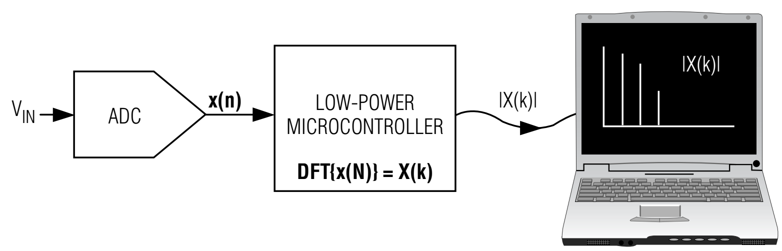
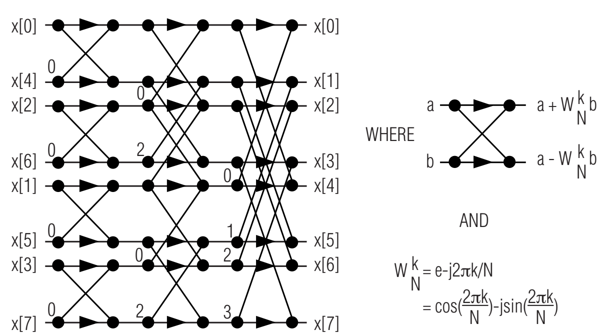
DALLAS SEMICONDUCTOR MAXIM

Engineering Journal

Volume Fifty-Seven



(1)



(2)

Figure 1. The spectrum of an input voltage is calculated using an FFT application. (See article inside, page 9.)

Figure 2. A butterfly computation is used to perform an FFT for N = 8. (See article inside, page 10.)

News Brief

MAXIM REPORTS RECORD REVENUES FOR ITS SECOND QUARTER 2006 AND 10% QUARTER OVER QUARTER BOOKINGS GROWTH

Maxim Integrated Products, Inc., (MXIM) reported a record for net revenues of $445.9 million for its second quarter ending December 24, 2005, a 5.1% increase over the $424.4 million reported for the first quarter of fiscal 2006. Pro forma net income excluding stock-based compensation expense for the quarter was $140.0 million or $0.42 diluted earnings per share and GAAP net income was $112.6 million including stock-based compensation or $0.33 diluted earnings per share. This compares to $133.2 million of pro forma net income or $0.39 diluted earnings per share reported for the first quarter of fiscal 2006 and GAAP net income of $105.4 million including stock-based compensation or $0.31 per diluted share.

Gross bookings for its second quarter were approximately $506 million, a 10% increase from the first quarter's level of $459 million. Gross turns orders received in the quarter were approximately $230 million, a 10% increase from the $208 million received in the prior quarter. Bookings increased in all geographic locations. Second quarter ending backlog shippable within the next 12 months was approximately $370 million, including approximately $329 million requested for shipment in the third quarter of fiscal 2006. The Company's first quarter ending backlog shippable within the next 12 months was approximately $330 million, including approximately $296 million that was requested for shipment in the second quarter of fiscal 2006.

Pro forma research and development expense (excluding stock-based compensation expense) was $92.6 million or 20.8% of net revenues in the second quarter and GAAP research and development expense was $116.9 million or 26.2% of net revenue including stock-based compensation of $24.3 million. Pro forma selling, general and administrative expense (excluding stock-based compensation expense) was $23.8 million in the second quarter or 5.3% of net revenues while GAAP selling, general and administrative expense was $31.1 million or 7.0% of net revenue including stock-based compensation of $7.2 million.

During the quarter, the Company repurchased 9.2 million shares of its common stock for $334.6 million, paid dividends of $40.0 million, and acquired $37.4 million in capital equipment. Accounts receivable increased $8.1 million in the second quarter to $221.0 million due to the increase in net revenues. Pro forma inventories (excluding stock-based compensation expense) increased to $186.5 million from the previous quarter. GAAP reported inventories for the second quarter increased to $197.8 million and includes $11.3 million for stock-based compensation.

The Company expects to implement a program that will allow its employees, excluding officers, holding vested stock options with an exercise price of at least $35 to exchange them for Restricted Stock Units (RSUs) vesting quarterly over the next 12 months at a specified exchange rate derived using the Black-Scholes model. In some cases, employees may elect to exchange these vested options for RSUs at a specified exchange rate that is greater than that derived using the Black-Scholes model and these RSUs will vest quarterly over the next 18 months. This program, details of which will soon be filed with the Securities and Exchange Commission (SEC) and communicated to those eligible to make an exchange, is designed to foster retention of our employees and to better align their interests with those of our stockholders. This exchange program may reduce the number of Maxim's outstanding employee stock options and provide ownership of Maxim stock to employees making the exchange election. A total of approximately 20 million vested options are covered by the exchange program and, if all options are tendered, approximately 4 million RSU's would be issued. Maxim continues to believe that equity-based forms of compensation are most effective in motivating employees and aligning their goals with shareholders' interests.

Employees holding stock options eligible for exchange in the program should carefully read the Company's Offer to Exchange certain stock options for RSU's, the Company's letter of transmittal and related tender offer materials when they become available because they will contain important information, including, among other things, the various terms and conditions governing the program. Copies of the Company's Offer to Exchange certain stock options for RSU's, the letter of transmittal and related tender offer materials will soon be mailed to all employees holding stock options eligible for exchange in the program and, once filed with the SEC, may be obtained at no charge from the SEC's web site at www.sec.gov.

Mr. Gifford commented: "Our second quarter performance is a positive reflection of our long-term strategy, which is to serve and gain market share in many analog industry market segments. We believe the future prospects for the analog industry are exciting and that we are well positioned for profitable growth."

Mr. Gifford concluded: "The Company's Board of Directors has declared a cash dividend for the third quarter of fiscal 2006 of $0.125 per share. Payment will be made on February 28, 2006 to stockholders of record on February 13, 2006."

For the complete Q206 press release, including safe harbor information, go to: www.maxim-ic.com/NewsBrief

Protecting the R&D Investment—Two-Way Authentication and Secure Soft-Feature Settings

In the age of identity theft and falsified IDs, assuring positive identification is of paramount importance. This is not only true for individuals, but also for electronic products. System vendors need to protect their products against hacker attacks from the "outside" and ensure that the security is not compromised on the "inside" through cloned hardware. The key to realizing these diverging security requirements is authentication.

What is Authentication?

Authentication is a process to establish proof of identity between two or more entities. In the case of one-way authentication, one party proves its identity to another. With two-way authentication, both parties prove their identity to each other. The most commonly used method of authentication is the password. The main problem with passwords is that they are exposed when used, making them vulnerable to spying.

After reviewing the historical use of cryptography, in 1883 the Flemish linguist Auguste Kerckhoffs published a groundbreaking article on military cryptography. Kerckhoffs argued that instead of relying on obscurity (e.g., an undisclosed, nonscrutinized algorithm), security should depend on the strength of the algorithm and its keys. In the event of a security breach, Kerckhoffs asserted, only the keys would need to be replaced, not the whole system.

Key-based authentication works as shown in **Figure 1**: a (secret) key and the to-be-authenticated data ("message") are taken as input to compute a message authentication code, or MAC. The MAC is then attached to the message. The recipient of the message performs the same computation and compares its version of the MAC to the one received with the message. If both MACs match, the message is authentic.

