Symbian OS Internals/02. Hardware for Symbian OS

Symbian OS Internals Table of Contents

by Jason Parker

If it draws blood, it's hardware.
Unknown

This chapter explores the hardware that Symbian OS is designed to run on: a mobile phone. This is often known as the device platform. I'll examine the core hardware that is needed to run Symbian OS, and try to help you to appreciate the design choices that underlie a world-class Symbian phone. With this knowledge, I hope that you will also gain a deeper insight into the operating environment of Symbian OS.

Information on running Symbian OS on the EKA2 emulator, the platform that you will use during development, is placed in context throughout the book. The aim of this material is to let you see where you can rely on the similarities of the emulator to your phone hardware, and where you need to be aware of the differences.

Inside a Symbian OS phone

Symbian OS phones are designed first and foremost to be good telephones, with quality voice calls and excellent battery life. On top of that, Symbian OS phones are usually open platforms that provide opportunities for interesting and novel software. Achieving these goals requires hardware designed specifically for the task, high enough performance in the key use cases and an obsession for low power usage.

Looking into Symbian OS phone design, there are two complementary computing domains, the mobile radio interface of the baseband processor (BP), also known as the modem, and the application processor (AP), which runs the user interface and high-level code, under Symbian OS. Surrounding these domains is a collection of peripherals that make up the product: battery, display, speakers, SIM card and more.

Figure 2.1 depicts a common two-chip solution, where the BP and the AP are self-contained systems, with a high speed inter-processorcommunication (IPC) link between them. This is the preferred design for 3G phones, in which each domain can re-use existing software and hardware sub-systems.

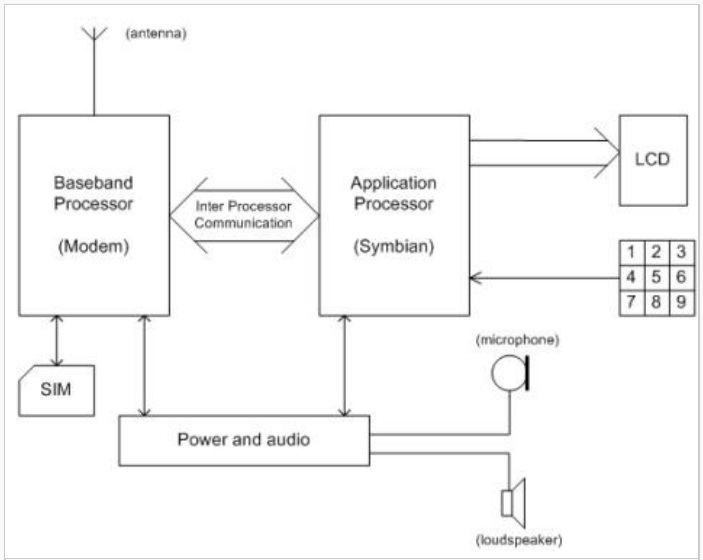

Figure 2.1 *A common two-chip solution*

The two-domain system of AP and BP isolates each processor from the other's requirements. The BP requires hard real-time software, periodic power management and provides security for the network. The AP expects to operate in two modes - full power when a user is interacting with the phone, and a deep sleep idle when nothing is happening. The AP code contains the frameworks and libraries for built-in applications and third-party code.

The overall quality of the product comes from the tight coupling between the two domains over the IPC, and their ability to co-ordinate the shared responsibilities of audio and power.

Designing hardware for Symbian OS phones requires a clear understanding of the end-user use cases, the performance requirements that these put on the design, and a continual focus on power management.

Baseband processor (BP)

The baseband processor is the voice and data modem for the phone. It contains all the electronics required for the radios used in 2.5G and 3G telephony, a DSP to run the algorithms to decode the information, and a CPU to run the call control stack, which co-ordinates with the network base stations and communicates with the AP.

The software on the BP is called the telephony stack, and known as the stack for short. The stack is a complex system of code that has grown in step with the evolving telephony standards and their type certification regimes. A typical stack will contain between 2 and 8 MB of code, and require up to 2 MB of working RAM to execute. GSM calls are scheduled to a 4.6 ms time frame, in which all of the call activity needs to have completed before the start of the

(C) Copyright Nokia 2013. All rights reserved.

next frame. This requires a real-time operating system (RTOS) environment. It is tuned and tested to meet the stringent timing requirements under worst case loads of voice and data calls.

BP power management is highly optimized for maximum call time and idle time, whilst still being connected to the network. When the phone is idle, the BP can put itself into a deep power-saving mode, only waking up every two seconds to listen to the paging channel for an incoming call or message.

The IPC interface to the AP has evolved from simple serial ports in early Symbian phones to 10 Mb/s multiplexed links. This link could use five bi-directional channels for telephony control, system control, packet data, streamed data and BP debug.

Audio data is routed to and from the BP through a dedicated digital audio bus, directly to the audio hardware. This bus provides minimal latency with guaranteed real-time performance and lower power consumption during a call by bypassing the AP. If voice call data was passed over the IPC to the AP, additional buffering would be incurred, a real-time load would be placed on the AP, and power consumption would go up.

The BP controls the SIM card, which contains the secret codes and algorithms required for network authentication.

The two-domain system of AP and BP provides many technical and engineering benefits, including design re-use, stability and security. These come at the cost of additional chips, physical size, and overall power consumption.

There are strong financial pressures towards the closer integration of the AP and BP domains for mid-tier phones. Example designs range from multiple cores on one ASIC, sharing memory but little else, up to the full integration of the telephony stack and Symbian OS. In this last case, the two co-exist on the same CPU, with all of the software integration issues that this incurs.

As you can see, baseband processors and their sophisticated telephony stacks are major topics that already fill several books on their own.

Application processor (AP)

The application processor is at the heart of a Symbian OS phone. Contained on a single piece of silicon, the AP is an example of a System-on-Chip. It has an ARM CPU, memory, display and audio interfaces, multimedia accelerators and many more peripherals. I will now focus on these components, their integration and how they interact with each other.

System-on-Chip (SoC)

SoCs are known by two other names: ASICs (Application-specific Integrated Circuits) for custom chips and ASSPs (Application-specific Semiconductor Parts) for commercial parts. All three terms are used imprecisely and interchangeably. SoCs are designed and manufactured by all of the major silicon companies: Texas Instruments have a product family called OMAP and Intel have their `XScale` range of processors. Figure 2.2 shows a typical System-on-Chip design.

Each sub-component within the SoC is an intellectual property (IP) block. The blocks are linked to interconnecting buses through industry standard interfaces. The IP blocks can be licensed from many sources. The most well known IP licensing company is ARM Ltd, who license ARM CPU cores.

