SiRFstarIIe System Development Kit User's Guide

Part 1 – Software

SiRF Technology, Inc.
148 East Brokaw Road
San Jose, CA 95112 U.S.A.
Phone: +1 (408) 467-0410
Fax: +1 (408) 467-0420
www.sirf.com

1050-0035
May 2002, Revision 1.4

SiRF and SiRF identity are trademarks of SiRF Technology, Inc. This document contains information on a product under development at SiRF. The information is intended to help you evaluate this product. SiRF reserves the right to change or discontinue work on this proposed product without notice.

SiRFstarIIe System Development Kit User's Guide Part 1 – Software

© 2000-2002 SiRF Technology, Inc. All rights reserved.

About This Document

This document contains information on SiRF products. SiRF Technology, Inc. reserves the right to make changes in its products, specifications and other information at any time without notice. SiRF assumes no liability or responsibility for any claims or damages arising out of the use of this document, or from the use of integrated circuits based on this document, including, but not limited to claims or damages based on infringement of patents, copyrights or other intellectual property rights. SiRF makes no warranties, either express or implied with respect to the information and specifications contained in this document. Performance characteristics listed in this data sheet do not constitute a warranty or guarantee of product performance. All terms and conditions of sale are governed by the SiRF Terms and Conditions of Sale, a copy of which you may obtain from your authorized SiRF sales representative.

Getting Help

If you have any problems installing or using your System Development Kit, call or send an e-mail to the SiRF Technology Customer Support Group:

phone +1 (408) 467-0410

e-mail support@sirf.com

Contents

Preface xv

1. System Development Kit Overview 1-1

The S2SDK Development Board 1-1

S2SDK Connections and Functions 1-1

Default Jumper Settings for the S2SDK 1-3

The GSW2 Software 1-3

Standard Variable Types 1-4

GPS Core Overview 1-5

Module Interface Overview 1-5

User Interface Overview 1-5

Tasking Overview 1-6

Memory Overview 1-6

UART Overview 1-7

Toolkit Software 1-7

SiRFdemo 1-7

SiRFflash 1-8

SiRFtest 1-8

SiRFsig 1-8

Additional Utilities 1-8

SiRFstarIIe System Development Kit CD 1-9

2. Installation 2-1

Installing the S2SDK 2-1

Environment Considerations 2-1

Connecting the S2SDK 2-1

Installing the Toolkit Software 2-2

Installing ARM Development Tools 2-3

The ADS Development Environment 2-3

The ARM Multi-ICE and Software 2-4

3. Available System Resources 3-1

ROM/RAM Requirements 3-1

Stack Requirements 3-1

Adding Elements to Battery Backed SRAM 3-1

Check Space Left in Battery-Backed Memory 3-2

Add User Element to Battery-Backed Memory Structure 3-2

Add New Code to SRAM Access Functions 3-2

4. Software Build Process 4-1

Software Build Process and Variants 4-1

Creating an SDK Build Using ADS 4-2

Basic Compile Switches 4-5

5. Flash Programming 5-1

Downloading Software using SiRFflash 5-1

Reading Flash Memory 5-3

Supporting Different Flash Types 5-4

6. Development and Debugging 6-1

Adding a User Version String 6-1

Multi-ICE Debugging 6-4

ADS and Multi-ICE Debugging 6-4
PRINTF Debugging 6-9
NMEA Debug Output 6-10
SiRF Binary Debug Output 6-11
S2SDK LED Activation 6-12
GPS Performance Testing 6-12
Using PROCOMM to Send NMEA Messages 6-13
NMEA Checksum Utility 6-13
Uploading Code to SiRFstarIIe without SiRFflash 6-14
Internal Boot Operation 6-15
7. Memory BUS and Components 7-1
Memory 7-1
Scatter Loading Files 7-1
Memory Areas in the GSP2e 7-5
Memory Map 7-7
Remap Function 7-7
Memory Map Configuration 7-8
8. Input/Output Messages 8-1
Changing Default Message Settings 8-1
Default Output Protocol 8-1
Default Baud Rate 8-2
Default Message Output Rates 8-3
Adding New Input/Output Messages 8-6
Limitations on Message Length 8-7
SiRF Binary 8-7
NMEA 8-17
9. Low Power Operation 9-1
TricklePower. 9-1
ECLK TricklePower 9-2
GPSCLK TricklePower 9-3
Enabling/Disabling TricklePower. 9-4

Effect of TricklePower on Message Rates 9-9
Push-to-Fix 9-10
Enabling/Disabling Push-to-Fix 9-10
Setting Low Power Acquisition Parameters 9-13
10. User Tasks, ASIC Interrupts, and the Scheduler 10-1
ASIC Interrupts. 10-2
Timer Interrupt 10-3
UART Interrupt 10-3
Low Power Operation Interrupt 10-3
Beacon Interrupt 10-3
Adding a User Task. 10-4
Start/Stop GPS Functions 10-5
11. DGPS Operation 11-1
Setting Differential Correction Source 11-1
SiRF Binary Messages for Differential 11-2
Set DGPS Source Control (MID 0x85). 11-2
DGPS Status (MID 0x1B) 11-3
Module Interface Routines for Differential. 11-4
12. Adding a New User Protocol 12-1
Protocol Implementation. 12-1
USER1 Protocol 12-4
Single Character Delivery. 12-12
13. GPIO Lines, Throughput and Wait States 13-1
GPIO Lines 13-1
Chip Select Wait States. 13-5
A. Converting UTC Time to GPS Week Number and TOW A-1
B. SiRF Binary Messaging Functions B-1
SiRF Binary Messages B-1
Functions for Input Messages B-4
Functions for Output Messages B-8
C. Module Interface Details C-1

Module Interface Events C-1
Module Interface Routines C-2
GetCOG C-2
GetDate C-3
MI_GetDatum C-4
ConvertTowtoUTC C-4
ConvertECEFtoLTP C-5
ConvertLTPtoECEF C-5
MI_Get50Bps C-6
MI_GetAlm C-7
MI_SetAlm C-7
UI_GetCPUClkRate C-8
MI_GetClkBias C-8
MI_GetClkDrift C-9
MI_GetDgpsSrc C-9
MI_SetDgpsSrc C-9
MI_GetDatum C-11
MI_SetDatum C-11
MI_GetDgps_Mode C-11
MI_SetDgps_Mode C-11
MI_GetDgpsAlm C-12
MI_GetDgpsCorrAge C-13
MI_GetDgpsBeacon C-14
MI_SetDgpsBeacon C-14
MI_GetDgpsSpecialMsg C-15
MI_GetDgpsStationID C-16
MI_GetDgpsStationPos C-16
MI_GetDopMask C-17
MI_SetDopMask C-17
MI_GetElevMask C-18
MI_SetElevMask C-18

MI_GetEph C-19
MI_SetEph C-19
MI_GetEstGPSTime C-20
MI_GetLPAcqParams C-20
MI_SetLPAcqParams C-20
MI_GetLowPower C-21
MI_SetLowPower C-21
MI_GetNavDops C-23
MI_GetNavFom C-24
MI_GetNavInit. C-25
MI_SetNavInit C-25
MI_GetNavMode C-26
MI_GetNavModeMask C-28
MI_SetNavModeMask C-28
MI_GetNavList C-29
MI_GetPosEcef C-30
MI_GetPositionLTP C-31
MI_GetPwrMask C-31
MI_SetPwrMask C-31
MI_GetRawTrkData. C-32
MI_GetSWVersion. C-33
MI_GetStaticNav C-34
MI_SetStaticNav C-34
MI_GetThroughput C-35
MI_GetGPSTime C-35
MI_GetTrkData C-35
MI_GetTrkStateList C-37
UI_GetUartClkRate C-37
MI_GetUTC. C-38
MI_GetVelEcef C-39
MI_GetVelNed C-39

MI_GetGSPVersion C-40
MI_GetVisList C-40
MI_SetComm C-41
MI_SetDgpsCorrs C-42
MI_SetNmeaProto C-43
MI_SetUiProto C-44
MI_SetBaud C-45
MI_GPSStop C-46
MI_GPSStart C-46
MI_GetEstPosError C-47
MI_GetPtfPeriod C-48
MI_SetPtfPeriod C-48
MI_GetTestModeData C-48
MI_SetTestMode C-48
MI_GetUserDRTimeout C-50
MI_SetUserDRTimeout C-50
MI_GetUserParams C-50
MI_SetUserParams C-50
MI_LpDbgOutput C-51
MI_GetUtcOffset C-51
MI_SetUtcOffset C-51
MI_SetSbasPrn C-52

D. Error Ellipse Functions D-1

Computation of Navigation Error Ellipse D-1

E. File Descriptions E-1

File Organization E-1
Start-Up E-1
GPS Core E-1
Tasking E-2
UART E-2
User Interface E-2

Memory E-2
Module Interface (including utility files) E-3
Individual File Descriptions E-3
Non SDK Source/Build Files E-10
F. Acronyms, Abbreviations and Glossary F-1

Figures

Figure 1-1 S2SDK Development Board with attached RFBL daughter board. . 1-1

Figure 1-2 SiRFstarIIe SDK Software Architecture 1-4

Figure 2-1 S2SDK Connections. 2-2

Figure 2-2 WinZip Self-Extractor Window 2-3

Figure 4-1 The SDK Code Directory Structure 4-2

Figure 4-2 Metrowerks CodeWarrier Start-Up Window used in the ADS Environment 4-3

Figure 4-3 Window Displaying How to Make a HwtFlash Variant 4-3

Figure 4-4 ADS Errors and Warnings Window. 4-4

Figure 4-5 Setting ADS Compiler Preprocessor #DEFINEs 4-7

Figure 5-1 The SiRFflash Software 5-2

Figure 6-1 Multi-ICE Server Program After Successful Connection 6-4

Figure 6-2 The *Load Debug Symbols* dialog.. 6-5

Figure 6-3 Debugger window after a the code has been stopped. 6-6

Figure 6-4 Multi-ICE Server Program After Successful Connection 6-7

Figure 6-5 Debugger window after a the code has been stopped. 6-9

Figure 9-1 Diagram for ECLK TricklePower Showing the Various States and Approximate Current Consumption. 9-3
Figure 10-1 Workings of the Scheduler 10-2
Figure 12-1 Protocol Redirection. 12-2
Figure 12-2 Overview of Uart Structure and Effect of Changing Protocol 12-4
Figure 12-3 Output Sequence for USER1 Protocol 12-5

Tables

Table 1-1 Additional Utilities 1-8

Table 1-2 CD Directory Structure 1-9

Table 4-1 User defined and changeable preprocessor options.. 4-5

Table 4-2 Essential preprocessor options that are not to be changed. 4-6

Table 4-3 Preprocessor options that are not supported and are not to be used. 4-6

Table 7-1 Memory Map for Internal Boot 7-6

Table 7-2 Memory Map for External Boot. 7-7

Table 7-3 Memory Map Without Remap 7-8

Table 7-4 Memory Map After Remap 7-9

Table 7-5 Memory Map Without Remap 7-9

Table 7-6 Memory Map After Remap 7-9

Table 9-1 Summary of Clock Sources for ECLK TricklePower States 9-4

Table 9-2 Summary of Clock Sources for GPSCLK TricklePower States. . . . 9-4

Table 13-1 GPIOs and Alternate Functions on the GSP2e. 13-3

Table 13-2 Clocks and Wait States to Access External Memory, with and without Cache Enabled 13-6

Table 13-3 Multiplier for Number of Clocks Required for Memory Access . . . 13-6

Preface

The *SiRFstarIIe System Development Kit User's Guide Part 1 – Software* describes the architecture, implementation and modification of the SiRF Software which runs on SiRFstarIIe enabled platforms. This manual guides you through compiling and loading a standard SiRF build to the generation of custom code for user specific applications. A great amount of flexibility and code infrastructure has been provided by SiRF to aid in the development of custom messaging, I/O protocols, low power operation, and user tasking.

Skills Required to Use the System Development Kit

The System Development Kit (SDK) may be used at very different levels. At a minimum, a developer who intends to use the SDK must be proficient with the C programming language and standard compiler/linker tools. This is enough knowledge to make minor changes to the operation of the SiRF software such as modifying or adding serial messages.

Read and understand the *SiRFstarIIe Evaluation Kit User's Guide*. It is highly recommended that you spend some time using the receiver module with the supplied `Sirfdemo` software. This is important to understanding the GPS and default module behavior because some topics in this manual are described at great length in the *SiRFstarIIe Evaluation Kit User's Guide*.

To modify hardware platforms, you will need a hardware engineer to design and debug the new platform. To make hardware changes to the SiRF platform you must have experience developing applications for embedded systems and knowledge of serial communications, In Circuit Emulators (ICE), device programmers, CPU startup code issues, and basic digital design. GPS knowledge is a plus but not necessarily a requirement. The depth of the required skills directly depends on the complexity of the application that you are developing.

How This Guide Is Organized

The *SiRFstarIIe System Development Kit User's Guide Part 1 – Software* is the first part in a series of three manuals that complete the *System Development Kit User's Guide*. The remaining parts provide information about the GSP2e chip and the S2SDK hardware platform.

The *SiRFstarIIe System Development Kit User's Guide Part 1 – Software* is organized in the following manner:

Chapter 1, "System Development Kit Overview" provides a brief overview of the SiRFstarIIe System Development Kit including a board description and the architecture.

Chapter 2, "Installation" describes the installation of the S2SDK board, the software, and the System Development Kit software tools.

Chapter 3, "Available System Resources" lists the amount of RAM/ROM, stack and battery-backed memory that is available for custom development.

Chapter 4, "Software Build Process" demonstrates how to build the Software using the ARM development tools. This chapter also lists the SiRF-defined preprocessor definitions and their functions along with implementation details.

Chapter 5, "Flash Programming" provides instructions for using the flash programming tool - SiRFflash.

Chapter 6, "Development and Debugging" provides an overview of debugging strategies, capabilities and tools for the S2SDK board.

Chapter 7, "Memory BUS and Components" details the memory map of the SiRFstarIIe for internal and external boot mode and provides information on the scatter loading files used to define the memory areas and locations at run-time and load-time.

Chapter 8, "Input/Output Messages" describes how to add new SiRF binary or NMEA type input and output messages. Explains how to set different protocols as default and change the default Baud rates. This chapter also describes how to set the output rates for different messages.

Chapter 9, "Low Power Operation" provides an overview of the low power operation of the SiRFstarIIe, and lists the relevant Low Power parameters and how to set them.

Chapter 10, "User Tasks, ASIC Interrupts, and the Scheduler" describes the implementation of user tasks and how the scheduler functions.

Chapter 11, "DGPS Operation" explains how to set up the default differential source and details new messages for use with a differential beacon receiver.

Chapter 12, "Adding a New User Protocol" provides an overview of the user interface and the UART structure used for serial communication. This chapter also describes how to implement a user protocol using the code infrastructure provided by SiRF.

Chapter 13, "GPIO Lines, Throughput and Wait States" provides general information on the startup code, including the setting of GPIO functions and how to change the source of the CPU clock and clock divider.

Appendix A, "Converting UTC Time to GPS Week Number and TOW" explains how to transform GPS time (normally presented as seconds into week) into UTC time.

Appendix B, "SiRF Binary Messaging Functions" describes the various SiRF Binary I/O message identifiers including those reserved for custom development. This chapter also explains the various functions used in SiRF binary I/O, grouping them according to type, return value and parameters.

Appendix C, "Module Interface Details" provides an overview of the various Module Interface routines and the interface structures that are used in the function calls.

Appendix D, "Error Ellipse Functions" demonstrates how to calculate an error ellipse based on the various DOP values provided by the receiver.

Appendix E, "File Descriptions" provides a brief description of the source files included in the SiRFstarIIe System Development Kit.

Appendix F, "Acronyms, Abbreviations and Glossary" provides a detailed summary of relevant terms and phrases.

Related Manuals

You can also refer to the following literature for additional information:

- *SiRFstarIIe System Development Kit User's Guide Part 2– GSP2e Chip*
- *SiRFstarIIe System Development Kit User's Guide Part 3– S2SDK Board*
- *SiRFstarIIe Evaluation Kit User's Guide*
- *ARM ADS User Guide*
- *ARM ADS Reference Guide*
- *ARM Multi-Ice User Guide*
- *NMEA 0183 Standard for Interfacing Marine Electronic Devices*
- *RTCM Recommended Standard for Differential Navstar GPS Service,* RTCM Special Committee No. 104

Troubleshooting/Contacting SiRF Technical Support

Address:

SiRF Technolog.y Inc.
148 East Brokaw Road
San Jose, CA 95112 U.S.A.

SiRF Technical Support:

Phone: +1 (408) 467-0410 (9 am to 5 pm Pacific Standard Time)

Email: `support@sirf.com`

General enquiries

Phone: +1 (408) 467-0410 (9 am to 5 pm Pacific Standard Time)

Email: `gps@sirf.com`

Helpful Information When Contacting SiRF Technical Support

Receiver Serial Number: _______________

Receiver Software Version: _____________

SiRFdemo Version: ___________________

System Development Kit Overview 1

The SiRFstarIIe Software Development Kit (SDK) is a complete and ready to use tool suite specifically designed to help users bring their SiRF-based GPS product to the market as rapidly and efficiently as possible.

Included with the SDK is a GSP2e based large form factor development board (S2SDK), the latest SiRFstarII GPS software (GSW2) provided in a mix of source and object code, PC software for testing and evaluation, and other information necessary to design and build GPS platforms based on the SiRFstarIIe chipset.

The S2SDK Development Board

The S2SDK provides a configurable GPS receiver hardware platform for software development and hardware prototyping. The S2SDK allows the GSW2 software to run and test the GPS receiver prototype prior to completion of its design. Additionally, the S2SDK allows for testing of different hardware configurations such as different processor bus widths, memories, wait-state numbers, and other system configuarations.

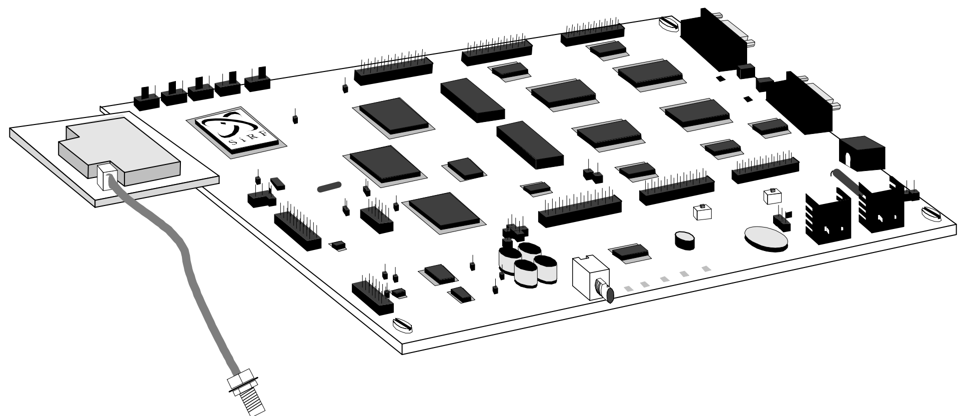

Figure 1-1 S2SDK Development Board with attached RFBL daughter board.

S2SDK Connections and Functions

The following section includes information about each of the connections present on the S2SDK and general board functions.

Com A and Com B

Two standard RS232 DB9 female communication ports are provided for S2SDK configuration, data logging, or to upgrade receiver software. Each port can be configured to operate in NMEA or SiRF protocol, or accept RTCM input data. However, you may not run the same protocol on multiple ports.

The following table lists the default settings for each of the communication ports.

Parameter	Com A	Com B
Input Protocol	SiRF Binary	RTCM SC-104
Output Protocol	SiRF Binary	None
Baud Rate	38400	9600
Parity	None	None
Stop Bits	1	1
Data Bits	8	8

The following table describes the pin-out configuration for Com A and Com B.

Pin Number & Name	Description
Pin 1 [DCD]	Connected to pin 4
Pin 2 [Rx data]	Transmit data from the GPS receiver
Pin 3 [Tx data]	Receive data to the GPS receiver
Pin 4 [DTR]	Connected to pin 1
Pin 5 [GND]	Connected to signal ground
Pin 6 [DSR]	Not connected
Pin 7 [RTS]	Connected to pin 8
Pin 8 [CTS]	Connected to pin 7
Pin 9 [RI]	Not connected

Antenna Options

Two antenna connections (female SMA and female BNC) are provided to allow different combinations of GPS and radiobeacon inputs. The SMA connector connects to a GPS-only antenna. The BNC connector connects to a combined GPS and beacon antenna, or a beacon-only antenna. There are three different combinations for GPS and beacon reception. These are:

- GPS antenna connected to SMA input.
- GPS antenna connected to SMA input and beacon antenna connected to BNC input.
- Combined GPS and beacon antenna connected to the BNC input.

Power

Located on the side of the S2SDK is a red LED that indicates when power is being applied to the board. There is no on/off switch. Required power input is 9 V and typically draws 160 mA.

LED Data Indicators

Green LEDs on the side of the board indicate whether data is being received or transmitted through ports A or B. These LEDs are a useful visual indicator when debugging problems or verifying correct receiver operation.

Default Jumper Settings for the S2SDK

The S2SDK is defaulted to a 16-bit external memory bus and external boot mode. For more information on the S2SDK and jumper settings see the *SiRFstarIIe System Developer Kit User's Guide Part 3 - S2SDK Board*. The default jumper settings for the S2SDK are provided below for convenience.

Note – Jumpers 21 to 25 are tied to data lines that are sampled at start-up to determine system configuration.

Jumper	Shorting Pin	Description
J21	1-2	DATA0: Internal Boot (nCS0)
J21	2-3*	DATA0: External Boot
J22	1-2	DATA1: reserved
J22	2-3*	DATA1: reserved
J24	1-2	DATA8: Big Endian
J24	2-3*	DATA8: Little Endian
J25:J23	2-3:2-3	DATA15,7: 8 bit Bus Mode
J25:J23	2-3:1-2*	DATA15,7: 16 bit Bus Mode
J25:J23	1-2:2-3	DATA15,7: 32 bit Bus Mode
J25:J23	1-2:1-2	DATA15,7: Illegal Bus Mode

* Denotes a Default Setting

Note – Three other groups of pins have jumpers provided. JP4 is default to 2-3 and provides 3V power to the GPS antenna. J29 is default to 2-3 and enables battery back-up. JP3 must not have any pins connected.

The GSW2 Software

The GSW2 software is designed to enable a significant amount of user customization. The SDK is offered as a combination of object files and source files in the C programming language. Figure 1-2 shows the software architecture and what parts of

the software can be modified. The shaded parts of the diagram are provided in object form only, while the unshaded parts are provided as source files. Figure 1-2 shows a general overview of the various parts.

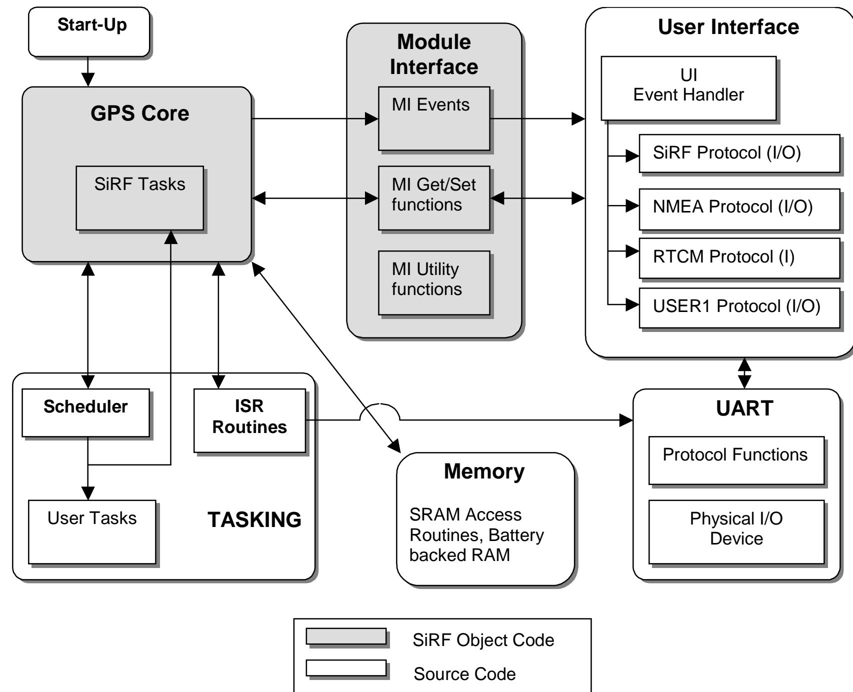

Figure 1-2 SiRFstarIIe SDK Software Architecture

Note – File descriptions and associations based on Figure 1-2 are given in Appendix E, “File Descriptions.”

The SDK code has been developed using the ARM development environment ADS (ARM Developers Suite). It is intended that all software development is completed using ADS and no other compiler option will be supported.

Certain portions of the receiver code are supplied in C programming language source format, and can be modified for specific user needs. The SiRF reference design uses the ARM7TDMI microprocessor and SiRF object code is supplied pre-compiled for this specific processor type. Using an appropriate compiler, you can process and link the modified source files to the existing set of object files to create a new executable file. This build process is described in Chapter 4, “Software Build Process.”

Standard Variable Types

Since different compilers and CPUs have different sized native variable types, SiRF has created typedefs which are used to specify variables of an exact size (e.g., `UINT32`, `UINT16`, `INT16`, etc.). These standard types are defined in `STDTYPE.H` for different hardware platforms and different compilers. It is highly recommended that these typedefs are used.

GPS Core Overview

The GPS Core module is unavailable for modification and includes GPS navigation and tracking code. It also contains the Receiver manager that controls the satellite search strategies and channel assignments. Interaction with the GPS Core is essentially limited to the start-up code, the Module Interface routines, and elements of the User Interface Memory structure (`UI_SRAM`). Since the code in the GPS Core cannot be modified, it is not described in this manual, but information may be found in any general book on GPS tracking and navigation. The start-up code initializes GSP chip registers and software data structures, including serial interface and memory chip selects.

Module Interface Overview

The Module Interface module is available as object code and provides an interface mechanism between the GPS Core and the User Interface Module controlling the I/O protocol for communicating with external devices. In general, the User Interface is driven by events (`MI_EVENT`) that are signaled by the GPS Core (a function call to `UI_EVENT()` in `UI_MSG.C`). When these events are received, the generic UI manager code determines the current protocol and calls an appropriate function in that protocol to handle the event. Inside the current I/O protocol, the event may signal that system information is supposed to be output. The current protocol can then use the Module Interface Get commands (`MI_Get###`) to determine state information and populate the appropriate output message. When a message is received, and the current protocol determines it is valid, the protocol can use Module Interface Set routines (`MI_Set###`) to change the state of the system. The Module Interface events, Get/Set functions and Utility functions are explained in Appendix C, "Module Interface Details."

User Interface Overview

The User Interface is developed to enable customized message input/output while still maintaining a generic interface to the GPS Core. The User Interface section is the most common point for modification of the SDK software. The User Interface is based on a series of protocols, each of which must have a defined set of functions for controlling I/O. The User Interface works through a generic interface (`UI_MSG.C`) and a series of function redirections based on the current protocol (set by the user, for default selection see "Basic Compile Switches" on page 4-5). See Chapter 13, "GPIO Lines, Throughput and Wait States" for details on the implementation of the User Interface. As mentioned in "Module Interface Overview" on page 1-5, the User Interface is mainly driven by Events generated by the GPS Core, but also by interrupts generated by the UART code for incoming messages.

There are currently four available protocols:

- SiRF Binary
- NMEA (ASCII)
- USER1
- RTCM (input only)

SiRF Binary and NMEA are currently fully functional and populated with messages while the USER1 protocol is basically a shell maintained for custom development. RTCM is available for reception of differential corrections. If new user messages are desired, SiRF strongly recommends adoption of the SiRF protocol since it is designed and tested for robustness in the field, and also lends itself to easy extension or modification of the current message base. This protocol also contains GPS messages that are essential for gauging the GPS performance of the unit. The SiRF binary protocol format is described in detail in Appendix B, "SiRF Binary Messaging Functions," Appendix C, "Module Interface Details," and in the *SiRFstarIIe Evaluation Kit User's Guide*. Adding a new message in SiRF protocol is covered in "SiRF Binary" on page 8-7. The NMEA protocol is covered in detail in the *SiRFstarIIe Evaluation Kit User's Guide* and adding new NMEA messages is covered in "NMEA" on page 8-17

Adding a new User Protocol can be an involved task depending on the extent of the differences between the desired user protocol and either SiRF binary or NMEA. See Chapter 12, "Adding a New User Protocol" for an example of adding a user protocol.

Tasking Overview

The SDK software enables some task-based user code to be implemented through the use of a scheduler. The scheduler utility is activated at periodic intervals to examine the task queue and determine if a higher priority task than the currently active one has been scheduled. If it determines that there is a pending task with a higher priority, it suspends the current task and activates the higher task. Currently, the scheduler is activated by the 100 ms interrupt resulting in a 100 ms time slicing capability. The SiRFstarIIe enables a higher rate of time-slicing using different counters, but extreme care must be exercised since this may detrimentally effect the GPS operation. The scheduler and adding a new user task is described in Chapter 10, "User Tasks, ASIC Interrupts, and the Scheduler." The implementation of the scheduler enables execution of user tasks during low-power operation when the CPU would normally be in standby mode.

Warning – GPS is extremely time critical and if a user task interferes with the GPS operation then tracking and navigation is adversely affected.

Memory Overview

This data storage area is preserved through battery backup during power off and is integrity checked using a CRC. This storage area is referred to in the code as SRAM structure, although it need not actually be SRAM, merely non-volatile in the sense that its contents are kept valid using a battery backup. Data elements stored in this segment include module state, protocol state (such as what current protocol is selected), message state (such as what messages are enabled), current Baud rate, etc. This information is used upon startup to initialize the module. Available space in the SiRFstarIIe battery-backed RAM is extremely limited, see "ROM/RAM Requirements" on page 3-1 for details. Within the space available, the user may add variables to this segment for use with the user protocol. For example, this enables the users to preserve which messages are enabled and maintain that state during power off

periods. Read access is available globally to these status variables, but write access must be restricted to calls to `UI_SetUiSram()` function so that CRC is properly recalculated after modifying a data element.

UART Overview

The SiRF reference design uses two serial communication ports, and this portion of code does full UART management, including all low level transmits and receive routines. A design overview of UART functions is given in Chapter 12, “Adding a New User Protocol.” The modification of the UART routines is recommended only for advanced users and could involve significant development time. An ARM Multi-ICE debugging tool must be considered for this type of development.

UART functions are provided to put data into the output queue and register callback functions that are called when a specific message ID is received. Callback functions may then read the data out of the buffer. To send a message, the code allocates a buffer, fills the buffer, and sends the buffer to the output queue. All buffer management to maintain free buffer lists, input queues, output queues, allocate, and free buffers are maintained by the SiRF supplied source code. Although internal buffers are of a fixed length, you can support messages that are large enough to span buffers on input and output messages.

Toolkit Software

The SiRFstarIIe PC Software is comprised of computer-based software utilities that are used for Evaluation Receiver operation, data logging, and data analysis. All software can be found on the SiRFstarIIe System Development Kit CD and include:

- SiRFdemo
- SiRFflash
- SiRFtest
- SiRFsig
- Additional Utilities

Note – Documentation for SiRFdemo, SiRFsig, and the additional utilities can be found in the *SiRFstarIIe Evaluation Kit User’s Guide.*

SiRFdemo

SiRFdemo is the Evaluation Receiver configuration and monitoring software. This software can be used to monitor real-time operation of the Evaluation Receiver, log data for later analysis, and configure the Evaluation Receiver operation. See the *SiRFstarIIe Evaluation Kit User’s Guide* for more information on the use and operation of SiRFdemo software.

SiRFflash

SiRFflash is a tool that is provided with the SDK to enable users to download an S-record file into the flash memory of the S2SDK or the Evaluation Receiver. In addition to downloading an S-record, SiRFflash also provides the user with greater flexibility when programming flash, as well as provide flash reading ability.

For complete information about SiRFflash operation, see Chapter 5, “Flash Programming.”

SiRFtest

SiRFtest provides the ability to quickly and effectively test whether a production board is operational. Using SiRF Binary Protocol information, test criteria can be defined and used to pass or fail the operation of the GPS board being tested.

The SiRFtest utility is on the SiRFstarIIe System Development Kit CD. For complete information about SiRFtest and its operation, see the *SiRFstarIIe System Development Kit User’s Guide Part 3 - S2SDK Board*.

SiRFsig

SiRFsig software enables you to analyze data that is collected in the field. SiRFsig analysis data includes antenna modeling, satellite tracking abilities, static and kinematic accuracy results, and time to first fix. See the *SiRFstarIIe Evaluation Kit User’s Guide* for more information on the use and operation of the SiRFsig software.

Additional Utilities

In addition to the main software tools, Table 1-1 lists other useful executables that are also provided.

Table 1-1 Additional Utilities

Executable	Function
Summary	Summarizes collected data
Parser	Separates collected data into different files of similar data types
Conv	Converts between ECEF (Earth Centered Earth Fixed) XYZ coordinates and WGS84 (World Geodetic Spheroid 84) coordinates.
Fixanal	Calculates TTFF (Time to First Fix) statistics
Cksum	Calculates checksum values
Datum	Converts between different datums
Calcpsr	Computes GPS measurement data and ephemeris parameters from raw data

See *SiRFstarIIe Evaluation Kit User’s Guide* for more information on the use and operation of each of the provided executables.

SiRFstarIIe System Development Kit CD

The CD that is included in the SiRFstarIIe System Development Kit contains the software tools, documentation in .pdf format, data set examples, current version of Evaluation Receiver software, and the SDK source and object code. The directory structure of this CD and a description of the contents are provided in Table 1-2.

Table 1-2 CD Directory Structure

Directory	Content
Application Notes	The collection of applicable Application Notes in PDF format.
Documentation	PDF format of the SiRFstarIIe System Development Kit User's Guide and other related documentation.
PC Software	All PC software provided with the System Development Kit. This includes SiRFdemo, SiRFsig, SiRFflash, and other useful utilities. Example data for SiRFsig is also provided.
Receiver Software	The receiver software that is currently loaded onto the Evaluation Receiver and the S2SDK.
Reference Designs	Information for the S2AM and S2AR reference designs including BOM's and schematics.
SDK_Code	The SDK source and object code of the GSW2 software.

Installation 2

This chapter provides instructions and requirements for installing the Toolkit software and the S2SDK hardware.

Installing the S2SDK

The S2SDK is shipped configured to perform in the same manner as the Evaluation Receiver shipped with the Evaluation Kit. The Flash memory contains the software and all jumpers are set for normal GPS operation.

Environment Considerations

The S2SDK board is intended to be used in a development environment and has not been designed for field operation or testing. Hence, when using the S2SDK, you must avoid these conditions:

- Exposure to water or moisture
- Extreme temperatures (-40 to +85 degrees C)
- Any vibration
- Electrostatic environments

Connecting the S2SDK

To connect the cables for operation of the S2SDK board:

1. Connect the GPS antenna to the antenna input (SMA connector) on the S2SDK shown in Figure 2-1.

Note – This is not necessary if you do not require any GPS signal.

---

2. To receive DGPS corrections from a radiobeacon network, connect a beacon antenna to the BNC connector as shown in Figure 2-1.
3. Connect one end of the serial cable to the appropriate communications port on your computer.

4. Connect the other end of the serial cable to Com port A (P1) on the S2SDK as shown in Figure 2-1.

5. Use the 110-220V power adapter to apply power to the S2SDK.

Note – The SiRFdemo software can be used to verify the operation of the S2SDK. For full instructions on the use of SiRFdemo, please see the *SiRFstarIIe Evaluation Kit User's Guide*.

.

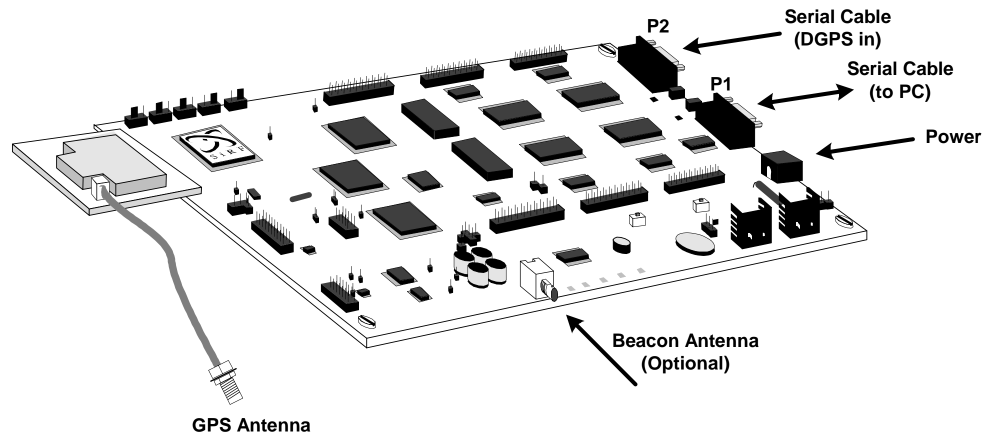

Figure 2-1 S2SDK Connections

Installing the Toolkit Software

This section describes how to install the System Development Kit Toolkit software.

To install the toolkit software:

1. Insert the SiRFstarIIe System Development Kit CD into your CD-ROM drive.

2. Copy the directory and contents of the `PC Software` directory from the SDK CD to your hard drive.

3. To install SiRFflash, double click on the `sirfflash204.exe` file. The *WinZip Self-Extractor* window is displayed as show in Figure 2-2.

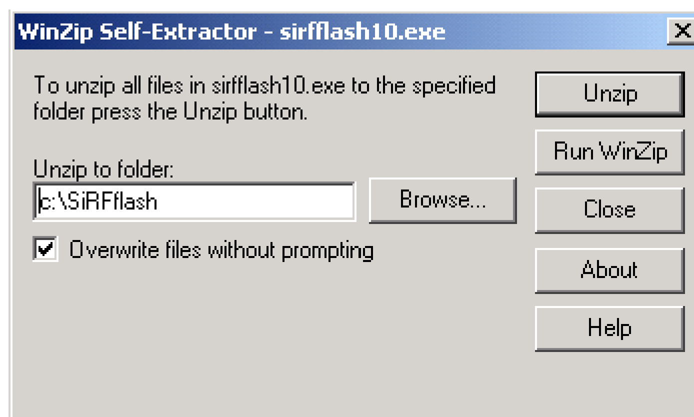

Figure 2-2 WinZip Self-Extractor Window

4. Use the *Browse...* button to locate the directory you would like SiRFflash installed in. Otherwise, type in the path name in the *Unzip to folder:* field.
5. Click on the *Unzip* button to extract all SiRFflash files to the selected directory.
6. To start SiRFflash, double-click on the `SiRFflash.exe` file.

   You may want to create a shortcut on your desktop and use this to start SiRFflash.
7. To start SiRFtest, double-click on the `sirftest.exe` file.

   You may want to create a shortcut on your desktop and use this to start SiRFtest.
8. To start SiRFdemo, double-click on the `sirfdemo.exe` file.

   You may want to create a shortcut on your desktop and use this to start SiRFtest.

Note – For information regarding the operation of SiRFsig, please refer to the SiRFstarIIe Evaluation Kit Users Guide.

Installing ARM Development Tools

Installation information for the ADS development environment is provided below as well as information about copying the required SDK files.

Note – For detailed installation information about the ARM development environments, refer to the appropriate ARM manual.

The ADS Development Environment

This section describes the steps required to setup an ADS development environment and the provided SiRF code:

1. Install the ARM ADS development environment.

A typical installation places the ADS software in the directory `C:\Program Files\ARM\ADSv1_2\BIN`, sets the path to the directory `C:\Program Files\ARM\ADSv1_2\BIN`, and sets the environment variables: `ARMCONF`, `ARMDLL`, `ARMHOME`, `ARMLIB`, and `ARMINC`. If a previous ARM ADS version exists on your PC, remove all paths to the old ADS software. Refer to the ARM ADS installation manual for full installation information.

Note – If you have a previous version of the ARM ADS software, it is recommended that the old version be un-installed before the new version is installed. Please make sure that the ARM license file is preserved for the upgrade.

---

2. Insert the SiRFstarIIe System Development Kit CD into the CD-ROM drive. Examine the contents of this CD and copy the contents of the `SDK_CODE\ADS` directory (including all files and subdirectories) into your development directory (e.g., "`C:\SiRFDev`").

3. Change the properties of the copied files to Read/Write.

   Since these files were copied from a CD, all files and subdirectories are Read Only. It is necessary to change all files to Read/Write before attempting to begin work. This can be done using Microsoft Explorer or another file management tool.

The ARM Multi-ICE and Software

This section describes the steps required to setup the ARM Multi-ICE interface unit and the required software. To setup the Multi-ICE and the software:

1. Install the Multi-ICE software from the ARM Multi-ICE Installation CD.

   For more information about the Multi-ICE software installation process see the *ARM Multi-ICE User Guide*.

2. Connect one end of the parallel cable to the parallel port of the host computer and the other end of the cable to the Multi-ICE interface unit.

3. Connect one end of the JTAG cable to the JTAG connector on the Multi-ICE interface unit, and the other end of the cable to the JTAG connector on the S2SDK development board.

   For more information about connecting the Multi-ICE interface unit to the target hardware platform see the *ARM Multi-ICE User Guide*.

4. Connect the power supply to the S2SDK development board.

5. Verify that the AXD Debugger for the ADS software is configured with the `Multi-ICE.dll` by starting the AXD Debugger and selecting Options | Configure Target...

   Click `ADD` and select the `Multi-ICE.dll` in the Multi-ICE installation directory. For more information about configuring the AXD Debugger see the *ARM Multi-ICE User Guide*.

Once the Multi-ICE interface unit is connected and the software is installed on the host PC correctly, a quick operational verification can be performed. To verify the operation of the Multi-ICE:

1. Start the Multi-ICE server application.
2. Reset the S2SDK development board by cycling the power on the board.
3. Auto-configure the Multi-ICE server application by selecting File | Auto-Configure

   The target ID is displayed by the Multi-ICE server application. The ID should be ARM7TDMI.

Note – If the target ID is not shown, it is highly probable that the Multi-ICE server is not configured properly. The installation procedure should be reviewed. It may be necessary to uninstall and reinstall the Multi-ICE server software.

Available System Resources 3

For custom development it is important to work within the constraints of the system. The available resources for the S2SDK board are detailed in this chapter. The GSP2e contains 1 Mbit of internal EDO DRAM to eliminate the need for external RAM. It is also designed to allow for 2Kb of internal RAM to be battery-backed during a power off period along with the Real-Time Clock. For custom boards you can increase the available memory size by changing the hardware. For information on changing the clock speeds and the available throughput, see Chapter 13, "GPIO Lines, Throughput and Wait States." Memory usage can be determined by examining the `ss2_sdk.map` file generated during the build process.

ROM/RAM Requirements

The following is taken from the ADS build:

- ROM: 267596 bytes used out of 524,288 bytes (512 Kb)
- RAM: 98476 bytes used out of 131,072 bytes (128 Kb)

Stack Requirements

It is important to realize that there is no stack protection in the software. The current stack requirement has been estimated at 4,188 bytes.

Adding Elements to Battery Backed SRAM

There is 2KB of memory in the GSP2e that can be backed up by the external battery. About half of the space is used by the SiRF application code. The following is a description of how to add elements to the battery-backed memory structure.

The process can be broken down into three basic steps. These steps are:

1. Verify that there is enough room left in the battery-backed memory space.
2. Add element to the battery backed structure.
3. Add additional code to the battery-backed memory access routine to handle the new user data.

Each of these steps are explained in more detail in the following sections.

Check Space Left in Battery-Backed Memory

The first step before implementing any user information is to verify that enough unused space exists. The proper information can be gleaned from the `*.map` file that can be generated using the ARM toolchain. For ADS, the following output is generated in the listing file:

```
Execution Region SRAM_DATA (Base: 0x60000000, Size: 0x000007ec,
Max: 0xffffffff, ABSOLUTE)
Base Addr    Size         Type   Attr  Idx  E Section Name
Object
0x60000000   0x000007ec Zero   RW 208 .bss sram.o
```

The size of the existing SRAM data in this example is 0x07ec. Since the available space is 0x1000, the remaining space is 2 bytes.