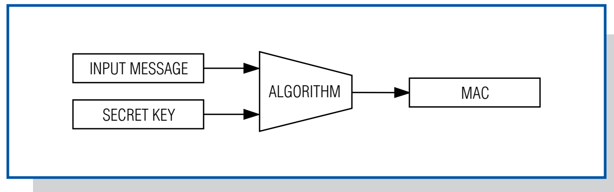


Figure 1. This MAC computation model exemplifies key-based authentication.

There is, however, a weakness with the basic MAC computational model. An intercepted message can later, or subsequently, be replayed by a nonauthentic sender and be mistaken as authentic. To circumvent this inherent MAC weakness and prove the authenticity of the MAC originator, the recipient generates a random number and sends it as a challenge to the originator. The MAC originator must then compute a new MAC based on the secret, message, *and* challenge and send it back to the recipient. If the originator generates a valid MAC for any challenge, it is quite certain that it knows the secret and, therefore, can be considered authentic (see **Figure 2**). This process is known as challenge-and-response authentication.

1-Wire is a registered trademark of Dallas Semiconductor Corp.

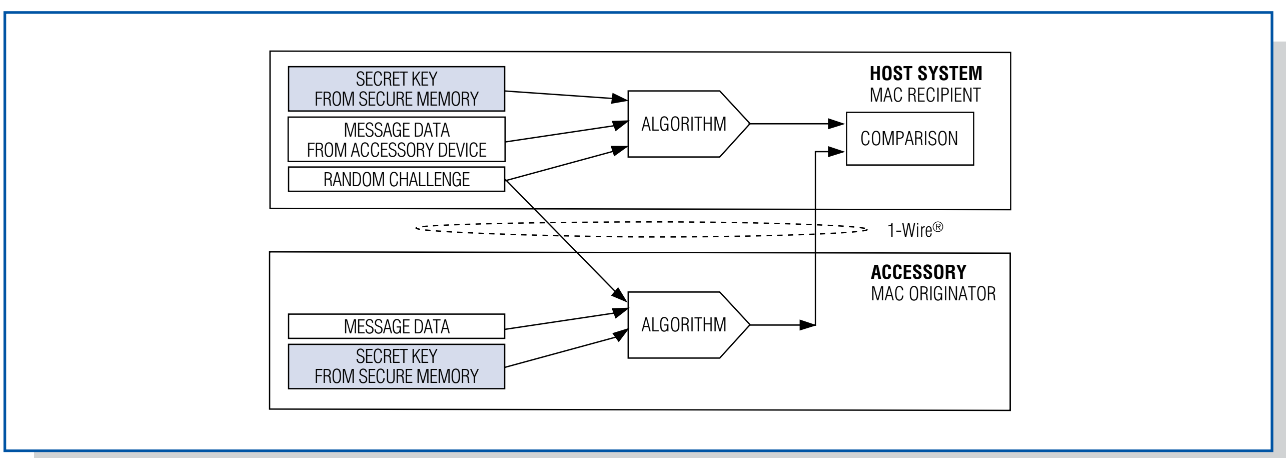


Figure 2. The MAC model's weakness, allowing an intercepted message to be mistaken as authentic, can be resolved by the challenge-and-response authentication process.

In cryptography, an algorithm that generates a fixed-length MAC from a message is called “one-way” hash function. One-way indicates that it is extremely difficult to deduce the usually larger message from the fixed-length MAC output. With encryption, in contrast, the size of the encrypted message is proportional to the original message.

A thoroughly scrutinized and internationally certified one-way hash algorithm is SHA-1, developed by the National Institute of Standards and Technology (NIST) (www.itl.nist.gov/fipspubs/fip180-1.htm). SHA-1 has evolved into the international standard ISO/IEC 10118-3:2004, and the math behind the algorithm is publicly available through the NIST website. Distinctive characteristics of the SHA-1 algorithm are: 1) irreversibility—it is computationally infeasible to determine the input corresponding to a MAC; 2) collision-resistance—it is impractical to find more than one input message that produces a given MAC; and 3) high avalanche effect—any change in input produces a significant change in the MAC result. For these reasons, as well as the international scrutiny of the algorithm, Maxim/Dallas Semiconductor selected SHA-1 for challenge-and-response authentication of its secure memories.

Low-Cost Secure Authentication—A Functional Implementation

Thanks to its 1-Wire interface, the DS2432 EEPROM device with a SHA-1 engine can easily be added to any circuit with digital processing capabilities, such as a microcontroller (µC). In the simplest case, all that is needed is one free I/O pin and a pullup resistor for the 1-Wire interface, as shown in **Figure 3**. If the computing capabilities on the board or the remaining program storage space are insufficient to compute a SHA-1 MAC, one can use a DS2460 SHA-1 coprocessor or leave this task to the nearest host in the system or network. The coprocessor has the additional advantage of storing the system secret in secure memory rather than in the host-process program code.

Embedded HW/SW License Management

Reference designs, which are subsequently licensed and possibly manufactured by third parties, require barriers to prevent illegal use of the intellectual property. For revenue reasons, it is also necessary to track and confirm the number of reference uses. A preprogrammed DS2432 (secret and memory settings installed prior to delivery to the third-party manufacturer) easily solves these requirements, and more. As a power-up self-check, the reference (**Figure 4**) performs an authentication sequence with the DS2432. Only a DS2432 with valid secret, known by just the licensing company and reference electronics, will successfully reply with a valid MAC. The reference processor will take appropriate, application-specific action if an invalid MAC is detected. The additional benefit of this approach is the ability to selectively license and enable reference features through settings in the DS2432’s secure memory. (For more information on this concept, see the section **Soft-Feature Management**.)

The DS2432 with a 64-bit valid secret is supplied to the licensee or third-party manufacturer through one of two secure methods: 1) preprogrammed by the company licensing the reference; or 2) preprogrammed by Maxim/Dallas Semiconductor per the licensing company’s input and then delivered to the third-party manufacturer. In either case, the number of devices sent to the licensee or manufacturer is known and can be used to validate license fees.

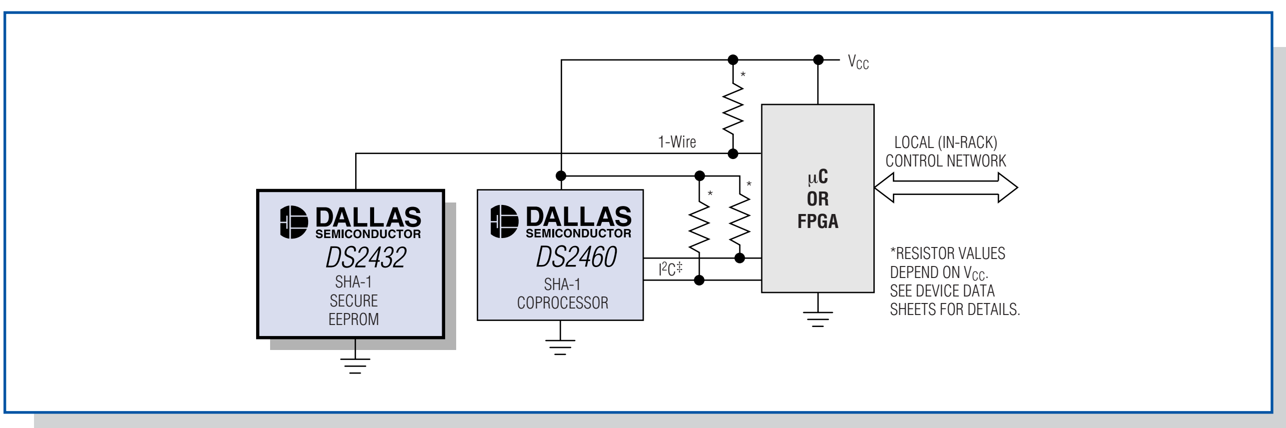


Figure 3. A DS2432 EEPROM device is implemented in a typical board environment through use of a free I/O pin and a pullup resistor.

