GCB11

Networked Microcontroller

REFERENCE
MANUAL

COACTIVE
AESTHETICS, INC

VERSION 1.4

GCB11

Reference Manual

Version 1.4

Covers:

GROM	**Version 1.4:**
GIOS	**Version 2.0**
GBUG	**Version 2.4**
GNET	**Version 1.3**
GAPP	**Version 1.3**
PCB	**Version C**

Coactive Aesthetics
P.O. Box 425967
San Francisco, California 94142

Table of Contents

CHAPTER 1 Introduction

The Coactive Aesthetics GCB11 is a complete hardware and software solution for embedded control using the Motorola MC68HC11 microcontroller. Built around the powerful F1 version of the HC11, the GCB11 has been designed specifically to be used in a distributed fashion. With the optional GNET networking software, the GCB11 can be used to implement an RS485 multi-drop network of up to 32 nodes.

1.1 The C Programming Language

The software and documentation shipped with the GCB11 assumes the user has a working knowledge of the C programming language. Although all the GCB11 software functionality is easily accessible from assembly language, the subroutine descriptions and argument passing conventions are given in C syntax. If you are not familiar with C, the following books are good places to start:

Kernighan, Brian W. and Ritchie, Dennis M. (1988). *The C Programming Language, Second Edition*. Englewood Cliffs, New Jersey: Prentice Hall.

Plauger, P.J. (1992). *The Standard C Library*. Englewood Cliffs, New Jersey: Prentice Hall.

1.2 Companion Disk

Included with the GCB11 is an MS-DOS companion disk which contains updates, corrections, include files for macro definitions and function prototypes, software utilities, and examples on how to use the GCB11 hardware and software. Consult the **`README`** file in the root directory of this disk for a complete list of files and information on each of the sample applications and utilities.

1.3 Symbolic Names

Throughout this manual, symbolic names are used in place of memory addresses and values to improve readability. All of these names are C macros or assembler EQU directives found in the include files distributed on the sample disk.

1.4 Chapter Descriptions

CHAPTER 1, Introduction, is this introduction.

CHAPTER 2, Hardware, covers the GCB11 hardware. This chapter lists specifications, discusses crystal speed, power, and communications issues, and describes all the jumpers and connectors and their pin-outs. A schematic and a parts list are also included at the end of the chapter.

CHAPTER 3, GROM, covers the GCB11 ROM, or GROM. GROM encompasses code and data in RAM, EPROM, EEPROM, and in the MC68HC11F1 I/O registers. The elements of GROM include GIOS, GBUG, and GNET, each of which is described in a separate chapter. The primary focus of this chapter is memory usage. The chapter also contains a comprehensive set of figures and tables covering all CPU and GROM resources.

CHAPTER 4, GIOS, covers the GGCB11 I/O Sub-system (GIOS). This includes the numerous I/O streams provided for the GCB11 and the system library. The system library provided in ROM is an ANSI C compatible library which provides functions ranging from printf to string manipulation. An interface description is provided to allow users to call these functions from assembly language programs if desired.

CHAPTER 5, GBUG, covers the ROM monitor/debugger (GBUG) which is provided with the GCB11. This monitor provides a wide variety of functions, from downloading code to setting breakpoints and modifying memory.

CHAPTER 6, GNET, covers the various communications issues related to the GCB11. It describes the methods for communicating with the GCB11. Special emphasis is given to the network communications library (GNET), which allows multiple GCB11s to be connected together and to communicate with each other.

CHAPTER 7, GAPP, describes the GCB11 Application Routines, or GAPP, which supply miscellaneous support functions for applications. These include such items as PWM motor control and event counting.

CHAPTER 8, Building Applications, serves as a general reference for users developing and running applications using GROM.

CHAPTER 9, Programmer's Reference, gives a detailed description of every function available in the GROM except the GIOS Standard C Library functions.

CHAPTER 10, Support, describes how to obtain support for using the GCB11.

CHAPTER 2

Hardware

2.1 Introduction

This chapter describes the GCB11-F1 hardware. It lists specifications, discusses crystal speed, power, and communications issues, and describes all the jumpers and connectors and their pin-outs. A schematic and a parts list are also included at the end of the chapter. Figure 2.1 is a simplified block diagram of the GCB11.

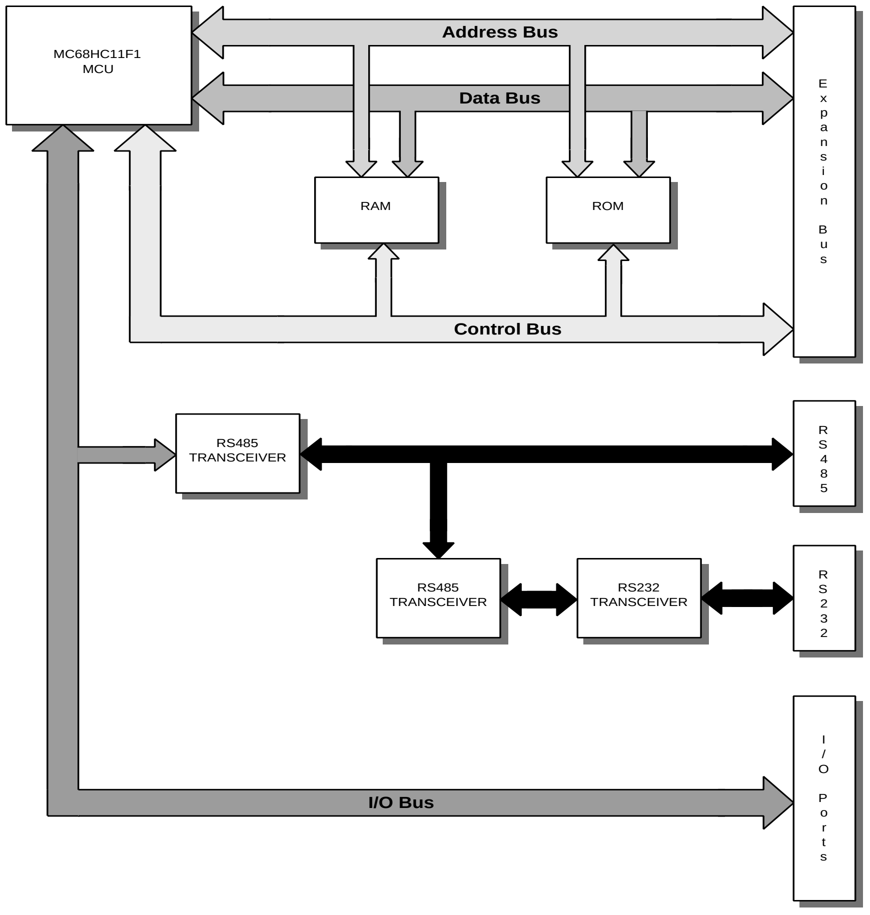


FIGURE 2.1 GCB11 Block Diagram

2.2 Specifications

The GCB11-F1 comes assembled on a 3” by 4” through-hole printed circuit board with socketed MC68HC11F1FN MCU, 32K x 8 bit static RAM, 32K x 8 bit EPROM, a two channel RS485 transceiver, and a four channel 5V only RS232 transceiver (only two channels are used). Features include:

- Motorola MC68HC11F1 microcontroller running in expanded non-multiplexed mode with low-voltage automatic reset circuitry. Refer to the M68HC11 Reference Manual and the MC68HC11F1 Technical Data book for specifications of the MC68HC11F1.
- Single supply +5 volt operation.
- Jumper and socket for a 5 volt voltage regulator.
- 28-pin DIP socket for either 16K x 8 bit or a 32K x 8 bit RAM.
- 28-pin DIP socket jumpered for either 16K x 8 bit or 32K x 8 bit ROM, EPROM, or battery-backed RAM.
- Two serial communication ports connected to the MC68HC11F1’s Serial Communications Interface with jumpers to support master-slave selection of each port independently on a RS485 multi-drop network.
- 4-contact modular jack or 4-pin header interfaced to a RS485 transceiver.
- 3-pin header interfaced to a RS232 transceiver.
- 10-pin header for the MC68HC11F1’s port E (8 x 8 bit A/D with high and low reference or 8 digital inputs).
- 20-pin header for the MC68HC11F1’s ports A, D, and G (up to 20 digital I/O, synchronous and asynchronous communications, input and output capture, and chip select pins).
- 36-pin header for the MC68HC11F1’s ports B, C and F, and other selected signals (address bus, data bus, and control bus).

2.3 Internal Bus Speed

At the time of this writing, the MC68HC11F1FN is available in three speeds:

- MC68HC11F1FN- maximum internal bus speed of 2.1 MHz
- MC68HC11F1FN3 - maximum internal bus speed of 3.0 MHz
- MC68HC11F1FN4 - maximum internal bus speed of 4.0 MHz

Note that the internal bus speed is one quarter of the crystal frequency.

The GCB11-F1 PCB has been tested at bus speeds up to 4.0MHz. The GCB11-F1 is available in two speeds from the factory:

- GCB11-F1, which contains either a MC68HC11F1FN or MC68HC11F1FN4 MCU and a 7.3728 MHz crystal.
- GCB11-F1-14, which contains a MC68HC11F1FN4 MCU and a 14.7456 MHz crystal.

To determine which version you have, just check the crystal speed which is printed on the crystal itself. The crystal speeds were chosen to allow the most flexibility in serial communications speeds, a very important consideration for networking.

Any crystal speed up to the rated speed of the installed MCU will work with the GCB11-F1. Changing the speed of the GCB11 requires the following steps:

- possibly upgrading the MCU unit to a faster version.
- possibly upgrading the memory to faster versions. Refer to the MC68HC11F1 technical specifications for the memory timing.
- replacing the crystal with one of the required value.

Note that changing the crystal speed on the GCB11 can cause it to become incompatible with other GCB11s and PCs in a network configuration because the communication baud rates are derived from the internal bus speed.

2.4 Power

All components on the GCB11 need only +5 volts and ground to operate. As shipped from the factory, the GCB11 accepts regulated +5 volt power at connector P1. If +5 volts is not available, there is space on the board for a 7805 voltage regulator in a TO-220 package (reference designation U10) to allow input power in the range of +7.5-35 volts. To install the 7805, simply solder in the part and configure jumper J12 as detailed in **Section 2.6.2, Power**.

Note that while a 7805 is installed, the GCB11 will no longer run off of a +5 volt supply (the 7805 needs at least +7.5 volts to operate).

2.5 Asynchronous Serial Communications

The GCB11 supports both RS232 and RS485 direct and RS485 multi-drop network asynchronous serial communications. All asynchronous serial communications on the GCB11 are done through the MC68HC11's Serial Communications Interface, or SCI. The SCI is interfaced to RS485 transceiver 1 on the GCB11. The differential signals from this transceiver are connected to RS485 transceiver 2 and to connector JP5. RS485 transceiver 2 changes the differential signals back into TTL, which is interfaced to a RS232 transceiver connected to JP6. This allows any RS232 device connected to JP6 to also transmit and receive RS485 signals on JP5 as illustrated in Figure 2.2 (also consult Figure 2.12, GCB11-F1C Schematic). Figure 2.2 depicts a typical scenario in which an external node is connected to JP6 while JP5 is used for network communications.

At the junction of the two RS485 transceivers and JP5 is an array of eight 3x1 0.1" headers (J1-8) which functions as a network master-slave select switch for the GCB11 (internal node) and for the optional external node connected to JP6 (JP5 is in this case used for network communications). These jumpers determine which of the differential lines are connected to the pins on JP5. This is necessary because of the full duplex nature of the RS485 multi-drop network and the fact that the master node must be configured opposite to the slave nodes. Note that in Figure 2.2 the master-slave select switch (J1-8) has been configured with the external node as the master and the GCB11 as a slave. Refer to Figure 2.1, GCB11 Block Diagram and Figure 2.12, GCB11-F1C Schematic, for details. **Section 2.6.5, Asynchronous Serial Communications** describes how to configure jumpers J1-8 for master or slave operation.