Add User Element to Battery-Backed Memory Structure

After verifying that there is enough space left in the battery-backed memory, add the user element to the “tSRAMUI” structure in “UI_INC.H.” It is important to add user elements to the end of the current structure. Two examples of new SRAM data are:

```
INT iUserData;
```

```
USER_DATA sUserData; /* where USER_DATA is supplied as well */
```

Verify that you have received the SRAM.C file in source code. Check your SDK directory, if you do not have this file, contact SiRF to get some later code.

Add New Code to SRAM Access Functions

Access to the battery-backed memory information for the purpose of changing it is handled by the “UI_SetUiSram()” function in “UI_SRAM.C”. It is important to use this controlled access for your new user variable because verifies that the CRC is calculated correctly when you are done. If the CRC is not calculated correctly then the SRAM is cleared by software after the next power-up. To add code to correctly access your variable see the following subsections.

Adding an Enumerated Type for the User Variable (Data)

There is an enumerated type which is used to identify each element in battery-backed memory so that the values can be modified. Add another value to the end of the “UI_SRAM_ID” enumerated type in “UI_IF.H.” Use a value such as “ID_USERSRAM.”

Adding a Case Statement in UI_SetUiSram() Function

The final step is to add a case statement to the UI_SetUiSram() function to handle the new element in battery-backed memory. The intent is to call the function with the specific enumerated type value for the variable of interest and to pass in a pointer to some data and/or an integer value. Note that you can use either the (VOID *) pointer argument or the UINT16 value to reference the new data for your variable. In any event, the case statement that add must be as follows:

```
case ID_USERSRAM:  /* new user value from SRAM_ID type */
if (/* validate user data somehow */ )
{
      /* For simple UINT16 value, use iVal argument */
      /* SRAM.UI.iUserData = iVal; */
      /* OR !!!!! */
     /* for structure elements, use (VOID *) indata for reference
*/
      memcpy((VOID *)SRAM.UI.sUserData, indata,
      sizeof(USER_DATA));
}
else
      valid = FALSE;
break;
```

To change the value of your user data, execute the “UI_SetUiSram()” function call as follows (in this case using “inData” as a pointer to the new information):

```
USER_DATA TempUserData;
/* Fill TempUserData with latest information */
UI_SetUiSram(ID_USERSRAM, (void *)&TempUserData, 0);
```

Software Build Process 4

The following chapter provides information about creating a software build using the provided source and object code, and the ARM development environment. Information about the different software variants and compile switches is also provided.

Software Build Process and Variants

Two different variants of the SiRFstarIIe SDK code have been provided to support the two current development environments:

- ARM Developer Suite (ADS)

At the time of printing this manual, the SDK code was built using the ADS version 1.2. In-house debugging was accomplished using the ARM Multi-ICE JTAG device. The SiRFstarIIe board comes equipped with a built-in 20-pin connector for the Multi-ICE to cut-down on development time.

Note – When connecting the Multi-ICE, the red wire on the ribbon cable must be on the side of the connector that is closest to J33 (S2SDK).

The SDK code directory structure as provided on the SDK CD is shown in Figure 4-1.

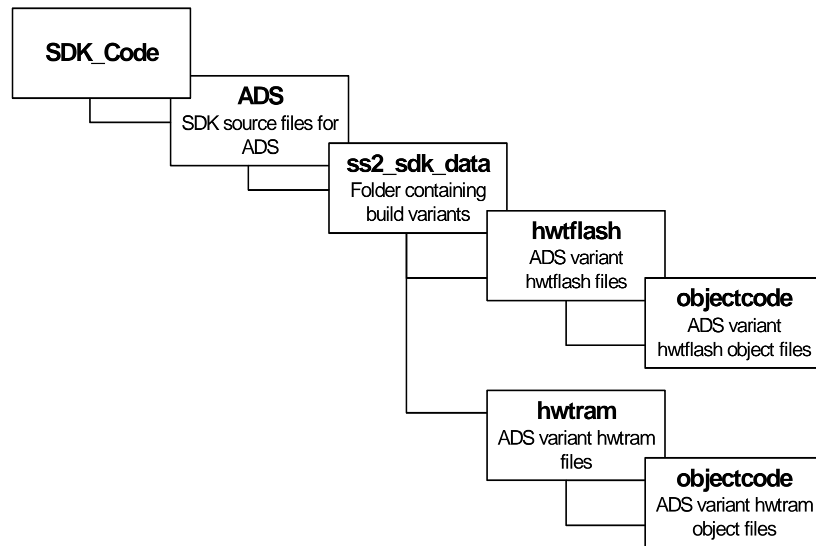

Figure 4-1 The SDK Code Directory Structure

Creating an SDK Build Using ADS

This section describes the process of creating an SDK build using the ARM ADS development environment.

1. In the ADS directory, open the ADS IDE file called `ss2_sdk.mcp`.

   Initially an `*.mcp` file may not be associated with the Code Warrior IDE executable called `IDE.exe` which is typically in the directory `C:\Program Files\ARM\ADSvx_x\BIN`. The ADS uses the Code Warrior IDE as a project management tool for Windows. This tool automates the routine operations of managing source files and building your software development projects. If the `*.mcp` file is already associated, then the Code Warrior IDE project file opens as shown in Figure 4-2 (see “Background About the Two ADS Build Variants” on page 4-4) otherwise, you must associate this file extension to the `IDE.exe`.

2. Select the variant you want to build.

   For our example, the `HwtFlash` variant is selected. Figure 4-2 indicates that the `*.s`, `*.c` and `*.o` files have a red check mark. This indicates that these files must be assembled or compiled and/or linked.

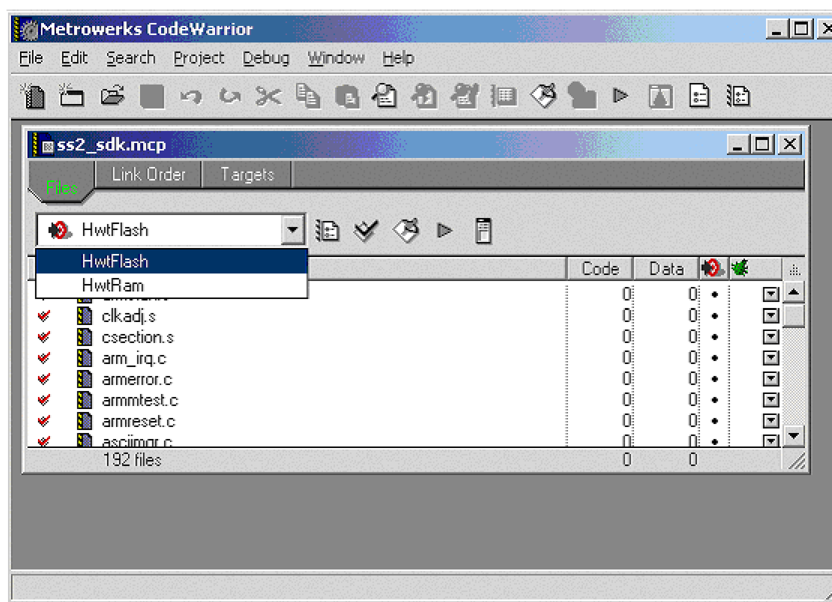

Figure 4-2 Metrowerks CodeWarrier Start-Up Window used in the ADS Environment

To rebuild the entire project, verify that all SDK source and object files in this list have a check mark (i.e., select all files and double-click on the white space underneath the check mark column). Select Project | Make as shown in Figure 4-3 to build a HwtFlash variant.

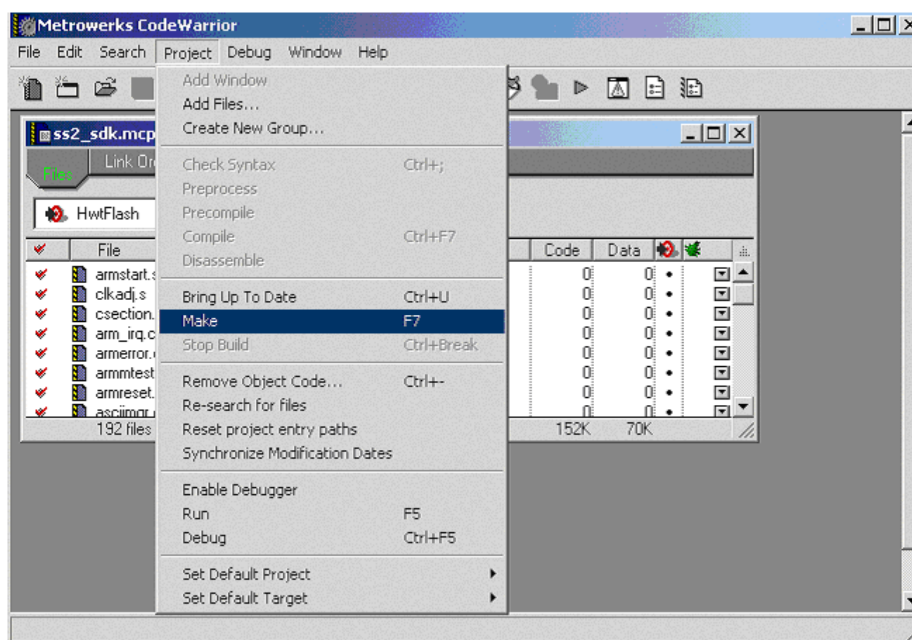

Figure 4-3 Window Displaying How to Make a HwtFlash Variant

During the Make process, a pop-up window called `Building ss2_sdk.mcp` indicates which file is currently being assembled or compiled and linked into the HwtFlash s-record file. If the build is not successful, an Errors & Warnings window appears and terminates the build process. If the build is successful, the variant is created with the Errors & Warnings window indicating no errors and no warnings. A HwtFlash s-record file called `ss2_sdk.s` and an `*.axf` file called `ss2_sdk.axf` are created in the SDK ss2_sdk_data\hwtflash subdirectory.

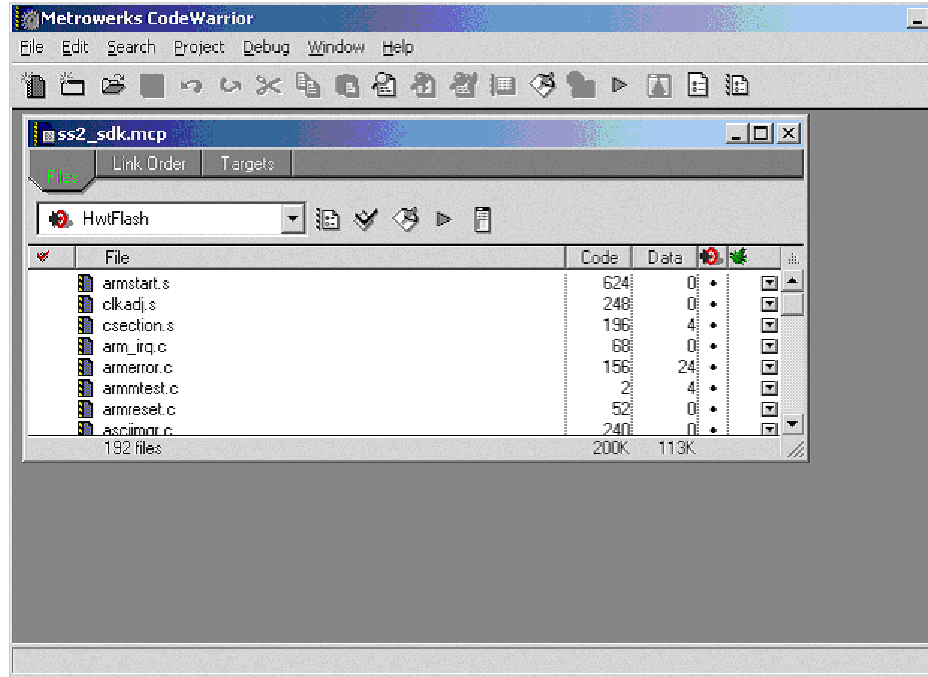

Figure 4-4 ADS Errors and Warnings Window

For a HwtRam variant, repeat the above procedure after selecting the HwtRam variant. If the build is successful, the variant is created without any indication that the build is successful. A `HwtRam *.axf` file called `ss2_sdk.axf` is created in the SDK `ss2_sdk_data\hwtram` subdirectory.

Background About the Two ADS Build Variants

There are two variants: `HwtFlash` and `HwtRam`. `HwtFlash` is used to create a build to download to flash. `HwtRam` is code used to download to RAM for use in conjunction with the Multi-ICE. They are both optimized for time (i.e., speed). For each variant, the object files are located in their respective folder: `ss2_sdk_data\hwtflash\objectcode` or `ss2_sdk_data\hwtram\objectcode`. Each variant creates an `ss2_sdk.axf` file (e.g., in `ss2_sdk_data\hwtflash`), but `HwtFlash`, in addition, generates a Motorola 32-bit Hex s-record from its `ss2_sdk.axf`. This s-record file, `ss2_sdk.s`, is the file used to download to flash.

Note – The s-record file can be downloaded via `SiRFflash` without the assistance of `FIXSREC.EXE`.

Basic Compile Switches

Preprocessor definitions can be set in the ADS environment. Most of the current preprocessor definitions are generated by the ARM toolchain build process.

The preprocessor definitions that currently exist are either:

- User defined and changeable
- Essential preprocessor definitions that are not to be changed by the user
- Preprocessor definitions that are not used in the code and are not supported by SiRF or intended for use

The following tables lists each of the preprocessor definitions.

Table 4-1 User defined and changeable preprocessor options.

Define	Description	Comment
TASK_PERIOD=0	Enables the user tasks if the value is non-zero. The value is the period of the user tasks in msec.	The RTC has a resolution of approximately 8 msec.
PPS_OFF	If defined, disables the PPS output.	Not defined by default.
NMEA	If defined, sets the protocol of serial port A to NMEA.	
USER1	if defined, sets the protocol of serial port A to USER1.	
CACHE	If defined, allows use of the cache. If not defined, cache cannot be used.	Defined by default.
SDKTEST_USERTASK	In conjunction with the TASK_PERIOD value, can be used to implement the user task test code.	Intended for running tests to understand user tasks only. Not defined by default.

Table 4-2 Essential preprocessor options that are not to be changed.

Define	Description	Comment
ARM	Defines the CPU platform.	Must always be defined.
ARM_ADS	Identifies that the compiler is ADS.	Must always be defined in the ADS project.
GPS_SERIAL	Enables serial communications.	Must always be defined in the project if serial communications is required.
GSP2	Defines that the GSP2e is the target.	Must always be defined.
HW_TRACK	Defines HW_TRACKER as active.	Must always be defined.
RELEASE	Defines whether the code is a release version.	Must always be defined.
REMOVE_SP	If defined, removes the SiRF Binary protocol.	Not defined by default. It is not recommended to define this.
BEACON	Define if beacon is to be used. If not defined, removes beacon use ability.	Must always be defined.
WAAS	Defined if WAAS is to be used. If not define, removes WAAS use ability.	Must always be defined. HALF_MSEC must be defined with WAAS.
HALF_MSEC	Defined if WAAS is to be used. Enables a 0.5 msec interrupt.	Must always be defined. Must be used in conjunction with WAAS.
DECODE_IONO	New message decoding scheme.	Must always be defined.

Table 4-3 Preprocessor options that are not supported and are not to be used.

Define	Description	Comment
NOPRINTF	Removes all debug messages from the stream.	Not defined by default.
ES_LOW	Enhanced sensitivity software option.	Not defined by default. Future development placeholder.
HOST_OS	Determines whether another OS is running apart from the SiRF OS.	Not defined by default. Future development placeholder.
SDK_CSI	Using CSI beacon module.	Not defined by default. True if CSI beacon is used.

With the ADS flashHwt variant selected, select Edit | HwtFlash Settings… the following window is displayed as shown in Figure 4-5. From this window select the Target Settings Panel | Language Settings | Thumb C Compiler | Preprocessor tab.

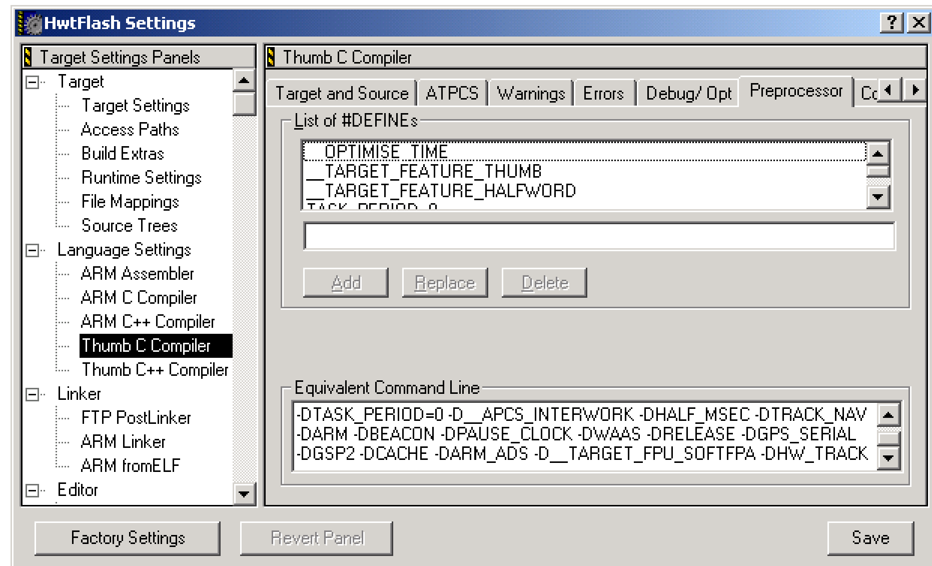

Figure 4-5 Setting ADS Compiler Preprocessor #DEFINEs

Flash Programming 5

The software tool provided with the SiRFstarII SDK for flash programming is SiRFflash. SiRFflash is a fast in-circuit flash memory programming utility for target systems based on SiRF Technology's GSP2 family of chips. It needs a serial line to communicate with the target system, and supports a variety of flash chips from leading manufacturers and multiple file formats. You can use it to program and read flash memory on your GSP2 family based target.

SiRFflash downloads builds for FLASH (either S-record or binary images), while the `hwtram` build can be downloaded using the Multi-ICE. Figure 5-1 shows what a SiRFflash looks like when it is executed.

Downloading Software using SiRFflash

To download new software to your SiRF-enabled GPS platform:

1. Verify that the external data bus width is properly selected on your target platform.

   On the S2SDK board, the external bus is 16-bit wide if jumper J23 is in position 1-2 and jumper J25 in position 2-3. The external bus is 32-bit wide if jumper J23 is in position 2-3 and jumper J25 in position 1-2.

   On the SiRF Technology Evaluation Kit box, the external bus is 16-bit wide.

2. Change the target platform operating mode to internal boot mode and power cycle the receiver.

   On the S2SDK board, the processor boots in internal boot mode if switch S3 is in the INT position.

   On the SiRFstarIIe Evaluation Receiver the processor boots in internal boot mode if the switch BOOT/DATA is in the BOOT position.

3. Connect serial port A of the target platform to one of the COM ports available on your PC.

   On the S2SDK board, the port A serial connector is marked as P1.

   On the SiRFstarIIe Evaluation Receiver the port A serial connector is marked as Com A.

4. Start the SiRFflash software by double-clicking on the `SiRFflash.exe` file or the shortcut if one has been created.

   The SiRFflash software launches as shown in Figure 5-1.

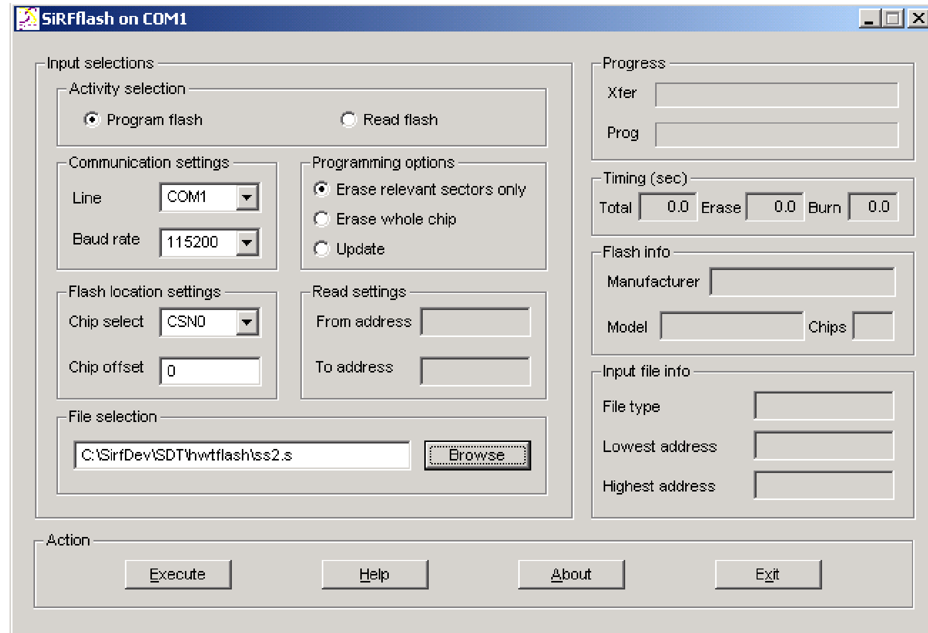

Figure 5-1 The SiRFflash Software

5. Select the *Program flash* radio button in the *Activity selection* box.

6. Select the *Line* and *Baud rate* in the *Communication settings* box.

   It is recommended that you use 115200 Baud to minimize the download time if the target platform is capable of supporting it. At 115200 Baud, SiRFflash can program more than 10 KB of data per second.

7. Select the *Chip select* and *Chip offset* in the *Flash location settings* box.

   These two parameters describe the location of the flash chip (or chips in 32-bit external bus case) that are accessed for programming or reading.

   The chip select parameter describes which chip select signal (CSNx) on the GSP2 processor is used to access the flash during programming/reading. If you select CSN0, it corresponds to address 0x40000000, CSN1 to address 0x41000000, CSN2 to address 0x42000000, etc. If the system is rebooted in external boot mode, flash programmed using CSN0 is visible at 0x0 and 0x40000000, CSN1 at 0x1000000 and 0x41000000, etc.

   Simpler systems have flash at offset 0 in the CSN0 area. In that case, the default value for Chip offset (CSN0) must be used. In more complex systems there might be multiple flash chips within certain CSNx areas and in that case a non-zero offset must be selected if access (programming or reading) to those chips is required.

Additional address space information is in the *System Development Kit User's Guide Part 2 - GSP2e Chip.*

8. Select the input file by typing the file name into the *File selection* field or use the *Browse* button to select the required file.

   The SiRFflash program supports three different input file formats:
   - Motorola S record (typical extension ".s")
   - Intel Hex (typical extension ".hex")
   - Binary (typical extension ".bin")

   The SiRFflash program interprets the file contents based on the content itself, not the file extension. If the first character in the file is 'S', it attempts to interpret the file contents as Motorola S records, if it is a ':', it attempts to interpret Intel Hex records; otherwise it considers the file to be Binary.

9. Select *Erase relevant sectors only*, *Erase chip*, or the *Update* radio button in the *Programming options* box.

   Programming flash memory with SiRFflash is a two step process. In the first step locations that must be written are erased; in the second step those locations are burned (written). The following table provides information about each programming option.

Selection	**Description**
Erase relevant sectors only	The minimum number of sectors are erased which results in the fastest programming time.
Erase whole chip	The whole flash chip is erased which ensures that all flash bytes that programming does not request have the value 0xff.
Update	The previous contents of locations not specified in the input file are preserved.

10. Click on the `Execute` button to start the flash programming process.

    When flash is programmed, two activities are performed in parallel - data transfer from the PC to the target and target activities related to flash programming (sector erase and burning data). The *Xfer* progress bar displays transfer progress and the *Prog* bar displays the programming progress.

    When the download has been completed successfully, SiRFflash displays a window that shows the download terminated without errors. This download process for a standard SSII build usually takes approximately 30 seconds to complete.

11. Set the target platform back to the data operating mode.

    Cycle power to the target platform after switching S3 back to its original position on the S2SDK, or switching the boot switch back to data on the Evaluation Receiver.

Reading Flash Memory

To read the flash memory of your SiRF enabled GPS platform:

1. Follow the steps 1 through to 4 as detailed in “Downloading Software using SiRFflash” on page 5-1.
2. Select the *Read flash* radio button in the *Activity selection* box.
3. Select the *Line* and *Baud rate* in the *Communication settings* box
4. Select the *Chip select* and *Chip offset* in the *Flash location settings* box.

   These two parameters describe the location of the flash chip (or chips in 32-bit external bus case) that are accessed for reading.
5. Select the output file by typing the file name into the *File selection* field or use the *Browse* button to select the required file.

   SiRFflash stores the result of flash reading in a plain Binary file. The value of the first byte in that file is the contents of the memory location whose address is calculated by adding the address corresponding to the Chip select chosen and the Chip offset.
6. Specify the *From address* and *To address* in the *Read settings* box.

Read Setting	**Description**
From address	Represents the offset (from the chip base specified in the flash location box) of the first location that is saved in the output file.
To address	Represents the offset of the last location that is saved in the output file.

7. Click on the `Execute` button to start the flash reading process.

   Once the flash has been read, information is displayed in the *Flash info* box. This includes the flash manufacturer, the flash model, the number of chips detected at the address specified.

Supporting Different Flash Types

SiRFflash has the ability to program a wide range of different flash types. Flash types directly supported are listed in the file called `CHIPLIST.TXT`. This file is located in the SiRFflash installation directory.

Each flash type is described by a line that contains the following information:

- Device name
- Manufacturer name
- Device code (ID)
- Manufacturer code (ID)
- Capacity and sector map
- Driver file name

If the flash type that you are using is not listed in the CHIPLIST.TXT file, it still may be possible that SiRFflash will work with that flash type if you add a line to the CHIPLIST.TXT file.

To support additional flash types:

1. Obtain the device code (ID), manufacturer code (ID), capacity, and sector map information for the intended flash type. This information is typically available from the flash data sheet.
2. Using a text editor, open the CHIPLIST.TXT file.
3. Using the same format that is already in the file, add the device name, manufacturer name, device code, manufacturer code, capacity and sector map information, and the driver file name at the bottom of the existing list.

   The driver file name must be GENERIC.BIN.
4. Follow the steps detailed in “Downloading Software using SiRFflash” on page 5-1.

Development and Debugging 6

For developing and debugging user applications, it is recommended to use the Multi-ICE JTAG debugger from ARM. This device is used by SiRF for the SiRFstarIIe in-house development. This chapter explains the use of a user version string for rudimentary version control and other topics related to debugging user specific code. For debugging purposes, the use of an ARM Multi-ICE, PRINTF style output and LED activation are described.

Adding a User Version String

Version control in any software development environment cannot be overstressed. When modifications are made to the SDK code, some form of version number must be applied to the software variant. This is an invaluable aid for providing performance benchmarks. This section provides a simple method of implementing user version control. The user version is output with the SiRF version number. Software versions are output in the places indicated by the following table.

Protocol	File	Function	When Output
NMEA	`UI_NMEA.C`	OutputStartup	Every receiver reset
SiRF binary	`UI_SIRF.C`	QueueStartup	Every receiver reset
SiRF binary	`UI_SIRF.C`	QueueSwVersion	Response to `MID_PollSWVersion`

Example:

To implement a user version, first add a user version string to `USERINIT.C` of the form:

```
const char userVersion[]= “user v1.0”;
```

Then add an `extern` statement to `user_if.h` to make it globally accessible.

```
extern const char userVersion[];
```

The files mentioned in the table above must be modified to output the version string. Modifications to the files are shown in bold type while file names are shown in bold and underlined. The complete output string cannot be longer than 20 characters.

```
UI_NMEA.C
…
#include "user_if.h"
…

static void OutputStartup (void)
{
   int fl;
   char        szString[50];
   MI_NAV_INIT ni;
   int clk;

   fl = DebugEnabled;
   DebugEnabled = True

   MI_GetSwVersion (szString);
   DebugPrintf ("Version %s", szString);

   DebugPrintf ("Version %s", userVersion);

   MI_GetNavInit (&ni);

    DebugPrintf ("TOW: %-8ld",  ni.timeOfWeek); /* always show init pos/clk*/
    DebugPrintf ("WK:  %-4d",   ni.weekno);
    DebugPrintf ("POS: %-8ld %-8ld %-8ld", ni.posX, ni.posY, ni.posZ);
    DebugPrintf ("CLK: %-8ld",  ni.clkOffset);
    DebugPrintf ("CHNL:%-2d\n", ni.chnlCnt);

#ifndef OFFLINE                   /* show protocol IFF serial debug WAS on */
   DebugPrintf ("Baud rate: %d  System clock: %.3fMHz",
                  SRAM.UI.NMEAbaud, clk/(float)1e6);
#endif

if (UC_GetState(UC_SDKBoard)) /* we are an SDK board */
      DebugPrintf ("HW Type: SDK2");
   else if (UC_GetState(UC_S2ARBoard)) /* we are an S2AR board */
      DebugPrintf ("HW Type: S2AR");
   else
      DebugPrintf ("HW Type: Unknown");
DebugEnabled = fl;
```

```
UI_SIRF.C
…
#include "user_if.h"
…

static WERR QueueSwVersion (void)
{
   UMBufHandle hBuf;
   UINT8    MsgId;
   UINT8    Version[SW_VERSION_LEN];

   hBuf = hComm->allocBuffer (hComm, GetMsgSize (MID_SwVersion));
   if (!hBuf)
   {
      return FAILURE;
   }

   MsgID = MID_SwVersion;
   MI_GetSWVersion ((char *) &Version);
   SEND_ITEM (hBuf, &MsgID);
   SEND_ARRAY(hBuf, Version);
   hComm->send (hComm, hBuf);

   /* output user version string */
   hBuf = hComm->allocBuffer (hComm, GetMsgSize (MID_SWVersion));
   if (!hBuf)
   {
      return FAILURE;
   }
   MsgID = MID_SwVersion;
   memset(Version,'\0',sizeof(Version));
   strcpy((char *)Version, userVersion);
   SEND_ITEM (hBuf, &MsgId);
   SEND_ARRAY(hBuf, Version);
   hComm->send (hComm, hBuf);

   return SUCCESS;
}
```

Multi-ICE Debugging

The following sections provides information about using the ARM Multi-ICE interface unit in the ADS development environments.

ADS and Multi-ICE Debugging

This section describes how to get the ARM Multi-ICE and the ARM Extended Debugger (AXD) in the ADS development environment up and running with the S2SDK development board. The following instructions assume that the Multi-ICE interface unit and software are already installed. For installation instructions for the Multi-ICE interface unit and software, see “Installing ARM Development Tools” on page 2-3.

For the ADS development environment, two build variants are possible - HwtFlash to create a build to download to flash, and HwtRam to download to RAM. Instructions are provided for each of these build variants.

Debugging Flash

When debugging from Flash, the code must be loaded separately onto the Flash before the Multi-ICE is run (to create a Flash build using the HwtFlash variant, see “Software Build Process and Variants” on page 4-1). Once the code is resident in Flash, the Multi-ICE interface unit can be used to debug the flash build.

To debug a flash build using the ARM Multi-ICE interface unit:

1. If not already running, launch the Multi-ICE server application by double clicking on the file `Multi-ICE Server.exe` or the shortcut if one has been created.

2. Select File | Auto-Configure. This displays the window shown in Figure 6-1 after successful connection.

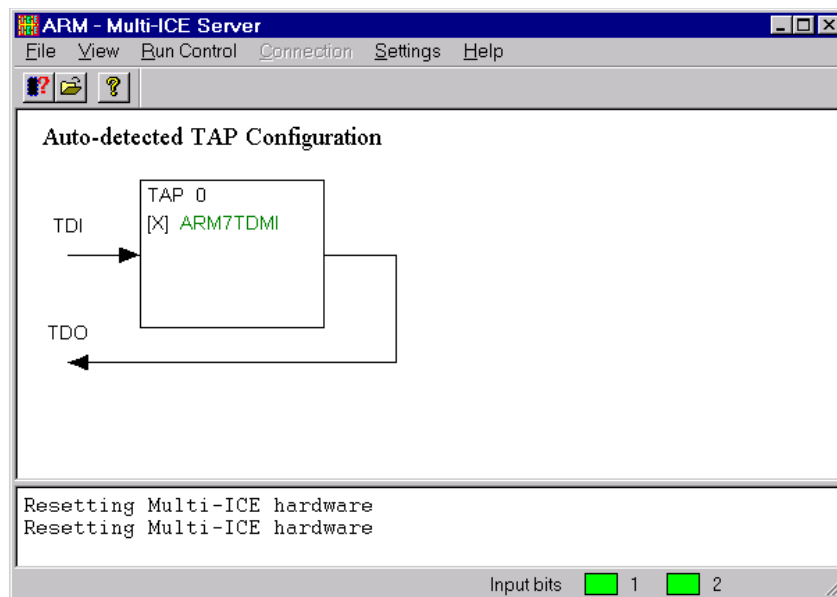

Figure 6-1 Multi-ICE Server Program After Successful Connection

3. Start the ADS Project Manager and run the debugger by selecting Project | Debug.

   If you are running multiple debug windows, close all of the debug windows except one. Verify that the debugger is not running a preloaded image. The debugger is running an image if you can see a “*Running Image*” message at the bottom of the screen.

4. If the debugger is running a preloaded image, select Execute | Stop to stop the image execution and then select File | Unload Current Image to unload the current image.

5. Open the Command Window in the debugger by selecting System Views | Command Line Interface and run the `init16.ads` script by typing the following command line:

```
obey <source path>\ss2_sdk_data\hwtflash\init16.ads
```

   If there are errors, verify that the path to the file is valid. Errors may also be caused by corrupted or badly initialized board registers. Cycling the power of the S2SDK development board should correct this problem.

6. Load the image symbols in the debugger by selecting File | Load Debug Symbols. The *Load Debug Symbols* dialog is displayed as shown in Figure 6-2.

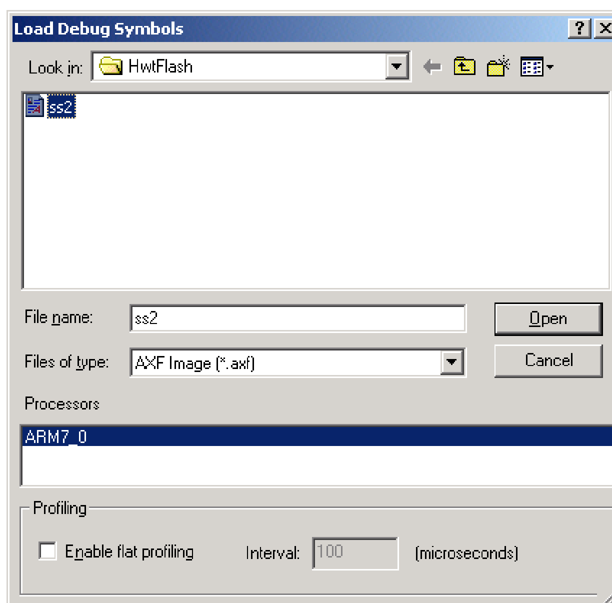

Figure 6-2 The *Load Debug Symbols* dialog.

   For the S2SDK development board, use the `ss2_sdk.axf` symbol file. Once the image is loaded, the files for the project are visible in the *Files* tab of the main debugger window.

After loading the symbol file, you may see the following error message in the *Debug Log* tab:

```
DBE Warning 00064: The image 'C:\pub\...\ss2_sdk.axf' was
compiled with the FPU option 2 (-fpu SoftFPA) that does not match
the debugger mode $target_fpu=1 (SoftVFP).
```

If you see the above warning message, open the Debugger Internals window by selecting System Views | Debugger Internals and changing the *$target_fpu* variable to *2*.

7. Set the debugger source path by selecting Options | Source Path.
8. Before execution, verify that the code is not being executed in flash. This can be done by checking to make sure that there is no serial traffic through the S2SDK development board serial port. If there is no serial traffic, the serial port lights will not be blinking.

   If this is not the case, it may be necessary to open the *Processor Properties* dialog and clear all of the vector catches. This can be done by selecting Options | Configure Processor and clicking the *Clear All* button in the *Vector Catch* group and the clicking *OK*.
9. To execute the code, select Execute | Go.

   The serial port lights on the S2SDK development board should start blinking.
10. The code execution can be stopped anytime by simply selecting Execute | Stop.

    Once stopped, the message window should display the breakpoint in the C file. An example of this is shown in Figure 6-3.

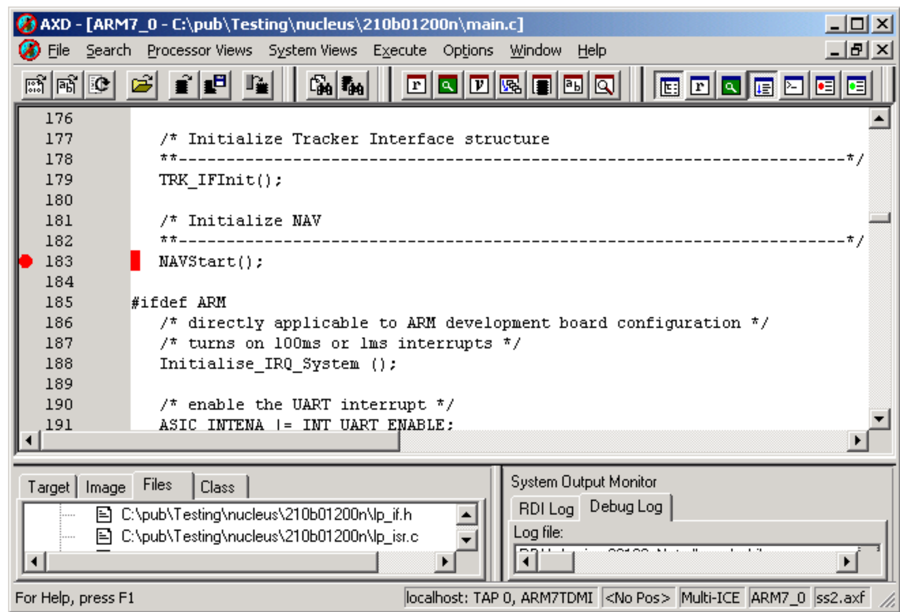

Figure 6-3 Debugger window after a the code has been stopped.

To confirm that the program is running from Flash memory, after the execution is stopped, the execution address from the disassembly screen displays 0x00xxxxxx. This is the Flash memory space. The data space is 0x6xxxxxxx.

Debugging RAM

The following instructions are to debug out of RAM on the S2SDK development board. This is advantageous because it takes less time to load a new build for testing. For debugging out of Flash, see “Debugging Flash” on page 6-4.

To debug a RAM build using the ARM Multi-ICE interface unit:

1. If not already running, start the Multi-ICE server application by double clicking on the file `Multi-ICE Server.exe` or the shortcut if one has been created.
2. Select File | Auto-Configure. This displays the window shown in Figure 6-1 after successful connection.

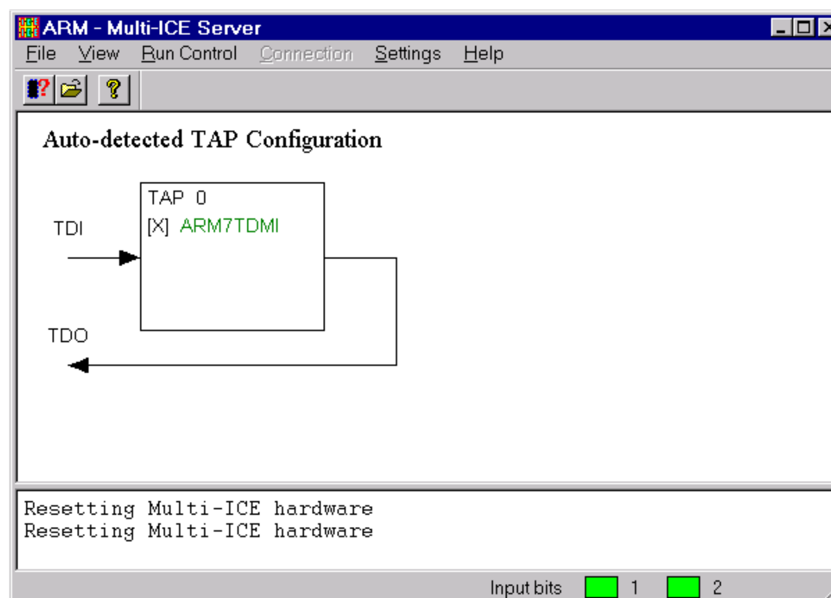

Figure 6-4 Multi-ICE Server Program After Successful Connection

3. Start the ADS Project Manager and run the debugger by selecting Project | Debug.

   If you are running multiple debug windows, close all of the debug windows except one. Verify that the debugger is not running a preloaded image. The debugger is running an image if you can see a “*Running Image*” message at the bottom of the screen.

4. If the debugger is running a preloaded image, select Execute | Stop to stop the image execution and then select File | Unload Current Image to unload the current image.

5. Open the Command Window in the debugger by selecting System Views | Command Line Interface and run the `init16.ads` script by typing the following command line:

```
obey <source path>\ss2_sdk_data\hwtram\init16.ads
```

   If there are errors, verify that the path to the file is valid. Errors may also be caused by corrupted or badly initialized board registers. Cycling the power of the S2SDK development board and restarting the ARM tools should correct this problem.

6. Change the remap register so that it is pointing to RAM instead of Flash. This is done by running the `remap.ads` script by typing the following command line:

```
obey <source path>\ss2_sdk_data\hwtram\remap.ads
```

7. Load the image symbols in the debugger by selecting File | Load Debug Symbols.

   For the S2SDK development board, use the `ss2_sdk.axf` symbol file. Once the image is loaded, the files for the project are visible in the *Files* tab of the main debugger window.

   After loading the symbol file, you may see the following error message in the *Debug Log* tab:

```
DBE Warning 00064: The image 'C:\pub\...\ss2_sdk.axf' was
compiled with the FPU option 2 (-fpu SoftFPA) that does not match
the debugger mode $target_fpu=1 (SoftVFP).
```

   If you see the above warning message, open the Debugger Internals window by selecting System Views | Debugger Internals and changing the *$target_fpu* variable to *2*.

8. Set the debugger source path by selecting Options | Source Path.

9. Before execution, verify that the code is not being executed in flash meaning that the processor is not running. This can be done by checking to make sure that there is no serial traffic through the S2SDK development board serial port. If there is no serial traffic, the serial port lights will not be blinking.

   If this is not the case, it may be necessary to open the *Processor Properties* dialog and clear all of the vector catches. This can be done by selecting Options | Configure Processor and clicking the *Clear All* button in the *Vector Catch* group and the clicking *OK*.

10. Change the PC program counter to 0x0000000. This is done by selecting the *PC Register* field in the *Registers* window and entering 0x0000000.

11. To execute the code, select Execute | Go.

    The serial port lights on the S2SDK development board should start blinking.

12. The code execution can be stopped anytime by simply selecting Execute | Stop.

Once stopped, the message window should display the breakpoint in the C file. An example of this is shown in Figure 6-3.