The example SoC is driven by an ARM 926 CPU for Symbian OS, and a DSP for multimedia codecs. These two cores are both masters on the system bus, which is a high-speed, low-latency, 32-bit wide bus, connected to the DRAM controller. The system bus and memory controller funnel all data accesses into main memory, so they must be designed for high bandwidth and low latency transfers to avoid starving the CPU and reducing its effective performance.

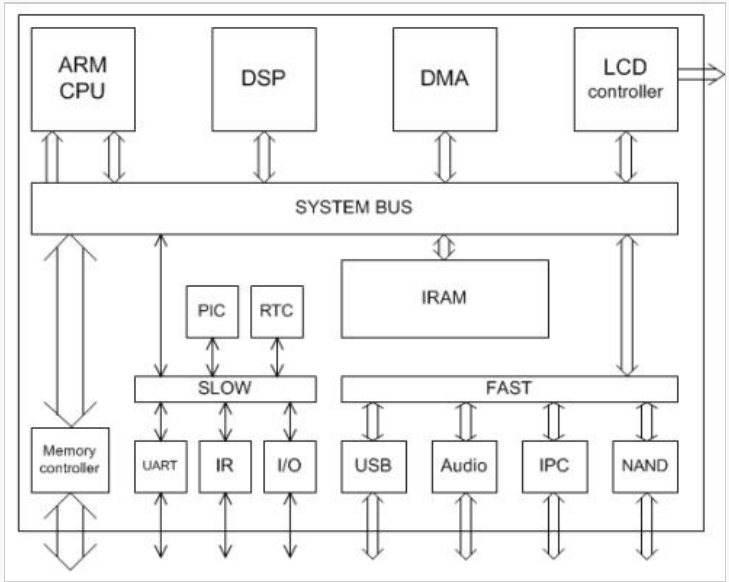

Figure 2.2 *System-on-Chip*

The DMA engine and LCD controller are additional bus masters, both accessing memory through the same bus. The remaining peripheral blocks are known as bus slaves - they cannot access memory directly, and require their masters to feed them with commands and data. The slave blocks are cascaded off the two peripheral buses, one of which is a relatively fast bus for DMA devices, and the other is a slow bus for simple low-bandwidth peripherals.

Peripheral buses are connected to the system bus through bus bridges. These translate between the bus formats and compensate for any speed differences, necessary since peripheral buses normally run slower than the system bus. A good SoC design will pay attention to these bridges to ensure that critical timed or high-bandwidth peripherals can be accessed quickly by the CPU.

Further information about ARM SoCs can be found in the book, ARM System-on-Chip Architecture by Steve Furber (*ARM System-on-Chip Architecture*, 2nd edn by Steve Furber. Addison-Wesley Professional).

(C) Copyright Nokia 2013. All rights reserved.

Physical memory map

The buses and their connections determine the physical address map of the chip - with 32-bit addressing there is 4 GB of addressable space. Symbian OS uses the CPU's Memory Management Unit (MMU) to remap the reality of the chip's address space layout into a consistent virtual address space for the software.

As an example, the 4 GB address space of the SoC might be divided into large regions by the system bus controller. By only decoding the top three address bits, it produces eight regions, each 512 MB in size:

Address start	**Address end**	**Content**
0x00000000	0x1FFFFFFF	ROM Bank 0 (Boot Rom)
0x20000000	0x3FFFFFFF	ROM Bank 1
0x40000000	0x5FFFFFFF	DSP
0x60000000	0x7FFFFFFF	Fast Peripheral Bus
0x80000000	0x9FFFFFFF	Slow Peripheral Bus
0xA0000000	0xBFFFFFFF	IRAM
0xC0000000	0xDFFFFFFF	DRAM Bank 0
0xE0000000	0xFFFFFFFF	DRAM Bank 1

These large regions are then further sub-divided. With 32 MB of RAM installed in DRAM Bank 0, the remaining 480 MB of address space will contain aliases of the RAM contents if address bits 25 to 28 are not decoded by the hardware, as is typical:

0xC0000000	0xC1FFFFFF	32 MB RAM
0xC2000000	0xC3FFFFFF	32 MB Alias 1
0xC4000000	0xC5FFFFFF	32 MB Alias 2
...	...	...
0xDE000000	0xDFFFFFFF	32 MB Alias F

The peripheral bus's regions are sub-divided by their peripherals. The fast peripherals in the example SoC each have 64 KB of address for their register sets:

0x60000000	0x6000FFFF	NAND Interface
0x60010000	0x6001FFFF	IPC
0x60020000	0x6002FFFF	Audio
0x60030000	USB	
...	...	...
0x600x0000	0x7FFFFFFF	Empty Space

In practice, every ASSP will have a different physical address space and most of it will be unused or aliased. Reads and writes to unused space will produce a bus error.

A good design will have a consistent memory map for all of the bus masters, removing the need for any physical address translation, and reducing the likelihood of errors.

Normally the CPU will have access to every peripheral in the system, whereas the other masters will only have visibility of appropriate slaves. The LCD controller needs to pull frame data from memory, and the DMA engine will work between the fast peripherals and memory.

The physical address map is used by the bootstrap code when configuring the MMU. The DMA engine and the other bus masters will not contain their own MMUs. They only understand physical addresses and software that programs these devices must translate virtual addresses back to their physical values before using them.

Central Processing Unit (CPU)

Symbian OS requires a 32-bit microprocessor that combines high performance with low power consumption. It must be little endian, with a full MMU, user and supervisor modes, interrupts and exceptions. ARM designs fit this bill exactly and when this book was written all Symbian OS phones had an ARM-based CPU, as did 80% of the world's mobile phones.

To take these requirements in turn:

High performance is a relative term for a battery-powered device. Symbian phones today have CPUs clocked between 100 and 200 `MHz`, which is more than an order of magnitude slower than an average 3 `GHz` PC - yet the power they consume is three orders of magnitude less. Future application demands will push the CPU speeds into the 300 to 400 `MHz` range for peak performance.

Low power consumption is a design requirement for all components in a Symbian OS phone. During the CPU design, engineering trade-offs are evaluated and features are added to produce the most power-efficient core. Power saving comes from lean hardware design, the selective clocking of circuits and the addition of multiple low-power modes: Idle, Doze, Sleep, Deep Sleep and Off. I will discuss the mapping from hardware into the software frameworks that Symbian OS provides for power management in Chapter 15, *Power Management*.

The MMU, with the user and supervisor modes the CPU provides, allow for the virtualization of the user memory. EKA2 constructs a robust execution environment for applications, each isolated from the others with its own protected memory space. Application code executes in user mode with limitations and kernel code uses supervisor mode with fewer limitations, but even kernel code is still constrained by its virtual memory map. I describe the memory models that Symbian OS provides in Chapter 7, *Memory Models*.