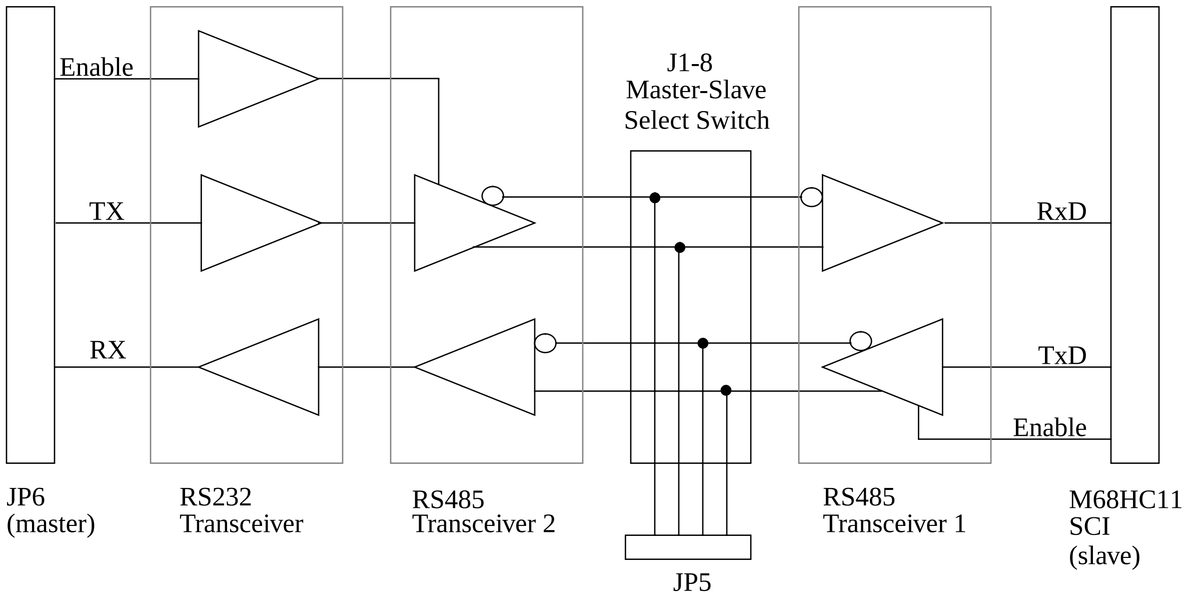
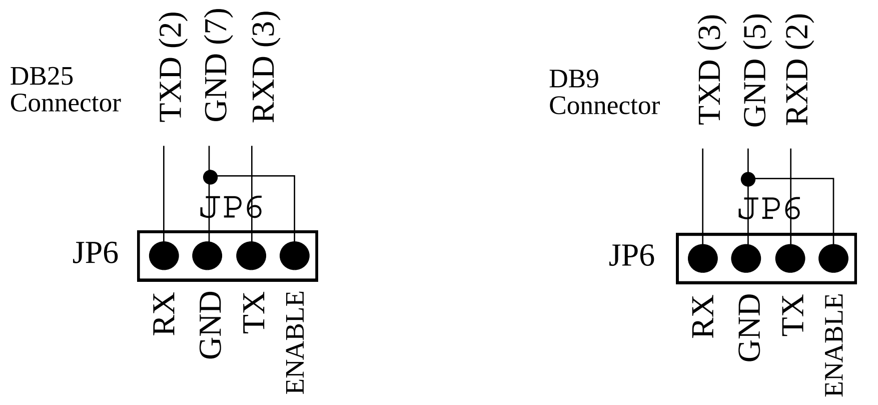


FIGURE 2.2 Typical Communication Transceiver Configuration

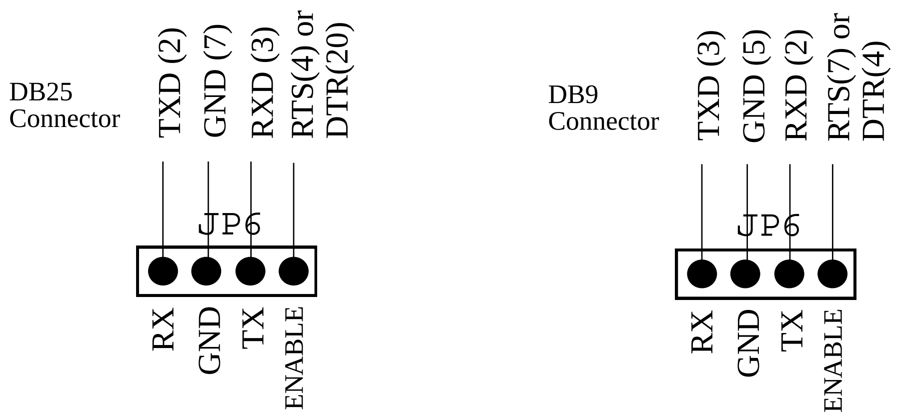
It is important when studying Figure 2.2 to note that all data transmitted and received on JP6 appears both on JP5 and at the M68HC11. Similarly, data transmitted and received on JP5 will appear at both JP6 and at the M68HC11. The user can change the appropriate jumpers among J1-8 to modify how this data is routed through JP5.

It is also important to note the existence of the enable lines on the RS485 drivers. These lines are required in any RS485 multi-drop network. If the RS232 connection at JP6 is intended to be on the multi-drop network through JP5, the enable line must be controlled appropriately by the device at JP6. For an IBM PC, the enable line should be connected to either DTR or RTS, both of which are used by the network drivers on the PC. If the connection through JP6 is intended only for direct communications (i.e. a dumb terminal), and there is nothing connected to JP5, then the enable line can be permanently enabled by connecting it to ground at the connector. Figure 2.3 shows the cabling for JP6 corresponding to the various communications options.

Note that the RS485 specifications support a maximum of 32 drivers and 32 receivers on a multi-drop network. Since each GCB11 contains two RS485 transceivers, to maximize the number of nodes possible one should remove the jumpers for the external node (J1-4) on all GCB11s which do not have an external connection. This disables the second RS485.



(a) Dumb Terminal Connection (no data on JP5).



(b) Network Connection Through JP5.

FIGURE 2.3 RS232 Cabling for JP6

2.6 Connectors and Jumpers

2.6.1 Overview

Figure 2.4 shows the locations of all the connectors and jumpers on the GCB11. The following sections describe each connector or jumper in detail. Note that each connector or jumper shown below is oriented such that the words “COACTIVE AESTHETICS” are in the lower left-hand corner of the board. Also note that the reference designators (labels) are shown in their actual location and orientation relative to the connector or jumper.

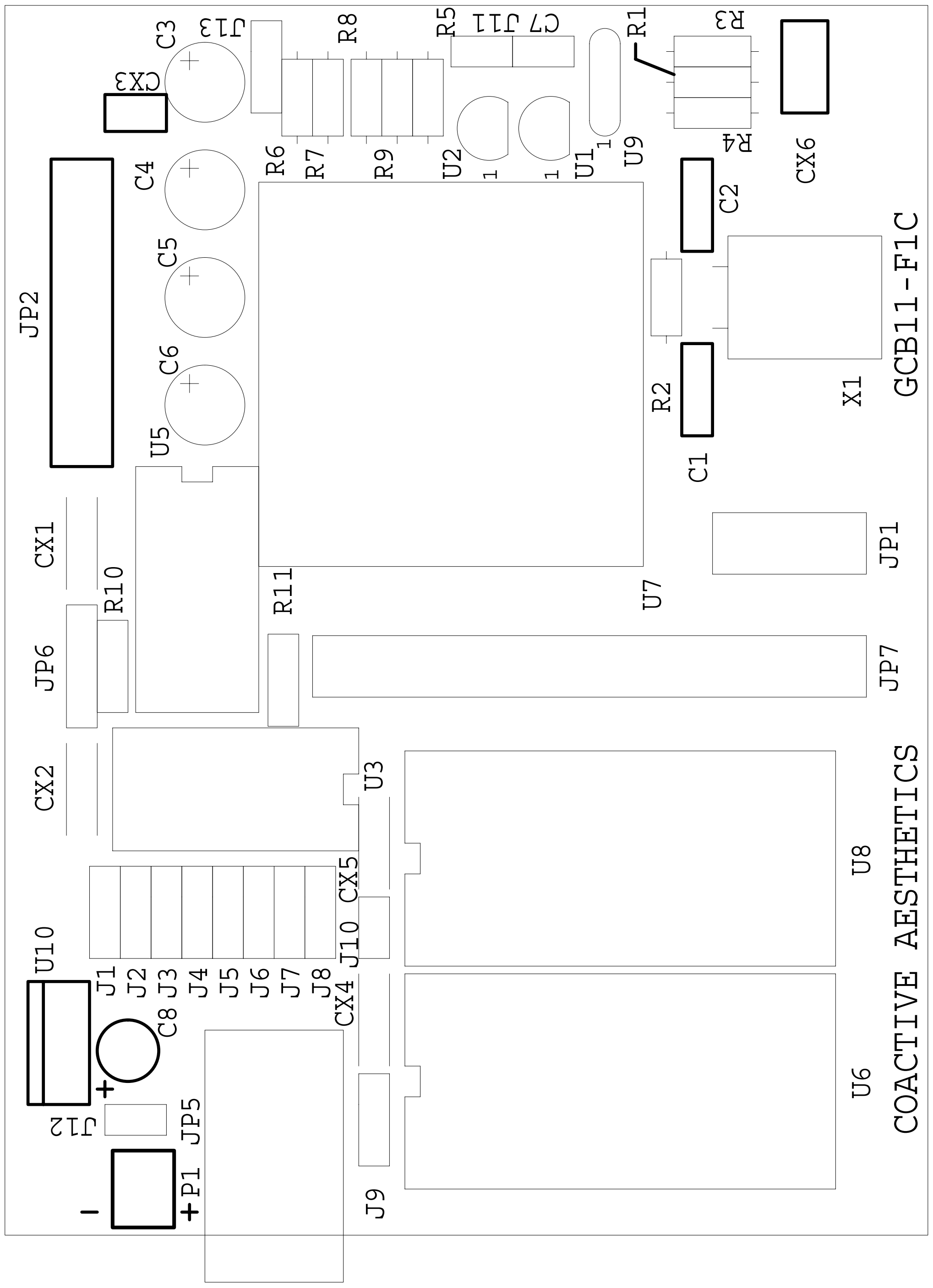


FIGURE 2.4 GCB11 Parts Location

2.6.2 Power

As shipped, the GCB11 is configured to accept +5 volts and ground at the power connector, P1. In this configuration, jumper J12 should be shorted. If the optional 7805 voltage regulator is installed, J12 should be open (see Table 2.1). In either case, P1 is marked with '+' and '-' symbols indicating positive voltage and ground respectively.

Jumper	7805 Not Installed	7805 Installed
J12	short	open

TABLE 2.1 J12: 7805 Jumper Configuration

2.6.3 Reset

The two-pin header J11 can be used to manually reset the GCB11. To reset the board, short the two pins together and then return them to the open state (see Table 2.2). Note that this input is debounced by the automatic reset circuitry.