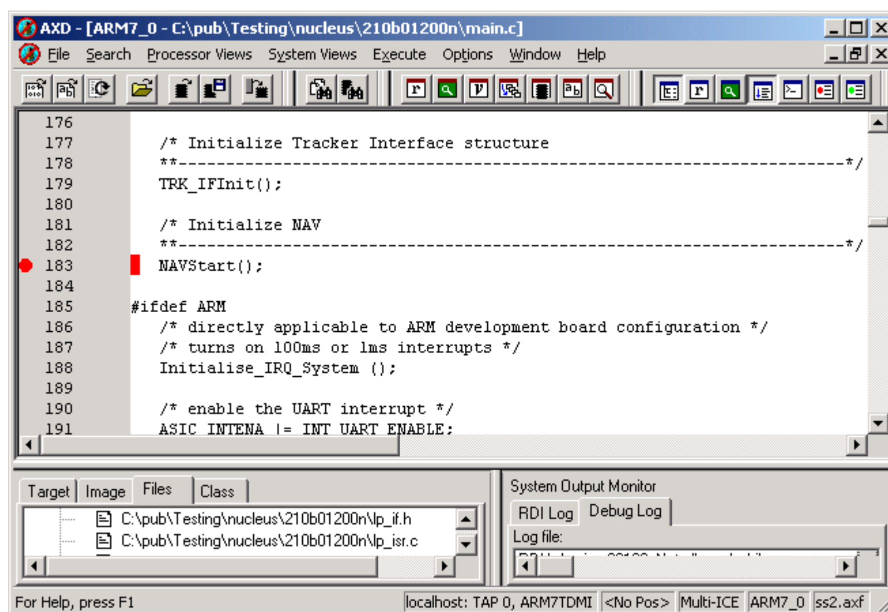

Figure 6-5 Debugger window after a the code has been stopped.

To confirm that the program is running from RAM:

1. Confirm that a remap command has been performed. If this has not been done, it is possible that code is still running from Flash.
2. Open the `main.c` file and add four breakpoints. If the code is running from Flash, only two breakpoints are allowed. RAM builds allow more than two breakpoints to be made during execution.
3. Stop the execution and look at the disassembly screen. The execution should be in the address space range of 0x41000000 to 0x41FFFFFF.

For the ADS development environment, in the “Thumb C Compiler” configuration, the optimization level must be set to “none” and at least the “Enable debug table generation” and “Include preprocessor symbols” must be checked. In the “ARM Assembler” configuration, the “Source Line Debug” and “Keep Symbols” boxes must be checked. In the “ARM Linker” configuration, the “Include debug information” box must be checked with at least “Totals” and “Sizes” also checked and a list file must be named.

PRINTF Debugging

You can print debug information out of one of the serial ports for display on a terminal program or for storing to a file. This is an easy way to visually inspect data or watch for software events. The implementation of a debug print statement depends on the active protocol. You can set one serial port to SiRF binary while the other one is used for custom development. This has two advantages, the SiRF binary debug print output can be used and the GPS performance of the board can be monitored using `SiRFdemo.exe`.

The debug output for SiRF binary is output on the port where the SiRF protocol is currently active. The actual debug message is encapsulated in a SiRF binary transport layer and has a message ID of 255 (0xFF).

Example:

To implement NMEA protocol on Port 1 and SiRF Binary Protocol on Port 2:

```
UI_SRAM.C

void UI_SetUiSram(SRAM_ID id, void *indata, UINT16 ival)
{
   UARTParams *pUParams;
   NMEACFG *nmeaCfg;
   int i;

   BOOL valid=TRUE; /*true till proven false*/

   /*x-fer data from *indata to UI_SRAM struct and recalc crc*/
   switch(id)
   {

      …
      …
      …
      case ID_INITIALIZE:
/* PRINTF("UI_SetUiSram: Initializing to default values"); TOO EARLY FOR ANY PRINTING, PORTS
NOT YET OPEN */

/* First clear the whole structure to zero */
memset (&SRAM.UI, 0, sizeof (SRAM.UI));

            /* set up default PROTOCOL*/
   #if 0
            SRAM.UI.ProtocolA=UI_PROTO_DEFAULT;
            SRAM.UI.ProtocolB=UI_PROTO_RTCM;
   #else
            SRAM.UI.ProtocolA=UI_PROTO_NMEA;
            SRAM.UI.ProtocolB=UI_PROTO_SIRF;
   #endif

                  …
                  …
      break;
      …
      …
   }
   …
}
```

NMEA Debug Output

- Print Debug output using `NmeaDebugPrintf`(), defined in `UI_NMEA.C`. To use this function a call outside of `UI_NMEA.C`, `NmeaDebugPrintf()` must be declared as an `extern` type in `UI_NMEA.H`. The header file must be included by files that use debug output.

- Debug output is enabled/disabled using the `DebugEnabled` variable. By default, this value is 0 (disabled). To enable debug output as default, set this value to 1 when it is declared.
- The value of `DebugEnabled` can be set at runtime using the `$PSRF105` input command.
- Debug output using this function is preceded by a `$` and ends with `\r\n` but has no checksum.
- This output can be used to determine why a NMEA input command was not accepted.

SiRF Binary Debug Output

- Print debug output using the `umDebugPrintf()` function, defined in `UMANAGER.C`. To use this function outside of this file, verify that `UMANAGER.H` is included at the top of the file of interest.
- To disable debug output at compile time and verify that it cannot be used at runtime, use the `NOPRINTF` compile define. See “Basic Compile Switches” on page 4-5.
- Debug output is enabled/disabled depending on the value of `umSerialDebugFlag`.
- Debug output can be enabled at runtime using `NAVSetSerialDebugFlag`(1) which sets `umSerialDebugFlag` to 1.
- Debug output can be disabled at runtime using `NAVSetSerialDebugFlag`(0) which sets `umSerialDebugFlag` to 0.
- Debug output can be enabled/disabled at runtime using the Navigation Initialization Message from `SiRFdemo.exe` by selecting Action | Initialize Data Source.
- The default value for `umSerialDebugFlag` depends on `SRAM.UI.SIRFMsgCntl`. This is a member of the UI_SRAM structure (defined in UI_INC.H) in battery-backed SRAM. The default value is controlled in `UI_SRAM.C` by the line `SRAM.UI.SIRFMsgCntl = SP_DBGOUT`. This enables debug output.
- To disable debug output as default, set the above line to `SRAM.UI.SIRFMsgCntl = 0x00.`
- When sending log files to SiRF for debugging, always verify that the debug output is enabled.

S2SDK LED Activation

The S2SDK board has five bicolor LEDs, which are connected to 10 GPIO lines of the GSP2e chip. These LEDs may be used by users for status display and debug. The GSP2e GPIO signals are 36 to 45 shown in the following table.

GPIO Line	LED	Ports and Bit
36	D4 Green	GPIO_PortDir1 and GPIO_PortVal1 Bit 0
37	D8 Red	GPIO_PortDir1 and GPIO_PortVal1 Bit 1
38	D8 Green	GPIO_PortDir1 and GPIO_PortVal1 Bit 2
39	D7 Red	GPIO_PortDir2 and GPIO_PortVal2 Bit 0
40	D7 Green	GPIO_PortDir2 and GPIO_PortVal2 Bit 1
41	D6 Red	GPIO_PortDir2 and GPIO_PortVal2 Bit 2
42	D6 Green	GPIO_PortDir2 and GPIO_PortVal2 Bit 3
43	D5 Red	GPIO_PortDir2 and GPIO_PortVal2 Bit 4
44	D5 Green	GPIO_PortDir2 and GPIO_PortVal2 Bit 5
45	D4 Red	GPIO_PortDir2 and GPIO_PortVal2 Bit 6

To light a LED, the Direction and Value of the GPIO pin must be set to 1. To clear the LED, the Direction must be set to 1 and the value set to 0. See the *SiRFstarIIe System Development Kit User's Guide Part 2 - GSP2e Chip* for GPIO register memory locations. The memory locations are included here for convenience.

GPIO_PortDir1 is at 0x80010138

GPIO_PortDir2 is at 0x8001013C

GPIO_PortVal1 is at 0x80010124

GPIO_PortVal2 is at 0x80010128

Example:

To toggle the LED at D5 (red or off) execute the following lines. Do not alter any bits in the register other than the one(s) you are interested in.

```
ASIC_GPIO_PORTDIR2 |= 0x10;
ASIC_GPIO_PORTVAL2 ^= 0x10;
```

GPS Performance Testing

To examine the GPS performance of the receiver, you must have SiRF Binary Protocol enabled. This enables you to use the `SiRFdemo.exe` utility provided with the Evaluation Kit and the SDK as well as the GPS analysis program `Sirfsig.exe`. More importantly, if you have problems with the GPS performance, you can log GPS information in SiRF Binary format along with the software version number and send this to SiRF along with a test report. This information can then be examined by our GPS specialists.

For more information on SiRFdemo.exe, Sirfsig.exe and other utility programs, see *SiRFstarIIe Evaluation Kit User's Guide*.

Using PROCOMM to Send NMEA Messages

Two Procomm Meta-key files are included on the SDK CD, they are NMEA100.KEY and DGPS.KEY. These files contain miscellaneous NMEA input commands and provide an example of the NMEA input messages in action. Using these files and Procomm simplifies sending the input commands detailed in the NMEA input command section of *SiRFstarIIe Evaluation Kit User's Guide*.

These files are for use with Procomm (versions 2 or 3.0, but may work for older versions). Select Metakeys | Tools, and then load this file. For Procomm Plus version 2.1, select Tools | Mete Key Editor, then load this file. This enables metakey buttons (on the bottom of the screen) to send the NMEA message out the comm port as defined in the metakey editor. The following list shows several of the many buttons possible.

Button	**Description**
Q0_GGA	Queries the GGA message.
Debug_on	Switches debug output on.
Debug_off	Switches debug output off.
Bad checksum	Sends a message with an intentionally bad checksum that should be rejected by the module. If debug is on, you are able to see the message with the bad checksum rejected by the module. If debug is off, the message with a bad checksum is ignored.

Note that keys like lla_warm were setup to init the module sending the LLA_INIT message, and contain parameters which are for a specific time/place and must be modified to correctly initialize your module.

Use the metakey editor to see the contents of each button. Note that the [ALT] key to the left of the metakeys in Procomm switches among four different sets of metakeys. Also notice that each string ends in /r/n. /r/n looks like ^M^J when editing the strings in ASCII mode.

Warning – If you send a sentence without this termination, it may appear that the message was ignored, but in fact the module is waiting for message termination before passing the message up to the NMEA input handlers in UI_NMEA.C.

NMEA Checksum Utility

Two optional files are included on the SDK source disk: DOS executable CKSUM.EXE, and the corresponding source file CK.C. The purpose of CKSUM.EXE is to read in a file containing NMEA sentences and calculate the correct NMEA checksum. This can be used to verify operation of NMEA output sentences, or to generate a checksum for an NMEA input message.

Example:

To create a text file containing an NMEA input sentence, for example, an input NMEA query message and determine the proper checksum.

```
J:\>type query0.txt
$PSRF103,00,01,00,01*xx

J:\>cksum query0.txt
INPUT FILE: query0.txt
inline:$PSRF103,00,01,00,01*xx
cksum:25
```

The correct checksum for this message is 25. You can then use Procomm, or some similar terminal program to send this message. `CK.C` can be compiled using any compiler capable of generating DOS programs, and is simple to modify for your own unique uses.

Uploading Code to SiRFstarIIe without SiRFflash

You can upload code to the SiRFstarIIe with a user-defined utility if the use of `Sirfflash.exe` is not appropriate. This section describes the interaction between `Sirfflash.exe` and the SiRFstarIIe board. There are two stages to uploading new software into Flash. The first is to use the simple internal SiRFstarIIe boot code to load a Flash programming utility into EDO RAM. The second is to use this utility program in RAM to read the new firmware from the same serial port and write it to Flash. The following steps define the loading process. You must become familiar with the format of the Motorola s-record format first.

1. Reset the board in internal boot mode.
2. Wait one second.
3. Use the internal boot code to upload a flash loader/serial interface program to EDO RAM (for example “`dlgsp2.bin`”). The internal boot code is serial receive at 38400 Baud. Send an S0 followed by four bytes representing the data length (MSB first), then the data, then another 4 bytes as 0x00,0x00,0x00,0x00. The last four bytes give the target reset vector (use 0x00000000). The boot loader starts program execution. The boot loader allows for some limited Baud rate changes.
4. Wait one or two seconds.
5. With `dlgsp2.bin` loaded, start sending s-records at 38400. If the first s-record received is a valid start line, the flash is erased. The board outputs a `SA[CR][LF]` response after every valid s-record line. If there is an error at any time, the SiRFstarIIe sends back `SE<error message>[CR][LF]`.
6. Wait a couple seconds and then reset in external boot mode.

Internal Boot Operation

The GSP2e chip contains an internal boot program to load a binary image into RAM for execution. It is executed when jumper 21 is set to the 1-2 position and the power is cycled. The boot loader is burnt into the chip and cannot be changed. It is very simple and implements only serial receive functionality. The boot program assumes a Baud rate of 38400 and may be adjusted by having the first characters received after reset being an ASCII ‘S’ (0x53) followed by the number 0, 1, or 2 for 38400, 57600, 115200 respectively. Following a Baud rate change another ‘S’ followed by a 0 must be sent at the new Baud rate following a delay of 1 ms. The internal boot loader runs off of the GPS clock in the RF section and the Baud rate is not affected by the rate of an external CPU clock.

The first data sent is 4 bytes of the payload file size. The MSB is sent first. This is immediately followed by the payload data. Finally a 4-byte target reset vector is sent (MSB first). The program counter is set to the target reset vector and the downloaded code is executed.

The following is an example of uploading a byte sequence (all at 38400 Baud):

53 00 00 00 00 08 06 00 00 EA FE FF FF EA 00 00 00 00

The following sequence enables start at 38400, and to switch to 57600 Baud after sending the first two characters:

53 01 53 00 00 00 00 08 06 00 00 EA FE FF FF EA 00 00 00 00

This process is performed automatically by `SiRFflash`. Use the internal boot to load your own code to run from RAM. An alternate flash loader can be installed using that scheme if a different load procedure is desired.

Memory BUS and Components 7

Memory

The SDK is delivered with the project file for the ADS toolchain. The project contains all files necessary to build a flash image file loadable into the S2SDK through `sirfflash.exe` or a RAM image file that can be loaded into the RAM on the S2SDK through a Multi-ICE JTAG device. As indicated, both software variants are contained inside the project files. See "Software Build Process and Variants" on page 4-1.

Each toolchain supports two scatter files that define the memory map for each variant. For the SDK toolchain, the RAM.SCT scatter files is used to define the memory map for the HwtRam variant and the FLASHSDT.SCT scatter file is used to define the memory map for the HwtFlash variant. For the ADS toolchain, the RAM.SCT scatter file is used to define the memory map for the HwtRam variant and the FLASHADS.SCT scatter file is used to define the memory map for the HwtFlash.

Scatter Loading Files

Both scatter files define an area for `SRAM.O`. This is an uninitialized section that contains a copy of the battery-backed RAM. There is also an uninitialized section containing `SDRAM.O`. This region contains the unpacked ephemeris data read and unpacked out of the battery-backed RAM area. The various memory types and memory sizes accessible on the S2SDK are described in the *SiRFstarIIe System Development Kit User's Guide Part 3 - S2SDK Board*.

HwtRAM

For the ADS toolchain, a typical memory map of a development board using a Multi-ICE to download to RAM looks like the file `RAM.SCT`:

```
LOAD_REGION 0x00000000
{
   VECTOR_CODE 0x00000000
   {
      armstart.o (Init, +FIRST)
   }
}
NEXT_REGION 0x41000000
{
   CODE 0x41000000
   {
      * (+CODE, +CONST)
   }
   SRAM_DATA 0x60000400
   {
      sram.o (+BSS)
   }
   SDRAM_DATA +0
   {
      sdram.o (+BSS)
   }
   GPS_DATA +0
   {
      autocor.o (+DATA,+BSS)
      coord.o (+DATA,+BSS)
; Commented out due to linker warning.  Uncomment it if global
data is added.
;        corr.o (+DATA,+BSS)
      cstd.o (+DATA,+BSS)
      dd_par.o (+DATA,+BSS)
      decode.o (+DATA,+BSS)
      dgpsrx.o (+DATA,+BSS)
      global.o (+DATA,+BSS)
      intrface.o (+DATA,+BSS)
      kflib.o (+DATA,+BSS)
      lp_*.o (+DATA,+BSS)
      mi_var.o (+DATA,+BSS)
      naprep.o (+DATA,+BSS)
      rxm.o (+DATA,+BSS)
```

```
  startup.o (+DATA,+BSS)
      nav_if.o (+DATA,+BSS)
      navinit.o (+DATA,+BSS)
      ntsc.o (+DATA,+BSS)
      ntsl.o (+DATA,+BSS)
      nl*.o (+DATA,+BSS)
      rm_*.o (+DATA,+BSS)
      tr_*.o (+DATA,+BSS)
      waas.o (+DATA,+BSS)
      waascorr.o (+DATA,+BSS)
   }
   DATA + 0
   {
      * (+DATA,+BSS)
   }
}
```

In this example, the external RAM is being used for the code and constant data space at address 0x41000000. The internal EDO RAM is being mapped using the `remap` register down to address 0x00000000 and is also being used at its normal address range for battery-backed RAM at address 0x60000400. The internal EDO RAM also contains zero initialized data for the SDRAM_DATA, GPS_DATA and DATA sub-regions.

If the user decides to add new non-GPS files, then these filenames must not begin with the following prefixes: lp_, nl, rm_, or tr_. Using a non-GPS filename with these prefixes will cause non-GPS variables to be wrongfully placed into the GPS_DATA sub-region instead of the DATA sub-region. If the user has a new non-GPS related file that does not begin with the following prefixes: lp_, nl, rm_, or tr_, there is no need to update the scatter file because it will be assigned to non-GPS (DATA) data region by default.

For the ADS tool-chain, the memory map for a flash build uses the scatter file FLASHADS.SCT:

```
LOAD_REGION 0x00000000
{
    CODE 0x0
    {
        armstart.o (Init,+FIRST)
        * (+CODE, +CONST)
    }
    SRAM_DATA 0x60000000
    {
        sram.o (+BSS)
    }
    SDRAM_DATA +0
    {
        sdram.o (+BSS)
    }
    FAST_CODE +0
    {
    rt_sdiv.o(.text)
    rt_udiv.o(.text)
    faddsub.o(x$fpl$fadd)
    fmul_mull.o(x$fpl$fmul)
    fdiv.o(x$fpl$fdiv)
    daddsub.o(x$fpl$dadd)
        dmul_mull.o(x$fpl$dmul)
    ddiv.o(x$fpl$ddiv)
        fnorm2.o(x$fpl$fnorm2)
    }
    GPS_DATA +0
    {
        autocor.o (+DATA,+BSS)
        coord.o (+DATA,+BSS)
; Commented out due to linker warning.  Uncomment it if global
data is added.
;        corr.o (+DATA,+BSS)
        cstd.o (+DATA,+BSS)
        dd_par.o (+DATA,+BSS)
        decode.o (+DATA,+BSS)
        dgpsrx.o (+DATA,+BSS)
        global.o (+DATA,+BSS)
```

```
intrface.o (+DATA,+BSS)
        kflib.o (+DATA,+BSS)
        lp_*.o (+DATA,+BSS)
        mi_var.o (+DATA,+BSS)
        naprep.o (+DATA,+BSS)
        rxm.o (+DATA,+BSS)
        startup.o (+DATA,+BSS)
        nav_if.o (+DATA,+BSS)
        navinit.o (+DATA,+BSS)
        ntsc.o (+DATA,+BSS)
        ntsl.o (+DATA,+BSS)
        nl*.o (+DATA,+BSS)
        rm_*.o (+DATA,+BSS)
        tr_*.o (+DATA,+BSS)
        waas.o (+DATA,+BSS)
        waascorr.o (+DATA,+BSS)
    }
    DATA + 0
    {
        * (+DATA,+BSS)
    }
}
```

In this example, the ADS Flash and the internal S RAM starts at address 0x00000000 and 0x60000000 respectively. The area 0x60000000 to 0x600001000 contains the copy of the batter-backed RAM. Unlike the SDT Flash region, the ADS Flash region contains a FAST_CODE sub-region used to store frequently used ARM math library code. This is done to further reduce processing time. The internal EDO RAM also contains zero initialized data for the SDRAM_DATA, GPS_DATA and DATA sub-regions.

If the user decides to add new non-GPS files, then these filenames must not begin with the following prefixes: lp_, nl, rm_, or tr_. Using a non-GPS filename with these prefixes causes non-GPS variables to be wrongfully placed into the GPS_DATA sub-region instead of the DATA sub-region. If the user has a new non-GPS related file that does not begin with the following prefixes: lp_, nl, rm_, or tr_, there is no need to update the scatter file because it will be assigned to non-GPS (DATA) data region by default.

Memory Areas in the GSP2e

The memory map at start-up is dependent on whether external or internal boot mode is used. In the case of internal boot mode, the internal ROM of the GSP2e is located at memory address 0x0000 and the internal boot loader executes. In the case of external boot mode, Flash (CSN0) is located at address 0x0000. In either case, executing the `remap` function shadows the S RAM at 0x60000000 down to 0x0000. This enables debugging from RAM as described in “Debugging Flash” on page 6-4. Table 7-1 and Table 7-2 show the memory map before and after `remap` for the internal and external

boot case. After `remap` the memory map is the same for each. The ARM project manager generates a `ss2_sdk.map` file that contains the breakdown of memory address allocation.

Note – The Bus Interface Unit (BIU) is broken up by Chip select (CS) with the default settings having CS0 at 0x40000000 (FLASH), CS1 at 0x41000000, CS2 at 0x42000000, etc.

Table 7-1 Memory Map for Internal Boot

	Before REMAP	**After REMAP**
Address (31:0)	**Interface Selected**	**Interface Selected**
FFFF FFFF … E000 0000	Internal peripherals	Internal peripherals
DFFF FFFF … C000 0000	ARM test	ARM test
BFFF FFFF … A000 0000	Internal RAM, if cache is disabled else it is an undefined area	Internal RAM, if cache is disabled else it is an undefined area
9FFF FFFF … 8000 0000	Internal peripherals	Internal peripherals
7FFF FFFF … 6000 0000	SRAM	SRAM
5FFF FFFF … 4000 0000	Bus interface unit	Bus interface unit
3FFF FFFF … 2000 0000	On-chip boot ROM	On-chip boot ROM
1FFF FFFF … 0000 0000	On-chip boot ROM	SRAM Internal

Table 7-2 Memory Map for External Boot.

	Before REMAP	**After REMAP**
Address (31:0)	**Interface Selected**	**Interface Selected**
FFFF FFFF … E000 0000	Internal peripherals	Internal peripherals
DFFF FFFF … C000 0000	ARM test	ARM test
BFFF FFFF … A000 0000	Internal RAM, if cache is disabled else it is an undefined area	Internal RAM, if cache is disabled else it is an undefined area
9FFF FFFF … 8000 0000	Internal peripherals	Internal peripherals
7FFF FFFF … 6000 0000	SRAM	EDO DRAM
5FFF FFFF … 4000 0000	Bus interface unit	Bus interface unit
3FFF FFFF … 2000 0000	On-chip boot ROM	On-chip boot ROM
1FFF FFFF … 0000 0000	Bus interface unit	EDO RAM

Memory Map

Remap Function

The Remap function remaps the SRAM module address space, and it becomes accessible at address 0x0000_0000 in addition to address 0x6000_0000 (default location). This facilitates execution of exception handlers from SRAM. The ARM exception branch vectors are fixed to the bottom of the address space:

0x00000000: Reset
0x00000004: Undefined instruction
0x00000008: Software interrupt
0x0000000C: Abort (prefetch)
0x00000010: Abort (Data)
0x00000014: Reserved
0x00000018: IRQ
0x0000001C: FIQ

If GSP2e is configured at power-up into Internal Boot Mode, or if executable code is loaded from external boot ROM into SRAM, the REMAP function enables GSP2e to execute exceptions in SRAM.

Remap Procedures

- Internal Boot Mode: The remap procedure is embedded into the on-chip boot code.
- External Boot Mode: After reset, boot code in external memory at CSN0 is executed starting at address 0x0. After application code has been loaded into the EDO module, a write to address 0x800A007C (the REMAP register), executes the REMAP function. Then the 128 KByte EDO module becomes accessible at addresses 0x0000000 and 0x6000000. CPU exceptions generated after the REMAP function is executed must be handled by the vectors loaded into the EDO.

Memory Map Configuration

Memory Map for the External Boot ROM Configuration

Table 7-3 Memory Map Without Remap

Address(31:0)	Selected Interface
FFFF FFFF … E000 0000	Internal peripherals
DFFF FFFF … C000 0000	ARM Test
BFFF FFFF … A000 0000	Internal RAM (only if the cache is disabled, otherwise it is an undefined area)
9FFF FFFF … 8000 0000	Internal peripherals
7FFF FFFF … 6000 0000	SRAM[1]
5FFF FFFF … 4000 0000	Bus interface unit
3FFF FFFF … 2000 0000	On-chip boot ROM
1FFF FFFF … 0000 0000	Bus interface unit[2]

1. Internal SRAM.

2. External memory.

Table 7-4 Memory Map After Remap

Address(31:0)	Selected Interface
FFFF FFFF … E000 0000	Internal peripherals
DFFF FFFF … C000 0000	ARM test
BFFF FFFF … A000 0000	Internal RAM (only if the cache is disabled, otherwise it is an undefined area)
9FFF FFFF … 8000 0000	Internal peripherals
7FFF FFFF … 6000 0000	EDO DRAM
5FFF FFFF … 4000 0000	Bus interface unit
3FFF FFFF … 2000 0000	On-chip boot ROM
1FFF FFFF … 0000 0000	SRAM

Memory Map for the Internal Boot ROM Configuration

Table 7-5 Memory Map Without Remap

Address(31:0)	Selected Interface
FFFF FFFF … E000 0000	Internal peripherals
DFFF FFFF … C000 0000	ARM test
BFFF FFFF … A000 0000	Internal RAM (only if the cache is disabled, otherwise its an undefined area)
9FFF FFFF … 8000 0000	Internal peripherals
7FFF FFFF … 6000 0000	EDO DRAM
5FFF FFFF … 4000 0000	Bus interface unit
3FFF FFFF … 2000 0000	On-chip boot ROM
1FFF FFFF … 0000 0000	On-chip boot ROM

Table 7-6 Memory Map After Remap

Address(31:0)	Selected Interface
FFFF FFFF … E000 0000	Internal peripherals
DFFF FFFF … C000 0000	ARM test
BFFF FFFF … A000 0000	Internal RAM (only if the cache is disabled, otherwise its an undefined area)
9FFF FFFF … 8000 0000	Internal peripherals
7FFF FFFF … 6000 0000	EDO DRAM
5FFF FFFF … 4000 0000	Bus interface unit
3FFF FFFF … 2000 0000	EDO DRAM
1FFF FFFF … 0000 0000	SRAM

Input/Output Messages 8

The following chapter provides information about input and output messaging that is supported by the SiRFstarIIe architecture:

- Changing the default message settings
- Adding new input and output messages

Changing Default Message Settings

The following section provides information about changing the default settings for the SiRF Binary Protocol, NMEA, and the USER1 protocol.

Default Output Protocol

There are three different I/O message protocols readily available on the SiRFstarIIe. These are SiRF Binary, NMEA (ASCII), and USER1. The default protocol (`UI_PROTO_DEFAULT`) on Port 1 is defined during compile time. This DEFINE (#define) is set based off the NMEA and USER1 tcc preprocesser definitions. `UI_IF.H` contains the DEFINE logic that determines the default protocol for Port 1, so NMEA and USER1 tcc preprocessor definitions cannot be defined at the same time. Only one can be selected. In SiRF's standard build, SiRF Binary is the default protocol for Port 1 and contains many SiRF defined messages for controlling the receiver and examining GPS performance. The default protocol for Port 2 is RTCM, which is an input-only protocol for accepting GPS pseudo range corrections. For more information on Radio Technical Commission for Marine Services (RTCM) GPS messages, see the *RTCM Recommended Standard for Differential Navstar GPS Service*, Version 2.1. Currently, the SiRFstarIIe supports RTCM messages 1, 2, 3 and 9 as serial input.

You can change the communication protocol on both ports, with the limitation that any given protocol can only be active on one port. To enable NMEA as the default output on Port 1, NMEA must be specified as a preprocessor definition as described in "Basic Compile Switches" on page 4-5. The USER1 protocol can also be specified in the same way, but this is only provided as a shell for potential user applications and must be completed by the user first as described in Chapter 12, "Adding a New User Protocol." To change the default outputs for both ports requires some code modification (see the example in "PRINTF Debugging" on page 6-9).

Default Baud Rate

In a typical application, Port 1 is used for communicating between the receiver and a console type device such as a laptop PC. Messages may be input to the receiver and data logged to the console. Port 2 is normally used to receive RTCM differential corrections. The default Baud rates for the SiRFstarIIe are set to the values given in the following table.

Protocol	Default Baud Rate
SiRF Binary	38400
NMEA	4800
RTCM differential	9600
USER1	9600

SiRF Binary Baud Rate

The default Baud rate for SiRF Binary is 38400. To change the Baud rate, the value must be changed in `UI_IF.H`. The parity, number of bits, and stop bit are contained in this file as well.

Example:

To change the default Baud rate from 4800 to 9600, change the following line:

```
#define SIRF_BAUD_RATE 4800
```

to:

```
#define SIRF_BAUD_RATE 9600
```

Note – The CSi BRFM1 has a maximum baud rate of 9600.

NMEA Baud Rate

The default Baud rate for the NMEA protocol is 4800. To change the Baud rate, the value must be changed in `UI_IF.H`. The Baud rate must be able to support the amount of data that is requested and the GSV output may be more than one message.

Example:

To change the default Baud rate from 4800 to 9600, change the following line:

```
#define NMEA_BAUD_RATE 4800
```

to:

```
#define NMEA_BAUD_RATE 9600
```

RTCM Baud Rate

The default Baud rate for the RTCM differential port is 9600. To change the Baud rate, the value must be changed in `UI_RTCM.H`.

Example:

To change the Baud rate from 9600 to 19200, change the following line:

```
#define RTCM_BAUD_RATE 9600
```

to:

```
#define RTCM_BAUD_RATE 19200
```

Default Message Output Rates

You can set the NMEA and SiRF default output rates for individual messages, although the implementation for each is quite different. The following sections describe both procedures. Currently, the basic unit for counting output periods is the navigation cycle (defined as `MI_PERIOD_CYCLES`). This is not seconds, although sometimes they can be equivalent. A cycle count for a given message is decremented every time the event associated with that message occurs. When this value reaches zero, the message is output and the cycle count is reset to the value of the output rate. See `SirfOutput()` in `UI_SIRF.C` for details of this bookkeeping procedure. If TricklePower is used, the actual output rate is a function of the message output rate and the TricklePower parameters. See “Effect of TricklePower on Message Rates” on page 9-9 for more details.

SiRF Binary Output Rates

The default SiRF message handlers and rates are mainly controlled by `amdSirf[]`, an array of message structures located in `UI_SIRF.C.` Each structure element in the array corresponds to a different message. Each structure contains the message identifier, the event on which the message waits, the function handler, the counting units (currently only cycles), the output rate and some bookkeeping variables. This structure controls both input and output messages with the input message actions given last and identified by their association with `MI_EV_INPUT` (representing an input event). The majority of default output rate changes can be accomplished by modifying this `amdSirf[]` array.

Example:

To change the output rate of the SiRF navigation message from 1 navigation cycle to 5 navigation cycles, change the following line:

```
{MID_MeasuredNavigation, MI_EV_NAV_COMPLETE, QueueMeasNav, MI_PERIOD_CYCLES,1,1,1,0},
```

to:

```
{MID_MeasuredNavigation, MI_EV_NAV_COMPLETE, QueueMeasNav, MI_PERIOD_CYCLES,5,1,1,0},.
```

The actual output rate in seconds depends on the rate of (in the case of the example) the navigation cycle. Typically, this value is one second for continuous operation or two seconds for the default Trickle power mode. To have a message default to no output, place a zero in the output rate field.

The second variable used to control SiRF protocol output is `SIRFMsgCntl`, a member of the `SRAM` structure declared in `SRAM.C`. (also see the tSRAMUI structure in SRAM_ICD.H). The `SRAM` structure is protected during power outages by battery backup. After a loss of battery back-up, or when first powered on, the `SRAM` structure is initialized by the `UI_SetUiSram()` function in `UI_SRAM.C`. The default value for `SIRFMsgCntl` is set in this function with the following line (see UI_SetUiSram( ) routine in UI_SRAM.C for details about ival);

case ID SIRFMSGCNTL:

```
SRAM.UI.SIRFMsgCntl=ival;
break;
```

`SIRFMsgCntl` is used to enable or disable blocks of messages based on the value of a number of bits (see SRAM_ICD.H for SP_XXX definitions). These blocks are essentially limited to raw GPS information and debug output. Implementation is best shown by the function `SiRFOpen()` in `UI_SIRF.C`. When the SiRF messaging protocol is first initialized, the value of `SIRFMsgCntl` is checked and if select bits are not set high, certain messages are disabled. Turning off a message is accomplished by calling `DisableSirfMsg()` which changes the output rate in the `amdSirf[]` array to zero for that particular message.

To enable debug output in SiRF Binary (see "Debugging Flash" on page 6-4), the `SP_DBGOUT` bit must be set high. To enable the raw track data, the `SP_RAWTRK` bit must be set high, this enables a block of messages in `SiRFOpen()`. The value of `SirfMsgCntl` is closely tied with the use of the Navigation Initialization message.

Example:

Consider defaulting the system to output raw GPS information. This can be accomplished by setting the `SP_RAWTRK` bit to one. In `UI_SRAM.C`, find the `UI_SetUiSram()` function, and remove the #ifndefine and #endif lines::

```
#ifndef RELEASE

SRAM.UI.SIRFMsgCnt1=SP DBGOUT | SP RAWTRK;
#endif
```

When SiRF binary protocol is initialized (in `SiRFOpen()`), the raw track output messages are enabled because the `SP_RAWTRK` bit is set. This change only takes place after the battery back-up is invalidated.

NMEA Default Output Rates

Changing the NMEA output rates is quite simple. Each of the seven available NMEA messages has its own `DEFINE` value for the output rate. All of these are located in `UI_IF.H` and are of the form `#DEFINE DEFAULT_###_RATE`, where ### is one of the message types (i.e., GGA). The default values along with the define values for each message are shown in the following table.

NMEA Message Identifier	Define Name	Value
GGA	DEFAULT_GGA_RATE	1
GLL	DEFAULT_GLL_RATE	0
GSA	DEFAULT_GSA_RATE	1
GSV	DEFAULT_GSV_RATE	5
RMC	DEFAULT_RMC_RATE	1
VTG	DEFAULT_VTG_RATE	0
MSS	DEFAULT_MSS_RATE	5

Currently, the NMEA default settings have the GGA, GSA and RMC messages output at one second intervals and the GSV message output at five second intervals. If the BEACON preprocessor definition is included, the MSS message is output and provides some Differential Beacon information. For more information on Differential operation see Chapter 11, “DGPS Operation.” To change the default NMEA message rates, the `UI_IF.H` file must be modified.

Example:

To output the GLL message at 2 second intervals the following line in `nmea.h`, change the following line:

```
#define DEFAULT_GLL_RATE 0
```

to:

```
#define DEFAULT_GLL_RATE 2
```

A zero output rate means that the message is not output at all. The output rate is also affected by TricklePower operation (see “Effect of TricklePower on Message Rates” on page 9-9). A 4800 Baud rate is not sufficient to output all of the messages at 1 Hz. Verify that the Baud rate is sufficient to output any required information.

Note – NMEA Compatibility and Fixed Length Fields – NMEA specification allows for some fields to have variable length. Some third-party software has been developed which parses NMEA messages but assumes that all fields are a fixed length. This often occurs when software is written for a specific GPS receiver which uses a fixed number of significant digits for one of the variable length fields.

The best way is to parse an incoming NMEA message based on comma separation. The format of the SiRF NMEA messages can be examined using the SiRFDemo executable provided with the SDK, or by examining the functions used to format the output messages. These are in `UI_NMEA.C` and generally have the form `Output###(VOID)`, where ### is the NMEA message identifier (i.e., GGA).

Example:

A GPS receiver offering 5 m level positioning may have a fixed latitude format of llll.lll. According to the NMEA specification, the first two digits are degrees, the second two are minutes and the part after the decimal place is variable and equal to decimal minutes. This gives a quantization level of 0.001 minutes, or about 1.9 m at sea level. A geodetic level receiver offering cm level positioning might require a latitude format of llll.llllll, giving a quantization level of 0.000001 minutes, or about 2 mm at sea level. Each length is fixed with respect to the receiver but external software must be written to accept either form.

Adding New Input/Output Messages

This chapter describes implementing a new user-specific message using the SiRF binary or NMEA protocol. It is recommended to use SiRF binary protocol because it is more compact, has better error checking, and enables the use of all SiRF-defined messages. This also allows you to examine the GPS performance of the unit using `SiRFdemo`. Both input and output messages can be implemented for either protocol. To add a new user message in a user-defined protocol is significantly more involved and is explained in Chapter 12, “Adding a New User Protocol.”

There are a number of ways to trigger the output of a message. The two most common are to wait for a recurring GPS event such as a navigation cycle or to reply to an input message (polled output). Both of these possibilities are covered in the following sections.

Limitations on Message Length

The default size of the UART buffers is 100 bytes, which includes 8 bytes for the transport layer. This data information is contained in `UARTBUF.C (see DEFINES prefixed with UART_).` If you want create a message with a data length longer than 92 bytes, you need to span buffers or increase the amount of memory allocated to the buffers. Allocating more memory to the buffers must be accompanied by an examination of available memory.

Note – There is no protection against the stack overwriting data space. It might be necessary to reduce in size or eliminate other buffers to maintain enough RAM for the stack and program execution.

There is some functionality provided for spanning a long message over a buffer. This is accomplished using a recursive function that constantly checks the remaining data length. Usage of this recursive function is demonstrated in `UI_SIRF.C`. The function of interest is `SirfPut()`. Some more information on the UART functions is given in Chapter 12, “Adding a New User Protocol.”

SiRF Binary

The SiRF protocol can be extended by adding another SiRF format message with a unique message ID or by replacing the data contents of a current SiRF input/output message. This documentation focuses on adding a completely new SiRF message. This is the recommended method for extending serial communications with the S2SDK.

To monitor SiRF output messages, you can use a console program. If the console program is set to display COMM activity in hexadecimal format then the start (0xA0A2) and end (0xB0B3) sequences for a SiRF binary message are easy to find. To isolate the new message and stop the output of any other messages follow these steps:

1. Stub the `SirfError()` function in `UI_SIRF.C` so no error messages are output.
2. Turn off the debug output by executing the following function call `NAVSetSerialDebugFlag(0)` (potentially at end of `SiRFOpen()`).
3. Comment out all other output (NOT input) messages in the `amdSiRF[]` array.

Adding a New SiRF Binary Output Message

Output messages can be polled (see “Adding a New SiRF Binary Input Message” on page 8-10) or output on the occurrence of a system (Module Interface) event as described in this section. Module Interface events are explained in more detail in “Module Interface Overview” on page 1-5. At this point, it is useful to glance through the `amdSirf[]` table in `UI_SIRF.C` and note the events that different messages wait on. This section focuses on adding a user message, output every time a Navigation Complete event (`MI_EV_NAV_COMPLETE`) is signaled by the GPS Core.

Each SiRF binary message has a unique Message ID (MID) to distinguish it from other messages. The format for SiRF binary messages is given in Appendix B, “SiRF Binary Messaging Functions,” and the MID numbers are provided in Appendix C, “Module Interface Details.” (Note that there are certain ranges of MIDs which are reserved for user applications.)

There are four steps to adding a new SiRF Binary user output message.

1. Define a message output structure.
2. Add an enumerated type MID to uniquely identify the output binary message.
3. Create a function to populate the message and call the output functions.
4. Add the message to the `amdSiRF[]` array and commit it to a trigger event.

Each of these steps are detailed in the following subsections.

Define a Message Output Structure

The new output message must have a defined data structure. As an example, the structure for the output message can be defined in `UI_IF.H` and have a `_MESSAGE` suffix.

Example:

```
typedef struct
{
    UINT8 MID;
    UINT8 data;
} USEROUT1_MESSAGE;
```

Note – The size of this message is 2 bytes. You cannot use the `sizeof()` function because it is possible that the compiler has padded your structure to align larger data elements (such as `UINT32`) to a word or long word boundary. The actual length of the message must be hardcoded into the user message output function. SiRF binary message lengths are contained in `UI_SIRF.C` in the `GetMsgSize()` routine.

Add an Enumerated Type Output MID

Each SiRF binary message must have a unique identifier to distinguish it from other input/output SiRF binary messages. For each new user message, a new MID must be added. It is important that the new MID be placed in the range reserved for the user, for output messages this is 0x61 (97) to 0x7F (127). See “SiRF Binary Messaging Functions” on page B-1 for more information.

Example:

To add a user output MID valued at 0x64 (100), the SPMessgeId enumerated type must be modified in `PROTOCOL.H`. This MID is placed in the range reserved for user output MIDs.

```
MID_Zero                         = 0x00,
MID_LookInMessage                = MID_Zero,

/* MODULE --> DEMO MESSAGES */
…
…
…
MID_MeasureData                  = 0x20,
MID_NavData                      = 0x21,
MID_WaasData                     = 0x22,
MID_TrkComplete                  = 0x23,
MID_TrkRollover                  = 0x24,
MID_TrkInit                      = 0x25,
MID_TrkCommand                   = 0x26,
MID_TrkReset                     = 0x27,
MID_TrkDownload                  = 0x28,
MID_GeodeticNav                  = 0x29,
MID_TrkPPS                       = 0x2A,
MID_CMD_PARAM                    = 0x2B,

MID_UserOutputBegin              = 0x61,
MID_UserOutput1                  = 0x64,
MID_UserOutputEnd                = 0x7F,

MID_NavigationInitialization = 0x80,
…
…
```

Create a Function to Output the Message

A function must be inserted into `UI_SIRF.C` that creates and outputs the new user message. Typically, data for fields in the new message can be obtained using a `MI_Getxxxx()` call. Other user system information can also be added.

Example:

Add prototype at beginning of file:

```
static WERR QueueUser1     (void);
```

Add function in main body:

```
WERR QueueUser1 (void)
{
    /* get buffer handle */
    UMBufHandle hBuf;
    USEROUT1_MESSAGE sMsg;

    hBuf = SirfAllocBuf (hComm, 2);
    if (!hBuf)
    {
        return FAILURE;
    }

    /* use MI_Get###() routines to populate message */

    sMsg.MID = MID_UserOutput1;
    sMsg.data = 0xFF;

    SEND_ITEM (hBuf, &sMsg.MID);
    SEND_ITEM (hBuf, &sMsg.data);
    SirfSend (hComm, hBuf);
    return SUCCESS;
}
```

Add the Message to amdSirf[] Array and Commit it to a Trigger Event

To output the new user message, a new entry in the `amdSirf[]` table must be added in `UI_SIRF.C`. The new entry must include the MID, the event that triggers the output, the associated output function and some bookkeeping variables. Each element in the `amdSirf[]` array has the structure form of `UI_SIRF_MSG_DEF (see UI_IF.H)` that is defined in `UI_SIRF.C`.

Example:

```
static UI_SIRF_MSG_DEF amdSirf[] =
{
    …
    {MID_DGPSStatus, MI_EV_NAV_COMPLETE, QueueDgpsSrc, MI_PERIOD_CYCLES,1,1,1,0}
    {MID_UserOutput1, MI_EV_NAV_COMPLETE, QueueUser1,    MI_PERIOD_CYCLES,1,1,1,0},
    {MID_OkToSend,    MI_EV_LAST_OUT,     QueueOkToSend, MI_PERIOD_CYCLES,1,1,1,0},

    …
}
```

This message is output based on the occurrence of a Navigation Complete event. Currently, the support for output based on time (`MI_PERIOD_MSEC)` is incomplete and all messaging output is based on navigation cycles (`MI_PERIOD_CYCLES`).