‡*Purchase of I^2C components from Maxim Integrated Products, Inc., or one of its sublicensed Associated Companies, conveys a license under the Philips I^2CPatent Rights to use these components in an I^2C system, provided that the system conforms to the I^2C Standard Specification defined by Philips.*

Verification of Hardware Authenticity

When verifying hardware authenticity, there are two cases to be considered (**Figure 5**): 1) a cloned circuit board with an exact copy of the firmware/FPGA configuration; and 2) a cloned system host.

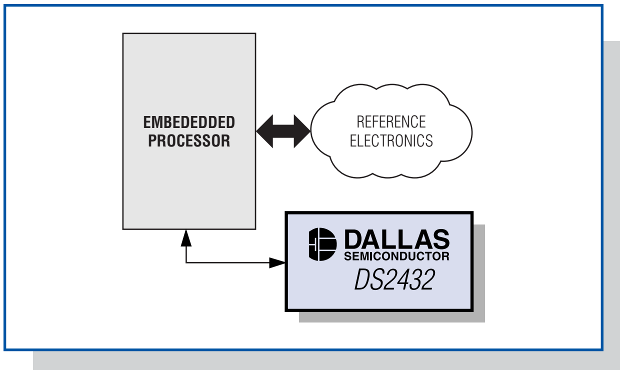


Figure 4. A reference design is authenticated through use of the DS2432.

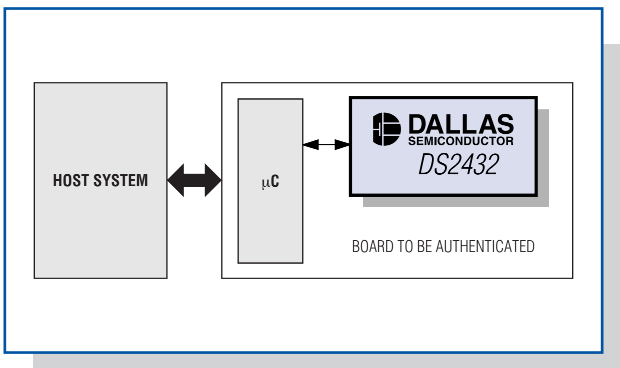


Figure 5. This HW authentication example shows a cloned circuit board with an exact copy of the firmware/FPGA configuration or a cloned system host.

In the first case, the firmware/FPGA attempts to authenticate the cloned circuit board. The clone manufacturer must load a secret into the DS2432 in order to write data into the user EEPROM. While this can make the data look correct, the secret is not valid in the system. Due to the complexity in changing the firmware/FPGA and to remain compatible with the host, the firmware/ configuration must be an exact copy of the original. If the board performs a challenge-and-response authentication of the DS2432 during the power-up phase, the MAC generated by the DS2432 will differ from the MAC computed by the microcontroller/FPGA. This MAC mismatch is strong evidence that the board is not authentic. The system performing a challenge/response sequence with the board would detect this mismatch and application-specific action would then be taken.

In the second case, the circuit board attempts to authenticate the host system. The board can verify the authenticity of the host using the following procedure: 1) generate a challenge and let the DS2432 compute a challenge-and-response authentication MAC; and 2) send the same MAC computation input data (except for the secret, of course) to the network host, which then computes and returns a challenge-and-response authentication MAC from that data and its own secret. If both MACs match, the board can assume that the host is authentic.

Soft-Feature Management

Electronic systems range from handheld products to units that fill several racks. The larger the unit's size, the more costly it is to develop. To keep the cost under control, there is a desire to construct a large system from a limited selection of smaller subsystems (boards). Often, not all features of a subsystem are needed in the application. Instead of removing these features, it is more cost-effective to leave the board as is, and to simply disable some features in the control software. This approach, however, creates its own new problem: a smart customer who needs several fully featured systems could just buy one fully featured unit and several units with reduced features. Then, using copied software, the simpler units behave like the fully featured unit but for a lower price, thus shortchanging the system vendor.

A DS2432 on the board of each subsystem protects the system vendor from this type of fraud. Besides performing challenge-and-response authentication, the same DS2432 can store the individual configuration settings in its user EEPROM. As explained later in the **Data Security** section, the data is protected from unauthorized changes, giving full control to the system vendor. The configuration settings can be stored in the form of a bitmap or code words, as deemed appropriate by the system designer. For practical reasons, the configuration should be as easy to set as possible. Due to the 1-Wire interface in the DS2432, the designer only needs to add a single transistor and a probe point, as shown in **Figure 6**. Through the probe point, the configuration can be written to the DS2432 without powering the rest of the board. The MOSFET isolates the DS2432 from the other circuitry without impeding normal access to the DS2432 when the subsystem is operated in its normal environment.

As an added security bonus, this method of setting configurations allows for remote feature upgrade/change after the system is installed at the customer's site. Any user EEPROM that is not used for configuration/feature management is available for board identification in the form of an electronic nameplate. This feature is explained in detail in Application Note 178, *Printed Circuit Board Identification Using 1-Wire Products*, on the Maxim website at www.maxim-ic.com/an178.

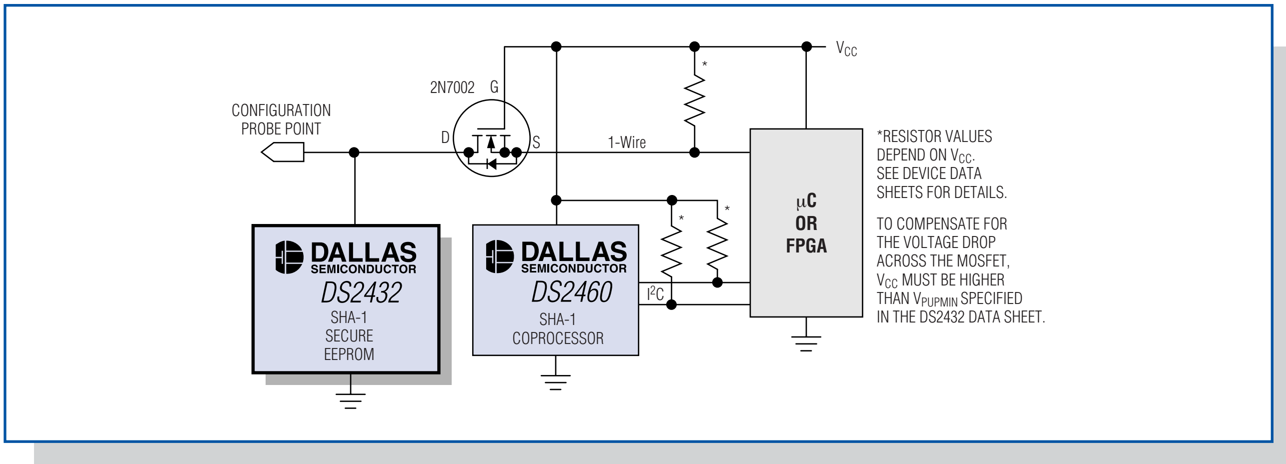


Figure 6. Configuration settings can be written to the DS2432 by adding a single transistor and a configuration probe point.