Jumper	GCB11 Run	GCB11 Reset
J11	open	short

TABLE 2.2 J11: Reset Jumper Configuration

2.6.4 Memory

The GCB11 has two 28-pin dip sockets for memory. U6 can be used for either a 16K x 8 bit or a 32K x 8 bit RAM. U8 can also be either 16K x 8 bit or 32K x 8 bit in size, and in addition, jumpers J9 and J10 allow it to be configured as either RAM or EPROM. See Figure 2.5 for jumper positions.

Note that if U8 contains a RAM, it must be battery-backed to preserve the interrupt vectors.

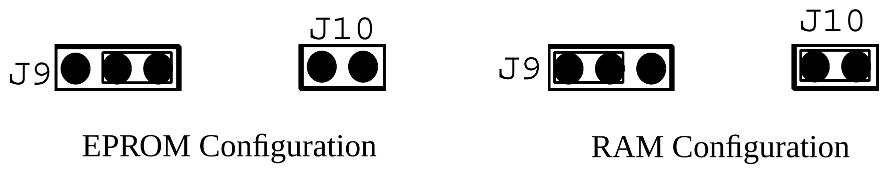


FIGURE 2.5 J9 & J10: U8 Jumper Configuration

2.6.5 Asynchronous Serial Communications

The GCB11 contains two asynchronous serial communications connectors, JP5 and JP6. JP5 is interfaced to two RS485 transceivers: one connected to the MC68HC11F1's SCI, and the other connected to the RS232 transceiver which is in turn connected to JP6 as discussed in **Section 2.5**. JP5 comes shipped from the factory as a 4-position, 4-contact modular jack. This connector can be removed and replaced by a 4x1 0.1" header if desired. JP6 is a 4x1 0.1" header. Figure 2.6 shows the pin-outs for these connectors.

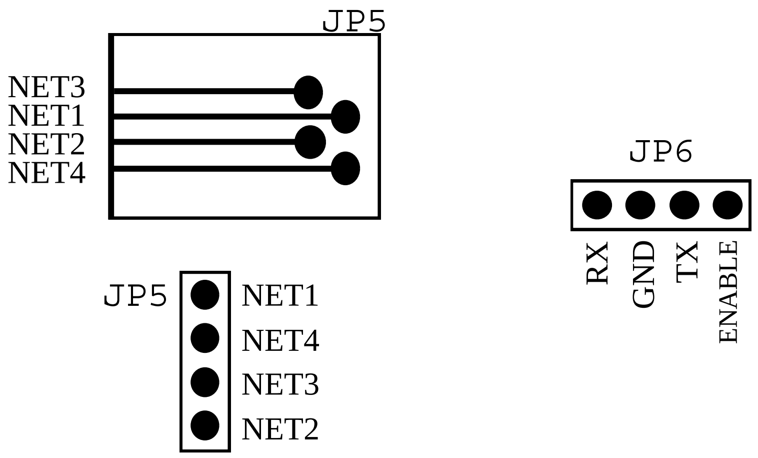


FIGURE 2.6 JP5: RS485 Port Pin-out and JP6: RS232 Port Pin-out

Jumpers J1 through J8 allow the drivers and receivers on the two RS485 transceivers to be interchanged for each channel independently with respect to the pins of connector JP5. When networked, this is used to select the master or slave status of the MC68HC11F1's SCI (internal node) and the external node connected to JP6. In a multi-drop network configuration, only one node on the network can be a master; all other nodes must be slaves.

As described in **Section 2.5**, there are numerous possible configurations for serial communications. Since some external device may be connected to the network via JP6 in addition to the M68HC11 on the GCB11, it is useful to define an external node as one which is connected to the network through JP6, and the internal node as the M68HC11 which is connected directly to JP5. Table 2.3 lists each configuration along with the jumper settings for the internal and external nodes. See Figure 2.7 for a description of how to configure each node as a master or a slave using jumpers J1 through J8.

Configuration	Internal Node (J5 - J8)	External Node (J1 - J4)
SCI to JP5 direct	master or slave	don't care
SCI to JP6 direct	master or slave	opposite of internal node
SCI to JP5 networked	master or slave	don't care
SCI to JP5 networked JP6 to JP5 networked	master or slave	master or slave
JP6 to JP5 networked	don't care	master or slave

TABLE 2.3 Serial Communication Configurations

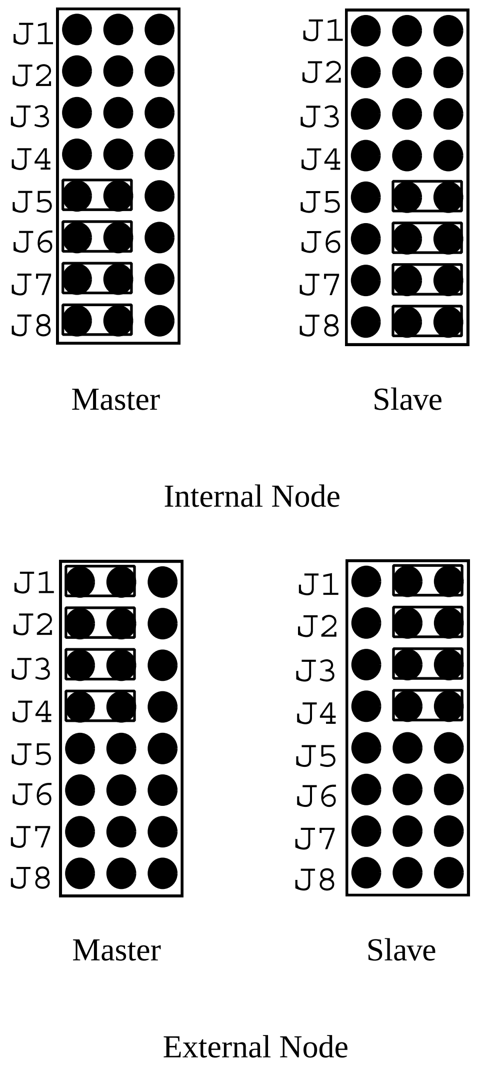


FIGURE 2.7 J1-J8: Communications Jumpers

RS485 Driver Enable Jumper

Jumper J13 is also used by the communications subsystem. It is used to determine which of two possible pins on the M68HC11 are used to enable the RS485 drivers. Currently either PG2 or PD2 can be used as an enable line. There is a pull-up resistor on the enable line so that if the jumper is left off, the RS485 driver will be permanently enabled. This can be done if there is no multi-drop network communications and there is simply a dumb terminal connected to the GCB11 through JP6. Figure 2.8 shows the possible configurations for this jumper.

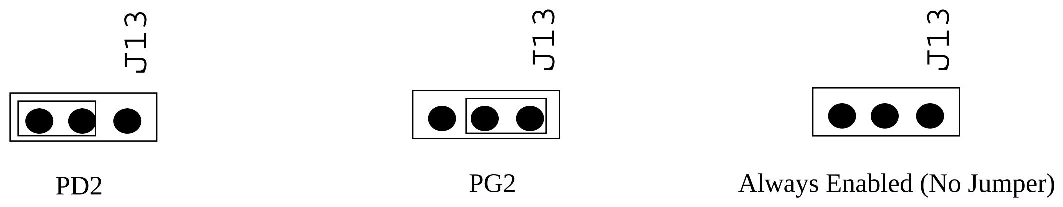


FIGURE 2.8 J13: RS485 Driver Enable Jumper

2.6.6 I/O Ports

The MC68HC11F1 ports A, D, and G are accessible through JP2. See Figure 2.9 for a pin-out of this connector. Note that if the SCI is being used, pins PD0 and PD1 are not available. This is also true for pins PG4 and PG5 if the I/O chip selects are in use.

WARNING: the user needs to insure that PD0 and PD1 do not accidentally become connected to user hardware on JP2 if the SCI is in use. Doing so may damage PD1. If the SCI is disabled there is no risk involved.

The MC68HC11F1 port E is accessible through JP1 along with the high and low reference voltages for the A/D system. See Figure 2.10 for a pin-out of this connector.

Special Bootstrap Jumper

Pins TXD/PD1 and RXD/PDO of JP2 form a special bootstrap jumper for GROM. Upon start-up GROM does a short loop-back test to see if these two pins are shorted. If so, default communications and start-up values are loaded from ROM into EEPROM. This is useful if the EEPROM becomes corrupted or the application accidentally reprograms it in such a manner that console I/O is lost. If it becomes necessary to use this special bootstrap jumper, the GCB11 should be removed from all connectors and isolated from the network. To use it, simply short PD1 and PD0 and then reset the GCB11. Note that this will only work if the firmware on the board supports this operation (the EPROM shipped with the GCB11 does). See **Section 3.6, GCB Registers** and **Section 4.2, GCB Registers** for more details.

Care should be taken when connecting user hardware to PD1, since this loop back test is always done at start-up from GROM. See **Section 2.6.6, I/O Ports** for additional warnings concerning PD1 and the SCI.

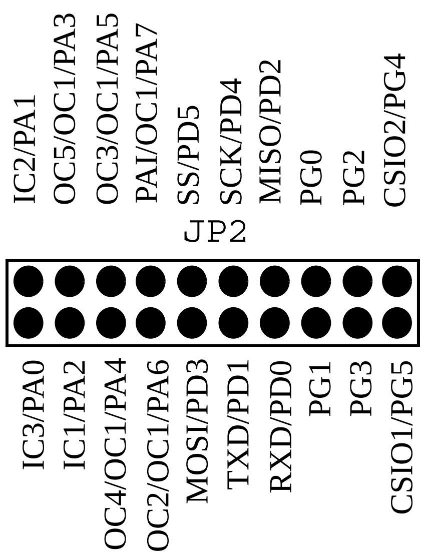


FIGURE 2.9 JP2: Multipurpose Digital I/O Port

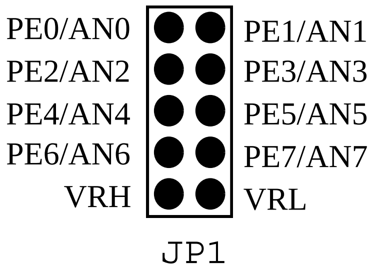


FIGURE 2.10 JP1: Analog Input Port

2.6.7 Expansion Buses

MC68HC11F1 Ports B, D, and F (address and data lines) as well as many other control signals are accessible through JP7, the GCB11 expansion bus. See Figure 2.11 for a pin-out of this connector. With the exception of $\overline{\text{READ}}$ and $\overline{\text{WRITE}}$, all signals on the expansion bus are described in the MC68HC11F1 Technical Data book.

$\overline{\text{WRITE}}$ - the R/$\overline{\text{W}}$ control signal from the MC68HC11F1.

$\overline{\text{READ}}$ - an inverted R/$\overline{\text{W}}$ signal (W/$\overline{\text{R}}$) which is created by U9 (see Figure 2.12).

