms (2 kHz), this ensures that the ramp rates or delayed start-up required by an application can be accurately timed. For instance if a converter's voltage output needs to ramp to target value of 2.0 V in 20 ms, then the soft-start function needs to allow for 40 voltage increments (20 ms/0.5 ms) of V_{REF} adjustment, i.e. 2 V/40 = 50 mV steps.

Recall that the BG loop rate is an average value, and is valid only over longer time periods. An "instantaneous" rate cannot be guaranteed. However, if a periodic rate of programmable frequency is required, such as in the two examples, then it is possible to tie the execution of tasks (state-machines) to known time-bases. Anytime asynchronous software (like the BGL) is tied or synchronized to a fixed time-base, some jitter will be introduced. Most loops or processes can tolerate jitter and typically does not present a problem, however it must be comprehended.

The amount of jitter will be related approximately to the ratio of BG loop rate and synchronizing time-base. For instance, in a previous example, a BGLR of 70 kHz was calculated. If a 1-kHz time-base synchronized loop is needed it will show approximately 1/70×100% = 1.4% jitter. For a PFC voltage loop this should be a non issue. For the soft-start function this is certainly no problem and in fact a jitter of 10~15% could be tolerated, as the ramp-up rates are not critical, for example a ramp time of 20 ms ± 10% would be an acceptable power supply specification.

Usually a digital power supply controller (e.g. UCD911x or F280x/UCD9501) has spare timers available which can be utilized to provide an accurate time-base for task synchronization. Note not all the BGL software needs to be synchronized, it is possible, and desirable to have a mix of software tasks, some slaved off a time base, and others running as fast as the BG loop rate permits.

Implementing timed task execution requires programming a controller's timer to set a flag every time a period match event occurs. The BGL code then checks this flag every time through its loop (e.g. at a rate of 70 kHz) if set then a state-machine function (or several) is triggered to execute. If enough timer resources exist, several time bases can be working concurrently and multi-rate State-machine execution is possible. On the F280x/UCD9501 controller, three timers are available as part of the CPU, i.e. in addition to the DPWM and GP PWM timers.

Below is a flow chart of how BGL software might be implemented. Here 6 state-machines are running, three are synchronized to hardware timers and three are executing at the BG loop rate. A tasks execute at a period interval set by timer 1. A and B tasks execute at a period interval set by timer 2. The code sample given after the flow chart shows a possible C implementation of such a scheme using F280x/UCD9501 CPU timers as the synchronizing hardware.

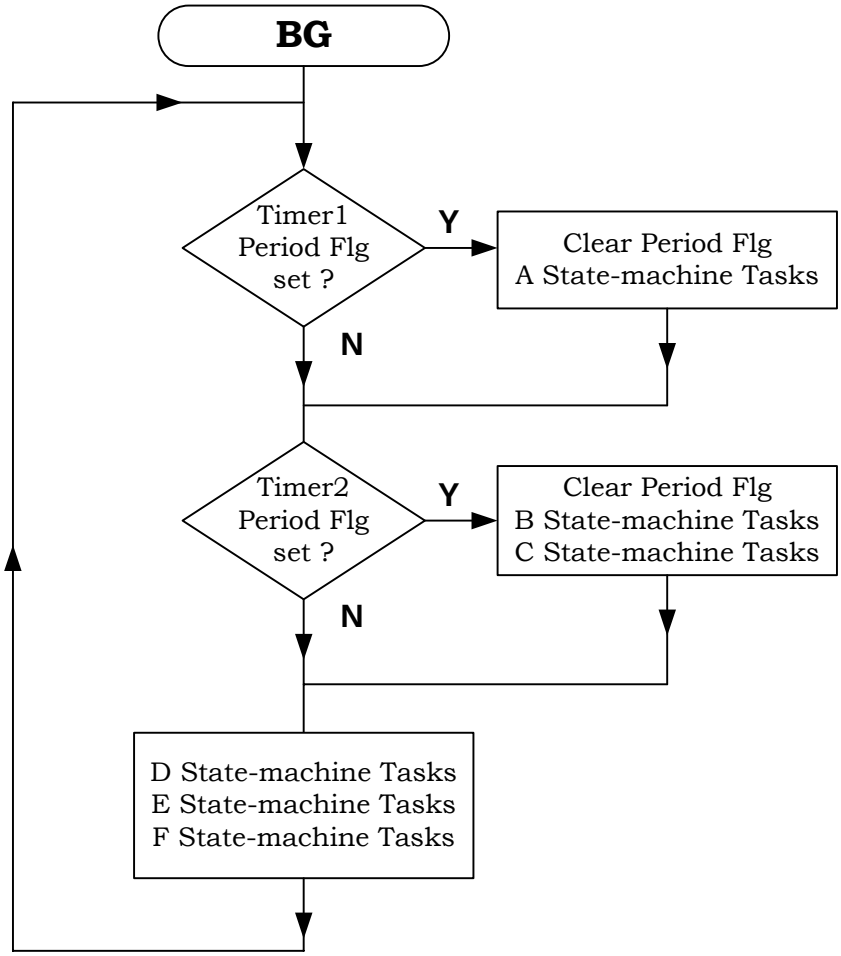


Fig. 26. Implementing BGL software.

```
//===========================================================
// State Machine Sync timers period initialization
//-----------------------------------------------------------
// Timing sync for background loops
CpuTimer0Regs.PRD.all = 40000;       // 400 uS (0.400 mS) - A tasks
CpuTimer1Regs.PRD.all = 100000;      // 1000 uS (1.000 mS) - B tasks
CpuTimer2Regs.PRD.all = 5000;        // 50 uS (0.050 mS) - C tasks


//===========================================================
// BG loop – Synchronous and Asynchronous Task execution
//-----------------------------------------------------------

                    ........
                    ........

       // loop rate synchronizer for A-tasks
       if(CpuTimer1Regs.TCR.bit.TIF == 1)
       {
              CpuTimer1Regs.TCR.bit.TIF = 1;       // clear flag
              //=========================================================
              (*A_Task_Ptr)();      // jump to an A Task (A1,A2,A3,...)
              //=========================================================
       }

       // loop rate synchronizer for B & C -tasks
       if(CpuTimer2Regs.TCR.bit.TIF == 1)
       {
              CpuTimer2Regs.TCR.bit.TIF = 1;       // clear flag
              //=========================================================
              (*B_Task_Ptr)();      // jump to a B Task (B1,B2,B3,...)
              (*C_Task_Ptr)();      // jump to a C Task (C1,C2,C3,...)
              //=========================================================
       }
       // Execute at the BG loop rate
       //=========================================================
       (*D_Task_Ptr)();      // jump to a D Task (D1,D2,D3,...)
       (*E_Task_Ptr)();      // jump to a E Task (E1,E2,E3,...)
       (*F_Task_Ptr)();      // jump to a F Task (F1,F2,F3,...)
       //=========================================================
```

Fig. 27. Timer synchronized implementation of BG loop software.