Exceptions are CPU events that change the instruction flow in the core. Interrupt exceptions are generated by peripherals that are seeking attention. Software interrupts are used as a controlled switch from user to supervisor mode. The MMU will generate a data abort if code tries to access memory that is not mapped, and a prefetch abort if the CPU jumps to a code address that is not mapped. See Chapter 6, *Interrupts and Exceptions*, for more on interrupts and exceptions.

ARM

ARM have been designing RISC-based CPUs for over 20 years, and successfully licensing them to all of the world's semiconductor manufacturers for inclusion into their own SoCs. Intel has licensed version 5 of the ARM architecture to build the software-compatible `XScale` microprocessor.

As ARM developed successive generations of CPUs, they have added new instructions and features, and deprecated some rarely used old features. The ARM architectural version number, with some additional letters, defines the feature set. It specifies the instruction set, the operation of the MMU, the caches and debugging.

(C) Copyright Nokia 2013. All rights reserved.

Symbian OS requires a CPU that supports ARM v5TE or greater. ARM v5TE is the baseline instruction set for all Symbian software. To ensure compatibility across multiple phones, application code should only use v5TE instructions. (The previous generation of EKA1 phones used the ARM v4T architecture.)

What does ARM v5TE actually mean? It is version 5 of the ARM architecture, with the THUMB instruction set and the Enhanced DSP instructions. The definition of the ARM v5TE instruction set can be found in the *ARM Architecture Reference Manual*.

ARM cores and SoCs that are currently compatible with Symbian OS phone projects include:

Core	Architecture	SoC
ARM926	v5TE	Philips Nexperia PNX4008
ARM926	v5TE	Texas Instruments OMAP 1623
Xscale	v5TE	Intel XScale PXA260
ARM1136	v6	Texas Instruments OMAP 2420

THUMB was introduced in architecture v4T. It is a 16-bit sub-set of the ARM instruction set, designed to resolve the common RISC issue of poor code density with instructions that are all 32 bits. By using a 16-bit encoding scheme, THUMB compiled code is approximately 70% of the size of the ARM equivalent, but it needs 25% more instructions to do the same task. THUMB and ARM code can inter-work on the same system, using the BLX instruction to switch mode.

Most code in Symbian OS is compiled as THUMB, since the size of the ROM is intimately linked to the cost of a Symbian OS phone. The kernel is built for ARM for increased performance and it requires instructions which are missing from THUMB, such as coprocessor instructions needed to access the MMUand CPU state control registers. Application code can be built for ARM by adding ALWAYS_BUILD_AS_ARM into the application's MMP file. Symbian does this for algorithmic code, since, for example, the JPEG decoder runs 30% faster when compiled for ARM.

The enhanced DSP instructions enable the more efficient implementation of 16-bit signal processing algorithms using an ARM CPU. These instructions are not used in normal procedural code and have little impact on the execution of Symbian OS.

Memory Management Unit (MMU)

Symbian OS requires a full Memory Management Unit to co-ordinate and enforce the use of virtual memory within an open OS. I discuss the use of the MMU in Chapter 7, *Memory Models*.

The MMU sits between the CPU and the system bus, translating virtual addresses used by software into physical addresses understood by the hardware. This lookup has to happen on every memory access.

The MMU breaks up the flat contiguous physical memory into pages. Mostly they are small, 4 KB pages, although larger 64 KB pages and 1 MB sections exist.

The Symbian OS virtual memory map re-orders scattered physical pages into an apparently ordered virtual memory space. The re-ordering information is expressed to the MMU through the use of page tables. Page tables are a hierarchical data structure that encodes the entire 4 GB virtual address space in a sparse manner. On ARM systems the table has two levels, the Page Directory and Page Tables.

In much the same way as a physical address is decoded by the bus controllers, the bit fields within a virtual address are decoded by the MMU into the Page Directory, the Page Table and index into the memory page. This is explained in detail in Section 7.2.1.

Bits	31 - 20	19 - 12	11 - 0
Address decode	Top 12 bits map to Page Directory	Middle 8 bits map to Page Table	Bottom 12 bits are offset in page

Virtual address decoding for a small page

On every memory access the MMU performs a virtual to physical lookup to get the correct bus address and to validate page permissions. The process of looking up a physical address from a virtual one is known as *walking* the page tables. This takes the time required for two memory accesses, the read of the Page Directory followed by the Page Table read.

To speed up the address translation the MMU caches recently looked up results within a Translation Look-aside Buffer (TLB). If a virtual to physical lookup cannot be found in a TLB, the MMU has table walking hardware to perform a new virtual lookup and it will save the result into a TLB entry.

The TLBs must be flushed if the page tables are changed. This can happen on a context switch, during debug or the unloading of code.

At startup the CPU will run using physical addresses and the bootstrap code will build the initial set of page tables. When the MMU is turned on the CPU will switch into virtual memory operation, ready for the kernel boot sequence.

Caches

The CPUs used in every Symbian phone require caches to achieve optimum performance. The job of a cache is to insulate the fast CPU from its slower memory system by holding local copies of recently accessed data or instructions.

ARM CPUs have Harvard architectures, with separate instruction and data ports, resulting in separate instruction and data caches (ICache, DCache).

Caches work by taking advantage of the repetitive local characteristics of executing code. Code that is in a loop will execute the same instructions and access the same or similar data structures. As long as the CPU's view into memory is consistent, it does not care where the data is located at any instant in time - whether it is found in RAM or in the cache. However, the kernel does not cache memory mapped peripherals because the controlling software requires strict ordering of reads and writes into the peripheral registers.

Caches are organized in cache lines, which typically contain 32 bytes of data from a 32-byte aligned address, and a tag to store this source address. A 16 KB cache would contain 512 lines.

When the CPU requests data, the cache tests to see if it has a line that matches the requested address. If it does, the data is returned immediately - this is called a cache hit. If the cache does not contain the data, then a cache miss has occurred. After a miss, there will be a cache line fill of the required data from the memory system, and then the CPU will continue execution. To make space for this new cache line, an older cache line will be evicted from the cache.

 (C) Copyright Nokia 2013. All rights reserved.

The efficiency of a cache is measured by its hit ratio - the ratio of cache hits to total accesses. On Symbian OS, the approximate numbers are 95% for the `ICache` and 90% for the `DCache`.

With a high hit rate, the CPU can run close to its maximum speed without being stalled by the memory system. This is good for performance and even better for battery life, since a cache hit requires substantially less power than an external memory access.