Adding a New SiRF Binary Input Message

Two common uses for a SiRF binary input message are to change some aspect of the internal state of the receiver (a set message) or to obtain some form of output message (a poll message). Messages that are used to poll for output are fairly easy to

implement. The messages used to set internal variables are set up slightly differently to enable the modification of the battery-backed RAM and the calculation of the CRC. When input messages of the set type are received, the system variables are not changed until the following event is signaled and `SirfOutput()` is called.

```
#define MI_EV_INPUT ((MI_EVENT)(MI_EV_WAIT_INITIAL_ACQ|MI_EV_NAV_COMPLETE))
```

Using a poll type message can accomplish most tasks (i.e., turning on an S2SDK LED or setting a nonbattery-backed user variable). An example of the set message type is included for completeness. When adding a new input message, it is easiest to copy and modify an existing set of input handlers. Some of the steps for implementing a new input message are very similar to the steps in “Adding a New SiRF Binary Output Message” on page 8-7.

To implement a new input message:

1. Define a message input structure
2. Add an enumerated type Message ID (MID) to uniquely identify the input binary message.
3. Create a Handler function to parse and perform bounds checking on incoming message. If this is a HandlePoll type input message then output the desired data. If this is a HandleSet type input message, set the `amdSirf[]` state so a corresponding `set` function (see step 5.) is called to alter the system variable after the next `MI_EV_INPUT` event.
4. For a HandlePoll message, if a SiRF binary message is desired, create a queue function to output the message. This may require some of the steps from “Adding a New SiRF Binary Output Message” on page 8-7
5. For a HandleSet message, generate a set function that alters the system variable. Add this function to the `amdSirf[]` table as an input event.
6. Register the input event handler in `SiRFOpen()`.

There are a number of ways to output information based on the reception of a SiRF binary message. The easiest is to use a debug output print statement, shown in “PRINTF Debugging” on page 6-9. The other option is to output a message following the SiRF binary message protocol. The following example assumes that the code example for the SiRF binary output message has been completed.

Testing with PROCOMM

Information on the SiRF Binary message structure, including the transport layer, see Appendix B, “SiRF Binary Messaging Functions” of this document and in Appendix C, “SiRF Binary Protocol Specification” in the *SiRFstarIIe Evaluation Kit User’s Guide*. To test the new input message, you can use PROCOMM or another terminal program that supports hexadecimal I/O. “Using PROCOMM to Send NMEA Messages” on page 6-13 provides some information on using meta keys for PROCOMM. Loading either the `NMEA100.KEY` or `DGPS.KEY` files (included in the SDK CD) into the meta key window provides some message examples.

Define a Message Input Structure

This is similar to “Define a Message Output Structure” on page 8-8 except the data structure is for input. There is no reason that these data structures cannot be identical. The structure can be defined in `UI_IF.H`.

Example:

```
typedef struct
{
UINT8 MID;
UINT8 data;
} USERIN1_MESSAGE;
```

Note – The size of this message is 2 bytes. You cannot use the `sizeof()` function because it is possible that the compiler has padded your structure to align larger data elements (such as `UINT32`) to a word or long word boundary. The actual length of the message must be hardcoded into the user function. SiRF binary message lengths are contained in `UI_SIRF.C`.

Add an Enumerated Type Input Message ID (MID)

This is similar to “Add an Enumerated Type Output MID” on page 8-8 except that the Input MID must be placed in range allocated for Input IDs instead of output IDs. The modification can still be made in `PROTOCOL.H`. The range for user input MIDs is 0xB4 (180) to 0xC7 (199). See “SiRF Binary Messaging Functions” on page B-1 for more information.

Example:

Add two input IDs, one for a set type message and one for a poll type message. A poll message can be used for most custom applications. To add a user input MID, the `MID_LookInMessage` must be added in `PROTOCOL.H`. These MIDs are placed in the range reserved for user input MIDs.

```
#define MID_Zero              (0x00)
#define MID_LookInMessage     MID_Zero,

/* MODULE --> DEMO MESSAGES */
…
…
#define MID_TailSync0xB3             = 0xB3,

#define MID_UserInputBegin           = 0xB4,
#define MID_UserInputPoll            = 0xBE, /* 190 */
#define MID_UserInputSet             = 0xBF, /* 191 */
#define MID_UserInputEnd             = 0xC7,

    #ifdef TEST_PACKET
    MID_TestPacket,
    #endif
    …
```

Create a Handler Routine to Parse Input Message

A handler function must be created that parses the input data. If this is a HandlePoll type input message, output the data as debug output or as a SiRF binary message. If this is a HandleSet type input message, modify the `amdSirf[]` table so a set function is called to alter the appropriate system variable after the next MI_EV_INPUT event occurs. Once the Handlers are created, they must be registered in the SiRF binary protocol and tied to the correct MID (see “Register the Input Handler” on page 8-16). These modifications are made in `UI_SIRF.C`.

Poll message example:

```
/* add header for Poll message handler */
static int HandlePollUserMsg   (UMHandle hMsg, UINT8 Id, UINT8 *pMsg, int Len);

/* local static structure to receive the message to */
static USERIN1_MESSAGE sUserInPoll;


/* Placed somewhere in file along with other INPUT message functions (i.e., Poll...)*/
/* handler function, outputs desired data, in this case calls         */
/* QueueUser1 from previous example. Note that this function, QueueUser1, should  */
/* be removed from the amdSirf[] table to prevent it from being output */
/* on both every navigation cycle event and every input message */
int HandlePollUserMsg (UMHandle hMsg, UINT8 Id, UINT8 *pMsg, int Len)
{
   sUserInPoll.MID = READ_RAW (pMsg, UINT8);
   sUserInPoll.data = READ_RAW (pMsg, UINT8);

   /* will output if NAVSetSerialFlag(1) */
   Printf("Received User Poll Message");

   /* call QueueUser1 to output User1 message (from previous example) */
   /* remove MID_UserOutput1 entry from amdSirf[] table               */

   if (QueueUser1() == SUCCESS)
   {
      PushAck (MID_UserInputPoll);
   }
   else
   {
      PushNak (MID_UserInputPoll);
   }
   return 1;
}
```

Set message example:

```
/* add header for Poll message handler */
static int HandleSetUserMsg    (UMHandle hMsg, UINT8 Id, UINT8 *pMsg, int Len);


/* local static structure to receive the message to */
static USERIN1_MESSAGE sUserInSet;


/* Placed somewhere in file along with other INPUT message functions (i.e., Set...) */
/* Add message handler. Note that this modifies the amdSirf table so that   */
/* the MID_UserInputSet entry will be executed on the next input event       */
int HandleSetUserMsg (UMHandle hMsg, UINT8 Id, UINT8 *pMsg, int Len)
{
   /* store input message info in global structure */
   /* for use by 'set' function                    */
   sUserInSet.MID = READ_RAW (pMsg, UINT8);
   sUserInSet.data = READ_RAW (pMsg, UINT8);

   /* signal the reception of this message */
   amdSirf[findidx(MID_UserInputSet)].State|=INPUT_READY;

   return 1;
}
```

Create a Queue Output Function for a Poll Message

For a Poll type input message, the message handler can output the desired information directly without waiting for an event. To output debug statements, the debug flag must be set. Normally, this is on by default but it can also be enabled using the `NAVSetSerialDebug(1)` command. Debug output might have been turned off following the directions in “SiRF Binary” on page 8-7. The following modifications are made in `UI_SIRF.C`.

Example:

This example makes use of the `Queue` function implemented in the example given in “Adding a New SiRF Binary Output Message” on page 8-7. To reuse this function, including its MID, it is necessary to comment out its entry in the `amdSirf[]` table. The line of interest (from “Add the Message to amdSirf[] Array and Commit it to a Trigger Event” on page 8-10) is:

```
{MID_UserOutput1, MI_EV_NAV_COMPLETE, QueueUser1,    MI_PERIOD_CYCLES,1,1,1,0},
```

Warning – If you do not do this, you will output this message on every Navigation Complete event and when the input message is received. This becomes difficult to debug.

Create a Set Function to Alter a System Variable for a Set Message

This step is only necessary if a system variable in battery-backed RAM is to be altered. Essentially, this mechanism enables the altering of the system variables in a timely manner. The handler function signals the function to be triggered after the next input event. The example below has not yet been fully populated but provides an idea of how to modify a user value that has been added to the battery-backed RAM structure. Note that there is very limited space left in this portion of RAM. See “ROM/RAM Requirements” on page 3-1 for details. Modifications are made in `UI_SIRF.C`.

Example:

```
/* put in a function prototype at the start of the file */
/* group this with the other 'set' functions            */
static WERR SetUserMsg    (void);

/* the function must be added to the amdSirf[] table,    */
/* note that this is last entry after MID_ChangeUartChnl */

static UI_SIRF_MSG_DEF amdSirf[] =
{
   …
   {MID_UserInputSet,     MI_EV_INPUT,SetUserMsg,   MI_PERIOD_CYCLES,1,1,1,0}
};


/* Add set function in INPUT section                                    */
/* Note that this function is not fully populated yet, but would allow */
/* for the setting of a battery-backed user value (in this case       */
/* identified by ID_USER in the UI_SetUiSram() function                    */
WERR SetUserMsg (void)
{
   if (!(amdSirf[findidx (MID_UserInputSet)].State & INPUT_READY))
   {
      return FAILURE;
   }
   amdSirf[findidx (MID_UserInputSet)].State &= ~INPUT_READY;

   umDebugPrintf("Received User Set Message");

   if (/*bounds check on input data*/)
   {
      /*potentially, an element of Battery-backed RAM could
      be modified here using UI_SetUiSram() function.
      UI_SetUiSram() would have to be modified, check for
      battery-backed memory space */

      /*    UI_SetUiSram (ID_USER, (void *) &sUserInSet.data, 0);   */
      PushAck (MID_UserInputSet);
      return SUCCESS;
   }

   PushNak (MID_UserInputSet);
   return FAILURE;
}
```

Register the Input Handler

The input handlers must be registered in the SiRF binary protocol so that they can be called when the appropriate MID is received. Modifications are made in `UI_SIRF.C`.

Poll example:

```
/*********************************************************************/
/* Inits appropriate UART objects. Registers UART handlers.          */
/*********************************************************************/
WERR SiRFOpen (UARTs Port, UMHandle hMsg)
{
   umSetSerialHandle (Port, hMsg); /* for SerialRxCheck [umanager.c] */
   hComm = hMsg;

   …
   …

   umRegisterForMessage (hMsg, (MessageId)MID_SetMsgRate,        HandleSetMsgControl);
   umRegisterForMessage (hMsg, (MessageId)MID_POLL_CMD_PARAM,    HandlePollCmdParam);
   umRegisterForMessage (hMsg, (MessageId)MID_UserInputPoll,     HandlePollUserMsg);
   …
   …

   return SUCCESS;
```

Set example:

```
/*********************************************************************/
/* Inits appropriate UART objects. Registers UART handlers.          */
/*********************************************************************/
WERR SiRFOpen (UARTs Port, UMHandle hMsg)
{
   umSetSerialHandle (Port, hMsg); /* for SerialRxCheck [umanager.c] */
   hComm = hMsg;

   …
   …

   umRegisterForMessage (hMsg, (MessageId)MID_SetMsgRate, HandleSetMsgControl);
   umRegisterForMessage (hMsg, (MessageId)MID_POLL_CMD_PARAM,HandlePollCmdParam);
   umRegisterForMessage (hMsg, (MessageId)MID_UserInputSet, HandleSetUserMsg);
   …
   …

   return SUCCESS;
}
```

NMEA

Adding additional NMEA messages is fairly straightforward. Although NMEA messages are not as compact as SiRF binary and are more error prone, they are often easier to implement since the output is ASCII and can be viewed using any console program. Note that in the NMEA protocol module, debug output is disabled. See "PRINTF Debugging" on page 6-9 for more details.

Adding a New NMEA Output Message

To add a new NMEA message, it is easiest to copy the implementation of an existing message. When `NUM_UI_NMEA_MSGS` changes, you must recompile multiple files. Any new NMEA messages must be placed at the end of the current NMEA message list so that communication with `SiRFdemo` functions correctly.

Note – Each new user NMEA output message adds another byte to the battery-backed `UI_SRAM` structure. You must verify that there is sufficient room for this operation (see Chapter 3, "Available System Resources").

Add a New XXX Message

Example:

Define the new defaults and add an internal ID for indexing purposes.

```
enum /*in UI_IF.H*/

{
    UI_NMEA_MSG_GGA=0,
    UI_NMEA_MSG_GLL,
    UI_NMEA_MSG_GSA,
    UI_NMEA_MSG_GSV,
    UI_NMEA_MSG_RMC,
    UI_NMEA_MSG_VTG,
#ifdef BEACON
    UI_NMEA_MSG_MSS,
#else
    PAD6,
#endif
    UI_NMEA_MSG_XXX,
    PAD7,
    PAD8,
    PAD9,
    NUM_UI_NMEA_MSGS
};

…

#define DEFAULT_XXX_RATE 1 /* in UI_NMEA.H*/
#define DEFAULT_XXX_CKSUM CHECKSUM_ON
```

Example:

The default values for the new output message must be storable in battery-backed RAM. The battery-backed RAM section of memory is initialized in `UI_SetUiSram()`. The message output rate and the checksum flag are packed into the same byte. Every added User NMEA output message adds one byte to the `UI_SRAM` structure and you must verify that there is enough room in battery-backed RAM.

UI_SRAM.C

```
void UI_SetUiSram(UI_SRAM_ID id, void *indata, UINT16 ival)
{
   UARTParams *pUParams;
   MI_NMEA_CFG *nmeaCfg;
   int i;

   BOOL valid=TRUE; /*true till proven false*/

   /*x-fer data from *indata to UI_SRAM struct and recalc crc*/
   switch(id)
   {
      …
      …
      …
      case ID_INITIALIZE:



/*        PRINTF("UI_SetUiSram: Initializing to default values"); TOO EARLY FOR ANY
PRINTING, PORTS NOT YET OPEN */

         /* First Clear the whole structure to zero */
         memset(&SRAM.UI, 0, sizeof(SRAM.UI));

         /* set up default PROTOCOL*/
         SRAM.UI.ProtocolA=UI_PROTO_DEFAULT;
         SRAM.UI.ProtocolB=UI_PROTO_RTCM;

         /* set up default comm parameters for port A AND Protocols*/
         SRAM.UI.NMEAbaud=NMEA_BAUD_RATE;
         SRAM.UI.SIRFcomm.baud=SIRF_BAUD_RATE;
         SRAM.UI.SIRFcomm.bits=SIRF_BITS;
         SRAM.UI.SIRFcomm.stop=SIRF_STOP;
         SRAM.UI.SIRFcomm.parity=SIRF_PARITY;

         /* USER1 Protocol setup*/
         SRAM.UI.USER1comm.baud  =USER1_BAUD_RATE;
         SRAM.UI.USER1comm.bits  =USER1_BITS;
         SRAM.UI.USER1comm.stop  =USER1_STOP;
         SRAM.UI.USER1comm.parity=USER1_PARITY;
```

Example (Continued):

```
 /* set up default comm parameters for port B*/
       SRAM.UI.DGPScomm.baud  =RTCMDefaultParams.baud;
       SRAM.UI.DGPScomm.bits  =RTCMDefaultParams.bits;
       SRAM.UI.DGPScomm.stop  =RTCMDefaultParams.stop;
       SRAM.UI.DGPScomm.parity=RTCMDefaultParams.parity;

         /* set up default NMEA msg controls*/
       SRAM.UI.nmeaCkSum = (DEFAULT_GGA_CKSUM << UI_NMEA_MSG_GGA) |
                            (DEFAULT_GLL_CKSUM << UI_NMEA_MSG_GLL) |
                            (DEFAULT_GSA_CKSUM << UI_NMEA_MSG_GSA) |
                            (DEFAULT_GSV_CKSUM << UI_NMEA_MSG_GSV) |
                            (DEFAULT_RMC_CKSUM << UI_NMEA_MSG_RMC) |
                            (DEFAULT_VTG_CKSUM << UI_NMEA_MSG_VTG);

       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_GGA] = DEFAULT_GGA_RATE;
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_GLL] = DEFAULT_GLL_RATE;
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_GSA] = DEFAULT_GSA_RATE;
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_GSV] = DEFAULT_GSV_RATE;
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_RMC] = DEFAULT_RMC_RATE;
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_VTG] = DEFAULT_VTG_RATE;
#ifdef BEACON
       SRAM.UI.nmeaOutputRate[UI_NMEA_MSG_MSS] = DEFAULT_MSS_RATE;
       SRAM.UI.nmeaCkSum |= DEFAULT_MSS_CKSUM << UI_NMEA_MSG_MSS;
#endif
       SRAM.UI.nmeaCtrl[UI_NMEA_MSG_XXX] = DEFAULT_XXX_RATE ;
       SRAM.UI.nmeaCkSum | = DEFAULT_XXX_CKSUM << UI_NMEA_MSG_XXX;…
       …
#ifdef RELEASE
       SRAM.UI.SIRFMsgCntl = SP_DBGOUT | SP_RAWTRK;
       …
       …
    }
}
```

Example:

```
UI_NMEA.C

char const *const apszNMEA[NUM_UI_NMEA_MSGS] =
{
    "GGA",/* GPS Fix Data */
    "GLL",/* Geographic position-lat/long */
    "GSA",/* GPS DOP and Active Satellites */
    "GSV",/* GPS Satellites in View */
    "RMC",/* Recommended minimum specific gps/transit data*/
    "VTG",/* Course Over Ground and Ground Speed */
#ifdef BEACON
    "MSS"   /* Internal beacon data*/
#else
    ""
#endif
    ,"XXX"
};
…
static WERR OutputXXX(void)
{
    char buf[SENTENCE_LENGTH];

    /* make calls to get system info, use MI_GET###() routines */
   /* note that '$' is added to beginning of string elsewhere */
    sprintf(buf,"GPXXX,<<< Comma delimited user info >>>");
    AddCheckSum(buf,UI_NMEA_MSG_XXX);

    return SendMsg(buf);
}
/* !!! IMPLEMENTATION NOTE: */
/*/* At this time, each message id  in this tablemust be unique and in     */
/* proper order.  This is because the table is indexed into using a       */
/* simple msg id as the index.  Later, if messagesmust be executed based */
/* on different MI_Event... events, changes need to be made to this file  */
/* and ui_sirf.c where SRAM.UI.NMEACfg[] array is accessed.               */

static UI_NMEA_MSG_DEF amdNmea[NUM_UI_NMEA_MSGS] =
{
{ UI_NMEA_MSG_GGA, MI_EV_NAV_COMPLETE, OutputGGA, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
{ UI_NMEA_MSG_GLL, MI_EV_NAV_COMPLETE, OutputGLL, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
{ UI_NMEA_MSG_GSA, MI_EV_NAV_COMPLETE, OutputGSA, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
{ UI_NMEA_MSG_GSV, MI_EV_NAV_COMPLETE, OutputGSV, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
{ UI_NMEA_MSG_RMC, MI_EV_NAV_COMPLETE, OutputRMC, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
{ UI_NMEA_MSG_VTG, MI_EV_NAV_COMPLETE, OutputVTG, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
#ifdef BEACON
{ UI_NMEA_MSG_MSS, MI_EV_NAV_COMPLETE, OutputMSS, MI_PERIOD_CYCLES, 1, 1, 1, 0 }
#else
 { PAD6,0,NULL, 0, 0, 0, 0, 0}
#endif
    { UI_NMEA_MSG_XXX,   MI_EV_NAV_COMPLETE,  OutputXXX,  MI_PERIOD_CYCLES, 1, 1, 1, 0 }
     { PAD8,0,NULL, 0, 0, 0, 0, 0}
     { PAD9,0,NULL, 0, 0, 0, 0, 0}
};
```

Adding a New NMEA Input Message

When adding a new NMEA input message it is easiest to copy an existing NMEA input message. The available user IDs for NMEA input messages are 200 to 255. An NMEA input message with ID number 200 has the form:

```
$PSRF200,< user specified data >……*cksum
```

To create an input handler, copy the NMEA100 handler. Only proprietary NMEA input messages are allowed. Information on using PROCOMM to test a new NMEA input message is provided in “Using PROCOMM to Send NMEA Messages” on page 6-13. The checksum for a new input message can be calculated using the `CKSUM.EXE` utility provided with the Evaluation and SDK packages. See Chapter 9, “SiRFstarIIe Toolkit Software” in the *SiRFstarIIe Evaluation Kit User’s Guide*.

Example:

Add a NMEA input message with ID 200.

```
UI_NMEA.C


static int Input100 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input101 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input102 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input103 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input104 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input105 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
static int Input106 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
#if 0
static int Input107 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);
#endif
static int Input200 (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Len);

WERR NmeaOpen (UARTs Port,UMHandle hMsg)
{
   …

   /* Register for the INPUT Messages to be handled*/
   umRegisterForMessage(hMsg, 100, Input100);
   umRegisterForMessage(hMsg, 101, Input101);
   umRegisterForMessage(hMsg, 102, Input102);
   umRegisterForMessage(hMsg, 103, Input103);
   umRegisterForMessage(hMsg, 104, Input104);
   umRegisterForMessage(hMsg, 105, Input105);
   umRegisterForMessage(hMsg, 106, Input106);
   umRegisterForMessage(hMsg, 200, Input200);

/* set DebugEnabled flag*/
   DebugEnabled = SRAM.UI.SIRFMsgCntl & SP_DBGNMEA? TRUE : FALSE;

   hComm=hMsg;
   return SUCCESS;
}
static int Input200 (UMHandle hMsg, UINT8, MsgId, UINT8 *pMsg, int MsgLen)
{
   /* add user code */
   static int Input200 (UMHandle hMsg, UINT8, MsgId, UINT8 *pMsg, int MsgLen)
   {
     int Scanf;
     Char buf[20];
     Scanf = sscanf((Char*) pMsg, "%s", buf);
     if ((!Scanf) || (Scanf==EOF))
   {
   DebugPrintf("Not Received User Message");
   Return 0;
   DebugPrintf("Received User Message");

   return 1;  /* one msg received */
}  /* end NMEA_ID: 200 */
```

Low Power Operation 9

The low-power receiver operation is a function that has been provided since the release of 2.0. There are two modes of low-power operation: TricklePower (TP) and Push-to-Fix (PTF). In TricklePower mode, the power to the SiRF chipset is cycled periodically, so that it operates only a fraction of the time. In Push-to-Fix mode, the receiver is generally off, but turns on frequently enough to collect ephemeris and maintain real-time clock calibration so that, upon user request, a position fix can be provided quickly after power-up. In both of the low power modes, the GPS hardware is controlled by GPIO4 and GPIO8 signals which provide an initial power up default condition, and are controlled thereafter by the operation of a hardware Finite State Machine (FSM) within the GSP2e and the GSW2 software.

TricklePower

This is a power-saving mode during which GPS is operational, and a position is output at a user-specified rate. During inactive periods, the various SiRF hardware components are either powered down or unclocked to reduce power consumption to very low levels.

There are two forms of TricklePower (TP) supported: external clock (E clock or ECLK) TP and GPS clock (GPSCLK) TP with ECLK TP being the lowest power solution and GPSCLK TP resulting in the lowest component solution. The main difference between ECLK TP and GPSCLK TP is that the ECLK implementation utilizes both the GPS clock and E clock, while the GPSCLK TP implementation utilizes only the GPS clock. There are schematic difference in the standard reference design which dictate which load options are to be used depending on the type of TP functionality desired.

There are three TricklePower states:

- Measurement (Tracking State)
- Navigation Computation (CPU)
- Inactive (TrickleState)

ECLK TricklePower

Tracking State - The tracking state is entered immediately after a hardware reset or a power up. In the tracking state, GPIO4 (connected to RF Regulator) and GPIO8 (connected to ECLK NAND Gate) are defaulted to logic high at power up. During the Measurement state, which can be as short as 200 msec, all SiRF hardware is powered On. The GPSCLK is used to clock the GPS tracking engine and to run the ARM7 CPU core elements. It is necessary to use the GPS clock to drive the CPU during periods when the RF section is on because the clock source for the CPU and the GPS DSP must be synchronous when the software is accessing registers on the GPS DSP side of the ASIC. The GPS satellite signals are acquired and tracked, and measurements are taken. Once the measurement interval (referred to hereafter as OnPeriod) has elapsed, the SiRF RF front end can be powered down, and the CPU state begins.

CPU State - During the transition to the CPU State, the control signal GPIO4 is switched from high to low, which causes the RF regulator to enter the shutdown mode. This effectively disables the RF front end including the GPSCLK source being output by the RF device. Immediately prior to this transition, CPU clock source is switched from the GPSCLK to the ECLK to maintain continuity of the clock. During the CPU state, position, velocity and time are computed based on the measurements taken during the tracking state. Other background tasks are also executed during this state.

Trickle State - During the transition to the Trickle State, the control signal GPIO8 is switched from high to low, which results in the ECLK NAND gate being switched off. Once navigation completes, the hardware reverts to the Trickle state during which time the ARM processor can be optionally put into a sleep state. This option is provided to allow the system to continue to communicate via the UARTS or to execute user tasks. A LPSetProcessorSleepAllowed() function is provided to force the processor to stay on at all times so the user may make use of the processor while GPS is idle.

In the situation where you want to keep the UARTs functioning at all times, you can set the UC_StopEClock value to FALSE in the UI_CONFIG.C file. This has the effect of leaving the ECLK powered during the TrickleState. The ARM is still in a wait for interrupt mode but the UARTs are active. This draws considerably more current and is only useful if you want to maintain constant communication

At the end of the Trickle State the real-time clock (RTC) is programmed with the time value that is equal to the user programmed interval minus the On time. The total time between subsequent TricklePower cycles is user-specified and is referred to as the TricklePowerInterval. The RTC is a count down timer. At the end of the countdown, the RTC generates a wake-up interrupt signal to the FSM and software, which causes the cycle to start over (i.e., the next Measurement state begins). The FSM toggles the GPIO8 line from low to high, which starts the CPU processing via ECLK, and the GPIO4 line is toggled from low to high via software control.

Note: The voltage to the GSP2e device is never removed. The lowest power state (trickle state) results from the fact that the internal clock tree driver internal to the GSP2e ASIC is switched Off to various sections of the device.

Figure 9-1 shows the TricklePower states over a period of two navigation cycles.

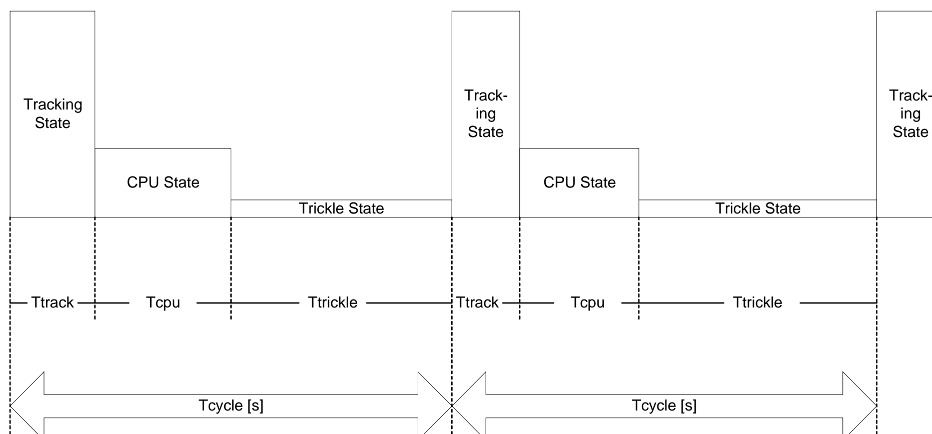

Figure 9-1 Diagram for ECLK TricklePower Showing the Various States and Approximate Current Consumption.

GPSCLK TricklePower

Tracking State - The tracking state is entered immediately after a hardware reset or a power up. In the tracking state, GPIO8 (connected to RF Regulator) and GPIO4 (connected to PWRCTL pin of the RF section) are defaulted to logic high at power up. In this configuration, the RF front end is fully powered On and provides the same functionality as ECLK TP.

CPU State - During the transition to the CPU State, the control signal GPIO4 is switched from high to low, which causes the RF section to enter the what is referred to as the "clock only" mode. This effectively disables the RF front from passing signals, but allows the RF section to continue to output the GPSCLK clock source to the GSP2e device.

Trickle State - During the transition to the Trickle State, the control signal GPIO8 is switched from high to low, which results in the RF Regulator to enter the shutdown mode. With no clock source available, as in ECLK TP, no UART communication or any other type of CPU functionality is supported during this state.

Note – In GPSCLK TP, the average current consumption will be slightly higher as compared to ECLK TP for the same setting.

The following tables summarizes the three TricklePower states in terms of the various system hardware elements and their operational states. The first column lists the hardware elements involved. All of these elements, except the RF section, are modules within the GSP2e.

Table 9-1 Summary of Clock Sources for ECLK TricklePower States

SiRF Hardware	Clock Source	TricklePower State		
		Tracking	**CPU**	**Trickle**
GRF2i	GPS Xtal	On		Off
GSP2e SSTE	GPSCLK	On	Clock Disabled	Clock Disabled
GSP I/O UARTS	(ECLK)	On	On	On (ECLK)or Clock Disabled
GSP ARM Processor	GPSCLK or (ECLK)	On (GPSCLK)	On (ECLK)	On (ECLK)or Clock Disabled
GSP Real-Time Clock	RTC Xtal	On	On	On

Table 9-2 Summary of Clock Sources for GPSCLK TricklePower States

SiRF Hardware	Clock Source	TricklePower State		
		Tracking	**CPU**	**Trickle**
GRF2i	GPS Xtal	On	On (RF section in Clock Only Mode)	Off
GSP2e SSTE	GPSCLK	On	Clock Disabled	Clock Disabled
GSP I/O UARTS	GPSCLK	On	On	Clock Disabled
GSP ARM Processor	GPSCLK	On	On	Clock Disabled
GSP Real-Time Clock	RTC Xtal	On	On	On

Enabling/Disabling TricklePower

You may enable or disable TricklePower operation through a call to `MI_SetLowPower()`. The prototype for this function is specified in `mi_if.h` (which must be included as header file) and is as follows:

```
WERR MI_SetLowPower (MI_LP_PARAM *pData);
```

where:

```
typedef struct
{
   INT16 PushToFix;
   INT32 OnTime;             /* in milliseconds */
   INT32 LPInterval;         /* in milliseconds */
    BOOL  PwrCyclingEnabled;
} MI_LP_PARAM;
```

- **`PushToFix`**
  0--> disable Push-to-Fix,
  1 --> enable Push-to-Fix

- **`OnTime`**
  Must be a multiple of 100. `OnTime` must be greater than or equal to 200 ms and less than or equal to 900ms. Must be set when setting TricklePower parameters, not needed when enabling Push-to-Fix.

- **`LPInterval`**
  Must be a multiple of 1000 (i.e., 1 second). Must be set when setting TricklePower parameters, not needed when enabling Push-to-Fix.

- **`PwrCyclingEnabled`**
  TRUE --> enable TricklePower
  FALSE --> disable TricklePower

Note – WERR is a SiRF type defined in stdtype.h that uses the values SUCCESS or FAILURE. The function returns FAILURE if TricklePower is not supported by the board on which the software is running. MI_SetLowPower also returns FAILURE if the OnTime is <200 ms or if the OnTime > 600ms and the LPInterval < 2s, or if the OnTime > 900 ms. If LPInterval <= OnTime (invalid TricklePower settings), then TricklePower is not enabled (enables Continuous Power) and Maximum Acquisition Time and Maximum Off Time are set to default values.

To deactivate TricklePower, the parameter PwrCyclingEnabled must be set to FALSE. If PwrCyclingEnabled is FALSE, the parameters OnTime and LPInterval are ignored.

To activate TricklePower, set PwrCyclingEnabled to TRUE, and set OnTime and LPInterval to the desired OnPeriod and TricklePowerInterval, respectively. Both quantities must be integer values with units of milliseconds. The PushToFix parameter must be set to 0, (i.e., Push-to-Fix disabled, when TricklePower is TRUE).

If invalid parameters are supplied, they are ignored, and TricklePower operation is disabled. When enabling/disabling TricklePower, the User task settings can be obtained by a call to UI_GetUserTaskParams() which returns UserTasksEnabled and UserTaskIntervalMs. To set these values and enable user tasks, see Chapter 10, "User Tasks, ASIC Interrupts, and the Scheduler." User tasks can be enabled or disabled independent of TricklePower or Push-to-Fix. Unless the user has supplied a user task, the parameter UserTasksEnabled must be set to FALSE. In this case, the UserTaskIntervalMs parameter is ignored.

Example:

This example sets TricklePower as default operation with a 300 ms on period and a 2-second TricklePower interval. This code modification is put in the UI_MSG.C file in the generic UI_Open() function call so it is enabled regardless of the I/O protocol that is going to be activated.

UI_MSG.C

```
WERR UI_Open (void)
{
    WERR Ret;
    MI_LP_PARAM UserLowPwr;
    struct UserParamStruct UserTaskParams;
    BOOL  UserTasksEnabled;
    int   UserTaskIntervalMs;

/* range check SRAM.UI.ProtocolA/B; set to default if exceeded */
   if ((SRAM.UI.ProtocolA >= UI_PROTO_MAX) ||
       (SRAM.UI.ProtocolB >= UI_PROTO_MAX) ||
       (!ValidateSRamNmeaCfg (SRAM.UI.NMEAbaud)) ||
      (SRAM.UI.crc != CompleteCRC16 ((UINT8 *) &SRAM.UI.ProtocolA, /* 1st element*/
                                     sizeof (SRAM.UI) - sizeof (SRAM.UI.crc))))
   {
      UI_SetUiSram (ID_INITIALIZE, 0, 0);
   }
    …
    …
    …

    /* Get current User Task information */
    UI_GetUserTaskParams (&UserTasksEnabled, &UserTaskIntervalMs);
    UserTaskParams.UserTasksEnabled = UserTasksEnabled;
    UserTaskParams.UserTaskIntervalMs = UserTaskIntervalMs;

    UserLowPwr.PushToFix   = 0;
    UserLowPwr.OnTime      = 300;
    UserLowPwr.LPInterval  = 2000;
    UserLowPwr.PwrCyclingEnabled = TRUE;

    /* Note we do not check return value */
    MI_SetLowPower(&UserLowPwr);
    /* Note we have not modified the user task params */
    UI_SetUserParameters (&UserTaskParams);

    return Ret;
}
```

Example:

This example uses the Poll SW version input message handler to adjust the TricklePower parameters. This code would never be implemented in practice, it is a demonstration only. This is actually a good debugging trick. If you want to adjust a parameter in the code, put this code in the Poll SW version input message handler and then use `SiRFdemo.exe` to execute it by selecting Poll | SW Version. In later versions of `SiRFdemo.exe`, you must disable the auto-detection by selecting Setup | Receiver S/W… to get this debug method to work. This example allows for a four stage change in TricklePower mode, ending with continuous power.

```
UI_SIRF.C

static int UserDebugCount = 0;
static MI_LP_PARAM UserLowPwr;


static int HandlePollSwVersion (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int Msg Len)
{



   pMsg += sizeof (UINT8);/*skip MsgId*/
   pMsg += sizeof (UINT8);/*skip Ctrl*/

   if (QueueSwVersion() == SUCCESS)
   {
   PushAck (MID_PollSWVersion);
   }
   else
   {
      PushNak (MID_PollSWVersion);
   }

   if (UserDebugCount++ > 3)
      UserDebugCount = 0;
```

Example (Continued):

```
    switch (UserDebugCount)
    {
       /* 300 ms on time, 1 second TP Interval */
       case 0:
          UserLowPwr.PushToFix   = 0;
          UserLowPwr.OnTime      = 300;
          UserLowPwr.LPInterval  = 1000;
          UserLowPwr.PwrCyclingEnabled = TRUE;
          break;
       /* 500 ms on time, 2 second TP Interval */
       case 1 :
          UserLowPwr.PushToFix   = 0;
          UserLowPwr.OnTime      = 500;
          UserLowPwr.LPInterval  = 2000;
          UserLowPwr.PwrCyclingEnabled = TRUE;
          break;
       /* 500 ms on time, 3 second TP Interval */
       case 2 :
          UserLowPwr.PushToFix   = 0;
          UserLowPwr.OnTime      = 500;
          UserLowPwr.LPInterval  = 3000;
          UserLowPwr.PwrCyclingEnabled = TRUE;
          break;
       /* continuous power */
       case 3 :
          UserLowPwr.PushToFix   = 0;
          UserLowPwr.OnTime      = 0;
          UserLowPwr.LPInterval  = 0;
          UserLowPwr.PwrCyclingEnabled = FALSE ;
          break;
    }

    /* Note we do not check return value */
    MI_SetLowPower(&UserLowPwr);

    return 1;
}
```

Effect of TricklePower on Message Rates

In continuous mode, navigation cycles occur every second. During TricklePower operation, the navigation cycle time is specified by the user and all messages are output at a rate dependent on the user-specified TricklePower parameters. The receiver outputs an OK to send message (MID 0x12) with a 0x01 data byte to indicate that it is ready to receive data. When the receiver is unable to receive data, it outputs an OK to send a message with a 0x00 data byte indicating that communication must stop. This type of handshaking is used by SiRFdemo as a form of flow control. Messages are not lost if this handshaking method is used. See the *SiRFstarIIe Evaluation Kit User's Guide* for more details.

The output of SiRF binary output messages is fairly straightforward under these conditions since the output rates are specified in multiples of cycles (see "User Interface Overview" on page 1-5). NMEA messages, on the other hand, have output rates defined in periods of seconds and thus are only output when the NMEA message period and a navigation cycle are coincident.

Example:

If the user specifies a 3-second TricklePower cycle and a NMEA output rate of 5 seconds for a GGA message, the message is output every 15 seconds.

Push-to-Fix

In Push-to-Fix mode, the receiver automatically awakens every Push-to-Fix period (default is every 30 minutes (1800 seconds)) to obtain a position fix, collect ephemeris (if needed), and calibrate the real-time clock (RTC) (if needed). This GPS functionality allows a quick navigation solution to be obtained when the user requests it by pushing the reset button (S2) to reset the receiver. Upon activation, it searches for up to the Maximum Acquisition Time to obtain a GPS solution. The maximum Acquisition Time is described in "Setting Low Power Acquisition Parameters" on page 9-13. If a solution is not obtained during that time, the receiver deactivates until the Push-to -Fix period is reached. There is no duty cycle in Push-to-Fix operation as there is in TricklePower. The receiver reactivates and tries again, and this cycle repeats until a successful GPS solution is computed. Once that has happened, the receiver deactivates for another Push-to-Fix period and the process repeats itself.

The default for the Push-to-Fix period is 30 minutes and this update rate can be changed via an MI function. In fact, the user settable range for the Push-to-Fix period can be set as low as 10 seconds and as much as 2 hours (7200 seconds).

Unless the receiver has failed to refresh ephemeris for approximately four hours, such that ephemeris has gone out of date, you can obtain a quick GPS solution (through a hot start) upon resetting the receiver.

Enabling/Disabling Push-to-Fix

Push-to-Fix operation is controlled through the function `MI_SetLowPower()` as described in the Section "Enabling/Disabling TricklePower" on page 9-4. The function prototype is reproduced here for convenience:

```
WERR MI_SetLowPower (MI_LP_PARAM *pData);
```

where:

```
typedef struct
{
   INT16 PushToFix;
   INT32 OnTime;            /* in milliseconds */
   INT32 LPInterval;        /* in milliseconds */
   BOOL  PwrCyclingEnabled;
} MI_LP_PARAM;
```

- **`PushToFix`**
  0--> disable Push-to-Fix
  1 --> enable Push-to-Fix
- **OnTime**
  Must be a multiple of 100. `OnTime` must be greater than or equal to 200ms and less than or equal to 900ms. This parameter is used to set the TricklePower on time and is not used when enabling Push-to-Fix
- **`LPInterval`**
  Must be a multiple of 1000 (i.e., 1 second). This parameter is used to set the TricklePower on time and is not used when enabling Push-to-Fix
- **`PwrCyclingEnabled`**
  TRUE --> enable TricklePower
  FALSE --> disable TricklePower

Note – `WERR` is a SiRF type defined in `stdtype.h` that uses the values `SUCCESS` or `FAILURE`. The function returns `FAILURE` if TricklePower is not supported by the board on which the software is running. `MI_SetLowPower` also returns `FAILURE` if the OnTime is <200 ms or if OnTime > 600ms and LPInterval < 2s, or if the OnTime>900ms.

Push-to-Fix may be deactivated by calling `MI_SetLowPower` with the PushToFix parameter set to 0. To activate Push-to-Fix, `MI_SetLowPower` must be called with the PushToFix parameter set to 1.

Push-to-Fix mode takes precedence over TricklePower and for Push-to-Fix operation both PwrCyclingEnabled and PushToFix must be set to TRUE. Maximum Acquisition Time and Maximum Off Time defaults to 120000 ms (120 sec) and 30000 ms (30 sec), respectively, just as in TricklePower. However, in Push-to-Fix mode the Maximum Off Time is not used, the PushToFix period is used. These may be modified by a subsequent call to `MI_SetAcqParams`, see “Setting Low Power Acquisition Parameters” on page 9-13. User tasks can be enabled during Push-to-Fix operation, as described in Chapter 10, “User Tasks, ASIC Interrupts, and the Scheduler.”

Example:

This code example sets the receiver into Push-to-Fix mode as default. To restart the S2SDK after the board has gone into Low Power mode press the reset button (S2).

```
UI_MSG.C
WERR UI_Open (void)
{
   WERR Ret;
   MI_LP_PARAM UserLowPwr;

   /* range check SRAM.UI.ProtocolA/B; set to default if exceeded */
   if ((SRAM.UI.ProtocolA >= UI_PROTO_MAX) ||
       (SRAM.UI.ProtocolB >= UI_PROTO_MAX) ||
       (!ValidateSramNmeaCfg (SRAM.UI.NMEAbaud)) ||
      (SRAM.UI.crc != CompleteCRC16 ((UINT8 *) &SRAM.UI.ProtocolA,
                                    sizeof (SRAM.UI) - sizeof (SRAM.UI.crc)))
     )
   {
      UI_SetUiSram (ID_INITIALIZE, 0, 0);
   }
…
…
…
   pProtoA = &ProtocolCfg[SRAM.UI.ProtocolA];
   pProtoB = &ProtocolCfg[SRAM.UI.ProtocolB];

   Ret  = pProtoA->InitMsgTable();  /* load saved SRAM table cfg */
   Ret |= pProtoB->InitMsgTable();
   Ret |= pProtoA->Open (UART_A, hportA);
   Ret |= pProtoB->Open (UART_B, hportB);

   /* Get current low power parameters */
   MI_GetLowPower(&UserLowPwr);

   /* Enable Push-to-Fix */
   UserLowPwr.PushToFix   = 1;
   UserLowPwr.PwrCyclingEnabled = TRUE;

   /* Set Push-to-Fix */
   MI_SetLowPower(&UserLowPwr);

   return Ret;
}
```

Setting Low Power Acquisition Parameters

In addition to `OnTime` and `LPInterval`, there are two additional TricklePower parameters that can be set. These are the Maximum Acquisition Time and Maximum Off Time. In Push-to-Fix mode, the Push-to-Fix period is used instead of the Maximum Off Time. The MI_SetPtfPeriod and MI_GetPtfPeriod routines must be used to change the Push-to-Fix period, see Appendix C for module interface details.

Although OnPeriod and TricklePowerInterval dictate the GPS receiver behavior in normal TricklePower operation when an unobstructed view of satellites are available, there are special provisions for situations in which a GPS position cannot be computed due to blocked visibility. In such cases, TricklePower is temporarily disabled, (i.e., switches to continuous power operation), until a navigation solution (fix) is obtained or the Maximum Acquisition Time is reached. Setting the Maximum Acquisition Time allows power to be saved even in situations where a search for satellites is unsuccessful. The Maximum Acquisition Time is the time at which the receiver gives up and deactivates if unable to compute a GPS solution. When this happens, the receiver deactivates for the Maximum Off Time while in TricklePower operation.

Example:

OnPeriod = 300 ms, TricklePowerInterval = 2 seconds. Thus, duty cycle = 0.3/2.0 = 0.15. If Maximum Acquisition Time = 120 seconds (2 minutes) and Maximum Off Time = 30 seconds, and the receiver is running in an obstructed location such as a parking garage, then the receiver searches for satellites for 2 minutes. If it does not find any satellites the receiver deactivates for the Maximum Off Time of 30 seconds (default value). If the user wants to maintain the same TricklePower duty cycle while the receiver is trying to find satellites, then the Maximum Off Time needs to be adjusted. For example; 120 sec divided by 0.15 = 800 seconds (13-1/3 minutes).