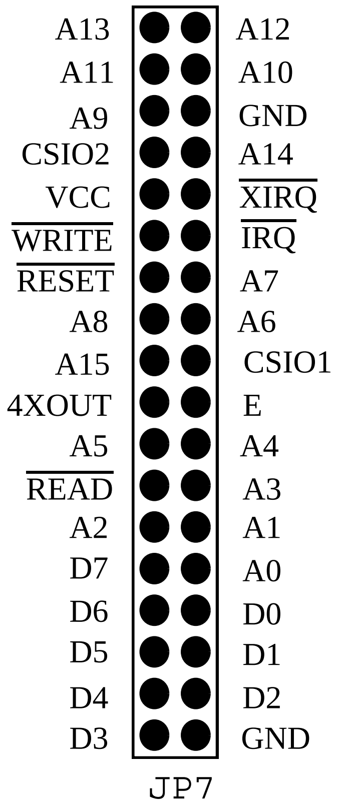


FIGURE 2.11 JP7: Expansion Bus

2.7 Tables and Drawings

Reference Designation	Description
C1-2	18 pF @ 100V monolithic
C3-6	0.1 uF @ 50V electrolytic
C7	1.0 uF @ 35V tantalum
C8	33uF @ 35V electrolytic
CX3	0.01 uF @ 63V monolithic
CX1,2,4,5	0.1 uF @ 63V monolithic
CX6	1.0 uF @ 63V monolithic
J9-13, JP6	single row 0.1” header
JP1,2,7,J1-8	double row 0.1” header
JP5	4 position - 4 contact modular jack
P1	0.1” friction lock connector
R1,3-11	4.7K Ohm, 5%, 1/8 W
R2	10M Ohm, 5%, 1/8 W
U1	MC34164P-5 undervoltage sensing circuit
U2	MC34064P-5 undervoltage sensing circuit
U3	SN751178N RS485 differential transceiver
U5	MAX232ACPE 5V RS232 transceiver
U6	32K x 8 bit static RAM
U8	32K x 8 bit EPROM
U7	MC68HC11F1FN
U9	DTC114EA digital transistor
U10	LM7805 Voltage Regulator
UX3, 5	16-pin DIP socket
UX6,8	28-pin DIP socket
UX7	68-pin PLCC socket
X1	AT-cut crystal

TABLE 2.4 GCB11-F1 Parts List

Notes:

- speed of U7 can be 2, 3, or 4 MHz
- speed of U6 and U8 depends on speed of U7
- X1 frequency can vary and is limited by speed of U7

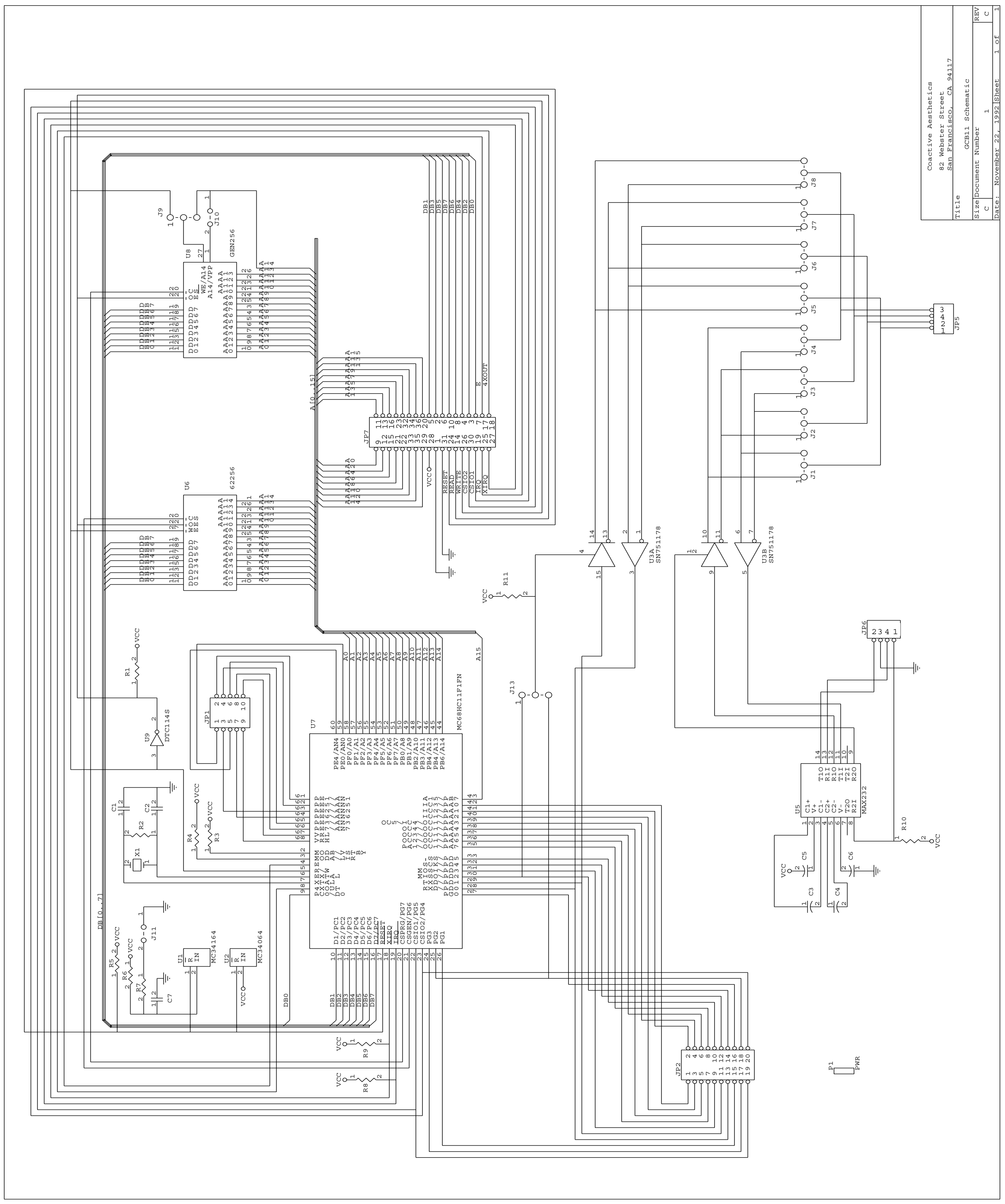


FIGURE 2.12 GCB11-F1 Schematic

CHAPTER 3 GROM

3.1 Introduction

This chapter discusses the GCB11 ROM, or GROM. GROM encompasses code and data in RAM, EPROM, EEPROM, and in the MC68HC11F1 I/O Registers. This primary focus of the chapter is on memory usage. It also contains a comprehensive set of figures and tables covering all CPU and GROM resources. While this chapter touches on every element of GROM, many topics are left to be discussed in detail in later chapters.

3.2 Memory Map

Figure 3.1 shows the memory map for the GCB11 running GROM. The column on the left shows which memory locations are selected by the MC68HC11F1 chip selects and config registers. The column on the right represents the type of memory available at each location. Many of the values shown in the figure are defined as macros in **`gcb11.h`** and **`gcb11.inc`**. Since GROM can be configured in a number of different ways, the end of free RAM and the start of EPROM are indicated by a "????". See the **`readme`** file on the companion disk for the exact values for each configuration. GROM does not use the MC68HC11F1 I/O chip selects.

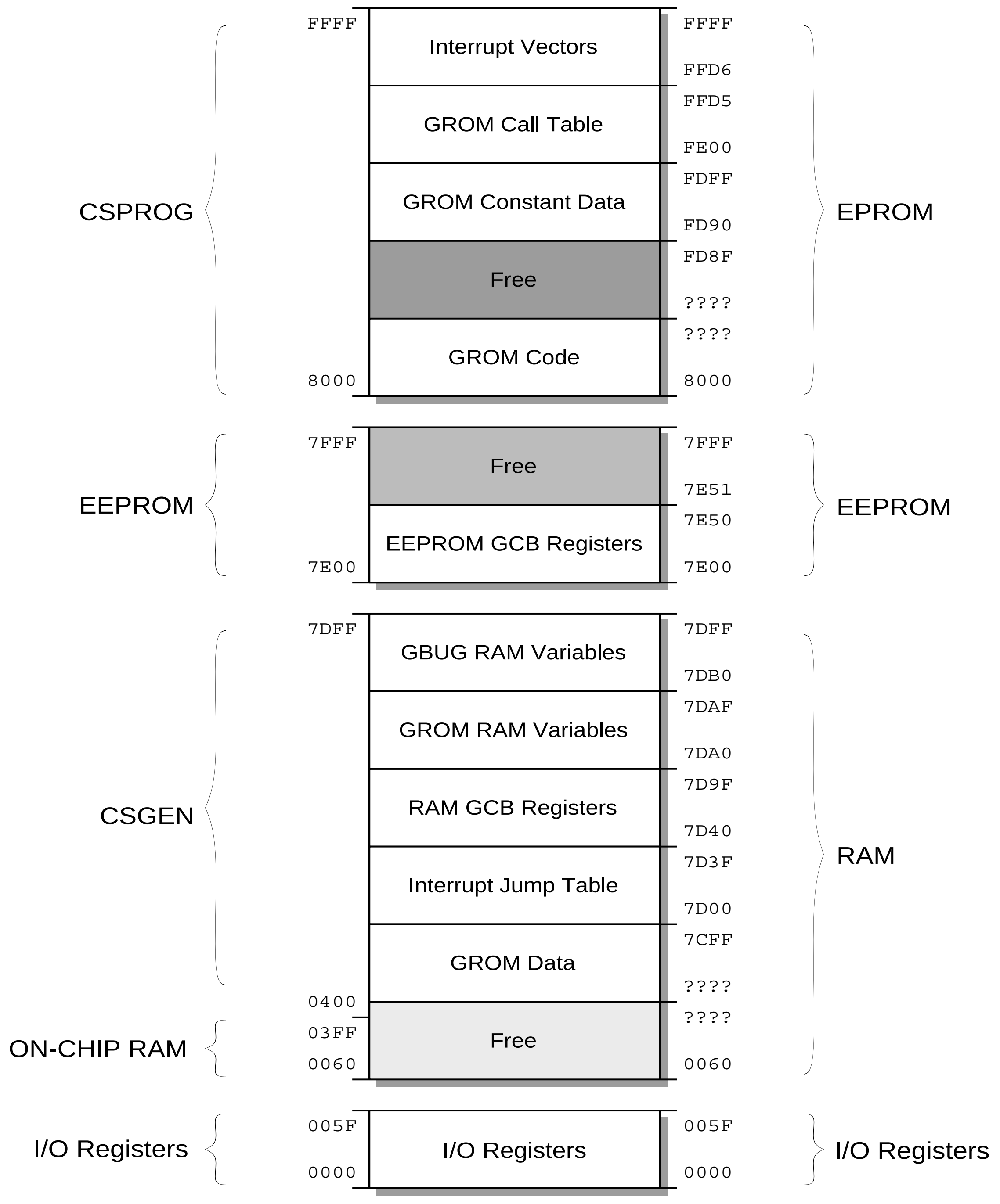


FIGURE 3.1 GROM Memory Map

3.3 M68HC11 I/O Registers

The term *I/O Register* is used here to signify any MC68HC11F1 status, control, or I/O register. GROM makes extensive use of the I/O Registers, both at reset for configuring the hardware and during run-time when using on-chip resources. The user should be aware of which Registers are being used and should be careful not to disturb them. See **CHAPTER 8, Building Applications**, for more details on developing applications on the GCB11.

After reset, GROM changes some of the MC68HC11F1 I/O Registers to configure the memory map as described in **Section 3.2**. This is done by GBUG if it is installed, or by user start-up code. See **Section 3.11, Start-up** and **CHAPTER 8, Building Applications** for more details on start-up code. These Registers should not be changed at any time and are listed in Table 3.1.

I/O Register	Value
CONFIG	7F
INIT	00
BPROT	00
CSGSIZ	01
CSGADR	00
CSCTL	05

TABLE 3.1 I/O Register Values Required by GROM

3.4 RAM Data

All RAM used by GROM is in a block at the top of RAM.