Line fill operations are optimized for the memory system, to take advantage of sequential burst modes from the memory chips. Once the MMU has been set up and the cache enabled, all of its operation is automatic and the system runs faster. EKA2 only needs to issue cache control commands if the caches need to be flushed. This can be due to any of the following:

- A memory map change on a context switch
- New code being loaded or existing code discarded
- Self-modifying code generation
- Using DMA from cached memory.

Virtual and physical caches

The CPU is isolated from the system bus by the MMU and caches, so their order of operation has an impact on Symbian OS (see Figure 2.3).

When the CPU is connected directly to the cache, the cache uses virtual addresses, and it in turn talks through the MMU to generate physical addresses on the system bus (left-hand side of Figure 2.3).

A physically addressed cache will be connected directly to the system bus and will be indexed with real physical addresses. The virtual to physical lookup will occur in the MMU, which is located between the CPU and the cache (right-hand side of Figure 2.3).

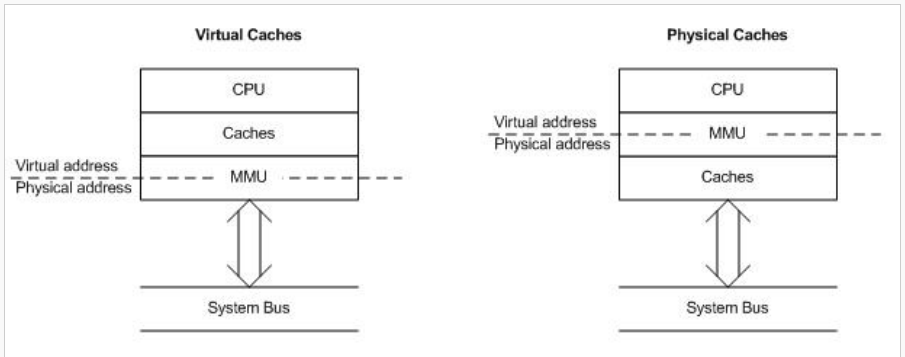

Figure 2.3 *Virtual and physical caches*

ARM v5 virtual cache

In ARM v5 and earlier systems, the MMU was placed outside the CPU and caches, resulting in caches that worked within the virtual memory domain of the MMU.

In this design, all data stored in the cache is indexed by its virtual address, so when the CPU requests data with a virtual address, no MMU translation is required for a cache hit. If there is a miss, the MMU is invoked to generate a physical bus address for the cache miss. This design reduces the workload on the MMU and saves power.

The downside to virtual caches is that the kernel has to empty them every time it modifies the page tables. The kernel can invalidate the `ICache` (discard it) but it has to flush all of the dirty data in the `DCache` to the memory system.

Unfortunately, the Symbian OS moving memory model modifies the virtual address map when it performs inter-process context switch. This means that a cache flush is needed that can take many milliseconds, resulting in a significant loss of system performance. In Section 7.2.1, you will see that Symbian OS uses a technique called fixed processes to minimize these cache flushes.

ARM v6 physical cache

ARM architecture v6 delivered a whole new MMU, with new page table layout, physical caches, process ASIDs, support for level 2 caches and IO memory space.

An ARM v6 CPU, such as the 1136, uses physical addresses at all times in the cache. This requires the MMU to work harder by performing a virtual to physical lookup on every request from the CPU core. This will normally be through a TLB lookup.

The advantage to this scheme is that the caches are always in sync with the physical memory map, no matter how the virtual map changes. This removes the cache flush penalty from context switches.

To further improve performance, the v6 MMU model introduced Application Space Identifiers, known as ASIDs. Each ASID has its own 4 or 8 KB PDE, for the bottom 1 or 2 GB of address space. Changing the ASID will instantly swap out this address space.

As I will explain in Section 7.4.2, EKA2 assigns an ASID to every process it creates, resulting in extremely fast context switches.

Instruction cache (ICache)

The `ICache` contains recently executed instructions, ready for their re-use. Instructions cannot be modified in the `ICache`, so it can be treated as a read-only device.

When a line needs to be evicted on a cache miss, it is immediately overwritten by the new instructions. This is permitted, since it is read-only and cannot have changed. Cache flushing operations are only needed when code is unloaded from the system and to ensure the coherency of generated or self-modifying code.

Code compiled for the THUMB instruction set gains an advantage over ARM by being able to fit twice as many instructions into the `ICache`. This helps to offset its slower performance.

(C) Copyright Nokia 2013. All rights reserved.

Data cache (DCache)

When the CPU is reading data, the DCache works in the same way as the ICache. Data hits within the cache are returned immediately and missed data will be sourced from main memory, replacing a recently evicted line.

The complexity comes with data writes into the DCache and the combinations of strategies to return it to memory. With write-through caching, every time the CPU writes data, it will be immediately written out to memory, through the write buffer, and the data will update the cached copy if it hits.

Write-through caching ensures that memory always stays coherent with the cache. Cache line evictions and cache cleaning operations do not need to write anything back to memory, enabling them to discard lines at will.

The downside is that the system bus has to support the full write speed of the CPU and the cache is only being effective for reads.

Symbian OS uses write-through caching for the LCD frame buffer, to ensure consistent pixels on the display. Writes of new pixel data will gain a small speed-up because of the write buffer, and read-modify-write operations will be aided by the cache. But running all of Symbian OS in a write-through cached system reduces performance by over 30%.

To make full use of the DCache for writes as well as reads, Symbian OS uses a scheme called copy-back. Write hits into the cache remain in the cache, where they may be overwritten again. Copy-back results in a massive reduction of write traffic to memory by only writing data when it is evicted.

Cache lines that have been modified are tagged by dirty bits - normally two of them for a pair of half lines. When it is time to evict the cache line, the dirty bits are evaluated, and dirty half lines are written back out to memory through the write buffer. Half lines are used to reduce the write bandwidth overhead of unmodified data, since clean data can be discarded.Flushing the entire contents of a copy-back cache will take some milliseconds, as it may be populated entirely with dirty data.

Random Access Memory (RAM)

Random Access Memory (RAM) is the home of all the live data within the system, and often the executing code. The quantity of RAM determines the type and number of applications you can run simultaneously, the access speed of the RAM contributes to their performance.

A Symbian OS phone will have between 8 and 64 MB of RAM. The OS itself has modest needs and the total requirement is determined by the expected use cases. Multimedia uses lots of RAM for mega-pixel cameras images and video recording. If NAND Flash memory is used, megabytes of code have to be copied into RAM, unlike NOR flashes that execute in place.

The RAM chip is a significant part of the total cost of a phone, both in dollar terms, and in the cost of the additional power drain required to maintain its state.