DS2432 Authentication Feature Details

General Device Architecture

The major data elements and the data-flow paths of the DS2432 1kb SHA-1 secure memory with 1-Wire interface are shown in **Figure 7**. Easily recognized are the 8-byte secret key and the buffer memory (scratchpad), which temporarily stores the challenge. Data elements not mentioned previously are the unique device ID number (a standard 1-Wire feature), four pages of user EEPROM, control registers, and system constants.

The device ID serves as a node address in 1-Wire networks, but also contributes to authentication. The user memory holds the major part of the to-be-authenticated "message." Seed constants are needed to meet formatting requirements and as padding to compose the 64-byte input data block for the SHA-1 computation. The control registers perform device-specific functions, such as optional write protection of the secret or EEPROM emulation mode; they do not contribute to the authentication process in general.

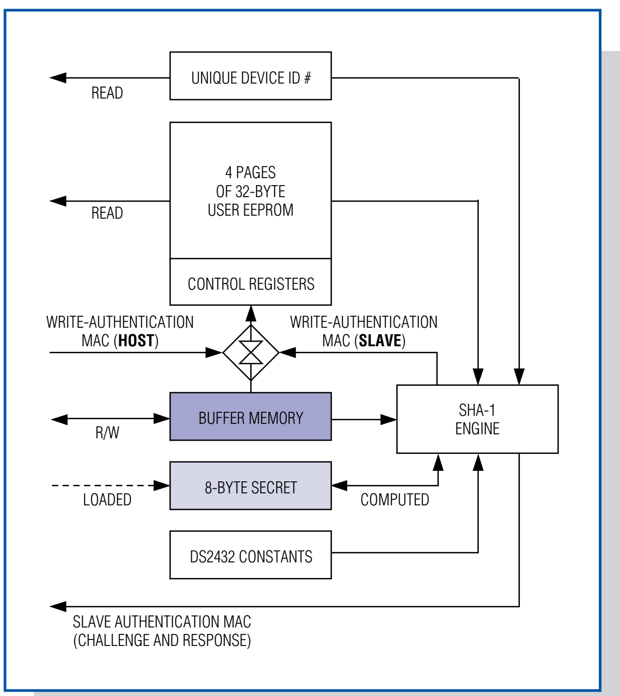


Figure 7. All major data elements and data-flow paths are shown for the DS2432 SHA-1 secure memory data-flow model.

Device ID number and user EEPROM can be read without restriction. There is full read/write access to the buffer memory. The secret can be loaded directly, but never read. Changing the content of the user memory or the registers requires that both host and slave (i.e., the DS2432) compute matching write-authentication MACs to open the path from the buffer memory to the EEPROM.

The DS2432's SHA-1 engine can be operated in three different ways, depending on the purpose of the MAC result. In any case, the SHA-1 engine gets 64 bytes of input data and computes from it a 20-byte MAC result. The differences are in the input data. As a fundamental requirement of secure systems, the host must either know, or be able to compute, the secret of a slave device that is valid/authentic in the application.

Challenge-and-Response Authentication MAC

A described previously in the application examples, the primary purpose of the DS2432 is challenge-and-response authentication. The host sends a random challenge and instructs the DS2432 to compute a response MAC from the challenge, the secret, data from one of the memory pages selected by the host, and additional data that together constitute the message (see **Figure 8**).

After it has finished computing, the DS2432 sends its MAC to the host for verification. The host then duplicates the MAC computation using a valid secret and the same message data that was used by the DS2432. A match of

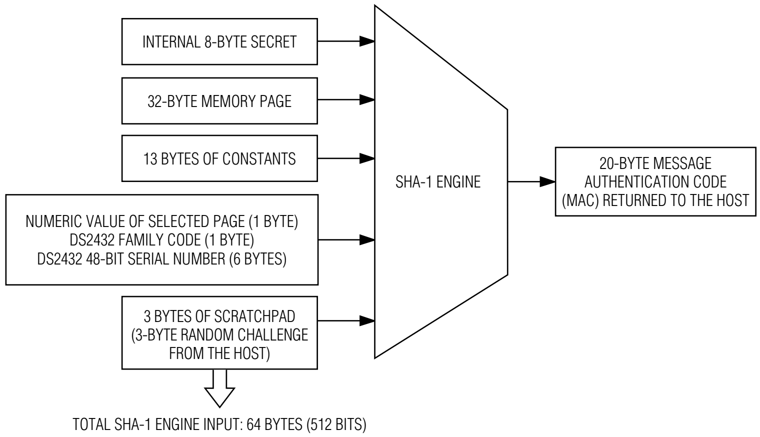


Figure 8. Specific DS2432 SHA-1 engine input data is shown for the challenge-and-response authentication MAC.

the MAC received from the DS2432 authenticates the device, as only an authentic DS2432 will respond to the challenge-and-response sequence correctly. It is crucial that the challenge is based on random data. A never-changing challenge allows replay attacks using a valid, static, recorded and replayed MAC instead of a MAC that is instantly computed by an authentic DS2432.

Data Security

Beyond proving the authenticity of a slave device, it is highly desirable to know that the data stored in the device can be trusted. For this reason, write access to the DS2432 EEPROM is securely restricted. Before copying data from its scratchpad buffer memory to the EEPROM or control registers, the DS2432 requires the requesting host to supply a write-access authentication MAC to prove its authenticity. The DS2432 computes this MAC from the new data in its scratchpad buffer memory, its secret, data from the memory page to be updated, and additional data (see **Figure 9**).

An authentic host knows the secret and computes a valid write-access MAC. When receiving the MAC from the host during the copy command, the DS2432 compares it to its own result. Data is transferred from the buffer memory to the destination in EEPROM only if both MACs match. Of course, memory pages that are write-protected cannot be modified, even if the MAC is correct.

Secret Protection

The architecture of the DS2432 allows direct load of a secret into the device. Secret protection is provided by both read protection and, if desired, write protection, which prevents the secret from ever being changed. This level of protection is effective so long as access to the secret is secure and controlled at the equipment production site.

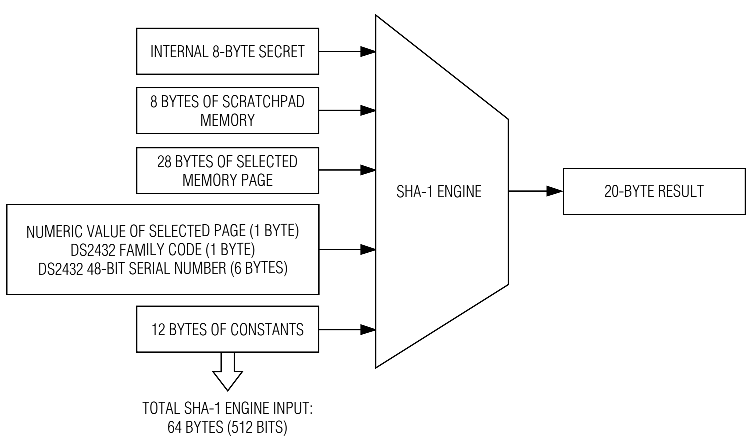


Figure 9. SHA-1 engine input data is used to compute a write-access authentication MAC.