3.5 Interrupts

In addition to the M68HC11 interrupt vectors located at the top of memory (locations FFC0 - FFFF), GROM provides a RAM interrupt jump table located at 0100. This jump table must be present for any of the GROM code modules to function. The jump table contains 20 extended addressing mode JMP instructions, one for each interrupt vector, with the exception of the reset (FFFE) and the reserved vectors. The reset vector is hard-coded to point to start-up code when a ROM is made (see **CHAPTER 8, Building Applications**, for more details on the reset vector). Each interrupt vector points to one of the jump instructions: FFD6 points to 0100, FFD8 points to 0103, etc. In this way, the locations of interrupt service routines can be changed by changing the extended addresses in the jump table. See **CHAPTER 8, Building Applications**, for more details on developing applications on the GCB11.

Upon start-up, each vector in the interrupt jump table is initialized to point to a dummy interrupt service routine which has a single RTI. When GROM initializes the code modules, each module replaces the vectors it uses to point to an interrupt service routine. See the following chapters on each of the code modules for more details.

3.6 GCB Registers

GCB Registers are variables used by GROM which are preserved in EEPROM through power-cycles. These variables are normally read from EEPROM into RAM during start-up and can be updated at any time to preserve changes. There are actually three copies of the GCB Registers in the system during run-time: one in RAM, one in EEPROM, and one in EPROM. The locations of the EEPROM and RAM copies are listed in **Section 3.2, Memory Map**. The EPROM GCB Registers are stored in GROM constant data (see **Section 3.8**).

GIOS provides functions for managing each of the GCB Register copies. Reading and writing the actual values in the Registers is left to the individual code modules or user applications; GROM only sets aside space for them in memory. All access by user applications should be to the Registers in RAM. **Section 3.11, Start-up**, discusses how the EEPROM copy is updated.

Table 3.2 lists each GCB Register and the code module which defines it. The offset is relative to the start of the GCB Registers in RAM, EEPROM, and EPROM. Refer to the chapter on each code module for more details. The first three Registers are also discussed in **Section 3.11, Start-up**.

GCB Register	Offset	Size	Module
SCIBAUD	00	1	GIOS, GNET
SCIENABLE	02	1	GIOS, GNET
SCISTART	03	1	GIOS, GNET
SPIBAUD	04	1	GIOS
SPIMODE	05	1	GIOS
SPITIMER	06	1	GIOS
GBUGSTRM	09	1	GBUG
GBUGRR	0A	2	GBUG
GNETCOUNT	0C	1	GNET
GNETMODE	0D	1	GNET
GNETNODE	0E	1	GNET
GNETPOLL	0F	1	GNET
GNETTABLE	10	64	GNET

TABLE 3.2 GCB Registers

3.7 GAPP, GBUG, GIOS, and GNET

The GCB11 comes shipped with an EPROM which contains all four GROM code modules:

- GCB11 Applications, or GAPP
- GCB11 Monitor/Debugger, or GBUG
- GCB11 I/O System, or GIOS
- GCB11 Network Software, or GNET (as of version 1.3, GNET is optional)

Each of these modules is described in detail in the following chapters. When building application EPROMs, the user can choose which of the GROM code modules are needed, and include only those in the new EPROM. See

CHAPTER 8, Building Applications, for more details. Note that GIOS is needed to configure the system and to use any of the other code modules.

3.8 Constant Data

GROM contains constant data in EPROM which the user can use directly. Since application programs are not linked directly with GROM, this data is stored at a fixed location in memory. Currently, there are three types of data stored in the GROM constant data:

- Default values for the GCB Registers.
- Pointer to the built-in Stream driver configuration tables.
- Constants which depend on the crystal speed of the GCB11.

3.8.1 GCB Registers

The default values for each of the GCB Registers is stored here. See **Section 3.11.3, GCB Registers**, for more details. The base address for the Registers is FD90. Use the offsets from **TABLE 3.2, GCB Registers** for the exact location of each register.

3.8.2 Stream Driver Configuration Tables

GROM provides three Stream drivers which the user can access using GIOS (see **Section 4.3.3**). The driver configuration tables for these are located in GROM Constant Data. Table 3.3 lists the addresses of these configuration tables.

Stream Driver	Address of Pointer to Configuration Table
SD_SCI	FDE0
SD_SPI	FDE2
SD_GNET	FDE4

TABLE 3.3 GROM Stream Driver Configuration Table Addresses

3.8.3 Crystal Speed Dependent Constants

GROM depends on several constants which change with the GCB11 crystal speed. The user may access these constants as well. Table 3.4 lists the addresses and sizes of each of these constants.

The SCI baud rates are used by GIOS and GNET and are described in more detail in **Section 3.11.4.1**.

The EEPROM_DELAY constant is used in programming the MC68HC11 EEPROM and is described in **Section 4.4**.

Constant	Address	Size
SCI_BAUD_38400	FDF0	1
SCI_BAUD_19200	FDF1	1
SCI_BAUD_14400	FDF2	1
SCI_BAUD_9600	FDF3	1
SCI_BAUD_4800	FDF4	1
SCI_BAUD_2400	FDF5	1
SCI_BAUD_1200	FDF6	1
SCI_BAUD_600	FDF7	1
SCI_BAUD_300	FDF8	1
EEPROM_DELAY	FDF9	2

TABLE 3.4 Crystal Speed Dependent Constants

3.9 Call Table

The main interface to GAPP, GIOS, and GNET is through function calls. Since the exact location of GROM functions can change depending on which code modules are present in the EPROM, all GROM functions are accessed through a call table which is fixed in memory just under the interrupt vectors (see **Section 3.2, Memory Map**, above). The call table contains a JMP instruction to the beginning of each function. User code should call GROM functions by executing a JSR instruction to the entry in the call table for the function they wish to use. The tables at the end of this section list the addresses in the call table for each GROM function. These addresses will never change, even in future releases of GROM.

All GROM functions follow the same calling conventions. Although the function descriptions in **CHAPTER 9, Programmer’s Reference**, are listed in C, the method by which arguments are passed to them is defined in assembly language. To call the functions from C, a wrapper function must be written for each function and linked to the application. These wrapper functions should take the arguments from the C call, reformat them for GROM, and perform a JSR instruction to the proper call table address.

Based on the ANSI C syntax used to describe the functions in the man pages, arguments should be passed using the following rules:

- there is no padding or alignment required, so only the minimum number of bytes needed to represent the type should be used.
- arguments are pushed on the stack in reverse order, with the exception of the first argument.
- if the first argument is a single byte type, the argument is placed in register B and register A is set to zero.
- if the first argument is a two byte type, it is placed in the D register.

Functions return values in the following manner:

- if the return value is a single byte type, it is returned in B.
- if the return value is a two byte type, is returned in D.

Table 3.5 lists the sizes for each ANCI C type specifier used by the GROM functions. Note that in all multi-byte types, the most significant bytes are in the lowest memory addresses.

Table 3.6 through Table 3.12 list all the GROM call table functions and their addresses. They are grouped alphabetically by include file. See the **`readme`** file on the companion disk for a complete list of include files.

Type	Size in Bytes
char	one
unsigned char	one
short	two
unsigned short	two
int	two
unsigned int	two
pointer	two

TABLE 3.5 Byte Size of GROM C Types

Function	Address
isalnum	FE03
isalpha	FE06
iscntrl	FE09
isdigit	FE0C
isgraph	FE0F
islower	FE12
isprint	FE15
ispunct	FE18
isspace	FE1B
isupper	FE1E
isxdigit	FE21
toupper	FE24
tolower	FE27

TABLE 3.6 ctype.h Functions Call Table

Function	Address
ga_get_counter	FE2A
ga_init_motor	FE2D
ga_motor_speed	FE30
ga_motor_dir	FE33
ga_motor_reverse	FE36
ga_motor_faster	FE39
ga_motor_record	FEC2
ga_motor_slower	FE3C
ga_motor_state	FEC5
ga_setup_motors	FE3F
ga_start_counter	FE42
ga_stop_counter	FE45
ga_set_counter	FE48

TABLE 3.7 gapp.h Functions Call Table

Function	Address
gi_gcb_checksum	FE4B
gi_gcb_etoee	FE4E
gi_gcb_eetor	FE51
gi_gcb_loopback	FE54
gi_drvr_close	FE57
gi_drvr_init	FE5A
gi_drvr_ioctl	FE5D
gi_drvr_open	FE60
gi_eeprom_erase	FE63
gi_eeprom_erase_bulk	FE66
gi_eeprom_erase_row	FE69
gi_eeprom_write	FE6C
gi_intr_disable	FE6F
gi_intr_dummy	FE72
gi_intr_enable	FE75
gi_intr_rti	FE78
gi_intr_swi	FE7B
gi_sci_free	FF83
gi_sci_reinit	FF86
gi_strm_attach	FE7E
gi_strm_detach	FE81
gi_strm_flush	FE84
gi_strm_getchar	FE87
gi_strm_gets	FE8A
gi_strm_init	FE8D
gi_strm_input	FE90
gi_strm_output	FE93
gi_strm_printf	FE96
gi_strm_pull	FE99
gi_strm_push	FE9C
gi_strm_putchar	FE9F
gi_strm_puts	FEA2
gi_strm_size	FEA5
gi_strm_stdio	FEA8

TABLE 3.8 gios.h Functions Call Table

Function	Address
gn_add_poll	FEAB
gn_auto_poll	FEAE
gn_close_port_l0	FEB1
gn_close_port_l1	FEB4
gn_config_node	FEB7
gn_exit_comms	FEBA
gn_get_config	FEBD
gn_get_message_l1	FEC0
gn_get_poll	FEC3
gn_get_stats	FEC6
gn_init_comms	FEC9
gn_message_rdy_l1	FECC
gn_open_port_l0	FECF
gn_open_port_l1	FED2
gn_reset_net	FED5
gn_restore_config	FED8
gn_save_config	FEDB
gn_send_message_l0	FEDE
gn_send_message_l1	FEE1
gn_set_poll	FEE4
gn_start_poll	FEE7
gn_suspend_poll	FEEA

TABLE 3.9 gnet.h Functions Call Table

Function	Address
getchar	FF50
gets	FF53
printf	FF56
putchar	FF59
puts	FF5C
sprintf	FF5F
vprintf	FF62
vsprintf	FF65

TABLE 3.10 stdio.h Functions Call Table

Function	Address
abs	FF68
atoi	FF6B
div	FF71

TABLE 3.11 stdlib.h Functions Call Table

Function	Address
memchr	FF89
memcmp	FF8C
memmove	FF8F
memcpy	FF92
memset	FF95
strcat	FF98
strchr	FF9B
strcmp	FF9E
strcpy	FFA1
strcspn	FFA4
strlen	FFA7
strncat	FFAA
strncmp	FFAD
strncpy	FFB0
strpbrk	FFB3
strrchr	FFB6
strspn	FFB9
strstr	FFBC
strtok	FFBF

TABLE 3.12 string.h Functions Call Table

3.10 Stack Requirements

Each GROM code module requires stack space to operate. This stack space is taken from the application stack when a GROM function call is made. In addition, any interrupts which occur use the application stack for storing the interrupt stack frame and for running the interrupt service routine. Table 3.13 lists the stack requirements of each GROM code module with the exception of GBUG which allocates its own stack during start-up (this memory is taken from GROM RAM data, see **Section 3.2**). Remember to add in the requirements of the application code when computing the stack size.

Module	Required Stack Size
GIOS	32
GIOS	256
GNET	32

TABLE 3.13 Stack Requirements of GROM Code Modules.

3.11 Start-up

GROM requires that certain initialization procedures occur during start-up.

- Initialize MC68HC11 I/O Registers
- Set up stack pointer
- Set up GCB Registers
- Initialize communications

Each of these steps is described in the following sections. Example initialization code, described in **Section 8.6, Initialization**, is supplied on the companion disk.