It is not only the CPU that places heavy demands on the memory subsystem; all of the bus master peripherals read and write into RAM too. Their demands and contention have to be considered during the system design. Working out real-world use cases, with bandwidth and latency calculations, is essential for understanding the requirements placed on the memory system.

Mobile SDRAM

In the last few years, memory manufacturers have started to produce RAM specifically for the mobile phone market, known as Low Power or Mobile SDRAM.

This memory has been optimized for lower power consumption and slower interface speeds of about 100 MHz, compared to normal PC memory that is four times faster.

Mobile memories have additional features to help maintain battery life. Power down mode enables the memory controller to disable the RAM part without the need for external control circuitry.

Data within a DRAM has to be periodically updated to maintain its state. When idle, DRAMs do this using self-refresh circuitry. Temperature Compensated Self Refresh (TCSR) and Partial Array Self Refresh (PASR) are used to reduce the power consumption when idle. The combination of TCSR and PASR can reduce the standby current from 150 to 115 µA.

Internal RAM (IRAM)

Memory that is embedded within the SoC is known as internal RAM (IRAM). There is much less of this than main memory.

When booting a system from NAND Flash, core-loader code is copied from the first block of NAND into RAM. IRAM is an ideal target for this code due to its simple setup. Once the core-loader is running from IRAM, it will initialize main RAM so that the core OS image can be decompressed and copied there.

IRAM can be used as an internal frame buffer. An LCD controller driving a dumb display needs to re-read the entire frame buffer 60 times a second. Thus a 16-bit QVGA display will require 8.78 MB of data in one second. By moving the frame buffer into IRAM, the system can make a large saving in power and main memory bandwidth. A dedicated IRAM frame buffer can be further optimized for LCD access and to reduce its power needs.

IRAM can also be useful as a scratch pad or message box between multiple processors on the SoC.

Note that putting small quantities of code into IRAM does not speed up the execution of that code, since the ICache is already doing a better and faster job.

Flash memory

Symbian phones use Flash memory as their principal store of system code and user data. Flash memory is a silicon-based non-volatile storage medium that can be programmed and erased electronically.

(C) Copyright Nokia 2013. All rights reserved.

The use of Flash memory is bound by its physical operation. Individual bits can only be transformed from the one state into the zero state. To restore a bit back to a one requires the erasure of a whole block or segment of Flash, typically 64 KB. Writing a one into a bit position containing a zero will leave it unchanged.

Flash memory comes in two major types: NOR and NAND. The names refer to their fundamental silicon gate design. Symbian OS phones make best use of both types of Flash through the selection of file systems - I will explain this in detail in Chapter 9, *The File Server*.

The built-in system code and applications appear to Symbian software as one large read-only drive, known as the Z: drive. This composite file system is made up of execute in place (XIP) code and code that is loaded on demand from the Read Only File System (ROFS). The Z: drive is sometimes known as the ROM image.

User data and installed applications reside on the internal, writable C: drive. The C: drive is implemented using one of two different file systems: LFFS (Log Flash File System) for NOR or a standard FAT file system above a Flash Translation Layer (FTL) on top of NAND.

A typical Symbian phone today will use between 32 and 64 MB of Flash for the code and user data - this is the total ROM budget.

Symbian uses many techniques to minimize the code and data sizes within a phone, such as THUMB instruction set, prelinked XIP images, compressed executables, compressed data formats and coding standards that emphasize minimal code size.

NOR Flash

NOR Flash is used to store XIP code that is run directly by the CPU. It is connected to a static memory bus on the SoC and can be accessed in random order just like RAM. The ARM CPU can boot directly out of NOR Flash if the Flash is located at physical address zero (0x00000000). For user data, Symbian uses the Log Flash File System (LFFS) on top of

NOR Flash. This file system is optimized to take advantage of NOR Flash characteristics. I describe LFFS in detail in Chapter 9, *The File Server*.

NOR flashes allow for unlimited writes to the same data block, to turn the ones into zeros. Flashes usually have a write buffer of around 32 to 64 bytes that allows a number of bytes to be written in parallel to increase speed. A buffered write will take between 50 and 600 µs depending on the bit patterns of the data already in the Flash and the data being written. All zeros or all ones can be fast, and patterns such as 0xA5A5 will be slow.

Erasing a NOR segment is slow, taking about half a second to one second. But erases can be suspended and later restarted - LFFS uses this feature to perform background cleanup of deleted data while remaining responsive to foreground requests.

Completed writes and erases will update the status register within the Flash, and may generate an external interrupt. Without an interrupt, the CPU will need to use a high-speed timer to poll the Flash for completion.

By using NOR flashes with Read-While-Write capabilities, it is possible to build a Symbian OS phone with one NOR part containing XIP code and LFFS data.

NAND Flash

NAND Flash is treated as a block-based disk, rather than randomly addressable memory. Unlike NOR, it does not have any address lines, so cannot appear in the memory map. This means that code cannot execute directly from NAND and it has to be copied into RAM first. This results in the need for extra RAM in a NAND phone compared to a similar NOR device. NAND Flash writes are about 10 times faster than those on NOR Flash.

A phone cannot boot directly from NAND. The process is more complex, requiring a set of boot loaders that build upon each other, finally resulting in a few megabytes of core Symbian OS image, the ROM, being loaded into RAM, where it will execute.

NAND is also inherently less reliable than NOR. New parts will come with defective blocks and are susceptible to bit errors during operation. To alleviate the second problem, an Error Correction Code (ECC) is calculated for every page of data, typically 512 bytes. On every read, the ECC is re-calculated and the difference between it and the stored ECC value is used to correct single-bit errors, at runtime. Multi-bit errors cannot be recovered and the page is considered corrupt.

The lower price of NAND compared to NOR makes it attractive for mass-market phone projects, even after taking into account the extra RAM required.

NAND Flash parts are attached to the SoC using a dedicated interface that is connected directly to the NAND chip pins through an 8-bit or 16\- bit bus. The interface block will use the DMA interface for data transfers and contains circuits to calculate the ECC on writes and reads.

The NAND interface reads and writes into the NAND Flash using pages of data. A small block NAND device uses 512-byte pages, and a large block device uses 2048-byte pages. Data is erased by block, where a block will contain 32 or 64 pages.

Interrupts

Peripherals in the system demand attention from the CPU by generating interrupts. Every peripheral will have one or more interrupt lines attached to the Programmable Interrupt Controller (PIC), which in turn will funnel the outstanding interrupts into the CPU. ARM cores only have two interrupt inputs, the normal Interrupt ReQuest (IRQ) and the Fast Interrupt reQuest (FIQ). The FIQ has higher priority than IRQ and an additional set of banked registers.