The quality of the secret can be increased in various ways: 1) let the DS2432 compute its secret; 2) let the DS2432 compute its secret in multiple stages performed at different sites; 3) create device-specific secrets by including the unique device ID number in the computation of the secret; or 4) a combination of 2 and 3.

In ‘1’ above, if each DS2432 computes its secret, only the ingredients of the secret are known; the secret itself is never exposed. If the secret is computed in multiple stages using different sites, as in ‘2’ above, only the “local” ingredients of the secret are known. This approach provides a method to control knowledge of the “final” secret. If the secrets are device-specific (‘3’ above), an additional computing step is required for the host. However, the potential damage is minimal if a device secret is accidentally discovered. If the secret is computed in multiple stages and made device-specific (‘4’ above), the highest possible secrecy is achieved. However, the hosts, like the slaves, need to be set up at different sites to ensure system secrecy.

Before computing a secret, it is necessary to first load a known value as secret. With the help of this known secret, 32 bytes of the data that will be used in computing the new secret must be written to one of the four memory pages. Next, a partial secret should be written to the DS2432 scratchpad buffer memory. The partial secret could, for example, be the number of the memory page used for the computation and the unique device ID number (excluding the CRC byte) or any other application-specific 8-byte value.

If instructed to compute a secret, the DS2432 starts its SHA-1 engine and computes a MAC using the input data items shown in **Figure 10**. The lower 8 bytes of the 20-byte MAC are automatically copied to the secret’s memory location and become effective immediately.

Conclusion

Knowing secure authentication functions and implementing them wisely gives a competitive advantage. Authentication not only protects intellectual property, but also helps reduce production cost through common HW platforms with secure, soft-feature settings. The DS2432’s data security even allows remote configuration changes, saving the technician valuable time. As the DS2432 exemplifies, a small silicon chip can make a big difference to the bottom line.

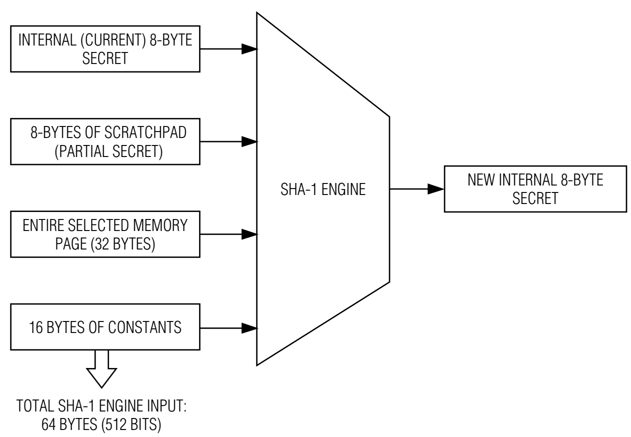


Figure 10. Input data is used for a device-performed secret computation; the lower 8 bytes of the 20-byte MAC result become the new secret.

Developing FFT Applications with Low-Power Microcontrollers

As low-power microcontrollers (µCs) begin including peripherals that were formerly reserved for larger microprocessors, ASICs, and DSPs, new opportunities for computing complex algorithms at low power levels are becoming possible. This article describes a Fast Fourier Transform (FFT) application (developed using a low-power µC) that includes a single-cycle hardware multiplier.

This FFT application computes, in real-time, the spectrum of an input voltage (V_{IN} in **Figure 1**). To accomplish this, an analog-to-digital converter (ADC) samples V_{IN} and transfers those samples to the µC. The µC then performs a 256-point FFT on the samples to obtain the spectrum of the input voltage. For testing purposes, the µC calculates the magnitude of the spectrum and transfers the results to a PC where they are displayed in real time.

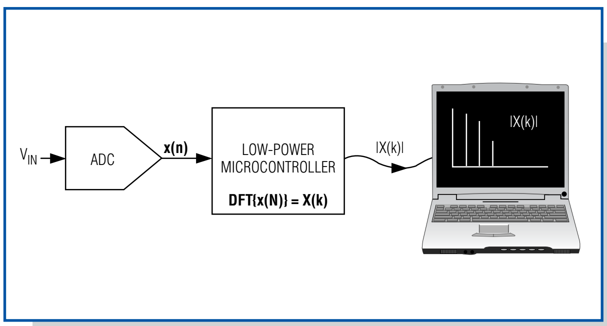


Figure 1. The spectrum of an input voltage is calculated using an FFT application.

The firmware for this FFT application is written in C for a 16-bit, low-power µC in the MAXQ2000 family. Interested readers can download the firmware and the circuit schematic for this project from the web article of the same name at www.maxim-ic.com/AN3722.

Background

To determine the spectrum of the sampled input signal, we must compute the Discrete Fourier Transform (DFT) of the input samples. The DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-j2\pi kn}{N}} \quad \text{for } 0 \le k \le N-1 \qquad \text{(Eq 1)}$$

where N is the number of samples, X(k) is the spectrum, and x(n) is the set of input samples. Expanding this summation using Euler's identity, and separating the input samples and spectrum into their real and imaginary components, yields the following equations:

$$\begin{aligned} X_{Re}(k) &= \sum_{n=0}^{N-1} \left[x_{Re}(n)\cos\left(\frac{2\pi kn}{N}\right) + x_{Im}(n)\sin\left(\frac{2\pi kn}{N}\right)\right] \\ &= \sum_{n=0}^{N-1} \left[x_{Re}(n)\cos\left(\frac{2\pi kn}{N}\right)\right] \end{aligned} \qquad \text{(Eq 2)}$$

$$\begin{aligned} X_{Im}(k) &= -\sum_{n=0}^{N-1} \left[x_{Re}(n)\sin\left(\frac{2\pi kn}{N}\right) - x_{Im}(n)\cos\left(\frac{2\pi kn}{N}\right)\right] \\ &= -\sum_{n=0}^{N-1} \left[x_{Re}(n)\sin\left(\frac{2\pi kn}{N}\right)\right] \end{aligned} \qquad \text{(Eq 3)}$$

The second term in the summation of equations 2 and 3 disappears because the input samples consist entirely of real numbers. Assuming we have N samples, computing equations 2 and 3 directly requires $2N^2$ multiplications and 2N(N - 1) additions. Therefore, our DFT with 256 input samples would require 131,072 multiplications and 130,560 additions. Enter the FFT!

Many FFT algorithms exist. The common radix-2 algorithm used in this implementation continuously decomposes the DFT into two smaller DFTs. For this to be possible, N must be a power of 2. The steps involved in the radix-2 FFT algorithm can be summarized with the butterfly computations illustrated in **Figure 2**. Observing these butterfly computations, we can determine that the radix-2 algorithm requires only $(N / 2)\log_2(N)$ multiplications and $N\log_2(N)$ additions. The values of W_N used in Figure 2 are commonly known as "twiddle factors" and can be computed before the algorithm is performed.