Therefore, the Maximum Off Time can be viewed as a limit on the amount of time the receiver turns off, regardless of how long it searches fruitlessly for satellite signals. Conversely, for applications in which power management is more important than limiting power-off periods, (e.g., remotely sited receivers in locations with intermittent blockages), where maximal power savings is desired, the Maximum Off Time can be set to a period greater than 800 seconds so that a lower duty cycle is maintained.

The user may set the Low Power acquisition parameters through a call to `MI_SetLpAcqParams()`. The prototype for this function is specified in `mi_if.h` and is as follows:

```
WERR MI_SetLpAcqParams (MI_LP_ACQ_PARAM *pData);
```

where:

```
typedef struct
{
    UINT32 MaxAcqTime;  /* in milliseconds */
    UINT32 MaxOffTime;  /* in milliseconds */
} MI_LP_ACQ_PARAM;
```

- **`MaxAcqTime`**
  Must be a positive integer not greater than MAX_INT32.
- **`MaxOffTime`**
  Must be a positive integer not greater than MAX_INT32.

Note – WERR is a SiRF type defined in `stdtype.h` that uses the values `SUCCESS` or `FAILURE`. The function returns `FAILURE` if TricklePower is not supported by the board on which the software is running. The function also returns `FAILURE` if MaxAcqTime or MaxOffTime is out of bounds (i.e., 0 or greater than `MAX_INT32`). The defaults of 120 s and 30 s are used in this case.

When `MI_SetLowPower` is called, the values of the Maximum Acquisition Time and Maximum Off Time are set to default values of 120 seconds and 30 seconds, respectively. Once Low Power is enabled (through a call to `MI_SetLowPower`), calling `MI_SetLpAcqParams` takes effect within the next TricklePower cycle, and the user's parameters `MaxAcqTime` and `MaxOffTime` are used instead of the default values.

If `MI_SetLpAcqParams` is called with `MaxAcqTime` or `MaxOffTime` out of range, both parameters are ignored, and the default values (120000 ms for `MaxAcqTime` and 30000 ms for `MaxOffTime`) are used instead.

The current settings for these parameters can be obtained through a call to `MI_GetLpAcqParams`.

```
WERR MI_GetLpAcqParams (MI_LP_ACQ_PARAM *pData);
```

User Tasks, ASIC Interrupts, and the Scheduler 10

Task based processing in the SiRFstarIIe is controlled by the scheduler. The scheduler is activated every time a 100 ms interrupt is received from the ASIC. When an ASIC interrupt is signaled, the `SCH_ISR()(SCHEDULE.C)` is called and the source of the interrupt is determined by examining the `ASIC_INTSTAT` (0x800A0010) register. If the source is the 100 ms interrupt (0x0040, bit 6) then the scheduler is activated. See the *SiRFstarIIe System Development Kit User's Guide Part 2 - GSP2e Chip* for more details about the interrupt controller and interrupt registers.

When the interrupt is received, the scheduler updates the system time and then schedules the highest priority task currently in the task queue to be launched after the interrupt routine terminates. In this way, a high priority task can be placed in the task queue and then launched after the next 100 ms interrupt, regardless of the low priority background task currently active. If a task is running, and the scheduler determines that no higher priority tasks have been scheduled since the last 100 ms interrupt, it continues to execute. Tasks are ordered by priority at compile time, with the most critical tasks given the highest priority. Task priorities are set in `SCH_ICD.H`.

For custom applications, the code for a user task has already been added. This implementation ensures that a user task will run at the desired interval, regardless of the Low Power state. See "Adding a User Task" on page 10-4 for implementation details.

Note – GPS is extremely time critical and if a user task interferes with the GPS operation, tracking and navigation are adversely affected.

Figure 10-1 demonstrates how the scheduler works. The key is that the IRQ ISR is re-entrant. Assume that a 100 ms interrupt has already occurred and a low priority task is currently running: this means that the IRQ handler was entered, the asicISR( ) function was executed and program execution reached the Dispatch function. The Dispatch function ends up executing the function associated with the low priority task. Just before entering the Dispatch function, interrupts are enabled.

Now assume that we get another 100 ms interrupt. The old IRQ is interrupted (remember that we are running the task function through Dispatch with interrupts enabled) and a new IRQ interrupt is started, forming a type of nested interrupt. Now when we get to the Dispatch function, it checks to see if the current highest priority task (of any type) is the old running task. If so, the IRQ terminates and the old IRQ resumes running (meaning the old task resumes). If there are higher priority tasks

which have been scheduled, the IRQ loops until the higher priority tasks are done (assuming it is not interrupted again). If it finishes all the other scheduled higher priority tasks, then the IRQ terminates and the old task (and IRQ) resume.

While looping in the Dispatch function, tasks are only executed if they are on the scheduled task list. Tasks are removed from the scheduled list as soon as they get executed. Because of this, the old running task is not executed again.

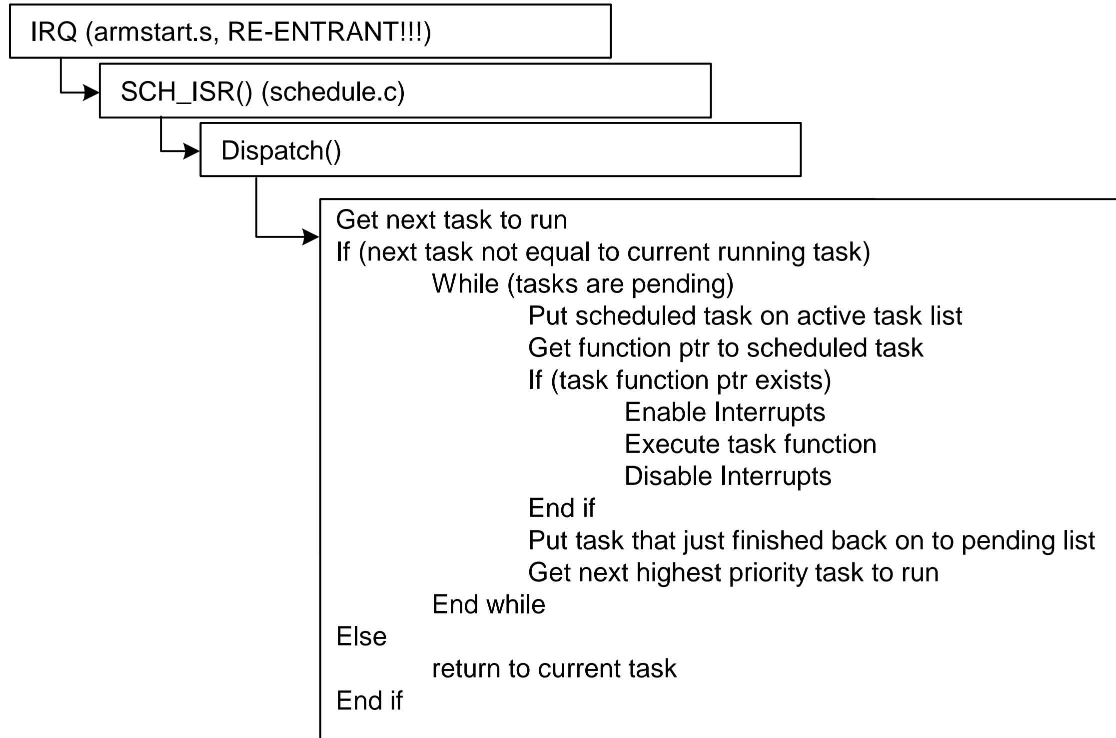

Figure 10-1 Workings of the Scheduler

ASIC Interrupts

The satellite signal tracking loop processing is handled by the satellite signal tracking engine (SSTE) implemented in hardware. The fastest periodic interrupt generated by the GSP2e is at a 100 ms rate. This rate is greatly reduced compared to the previous generations of GPS ASICs requiring 1 ms interrupt servicing. As a result of this, the software scheduler currently runs at a maximum rate of 10 Hz (100 ms). The GSP2e is still capable of generating both 1 ms and 20 ms interrupts. These signals are disabled by default, but can be activated by adjusting the interrupt enable register `ASIC_INTENA` (0x800A0004). Bits 7 and 8 (0x0080 and 0x0100) are used for the 20 ms and 1 ms interrupts respectively. The timer interrupts must be acknowledged by writing to the timer interrupt acknowledge register `ASIC_TIMERACK` (0x800A0038) with bit 2 (0x0004) acknowledging the 20 ms interrupt and bit 4 (0x0010) acknowledging the 1 ms interrupt.

The GSP2e has two interrupt levels, IRQ and FIQ, with FIQ being the highest. All ASIC interrupts are generated on the same ARM IRQ interrupt level. These must be identified based on the contents of a status register. ARM FIQ interrupt level is not

used at present, and therefore available for user implementations. The ASIC interrupt service routine `asicISR()` is called in response to any IRQ-level interrupt. Based on the bits set in the interrupt status register, the appropriate interrupt handler(s) in the following subsections are called. Initial Acquisition processing is also called from this handler when the 100 ms interrupt is present during initial acquisition. For more information on interrupt handling, see the *SiRFstarIIe System Development Kit User's Guide Part 2 - GSP2e Chip*.

Timer Interrupt

Timer interrupts are generated periodically based on the system clock (different from the processor clock). The timing of this interrupt is controlled by S/W. Currently it is set at 100 ms which is the minimum rate fully supported by the Satellite Signal Tracking Engine, or Tracker (SSTE). The GPS S/W tasks are scheduled from this interrupt handler, including the 100 ms task, the 1 Hz execution task, and the background task (scheduled at 1 Hz). The 100 ms task performs processing of measurement data of channels that are tracking, or processing of reacquisition data if the channels are in search. The 1 Hz execution task controls the receiver manager, 50 BPS data processing, navigation, and user interface through the serial communication port. The Background task performs Satellite State Computation and Satellite Selection.

UART Interrupt

UART interrupts are generated by the UART on the ASIC whenever there is data to be transmitted or data is received. The processing of these interrupts is done by the UART functions called `uartISRHandler()`. This handler calls the appropriate transmit or receive processing for UART A and/or UART B. Note that the ISR does not deliver the data directly to the application layer. The data is buffered and passed to the application layer (user interface) after the GPS core has signaled a MI_EV_NAV_COMPLETE or MI_EV_WAIT_INITIAL_ACQ event.

Low Power Operation Interrupt

Low Power Operation interrupts are generated by the Real Time Clock (RTC). The RTC is programmed to generate a wake up pulse after the required off-time elapses. The RTC is used for scheduling user tasks and to activate the receiver for the start of a new low power mode navigation cycle. More information on User tasks can be found later in this chapter. The Low Power software has built-in logic to determine which task is appropriate.

Beacon Interrupt

The Beacon interrupts are generated by the internal Beacon only if the software has been compiled with the Beacon option. The Beacon option is no longer user settable. The processing of these interrupts is handled by the Beacon Interrupt Handler which acknowledges the interrupt, performs initial data processing, and schedules the execution of the function to handle the DGPS interrupt data for further processing if

necessary. The function to handle the DGPS interrupt data is the primary routine for the internal DGPS receiver data processing that processes the data or checks the data validity.

Adding a User Task

A provision for periodic scheduling of a user task is included, and can be activated by compiling with the preprocessor macro `TASK_PERIOD` defined (see "Basic Compile Switches" on page 4-5). `TASK_PERIOD` must be an integer in units of milliseconds. For example, to periodically schedule a task to be executed every 200 ms, define `TASK_PERIOD` as 200. Source files affected by `TASK_PERIOD` are `schedule.c`, `exec_if.h`, `sch_icd.h`, and `user.c`. To disable user tasks, define `TASK_PERIOD` as 0.

The user task priority is set to 4, which is higher priority than the SiRF navigation task (1 sec task) but lower than the tracker task (100 ms task). Task priorities are defined in `exec_if.c`. For TricklePower, you must verify that the user task does not demand excessive processor throughput; (i.e., there must be enough throughput available so that the SiRF navigation task has enough time to complete before the next TricklePower wakeup interrupt). You must also verify that the user task can be completed before the next user task is scheduled `TASK_PERIOD` milliseconds later.

With `TASK_PERIOD` defined to the desired period, a call to the user function must be placed in `UI_UserTaskFunction()`, which is defined in `user.c`. User tasks are scheduled using the Real-Time Clock (RTC) regardless of whether a Low Power mode is active or not. During deactivated periods, the processor activates every `TASK_PERIOD` milliseconds, executes the user task, and then deactivates. Some other notes are given below.

- You must verify that the task can run before the next one is scheduled. With the SiRF scheduler, the user task is not called re-entrantly if there is already one running when another is scheduled, it goes on to the pending task list. However, if you consistently schedule user tasks faster than they are being executed, you can end up losing some.
- If you are using TricklePower, the TricklePower interval must be an integer multiple of the task interval because the RTC is driving both the user task and TricklePower. A TricklePower cycle is launched in response to an RTC interrupt (same interrupt driving the user task) provided that the TP interval has elapsed. If you set the task interval longer than the TP interval, the TP interval will in effect be equal to the task interval. Thus the RTC is not the right mechanism for launching occasional user tasks that run less frequently than the TP cycle.
- If you take too much throughput at a priority higher than GPS, you receive one-second overruns and the GPS may fail (stops navigating).

Example:

Adding a user task at 200 ms that runs out 20 ms of processor time and flashes one of the S2SDK LEDs. The 20 ms period is broken up into 2 ms slices. This is because the `DelayMicroSeconds()` function disables interrupts and this cannot be allowed over a full 20 ms period. First you must set the preprocessor definition TASK_PERIOD = 200 (see "Basic Compile Switches" on page 4-5).

Example (Continued):

```
USER.C

void UI_UserTaskFunction(void)
{
   /* test task -- gobble up 20 ms in 2 ms chunks. Interrupts are turned off during
   * DelayMicroSeconds. Don't want to have interrupts turned off the full 20 ms because
   * it will kill I/O, which interrupts us every ~4 ms */

   int i;

   for (i = 0; i < 10; i++)
   {
      DelayMicroSeconds(2000); /* 2 ms delay */
      /* toggle LED */
      ASIC_GPIO_PORTDIR2 |= 0x20;
      ASIC_GPIO_PORTVAL2 ^= 0x20;
   }

   return;
}
```

Start/Stop GPS Functions

Two functions, `LPUserStartGPS()` and `LPUserStopGPS()`, are provided so that you can control SiRF GPS operation. When `LPUserStopGPS()` is called, the GRF chip is turned off immediately. GPS tasks that are currently executing or scheduled run to completion, after which no further GPS tasks are run. The processor remains activated. If `LPUserStartGPS()` is subsequently called, the software resets and GPS operation resumes.

Note – Any user tasks running when `LPUserStartGPS()` is called, including the task from which `LPUserStartGPS()` is called, are terminated by the reset. Also, when `LPUserStartGPS()` is called, it resets the whole receiver. When the GPS portion of the receiver is turned off, there are no events generated by the GPS Core and hence no message input/output is recognized. To use any Protocol event handlers, you must generate your own event. Two examples of using the GPS start/stop commands follow.

Example:

Use a user task to shut down the GPS after 60 seconds and then back on after 10 seconds. The preprocessor definition TASK_PERIOD=200 must be set (see “Basic Compile Switches” on page 4-5”). Note the preprocessor definition SDKTEST_GPSONOFF can also be used to enable the code found in USER.C file.

```
USER.C

#if defined(SDKTEST_USERTASK)//defined(SDKTEST_GPSONOFF)
static long int StartStopCount = 0;
#if defined(SDKTEST_GPSONOFF)
static long int CountWhenStopped = 0;

void UI_UserTaskFunction(void)
{
    StartStopCount++;
    if (!LPQueryGPSStopped())
    {
        if (StartStopCount > 300) /* 60 seconds * 5 user tasks per second */
        {
            umDebugPrintf("Turn off GPS");
            LPUserStopGPS();
            CountWhenStopped = StartStopCount;
        }
    }
    else
    {
        if (StartStopCount - CountWhenStopped > 50) /* 10 sec * 5 user tasks per sec */
        {
            umDebugPrintf("Turn on GPS");
            LPUserStartGPS(); /* this will reset the receiver */
        }
    }
     return;
}
```

Example:

Use the Poll SW Version input message to start and stop the GPS. See the second example in “Enabling/Disabling TricklePower” on page 9-4 for more information on using the Poll SW version command in `SiRFdemo.exe` as a debugging aid. The

preprocessor definition TASK_PERIOD=200 must be set (see “Basic Compile Switches” on page 4-5”). To use the protocol functions, a user event must be generated by the user task. This test is done under continuous power.

```
MI_IF.H

typedef enum
{
   MI_EV_NONE                 = 0,
   MI_EV_MEASUREMENT_RCVD     = 1,
   …
   …
   …
   MI_EV_NL_INIT_DONE         = 2048
   ,MI_EV_USER                 = 4096 /* add user event */
} MI_EVENT;
```

```
USER.C

static Count200ms = 0;

void UI_UserTaskFunction(void)
{
   /* Set up user function to trigger a user event once a second */
   if (Count200ms++ == 5)
   {
      /* Note that this debug output will only work */
      /* because we are signaling an event          */
      umDebugPrintf("User Task one second");
      Count200ms = 0;
      /* If GPS is stopped, signal our own one second event */
      if (LPQueryGPSStopped())
      {
      UI_Event(MI_EV_USER, 0);
      }
   }
   return;
}
```

```
UI_SIRF.C

/* Make sure input messages used to set variables are still accepted */
#define MI_EV_INPUT     ((MI_EVENT)(MI_EV_WAIT_INITIAL_ACQ|MI_EV_NAV_COMPLETE|MI_EV_USER))
#define MI_EV_LAST_OUT  ((MI_EVENT)(MI_EV_MEASUREMENT_RCVD|MI_EV_WAIT_INITIAL_ACQ))
...
...
...
```

Example (Continued):

```
}WERR SirfInput (MI_EVENT Event, UINT32 TimeOutput)
{
   int i;

   if ( ( (Event & MI_EV_NAV_COMPLETE) != MI_EV_NAV_COMPLETE ) &&
        ( (Event & MI_EV_USER) != MI_EV_USER) )
   {
      return SUCCESS;
   }

   umSerialRxCheck();  /* check for new input msgs */

   /* Process any input msgs */
   for (i = 0; i < sizeof (amdSirf) / sizeof (UI_SIRF_MSG_DEF); i++)
   {
      if (amdSirf[i].State & INPUT_READY)
      {
         if (amdSirf[i].Handler)
         {
            amdSirf[i].Handler();
         }
      }
   }

   return SUCCESS;
...
...
...
static int HandlePollSwVersion (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int MsgLen)
{
   pMsg +=sizeof(UINT8); /*skip MsgId*/
   pMsg +=sizeof(UINT8); /*skip ctrl*/
   if(QueueSwVersion()==SUCCESS
   {
      PushAck(MID_PollSwVersion);
   }
   else
   {
      PushNak(MID_PollSwVersion);
   }
   return 1;
   }
```

Example (Continued):

```
#if 1
    /* If the GPS is currently off, turn it on (reset the board) */
    if (LPQueryGPSStopped())
        LPUserStartGPS();
    /* else shut down the GPS */
    else
        LPUserStopGPS();
#endif

    return 1;
}
```

DGPS Operation 11

The SiRFstarIIe is capable of differential GPS operation, enabling a number of possible sources for GPS range corrections. In Differential GPS, range corrections to each visible satellite are generated at a known stationary site and then transmitted to the SiRFstarIIe. These range corrections are then applied to the measured ranges, and compensate for the majority of the atmospheric error. The SiRFstarIIe can obtain these corrections through the Wide Area Augmentation System (WAAS), a beacon receiver or an independent system supporting RTCM corrections. More information about each of these sources is provided in the following sections. The `WAAS` and `BEACON` preprocessor definitions must be set to have WAAS and Beacon differential capability (see “Basic Compile Switches” on page 4-5).

Setting Differential Correction Source

There are four possible settings for differential correction sources. The first three are ready to use and include WAAS, internal beacon, and external RTCM. There is also provision for a user defined interface that provides the DGPS corrections directly to the GPS Core through a Module Interface routine. WAAS provides GPS corrections through a series of Geostationary satellites. The advantage of WAAS is that the signal is broadcast on the same frequency as GPS and no extra hardware is needed. The disadvantage is that WAAS is a fairly weak signal, similar to GPS, and the receiver must have a line-of-sight to the satellite.

The S2SDK comes equipped with an internal beacon receiver, capable of tracking the Coast Guard Radio Beacons that provide GPS corrections. This is a fairly robust Differential GPS provider since it does not require line-of-sight communication. The disadvantage is that the beacon receiver requires its own antenna and RF section.

The SiRFstarIIe can also be used with an external source of DGPS corrections that uses RTCM format messages. The SiRFstarIIe accepts RTCM type 1, 2, 3 and 9 messages through one of the serial ports. The default differential behavior is to accept RTCM messages through Port 2. In a custom application, it is also possible to input the differential corrections directly into the GPS Core using a Module Interface Routine `MI_SetDgpsCorrs()`. Usage of this method is only recommended for advanced users of GPS.

To change the default source of differential corrections, you must modify the value of `DefaultCorrectionType` in `dgpstype.h`. The possible values are:

- **`COR_NONE`**
  No differential values accepted.
- **`COR_WAAS`**
  Uses one GPS channel to track a WAAS satellite and provide corrections.
- **`COR_SERIAL`**
  External source providing RTCM type corrections.
- **`COR_INTERNAL_BEACON`**
  Uses the internal beacon to track a MSK radio-beacon with another antenna.
- **`COR_SOFTWARE`**
  Corrections provided using a Module Interface routine in custom user application.

Example:

To change the default source of differential corrections to the internal beacon receiver, verify that:

```
#define DefaultCorrectionType COR_INTERNAL_BEACON
```

SiRF Binary Messages for Differential

There are a two SiRF binary messages that are of interest for differential operation of the SiRFstarIIe. These messages include a control message (`MID_DGPSSourceControl`) and a poll message (`MID_DGPSStatus`). Both are described in the following subsections. The HandleSetDGPS_Src( ) routine in UI_SIRF.C provides details about the MID_DGPSSourceControl message and the QueueDgps_Src( ) routine in UI_SIRF.C provides details on the MID_DGPSStatus message.

Set DGPS Source Control (MID 0x85)

Description

A control message defined with an ID name of `MID_DGPSSourceControl`. This input message is used to set the DGPS source.

Arguments

Note – Variable names do not reflect the actual variable names used in a routine.

```
UINT8  DGPSSource

UINT32  DGPSfrequency

UINT8  DGPSbitRate
```

- `DGPSSource` selects the desired source of DGPS corrections and are defined as:

  0 = No Source Selected

  1 = WAAS receiver channel

  2 = Serial port via RTCM messages

  3 = Internal Beacon Receiver

- `DGPSfrequency` is the desired beacon frequency of the internal beacon receiver. This field has no effect if beacon is not selected. The frequency can be set to a specified value or the auto-scan mode engaged by setting the frequency to 0.
- `DGPSbitRate` is the desired beacon bit rate. The valid values are: 25, 50, 100 and 200 bits per second. Setting the bit rate to zero enables an auto scan mode of all bit rates.

DGPS Status (MID 0x1B)

Description

A status message defined with an ID name of `MID_DGPSStatus`. This message is output if the DGPS correction source is switched from `COR_NONE` to another value.

Arguments

Note variable names do not reflect the actual variable names used in routine.

```
UINT8  DGPSSource
UINT32  DGPSfrequency
UINT16  DGPSbitRate
UINT8  DGPSstatusBits
INT32  signalMag
INT16  signalStrength
INT16  signalSNR
```

- `DGPSSource` reports the selected source of DGPS corrections and are defined as:

  0 = No Source Selected

  1 = Internal Beacon Receiver

  2 = WAAS receiver channel

  3 = Serial port via RTCM messages

  4 = Software provided corrections

- `DGPSfrequency` reports the current frequency of the internal beacon receiver.
- `DGPSbitRate` reports the current bit rate of the internal beacon receiver.

- `DGPSstatusBits` report status of the receiver as follows:

  Bit 0 - Internal Beacon locked

  Bit 1 - Internal Beacon auto frequency

  Bit 2 - Internal Beacon auto bit rate

  Bit 3 - WAAS tracking

  Bit 4 - RTCM received on serial port
- `signalMag` reports the raw signal count from DGPS receiver.
- `signalStrength` reports the signal strength in dBm from the internal beacon receiver.
- `signalSNR` reports the signal to noise figure of the internal beacon receiver.

Module Interface Routines for Differential

This section lists a number of Module Interface routines that are used regardless of the differential operation of the receiver. These routines are described in detail in Appendix C, “Module Interface Details. These functions can be implemented in custom user application code or called through the SiRF binary messages described in “SiRF Binary Messages for Differential” on page 11-2.

- WERR MI_GetDgps_Mode (MI_DGPS_MODE *pData);
- WERR MI_GetDgpsStationID (INT16 *pData);
- WERR MI_GetDgpsCorrAge (float *pAge);
- WERR MI_GetDgpsSpecialMsg (char buf[91]);
- WERR MI_GetDgpsAlm (UINT16 entry, DGPSAlmType *pData);
- WERR MI_GetDgpsStationPos (double ecefPos[3]);
- WERR MI_GetDgpsSrc (int *pData);
- WERR MI_SetDgpsSrc (int *pData);
- WERR MI_GetDgps_Mode (MI_DGPS_MODE *pData);
- WERR MI_SetDgps_Mode (MI_DGPS_MODE *pData);
- WERR MI_SetDgpsCorrs(INT16 SVID,double GPSTime,double PRC,double RRC,INT16 IOD);
- WERR MI_SetDgpsSrc (MI_DGPS_SRC *pData);
- WERR MI_SetSbasPrn(int prn);

Adding a New User Protocol 12

SiRF recommends using the existing protocols when making modifications to the serial interface. New input and output messages can then be generated using the methods in “Adding New Input/Output Messages” on page 8-6. The added advantage is to enable the use of the already defined messages in the testing and evaluation of the unit. In the case that a new user protocol is desired, a code framework has been provided to aid in development. This Protocol has the name USER1 and can be set as the default output on Port 1 using the `USER1` preprocessor definition.

Protocol Implementation

Before adding a new Protocol, it is important to understand how Protocols are used in the SDK code. Each Protocol has a similar set of functions which fully define its capabilities. To enable a specific protocol, it is necessary to redirect the generic function calls from the GPS Core to the specific function call of the desired Protocol. This redirection makes it possible to rigidly define the Module Interface to the GPS Core while allowing a broad range of modification on the Protocol side. The redirection is done in `UI_MSG.C`. A look at this file reveals that the Protocols are defined in `ProtocolCfg[]` as a series of structures, each of which has a set of unique (with maybe some default) functions. Figure 12-1 shows the functions which are required by each protocol and the concept of the Module Interface redirection.

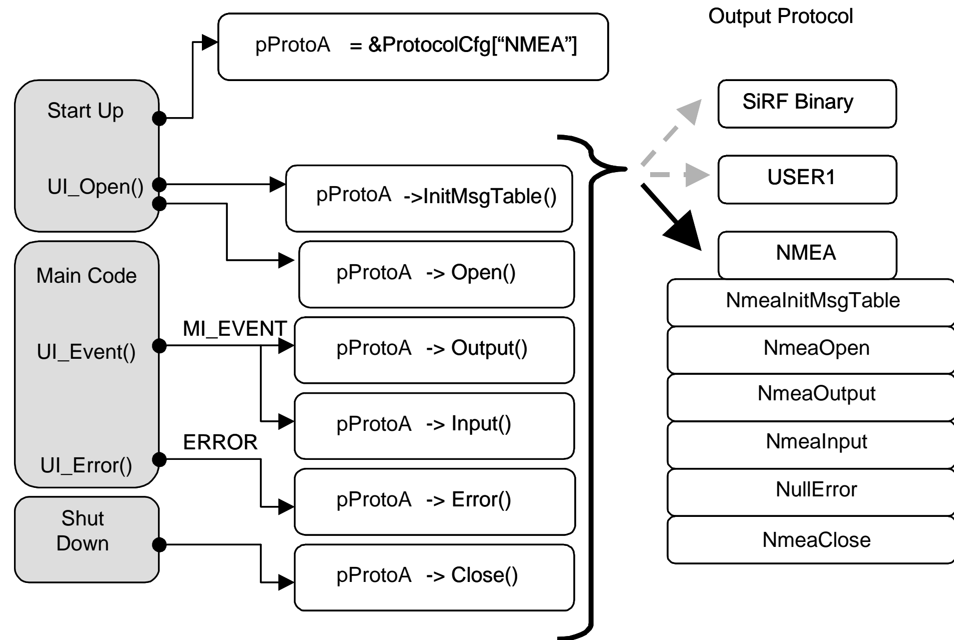

Figure 12-1 Protocol Redirection

The NMEA indicator in this case is the value of `SRAM.UI.ProtocolA`. For the case of the NMEA Protocol, the functions that actually end up getting called are shown as well. This is a fairly straightforward naming procedure and each protocol is similar. Note that the NMEA protocol does not have its own error function associated with it but instead uses a default function called `NullError`. An examination of the `ProtocolCfg[]` structure in `UI_MSG.C` reveals that several protocols make use of the NullError function. Also note that in the RTCM protocol, the `UI_Output` function is defaulted to `NullOutput`. This is possible because the RTCM protocol is not used to output information and there is no point in generating a unique output function. If you do not require the functionality, use the NULL default function. The actual purpose of each function shown in Figure 12-1 follows.

- **`InitMsgTable`**
  Used to initialize Protocol variables. Can be used to restore variables from battery-backed SRAM, for example, loading a message configuration table. This is called when the Protocol is first instantiated.

- **`UI_Open`**
  This function initializes the UART (Baud rate, parity, and bits), registers the protocol for that UART, registers message handlers to parse incoming messages and overrides some default protocol functions (put, send, deliver, and allocBuffer) if necessary.

- **UI_Output**
  Called every time an event is signaled by the GPS Core. When the event is signaled, the protocol can determine what action to take. As an example, for an MI_EV_NAV_COMPLETE (navigation complete) event, a position and time may be output. The event handler can retrieve required data using the MI_GetXXX() routines. See MI_IF.H for a list of these Module Interface routines. For more information on the implementation of output messages see "Adding a New SiRF Binary Output Message" on page 8-7.

- **UI_Input**
  Called just after UI_Output when an event has been signaled by the GPS Core. It is provided to enable processing of input message set commands in an ordered manner. When an input command is received to set some internal variable, MI_SetXXX() routines can be used to interact with the GPS Core and potentially the battery-backed area of SRAM. For more information on the implementation of input messages see "Adding a New SiRF Binary Input Message" on page 8-10.

- **UI_Error**
  This function is provided to handle exception conditions. For example, when UART buffers are exhausted. The protocol may decide whether to ignore or handle the error condition. Normal operation may result in calls to this function. For example, when a bad parity condition is detected in the 50 bps GPS navigation message. This may be due to normal signal blockage, but in an open sky environment may indicate problems with the RF front end.

- **UI_Close**
  This function closes the UART subsystem. This is called just prior to a module restart and enables the protocol to save status information or send out a termination message.

When a protocol is first initiated by a call to its UI_Open function, it makes a number of modifications to the way information is handled by the UART. The Protocol can be considered a part of a larger UART structure that is handling UART serial communication. Figure 12-2 shows a general overview of the UART structure. The UART structure contains information covering both interrupt processing and normal processing at the User Interface level. On the User Interface side, operation is driven by events from the GPS Core. On both sides, this UART structure covers buffer allocation and management. The ISR routines make use of some protocol specific information including the Baud rate, bits, parity and start/stop characters. The Rx and Tx ISR handlers are contained in ASICUART.C.

The Rx ISR recognizes single or double sequence termination characters and then deliver the whole buffer over to the application side. If your user protocol does not have defined termination characters then a different mechanism for delivery from the ISR layer to the application code is necessary. See "Single Character Delivery" on page 12-12 for an example of single character delivery to the application layer. The Protocol also defines a function (fpURcvr) that delivers the message to a receiver queue that can be accessed by the User Interface code. The default function that fpURcvr calls is umRcvrProtocol in UMANAGER.C.

On the User Interface side, a new Protocol can override the default send, put, deliver and allocBuffer functions. The default functions are umDefaultSend, umDefaultPut, umDefaultDeliver, and umDefaultAlloc defined in UMANAGER.C. The Send and Put functions are used for message output while the

`Deliver` function is for message input. The Protocol also can register message handlers. To see how the various protocols are initialized, see the associated `UI_Open` functions. Implementing a User Protocol is described in the next section.

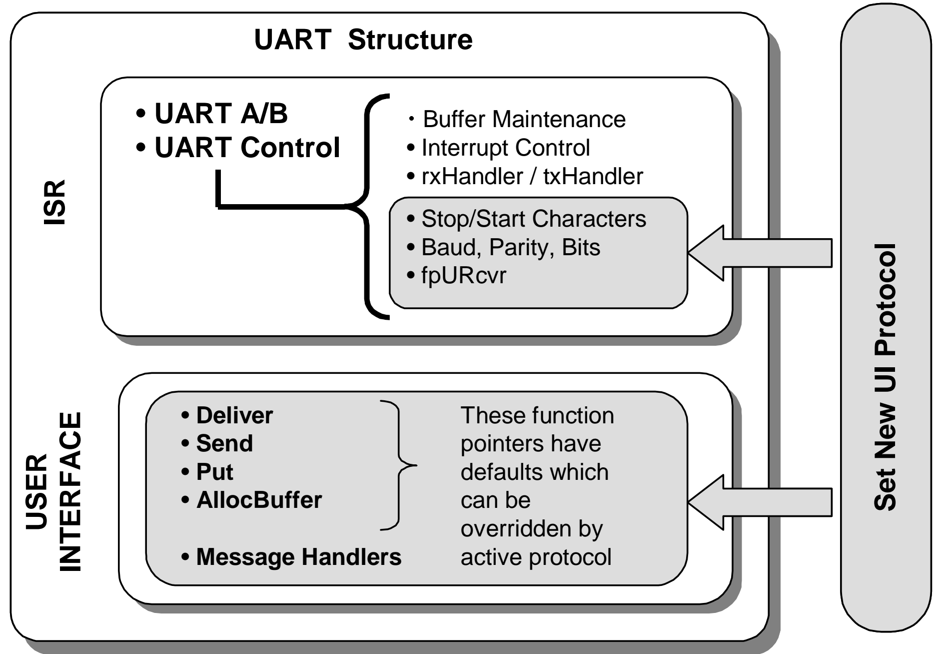

Figure 12-2 Overview of Uart Structure and Effect of Changing Protocol

USER1 Protocol

The SDK already contains the shell for a user protocol known as `USER1`. Setting the preprocessor definition USER1 as shown in "Basic Compile Switches" on page 4-5 defaults the Port 1 protocol to the User Protocol. Take a moment to run this build and review the output on a terminal program. The S2SDK must output the following string:

```
*****USER1_PROTOCOL4B
```

In `User1Open` the `send`, `allocBuffer` and `deliver` functions are redirected, but the `put` function is left as the default function (`umPut`). Figure 12-3 shows the output sequence for this message. The `4B` on the end of the output string is a 16 bit CRC added by the `User1Send` function in `User1Open` that overrides the default `send` function in `User1Open`. The `*****` preamble is added to the beginning of the output buffer when it is first allocated in `User1AllocBuf`. The `SEND_ITEM` call in `OutputUSER1()` is a macro that calls the User Protocol `put` function.

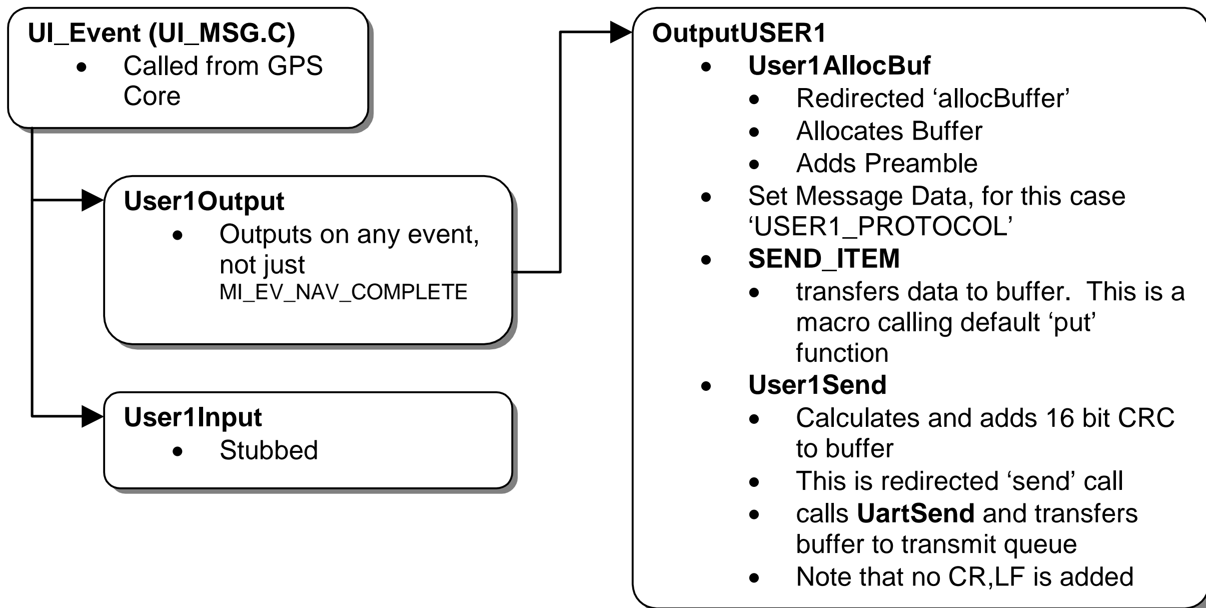

Figure 12-3 Output Sequence for USER1 Protocol

Example:

The following example implements a user protocol with two input messages and one output message. One of the messages is used to switch the Port over to SiRF Binary protocol, and the other is used to start/stop the GPS Core. This is similar to the second example given in "Start/Stop GPS Functions" on page 10-5. You must add a User task and event to trigger the I/O capability because no GPS events are generated when the GPS is off. The default setting for Port one is USER1 at 9600 Baud. A terminal program can be used to test this implementation.

Protocol Definition:

The Log (Output) Message format is contained inside a pair of brackets and followed by a carriage return, line feed. The line contains a 16 bit CRC that is calculated using all characters between (but excluding) the first open bracket and the asterisk (*). When GPS is turned off the GPS TOW and ECEF coordinate fields are zero.

```
(L[0..9], [Y..N], GPS TOW, ECEF X coord, ECEF Y coord, ECEF Z coord*[CRC1][CRC2] ) [CR][LF]
```

where:

L[0..9] is the message identifier, L for a log message with ten possible unique identifiers (0 to 9).

[Y..N] indicates if GPS is enabled, Y for Yes, N for No.

GPS TOW is the GPS Time of Week obtained from the GPS Core.

ECEF X coord is the Earth-Centered Earth Fixed X coordinate in meters to one decimal place.

ECEF X coord is the Earth-Centered Earth Fixed X coordinate in meters to one decimal place.

ECEF X coord is the Earth-Centered Earth Fixed X coordinate in meters to one decimal place.

***** is a end of data indicator.

[CRC1] is the MSB of the 16 bit CRC.

[CRC2] is the LSB of the 16 bit CRC.

[CR] is a carriage return.

[LF] is a line feed.

Example Output Message:

```
(L0,Y,512915.5,-2682820.5,-4307710.1,3850638.1*B3)
```

The Command (Input) message is much simpler and does not contain a CRC (although the user can add one). This implementation is simple and only looks at the message identifier and not any contents. The format allows for ten possible unique message identifiers.

```
(C[0..9] ) [CR][LF]
```

Example Input Messages:

(C0) Switch the Port to SiRF Binary protocol at 19200 Baud.

(C1) Toggle the GPS state between off and on.