The EKA2 interrupt dispatch code determines the source of an interrupt by reading the PIC registers, and then calls the correct service function. This is all explained in Chapter 6, *Interrupts and Exceptions*.

In Figure 2.4, you can see a 62 interrupt system, incorporating a two-layer design that re-uses the same PIC block to cascade from level 2 into level 1.

 (C) Copyright Nokia 2013. All rights reserved.

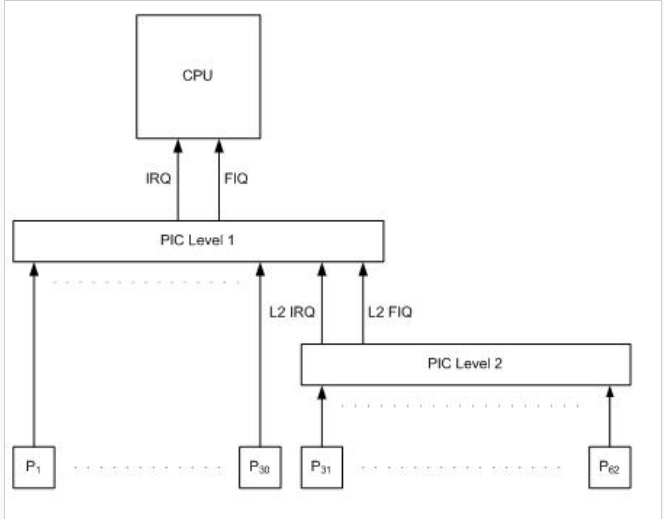

Figure 2.4 *Two-layer interrupt controller*

Each interrupt within a layer will be represented by one of the 32 bits inside the controlling register set. The registers allow the interrupt dispatch software to configure, enable and detect the correct interrupt:

Interrupt Type	**IRQ**	**FIQ**
Output Status	True	False
Enabled	True	False
Latched Interrupt Input	True	False
Detect Type	Edge	Level
Polarity	Rising Edge/High Level	Falling Edge/Low Level

Examples of interrupt types in use include:

A serial port output FIFO will have a half-empty setting, generating a high level whenever there is space within the FIFO. Once enough data has been written into the FIFO by an interrupt service routine, the interrupt output will drop back.

A rising edge interrupt would be used to detect the short VSync pulse generated by the LCD controller on the start of a new output frame.

To determine the current pending interrupt, the software dispatcher must read the status and enable registers from the PIC, AND them together, and look for a set bit to determine which interrupt to dispatch. The bit number is then used to index a table of function pointers to the correct interrupt service routine (ISR). By putting the highest priority interrupt in the upper bits of level 1, the first valid bit can be found quickly using the count leading zeros (CLZ) instruction.

The interrupt dispatch latency is the time between the IRQ input being raised and the execution of the first instruction in the ISR, and is the time taken up by the software dispatcher described in the previous paragraph and the thread preamble code. It will run for tens of instructions, resulting in about a 50-cycle interrupt cost, or 250 ns on a 200 MHz core. The total overhead of an interrupt, once the post-amble code and kernel dispatcher is taken into account, will be approaching 1 µs.

You can use more sophisticated PICs to reduce the interrupt latency. Bit patterns are replaced by a status register containing the highest priority interrupt number for the dispatch software to use immediately.

The ARM vectored interrupt controller (VIC) is an even more complex system, in which the CPU has a dedicated VIC port. It allows for the ISR address to be injected directly into the CPU, removing the overhead of software interrupt dispatch, and saving a few tens of cycles per interrupt. Symbian OS phones do not require a VIC and its associated silicon complexity, as they do not have hard real-time interrupts with nanosecond latencies.

When designing a phone you should aim to minimize the number of active interrupts within the system, as this will increase the overall system interrupt response robustness and reduce power consumption. This can be done by using DMA interfaced peripherals, and by not polling peripherals from fast timers.

Timers

In Chapter 5, *Kernel Services*, I will explain EKA2's use of the millisecond timer. EKA2 uses a 1 ms tick timer to drive time slicing and the timer queues, and to keep track of wall clock time.

The minimum hardware requirement is for a high-speed timer capable of generating regular 1 ms interrupts without drifting. The timer counter needs to be readable, writable, and the maximum cycle period should be many seconds.

The speed of the timer clock source is not essential to Symbian OS, but somewhere between 32 kHz and 1 MHz is common. Slower clock sources have lower power consumption, and faster clock rates allow more flexible use of the timers, beyond the kernel millisecond tick (see Figure 2.5).

The preferred hardware implementation is a free-running 32-bit counter coupled with a set of 32-bitmatch registers to generate the timer interrupts. They enable simple software schemes for anti-jitter, idle tick suppression and profiling. Self-reloading countdown timers are an alternative hardware option, but they are less flexible.

(C) Copyright Nokia 2013. All rights reserved.

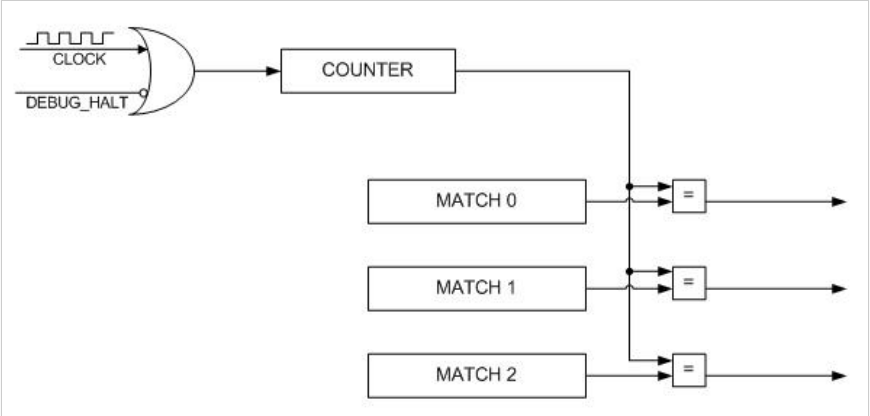

Figure 2.5 *High-speed timer with three match registers*

The normal operation of the millisecond timer with match registers is straightforward. The external clock source drives the counter, and on every increment the match registers are tested. If they match, their interrupt line is raised, the millisecond timer ISR will execute, kernel millisecond tick processing will occur and then the ISR will re-queue the interrupt by adding 1 ms worth of clock ticks to the match register.

The counter is always allowed to be free-running and the match register is always incremented from the previous match value. This produces a drift-free millisecond interrupt. If the input clock frequency is not an exact multiple of 1 Hz, anti-jitter software will generate an average 1 ms timer, by adding or removing a few extra clock cycles per millisecond interrupt.