In Figure 2, the input to the FFT is shown in its peculiar, original order with the indices bit-reversed. Therefore, computing the radix-2 FFT algorithm with N = 8 requires the input data to be reordered from:

0 (000b), 1 (001b), 2 (010b), 3 (011b), 4 (100b), 5 (101b), 6 (110b), 7 (111b)
to:
0 (000b), 4 (100b), 2 (010b), 6 (110b), 1 (001b), 5 (101), 3 (011), 7 (111).

The FFT output appears in the correct order. Figure 2 also reveals that the results of the individual butterfly computations are the only data required for the next stage of the FFT. Because the computations are done "in place," new values can replace old values and only 2N variables are required to compute an FFT with N samples (2N variables are required because each value has a real and an imaginary part).

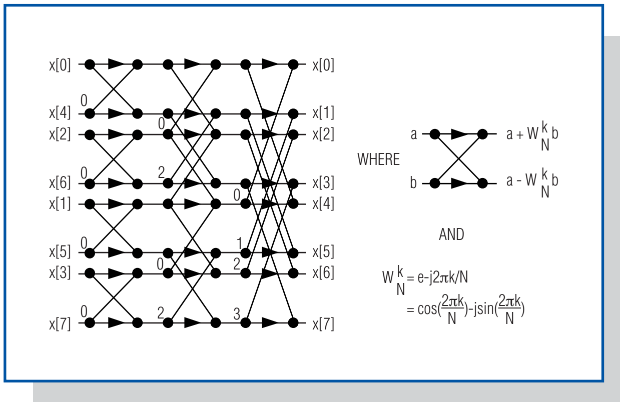


Figure 2. A butterfly computation is used to perform an FFT for N = 8.

When the FFT is complete, the results are in complex notation. Equations 4 and 5 convert the results into polar notation:

$$X_{MAGN}(k) = \sqrt{X_{\mathrm{Re}}^2(k) + X_{\mathrm{Im}}^2(k)} \qquad \text{(Eq 4)}$$

$$X_{PHASE}(k) = \arctan\left(\frac{X_{\mathrm{Im}}(k)}{X_{\mathrm{Re}}(k)}\right) \qquad \text{(Eq 5)}$$

The DSP literature includes many optimizations for the DFT/FFT algorithm described above to make it faster and smaller. One of the most important optimizations (and also perhaps the easiest to implement) arises from the realization that the magnitude of the DFT of a real-valued signal is symmetric around X(N / 2), therefore:

$$X(k) = X * (N / k) \qquad \text{(Eq 6)}$$

Writing an FFT is not simple. Several limitations of low-power µCs complicate the task even further.

Memory The selected µC has 2kB of RAM. Knowing that the algorithm requires 2N 16-bit variables for FFT data, our µC can perform FFTs with N up to 512. However, other parts of the firmware also require a few bytes of RAM. For our implementation, we therefore limit N to 256. Using 16-bit variables to represent the real and imaginary parts of every value, 1024 bytes of RAM are required for FFT data.

Speed Despite having a high MIPS/mA rating, low-power µCs still require some optimization to minimize the number of instructions required to run the FFT. Fortunately, the C compiler used for this application (IAR Embedded Workbench for MAXQ at www.iar.com) includes a number of optimization levels and settings. Careful use of the hardware multiplier allows the code to be optimized to an acceptable level.

No Floating-Point Capability The typical low-power µC chosen does not have floating-point capability. Consequently, fixed-point arithmetic is required for all computations. To represent fractional numbers, the firmware uses signed Q8.7 notation. The firmware therefore assumes:

- Bits 0 to 6 represent the fractional part of every number
- Bits 7 to 14 represent the integer part of every number
- Bit 15 represents the sign bit (two's complement)

These assumptions have no effect on additions and subtractions, but care must be taken during multiplications to align the numbers to Q8.7 format.

The notation chosen also accommodates the largest number that the FFT algorithm may encounter, while providing the highest accuracy. For example, our ADC provides signed 8-bit samples in two's complement format. If our input is a DC voltage with maximum amplitude (127 for signed 8-bit samples), the spectrum would be entirely contained in X(0) and be equal to 32512 in Q8.7 notation. This number fits into a single, signed, 16-bit value.

The Firmware

The following sections describe the firmware that computes a radix-2 FFT on a low-power µC. When the samples are read from the ADC, they are stored in the `x_n_re` array. This array represents the real values of x(n). The imaginary values, initialized to zero before the FFT begins, are stored in the `x_n_im` array. When the FFT is complete, the spectrum results will have replaced the original sampled values and be stored in `x_n_re` and `x_n_im`.

Gathering Samples

The FFT algorithm assumes that the samples are taken at a fixed sampling frequency. Gathering samples for an FFT can cause difficulties if not done carefully. For example, jitter in the sample interval causes errors in the FFT results and should be minimized.

A decision statement in the ADC sample loop can cause jitter in the sample interval. For example, our system reads signed, 8-bit samples from an ADC and stores them in an array of 16-bit variables. Two pseudo-code algorithms for performing this ADC read-and-store function are shown in **Listing 1**. The method presented in Algorithm 1 will cause jitter in the sample interval because a negative sample requires more time to read and store than a positive sample.

Listing 1. Two pseudo-code algorithms for ADC sampling are illustrated. The second algorithm does not cause the same problem as the first—jitter in the sample interval.

```
// ALGORITHM 1: INCONSISTENT SAMPLING
FREQUENCY — BAD!
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
```

```
     doADCSampleConversion()              //
Instruct ADC to sample Vin
     sample[i] = read8BitSampleFromADC() //
Read 8-bit sample from ADC
     if (sample[i] & 0x0080)              // If
the 8-bit sample was negative
          sample[i] = sample[i] + 0xFF00 //
Make the 16-bit word negative
end
// ALGORITHM 2: FIXED SAMPLING FREQUENCY —
GOOD!
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
     doADCSampleConversion()              //
Instruct ADC to sample Vin
     sample[i] = read8BitSampleFromADC() //
Read 8-bit sample from ADC
end
for i = 0 to (N-1)
begin
     if (sample[i] & 0x0080)              // If
the 8-bit sample was negative
          sample[i] = sample[i] + 0xFF00 //
Make the 16-bit word negative
end
```

Trigonometric Lookup Tables

The FFT algorithm uses lookup tables (LUTs) instead of calculating the value of cosine or sine. The declarations for the sine and cosine LUTs are given in **Listing 2**; comments in the actual firmware include source code for the program used to automatically generate these LUTs. Both LUTs have N / 2 elements because the indices of the twiddle factors vary from 0 to (N / 2) - 1 (see Figure 2).

Listing 2. LUTs are shown for sine and cosine functions.

```
const int cosLUT[N/2] = {+128,+127,+127, ...
,-127,-127,-127};
const int sinLUT[N/2] = {+0  ,+3  ,  +6, ...
,+9  ,  +6,  +3};
```

The arrays containing these LUTs are declared as `const`, forcing the compiler to store them in code space instead of data space. Because the LUT values must be in Q8.7 notation, they correspond to the actual cosine and sine values multiplied by 2^7.