Code Changes:

To run this example, the following preprocessor definitions must be defined, USER1 and TASK_PERIOD=200 (see “Basic Compile Switches” on page 4-5). As shown below, you must add a user event and the User task to trigger the event when GPS is off.

```
MI_IF.H

typedef enum
{
   MI_EV_NONE                 = 0,
   MI_EV_MEASUREMENT_RCVD     = 1,
   …
   …
   …
   MI_EV_NL_INIT_DONE         = 2048
   ,MI_EV_USER                 = 4096 /* add user event */
} MI_EVENT;
```

UI_NMEA.C

```
/* remove static from Float2Ascii prototype and function definition*/
void Float2Ascii (float FValue, INT16 right, char* pOutString);
```

UI_USER1.C

```
/* need some other include files for extra functions */
#include <stdlib.h>
#include <string.h>
...
...
...
/* add two message handlers which will be registered in User1Open */
static int  HandleSetSiRFProtocol(UMHandle hMsg, UINT8 ID, UINT8 *pMsg, int Len);
static int  HandleToggleGPS      (UMHandle hMsg, UINT8 ID, UINT8 *pMsg, int Len);
...
...
...
/*Add more functionality to handle new user input messages.  Now we can check the */
/* contents and pass the MID on to MsgDeliver which will call our registered Handler */
static int User1Deliver (PrivateHandle hPriv, UINT8 *pMsg, int MsgLen, UINT8 MsgId)
{
   UMHandle hMsg;
   hMsg = (UMHandle) hPriv;
   switch (MsgId)
   {
   case MID_LookInMessage:
      {
      /* check header */
      if (!strncmp( (CHAR *)pMsg,"(C",2))
      {
      /* Get the MID of the current message, currently we want 0 or 1 */
       CHAR cID = atoi((CHAR *)(pMsg+2));
      return msgDeliver (&hMsg->handlers, (UINT8) cID, pMsg, MsgLen);
      }
    }
      break;
   case MID_BufferFull:
   case MID_ParityError:
   case MID_RcvFullError:
   case MID_RcvOverrunError:
   case MID_FrameError:
```

```
    return msgDeliver (&hMsg->handlers, MsgId, pMsg, MsgLen);
    default:
    break:
    }
    /* default comes here */
    return msgDeliver (&hMsg->handlers, MID_TransportDataError, pMsg, MsgLen);
}


/* We have to modify this function to include umSerialRxCheck to check the  */
/* buffers which are filled by the UART interrupt service routines!!         */
/* Note that we will check either on a Navigation event, or, if GPS is not   */
/* active we can use the user generated event                                */

WERR User1Input (MI_EVENT Event, UINT32 TimeOutput)
{
    if (((Event & MI_EV_NAV_COMPLETE)!= MI_EV_NAV_COMPLETE &&
((Event & MI_EV_USER) != MI_EV_USER))
    {
        return SUCCESS;
    }

    umSerialRxCheck();  /* check for new input msgs */

    return SUCCESS;
}


/* Again, we have to modify the Output handler so that we can trigger */
/* on a user event as well as a Navigation complete event.            */
WERR User1Output (MI_EVENT Event, UINT32 TimeOutput)
{

    if (((Event & MI_EV_NAV_COMPLETE)==_MI_EV_NAV_COMPLETE) ||
((Event & MI_EV_USER) == MI_EV_USER))
        QueueSample();

    return SUCCESS;
}


static WERR QueueSample (void)
{
    UMBufHandle hBuf;
    static ECEF  ecef;
    MI_GPS_TIME timegps;
    CHAR buf[50];

    hBuf = User1AllocBuf (hComm, 0);
    if (!hBuf)
        return FAILURE;
```

```
    /* If GPS is stopped, output no position */
    if (LPQueryGPSStopped())
    {
       strcpy(buf,"0,N,0,0,0,0");
       umPuts(hBuf, buf);
    }
    /* If GPS is active, get the time and ECEF position and send them out! */
    else
    {
        CHAR TimeBuff[15], XBuff[15], YBuff[15], ZBuff[15];

       /* Note the use of these Module Interface routines to obtain data from */
       /* the GPS Core.  Check MI_IF.H for prototype details                  */
       MI_GetGpsTime (&timegps);
       MI_GetPosEcef (&ecef);

       /* Note that this function is in UI_NMEA.H and we must declare */
       /* as an external       */
       Float2Ascii(timegps.Tow,1,TimeBuff);
       /* note that Float2Ascii will not handle more than one */
       /* decimal point here */
       Float2Ascii(ecef.X,1,XBuff);
       Float2Ascii(ecef.Y,1,YBuff);
       Float2Ascii(ecef.Z,1,ZBuff);

       sprintf(buf,"0,Y,%s,%s,%s,%s", TimeBuff, XBuff, YBuff, ZBuff);
       /* Use the default Put string function to transfer to buffer */
       umPuts(hBuf, buf);
    }

    /* Send the whole message, commit to transmit queue */
    return (hComm->send (hComm, bufHandle));
}  /* OutputUSER1()*/
...
...
...
/* We must register the start/stop characters, and the new message handlers. */
/* Note that we also override the default deliver, send, and allocbuffer     */
/* commands here but not the put command                                     */

WERR User1Open (UARTs Port, UMHandle hMsg)
{
    umSetSerialHandle (Port, hMsg);  /* for SerialRxCheck [umanager.c]          */
    NAVSetSerialDebugFlag (0);      /* debug flag OFF for USER1, NO PRINTF()'s */
    hComm = hMsg;

    /* structure copy UI_SRAM USER1 info into ProtocolCfg[UI_PROTO_USER1].comm */
    UI_GetProtocolCfg(UI_PROTO_USER1)comm = UI_SRAM.USER1comm;

    hMsg->device = uartInit (hMsg->nUART,
              UI_GetProtocolCfg(UI_PROTO_USER1)comm.bits,
              UI_GetProtocolCfg(UI_PROTO_USER1)comm.baud,
              UI_GetProtocolCfg(UI_PROTO_USER1)comm.stop,
              UI_GetProtocolCfg(UI_PROTO_USER1)comm.parity);
```

```
    /* Register the protocol start/stop characters */
    uartRegisterProtocol (hMsg->device, umRcvrProtocol, '(', CR, LF, hMsg);

    /* Register the default error message handlers */
    msgHandlersInit (&hMsg->handlers,
           (MessageHandler) USER1NoOpHandler,
           (MessageHandler) USER1ErrorHandler);

    /* set up the USER1 protocol handlers*/
/*hMsg->put =*/
    hMsg->deliver     = User1Deliver;
    hMsg->send        = User1Send;
    hMsg->allocBuffer = User1AllocBuf;

    /* Register for the INPUT Messages to be handled*/
/* umRegisterForMessage (hMsg, MID....,xxxx handler routine);*/
    umRegisterForMessage(hMsg,0, HandleSetSiRFProtocol);
    umRegisterForMessage(hMsg,1, HandleToggleGPS);

    return SUCCESS;
}
...
...
...
/* Use the AllocBuffer function to add our preamble before */
/* adding the new message.  Note that the "L" is for a     */
/* log output message.  It is not necessary to do this     */
/* here but it saves some work                             */

static UMBufHandle User1AllocBuf (UMHandle hMsg, INT16 MsgLen)
{
    UmBufHandle hBuf;
    hBuf = umMakeUBuf (hMsg);
    if (hBuf)    /* put in the USER1 preamble message*/
    {
        *hBuf->pBuffer++ = '(';
        *hBuf->pBuffer++ = 'L';
        hBuf->bytesInBuffer = 2;
    }
    return hBuf;
}
...
...
...
/* The User1Send function is used to tack on the last bit of our       */
/* transport layer.  We want to add a 16 bit Checksum, a close bracket */
/* and a carriage return line feed.                                    */

static int User1Send (UMHandle hMsg, UMBufHandle hBuf)
{
    UINT8          *pMsg;
    UINT16         crc;
if (!hMsg)
{
```

```
return 0;
}
   pMsg = hBuf->pIBuffer+hBuf->bytesInBuffer;
   crc = CompleteCRC16 (hBuf->pIBuffer + 1,
             hBuf->bytesInBuffer - 1);
   sprintf ((char *) pMsg, "*%02X", crc);  /* fill in the CRC */
   hBuf->bytesInBuffer += 3;

   /* add CR, LF to end */

   pMsg = hBuf->pIBuffer+hBuf->bytesInBuffer;
   *pMsg++ = ')';
   *pMsg++ = CR;
   *pMsg = LF;
   hBuf->bytesInBuffer += 3;

   return uartSend (hMsg->device, hBuf);
}


/* We must add our two new message handlers !! */
static int  HandleSetSiRFProtocol(UMHandle hMsg, UINT8 ID, UINT8 *pMsg, int Len)
{
   UARTParams comm;  /* comm params: baud,bits,stop,parity,pad0 */
   int        protocol;

   comm.baud   = 19200;
   comm.bits   = SERIAL_BITS;
   comm.stop   = SERIAL_STOP;
   comm.parity = SERIAL_PARITY;

   protocol = 0; /* SiRF binary */

   MI_SetUi_Proto (UI_PROTO_USER1, protocol, TRUE, FALSE);
   MI_SetComm (protocol, (void *) &comm, FALSE);
   NAVForceReset(PROTOCOL_CHANGE,"Set SiRF Protocol");

   return 1; /* we received one message */
}


static int  HandleToggleGPS      (UMHandle hMsg, UINT8 ID, UINT8 *pMsg, int Len)
{
   if (LPQueryGPSStopped())
      LPUserStartGPS();
   else
      LPUserStopGPS();
   return 1;
}
```

USER.C

```
/* This is our user task function which will execute every 200 ms */
void UI_UserTaskFunction(void)
{
   /* execute once a second */
   /* We have to execute our own event every second */
   /* to keep the I/O going */
   static int Count200ms=0;
   if (Count200ms++ == 5)
   {
      Count200ms = 0;
      if (LPQueryGPSStopped())
      {
         /* toggle LED to show we are still alive*/
         ASIC_GPIO_PORTDIR2 |= 0x20;
         ASIC_GPIO_PORTVAL2 ^= 0x20;
         UI_Event(MI_EV_USER, 0);
      }
   }
   return;
}
```

Single Character Delivery

In some cases it might be necessary to use a protocol that cannot be handled based on termination characters. There are two options in this case, you can modify the ISR routines in `ASICUART.C` to correctly fill the buffer and then pass it to the user interface code as a complete message using `fpURcvr` and the standard `Deliver` function, or you can use the ISR to just pass each character individually and do the parsing and buffering in the user interface code. A good start for this is the RTCM protocol. This protocol bypasses most of the ISR buffering and message validation and instead uses character by character transmission to the application layer.

Example:

With the receiver set up in the factory default mode, SiRF binary on Port 1 and RTCM input on Port 2, echo characters received through the RTCM port as a debug output on the SiRF Binary Port.

RTCMMGR.C

```
static void RcvrProtocol (void *pCtxt, UARTBuffer *pUB, UARTProtocolReason Byte)
{
   UI_RTCM_BUF *pCur;
   pCur = &RtcmBuf[BufSwitch];

   umDebugPrintf("echo : %d",Byte);

   if (pCur->Cnt < RTCM_BUF_SIZE)
   {
      pCur -> Buf[pCur->Cnt] = (UINT8) Byte;
      pCur -> Cnt++;
      return;
   }
      pCur->Cnt = 0;
}
```

GPIO Lines, Throughput and Wait States 13

This chapter describes how to set the alternate functions for the GPIO lines and provides an overview of CPU clock considerations, wait state settings, and throughput measurements. See the *System Development Kit User's Guide Part 2 - GSP2e Chip* for more information.

GPIO Lines

The GSP2e 100 pin TQFP has 13 GPIO lines, while the GSP2e 144 pin LQFP has 46. The actual number of available GPIO lines for user development depends on the hardware layout of the board. Most GPIO pins are multi-functional and are often used for other purposes. Examples include chip selects, debugging support pins, microwire interface and external interrupts. Some of the GPIO lines are also used for power control during Low Power operation and may not be available. For the GSP2e 144 pin LQFP, selecting the 32 bit bus mode automatically sets the alternate functions of GPIO[35:20] to ED[31:16]. See the *System Development Kit User's Guide Part 2 - GSP2e Chip* for GPIO functions and default settings. To enable the alternate function(s) of a GPIO, the `ASIC_GPIO_SEL(0x80010100)` register must be modified. For information on how the S2SDK LEDs are controlled using GPIO lines (see "S2SDK LED Activation" on page 6-12).

Note – One of the first actions in `ARMSTART.S` is to initialize the `GPIO_SEL` register (0x80010100). GPIO_SEL is also referred to as ASIC_GPIO_SEL. For the Multi-Ice JTAG and the Icebreaker Macrocell in the GSP2e to function properly, the lowest bit (bit 0) in the GPIO Select register must be set high (alternate function) in the software. If this is not done, the JTAG device will have difficulty connecting to the board.

The value of the GPIO select register is also set in `MAIN.C` using the `ASIC_SetGPIO_Select` routine. User modifications to the GPIO select register can be added here.

Example:

At startup, in file MAIN.C a statement was added to configure the alternate functions of some of the General Purpose Input/Output (GPIO) lines as follows:

```
/* set the default GPIO_SEL register */
   gpioBits =
        GS_ICE_BREAKER
//    | GS_SERIAL_AGC
#ifdef BEACON
      | GS_uWIRE
#endif
//    | GS_GRF_ACTIVE
#ifndef PPS_OFF
      | GS_1PPS
#endif
/*       | GS_EXT_INT0 */
/*       | GS_EXT_INT1 */
/*       | GS_EXT_INT2 */
      | GS_CS1
      | GS_CS2  /* This may be overridden by UC_EstablishConfig below */
/*       | GS_CS3 */
/*       | GS_CS4 */
/*       | GS_CS5 */
/*       | GS_CS6 */
/*       | GS_CS7 */
       ;
ASIC_SetGPIOSelect(GS_GRF_ACTIVE);
ASIC_SetGPIOSelect(gpioBits);
```

You can comment out or remove the comment to enable and disable functions as necessary. The GS_ICE_BREAKER define is used to enable the DBGEN pin so that the JTAG functions.

Table 13-1 lists the GPIO lines and their alternate functions for the GSP2e. The Alternate Enable Bit is the bit in the ASIC_GPIO_SEL that must be set high to enable the alternate function. When alternate input is not selected, the default input is specified in the Default Alternate Input Value column. On most GPIO lines, either a pull-up or pull-down resistor is built into the chip. The last entry details the timer synchronization function. Some GPIOs (0, 3, 5, 13, 9, 14 and 15) can be used as a timer input by setting the value of the GPIO Timer synchronization register in 0x80010148. You must add a DEFINE value for this register location since there is not one currently. When a logic pulse is received through a pin set for timer synchronization, the GSP2e latches the segment counter into the Segment Count Latch Register 0x800d1246. This value can then be read by the CPU at a later time.

Table 13-1 GPIOs and Alternate Functions on the GSP2e

No.	Alternate Enable Bit	GSP2e-7400 100 Pin	GSP2e-7401 100 Ball	GSP2e-7410 144 Pin	GSP2e-7411 144 Ball	Pin/Ball Name	Default Alternate Input Value	Internal Pull-up/ Pull-down	Notes
0	0	49	J3	67	A4	DBGRQ/ GPIO0/ TREQA	0	Pull-down	1
1	0	48	K1	66	B4	DBGEN/ GPIO1/ TREQB	0	Pull-down	1
2	0	N/A	N/A	114	N4	BREAKPT/ GPIO2/ TACK	0	Pull-down	1
3	1	78	J9	115	P4	GPIO3	--	Pull-down	
4	1	76	K11	117	N5	GPIO4	--	None	
5	2	44	H3	62	C6	SI/GPIO5	0	Pull-down	
6	2	43	H4	61	B6	SO/GPIO6	--	Pull-down	
7	2	45	J2	63	D5	SK/GPIO7	0	Pull-down	
8	3	47	G5	65	B5	PWRCTL/ GPIO8	--	None	
9	4	42	J1	60	A6	TIMEMARK/ GPIO9	--	Pull-down	
10	5	46	H5	64	A5	EIT0_N/ GPIO10	1	Pull-up	
11	6	N/A	N/A	75	D3	EIT1_N/ GPIO11	1	Pull-up	
12	7	N/A	N/A	109	P1	EIT2_N/ GPIO12	1	Pull-up	
13	8	54	J4	81	E1	CS1_N/ GPIO13	--	Pull-up	
14	9	53	J5	80	E2	CS2_N/ GPIO14	--	Pull-up	
15	10	52	M4	79	E3	CS3_N/ GPIO15	--	Pull-up	
16	11	N/A	N/A	77	D1	CS4_N/ GPIO16	--	Pull-up	
17	12	N/A	N/A	3	M13	CS5_N/ GPIO17	--	Pull-up	
18	13	N/A	N/A	2	L12	CS6_N/ GPIO18	--	Pull-up	

Table 13-1 GPIOs and Alternate Functions on the GSP2e *(Continued)*

No.	Alternate Enable Bit	GSP2e-7400 100 Pin	GSP2e-7401 100 Ball	GSP2e-7410 144 Pin	GSP2e-7411 144 Ball	Pin/Ball Name	Default Alternate Input Value	Internal Pull-up/ Pull-down	Notes
19	14	N/A	N/A	1	M12	CS7_N/ GPIO19	--	Pull-up	
20	ED width	N/A	N/A	72	B2	ED[16] /GPIO20		Pull-up	
21	ED width	N/A	N/A	70	B3	ED[17]/ GPIO21		Pull-up	
22	ED width	N/A	N/A	43	C10	ED[18]/ GPIO22		Pull-up	
23	ED width	N/A	N/A	41	A11	ED[19]/ GPIO23		Pull-up	
24	ED width	N/A	N/A	38	A12	ED[20]/ GPIO24		Pull-up	
25	ED width	N/A	N/A	36	C14	ED[21]/ GPIO25		Pull-up	
26	ED width	N/A	N/A	34	C12	ED[22]/ GPIO26		Pull-up	
27	ED width	N/A	N/A	32	D13	ED[23]/ GPIO27		Pull-up	
28	ED width	N/A	N/A	71	A3	ED[24]/ GPIO28		Pull-up	
29	ED width	N/A	N/A	68	C5	ED[25]/ GPIO29		Pull-up	
30	ED width	N/A	N/A	42	C11	ED[26]/ GPIO30		Pull-up	
31	ED width	N/A	N/A	39	B12	ED[27]/GPIO31		Pull-up	
32	ED width	N/A	N/A	37	B13	ED[28]/ GPIO32		Pull-up	
33	ED width	N/A	N/A	35	C13	ED[29]/GPIO33		Pull-up	
34	ED width	N/A	N/A	33	D14	ED[30]/ GPIO34		Pull-up	
35	ED width	N/A	N/A	30	E13	ED[31]/ GPIO35		Pull-up	
36	15	N/A	N/A	144	P13	GPIO36		Pull-down	
37	15	N/A	N/A	143	N13	GPIO37		Pull-down	
38	15	N/A	N/A	142	N12	GPIO38		Pull-down	
39	15	N/A	N/A	141	P12	GPIO39		Pull-down	
40	15	N/A	N/A	112	N3	GPIO40		Pull-down	

Table 13-1 GPIOs and Alternate Functions on the GSP2e *(Continued)*

No.	Alternate Enable Bit	GSP2e-7400 100 Pin	GSP2e-7401 100 Ball	GSP2e-7410 144 Pin	GSP2e-7411 144 Ball	Pin/Ball Name	Default Alternate Input Value	Internal Pull-up/ Pull-down	Notes
41	15	N/A	N/A	111	P3	GPIO41		Pull-down	
42	15	N/A	N/A	110	N2	GPIO42		Pull-down	
43	15	N/A	N/A	108	N1	GPIO43		Pull-down	
44	15	N/A	N/A	107	M1	GPIO44		Pull-down	
45	15	N/A	N/A	106	M2	GPIO45		Pull-down	
--						TimerSynch	0		2

1 The TIC interface uses DBGRQ, DBGEN and BREAKPT as I/O. The TIC interface must use the pins when io_TMODE is low and io_TMBIST is high. The GPIO uses the ICE BREAKER interface as the alternate function for these pins. The TIC interface is MUXed separately.

2 The TimerSynch input is a second alternate function for the following GPIOs: 0, 3, 5, 13, 9, 14 and 15. The default value for this input is 0. If the second alternate function has to be used for a particular GPIO, that GPIO must be configured as a GPIO (i.e., not alternate function) in input mode. The GPIO for the second alternate function can be selected from the GPIO_TimerSynch register.

Chip Select Wait States

This section gives information on the access time and wait states required for the various memory configurations based on CPU Clock speed. The wait states are set in the Chip select registers. Some of these registers (CSN0, CSN1 and CSN2) are initialized in `CLKADJ.S`. If you want to use other chip selects, you must add this code and verify that the memory selected can be accessed using the wait state setting.

Access time for external memory may be decreased through use of the cache. Table 13-2 shows the access times and wait states for external memory speeds and CPU clock speeds. Wait states can be selected from 0 to 7 and the wait state value is normally the expected access time in cycles, decreased by two. Wait states to access internal memory (CACHE, battery-backed RAM) are handled automatically by internally generated handshake signals. Wait states for external memory are programmed in the BIU chip select registers.

Table 13-2 Clocks and Wait States to Access External Memory, with and without Cache Enabled

	GPS 49.1 MHz	**ACQ 38.19 MHz**	**GPS/2 24.55 MHz**	**ACQ/2 19.09 MHz**	**GPS/4 12.28 MHz**	**ACQ/4 9.5 MHz**
ROM + CACHE ON + HIT	1 clk	1 clk	1 clk	1 clk	1 clk	1 clk
ROM + CACHE ON + MISS	See for memory type below	See for memory type below	See for memory type below	See for memory type below	See for memory type below	See for memory type below
15 ns memory	2 clks 0 WS	2 clks 0 WS	2 clks 0 WS	2 clks 0 WS	2 clks 0 WS	2 clks 0 WS
70 ns memory	5 clks 3 WS	4 clks 2 WS	3 clks 1 WS	2 clks 0 WS	2 clks 0 WS	2 clks 0 WS
90 ns memory	6 clks 4 WS	5 clks 3 WS	4 clks 2 WS	3 clks 1 WS	2 clks 0 WS	2 clks 0 WS
120 ns memory	7 clks 5 WS	6 clks 4 WS	5 clks 3 WS	3 clks 1 WS	2 clks 0 WS	2 clks 0 WS

The above numbers indicate the number of clocks for a 16-bit access on a 16-bit bus. For other configurations, the number of clocks needs to be adjusted by a multiplier as given in the following table. For combination of ROM + CACHE ON + MISS, the pipeline has to be refilled with four 32-bit words. That is 4x number of clocks for selected 32 bit memory, 8x number of clocks for 16 bit memory and 16x number of clocks for 8 bit memory. Table 13-3 shows the Multiplier for number of clocks required for memory access as given in Table 13-2.

Table 13-3 Multiplier for Number of Clocks Required for Memory Access

Compiler Configuration	**Bus Mode**		
	8 bit	**16 bit**	**32 bit**
ARM (32 bit)	x4	x2	x1
Thumb (16 bit)	x2	x1	x1

Converting UTC Time to GPS Week Number and TOW

This appendix provides an example of converting UTC time to GPS week number and TOW.

```
#include <stdio.h>

typedef long int INT32;
typedef int      INT16;

typedef unsigned long int UINT32;
typedef unsigned int      UINT16;

typedef unsigned char     BYTE;

typedef struct
{
   BYTE  hh;
   BYTE  mm;
   BYTE  ss;
} TIME;

typedef struct
{
   BYTE   day;
   BYTE   month;
   UINT16 year;
} DATE;


typedef struct
{
   UINT32    timeOfWeek;   /* GPS Time of Week, in seconds */
   UINT16    weekno;       /* Week number */
} GPS_TOW;

#define SECONDS_IN_WEEK       604800   /* # of seconds in a week   */
#define SECONDS_IN_DAY         86400   /* # of seconds in a day    */
#define SECONDS_IN_HOUR         3600   /* # of seconds in an hour  */
#define SECONDS_IN_MINUTE60    /* # of seconds in a minute */
```

```
/*=================================================================*/
/* UTCtoTOW                                                        */
/*   input:                                                        */
/*      _TIME, _DATE contain current time and date                 */
/*   output                                                        */
/*      GPS_TOW contains week number and TOW                       */
/*-----------------------------------------------------------------*/

void UTCtoTOW(TIME time, DATE date, GPS_TOW *gtm)
{
   UINT16 years, days, leap_days, total_days, left_days;

   UINT16 days_in_month[] = {0,0,31,59,90,120,151,181,212,
                            243,273,304,334 };
                            /*  +  31, 28,31, 30, 31, 30, 31, 31, 30, 31, 30 */

   /* UTC --> GPS time conversion.
      PS time started on Sunday, January 6th 1980
   */
   years = date.year - 1980;             /* years since 1980 */
   days  = days_in_month[date.month];    /* days since 1st January */
   days += date.day;

   /* after feb in the leap year */
   if ( (date.year % 4 == 0) && (date.month > 2) )
   days += 1;                        /* add the february 29th */

   /* # of leap days up to December 31 of the previous year */
   leap_days   = ((years-1) / 4) + 1;
                                  /* 1980 was a leap year  */
   total_days  = (UINT16)(years * 365 + days + leap_days - 6);
   gtm->weekno =  total_days / 7;
   left_days   =  total_days % 7;

   gtm->timeOfWeek = (UINT32)(((double)(left_days) * SECONDS_IN_DAY)
             + ((double)time.hh  * SECONDS_IN_HOUR)
             + ((double)time.mm  * SECONDS_IN_MINUTE)
             +  (double)time.ss
             +  ((double)13.0));
             /* 13 == UTC offset from 1st Jan 1999 */

   if(gtm->timeOfWeek >= SECONDS_IN_WEEK)
   {
   gtm->weekno += 1;
   gtm->timeOfWeek -= SECONDS_IN_WEEK;
   }

} /* UTCtoTOW */
```

```
void main()
{
   TIME    tm;
   DATE    dt;
   GPS_TOW gtm;

   tm.hh = 22;
   tm.mm = 13;
   tm.ss = 45;

   dt.year = 1999;
   dt.month= 5;
   dt.day  = 10;

   UTCtoTOW(tm,dt,&gtm);   /* convert time/date to gps time */

   /* for use with the SiRF binary MID_NavigationInitialization
   command, multiply the TOW(seconds) * 100 before sending command
   */

   printf("week num:%d\n", gtm.weekno);
   printf("TOW     :%lu" , gtm.timeOfWeek);

}  /* main */
```

SiRF Binary Messaging Functions B

SiRF Binary Messages

SiRF has defined a series of input and output messages for the SiRFstar series receivers. Each SiRF binary message has a unique Message Identifier (MID), defined as a value from 0 to 255. To verify compatibility between user defined messages and future SiRF development, certain ranges of numbers have been set aside for user development. Table B-1 shows the reserved user MID values.

Table B-1 Reserved User MID Values

Output MID Values	Input MID Values
0x61 to 0x7F	0xB4 to 0xC7

Except as noted, the receiver module responds with an Ack/Nak message to indicate that an input message was accepted or rejected. All SiRF binary message structures are defined in `UI_IF.H` and `UI_SIRF.C`, messages are sent in `UI_SIRF.C` and received through registered handlers for a specific message type in `UI_SIRF.C`

Some of the following structures list defaults for some values. To be certain of a particular element default value, refer to the header files.

Table B-2 lists the current SiRF reserved input and output MID values. These are defined in the code in `PROTOCOL.H`.

Table B-2 SiRF Defined MID Values

Output Messages			**Input Messages**		
Dec.	**Hex**	**Enum**	**Dec.**	**Hex**	**Enum**
0	0x00	MID_LookInMessage	128	0x80	MID_NavigationInitialization
1	0x01	MID_TrueNavigation	129	0x81	MID_SetNMEAMode
2	0x02	MID_MeasuredNavigation	130	0x82	MID_SetAlmanac
3	0x03	MID_TrueTracker	131	0x83	MID_FormattedDump
4	0x04	MID_MeasuredTracker	132	0x84	MID_PollSWVersion
5	0x05	MID_RawTrkData	133	0x85	MID_DGPSSourceControl
6	0x06	MID_SWVersion			

Table B-2 SiRF Defined MID Values *(Continued)*

Output Messages			Input Messages		
Dec.	**Hex**	**Enum**	**Dec.**	**Hex**	**Enum**
7	0x07	MID_ClockStatus	134	0x86	MID_SetSerialPort
8	0x08	MID_50BPS	135	0x87	MID_SetProtocol
9	0x09	MID_ThrPut	136	0x88	MID_SET_NAV_MODE
10	0x0A	MID_Error	137	0x89	MID_SET_DOP_MODE
11	0x0B	MID_Ack	138	0x8A	MID_SET_DGPS_MODE
12	0x0C	MID_Nak	139	0x8B	MID_SET_ELEV_MASK
13	0x0D	MID_VisList	140	0x8C	MID_SET_POWER_MASK
14	0x0E	MID_Almanac	141	0x8D	MID_SET_EDITING_RES
15	0x0F	MID_Ephemeris	142	0x8E	MID_SET_SS_DETECTOR
16	0x10	MID_TestModeData	143	0x8F	MID_SET_STAT_NAV
17	0x11	MID_RawDGPS	144	0x90	MID_PollClockStatus
18	0x12	MID_OkToSend	145	0x91	MID_SetDGPSPort
19	0x13	MID_RxMgrParams	146	0x92	MID_PollAlmanac
20	0x14	MID_TestModeData2	147	0x93	MID_PollEphemeris
21	0x15	MID_NetAssistReq	148	0x94	MID_FlashUpdate
22	0x16	MID_StopOutput	149	0x95	MID_SetEphemeris
23	0x17	MID_CompactTracker	150	0x96	MID_SwitchOpMode
24	0x18	MID_DRCritSave	151	0x97	MID_LowPower
25	0x19	MID_DRStatus	152	0x98	MID_PollRxMgrParams
26	0x1A	MID_DRHiRateNav	153	0x99	MID_TOWSync
27	0x1B	MID_DGPSStatus	154	0x9A	MID_PollTOWSync
28	0x1C	MID_NL_MeasData	155	0x9B	MID_EnableTOWSyncInterrupt
29	0x1D	MID_NL_DGPSData	156	0x9C	MID_TOWSyncPulseResult
30	0x1E	MID_NL_SVStateData	157	0x9D	MID_DRSetup
31	0x1F	MID_NL_InitData	158	0x9E	MID_DRData
32	0x20	MID_MeasureData	159	0x9F	MID_DRCritLoad
33	0x21	MID_NavData	160	0xA0	MID_HeadSync0
34	0x22	MID_WaasData	161	0xA1	Free Space
35	0x23	MID_TrkComplete	162	0xA2	MID_HeadSync1
36	0x24	MID_TrkRollover	163	0xA3	MID_E911HSPingRef
37	0x25	MID_TrkInit	164	0xA4	MID_E911RefPingHS
38	0x26	MID_TrkCommand	165	0xA5	MID_ChangeUartChnl
39	0x27	MID_TrkReset	166	0xA6	MID_SetMsgRate
40	0x28	MID_TrkDownload	167	0xA7	MID_LPAcqParams
41	0x29	MID_GeodeticNav	168	0xA8	MID_POLL_CMD_PARAM

Table B-2 SiRF Defined MID Values *(Continued)*

42	0x2A	MID_TrkPPS	169	0xA9	MID_SetDatum
43	0x2B	MID_CMD_PARAM	170	0xAA	MID_SetSbasPrn
44	0x2C	MID_LLA			
45	0x2D	MID_TrkADCOdoGPIO			
46	0x2E	MID_TestModeData3			

Functions for Input Messages

Functions #1

static int HandleFormattedDump(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetNavInit(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetNmeaProto(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetDgpsComm(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetSirfComm(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetUiProto(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetNavModeMask(UMHandle *hMsg*,UINT8 MsgId,
UINT8**pMsg*,int *MsgLen*);

static int HandleSetDopMask(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetDgpsMode(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetDgpsSrc(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetElevMask(UMHandle hMsg,UINT8 MsgId,UINT8 *pMsg,int MsgLen);

static int HandleSetPwrMask(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetStaticNav(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetAlm(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetEph(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetOpMode(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetLowPwr(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetLPAcqParam(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetUartChnls(UMHandle *hMsg*,UINT8 MsgId,
UINT8 **pMsg*,int *MsgLen*);

static int HandleSetMsgCtrl(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandleSetSbasPrn(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

Return Value

1 if the message is correctly handled else 0.

Parameters

hMsg	Message handle.
MsgId	Message ID.
pMsg	Points to the message buffer.
MsgLen	Message length.

Remarks

These functions are mapped through umRegisterForMessage() in SirfOpen() to handle the input SiRF-binary messages that are used for setting parameters within the receiver. The function parses the input message and signals the UI-event handler that the message was received.

Functions #2

static int HandlePollSwVersion(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandlePollClkStatus(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandlePollAlm(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandlePollEph(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandlePollRcvrParam(UMHandle *hMsg*,UINT8 MsgId,UINT8 **pMsg*,int *MsgLen*);

static int HandlePollCmdParam (UMHandle hMsg, UINT8 MsgId, UINT8 *pMsg, int MsgLen);

Return Value

1 if the message is correctly handled else 0.

Parameters

hMsg	Message handle.
MsgId	Message ID.

pMsg Points to the message buffer.

MsgLen Message length.

Remarks

These functions are mapped through umRegisterForMessage() in SiRFOpen() to handle the input SiRF-binary messages that are used for polling output SiRF-binary messages. The function parses the input message and queues the desired output message.

Functions #3

static WERR SetDgpsComm (void);

static WERR SetSirfComm (void);

static WERR SetNmeaProto (void);

static WERR SetUiProto (void);

static WERR SetNavModeMask (void);

static WERR SetDopMask (void);

static WERR SetDopMode (void);

static WERR SetDgpsSrc (void);

static WERR SetElevMask (void);

static WERR SetPwrMask (void);

static WERR SetStaticNav (void);

static WERR SetAlm (void);

static WERR SetOpMode (void);

static WERR SetLowPwr (void);

static WERR SetLpAcqParam (void);

static WERR SetUartChnls (void);

static WERR SetSbasPrn (void);

Return Value

SUCCESS (0) if the new settings are accepted else FAILURE (-1).

Remarks

Each function reflects an input SiRF-binary message that is used for setting parameters within the receiver. It is called periodically through the navigation cycle, but it does nothing unless the UI-event for the message is signaled. If the event is signaled, it de-signals the event and tries to set the new parameters. If the new settings are accepted, it pushes an acknowledgment of acceptance (ACK) else it pushes an acknowledgment of rejection (NAK).

Functions for Output Messages

Functions #1

```
static WERR QueueStartup (void);
static WERR QueueMeasNav (void);
static WERR QueueMeasTrk (void);
static WERR QueueInitTrk (void);
static WERR QueueRawTrk (void);
static WERR QueueSwVersion (void);
static WERR QueueClkStatus (void);
static WERR Queue50bps (void);
static WERR QueueThruPut (void);
static WERR QueueVisList (void);
static WERR QueueAlm (void);
static WERR QueueTestMode1 (void);
static WERR QueueTestMode2 (void);
static WERR QueueTestMode3 (void);
static WERR QueueRtcm (void);
static WERR QueueOkToSend (void);
static WERR QueueRcvrParam (void);
static WERR QueueDgpsSrc (void);
static WERR QueueNlMeas (void);
static WERR QueueNlDgps (void);
static WERR QueueNlSvState (void);
static WERR QueueNlInit (void);
static WERR QueueCmdParam( UINT8 MsgId);
```

Return Value

SUCCESS (0) if the message is queued correctly else FAILURE (-1).

Remarks

Each function reflects an output SiRF-binary message, with the exception of MID_Error, MID_Ack, and MID_Nak, which is being queued into the output buffers. It builds the message and allocates enough buffer space for the message. Then, it puts the message fields into the buffer(s) and queues the buffer(s) into the output buffers.

Function #2

WERR UI_PushOkToSend(BOOL OkToSend);

Return Value

SUCCESS (0) if the message is pushed correctly else FAILURE (-1).

Parameters

OkToSend if TRUE accept input message else do not accept input message.

Remarks

This function reflects the MID_OkToSend SiRF-binary message that is being pushed onto the output buffers. It builds the message and allocates enough buffer space for the message. Then, it puts the message fields into the buffer(s) and pushes the buffer(s) onto the output buffers.

Function #3

static WERR QueueEph(UINT8 *SvId*);

Return Value

SUCCESS (0) if the message is queued correctly else FAILURE (-1).

Parameters

SvId	Satellite ID of ephemeris data that is being queued.

Remarks

The function reflects the MID_PollEphemeris SiRF-binary message that is being queued into the output buffers. It builds the message and allocates enough buffer space for the message. Then, it puts the message fields into the buffer(s) and queues the buffer(s) into the output buffers.

Function #4

static WERR QueueError(UINT16 *ErrId*, UINT16 *Cnt*, UINT32 **pParams*);

Return Value

SUCCESS (0) if the message is queued correctly else FAILURE (-1).

Parameters

ErrId	Identification of the error.
Cnt	Number of parameters of the error.
pParams	Points to the parameter list of the error.

Remarks

This function is used to send an error through a SiRF-binary message.

Functions #5

static WERR PushAck(UINT8 *MsgId*);

static WERR PushNak(UINT8 *MsgId*);

Return Value

SUCCESS (0) if the message is queued correctly else FAILURE (-1).

Parameters

MsgId	ID of message being ACK'ed or NAK'ed.

Remarks

These functions are used to push an acknowledgement to accept(ACK) or reject(NAK) the reception of a MsgId message.

Module Interface Details

An overview of the Module Interface is provided in “Module Interface Overview” on page 1-5. This appendix describes the Module Interface events and the data structures used by the various Module Interface routines when communicating with the GPS Core.

Module Interface Events

The Module Interface events are signaled by the GPS Core and used to control system I/O. The events are defined in mi_if.h in an enumerated type called MI_EVENT. Descriptions of the events are listed below.

Event	Description
MI_EV_NONE	Null event (never used).
MI_EV_MEASUREMENT_RCVD	Measurements have been read from the tracker subroutines.
MI_EV_NAV_COMPLETE	Navigation routines are complete, a new navigation solution might be available.
MI_EV_INITIAL_POSITION	Event signaled after the initialization of position, time, and clock prior to the initialization of the navigation information (MI_EV_NL_INIT_DONE event).
MI_EV_NEW_VISIBLE_LIST	A new satellite visible list has been computed.
MI_EV_NEW_50BPS	At least one channel has new 50 bps subframe data available.
MI_EV_NEW_ALMANAC	New GPS almanac data is available.
MI_EV_NEW_EPHEMERIS	New GPS ephemeris data is available.
MI_EV_INITIAL_ACQ_COMPLETE	Initial acquisition is complete, at least one channel is, or has been tracking since startup.
MI_EV_KRAUSE_COMPLETE	Not used for SiRFstarIIe.

Event	Description
MI_EV_WAIT_INITIAL_ACQ	Occurs once a second until initial acquisition is complete. This event is generated each time a new SV is searched for during initial acquisition. The time varies depending on the search range given to the tracker. For example, a warm start uses a narrow range to acquire, and those searches are relatively fast. If that fails, the search is expanded which causes the time between events to increase. If necessary, you may call MI_GetTrkData to determine the ID of the satellite that is currently being searched for.
MI_EV_MEASUPDATE	Measurements have been received by the one-second navigation update routine and are ready to be processed.
MI_EV_NL_INIT_DONE	Event signaled after the internal navigation filter has initialized all navigation associated variables.

Module Interface Routines

This section summarizes the Module Interface routines that can be used in custom development to interact with the GPS Core. Also included are Module Interface Utility routines which might be useful. The following list is alphabetical, but certain MI_Set functions have been grouped with the corresponding MI_Get function if it exists.

GetCOG

Description

Function to return the Course Over Ground for a given horizontal velocity vector. The return value is an azimuth (direction of travel) given in degrees (0 to 360) going clockwise from true North.

Prototype

```
double GetCOG (VNED vned, char *cogBuf);
```

Arguments

```
typedef struct
{
    DOUBLE  Vn;   /* North Velocity component */
    DOUBLE  Ve;   /* East Velocity component  */
    DOUBLE  Vd;   /* Down Velocity component */
}
VNED;
```

Parameter	Description
cogBuf	Pointer to a buffer of characters where the ascii value of the azimuth is stored with two decimal places. Useful for ascii output.

Returns

The value of the azimuth (0 to 360) going clockwise from North.

GetDate

Description

Function to convert the GPS TOW (time of week) and year number into UTC day, month, and year. This function is used in conjunction with the FindDay and FindMonth functions. This function can also be used in conjunction with the ConvertTowtoUTC function which returns the UTC hour, minutes and seconds into the current UTC day.

Prototype

```
void GetDate(int WkNum,double Tow,int *pYr,int *pMth,int *pDay);
```

Arguments

Parameter	Description
WkNum	Extended GPS week number (i.e., full number of weeks since 01/06/80 with no 1024 GPS rollover). Obtained from the navigation message plus 1024.
TOW	GPS time of week from the navigation message.
pYr	Pointer for the return value of the current year.
pMth	Pointer for the return value of the current month.
pDay	Pointer for the return value of the current day.

MI_GetDatum

Description

Outputs the latitude, longitude and altitude for the current Datum selected in SRAM.UI.datum. The default datum is WGS84. If the current datum is WGS84, then the ECEF position is converted to LLA. If a different datum is selected, the ECEF coordinates are adjusted to fit the new datum and then the LLA values are calculated based on the new ellipsoid. New datums can be entered in this function, see "MI_SetDatum" on page C-11 for more details. Altitude in this case is reference to the ellipsoid. Note that this function is mathematically intensive and must be used sparingly.

Prototype

```
WERR MI_GetDatum (LTP *ltp);
```

Arguments

```
typedef struct_
{
    DOUBLE  Lat;
    DOUBLE  Lon;
    DOUBLE  Alt;    _/* should change to Ht (above ellipsoid) */
}
LTP; /* Local Tangential Plan */
```

ConvertTowtoUTC

Description

Function to convert GPS TOW (time of week) into UTC hours, minutes and seconds. This does not apply the leap second offset. You must use the MI_GetUTC function to get the true UTC time after it has been corrected by the UTC parameters in the GPS

navigation message. GPS time of week is the number of seconds that have elapsed since Saturday midnight at Greenwich plus a number of leap seconds. GPS time has been continuous since 01/06/80 and does not include the leap second adjustments that are added to UTC time to match the earth's orbit.

Prototype

```
void ConvertTowtoUTC(double TOW, int *pHr, int *pMin,double
*pSec);
```

Arguments

Parameter	Description
Tow	GPS TOW (time of week) in seconds from GPS navigation message.
pHr	Pointer for return value of UTC hours in day.
pMin	Pointer for return value of UTC minutes in hour.
pSec	Pointer for return value of UTC seconds in minute.

ConvertECEFtoLTP
ConvertLTPtoECEF

Description

Functions to transform the WGS84 cartesian ECEF coordinates to WGS84 latitude, longitude and height. To get latitude, longitude and height for a different datum see "MI_GetDatum" on page C-4. To convert from LLA to ECEF, the function uses a modified Carlson (iterative) method to obtain geodetic coordinates from ECEF coordinates in three iterations.

Prototypes

```
void ConvertECEFtoLTP (ECEF *pecef, LTP *pltp);
void ConvertLTPtoECEF (LTP *plla, ECEF *pecef);
```

Arguments

```
typedef struct
{
    DOUBLE  X;
    DOUBLE  Y;
    DOUBLE  Z;
}
```

```
ECEF; /* Earth Centered Earth Fixed */

typedef struct
{
    DOUBLE  Lat;
    DOUBLE  Lon;
    DOUBLE  Alt;/* should change to Ht (above ellipsoid) */
}
LTP; /* Local Tangential Plane */
```

MI_Get50Bps

Description

Function outputs the current GPS navigation subframe for a given channel. The output includes the channel number, SV ID and 300 bits of subframe data. GPS subframes are concluded every 6 seconds. A full frame takes 30 seconds to decode and contains the ephemeris data necessary to compute the satellite position and get the satellite clock offset. It takes 25 frames (12.5 minutes) to obtain a complete almanac. If there is no satellite being tracked or no subframe data available, the function will output NULL to pData.

Prototype

```
WERR MI_Get50BPS (MI_50BPS *pData, int Chnl);
```

Returns

SUCCESS (0) if 50 BPS data can be retrieved else FAILURE (-1)

Arguments

```
typedef struct _MI_50BPS
{
   UINT8  Chnl;
   UINT8  SVID;
   UINT32 Word[10];
} MI_50BPS;
```

Parameter	Description
Chnl	The channel number from which to get the subframe information.

MI_GetAlm

MI_SetAlm

Description

Functions to get/set the current GPS satellite almanac information.

Prototypes

```
WERR MI_GetAlm (MI_ALM *pAlm);
WERR MI_SetAlm (MI_ALM *pAlm);
```

Returns (MI_SetAlm)

SUCCESS (0) if the settings are accepted else FAILURE (-1).

Returns (MI_GetAlm)

SUCCESS (0) if almanac can be retrieved else FAILURE (-1).

Notes

MI_GetAlm

Retrieve entire almanac for all 32 satellites, 448 INT16 values; 448 arrived at as defined constants:

```
ALMANAC_SIZE = CMAX_SVID_CNT * ALMANAC_ENTRY
```

Almanac information for satellites that are not available are zero-filled.

MI_SetAlm

Set entire Almanac for all 32 satellites, 448 INT16 values; 448 arrived at as defined constants:

```
ALMANAC_SIZE = CMAX_SVID_CNT * ALMANAC_ENTRY
```

Almanac information for satellites that are not healthy are zero-filled. No validity checks are done on the almanac data. You must reset the receiver immediately after setting the Almanac, this can be done by using the NAVForceReset() function. In the SiRF protocol, this is accomplished by sending the MID_NavigationInitialization command following a MID_SetAlmanac command that then causes a reset.

UI_GetCPUClkRate

Description

Function to get the current CPU clock rate. This function views the BCLK_SEL register and the BCLK_DIV register to get the current clock source and clock divider. The function checks for GPS, ACQ and external clock sources. It bases its output on the following DEFINE clock values for these three clock sources as shown in Table C-1. The definitions are in UI_MSG.C.

Table C-1 DEFINE Oscillator Values for the Various Clock Sources

Oscillator Source	**Frequency (Hz)**
GPS Clock	49107000L
ACQ Clock	38194000L
External Clock (S2AR)	25000000L
External Clock (SDK)	25000000L
External Clock (Unknown board)	25000000L

Prototype

```
WERR UI_GetCpuClkRate (int *pClkRate);
```

Returns

SUCCESS (0) if CPU clock rate can be retrieved else FAILURE (-1).

Arguments

Parameter	**Description**
pClkRate	The default clock frequency given by the clock select register (0x80010014) divided by the clock divider value obtained from the clock divider register (0x8001002c).

MI_GetClkBias

Description

Function returns the absolute difference between GPS and local internal time. The difference is returned in units of nanoseconds.

Prototype

```
WERR MI_GetClkBias (INT32 *pData);
```

Returns

SUCCESS (0) if clock bias can be retrieved else FAILURE (-1).

Arguments

Parameter	Description
pData	Pointer for value of current offset in nanoseconds.

MI_GetClkDrift

Description

Function returns the current value of the oscillator clock drift in Hz. The default value is 96000 Hz. After initial start-up and a position has been generated, the actual clock drift is stored in battery-backed RAM for use every time the receiver is powered on.

Prototype

```
WERR MI_GetClkDrift (INT32 *pData)
```

Returns

SUCCESS (0) if clock drift can be retrieved else FAILURE (-1).

Arguments

Parameter	Description
pData	Pointer for value of current drift in Hz.