3.11.1 MC68HC11 I/O Registers

Table 3.1 in **Section 3.3** describes the I/O Register values required for GROM to operate. See the example in **Section 8.6, Initialization** for the recommended order of initialization. Note that some of these Registers must be changed in the first 64 E cycles of the MC68HC11. See the M68HC11 Reference Manual for more details.

3.11.2 Stack Pointer

The MC68HC11 stack pointer, or `SP` register, must be initialized before any JSR calls are made. **Section 3.10** gives the required stack size for each code module.

3.11.3 GCB Registers

As described in **Section 3.6**, GROM contains three copies of the GCB Registers. Each copy is described below:

RAM - this copy is the one actually used for reads and writes during run-time. The values must be copied from EEPROM to RAM before being used by calling the `gi_gcb_eetor` function. If GBUG is installed, this is handled automatically. The function can be called at any time to reset the RAM GCB Registers to their EEPROM defaults.

EEPROM - this copy is used as the default at reset. Because it is located in EEPROM, the values are preserved through power-cycles. These values can be written from the EPROM copy using the GIOS function `gi_gcb_etoee`. The EEPROM Registers are updated from RAM individually by each code module.

EPROM - these are the factory defaults and cannot be changed.

Because the GCB Registers are responsible for configuring most of the I/O on the GCB11, if they become corrupted it can become impossible to communicate with the board by any means. To recover from such a situation, two functions have been included in GIOS for detecting the need for the GCB Registers to be restored to their factory defaults using **`gi_gcb_etoee`**. These functions should be called in the initialization code of any custom ROM.

- Each time the EEPROM copy of the GCB Registers is written, a checksum is computed and stored in EEPROM immediately following the last Register using the GIOS function **`gi_gcb_checksum`**. **`gi_gcb_checksum`** may be used to recompute the checksum and compare it to the value stored during the last write. If the checksums do not match, the function returns **`FAILURE`**.
- **`gi_gcb_loopback`** performs a loop-back test on pins PD0 and PD1 of JP2 (see **Section 2.6.6** for the location of these pins). If all connectors are removed from the GCB11 and these two pins are shorted together during the **`gi_gcb_loopback`** call, the function returns **`FAILURE`**.

If **`gi_gcb_checksum`** or **`gi_gcb_loopback`** returns **`FAILURE`**, **`gi_gcb_etoee`** should be called. GBUG does this in its initialization code.

3.11.4 Communications

The GCB11 has several communication drivers which may be configured to initialize on start-up. The configurations for these drivers are stored in three GCB Registers:

- **`SCIBAUD`**
- **`SCIENABLE`**
- **`SCISTART`**
- **`SPIMODE, SPITIMER, and SPIBAUD`**

See **Section 3.6** for more details on GCB Registers. Note that the start-up code should check these values and perform the necessary initialization. See the example initialization code on the companion disk, described in **Section 8.6, Initialization**, for more details

There are also two GROM ram variables which affect communications:

- **`SCIDRIVER`**
- **`STDIO`**

Each of these Registers is described in the following sections.

3.11.4.1 SCIBAUD

The **`SCIBAUD`** GCB Register contains the baud rate for the MC68HC11 SCI. This baud rate is used for both the GNET network driver and the SCI and GNET Stream drivers. The values stored in this Register can be converted to the actual values programmed into the BUAD I/O register by using the **`SCI_BAUD()`** macro in **`gios.h.`** Since the baud rate is dependent upon the crystal speed of the GCB11, **`SCI_BAUD()`** uses data stored in GROM constant data to compute the register value. See **Section 3.8.3** for the exact location of each baud rate constant.

The possible values for **`SCIBAUD`** are:

- **`BAUD_38400 (0)`**
- **`BAUD_19200 (1)`**
- **`BAUD_14400 (2)`**
- **`BAUD_9600 (3)`**
- **`BAUD_4800 (4)`**

- **`BAUD_2400 (5)`**
- **`BAUD_1200 (6)`**
- **`BAUD_600 (7)`**
- **`BAUD_300 (8)`**

3.11.4.2 SCIENABLE

This parameter determines which of the MC68HC11 pins PG2 or PD2 are used to enable the RS485 drivers. This Register may take on the following values:

- **`NO_ENABLE (0)`**
- **`PG2_ENABLE (2)`**
- **`PD2_ENABLE (8)`**

As defined in **`gcb11.h`** and **`gcb11.inc`**, **`PG2_ENABLE`** and **`PD2_ENABLE`** are the actual addresses of the G and D Ports on the MC68HC11.

3.11.4.3 SCISTART

This Register is used to specify which device driver, if any, is configured on the SCI at start-up. The possible values for this Register are:

- **`SCI_START_NONE`** (0) - no SCI initialization
- **`SCI_START_TERM`** (1) - initialize the SCI Stream driver
- **`SCI_START_GNET`** (2) - initialize GNET
- **`SCI_START_GNET_STRM`** (3) - initialize GNET and the GNET Stream driver.

3.11.4.4 SPIMODE, SPITIMER, and SPIBAUD

These registers control the SPI Stream driver. These are described in detail in **Section 4.3.3.2, SPI Driver**.

3.11.4.5 SCIDRIVER

This GROM RAM variable determines which driver currently has control of the SCI. It should not be set by the user, but should be read to insure that it is **`NO_DRIVER`** (0) before programming the SCI from the application program.

3.11.4.6 STDIO

The STDIO GROM RAM variable holds the address of the default GIOS Stream set by **`gi_strm_stdio`**. See section **Section 4.3.1.3, Level 1 I/O** for more details.

CHAPTER 4 GIOS

4.1 Introduction

GIOS is a code module in GROM used for accessing and controlling the various components of the GCB11 system. It contains functions for using GCB Registers, Streams, and EEPROM. GIOS also includes a subset of the Standard C Library functions.

GIOS functions can be divided into the following categories:

- GCB Register functions - routines for updating and validating GCB Registers.
- GIOS Stream functions - routines for sending and receiving characters through any number of devices on the GCB11.
- EEPROM functions - routines to write and erase EEPROM.
- Interrupt Utilities - routines for enabling, disabling, and returning from interrupts.
- Standard Library functions - a subset of the ANSI Standard C Library functions.

All GIOS functions are called through the GROM call table described in **Section 3.9**.

4.2 GCB Registers

GCB Registers are variables used by GROM which are preserved in EEPROM through power-cycles. A complete description of GCB Registers in given in **Section 3.6**. GIOS supplies four functions for operating on GCB Registers:

`gi_gcb_checksum` - computes and writes the checksum of the EEPROM GCB Registers. Optionally compares this to the previous value stored in EEPROM and returns **`FAILURE`** if it is different.

`gi_gcb_eetor` -copies the EEPROM GCB Registers to RAM.

`gi_gcb_etoee` - copies the factory default EPROM GCB Registers to EEPROM. The Registers are checksummed and this value is written to EEPROM immediately following the last Register.

`gi_gcb_loopback` - performs a loop-back test between pins PD0 and PD1 on JP2 (see **Section 2.6.6, I/O Ports**, for details on this connector). If the pins are shorted together, it returns **`FAILURE`**.

4.3 GIOS Streams

GIOS Streams allow programs to transfer ASCII data to and from the GCB11 in a consistent manner independent of the hardware used for the physical communication. Similar to the concept of data streams in C, GIOS Streams can be opened on communication device drivers and then written to and read from using functions like **`getchar()`**, **`putchar()`**, and **`printf()`**. The device drivers used by GIOS Streams are called Stream drivers. Figure 4.1 shows how Streams and Stream drivers are layered between an application and the GCB11 hardware. GROM supplies Stream drivers for the SCI, SPI, and GNET which are described below in **Section 4.3.3, GROM Stream Drivers**. In addition, GIOS supplies functions which allow users to create their own Stream drivers.

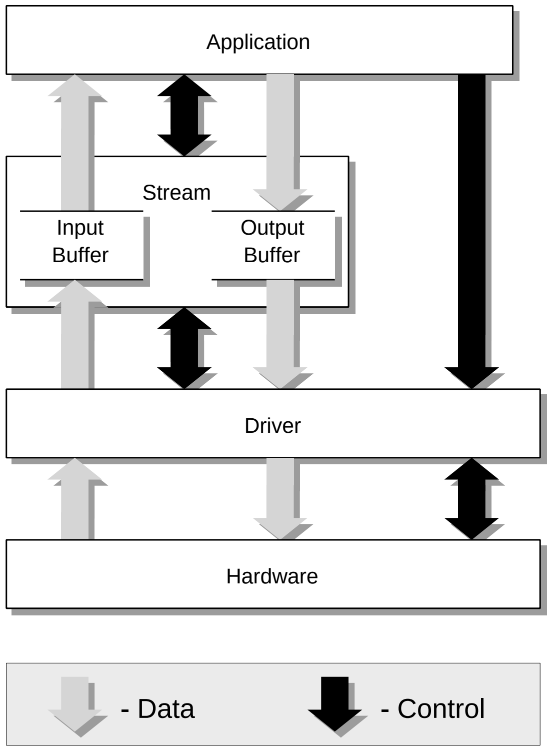


FIGURE 4.1 GIOS Streams

GIOS Streams and Stream drivers can be initialized and opened independently. GIOS supplies functions which attach and detach Streams and Stream drivers dynamically. When a Stream is opened, it is not attached to any Stream driver; all output sent to the Stream is stored in internal buffers and no input is possible. When a Stream driver is opened, it is not attached to any Stream; all input received by the driver is thrown away, and no output is possible. GIOS acts like a switching station between Streams and drivers, connecting any Stream to any driver at the application program's request. This allows programs like GBUG to provide I/O with a number of hardware devices without changing its high-level interface.

The interface between a Stream and its driver is accomplished by sharing two data buffers: an input buffer and an output buffer. These buffers belong to the Stream, and all application I/O on the Stream is actually performed with these buffers. Reads on the Stream take data from the Stream's input buffer. Writes on the Stream put characters in the Stream's output buffer. When a driver is attached to the Stream, the driver can call functions which do reads and writes on these buffers in an inverse fashion relative to the application. A driver writes to a Stream's input buffer and reads from a Stream's output buffer.

GIOS allows applications on the GCB11 to designate a Stream as the default Stream for all standard I/O. Similar to the way UNIX uses **`stdin`** and **`stdout`**, this default Stream is used when functions such as **`printf()`** and **`getchar()`** are used.

GIOS Streams also support both software and hardware flow control. Software flow control is built into the Stream functions. Hardware flow control is handled by the Stream drivers.

4.3.1 Stream Functions

This section describes the functions supplied by GIOS for using Streams. These functions can be divided into four categories, each of which is described in a section below:

- Initialization - initialize, open, close, attach, and detach.
- Level 0 I/O - read, write, flush, and return the size.
- Level 1 I/O - read and write formatted characters and strings.
- Driver Calls - functions for use by Stream drivers.

In addition to the following sections and the man pages in **CHAPTER 9, Programmer's Reference**, users should consult the companion disk, which contains a complete example application that makes use of multiple Streams and Stream drivers.

4.3.1.1 Initialization

All the Stream functions require a pointer to a **`Stream`** structure, defined in **`gios.h.`** The storage for the Stream must be allocated by the user before calling any of these functions.

`gi_strm_init` - initializes a Stream and assigns the input and output buffers. These buffers must be allocated by the user. Also sets flow control parameters and the type of flow control to use for the Stream. Note that these parameters cannot be changed without first closing the Stream.

`gi_strm_attach` - attaches a Stream to a driver. Detaches any existing driver first. Any data in the Stream output buffer is sent to the driver.

`gi_strm_detach` - detaches a Stream from a driver. The Stream's input and output buffers are preserved. Reads and writes on the Stream can still be performed.