Bit Reversal

The bit-reversal order (where N is known) can be calculated at runtime, indexed using a lookup table, or written directly with an unrolled loop. Each of these methods has trade-offs in regard to the size of the source code and execution speed. This FFT application performs bit reversal using an unrolled loop, which results in longer source code but faster execution. The code in **Listing 3** shows the implementation of this unrolled loop. Comments in the applications firmware include source code for the program that automatically generates this unrolled loop.

Listing 3. An unrolled loop with N = 256 is used for bit reversal.

```
i=x_n_re[  1]; x_n_re[  1]=x_n_re[128];
x_n_re[128]=i;
i=x_n_re[  2]; x_n_re[  2]=x_n_re[ 64];
x_n_re[ 64]=i;
i=x_n_re[  3]; x_n_re[  3]=x_n_re[192];
x_n_re[192]=i;
i=x_n_re[  4]; x_n_re[  4]=x_n_re[ 32];
x_n_re[ 32]=i;
...
i=x_n_re[207]; x_n_re[207]=x_n_re[243];
x_n_re[243]=i;
i=x_n_re[215]; x_n_re[215]=x_n_re[235];
x_n_re[235]=i;
i=x_n_re[223]; x_n_re[223]=x_n_re[251];
x_n_re[251]=i;
i=x_n_re[239]; x_n_re[239]=x_n_re[247];
x_n_re[247]=i;
```

The Radix-2 FFT Algorithm

After the samples have been reordered using bit reversal, the FFT can be computed. The firmware for this implementation of the radix-2 FFT performs the butterfly computations seen in Figure 2 with three main loops. The outside loop counts through the $\log_2(N)$ stages of the FFT computation. The inner loops perform the individual butterfly computations of each stage.

The heart of the FFT algorithm is the short block of code that performs each butterfly computation. This block, shown in **Listing 4**, is unfortunately the only nonportable firmware in this implementation. The `MUL_1` and `MUL_2` macros use the µC's hardware multiplier to perform multiplications in a single instruction cycle. The contents of these macros, which are specific to the MAXQ2000, can be fully examined in the actual firmware.

Listing 4. Butterfly computation is performed in C.

```
/* (1) Macro MUL_1(A,B,C): C=A*B      (result
in Q8.7)*/
/* (2) Macro MUL_2(A,C)  : C=A*last_B (result
in Q8.7)*/
MUL_1(cosLUT[tf],x_n_re[b],resultMulReCos);
MUL_2(sinLUT[tf],resultMulReSin);
MUL_1(cosLUT[tf],x_n_im[b],resultMulImCos);
```

```
MUL_2(sinLUT[tf],resultMulImSin);

x_n_re[b] = x_n_re[a]-
resultMulReCos+resultMulImSin;

x_n_im[b] = x_n_im[a]-resultMulReSin-
resultMulImCos;

x_n_re[a] = x_n_re[a]+resultMulReCos-
resultMulImSin;

x_n_im[a] =
x_n_im[a]+resultMulReSin+resultMulImCos;
```

Complex to Polar Conversion

To determine the magnitude for the spectrum of V_{IN}, we must convert the complex values of X(k) into polar notation. The firmware that implements this conversion is shown in **Listing 5**. The magnitude values replace the original results of the FFT that are no longer needed by the firmware.

Listing 5. FFT results are converted from complex to polar notation.

```
const unsigned char magnLUT[16][16] =
{
{0x00,0x10,0x20, ... ,0xd0,0xe0,0xf0},
{0x10,0x16,0x23, ... ,0xd0,0xe0,0xf0},
                 ...
{0xe0,0xe0,0xe2, ... ,0xff,0xff,0xff},
{0xf0,0xf0,0xf2, ... ,0xff,0xff,0xff}
};
                 ...
                 ...
/* Compute x_n_re=abs(x_n_re) and
x_n_im=abs(x_n_im) */
                 ...
                 ...
x_n_re[0] = magnLUT[x_n_re[0]>>11][0];

for(i=1; i<N_DIV_2; i++)
x_n_re[i] =
magnLUT[x_n_re[i]>>11][x_n_im[i]>>11];

x_n_re[N_DIV_2] =
magnLUT[x_n_re[N_DIV_2]>>11][0];
```

A two-dimensional LUT determines the magnitude instead of the computation from equation 4. The first index is 4 most significant bits (MSB) of the real part of the spectrum, while the second index is 4 MSB of the imaginary part of the spectrum. To obtain these 4 MSB, the signed, 16-bit values are right shifted 11 times. Before the real and imaginary parts of the spectrum can be used as indices, they are replaced by their absolute values. Therefore, the sign bit will be zero.

Because it is known from equation 6 that the magnitude of the spectrum is symmetric with respect to X(N / 2), only the magnitudes of the first (N / 2) + 1 spectrum values are converted to polar notation. Also, it can be shown that the imaginary parts of X(0) and X(N / 2) are always zero for real-valued input samples. These two magnitudes are therefore calculated separately. Comments in the actual firmware for this project include source code for the program that automatically generates the LUT for the magnitude of X(k).

Hamming or Hann Windows

The firmware for this project includes LUTs (in Q8.7 format) to apply a Hamming window or a Hann window to the input samples. Windowing is useful to reduce spectral leakage that can result from truncating x(n) in the time domain. The equations for the Hamming and Hann window functions are shown in equations 7 and 8, respectively.

$$h(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N - 1}\right) \qquad \text{(Eq 7)}$$

$$h(n) = 0.5\left[1 - \cos\left(\frac{2\pi n}{N - 1}\right)\right] \qquad \text{(Eq 8)}$$

Listing 6 shows the code for the implementation of these functions. Again, comments in the actual firmware for this project include source code for the program that automatically generates the LUTs for these windowing functions.

Listing 6. LUTs are shown for the implementation of Hamming and Hann window functions.

```
const char hammingLUT[N] = {+10, +10, +10, ...
,+10, +10, +10};

const char hannLUT[N]    = { +0,  +0,  +0, ...
, +0,  +0,  +0};
...
...
for(i=0; i<256; i++)
{
   #ifdef WINDOWING_HAMMING

MUL_1(x_n_re[i],hammingLUT[i],x_n_re[i]); //
x(n)*=hamming(n);
   #endif
   #ifdef WINDOWING_HANN
      MUL_1(x_n_re[i],hannLUT[i]),x_n_re[i]);
// x(n)*=hann(n);
   #endif
}
```

Testing the Results

To test the result of the FFT application, the firmware uploads the magnitude of X(k) to a PC using the µC's UART port. *FFT Graph*, software written specifically to read these magnitude values from the PC's serial port, graphs the magnitude of the calculated spectrum in real time. (This software is included with the firmware for this project.) **Figure 3** shows the results displayed from *FFT Graph* for four different input signals with the µC sampling the input voltage at 200ksps:

a) 4.3V DC signal

b) 50kHz sine wave

c) 70kHz sine wave

d) 6.25kHz square wave

What Is Next?

The interested reader can spend an unlimited amount of time optimizing and configuring this FFT implementation. Although the radix-2 algorithm was chosen for this article, other algorithms can dramatically reduce the number of additions and multiplications required. Many optimizations not presented in this article also exist for increasing the speed of a FFT. For example, with real-valued input samples, the imaginary part of the input samples is always zero, and only the first half of the spectrum is significant. Using this information, the first and last stages of the FFT can be optimized for faster execution, but more program space may be required.