MI_GetDgpsSrc

MI_SetDgpsSrc

Description

Module Interface functions provided to set/get the differential correction source.

Prototypes

```
WERR MI_GetDgpsSrc(int *pData);
WERR MI_SetDgpsSrc(MI_DGPS_SRC *pData)
```

Arguments

```
typedef struct _MI_DGPS_SRC
{
   UINT8 Src;
   UINT32 Freq;
   UINT8 BitRate;
} MI_DGPS_SRC;
```

Return Value

The MI_SetDgpsSrc _function returns 1 on FAILURE due to illegal correctionType and a 0 on SUCCESS. The `MI_GetDgpsSrc` _function returns SUCCESS (0) if DGPS correction source can be retrieved else FAILURE (-1) if GPS is not running.

Parameter	Description

pData	Pointer for value of the differential source. This can take the values of the following enum:
	`typedef enum CorrectionTypes`
	`{`
	`COR_NONE = 0, /* Use no corrections */`
	`COR_WAAS = 1, /* Use WAAS channel */`
	`COR_SERIAL = 2, /* Use external receiver */`
	`COR_INTERNAL_BEACON = 3, /* Use internal DGPS Beacon */`
	`COR_SOFTWARE = 4 /* Corrections via MI_SetDgpsCorrs*/`
	`} CorrectionTypes;`

Notes

1. The function returns non-zero for undefined values of CorrectionTypes.
2. Calls to `MI_SetDgpsCorrs` override the set correction type and corrections provided through the function call are set as the exclusive source.
3. The default correction type can be changed by editing the DefaultCorrectionType defined in `DGPSTYPE.H`.

MI_GetDatum

MI_SetDatum

Description

Functions to get/set the current datum. The current Datum is used whenever the MI_GetDatum() function is called. This function takes the current WGS84 ECEF position, performs a transformation to get the ECEF coordinates in the new Datum and then calculates the latitude, longitude and height for the new Datum and ellipsoid. Currently, only five Datums are supported as shown in Table C-2.

Table C-2 Currently Supported Datums

Datum	Reference Number
WGS84	21
TOKYO_MEAN	178
TOKYO_JAPAN	179
TOKYO_KOREA	180
TOKYO_OKINAWA	181

Prototypes

```
WERR MI_GetDatum (UINT8 *pData);
WERR MI_SetDatum (UINT8 pData);
```

Arguments

The argument is either a pointer to a UINT8 value that is to be filled with the number of the current Datum (as shown in Table C-2) or the new value of a desired Datum.

Return Value

The MI_SetDatum function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetDatum function returns SUCCESS (0) if the datum can be retrieved else FAILURE (-1)if GPS is not running.

MI_GetDgps_Mode

MI_SetDgps_Mode

Description

Functions to get/set the current DGPS Mode. The user can select whether to ignore differential corrections entirely, use differential corrections when they are available, or navigate only when differential corrections are available.

Prototypes

```
WERR MI_GetDgps_Mode (MI_DGPS_MODE *pData);
WERR MI_SetDgps_Mode (MI_DGPS_MODE *pData);
```

Arguments

```
typedef struct _MI_DGPS_MODE
{
   UINT8 Mode;
   UINT8 Timeout;
} MI_DGPS_MODE;
```

Parameter	Description
Mode	0 = auto (use differential corrections if available). 1 = exclusive (only navigate if differential corrections are available). 2 = never (never use differential corrections).
Timeout	Maximum age of differential corrections. A value of 0 has no timeout and the corrections are used indefinitely.

Return Value

The MI_SetDgps_Mode function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetDgps_Mode function returns SUCCESS (0) if DGPS mode can be retrieved else FAILURE (-1).

MI_GetDgpsAlm

Description

Module Interface function to get an almanac entry. This function also returns the number of available entries.

Prototype

```
WERR MI_GetDgpsAlm(UINT16 entry, DGPSAlmType *pData)
```

Arguments

UINT16 entry: The number of the almanac entry to be retrieved.

DGPSAlmType *alm: The structure into which the almanac information is written. This structure has the form:

```
typedef struct
{
```

```
   INT16  lat;          /* degs, scale factor 0.002747 deg */
   INT16  lon;          /* degs, scale factor 0.005493 deg */
   UINT16 range;        /* in km scale 0 to 1023 km */
   UINT16 freq;         /* 100Hz 0=190kHz 0xfff=599.5kHz */
   UINT16 stationId;    /* 0 to 1023 */
   UINT8 health;       /* 0=normal */
                        /* 1=no integrity monitoring */
                        /* 2=No info available */
                        /* 3=Don't use this beacon */
   UINT8 bitRateMode; /* Bits 0-2 */
                        /* 0=25, 1=50, 2=100, 3=110, 4=150, 5=200, 6=250, 7=300*/
                        /* bit 4= modulation 0= MSK, 1= FSK */
                        /* bit 5= SYNC type 0= Async, 1= sync */
                        /* bit 6= broadcast coding, 0= no coding, 1= FEC coding*/
} DGPSAlmType;
```

Return Value

The MI_GetDgpsAlm function returns SUCCESS (0) if DGPS almanac can be retrieved else FAILURE (-1).

Notes

There is no check to see that the entered argument value for entry is within the number of currently available almanac entries. If this is the case, the provided structure are not modified.

MI_GetDgpsCorrAge

Description

Function to get the age of the current set of differential corrections.

Prototype

```
WERR MI_GetDgpsCorrAge (float *pAge);
```

Returns

SUCCESS (0) if DGPS correction age can be retrieved else FAILURE (-1).

Arguments

Parameter	Description
pAge	Pointer for value of differential correction age.

MI_GetDgpsBeacon
MI_SetDgpsBeacon

Description

This Module Interface function is provided to get/set the current parameters of the internal beacon receiver. The information is passed in or out into the structure provided. This includes the beacon frequency and the bit rate.

Prototype

```
WERR MI_GetDgpsBeacon(DGPSI_PARAMS *pData)
WERR MI_SetDgpsBeacon (INT32 Freq, INT8 BitRate)
```

Arguments (MI_GetDgpsBeacon)

```
typedef struct
{
   BOOL16 AutoFrequency;  /* TRUE if there is frequency auto scan mode, FALSE otherwise. */
   BOOL16 AutoBitRate;    /* TRUE if there is bit rate auto scan mode, FALSE otherwise.  */
   INT32  ReceiverFreqHz; /* Receiver frequency in Hz. */
   INT16  ReceiverBitRateBPS; /* Receiver bit rate in bps. */
   BOOL16 SignalValidity; /* returns TRUE if a valid DGPS signal is being received,
        FALSE otherwise  */
   INT32  SignalMagnitude; /* Returns a 32-bit integer corresponding to the signal level  */
   INT16  SignalStrength_dB; /* returns signal strength in dB uV/m (calibrated for
        CSI H-field beacon antennas, for both CSI and SiRF RF sections).*/
   INT16  SNR_dB; /* Currently, returns 0.  but in later versions will return an SNR estimate */
   FLOAT  Prf;    /* returns the percentage of correctly processed words.  This is the
        main indicator of the signal reception quality. */
} DGPSI_PARAMS;
```

Arguments (MI_SetDgpsBeacon)

Parameter	Description
Freq	Frequency you want to set (in Hz).
BitRate	Bit rate you want to set (in bps).

Return Value

The MI_GetDgpsBeacon function returns FAILURE (-1) if the frequency is smaller than the minimum beacon frequency or greater than the maximum beacon frequency OR if the bitrate is different than 25, 50, 100, or 200. Otherwise, the frequency and bitrate are used to set the internal DGPS beacon receiver and SUCCESS (0) is returned.

Note

If you want to auto scan a parameter, pass zero value for it.

MI_GetDgpsSpecialMsg

Description

Module Interface function that retrieves the last message sent in an RTCM type 16 input message. The RTCM type 16 was designed to allow for the encapsulation of a text message inside the RTCM protocol. The function enters a null terminated string into the user provided buffer buf. The buffer must be at least 91 character long to provide for the longest possible returned string.

Prototype

```
WERR MI_GetDgpsSpecialMsg(char buf[91])
```

Arguments

A buffer to be filled by the contents of the last type 16 message. The largest message size is 90 bytes according to the RTCM specification. The buffer will be null terminated.

Return Value

SUCCESS (0) if the settings are accepted else FAILURE (-1).

MI_GetDgpsStationID

Description

Function to obtain the ID of the last/current station that is broadcasting the current differential corrections.

Prototype

```
WERR MI_GetDgpsStationID (INT16 *pData);
```

Arguments

Parameter	Description
pData	Pointer for ID of differential station broadcasting current or last used corrections.

Return Value

SUCCESS (0) if the DGPS station ID can be retrieved else FAILURE (-1).

MI_GetDgpsStationPos

Description

Function to obtain the location of the differential station that is broadcasting the currently used (or last used) set of GPS corrections.

Prototype

```
WERR MI_GetDgpsStationPos (double ecefPos[3]);
```

Arguments

Parameter	Description
ecefPos[3]	A vector of three doubles that contain the WGS84 ECEF position of the current differential correction station.

Return Value

SUCCESS (0)

MI_GetDopMask

MI_SetDopMask

Description

Functions to get/set the Dilution of Precision (DOP) mask for the position output. The DOP is a geometric figure of merit based on the direction cosine matrix.

Prototypes

```
WERR MI_GetDopMask (MI_DOP_MASK *pData);
WERR MI_SetDopMask (MI_DOP_MASK *pData);
```

Arguments

```
typedef struct _MI_DOP_MASK
{
   UINT8 Mode;
   UINT8 GDOP_Th;
   UINT8 PDOP_Th;
   UINT8 HDOP_Th;
} MI_DOP_MASK;
```

Parameter	Description
Mode	0 = auto PDOP/HDOP 1 = PDOP 2 = HDOP 3 = GDOP 4 = never
GDOP_Th	GDOP threshold
PDOP_Th	PDOP threshold
HDOP_Th	HDOP threshold

Returns

The MI_SetDopMask function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetDopMask function returns SUCCESS (0) if the DOP mask can be retrieved else FAILURE (-1).

MI_GetElevMask
MI_SetElevMask

Description

Functions to get/set the satellite elevation mask angle for either tracking or navigation. Satellites at a lower elevation than the corresponding mask angle are not tracked/included in position solution.

Prototypes

```
WERR MI_GetElevMask (MI_ELEV_MASK *pData);
WERR MI_SetElevMask (MI_ELEV_MASK *pData);
```

Arguments

```
typedef struct _MI_ELEV_MASK
{
   INT16 Trk;
   INT16 Nav;
} MI_ELEV_MASK;
```

Parameter	Description
Trk	Mask for tracking (in 1/10 degree).
Nav	Mask for navigating (in 1/10 degree).

Returns

The MI_SetElevMask function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetElevMask function returns SUCCESS (0) if the DOP mask can be retrieved else FAILURE (-1).

MI_GetEph
MI_SetEph

Description

Functions to get or set the available current satellite ephemeris data.

Prototypes

```
WERR MI_GetEph (MI_PACKED_EPH *pData);
WERR MI_SetEph (MI_PACKED_EPH *pData);
```

Arguments

```
typedef struct _MI_PACKED_EPH
{
   UINT16 subframe[3][15];
} MI_PACKED_EPH;
```

Parameter	Description
subframe	Subframe of ephemeris data.

Return Value

The MI_SetEph function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetEph function returns SUCCESS (0) if ephemeris data can be retrieved else FAILURE (-1).

Notes

Data consisting of 45 16-bit unsigned integers that make up 3 subframes of data with each consisting of 15 unsigned 16-bit integers. This data is the ephemeris subframe data collected from the 50 bps data stream, and compressed by packing each subframe from 10 subframe words (32 bits/word) into 15 words (16 bits/word) with the tlm and parity words stripped off. For data packing details, see `CSTD.C`.

MI_GetEstGPSTime

Description

Function to get the estimated GPS time of the measurements that are used in the current navigation solution. The result is given in milliseconds.

Prototype

```
WERR MI_GetEstGPSTime (UINT32 *pData);
```

Arguments

Parameter	Description
pData	Pointer for return value of GPS time of the last measurement set that was used in the navigation solution.

Return Value

The MI_GetEstGPSTime function returns SUCCESS (0) if the estimated GPS time can be retrieved else FAILURE (-1).

MI_GetLPAcqParams

MI_SetLPAcqParams

Description

The user may set or get the Low Power acquisition parameters through a call to `MI_SetLPAcqParams()` or `MI_GetLPAcqParams()`. See Chapter 9, “Low Power Operation” for more details.

Prototypes

```
WERR MI_GetLPAcqParams (MI_LP_ACQ_PARAM *pData);
WERR MI_SetLPAcqParams (MI_LP_ACQ_PARAM *pData);
```

Arguments

```
typedef struct _MI_LP_ACQ_PARAM
{
   UINT32 MaxAcqTime;  /* in milliseconds */
   UINT32 MaxOffTime;  /* in milliseconds */
} MI_LP_ACQ_PARAM;
```

Parameter	Description
MaxAcqTime	When TricklePower is enabled, *MaxAcqTime* (in ms) is the maximum allowable interval from the start of a TricklePower cycle to the time a valid position fix is obtained from navigation. If this time elapses and no fix is obtained, the receiver is deactivated for up to *MaxOffTime*, and when the receiver reactivates, a hot start is commanded. The integer must be in multiples of 1000 ms. The smallest allowable value is 1000 ms. There is no upper limit.
MaxOffTime	The longest period (in ms) for which the receiver will deactivate due to the *MaxAcqTime* timeout. The actual deactivated period may be less if the user-specified duty cycle (*OnTime* / *LpInterval*) can be maintained. It must be a positive number. The smallest allowable value is 1000 ms. The largest allowable value is 1800000 ms (i.e., 1800 seconds, or 30 minutes.)

Return Value

`WERR` is a SiRF type defined in stdtype.h; it has the values `SUCCESS` or `FAILURE`. The function returns `FAILURE` only if TricklePower is not supported by the board on which the software is running. The `MI_GetLPAcqParams` function returns SUCCESS (0) if the low power parameters can be retrieved else `FAILURE` (-1).

MI_GetLowPower

MI_SetLowPower

Description

Function to get or set the Low Power modes of the receiver, including TricklePower and Push-to-Fix. See Chapter 10, “User Tasks, ASIC Interrupts, and the Scheduler” for more details.

Prototypes

```
WERR MI_GetLowPower (MI_LP_PARAM *pData);
WERR MI_SetLowPower (MI_LP_PARAM *pData);
```

Arguments

```
typedef struct _MI_LP_PARAM
{
   INT16 PushToFix;
   INT32 OnTime;            /* in milliseconds */
   INT32 LPInterval;        /* in milliseconds */
   BOOL  UserTasksEnabled;
   INT32 UserTaskInterval;  /* in milliseconds */
   BOOL  PwrCyclingEnabled;
} MI_LP_PARAM;
```

Parameter	Description
PushToFix	0 = Disable Push-to-Fix. 1 = Enable Push-to-Fix. (For Push-to-Fix to be enabled, `PwrCyclingEnabled` must be TRUE and `OnTime` = 200, `LPInterval` = 1000.)
OnTime	Must be a multiple of 100 (if not, it is rounded up to the nearest multiple of 100). `OnTime` must be greater than or equal to 300 ms.
LPInterval	Must be an integer value greater than or equal to 1000 (i.e., 1 second). `LPInterval` does not need to be a multiple of 100.
UserTasksEnabled	FALSE = Disable user tasks. TRUE = Enable user tasks. (User must provide a task function as described in Chapter 11, “DGPS Operation.”)
UserTaskInterval	Period at which user task is to be scheduled. You must verify that interval is large enough that the previous user task completes before the next one is scheduled.
PwrCyclingEnabled	TRUE = Enable TricklePower. FALSE = Disable TricklePower.

Return Value

`WERR` is a SiRF type defined in stdtype.h; it has the values `SUCCESS` or `FAILURE`. The function returns `FAILURE` if TricklePower is not supported by the board on which the software is running. It also returns `FAILURE` if the OnTime is <200 ms. The MI_GetLowPower function returns SUCCESS (0) if the low power parameters can be retrieved else FAILURE (-1).

Notes

1. To deactivate TricklePower, the parameter `PwrCyclingEnabled` must be set to `FALSE`. If `PwrCyclingEnabled` is `FALSE`, the parameters `OnTime` and `LPInterval` are ignored.
2. To activate TricklePower, set `PwrCyclingEnabled` to `TRUE`, and set `OnTime` and `LPInterval` to the desired OnPeriod and TricklePowerInterval, respectively. (These quantities are defined in "TricklePower" on page 9-1.) Both quantities must be integer values with units of milliseconds. The `PushToFix` parameter must be set to 0, (i.e., Push-to-Fix disabled, when TricklePower is TRUE).
3. If invalid parameters are supplied, they are ignored, and TricklePower operation is disabled. When enabling/disabling TricklePower, the User task settings can be obtained by a call to `UI_GetUserTaskParam()` which returns `UserTasksEnabled` and `UserTaskInterval`. To set these values and enable user tasks, see Chapter 10, "User Tasks, ASIC Interrupts, and the Scheduler." User tasks can be enabled or disabled independent of TricklePower or Push-to-Fix. Unless the user has supplied a user task as explained in that section, the parameter `UserTasksEnabled` must be set to `FALSE`. In this case, the `UserTaskInterval` parameter is ignored.

MI_GetNavDops

Description

Function to obtain the Dilution of Precision (DOP) values for the current navigation solution. The DOP values represent the geometric merit of the current satellites being tracked.

Prototype

```
WERR MI_GetNavDops (MI_DOPS *pData);
```

Arguments

```
typedef struct
{
   double GDOP;  /* Geometric  Dilution of Precision */
   double HDOP;  /* Horizontal Dilution of Precision */
   double PDOP;  /* Position   Dilution of Precision */
   double TDOP;  /* Time       Dilution of Precision */
   double VDOP;  /* Vertical   Dilution of Precision */
   BOOL   maskExceed;
} MI_DOPS;
```

Parameter	Description
maskExceed	High if the current DOPs output by the navigation solution are higher than the current DOP mask.

Returns

The MI_GetNavDops function returns SUCCESS (0) if the DOPs can be retrieved else FAILURE (-1).

MI_GetNavFom

Description

This function is somewhat misnamed. It is supposed to return a figure of merit but instead returns a value indicating that the current position fix has been validated or that it has timed out.

Prototype

```
WERR MI_GetNavFom (UINT8 *pData);
```

Arguments

Parameter	Description
pData	0x00 = Solution has not been validated. 0x02 = Solution has been validated. 0x04 = Solution has timed out for dead reckoning (degraded mode). 0x08 = Velocity is valid.

Returns

The MI_GetNavFom function returns SUCCESS (0) if the FOM can be retrieved else FAILURE (-1).

MI_GetNavInit

MI_SetNavInit

Description

Function to get/set the navigation initialization parameters. Used as a set function that restarts the receiver in one of the various reset modes (hot start, warm start, cold start, or factory start).

Prototypes

```
WERR MI_GetNavInit (MI_NAV_INIT *pData);
WERR MI_SetNavInit (MI_NAV_INIT *pData);
```

Arguments

```
typedef struct
{
   INT32  posX;
   INT32  posY;
   INT32  posZ;
   INT32  clkOffset;
   UINT32 timeOfWeek;
   UINT16 weekno;
   UINT8  chnlCnt;
   UINT8  resetCfg;
} MI_NAV_INIT;
```

Parameter	Description
posX	ECEF X position (in meters).
posY	ECEF Y position (in meters).
posZ	ECEF Z position (in meters).
clkOffset	Clock offset (in Hz).
timeOfWeek	GPS time of week (in sec.).
weekno	GPS week number.
chnlCnt	Number of channels.
resetCfg	Reset configuration bits. Bit 0: Valid initialization. Bit 1: Clear ephemeris flag. Bit 2: Clear memory flag. Bit 3: Factory reset. Bit 4: Enable debug output data for navigation library. Bit 5: Enable SiRF debug output data. Bit 6: Enable NMEA debug output data.

Returns

The MI_SetNavInit function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetNavInit function returns SUCCESS (0) if the NavInit can be retrieved else FAILURE (-1).

Notes

1. MI_GetNavInit retrieves current settings that the receiver uses upon reset in the preceding structure.

2. MI_SetNavInit resets parameters consist of the initial position, time, clock offset, channel count and start mode information. By modifying members of the structure outlined above and calling MI_SetNavInit(), software performance can be evaluated under predetermined conditions. Set the parameters that the receiver uses upon reset in the preceding structure. After executing this function, on the next 1-second navigation cycle, the module is reset with the new parameters.

MI_GetNavMode

Description

This function returns the mode information of the latest position fix. This includes the number of satellites used in solution, the type of solution and an altitude hold indicator. The type of solution can either be a least squares type (used at start-up) or a Kalman filter solution.

Prototype

```
WERR MI_GetNavMode (UINT8 *pData);
```

Arguments

The value of pData has the following form:

B	I	N	A	R	Y			Hex	Decimal	Description	SiRFdemo Comment
7	6	5	4	3	2	1	0			Nav Fix Type (Exclusive)	
					0	0	0	0X00	0	No Navigation	No Nav
					0	0	1	0x01	1	1SV Degraded Solution	1 SV
					0	1	0	0x02	2	2SV Degraded Solution	2 SV
					0	1	1	0x03	3	3SV Alt Fixed Solution	3 SV
					1	0	0	0x04	4	>=4 SV Full Solution	3D Fix
					1	0	1	0x05	5	Least Sqr 2D Fix	LstSq 2D
					1	1	0	0x06	6	Least Sqr 3D Fix	LstSq 3D
					1	1	1	0x07	7	DR Solution (0 SV)	DR
										TricklePower Bit (Anded)	
				1					+ 8	Trickle Power Position	
				0						Full Power Position	
										Altitude Hold Indicator (Anded)	
		0	0							No Altitude Hold	
		0	1						+ 16	Altitude Used From Filter	
		1	0						+ 32	Altitude Used From User	
		1	1						+ 48	Forced Altitude (From User)	
										Dop Mask Bit (Anded)	
	1								+ 64	Dop Mask Exceeded	Based on Dop Mask CTRL
	0									Dop <mask	
										DGPS Bit (Anded)	
1								0x08	+ 128	DGPS Position	
0										SPS Position	

Returns

The MI_GetNavMode function returns SUCCESS (0) if the navigation mode can be retrieved else FAILURE (-1).

MI_GetNavModeMask
MI_SetNavModeMask

Description

Functions to set or get control value for the navigation processing. Most of these values concern operation under degraded conditions.

Prototypes

```
WERR MI_GetNavModeMask (MI_NAV_MODE_MASK *pData);
WERR MI_SetNavModeMask (MI_NAV_MODE_MASK *pData);
```

Arguments

```
typedef struct
{
   UINT8 Enable3D;
   UINT8 EnableConAlt;
   UINT8 DegradedMode;
   UINT8 Pad;
   UINT8 EnableDR;
   INT16 AltInput;
   UINT8 AltMode;
   UINT8 AltSrc;
   UINT8 CoastTimeout;
   UINT8 DegradedTimeout;
   UINT8 DR_Timeout;
   UINT8 bTrkSmooth;
} MI_NAV_MODE_MASK;
```

Parameter	Description
AltInput	Fixed altitude value in altitude hold.
AltMode	Mode for altitude hold: 0 = auto 1 = always 2 = never
AltSrc	Altitude source for altitude hold: 0 = last 1 = fixed 2 = dynamic
DegradedTimeout	Timeout for degraded mode.
DegradedMode	0 = direction then time 1 = time then direction 2 = direction only 3 = time only 4 = disabled
DR_Timeout	Timeout for dead-reckoning mode.
EnableDR	Dead-reckoning mode: 0 = disable 1 = enable
Enable3D	0 = disable 1 = enable
bTrkSmooth	Track smooth algorithm: 0 = disable 1 = enable
EnableConAlt	Not used.
CoastTimeout	Not used.
Pad	Not used.

Returns

The MI_SetNavModeMask function returns SUCCESS (0) if the settings are accepted else FAILURE (-1). The MI_GetNavMask function returns SUCCESS (0) if the navigation mode mask can be retrieved else FAILURE (-1).

MI_GetNavList

Description

Function that returns a list of satellites currently used in the navigation solution. The only valid entries are from 0 to (SVIDCnt – 1).

Prototype

```
WERR MI_GetNavList (MI_NAV_LIST *pData);
```

Arguments

```
typedef struct
{
   UINT8 SVIDCnt;
   UINT8 aSVID[NUM_OF_CHANNELS];
} MI_NAV_LIST;
```

Returns

The MI_GetNavList function returns SUCCESS (0) if the SV ID list can be retrieved else FAILURE (-1).

MI_GetPosEcef

Description

Function to return the current (or last) WGS84 ECEF position solution. This is the most common form of output for the SiRFstarIIe. Since all calculations in the SiRFstarIIe are performed in WGS84, there is no extra computation needed to output these values, unlike solutions that must be presented in other datums. All the output values are expressed in meters.

Prototype

```
WERR MI_GetPosEcef (ECEF *pData);
```

Arguments

```
typedef struct
{
        DOUBLE  X;
        DOUBLE  Y;
        DOUBLE  Z;
}
ECEF;  /* in WGS84 */
```

Returns

The MI_GetPositionEcef function returns SUCCESS (0) if the position data can be retrieved else FAILURE (-1).

MI_GetPositionLTP

Description

This function returns the current position of the receiver expressed in latitude [radians], longitude [radians] and altitude [m]. The reference ellipsoid in this case is WGS84.

Prototype

```
WERR MI_GetPositionLTP (LTP *pData);
```

Arguments

typedef struct

```
{
    DOUBLE  Lat;
    DOUBLE  Lon;
    DOUBLE  Alt;/* should change this to Ht (above ellipsoid) */
}
LTP;
```

Returns

The MI_GetPositionLTD function returns SUCCESS (0) if the position data can be retrieved else FAILURE (-1).

MI_GetPwrMask

MI_SetPwrMask

Description

Function to get/set the current satellite power mask for tracking or navigation processing. Satellites with a power level below this value are not used.

Prototypes

```
WERR MI_GetPwrMask (MI_PWR_MASK *pData);
WERR MI_SetPwrMask (MI_PWR_MASK *pData);
```

Arguments

```
typedef struct _MI_PWR_MASK
{
   UINT8 Trk;
   UINT8 Nav;
} MI_PWR_MASK;
```

Parameter	Description
Trk	Mask for tracking (in dB-Hz).
Nav	Mask for navigating (in dB-Hz).

Returns

The MI_SetPwrMask function returns SUCCESS (0) if the settings are accepted else FAILURE (-1).

MI_GetRawTrkData

Description

Function is used to obtain raw tracking information for a given channel. This includes the raw measurement data. For information on how to interpret the measurement data, see the *SiRFstarIIe Evaluation Kit User's Guide*.

Prototype

```
WERR MI_GetRawTrkData (MI_RAW_TRK *pData, int Index);
```

Arguments

```
typedef struct
{
   INT32 Channel;            /* Channel number in tracking [0 to 11] */
   INT16 SVID;               /* Satellite id  [1 to 32] */
   INT16 State;          /* Status of the tracker channel, see Table D.3 (mask = 0x1FF) */
   INT32 BitNumber;          /* Bits at 50 bps = 20 ms */
   INT16 MsecNumber;         /* Represents time in units of msec */
   INT16 ChipNumber;         /* Represents time in units of CA chips */
   INT32 CodePhase;          /* Represents time in units of chips */
   INT32 CarrierDoppler;     /* Doppler frequency */
   INT32 MeasureTimetag;     /* Measurement time tag */
   INT32 DeltaCarrierPhase;  /* Current carrier phase */
   INT16 SearchCnt;          /* How many times to search for a SV */
   UBYTE aCNo[SAMPLES_SEC];  /* C/No in dB-Hz */
```

```
    BYTE  power_bad_count;    /* Count of Power in 20 ms below 31 dB-Hz */
    BYTE  phase_bad_count;    /* Count of Power in 20 ms below 31 dB-Hz */
    INT16 delta_car_interval; /* Count of ms contained in delta_carrier phase */
    INT16 correl_interval;    /* Correlation interval */
} MI_RAW_TRK;
```

Parameter	Description
Index	The channel for which the above data must be output.

Table C-3 shows the tracking state of the channel. Multiple bits can be set at once with 0xBF being the highest state.

Table C-3 Tracking State of Channel

DEFINE Value in Code	Bit Value	Description
ACQ_SUCCESS	0x0001	Set if acq/reacq is done successfully.
DELTA_CARPHASE_VALID	0x0002	Integrated carrier phase valid.
BIT_SYNC_DONE	0x0004	Bit sync completed flag.
SUBFRAME_SYNC_DONE	0x0008	Subframe sync is done.
CARRIER_PULLIN_DONE	0x0010	Carrier pull-in is done.
CODE_LOCKED	0x0020	Code locked.
ACQ_FAILED	0x0040	Failed to acquire SV.
GOT_EPHEMERIS	0x0080	Ephemeris data available.

Returns

The MI_GetRawTrkData function returns SUCCESS (0) if the raw track data can be retrieved else FAILURE (-1).

MI_GetSWVersion

Description

This function returns a string containing the current software version number.

Protoype

```
WERR MI_GetSWVersion (char *pData);
```

Arguments

Parameter	Description
pData	Pointer to character buffer where software version string is placed.

Returns

The MI_GetPSWVersion function returns SUCCESS (0) if the software version can be retrieved.

MI_GetStaticNav

MI_SetStaticNav

Description

Functions to set/get the static navigation mode or determine if static mode is enabled/disabled.

Prototypes

```
WERR MI_GetStaticNav (UINT8 *pData);
WERR MI_SetStaticNav (UINT8 Data);
```

Arguments

Parameter	Description
Data	One byte: 1 = enable 0 = disable

Returns

The MI_SetStaticNav function returns SUCCESS (0) after settings are accepted. The `MI_GetStaticNav` function returns SUCCESS (0) if the static mode can be retrieved else FAILURE (-1).

MI_GetThroughput

Description

This Module Interface routine is currently not implemented for the SiRFstarIIe.

MI_GetGPSTime

Description

This function returns the current GPS Time. GPS time is represented as the total time since 01/06/80. This is given as a number of weeks and the time of week (TOW) in seconds. The actual week number rolls over every 1024 weeks which occurred in 09/1999. The value returned by this function is actually an extended week number and is the total number of weeks since 01/06/80 (i.e., it is larger than 1024).

Prototype

```
WERR MI_GetGPSTime (MI_GPS_TIME *pData);
```

Arguments

```
typedef struct
{
   INT16  WkNum;
   DOUBLE TOW;
} MI_GPS_TIME;
```

Returns

The MI_GetTimeGPS function returns SUCCESS (0) if the GPS time can be retrieved else FAILURE (-1).

MI_GetTrkData

Description

Function returns the tracking state of a series of channels. Verify that the starting channel number (Chnl) plus the number of channels to output (Cnt) is not greater than 11.

Prototype

```
WERR MI_GetTrkData (MI_TRACK_DATA *pData, int Chnl, int Cnt);
```

Arguments

```
typedef struct
{
   INT16  SVID;
   DOUBLE Azimuth;
   DOUBLE Elevation;
   INT16  State;
   UBYTE  AvgCNo;
   UBYTE  aCNo[SAMPLES_SEC];  /* SAMPLES_SEC currently 10 */
} MI_TRK_DATA;
```

Parameter	Description
Chnl	Channel number at which to start outputting track data.
Cnt	Number of channels for which to output data, starting at Chnl.

Returns

The MI_GetTrkData function returns SUCCESS (0) if the track state can be retrieved else FAILURE (-1).

Notes

To output the data for one channel, use a function call like the following where i is the channel of interest (0 to 11).

```
MI_GetTrkData(&trkdata, i, 1)
```

To output data for all channels, use:

```
MI_TRACK_DATA trkdata[NUM_OF_CHANNELS];
MI_GetTrkData ((MI_TRK_DATA *) &trkdata, 0, NUM_OF_CHANNELS)
```

This cycles from 0 to 11 and output the tracking data for each satellite into an array element of trkdata.

MI_GetTrkStateList

Description

This function returns the tracking state for all of the channels.

Prototype

```
WERR MI_GetTrkStateList (MI_TRK_STATE_LIST *ptslist)
```

Argument

```
typedef struct
{
  INT16 aState[NUM_OF_CHANNELS]; /* Currently NUM_OF_CHANNELS =12 */
} MI_TRK_STATE_LIST;
```

Returns

The MI_GetTrkStateList function returns SUCCESS (0) if the tracking states can be retrieved else FAILURE (-1).

UI_GetUartClkRate

Description

This function gets the current oscillator frequency for the purpose of setting the proper Baud rate in the UART_A_BAUD register (0x80030008) or UART_B_BAUD register (0x80030018). The clock frequency is set to the GPS, ACQ or External clock sources. The GPS and ACQ clocks are generated by the RF section and are the same regardless of board type. The external clock value is set in UI_MSG.C and is based on the board type, either SDK, S2AR or Undefined. If an external crystal is attached that is not at the default value of 25 MHz, you must change the #DEFINE SDK_CLOCK_FREQ, S2AR_CLOCK_FREQ and DEFAULT_CLOCK_FREQ values. These values are given in Hz. The clock divider for the Baud rate is calculated correctly regardless of the external clock frequency. If you change the external clock, it might be necessary to change the number of wait states for each chip select. See Chapter 13, “GPIO Lines, Throughput and Wait States” for more details.

Prototype

```
WERR UI_GetUartClkRate(int *clkRate)
```

Arguments

The CPU clock value in Hz. This value can currently have the following values.

```
#define GPS_CLOCK_FREQ       49107000L
#define SDK_CLOCK_FREQ       25000000L
#define S2AR_CLOCK_FREQ      25000000L
#define DEFAULT_CLOCK_FREQ   25000000L
```

Parameter	Description
clkRate	The CPU clock value in Hz.

Returns

The UI_GetUartClkRate function returns SUCCESS (0) if the clock rate can be retrieved else FAILURE (-1).

MI_GetUTC

Description

This function returns the current hours, minutes, seconds, TOW and week number all corrected for the UTC parameters received in subframe four of the navigation message. If UTC correction parameters have not yet been decoded from the 50 bps satellite navigation information, the routine returns GPS time in the passed variables (since this is the best information available).

Prototype

```
WERR MI_GetUTC(INT16 *pHrs,INT16 *pMins,double *pSecs,INT16
*pWkNum,double *pTOW);
```

Arguments

Parameter	Description
pHrs	Number of hours in the day (corrected for UTC if possible).
pMins	Number of minutes in the hour (corrected for UTC if possible).
pSecs	Number of seconds in minute (corrected for UTC if possible).
pWkNum	Number of GPS weeks (corrected for UTC).
pTOW	GPS Time of Week adjusted for UTC leap seconds if possible.

Returns

A return of SUCCESS indicates that the best available time was returned. If time can not be retrieved, it returns FAILURE.

MI_GetVelEcef

Description

Function returns the current WGS84 ECEF velocity vector.

Prototype

```
WERR MI_GetVelEcef (VECEF *pData);
```

Argument

```
typedef struct
{
    DOUBLE  Vx;
    DOUBLE  Vy;
    DOUBLE  Vz;
}
VECEF;
```

Returns

The MI_GetVelEcef function returns SUCCESS (0) if the velocity can be retrieved else FAILURE (-1).

MI_GetVelNed

Description

Outputs the current velocity vector in North, East and Down components.

Prototype

```
WERR MI_GetVelNed (VNED *pData)
```

Argument

```
typedef struct
{
    DOUBLE  Vn;
    DOUBLE  Ve;
    DOUBLE  Vd;
}
VNED;
```

Returns

The MI_GetVelNed function returns SUCCESS (0) if the velocity can be retrieved else FAILURE (-1).

MI_GetGSPVersion

Description

This function returns the internal version of the GSP chip.

Prototype

```
WERR MI_GetGSPVersion (char *pData);
```

Arguments

Parameter	Description
pData	Pointer to a buffer that receives the GSP version string. The GSP2e chip returns GSP2.0.

Returns

The MI_GetGSPVersion function returns SUCCESS (0) if the version of the GSP chip can be retrieved.

MI_GetVisList

Description

Function returns the current satellite list. A visible list may not be available until first acquisition. The azimuth and elevation for each satellite is not correct until a first fix is generated. The visible list does not contain any unhealthy satellites.

Prototype

```
WERR MI_GetVisList (MI_VIS_LIST *pData);
```

Arguments

```
typedef struct
{
   INT16  SVID;        /* sat PRN id */
   double Azimuth;
```

```
    double Elevation;
} MI_VISIBLE;

typedef struct
{
    UINT8      SVIDCnt;  /* number of valid entries in aVis[] element*/
    MI_VISIBLE aVis[MAX_SVID_CNT]; /* visible sat info */
} MI_VIS_LIST;
```

Returns

The MI_GetVisList function returns SUCCESS (0) if the visible list can be retrieved else FAILURE (-1).

MI_SetComm

Description

Function to set the serial communication parameters for a given protocol (independent of port). The function can also be used to reset the receiver so the new settings will have effect.

Prototype

```
WERR MI_SetComm (UINT8 UI_ProtoSrc, void *pParams, BOOL
bInitialize);
```

Arguments

```
typedef struct _UARTParams  /* from uart.h */
{
   UINT32 baud;     /* baud rate */
   UINT8  bits;     /* data bits */
   UINT8  stop;     /* stop bits */
   UINT8  parity;   /* parity    */
   UINT8  pad0;     /* not used  */
} UARTParams;
```

Parameter	Description
UI_ProtocSrc	UI protocol to be changed (finds the current port that this protocol is active on). 0 = SiRF Binary 1 = NMEA 2 = ASCII 3 = RTCM 4 = USER1
pParams	Points to data structure of type UARTParams to be set.
bInitialize	Flag to initialize communication port for new settings to take effect. If this value is TRUE, the board resets.

Returns

SUCCESS (0) if the settings are accepted else FAILURE (-1).

MI_SetDgpsCorrs

Description

This function enables the user to input a set of differential corrections directly into the GPS Core. This might be useful for a user-defined interface that can obtain the information from an outside source. See Chapter 12, “Adding a New User Protocol” for more information. When this function is called, it automatically sets the DGPS correction source as COR_SOFTWARE. Currently, this function transforms every correction into RTCM type 1 format before submitting it to the GPS core. See the *RTCM Recommended Standard for Differential Navstar GPS Service,* Version 2.2 RTCM Special Committee No. 104 document for details.

Prototype

```
WERR MI_SetDgpsCorrs(INT16 SVID,double GPSTime,double PRC,double
RRC,INT16 IOD);
```

Arguments

Parameter	Description
SVID PRN	GPS Satellite ID for the current correction.
GPSTime	Time of the correction in seconds (TOW).
PRC	Pseudorange correction in meters.
RRC	Pseudorange rate of change in meters/second.
IOD	Issue of Date (IOD) of the ephemeris used to generate the correction.

Returns

Currently always returns SUCCESS.

MI_SetNmeaProto

Description

This function attempts to find a port running a certain user interface protocol. If such a port exists, it changes the port to NMEA protocol with the given message settings and then reset the receiver so the new settings take effect.

Prototype

```
WERR MI_SetNmeaProto (UINT8 Proto, MI_NMEA_INIT *pData);
```

Arguments

Parameter	Description
Proto	Protocol to be found on a given port and changed to NMEA.

```
typedef struct
{
   UINT8 rate;             /* update rate in secs  */
   UINT8 cksum;            /* cksum on/off         */
} MI_NMEA_CFG;

typedef struct
{
   UINT8   mode;        /* ON :turn debug print on in SiRF protocol */
                          /* OFF:no NMEA msg in SiRF protocol mode */
                           /* ENABLE:Set receiver to NMEA mode */
   MI_NMEA_CFG NMEACfg[10];  /* messages are contained in amdNMEA[]
table */
   UINT16  baudrate;     /* desired baud rate    */
} MI_NMEA_INIT;
```

Returns

SUCCESS (0) if the settings are accepted else FAILURE (-1).

MI_SetUiProto

Description

Function that determines if a given User Interface protocol is being used on any port and then changes it to a new protocol. The function is hardcoded to ignore the USER1 protocol if the bChkValid flag is TRUE. It does not set a new protocol if that protocol already exists on another port.

Prototype

```
WERR MI_SetUiProto (UINT8 FrProto, UINT8 ToProto, BOOL bChkValid,
BOOL bReset);
```

Arguments

Parameter	Description
FrProto	Current UI protocol to be changed (if it is currently being used).
ToProto	New Protocol to set on the port running FrProto.
bChkValid	Flag to indicate that new protocol must be checked for validity.
bReset	Flag to indicate a receiver reset, so new protocol setting can take effect.

Return Value

SUCCESS (0) if the settings are accepted else FAILURE (-1).

MI_SetBaud

Description

Function sets the baud rate for the given UI_Protocol only if the UI_Protocol value is currently active on one of the two UART ports. If one of the ports is using the given protocol and the baud rate value is valid, then the baud rate will be set.

Prototypes

```
WERR MI_SetBaud (UINT8 UI_ProtoSrc, UNIT32 BaudRate);
```

Arguments

Parameter	Description
UI_ProtoSrc	UI PROTOCOL is an enum type that is defined in the `UI_INC.H` file. This value is used as an index into the PROTOCOL_CFG ProtocolCfg[] array defined in UI_MSG.C. Verify that the UI_PROTOCOL enum type value does not exceed the elements found in ProtocolCfg[] array. For example, if the user decides to add a new UI_PROTOCOL called UI_PROTO_USER2, then the user must make sure this new protocol element is appended to the ProtocolCfg[] array. The `MI_SetBaud` routine finds the current port that this protocol is active on. 0 = UI_PROTO_SIRF 1 = UI_PROTO_NMEA 2 = UI_PROTO_ASCII 3 = UI_PROTO_RTCM 4 = UI_PROTO_USER1 5 = UI_PROTO_NULL
BaudRate	Valid baud rates are 1200, 2400, 4800, 9600, 19200, 38400, and 57600.

Returns

The `MI_SetBaud` function returns SUCCESS (0) if the baud rate setting is accepted else FAILURE (-1).

MI_GPSStop

MI_GPSStart

Description

The stop function stops all GPS tasks from running by blocking all tasks in the task list. The stop function disables 100 ms interrupt. If user task is enabled (TASK_PERIOD is non zero), then user task will be unblocked and allowed to run.

The start function turns the ASIC on only if the ASIC is off and then forces a Nav Reset.

Prototypes

```
WERR MI_GPSStop (void);
WERR MI_GPSStart (void);
```

Arguments

None.

Returns

This function returns FAILURE if GPS reset is in process or GPS has been stopped by the user. Otherwise it returns SUCCESS.

MI_GetEstPosError

Description

Function gets an estimated position error for a given sigma. The estimated position error is communicated through a pointer to the given error ellipse structure. This structure is populated with error parameters if the function is successful.

Prototypes

```
int MI_GetEstPosError (int Sigma, tErrorEllipsePtr EPtr);
```

Arguments

Parameter	Description
Sigma	The degree of confidence that the position lies with in the error ellipse. Valid range of values are [1,10]. [1,10] denotes 1 <= Sigma <= 10.

```
typedef struct ErrorEllipse_ {
    int GPSWeek;
    double GPSTow;
       double Lat;
       double Lon;
       float Alt;
    float MajorAxisMeter;
    float MinorAxisMeter;
    float HeadingRad;
    float VerticalErrorMeter;
    tNavPosSrc PosSrc;
} tErrorEllipse, *tErrorEllipsePtr;
```

Returns

The `MI_GetEstPosError` function returns an integer value ranging from [0,3]. If 0 is returned then the error ellipse structure was successfully filled with the latest navigation state data. A failure is indicated with a return value from [1,3] where 1 indicates the Nav library was not initialized, 2 indicates the sigma value is out of range [1,10], and 3 indicates the Nav state is corrupted and position source is invalid.

MI_GetPtfPeriod

MI_SetPtfPeriod

Description

Functions to get/set PushToFix cycle time.