For the kernel to keep accurate track of time when the CPU is asleep, the timer input clock and the counter circuits must be powered from an independent source to the core.

To debug software running in a system with high-speed timers, it is essential that the JTAG debugger hardware suspends the timers while it halts the CPU. It does this by the input of a DEBUG_HALT signal into the timer block. Stopping the timers ensures that the OS is not flooded by timer interrupts during debug single steps, and that the kernel timer queues are not broken by too much unexpected time elapsing.

Multiple timers are required within a real system even though EKA2 itself only needs one. Peripherals with sub-millisecond timing requirements, for example those polling a NOR Flash for write completion, will use an extra timer. Spare timers can also be used for accurate performance profiling.

Direct Memory Access (DMA)

Direct Memory Access (DMA) is used by Symbian OS to offload the burden of high bandwidth memory to peripheral data transfers and allow the CPU to perform other tasks. DMA can reduce the interrupt load by a factor of 100 for a given peripheral, saving power and increasing the real-time robustness of that interface.

Chapter 13, *Peripheral Support*, will describe how the EKA2 software framework for DMA is used with the different DMA hardware options and device drivers.

A DMA engine is a bus master peripheral. It can be programmed to move large quantities of data between peripherals and memory without the intervention of the CPU.

Multi-channel DMA engines are capable of handling more than one transfer at one time. SoCs for Symbian phones should have as many channels as peripheral ports that require DMA, and an additional channel for memory-to-memory copies can be useful.

A DMA channel transfer will be initiated by programming the control registers with burst configuration commands, transfer size, the physical addresses of the target RAM and the peripheral FIFO. This is followed by a DMA start command. The transfers of data will be hardware flow controlled by the peripheral interface, since the peripherals will always be slower than the system RAM.

In a memory to peripheral transfer, the DMA engine will wait until the peripheral signals that it is ready for more data. The engine will read a burst of data, typically 8, 16 or 32 bytes, into a DMA internal buffer, and it will then write out the data into the peripheral FIFO. The channel will increment the read address ready for the next burst until the total transfer has completed, when it will raise a completion interrupt.

A DMA engine that raises an interrupt at the end of every transfer is single-buffered. The CPU will have to service an interrupt and re-queue the next DMA transfer before any more data will flow. An audio interface will have a real-time response window determined by its FIFO depth and drain rate. The DMA ISR must complete within this time to avoid data underflow. For example, this time would be about 160 µs for 16-bit stereo audio.

Double-buffered DMA engines allow the framework to queue up the next transfer while the current one is taking place, by having a duplicate set of channel registers that the engine switches between. Double-buffering increases the real-time response window up to the duration of a whole transfer, for example about 20 ms for a 4 KB audio transfer buffer.

Scatter-gather DMA engines add another layer of sophistication and programmability. A list of DMA commands is assembled in RAM, and then the channel is told to process it by loading the first command into the engine. At the end of each transfer, the DMA engine will load the next command - until it runs out of commands. New commands can be added or updated in the lists while DMA is in progress, so in theory my audio example need never stop.

Scatter-gather engines are good for transferring data into virtual memory systems where the RAM consists of fragmented pages. They are also good for complex peripheral interactions where data reads need to be interspersed with register reads. NAND controllers require 512 bytes of data, followed by 16 bytes of metadata and the reading of the ECC registers for each incoming block.

Power savings come from using DMA, since the CPU can be idle during a transfer, DMA bursts don't require interrupts or instructions to move a few bytes of data, and the DMA engine can be tuned to match the performance characteristics of the peripheral and memory systems.

(C) Copyright Nokia 2013. All rights reserved.

Liquid Crystal Display (LCD)

The primary display on Symbian OS phones is a color liquid crystal display. The job of the display is to turn pixels in the frame buffer into images we can see.

Displays in Symbian phones come in common sizes dictated by the user interface software layers, since the latter are graphically optimized to the screen size. The most common resolutions are 176 × 208 pixels as used in S60 phones and 240 × 320 pixels for UiQ.

The frame buffer is an area of contiguous physical memory, large enough to contain an array of pixels with the same dimension as the display.

The base port reserves memory for the frame buffer during initialization, and ensures the MMU maps it with write-through caching. A frame buffer for 16-bpp QVGA display will require 150 KB (320 × 240 × 2) of RAM.

Pixels have common formats depending on their storage containers. Most current mobile phones use 16 bits per pixel (bpp) in 565 format, where the top 5 bits are red, the middle 6 bits are green, and the bottom 5 bits are blue - giving 65,535 (2^{16}) unique colors:

15 ⇒ 11	**10 ⇒ 5**	**4 ⇒ 0**
RED	GREEN	BLUE

Phones with genuine 18-bpp displays are starting to be common, they display 262,144 colors. Symbian OS does not support 18-bit words - instead a 24-bpp representation is used inside a 32-bit word. This has an aRGB, 8888 format, where the *a* is empty space or an alpha value. The LCD controller will discard the bottom two bits of each color component byte:

31 ⇒ 24	**23 ⇒ 16**	**15 ⇒ 8**	**7 ⇒ 0**
alpha	RED	GREEN	BLUE

Mobile phone LCDs come in two distinct types, dumb and smart. A dumb display does not contain any control electronics; instead its pixels are driven directly by the LCD controller in the SoC. On the other hand, a smart display contains its own LCD controller, memory for the frame buffer, and a medium-bandwidth interface back to the SoC.

The dumb display controller within an SoC has to output a new copy of the display image around 60 times per second to persist the image on the LCD.

This 60 Hz update requires the controller to continually transfer data from memory using DMA as long as the display is switched on. Using IRAM as a frame buffer can help reduce the power cost and bandwidth overhead of the display. As I said earlier, a 16-bit QVGA with a 60 Hz refresh rate will require 8.78 MB of data every second, and looking to the future, a full 32-bit VGA display will require eight times as much data.

The interface to a smart display is optimized for two uses: incremental updates to the display when small elements change, and full bandwidth operation when multimedia video or games require it. For the small updates, the updated screen region is transferred into the display interface using a smart 2D DMA engine.

Smart displays save power by removing the need for full-bandwidth updates most of the time. Their internal circuitry is optimized to match the display characteristics exactly. They have additional low-power modes for idle phones that only want to display the clock. These have partial display refreshing and limited colors.

Audio

The audio sub-system of a Symbian OS phone contains two, mostly independent, streams of audio data. One is the telephony voice data and the other is multimedia data.

Two vital phone use cases are to have good quality voice calls and long talk times. Digital audio buses are dedicated to voice data to ensure this is the case.