4.3.1.2 Level 0 I/O

Level 0 Stream functions operate directly on the **`Stream`** structure and the input and output buffers. These functions provide the most efficient access to Streams. The level 1 functions described in the following section are built on top of these functions.

`gi_strm_input` - reads a byte from the input buffer

`gi_strm_output` - writes a byte to the output buffer

`gi_strm_flush` - wait for output buffer to empty

`gi_strm_size`- returns the size of the input or output buffer

4.3.1.3 Level 1 I/O

Level 1 Stream functions are built on top of the level 0 functions. They provide an interface similar to that of the C Standard Library character I/O functions. See **Section 4.6.1, Stdio Functions**, for more details.

With the exception of a Stream pointer argument, the following functions behave exactly as the Standard Library function which is named in their suffix. See any reference which describes the ANSI C Standard for more details.

`gi_strm_putchar`- writes a single character to the output buffer

`gi_strm_puts` - writes a string to the output buffer

`gi_strm_printf` - writes a formatted string to the output buffer

`gi_strm_getchar` - reads a single character from the input buffer

`gi_strm_gets` - reads a string terminated by a carriage return from the input buffer

The character I/O Standard C Library functions described in **Section 4.6.1** use the default GROM Stream for their I/O. This Stream is set using the **`gi_strm_stdio`** function. **`gi_strm_stdio`** stores the address of the Stream it is passed and places it in the **`STDIO`** GCB Register. Any function which takes an explicit Stream pointer argument can use **`STDIO`** to access this default Stream. See **Section 3.6** for a complete description of GCB Registers.

`gi_strm_stdio` - sets the default Stream for the character I/O Standard C Library Functions.

4.3.1.4 Driver Calls

GIOS provides two functions for Stream drivers to access a Stream's input and output buffers.

`gi_strm_pull` - reads the next byte from the output buffer

`gi_strm_push` - writes a byte to the input buffer

4.3.2 Stream Drivers

GIOS Stream drivers form the interface between Streams and the communications hardware on the GCB11. Typically a driver is implemented as an interrupt service routine. When a Stream is attached to a driver, the Stream obtains a pointer to the driver and vice versa. This allows the driver to use Stream functions on its attached Stream, and allows the Stream to invoke driver functions.

Drivers communicate with Streams by calling special functions in GIOS designed for this purpose. **`gi_strm_push`** and **`gi_strm_pull`** write and read bytes from the Stream's input and output buffers respectively. These are the only Stream functions which should be called from within an interrupt service routine. In addition, when calling these functions, care must be taken to ensure that application code cannot make calls to the Stream functions. This is usually accomplished by disabling interrupts (done automatically if the driver is an interrupt service routine).

Streams communicate with drivers by invoking call-back functions which must be specified during driver initialization. These call-backs, in addition to initialization and configuration function pointers and custom parameters used by each driver, are stored in memory in a Driver Configuration Table. The format of this table is defined in **gios.h** as the **DriverTable** structure:

```
typedef struct {
     /*
     **   initialization and configuration functions
     */
     int  *init;
     int  *open;
     int  *close;
     int  *ioctl;
     /*
     **   run-time call-backs
     */
     void *start_send;
     void *intr_disable;
     void *intr_enable
     void *hfc_stop
     void *hfc_start;
     /*
     **   custom parameters should follow in memory
     */
}
     DriverTable;
```

The following two section describe each of these functions pointed to in this table.

4.3.2.1 Run-Time Call-Backs

The run-time call-back functions are called by internal GIOS Stream code and never accessed directly by the user. If a driver does not support one or more of these functions, they should be set to **NULL** in the Driver Configuration Table. These functions should preserve all registers (including D). Each of the run-time call-backs is described below:

start_send - this function is used to tell the driver that data is available in the Stream output buffer which needs to be sent out. Although this function is only called when a byte is added to an empty output buffer by an application or when software flow control characters need to sent out, drivers should be able to respond to this call at all times. This function is useful when drivers are implemented as interrupt service routines using interrupts that can be disabled when there is nothing to do, such as the SCI.

intr_disable - this function is used by the Stream functions to ensure that they cannot be interrupted by the driver which is attached to the Stream (by calling **gi_strm_push** or **gi_strm_pull**). This is needed to ensure that the Stream buffer information is not corrupted during manipulation. For example, if the driver is implemented as an interrupt service routine, **intr_disable** would disable the interrupts associated with the driver. Note that **intr_disable** will always be followed by a **intr_enable**.

intr_enable - this function is called by the Stream functions when it is safe for the driver to make calls to the Stream (see **intr_disable** above). Note that **intr_enable** will always be preceded by a **intr_disable**.

hfc_stop - this function is called when hardware flow control is enabled and the input buffer reaches the **max_in** level specified for the Stream by **gi_strm_init**. If the driver supports hardware flow control, this function should do whatever is necessary to stop the incoming data flow.

hfc_start - this function is called when hardware flow control is enabled, flow has been stopped by **hfc_stop** (see above), and the input buffer reaches the **min_in** level specified for the Stream by **gi_strm_init**. If the driver supports hardware flow control, this function should do whatever is necessary to restart the incoming data flow.

4.3.2.2 Initialization and Configuration Functions

The initialization and configuration functions for a Stream driver are called indirectly using GIOS functions which access the driver functions through the pointers specified in the Driver Configuration Table. Before using a Stream driver, a **Driver** structure should be created. The **Driver** structure from **gios.h** is listed here for reference.When defining and initializing a **Driver** in application code, the user must be sure to set **next** and **stream** to **NULL** and copy the pointer to the **DriverTable** into **table**.

```
typedef struct{
     void           *next;    /* pointer to next driver */
     void           *stream;  /* pointer to connected stream */
     DriverTable    *table;   /* ptr to Driver Config Table */
     /*
     **   custom parameters should follow in memory here
     */
}
     Driver;
```

The functions listed below call the initialization and configuration function which are contained in the **DriverTable** structure pointed to in the **Driver** structure:

gi_drvr_init - perform any initialization specific to the driver. Does not need to activate the driver.

gi_drvr_open - activate the input and output channels. This call should allocate all the necessary system resources to allow a Stream to use the driver.

gi_drvr_close - deactivate the input and output channels. This call should free all the necessary system resources allocated in **gi_drvr_open**.

gi_drvr_ioctl - pass a variable argument list to the driver. This call can be used to do custom configuration or to change default driver parameters.

4.3.3 GROM Stream Drivers

GROM contains code and configuration tables for three Stream Drivers:

- the SCI
- the SPI
- GNET

The configuration tables for each driver are located in GROM constant data. See **Section 3.8, Constant Data**, for details.

Each GROM driver is described in detail in the following sections, including which call-back functions are supported and any limitations of the driver. Each of the functions described in **Section 4.3.3, GROM Stream Drivers** is supported by all three drivers. The arguments for the `gi_drvr_ioctl` functions for each driver are detailed below.

Note that the SCI and GNET drivers both share the MC68HC11 asynchronous serial communications interface, or SCI. To avoid conflicts between these two drivers, two variables are used control access to the SCI. The first, the GCB Register **`SCISTART`**, controls what communications driver, if any, is assigned to the SCI at start-up. The second, GROM RAM variable **`SCIDRIVER`**, indicates what GROM code currently has control of the SCI. Application programs should be sure that **`SCIDRIVER`** is set to **`NO_DRIVER`** (0), indicating that the SCI is not being used by GROM, before programming the SCI directly. See **Section 3.11.4** for a more detailed description of these variables.

Table 4.1 shows all the MC68HC11 resources used by the GROM drivers.

Driver	Ports	Registers Used	Interrupt Vector
SCI	PORTD-PD0-1 PORTD-PD2 or PORTG-PG2	BAUD SCCR1 SCCR2 SCDR SCSR	INTVSCI
SPI	PORTD-PD2-5	SPCR SPDR SPSR	INTVSPI
GNET	PORTD-PD0-1 PORTD-PD2 or PORTG-PG2	BAUD SCCR1 SCCR2 SCDR SCSR TFLG2 - RTIF TMSK2 - RTII TMSK2 - PR1 TMSK2 - PR0	INTVSCI INTVREALTIME

TABLE 4.1 GROM Stream Driver MC68HC11 Resource Usage

4.3.3.1 SCI Driver

The SCI Stream driver is implemented as an SCI interrupt service routine. Hardware flow control is not supported. The only parameter which can be configured by **`gi_drvr_ioctl`** is the SCI baud rate. When **`gi_drvr_init`** is called on the SCI driver, the baud rate is initialized to the GCB Register SCIBAUD. See the Companion disk for the details of the **`gi_drvr_ioctl`** calls.

4.3.3.2 SPI Driver

The SPI Stream driver uses the MC68HC11 Serial Peripheral Interface, or SPI. See Chapter 8 in the M68HC11 Reference Manual for details on the SPI. The SPI driver allows two GCB11s to be connected together using their SPI ports (note only one slave is allowed). See **Section 6.3** for a description of this interconnection. By routing all character I/O through the SPI and using the `SPI_MON` example program on the companion disk, the SCI can be freed up for some other purpose.

The SPI driver can be initialized in two modes, master or slave. The default configuration is as a slave. `gi_drv-r_ioct`l can be used to change the configuration to master, as well as to set the baud rate and to assign a timing source for the master. The master can use the real-time interrupt or any of the output compare interrupts as a clock source. In any case, the driver is implemented as an interrupt service routine. Note that in slave mode, the timer and baud rate configurations are not used.

See the Companion disk for the details of the `gi_drvr_ioctl` calls.

Note that the SPI driver will not send a data byte of 0, prohibiting the transmission of binary data.

4.3.3.3 GNET Driver

The GNET Stream driver allows Stream I/O to be routed through the GNET network software to another node on the network. When the GNET Stream driver is open, GNET multiplexes the Stream I/O with the normal network traffic of the node. The GNET Stream driver sends and receives the Stream data by packing the data bytes in a GNET message. When sending, all bytes in the Stream output buffer are sent in a single message. Any bytes received are placed in the input buffer. Stream data can only be exchanged between Stream partner nodes. Stream partners are described in **Section 6.4**.

In GROM 1.4, The `gi_drvr_ioct`l function for the GNET driver provides no functionality. All configuration of the network should be done using GNET function calls.

4.3.3.4 Sharing the SCI with GBUG

GBUG uses the GROM Stream drivers to communicate with the outside world. This can be a problem when an application wishes to use the SCI for some other purpose (e.g. a different Stream driver than the one GBUG is using). The problem occurs because closing a Stream driver requires knowledge of its address, and GBUG data is hidden from applications. There are two special functions in GROM to help overcome this problem.These functions allow an application to close ANY SCI driver that is being used by GBUG, and also allows the driver to be reinitialized when the application is finished so GBUG will continue to operate.

`gi_sci_free` -frees the SCI driver which is being used by GBUG. This function returns a pointer to the Stream that GBUG was using and the state of the `SCIDRIVER` GROM RAM variable indicating which driver was being used and how it was configured.

`gi_sci_reinit` -returns the SCI to the state it was in before `gi_sci_free` was called and reattaches the Stream which GBUG was using. Uses the data returned by `gi_sci_free`.

4.3.4 Custom Stream Drivers

To write a custom Stream driver, the code for each of the nine functions in the `DriverTable` should be compiled and a `DriverTable` initialized to point to each one.

4.4 EEPROM

GROM contains functions for writing and erasing EEPROM. Each of these functions is taken directly from Section 4.3.5.1 of the M68HC11 Reference Manual. Users should read all of Chapter 4 in the M68HC11 Reference Manual before using EEPROM. The delay used for programming the EEPROM is stored in GROM constant data. See section **Section 3.8.3, Crystal Speed Dependent Constants**, for more details.