Prototypes

```
WERR MI_GetPtfPeriod( UINT32* pData );
WERR MI_SetPtfPeriod( UINT32 Data );
```

Arguments

Parameter	Description
PData	Pointer for the Push-to-Fix cycle time in seconds.
Data	Push-to-Fix cycle time in seconds. Valid range is [10,7200] seconds. [10,7200] denotes 10 <= Data <=7200.

Returns

The `MI_GetPtfPeriod` function returns FAILURE(-1) if the GPS is stopped else SUCCESS(0) indicates the Push-to-Fix cycle time was retrieved.

The `MI_SetPtfPeriod` function returns FAILURE(-1) if the Push-to-Fix cycle time is not within [10,7200] seconds else SUCCESS(0) indicates Push-to-Fix cycle time was set.

MI_GetTestModeData

MI_SetTestMode

Description

Functions to get/set test mode data/parameters respectively.

Prototypes

```
WERR MI_GetTestModeData (MI_TEST_MODE *pData);
WERR MI_SetTestMode (MI_OP_MODE *pData);
```

Arguments

```
typedef struct _MI_OP_MODE
{
   UINT16 Mode;    /* OpMode: 0 = Normal, 0x1E51 = Test, 0x1E52 - Test2, 0x1E53  -Test3 */
   UINT16 SVID;    /* SVid to search for (in test mode)   */
   UINT16 Period;  /* output message period (test mode)   */
} MI_OP_MODE;

typedef struct
{
   UINT16 SVid;            /* fixed SVid to search for on all channels */
   UINT16 period;          /* number of seconds statistics are accumulated over */
   UINT16 bitSynchTime;    /* time to first bit synch */
   UINT16 bitCount;        /* Count of data bits came out during a period */
   UINT16 poorStatusCount; /* Count of 100ms periods tracker spent in any status < 3F */
   UINT16 goodStatusCount; /* Count of 100ms periods tracker spent in status 3F */
   UINT16 parityErrorCount;/* Number of word parity errors */
   UINT16 lostVCOCount;    /* number of msec VCO lock loss was detected */
   /* following members added for testmode II */
   UINT16 frameSynchTime;  /* time to first frame synch */
   INT16 cNoMean;         /* c/No mean in 0.1 dB-Hz */
   INT16 cNoSigma;        /* c/No sigma in 0.1 dB */
   INT16 clockDrift;      /* clock drift in 0.1 Hz */
   INT32 clockOffset;     /* clock offset in 0.1 Hz */
   /* for bit test at a high c/no */
   INT16 bad1KhzBitCount;  /* bad bit count out of 10,000 (10 seconds * 1000 bits ) */
   INT32 absI20ms;         /* phase noise estimate I20ms sum*/
   INT32 absQ1ms;          /* phase noise estimate Q1ms sum*/
   INT32 rsvd1;
   INT32 rsvd2;
   INT32 rsvd3;
} MI_TEST_MODE;
```

Returns

The `MI_GetTestModeData` function returns FAILURE(-1) if no test data is available else returns SUCCESS(0) if test data is available.

The `MI_SetTestMode` function returns SUCCESS(0) if test mode parameters are set.

MI_GetUserDRTimeout

MI_SetUserDRTimeout

Description

Functions to get/set user dead reckoning timeout value.

Prototypes

```
UINT32 MI_GetUserDRTimeout( void );
void MI_SetUserDRTimeout( UINT32 Input );
```

Arguments

Parameter	Description
Input	Dead reckoning timeout value (int value) in seconds.

Returns

The `MI_GetUserDRTimeout` function returns the user dead reckoning timeout value.

The `MI_SetUserDRTimeout` function returns nothing.

MI_GetUserParams

MI_SetUserParams

Description

Functions to get/set user task parameters.

Prototypes

```
WERR MI_GetUserParam (MI_USER_PARAM *pData);
WERR MI_SetUserParam (MI_USER_PARAM *pData);
```

Arguments

```
typedef struct _MI_USER_PARAM
{
   BOOL  UserTasksEnabled;
   INT32 UserTaskInterval;  /* in milliseconds */
} MI_USER_PARAM;
```

Returns

The `MI_GetUserParams` function returns SUCCESS(0) if user task parameters were set, else returns FAILURE(-1) if GPS has stopped (no GPS tasks are running).

The `MI_SetUserParams` function returns SUCCESS(0).

MI_LpDbgOutput

Description

Function to output debug messages for Low Power. Outputs message of continuous power, Push to Fix, or Trickle Power enabled. Outputs message of user task enabled or disabled.

Prototypes

```
WERR MI_LpDbgOutput(void);
```

Returns

Function returns SUCCESS(0).

MI_GetUtcOffset

MI_SetUtcOffset

Description

Functions get/set UTC offset.

Prototypes

```
int MI_GetUtcOffset( void );
void MI_SetUtcOffset( int Input );
```

Arguments

Parameter	Description
Input	UTC offset value (int value) in seconds.

Returns

The `MI_GetUtcOffset` function returns the UTC offset value.

MI_SetSbasPrn

Description

Function that sets SBAS (Satellite Based Augmentation System) prn value.

Prototypes

```
WERR MI_SetSbasPrn(int prn);
```

Arguments

Parameter	Description
prn	Prn value of the SBAS satellite of interest. If prn = 0, then automatic selection of SBAS satellite is performed. Valid prn are values: 0,[120,138]. [120, 138] denotes 120 >= prn <= 138.

Returns

The `MI_SetSbasPrn` function returns SUCCESS (0) if the prn input is 0 or a value greater or equal to 120 and a value smaller or equal to 138. Otherwise FAILURE (-1) is returned.

Error Ellipse Functions D

Computation of Navigation Error Ellipse

The navigation error ellipse is an estimation of the sensitivity of the navigation solution to errors in the measurements. It is derived from the elements of the system covariance matrix. Traditionally the covariance matrix from the Dilution of Precision (DOP) computation is used because it is instantaneous and reflects only the measurements used in the corresponding navigation computation. Alternatively, the covariance matrix from the Kalman filter can be used, but that covariance matrix is weighted and is a function of navigation history.

The trace of the DOP covariance matrix represents the expected error sensitivity in the north, east, up and time axis. The off diagonal elements are functions of the cross-correlation between the errors in primary directions. The 2-D navigation error ellipse is the uncorrelated representation of the system variances in the horizontal plane. It is obtained from the north, east and north-east elements of the DOP covariance matrix. The ellipse is described by three parameters, the lengths of the major and minor axis and the orientation of the major axis with respect to north. The uncorrelated covariance matrix, U, is obtained from the correlated DOP covariance matrix C by matrix-rotating U so that diagonal elements become zero. If R is the rotation matrix then:

$$U = R \bullet C \bullet R^{T}$$

where

$$C = \begin{bmatrix} \nu^2_{North} & \nu^2_{North_East} \\ \nu^2_{North_East} & \nu^2_{East} \end{bmatrix}$$

$$R = \begin{bmatrix} cos(r) & -sin(r) \\ sin(r) & cos(r) \end{bmatrix}$$

and

$$U = \begin{bmatrix} U_{11} & 0 \\ 0 & U_{22} \end{bmatrix}$$

The required rotation angle, r, is computed from the components of the covariance matrix C.

$$sin(2r) = 2 * cos(r) * sin(r) = 2 * C_{12} = 2 * \nu^2_{North_East}$$

$$cos(2r) = cos(r)^2 - sin(r)^2 = C_{11} - C_{22} = \nu^2_{North} - \nu^2_{East}$$

and

$$r = \frac{1}{2} * \tan^{-1}\left(\frac{\sin(2r)}{\cos(2r)}\right) = \frac{1}{2} * \tan^{-1}\left(\frac{2 * \cos(r) * \sin(r)}{\cos(r)^2 - \sin(r)^2}\right) = \frac{1}{2} * \tan^{-1}\left(\frac{2 * \nu^2_{North_East}}{\nu^2_{North} - \nu^2_{East}}\right) = \frac{1}{2} * \tan^{-1}\left(\frac{2 * C_{12}}{C_{11} - C_{22}}\right)$$

Once the ellipse's orientation, r, is determined, the uncorrelated matrix, U, can be computed from the correlated DOP matrix, C.

$$U_{11} = \left| C_{11} * cos(r)^2 + C_{22} * sin(r)^2 - 2 * C_{12} * cos(r) * sin(r) \right|$$

$$U_{22} = \left| C_{11} * sin(r)^2 + C_{22} * cos(r)^2 + 2 * C_{12} * cos(r) * sin(r) \right|$$

The semi-major axis of the error ellipse is the larger of U_{11} and U_{22}, and the semi-minor axis is the smaller of the two. If U_{22} is the larger axis then the rotation angle for the ellipse is r plus 90 degrees.

If $U_{11} \geq U_{22}$

Semi-major = U_{11}

Semi-minor = U_{22}

Rotation angle = r

Else if $U_{22} > U_{11}$

Semi-major = U_{22}

Semi-minor = U_{11}

Rotation angle = r+90 degrees.

File Descriptions

The following section provides a brief description of the source files included in the SiRFstarIIe SDK. First, the files are broken up according to the software architecture diagram (Figure 1-2 on page 1-4). This is followed by the file descriptions ordered alphabetically.

File Organization

The grouping of the files following is not exact since some files may have multiple purposes. The intent is to familiarize the reader with the files that can be associated with each part of the architecture.

Start-Up

arm_irq.c	armerror.c	main.c
armstart.s	clkadj.s	uiconfig.c

GPS Core

armmtest.c	iono.h	nldefalt.h
armreset.c	intrface.h	rtc.c
dgpstype.h	navconst.h	
ephdefs.h	nlsdk.c	

Tasking

asic_if.c	csection.s	schedule.c
asic_isr.c	sch_icd.h	

UART

asicuart.c	msgmgr.h	uartbuf.h
asicuart.h	protocol.h	umanager.c
list.c	uart.c	umanager.h
list.h	uart.h	umconfig.c
msgmgr.c	uartbuf.c	umconfig.h

User Interface

protocol.h	ui_nmea.h	ui_user1.c
ui_ascii.c	ui_rtcm.c	ui_user1.h
ui_ascii.h	ui_rtcm.h	user.c
ui_if.h	ui_sirf.c	user_if.h
ui_msg.c	ui_sirf.h	
ui_nmea.c	ui_sram.c	

Memory

asic_icd.h	ram.sct	ui_sram.c
cache.c	sdram.c	update.c
crc.c	sram.c	
flash.sct	sram_icd.h	

Module Interface (including utility files)

asic_if.c	nl_if.h	trk_if.h
asic_if.h	ntsc_if.h	ui_if.h
bit_if.h	rtc_if.h	uiconfig.c
dgps_if.h	rxm_if.h	user_if.h
exec_if.h	sd_if.h	userinit.c
gpstype.h	sio_dbug.h	utc.h
init_if.h	sio_if.h	util_if.h
lp_if.h	stdtype.h	waas_if.h
mi_if.h	swt_if.h	

Individual File Descriptions

.

arm_irq.c — Contains IRQ initialization. Function called from main.c. Interrupts to be enabled at startup could be added to this file.

armerror.c — Contains routines to handle exceptions. Called from the embedded library.

armmtest.c — Memory test code, not used currently.

armreset.c — The last stop before the board executes a soft reset. User resets must be called through MI_ForceReset.

armstart.s — Assembly file. Contains code entry point. Sets up exception vector table and provides assembly handlers that can call C-code for various exceptions. Also sets up stack pointers and initializes memory. Processor starts up in 32 bit ARM instruction mode, the exception handlers must branch to 16 bit THUMB instructions since this is what all the C-code is compiled for. After reset, this file eventually branches to main() in main.c.

asic_icd.h — Contains #define values for the ASIC registers.

asic_if.c — ASIC related functions.

asic_if.h — Contains a few #define values along with ASIC related functions prototypes and macros.

asic_isr.c — Most functions in this file are not used with the GSP2 hardware tracker. The primary function in this file is asicISR(). This is called by the IRQ exception handler through SCH_ISR(). It is the asicISR() function that determines the source of every interrupt enabled in the standard software and calls the appropriate handler. Note that there are two interrupt levels in the ARM, a fast interrupt (FIQ) and a normal interrupt (IRQ). The software only uses the IRQ interrupt so multiple sources for this interrupt are possible. When an interrupt is received, it must be acknowledged.

asicuart.c	Contains the ISR routines for the GSP2 UARTs. Also contains code to initialize the UART structures and enable/disabled interrupts. uartISRHandler() is called from asicISR() in asic_isr.c if a UART interrupt occurs.
asicuart.h	Contains macros and bit masks for working with the ASIC UART registers.
bit_if.h	Contains #defines, variables/structures, and function prototypes associated with Bit in Test and debug.
cache.c	Code to enable/disable the cache. Cache and initial acquisition are mutually exclusive. When enabling cache, the TAG memory must be cleared.
ck.c	Not part of the GSW2 embedded software.
cksum.exe	Not part of the GSW2 embedded software.
clkadj.s	Assembly file. Contains two functions for switching the clock source, one for switching to ECLK and one for GPSCLK. GPSCLK is the default at start-up. Also includes code in these functions for setting the proper wait states for each chip select and for selecting the clock divisor.
crc.c	Cyclic redundancy functions. Used mainly to protect battery-backed memory. Also used by the USER1 protocol generic output.
csection.s	Contains the critical section enter/exit code.
dgps.key	Not part of the GSW2 embedded software.
dgps_if.h	Contains prototypes, defines, and constants related to DGPS that are used by other modules.
dgpstype.h	Contains defines and structures related to DGPS. Not for SDK development and must not be modified.
ephdefs.h	Not for SDK development. Must not be modified.
exec_if.h	Contains information for update.c, #defines for tasks, and prototype information regarding intrface.c (core file).
flash.sct	Scatter load file for HwtFlash build.
gpstype.h	Contains values and structures used for position calculations.
init_if.h	Contains #defines/enum types, variables/structures, and function prototypes associated with initialization and start-up modes.
intrface.h	Not for SDK development. Must not be modified.
iono.h	Not for SDK development. Must not be modified.
list.c	Functions for the formation and management of singly-linked lists (used for UART buffering code). The UART buffering system is managed as a series of singly-linked lists (two sets, one for each port). The list.c functions are generic singly-linked list functions. UART Buffer specific code is implemented elsewhere.

list.h	Defines generic linked list structures. These singly-linked list structures are the basis for the UART buffering code. See list.c for more details.
lp_if.h	Must not be modified by SDK user. Function prototypes for TricklePower operation. Of interest is the DelayMicroSeconds() and DelayMicroSecondsIntEnabled() functions. These offer the SDK user a means to delay code execution by a desired number of milliseconds. This delay is based on the Real-Time Clock (RTC).
main.c	Contains the C-entry point for the code. MAIN.C function is branched to from armstart.s. Sets alternate functions of GPIOs, calls user configuration code and hardware detection code. Sets source of UART clock. Starts Nav and falls to a form of background loop. After this, the code is interrupt driven.
mi_if.h	Important file for Module Interface functions. Includes all the function prototypes for the Module Interface Get and Set functions. These functions provide the SDK user a means to obtain data from the GPS core. There are also many utility functions that are provided. Contains a list of the possible events that can be signaled by the GPS core. Events are used as I/O message triggers and an event is signaled by calling UI_Event() in UI_MSG.C. Also included here are the definitions of structures used by the MI routines.
mkdefalm.exe	Not part of the GSW2 embedded software.
msgmgr.c	Functions to support input message handler architecture. The I/O protocol architecture is set up to enable the registration of message handlers for each active protocol. Each message handler is tied to a unique Message ID (MID) between 0 and 255. When a input message is received (meaning the ISR recognized the termination characters), the code eventually checks for a MID and calls the associated message handler. If there is no associated message handler for that MID or there has been a COM error, an error handler is called instead.
msgmgr.h	Header file for msgmgr.c, check this file for specifics.
navconst.h	Not for SDK development. Must not be modified.
nl_if.h	Some enumerated types in this file are used to set values in NLDEFALT.H. Otherwise, this file is not intended for SDK development.
nldefalt.h	Contains default settings for the navigation core. These settings may be changed by the user. Note that SiRF has not run full test suites on all combinations of the values. If you change these values, it is recommended that you verify the proper behavior with adequate testing.
nlsdk.c	One function called at start-up that passes user variables to the GPS core. This file must not be modified by the SDK user. Most of the user settable #DEFINE values are contained in NLDEFALT.H and can be changed by the user.

ntsc_if.h	Not for SDK development. Must not be modified.
protocol.h	Contains all the Message ID's (MID's) for the SiRF binary protocol.
ram.sct	Scatter load file for HwtRam build.
rtc.c	Contains files for manipulating, initializing and reading the RTC registers. In general, the RTC (real-time clock) code is closely tied to the GPS code and must be left alone. This code ensures that the RTC accuracy is sufficient to allow for TricklePower operation. Some functions at the bottom of the file can be used by the SDK user to obtain time differences based on the RTC counters.
rtc_if.h	Header file for RTC.C. Contains function prototypes for the functions mentioned in the RTC.C description. To use these functions, include this header file.
rxm_if.h	Contains prototypes, defines, and structures related to receiver manager that are used by other modules.
sch_icd.h	Defines task priorities and functions associated with each task. Closely tied to the scheduler code in scheduler.c.
schedule.c	Contains the C-code IRQ interrupt handler (SCH_ISR()) that is called from the assembly IRQ exception handler in armstart.s. SCH_ISR() then calls asicISR() (see asic_isr.c file for more details). This file also contains the code for implementing the scheduler (basic OS). The function that determines the appropriate task to execute is dispatch(). There are also several access functions in this file that are called from the SiRF object code. These are in place so the user can adjust the default task priorities without breaking the SiRF object code.
sd_if.h	Satellite data interface declarations.
sdram.c	Contains declaration of zero-initialized data structure that is maintained in SDRAM. This structure occupies the RAM space directly after the first 4kB of SRAM where the battery-backed memory is resident. The data associated with this SDRAM file is not cleared on a soft reset. There is no reason for the SDK user to modify this file.
sio_dbug.h	Contains serial Input/Output related information used for debugging.
sio_if.h	Contains serial Input/Output related information regarding UART/Serial communication and message/buffer handling. Also contains UART structure for UARTControl.
sram.c	Contains access routines for SRAM data area.
sram_icd.h	Contains #defines, variables/structures, and function prototypes regarding SRAM data area.
ss2_sdk.mcp	ARM ADS tool-chain project file.
stdtype.h	Contains processor specific types (i.e. UINT32 for an unsigned long int) that must be used in coding.
swt_if.h	Contains information regarding software time.

trk_if.h	Contains prototypes, defines, and structures related to receiver manager that are used by other modules.
uart.c	Contains functions that are used to implement the UART buffering scheme. The UART buffering scheme is based on a series of linked lists. The basic singly-linked list functions are defined in LIST.C. The specific UART buffer function pointers are typecast to these basic functions in UARTBUF.H and LIST.H. See also LIST.C, LIST.H, UARTBUF.C and UARTBUF.H.
uart.h	Contains function prototypes for the uart code, mainly uart.c. A pointer to this structure type (*UARTDevice) is included in the UMProtocolData structure declared in umanager.
uartbuf.c	This file contains the majority of global variables that are used by the UART buffering scheme to reserve space in RAM. This space is managed by a series of pointers arranged as a group of singly-linked lists. Functions in the file enable external files to manipulate these pointers and set up the data space for use in the lists. The actual UART structures are declared in UMANAGER.C (UMProtocolData UMPData[NUM_UARTS]). Also check LIST.C, LIST.H, UART.C and UARTBUF.H.
uartbuf.h	This file derives the specific UART buffer class from the generic linked list functions and structures of list.h and list.c. See uartbuf.c for more details.
ui_ascii.c	Contains the framework (shell) used for ASCII protocol.
ui_ascii.h	Header file for UI_ASCII.C. Contains function prototypes for ASCII protocol. The ASCII protocol is not currently used in the code.
ui_if.h	Contains the function prototypes for the UI_MSG.C file. These files along with UMANAGER.C form the application layer just above the protocol specific implementation files. See UI_MSG.C for more details.
ui_msg.c	This file exists as a layer just above the individual protocols. There are two sets of functions in this file. The first is a set of default NULL functions that are assigned to each port until a specific protocol is associated with that port. When the protocol is being instantiated, these NULL functions are replaced by the protocol specific functions (for SiRF binary these are in UI_SIRF.C, for NMEA these are in UI_NMEA.C, RTCM is in UI_RTCM.C and USER1 is in UI_USER1.C). The second set of functions are called out of the GPS core and are meant to be protocol independent. This independence is generated by using function pointers and allowing these function pointers to be modified by specific protocols. This file also contains two handles for the UMProtocolData structure in UMANAGER.C, one for each UART. Note the default clock frequency #define values are defined here so customers with different ECLK frequencies can set them here.

ui_nmea.c	Contains most files associated directly with I/O NMEA messaging. This includes input message handlers that are registered in NmeaOpen() (see msgmgr.c for more details). This also includes output message handlers that create standard NMEA strings. The data (time, position, velocity, etc.) for generating the output payload is obtained from the GPS core using the MI_Get() functions (see MI_ICD.H). There are also functions that are called directly from the GPS core through UI_MSG.C. These functions are prefaced with UI_ and are set up as a series of function pointers. See UI_Open() in UI_MSG.C for details on the redirection. The NMEA specific functions that handle buffer transport between the application layer and the ISR are contained in NMEAMGR.C.
ui_nmea.h	Header file for ui_nmea.c. Contains default message settings and Baud rate.
ui_rtcm.c	Contains routines for implementing the RTCM protocol. RTCM is an input only protocol (receive only). Also, the data for RTCM input is passed from the ISR to the application layer one byte at a time instead of one buffer at a time.
ui_rtcm.h	Header file for UI_RTCM.C. Several of the #define values in this file are not currently used.
ui_sirf.c	This file is similar in function to UI_NMEA.C. It also includes input message handlers and output message handlers. The input message handlers are registered in SirfOpen() and are managed using the functions in MSGMGR.C. The input message handlers use MI_SetXXX() functions to set variables in the GPS core. The output messages are triggered by Events generated in the GPS Core (i.e. a function call to UI_Event() in UI_MSG.C that calls the output function pointers for each protocol). The payload data (time, position, velocity, etc.) for each message is obtained from the GPS core using MI_GetXXX() functions (see MI_ICD.H). There are also functions that are called directly from the GPS core through UI_MSG.C. These functions are prefaced with UI_ and are set up as a series of function pointers. See UI_Open() in UI_MSG.C for details on the redirection. The SiRF binary specific functions that handle buffer transport between the application layer and the ISR are contained in UI_SIRF.C.
ui_sirf.h	Header file for UI_SIRF.C. Contains some default Protocol settings for baud rate and parity.
ui_sram.c	Functions for the manipulation of the user portion of battery-backed SRAM. The default settings for the user portion of battery-backed memory are set in this file. Specifically, in the UI_SetUiSram() function there is a case statement for ID_INITIALIZE that sets the default values if the battery-backed memory is determined to be corrupt at start-up. Note that the user portion of battery-backed memory is protected by a CRC. The UI_SetUiSram() function can be used to set the various elements in the structure as it re-calculates the CRC before terminating.

ui_user1.c	This file provides two sets of functions. The first set is for implementing a USER1 protocol while the second is for implementing user tasks. For the USER1 protocol, this file contains the same type of functions found in UI_NMEA.C for NMEA and for UI_SIRF.C for SiRF binary. Read the sections on those files for implementation of the I/O functions through function pointers. The UI_UserTask() function is the function called by the scheduler when the user tasks are activated. The user tasks are activated by setting TASK_PERIOD to a non-zero value in the preprocessor settings. The task priorities and associated functions are set in SCH_ICD.H. Note that all of the code associated with the user task in the standard code is for testing purposes and must not be enabled by the SDK user for their own application.
ui_user1.h	Header file for UI_USER1.C. Contains some default protocol settings (i.e. Baud rate). See UI_USER1.C for more details.
uiconfig.c	This file contains functions and a global array that determines the hardware setup. The global array is used to determine hardware type (S2AR, S2SDK), clock type (GPSCLK, ECLK) and Beacon type (Internal, CSI). There is a hardware configuration routine in this file that determines the board type based on GPIO36 and GPIO14/CS2. These values can also be hardcoded. Note the default clock offset (DEFAULT_CLKOFFSET) can now be set in this file formerly found in MI_UTIL.C.
umanager.c	This file along with the UI_MSG.C file forms part of the application layer just above the individual protocols. Many of the functions in this file are default functions, assigned to function pointers in the UART control structure (UMProtocolData UMPData[NUM_UARTS]). Depending on the protocol, some of these default function pointers can be overwritten to point to protocol specific functions. The primary structure used to control UART operation is declared here (UMProtocolData ...).
umanager.h	Header file for UMANAGER.C. Contains the definition of the UMProtocolData structure that is used to control the UARTs.
umconfig.c	Used in conjunction with UMANAGER.C. Mostly to manipulate handles to the various UART control structures.
umconfig.h	Header file for UMCONFIG.C.
update.c	This file was used in the SSI to jump to the boot loader contained in the first section of Flash memory. The boot code for the SSII is implemented inside the chip so this file has no relevance and is not currently used.
user.c	Contains code associated with user tasks and other user interface routines.
user_if.h	Contains #defines, variables/structures, and prototypes associated with USER.C.

userinit.c	Contains a few global variables that can be used to modify the behavior of the board. The userResetDelayCount variable is used to indicate the number of extra times that the board must attempt a warm start before falling back to a cold start. The user PPS offset variable is used to adjust the occurrence of the 1 PPS pulse.
utc.h	Contains UTC structures.
util_if.h	Prototypes for utilities like GetDate, ConvertTowToUtc, GetCog, etc.
waas_if.h	Contains prototypes, #defines, and structures related to WAAS that are used by other modules.

Non SDK Source/Build Files

ck.c	Source code for cksum.exe
cksum.exe	Microsoft C command line executable that creates NMEA checksums from NMEA sentences.
dgps.key	Procomm meta key file.
mkdefalm.exe	Executable used to create new almanac (DEFALM.H file). This is currently not supported as DEFALM.H is not provided as an SDK file.
nmea100.key	Procomm meta key file to send NMEA input messages.

Acronyms, Abbreviations and Glossary

F

This appendix describes all acronyms, abbreviations, and selected terms used in this document.

2-D	Two dimensional.
3-D	Three dimensional.
A/D	Analog to Digital.
Almanac	A set of orbital parameters that allows calculation of the approximate GPS satellite positions and velocities. A GPS receiver uses the almanac as an aid to determine satellite visibility during acquisition of GPS satellite signals. The almanac is a subset of satellite ephemeris data and is updated weekly by GPS Control.
Altitude	The distance between the current position and the nearest point on WGS84 reference ellipsoid. Altitude is usually expressed in meters and is positive outside the ellipsoid. In terms of the SiRFstar Evaluation Unit, this has no bearing on the height above mean sea level.
Altitude Hold	A technique that allows navigation using measurements from three GPS satellites plus an independently obtained value of altitude.
Altitude Hold Mode	A Navigation Mode during which a value of altitude is processed by the Kalman Filter as if it were a range measurement from a satellite at the Earth's center (WGS-84 reference ellipsoid center).
Baud	(See bps.)
bps	Bits per second (also referred to as a Baud rate).
C	Celsius, a unit of temperature.
C/A Code	Coarse/Acquisition Code. A spread spectrum direct sequence code that is used primarily by commercial GPS receivers to determine the range to the transmitting GPS satellite.
CEP	Circular Error Probable. The radius of a circle, centered at the user's true location, that contains 50 percent of the individual position measurement made using a particular navigation system.
Clock Error	The uncompensated difference between synchronous GPS system time and time best known within the GPS receiver.
C/No	Carrier-to-Noise density ratio.
Cold Start	A condition in which the GPS receiver can arrive at a navigation solution without initial position, time, current Ephemeris, and almanac data.
Control Segment	The Master Control Station and the globally dispersed Monitor Stations used to manage the GPS satellites, determine their precise orbital parameters, and synchronize their clocks.

dB	Decibel.
dBiC	Decibel-Isometric-Circular (measure of power relative to an isometric antenna with circular polarization).
dBm	Decibels per milliwatt.
dBW	Decibel-Watt (measure of power relative to one watt).
DC	Direct Current.
DGPS	Differential GPS. A technique to improve GPS accuracy that uses pseudorange errors recorded at known locations to improve the measurements made by other GPS receivers.
Doppler Aiding	A signal processing strategy that uses a measured doppler shift to help a receiver smoothly track a GPS signal to allow a more precise velocity and position measurement.
DoD	Department of Defense.
DOP	Dilution of Precision (see GDOP, HDOP, PDOP, TDOP, and VDOP).
DSP	Digital Signal Processor.
DTR	Data Terminal Ready.
ECEF	Earth-Centered Earth-Fixed. A Cartesian coordinate system with its origin located at the center of the Earth. The coordinate system used by GPS to describe 3-D location. For the WGS-84 reference ellipsoid, ECEF coordinates have the Z-axis aligned with the Earth's spin axis, the X-axis through the intersection of the Prime Meridian and the Equator and the Y-axis is rotated 90 degrees East of the X-axis about the Z-axis.
EEPROM	Electrically Erasable Programmable Read Only Memory.
EHPE	Expected Horizontal Position Error.
EMC	Electromagnetic Compatibility.
EMI	Electromagnetic Interference.
Ephemeris	A set of satellite orbital parameters that is used by a GPS receiver to calculate precise GPS satellite positions and velocities. The ephemeris is used to determine the navigation solution and is updated frequently to maintain the accuracy of GPS receivers.
EPROM	Erasable Programmable Read Only Memory.
EVPE	Expected Vertical Position Error.
FP	Floating-Point mathematics, as opposed to fixed point.
FRP	Federal Radionavigation Plan. The U.S. Government document that contains the official policy on the commercial use of GPS.
GaAs	Gallium Arsenide, a semiconductor material.
GDOP	Geometric Dilution of Precision. A factor used to describe the effect of the satellite geometry on the position and time accuracy of the GPS receiver solution. The lower the value of the GDOP parameter, the less the errors in the position solution. Related indicators include PDOP, HDOP, TDOP, and VDOP.
GMT	Greenwich Mean Time.
GPS	Global Positioning System. A space-based radio positioning system that provides suitably equipped users with accurate position, velocity, and time data. GPS provides this data free of direct user charge worldwide, continuously, and under all weather conditions. The GPS constellation consists of 24 orbiting satellites, four equally spaced around each of six different orbital planes. The system is developed by the DoD under Air Force management.

GPS Time	The number of seconds since Saturday/Sunday Midnight UTC, with time zero being this midnight. Used with GPS Week Number to determine a specific point in GPS time.
HDOP	Horizontal Dilution of Precision. A measure of how much the geometry of the satellites affects the position estimate (computed form the satellite range measurements) in the horizontal (East, North) plane.
Held Altitude	The altitude value that is sent to the Kalman filter as a measurement when in Altitude Hold Mode. It is an Auto Hold Altitude unless an Amended Altitude is supplied by the application processor.
Hot Start	Start mode of the GPS receiver when current position, clock offset, approximate GPS time and current ephemeris data are all available.
Hz	Hertz, a unit of frequency.
I/O	Input/Output.
IF	Intermediate Frequency.
IGRF	International Geomagnetic Reference Field.
IODE	Issue of Data Ephemeris.
JPO	Joint Program Office. An office within the U.S. Air Force Systems Command, Space Systems Division. The JPO is responsible of managing the development and production aspect of the GPS system and is staffed by representatives from each branch of the U.S. military, the U.S. Department of transportation, Defense Mapping Agency, NATO member nations, and Australia.
Kalman Filter	Sequential estimation filter which combines measurements of satellite range and range rate to determine the position, velocity, and time at the GPS receiver antenna.
Km	Kilometer (1Km = 1000 meters)
L1 Band	The 1575.42 MHz GPS carrier frequency which contains the C/A code, P-code, and navigation messages used by commercial GPS receivers.
L2 Band	A secondary GPS carrier, containing only P-code, used primarily to calculate signal delays caused by the atmosphere. The L2 frequency is 1227.60 MHz.
Latitude	Halfway between the poles lies the equator. Latitude is the angular measurement of a place expressed in degrees north or south of the equator. Latitude runs from 0° at the equator to 90°N or 90°S at the poles. When not prefixed with letters N or S, it is assumed positive north of Equator and negative south of Equator. Lines of latitude run in an east-west direction. They are called parallels.
LLA	Latitude, Longitude, Altitude geographical coordinate system used for locating places on the surface of the Earth. Latitude and longitude are angular measurements, expressed as degrees of a circle measured from the center of the Earth. The Earth spins on its axis, which intersects the surface at the north and south poles. The poles are the natural starting place for the graticule, a spherical grid of latitude and longitude lines. See also Altitude.
Longitude	Lines of longitude, called meridians, run in a north-south direction from pole to pole. Longitude is the angular measurement of a place east or west of the prime meridian. This meridian is also known as the Greenwich Meridian, because it runs through the original site of the Royal Observatory, which was located at Greenwich, just outside London, England. Longitude runs from 0° at the prime meridian to 180° east or west, halfway around the globe. When not prefixed with letters E or W, it is assumed positive east of Greenwich and negative west of Greenwich. The International Date Line follows the 180° meridian, making a few jogs to avoid cutting through land areas.
LPTS	Low Power Time Source.

LSB	Least Significant Bit of a binary word.
LTP	Local Tangent Plane coordinate system. The coordinates are supplied in a North, East, Down sense. The North is in degrees or radians, East in same units and Down is height below WGS84 ellipsoid in meters.
m/sec	Meters per second (unit of velocity).
m/sec/sec	Meters per second per second (unit of acceleration).
m/sec/sec/sec	Meters per second per second per second (unit of impulse or "jerk").
Mask Angle	The minimum GPS satellite elevation angle permitted by a particular GPS receiver design.
Measurement	The square of the standard deviation of a measurement quality. The standard deviation Error Variance is representative of the error typically expected in a measured value of the quantity.
MID	Message Identifier. In case of SiRF protocol, it is a number between 1 and 256.
MHz	Megahertz, a unit of frequency.
MSB	Most Significant Bit within a binary word or a byte.
MSL	Mean Sea Level.
MTBF	Mean Time Between Failure.
Multipath Error	GPS positioning errors caused by the interaction of the GPS satellite signal and its reflections.
mV	Millivolt.
mW	Milliwatt.
NED	North, East, Down coordinate system. See LTP.
NF	Noise Factor.
NMEA	National Marine Electronic Association. Also commonly used to refer to Standard For Interfacing Marine Electronic Devices.
NVRAM	Non-volatile RAM, portion of the SRAM that is powered by a backup battery power supply when prime power is removed. It is used to preserve important data and allow faster entry into the Navigation Mode when prime power is restored. All of the SRAM in SiRFstar receiver is powered by the backup battery power supply (sometimes also referred to as "keep-alive" SRAM).
Obscuration	Term used to describe periods of time when a GPS receiver's line-of-sight to GPS satellites is blocked by natural or man-made objects.
OEM	Original Equipment Manufacturer.
Overdetermined Solution	The solution of a system of equations containing more equations than unknowns. The GPS receiver computes, when possible, an overdetermined solution using the measurements from five GPS satellites, instead of the four necessary for a three-dimensional position solution.
P-Code	Precision Code. A spread spectrum direct sequence code that is used primarily by military GPS receivers to determine the range to the transmitting GPS satellite.
Parallel Receiver	A receiver that monitors four or more satellites simultaneously. SiRFstar Evaluation Unit can monitor up to 12 satellites simultaneously, due to the capabilities of the SiRF chipset it uses.
PDOP	Position Dilution of Precision. A measure of how much the error in the position estimate produced from satellite range measurements is amplified by a poor satellite geometry with respect to the receiver antenna.
Pi	The mathematical constant having a value of approximately 3.14159.

P-P	Peak to Peak.
PPS	Precise Positioning Service. The GPS positioning, velocity, and time service that are available on a continuous, worldwide basis to users authorized by the DoD.
PRN	Pseudorandom Noise Number. The identity of the GPS satellites as determined by a GPS receiver. Since all GPS satellites must transmit on the same frequency, they are distinguished by their pseudorandom noise codes.
Pseudorange	The calculated range form the GPS receiver to the satellite determined by measuring the phrase shift of the PRN code received from the satellite with the internally generated PRN code from the receiver. Because of atmospheric and timing effects, this time is not exact. Therefore, it is called a pseudorange instead of a true range.
PVT	Position, Velocity, and Time.
RAM	Random Access Memory.
Receiver Channels	A GPS receiver specification that indicates the number of independent hardware signal processing channels included in the receiver design.
RF	Radio Frequency.
RFI	Radio Frequency Interference.
ROM	Read Only Memory.
RTCA	Radio Technical Commission of Aeronautics.
RTCM	Radio Technical Commission of Maritime Services. Also commonly used as a reference to the standard format that DGPS corrections data is distributed in *RTCM Recommended Standard for Differential Navstar GPS Service*. SiRFstar receiver supports the latest Version 2.1 of this standard.
SA	Selective Availability. The method used by the DoD to control access to the full accuracy achievable with the C/A code.
Satellite Elevation	The angle of the satellite above the horizon.
SEP	Spherical Error Probable. The radius of a sphere, centered at the user's true location, that contain 50 percent of the individual 3-D position measurements made using a particular navigation system.
Sequential Receiver	A GPS receiver in which the number of satellite signals to be tracked exceeds the number of available hardware channels. Sequential receivers periodically reassign hardware channels to particular satellite signals in a predetermined sequence.
SNR	Signal-to-Noise Ratio, often expressed in decibels.
SPS	Standard Positioning Service. A position service available to all GPS users on a continuous, worldwide basis with no direct charge. SPS uses the C/A code to provide a minimum dynamic and static positioning capability.
SRAM	Static Random Access Memory. In context of this document, see also NVRAM.
SV	Satellite Vehicle.
TDOP	Time Dilution of Precision. A measure of how much the geometry of the satellites affects the time estimate computed from the satellite range measurements.
3-D Coverage	The number of hours-per-day with four or more satellites visible. Four visible satellites are required to determine a 3 dimensional position.
3-D Navigation	Navigation Mode in which altitude and horizontal position are determined from satellite range measurements.

TTFF — Time-To-First-Fix. The actual time required by a GPS receiver to achieve a position solution. This specification varies with the operating state of the receiver, the length of time since the last position fix, the location of the last fix, and the specific receiver design. See also Hot Start, Warm Start, and Cold Start mode descriptions.

2-D Coverage — The number of hours-per-day with three or more satellites visible. Three visible (hours) satellites can be used to determine location if the GPS receiver is designed to accept an external altitude input (Altitude Hold).

2-D Navigation — Navigation Mode in which a fixed value of altitude is used for one or more position calculations while horizontal (2-D) position can vary freely based on satellite range measurements.

UART — Universal Asynchronous Receiver/Transmitter that produces an electrical signal and timing for transmission of data over a communications path, and circuitry for detection and capture of such data transmitted from another UART.

UDRE — User Differential Range Error. A one sigma estimate of the pseudo range measurement error due to ambient noise and residual multipath.

UERE — User Equivalent Range Error.

Update Rate — The GPS receiver specification that indicates the solution rate provided by the receiver when operating normally. It is typically once per second.

UTC — Universal Time Coordinated. This time system uses the second defined true angular rotation of the Earth measured as if the Earth rotated about its Conventional Terrestrial Pole. However, UTC is adjusted only in increments of one second. The time zone of UTC is that of Greenwich Mean Time (GMT).

VCO — Voltage Controlled Oscillator.

VDOP — Vertical Dilution of Precision. A measure of how much the geometry of the satellites affects the position estimate (computed from the satellite range measurements) in the vertical (perpendicular to the plane of the user) direction.

VSWR — Voltage Standing Wave Ratio.

Warm Start — Start mode of the GPS receiver when current position, clock offset and approximate GPS time are input by the user. Almanac is retained, but ephemeris data is cleared.

WGS-84 — World Geodetic System (1984). A mathematical ellipsoid designed to fit the shape of the entire Earth. It is often used as a reference on a worldwide basis, while other ellipsoids are used locally to provide a better fit to Earth in a local region. GPS uses the center of the WGS-84 ellipsoid as the center of the GPS ECEF reference frame.

ADDITIONAL AVAILABLE PRODUCT INFORMATION

Part Number	Description
	Product Inserts
	SiRFstarIIe Evaluation Kit
	SiRFstarIIe System Development Kit
	Product Briefs
1055-1016	GSP2e
1055-1017	GRF2i
	Application Notes
APNT0003	Troubleshooting Guide
APNT0004	System RF Front-end Requirements for SiRFstar Architectures
APNT0006	PCB Design Guidelines
APNT0007	Open Short Detector
APNT0010	GRF2i QFN Introduction
APNT0011	S2AB Design Upgrade
APNT0012	GSP2e Hardware Implementation
APNT0013	SiRFstarI LXHS Back-up Power Operation
APNT0014	Connecting to the GSP2e Board Using the ARM Multi-ICE
APNT0015	SiRFstarII S2AR Back-up Power Operation
APNT0016	SiRFstarII Alternate Flash Programming Algorithms
APNT0017	Board Level Design for GSP2e
APNT0018	SiRFstarIIe Low Power Operating Modes
APNT0019	SSII CPU Clock and Hardware Detection
APNT0020	Implementing User Tasks on the SiRFstarIIe
APNT0021	S2AM Hardware Reference Design Description
APNT0022	Integrating Patch Antenna's with SiRF GPS Receivers
APNT0023	Effect of Increasing User Task Duty Cycle on Performance
APNT0024	SiRFstarIIe GPS Receiver Jamming Immunity
APNT0025	GPS Reset for SiRFstarII
APNT0026	Adding Elements to Battery-Backed SRAM
APNT0028	Battery Backed SRAM Operation at 49MHz with the GSP2e
APNT0029	GSP2e and GSP2e Cache
APNT0030	EHPE and EVPE Calculations
APNT0032	Interfacing a 3-wire Serial EEPROM with the GSP2e

SiRF Technology Inc.
148 East Brokaw
San Jose, CA 95112
Tel: +1-408-467-0410
Fax: +1-408-467-0420
Email: gps@sirf.com
Website: http://www.sirf.com

SiRF France
Tel: +33-3-82860415
Fax: +44-1344-668157
Email: rocky@sirf.com

SiRF Texas
Tel: +1-972-239-6988
Fax: +1-972-239-0372
Email: jdaniels@sirf.com

SiRF Germany
Tel: +49-81-529932-90
Fax: +49-81-529931-70
Email: peterz@sirf.com

SiRF United Kingdom
Tel: +44-1344-668390
Fax: +44-1344-668157
Email: aellis@sirf.com

SiRF Taiwan
Tel: +886-2-2723-7853
Fax: +886-2-2723-7854
Email: sirf_taiwan@sirf.com

SiRFstarIIe System Development Kit User's Guide, Part 1 - Software
© 2000-2002 SiRF Technology Inc. All rights reserved.

Protected by U.S. Patents #9740398VV, #5,897,605, #5,901,171, #5,719,383, #6,018,704, #6,037,900, #6,041,280, and #6,047,017 and #6,081,228. Other U.S. and foreign patents are pending. SiRF, the SiRF logo, and SiRFstar are registered trademarks of SiRF Technology, Inc. SnapLock, Foliage Lock, TricklePower, SingleSat, SiRFLoc, SiRFDRive, and WinSiRF are trademarks of SiRF Technology, Inc. Other trademarks are property of respective companies.

This document contains information on SiRF products. SiRF reserves the right to make changes in its products, specifications and other information at any time without notice. SiRF assumes no liability or responsibility for any claims or damages arising out of the use of this document, or from the use of integrated circuits based on this data sheet, including, but not limited to claims or damages based on infringement of patents, copyrights or other intellectual property rights. No license, either expressed or implied, is granted to any intellectual property rights of SiRF. SiRF makes no warranties, either express or implied with respect to the information and specification contained in this document. Performance characteristics listed in this document do not constitute a warranty or guarantee of product performance. SiRF products are not intended for use in life support systems or for life saving applications. All terms and conditions of sale are governed by the SiRF Terms and Conditions of Sale, a copy of which may obtain from your authorized SiRF sales representative.

May 2002