The de facto raw hardware audio format used in a Symbian OS phone is 16-bit pulse code modulated (PCM) data. The quality ranges from 8 kHz mono for telephony audio up to 48 kHz stereo for music playback.

PCM audio hardware can be quite simple, requiring little setup to ensure the volume and correct output path are selected. Then all that is needed is to feed data to the hardware at the rate it demands - DMA hardware is very good for this. If data transfer stalls, the audio hardware will immediately produce audible pops, clicks and stutters.

Telephony audio

The telephony voice data is the essence of a phone call. It has stringent latency restrictions placed upon it to ensure that the user has a high-quality call without the effects of transatlantic satellite lag. To ensure this is the case, the system designers have optimized the controlling software and hardware paths for low latency response and low power consumption during a voice call.

The BP contains a DSP that performs the processing for voice band audio without passing through Symbian OS. During a call, Symbian OS will be in a low-power mode, only needing to wake up when the display needs updating.

A normal call will end up in the analogue audio circuits. They contain the analogue to digital and digital to analogue converters, which in turn are connected to the microphone and speakers. When using a BlueTooth (BT) headset the PCM data is transported directly into the BT module via its own dedicated interface.

Symbian OS needs an additional audio path to allow it to inject system sounds into the ongoing call. These are for such things as message alerts, low battery and a second incoming call. This injection of sounds can be done by passing the raw audio data over the IPC link to the BP, where the DSP will mix it into the audio stream.

Multimedia audio

(C) Copyright Nokia 2013. All rights reserved.

Multimedia audio is a general term for every generated sound in the system that is not voice data.

The main multimedia sounds are:

- Ring tones, in many formats
- Alerts, for incoming messages
- Alarms, from clock and calendar
- Video telephony
- MP3 playback
- Games
- Recorded voice, dictaphone
- Video capture and playback.

The higher levels are all controlled by the Symbian multimedia framework (MMF) for media players, file formats and plug-ins.

The Multimedia Device Framework (MDF) will contain the codecs, and it will transfer PCM data to and from the device driver layer, DevSound. Video telephony (VT) is a special case, in which live audio data does pass through Symbian OS. The audio elements of the call are multiplexed into the 64-kb/s data stream along with the video. The VT call system has to de-multiplex the incoming data stream, decode the audio and video elements, and then play them in sync. This is normally done in dedicated hardware or a DSP, since it would require all of a 200 MHz ARM CPU just to run the codecs.

The main complexity in the audio systems is the ever-growing number of audio sources and sinks, and the possible ways in which they can be connected. For example, current phones have multiple headsets, speakers, Bluetooth and FM radios. This is likely to remain an issue until hardware is capable of mixing and routing every audio source in all possible combinations. Today some audio use cases will be incompatible with others, requiring them to interrupt each other.

Power management

All Symbian OS phones are battery powered, and as I have stressed throughout this chapter, effective power management (PM) is crucial in designing a successful Symbian OS phone.

The overall design goals of the SoC team must be focused on adequate performance at low power. At every decision point the system's designers must ask themselves, *How will this affect the power consumption?*

Can I design this in another way that will have the same performance with lower power? *Only by constant attention to power use will a single battery charge give hours of talk and play time, and still leave hundreds of hours for standby.*

The ARM CPUs used in Symbian OS phones are designed as answers to these questions, as they are the best low-power peripheral blocks in use. Schemes to reduce power consumption within the SoC include self-clocking circuits, serial ports that power down when idle and memory controllers that put RAM into low-power modes whenever possible.

Most importantly, the SoC should only include hardware capabilities that will be used. Use case analysis and feature justification are a good way of removing unnecessary power loads at the start of the design. An example of a bad design is an over-specified internal bus with more bandwidth than all of its slave peripherals.

I explain the EKA2 power model in Chapter 15, *Power Management*. This power management architecture is based on the runtime use of shared power resources by the peripherals. To closely match this model, the SoC power architecture needs to be partitioned for clean and orthogonal PM, enabling peripherals to have well-defined power and clock sources that can be turned on and off as needed.

The designers of CPUs like to invent all sorts of named power modes, for example Turbo, Run, Idle, Doze, Sleep, Deep Sleep and Off. The more power-saving the mode is, the more work is required to get into or out of the state. For example, a Deep Sleep state is likely to require flushing all the state out of the caches, and turning off the core, and waking from Deep Sleep will initially look like a boot to the CPU.

Almost all of these modes will map onto the Symbian OS idle mode, where the system is powered but doing nothing. Symbian OS tends to spend most of its time in idle mode, even going into it between presses on a phone keypad. As I've already hinted, the most important difference between the CPU power modes to Symbian OS is the time it takes to transition from Run to a mode and the time taken to get back to Run mode again. The decision about which CPU mode to use is determined heuristically within the idle software, based on recent use and pending timer events.

Dedicated hardware accelerators allow for excellent performance with low power consumption, whereas general purpose ARM software is infinitely reprogrammable, but execution is more expensive in power and time. The trick is to get the right balance between fixed hardware and flexible software, and ensure that the real phone requirements can be met two years after the silicon is finalized.

Power management is a pervasive topic. It requires careful hardware design and focused software to achieve the system design goals.

Summary

In this chapter I have described the core hardware that is needed to run Symbian OS, with some emphasis on the power management implications of those hardware choices. There are a great many more peripherals and topics that I do not have space to cover here, including:

- Real-time clock
- Touch screen
- IPC interface to BP
- Debug interfaces
- Flash programming
- Multiple displays
- IRDA
- Booting

(C) Copyright Nokia 2013. All rights reserved.

- Removable media, SD, MMC
- 2D Graphics
- 3D Graphics
- DSP
- Multimedia accelerators
- USB interfaces
- Advance power managemen
- Bluetooth modules
- Watchdogs and resets
- Security hardware.

Within Symbian, we refer to the software layers that control the hardware, including the bootstrap, kernel port and device drivers as the baseport, and more generically as the Board Support Package (BSP).

Designing hardware for a Symbian OS phone requires a system view of the final product. The designers need to consider real-world performance use cases, select hardware with enough (but not too much) capacity, and with every decision, they need to analyze power consumption.

I hope that this chapter will have given you an insight into the design choices you will need to make if you are building a Symbian OS phone. In the next chapter, I will start to look at the fundamental entities that underlie the running of code on Symbian OS - threads, processes and libraries.

© 2010 Symbian Foundation Limited. This document is licensed under the Creative Commons Attribution-Share Alike 2.0 license. See http://creativecommons.org/licenses/by-sa/2.0/legalcode for the full terms of the license.

Note that this content was originally hosted on the Symbian Foundation developer wiki.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.