The algorithm presented in this article is, however, a good starting point for an FFT algorithm written specifically for a low-power µC. For more information and implementation details, please review the well-commented firmware for this application.

Resources

Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Computation of Complex Fourier Series," *Mathematics Computation*, Vol. 19, pp 297-301, 1965.

Lemieux, Joe, "Fixed-point math in C," *Embedded Systems Programming*, October 2003.

Proakis, John G. and Dimitris G. Manolakism, *Digital Signal Processing Principles, Algorithms, and Applications*, 3rd Edition, Prentice Hall, 1996.

Smith, Steven W., *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd Edition, California Technical Publishing, 1999.

A similar article appeared in the October, 2005 issue of Embedded Systems Design.

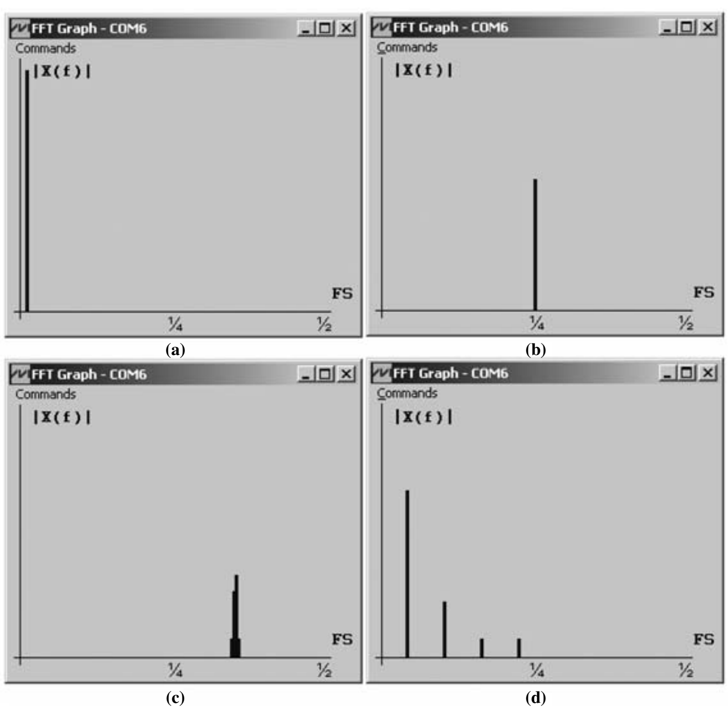


Figure 3. FFT Graph *is used to plot the results of spectra calculated by a low-power µC.*

DESIGN SHOWCASE

Precision Circuit Monitors Negative Supply Current

Supply-current monitoring is a necessary feature in high-reliability systems where excessive current can cause damage or compromise safety. Such systems avoid overload faults by monitoring their power supply and shutting it down before a fault occurs. Most current-monitoring ICs, however, are designed for positive-voltage supplies. For negative supplies, the circuit of **Figure 1** monitors load current and provides a proportional output voltage.

Voltages at the inverting and noninverting terminals of the op amp (IC1A) are forced to be equal by an active-feedback current mirror. $V_{R1} = V_{SENSE}$ and therefore:

$$I_{R1} = I_O \frac{R_{SENSE}}{R_1}$$

Three alternatives are now possible. You can convert the output current (I_{R1}) to voltage by connecting resistor R_O to ground, to V_{CC}, or to an inverting amplifier. Connecting R_O to ground (GND) eliminates the need for a positive supply. In that case, the output voltage is negative and proportional to load current:

$$V_O = -I_O \frac{R_{SENSE}}{R_1} R_O \qquad (R_O \text{ connected to GND})$$

You can connect R_O to V_{CC} for applications that require a positive output voltage, but the output will be referenced to V_{CC}:

$$V_O = V_{CC} - I_O \frac{R_{SENSE}}{R_1} R_O \qquad (R_O \text{ connected to } V_{CC})$$

To reference the positive output voltage to ground, you must use an inverting amplifier (IC1B), as shown in Figure 1:

$$V_O = I_O R_{SENSE} \frac{R_2}{R_1} \qquad (R_O \text{ connected to an inverting amplifier})$$

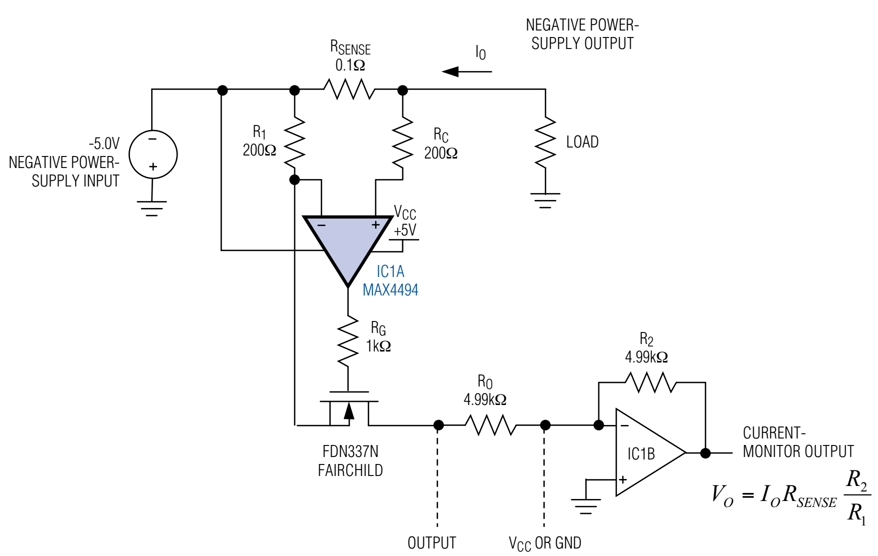


Figure 1. This current-sensing circuit monitors a negative power supply and provides a positive output voltage proportional to the load current.

DESIGN SHOWCASE

Note that R_O does not affect output voltage for the inverting-amplifier, but this resistor is usually needed for stability. R_G can be optional, but it also provides stability by isolating the op amp from the capacitive load of the MOSFET gate. Finally, R_C compensates for the op amp's input bias current.

Figure 2 shows measurement error vs. load current for the Figure 1 circuit. To ensure accurate current measurements, the resistors (except for R_G and R_C) should have a tolerance of 1% or better. R_{SENSE} must be rated to dissipate the power associated with high load currents.

A similar article appeared in the September, 2005 issue of Power Electronics Technology.

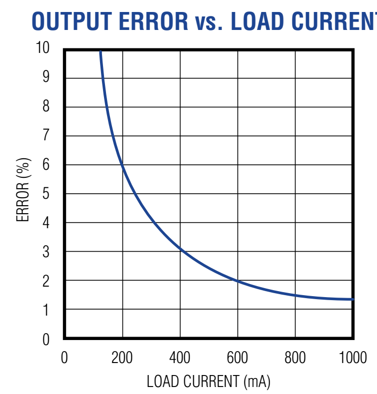


Figure 2. Error for the current sensor of Figure 1 is less than 2% at full scale, but the op amp's inherent input-offset voltage reduces the accuracy at lower levels of current.