`gi_eeprom_erase` - erases a single byte of EEPROM.

`gi_eeprom_erase_bulk` - erases all of EEPROM.

`gi_eeprom_erase_row` - erases a single row of EEPROM. A row is sixteen bytes starting on an even sixteen -byte boundary (7E00, 7E10, 7E20, etc...).

`gi_eeprom_write` - writes a single byte of EEPROM.

Table 4.1 shows all the MC68HC11 resources used by the EEPROM routines.

Function	Ports	Registers	Interrupt Vectors
All		PPROG	

TABLE 4.2 GIOS EEPROM MC68HC11 Resource Usage

4.5 Interrupt Utilities

GIOS provides the following routines for using interrupts:

`gi_intr_disable` - disables interrupts using the SEI instruction.

`gi_intr_dummy` - dummy interrupt service routine which executes a RTI instruction.

`gi_intr_enable` - enables interrupts using the CLI instruction.

`gi_intr_rti` - issues an RTI instruction. This can be used when writing interrupt service routines in C. Note that this routine works only if the SP register points to the interrupt stack frame (i.e.the SP is the same value as it was when entering the ISR). Without special care, this will not always be true. Some compilers always mess with the stack upon entering a C function (e.g. GCC 2.3.3). Also, declaring automatic variables in the ISR will usually change the SP.

`gi_intr_swi` - issues an SWI instruction. This can be used to return to GBUG from a C routine.

4.6 Standard Library

GIOS includes a subset of the functions in the Standard C Library. Most of the commonly used functions are included. There is no long or floating point support. The Standard C Library include files on the GCB11 companion disk contain ANSI C prototypes for each of the functions. For a complete list of supported functions, see the tables in **Section 3.9, Call Table**. Man pages for these functions are not included in **CHAPTER 9, Programmer's Reference**. For a complete description of these functions, see any reference on the ANSI C Standard, for example *The C Programming Language* by Kernighan and Ritchie or *The Standard C Library* by Plauger.

4.6.1 Stdio Functions

The following character I/O functions use the GCB Register **`STDIO`** as a pointer to a GIOS Stream. **`STDIO`** is set with the **`gi_strm_stdio()`** function call.

- **`getchar`**
- **`gets`**
- **`printf`**
- **`putchar`**
- **`puts`**
- **`vprintf`**

CHAPTER 5

GBUG

5.1 Introduction

This chapter describes the GROM Monitor and Debugger, or GBUG. GBUG provides the user with a simple interface for downloading data, inspecting memory, and running and debugging programs.

Note that because of space limitations in the ROM, some of the functionality described in the section will not always be available in every GROM configuration. This may include the GROM EPROM shipped with the GCB11. See the **README** file on the companion disk for details of what functionality is missing in each configuration.

5.2 Getting Started

5.2.1 Getting a Connection

GBUG performs input and output through a dedicated GIOS Stream (see **Section 5.3.1, I/O Streams**, for more details). When shipped from the factory, GBUG's Stream is configured for a direct connection through the SCI to ports JP5 or JP6 (RS485 or RS232 respectively) with the following communications protocol:

- 9600 baud
- 8 data bits
- no parity
- 1 stop bit

If a terminal with the above protocol is connected to either of these ports and the GCB11 is reset, you should see the following:

```
GROM 1.4, Copyright(c) 1993 Coactive Aesthetics, All Rights Reserved
A:00  D:0000  IX:0000  SP:0000  CCR:00  (00000000)
B:00          IY:0000  PC:0000          (SXHINZVC)
0000: FF 00 FF       STX  00FF
]
```

This is a display of the GROM version number, the CPU registers, the next line of assembly language to execute, and a GBUG command prompt. If this display does not appear, try inserting a null modem into the serial connection and resetting the GCB11 again. If characters still do not appear, reset the GCB registers as described in **Section 3.11.3** and try this procedure again.

5.2.2 Entering Commands

After outputting a prompt, GBUG is ready to accept a command. Commands take the form of a one or two character mnemonic followed by any arguments. When entering commands, case is not important and commands and arguments can be separated by any number of spaces or tabs. Here's an example:

```
] dm 100 10
0100: 00 00 D0 00 F0 FF 00 06 27 20 00 00 00 00 F3 B2     |........% ......|
]
```

This command, Display Memory or **DM**, tells GBUG to display16 bytes of memory starting at address 100. After outputting the result, GBUG prints another prompt.

5.2.3 Downloading

While programs and data can be entered one byte at a time with the **MM** command, GBUG also accepts data formatted as Motorola S-records for a faster program development cycle. To download S-records to the GCB11, use the **TD** command. Its simplest form is:

```
] td
```

Notice that no prompt is displayed after this command is entered. GBUG is now ready to accept S-record data. Characters received are not echoed while downloading S-records. When the download is complete, the prompt will return and more commands can be entered. See **Section 5.6, GBUG Commands**, for more details on **TD** and related commands.

5.2.4 Running a Program

Once a program is in memory, it can be run using the **R** command. Assuming that a program to print "hello world" starts in memory at location 1000 hex, we have the following:

```
] R 1000
hello world
]
```

5.2.5 Help

One of the most useful commands in GBUG is **?**, which is short for *help*. This command will print a screen full of information summarizing all the available commands and what arguments each one accepts.

5.3 Concepts

5.3.1 I/O Streams

GBUG receives commands and displays output using a dedicated GIOS Stream. GIOS Streams are discussed in detail in **Section 4.3**. The Stream driver used by GBUG is controlled by the GCB Register `GBUGSTRM`. GCB

Registers are discussed in detail in **Section 3.6**. **GBUGSTRM** can have one of three values, indicating which of the three built-in Stream drivers GBUG will use for it's I/O:

- STRM_SCI (0) - SCI Stream driver
- STRM_SPI (1) - SPI Stream driver
- STRM_GNET (2) - GNET driver

The SPI and GNET drivers allow GBUG to be run remotely from a second GCB11 or from a PC respectively.

User code run from GBUG can use GBUG's Stream by using (and not changing) the **STDIO** Stream. Note that if this code is run without GBUG (e.g. when making a custom EPROM), it will have to allocate and initialize a Stream and driver itself and assign it to **STDIO**.

All three drivers listed above are initialized by GBUG at reset, and should not be re-initialized by user code if the user program is going to use the Stream configured by GBUG. Again, if this code is run without GBUG, initialization code will be needed.

5.3.2 Commands

A GBUG command consists of a list of tokens followed by a carriage return. Tokens are strings of visible characters separated by any number of spaces or tabs. The maximum number of characters in a command line is 80.

The general form of a GBUG command is:

command [**arguments**...]

Command is always present and consists of a single token containing one or two characters. **Command** determines the functionality of the GBUG command.

Arguments specify data which the command operates on. **Arguments** can be flags, numerical values, mnemonics, or expressions. Each of these types is detailed below.

5.3.3 Arguments

GBUG arguments can be separated into three types, each of which is detailed in a section below:

- flags
- integers
- assembly code

5.3.3.1 Flags

Flags are used by commands to choose between modes and to indicate the presence of special arguments on the command line. Flags are always single characters.

5.3.3.2 Integers

Integers are entered in hexadecimal format.

5.3.3.3 Assembly Code

Assembly input can only be entered using the Modify Assembly (**MA**) command. The syntax for GBUG assembly code follows that given in the Motorola Freeware PC-Compatible 8-Bit Cross Assemblers User's Manual with the following exceptions:

- directives are not supported
- symbols are not supported
- comments are not supported
- binary data is not supported
- octal data is not supported
- expressions are not supported.
- all numbers are hexadecimal by default and other radices and input types use the prefix notations described in the proceeding sections.
- extended addressing can be forced by preceding the operand with a '>'; direct addressing can be forced by preceding the operand with a '<'.
- the size of immediate addressing mode operands depends on the size of the register involved in the operation.

5.3.4 Display Data

GBUG can display five types of data:

- integers
- characters
- addresses
- registers
- assembly code

The type of data displayed by each command is fixed and is specified in the **Section 5.6, GBUG Commands**, for each command.

5.3.5 Program Execution

The primary function of GBUG is to aid in the debugging of programs. Without an In-Circuit Emulator, this can only be accomplished by providing a complete set of display, modify, and program execution commands. This section describes how to use the six instructions that are available to control program execution:

BS - Breakpoint Set

BC - Breakpoint Clear

R - Run

RU - Run Until

SS - Single Step

SO - Step Over

Detailed descriptions and examples of each of these commands are given in **Section 5.6, GBUG Commands**.

In this section, the code being executed is referred to as the *user program* or *user code*, and this program's registers are called *user CPU registers*.

Note that because all breakpoints supported by these commands are software breakpoints, they can only be set in RAM.

5.3.5.1 Starting Execution

Execution of user code is started from GBUG using one of the run or step commands: **R**, **RU**, **SS**, or **SO**. In all cases, the user CPU registers are loaded on the stack and execution is passed to the user program by executing a Return From Interrupt (RTI) instruction.

Run Commands

The GBUG run commands, **R** and **RU**, transfer control of the CPU to the user code. Execution returns to GBUG only if one of the following conditions is met:

- a breakpoint is reached
- a break signal is detected
- user code SWI instruction
- the GCB11 is reset
- an interrupt occurs which is not handled by user code

Both commands have a flag to disable sticky breakpoints. Breakpoints are described in detail below.

Step Commands

The Single Step command, **SS**, executes a single instruction and then returns to GBUG.

The Step Over command, **SO**, continues execution until the next line is reached. This command is included to allow *stepping over* subroutine and branch calls. Execution can return to GBUG prematurely if any of the conditions listed above for **Run Commands** are met.

5.3.5.2 Stopping Execution

Unless the GCB11 is reset, execution returns to GBUG by means of an interrupt. GBUG takes the user CPU registers off the stack and makes them available for display and modification.

Breakpoints

Breakpoints are special instructions inserted into executable code in RAM to aid in debugging. When the Program Counter is equal to the address of a breakpoint, execution of the code is interrupted and control is retuned to GBUG.

GBUG allows both sticky and non-sticky breakpoints:

- Sticky breakpoints are inserted and deleted using the Breakpoint Set (**BS**) and Breakpoint Clear (**BC**) commands respectively. Both commands accept the address of the breakpoint to set or clear as an argument. GBUG supports ten sticky breakpoints. If a program is run using **R**, **RU**, or **SO** without the no breakpoints (N) flag, all sticky breakpoints are installed. Current sticky breakpoints can also be viewed with the **BS** command.

- Non-sticky breakpoints are only installed for the duration of a single GBUG command. There are two non-stick breakpoints available in GBUG and they are set indirectly with the **RU, SO,** and **SS** commands. These commands use non-sticky breakpoints to return control to GBUG after they complete.

Note that breakpoints are represented physically by the Software Interrupt instruction (SWI) inserted into RAM. GBUG saves the data in the bytes which are replaced by SWIs. If the GCB11 is reset while breakpoints are installed, these SWI instructions will remain in memory and the original data will be lost.

Also note that breakpoints should not be placed in interrupt service routines. See **Section 5.3.7, Interrupts** for more details.

Break Signal

A break signal received by the GBUG Stream while user code is running will cause execution to return to GBUG. Note that if user code disconnects the GBUG Stream from its driver, the break facility will not function (see **Section 5.3.1, I/O Streams**, for more details). Both GPC and the SPI Stream driver translate break commands from the PC or from a second GCB11 running the **`SPI_MON`** program respectively.

SWI Instruction

If user code executes an SWI instruction (or calls the **`gi_intr_swi`** GIOS function), execution will return to GBUG.

Reset

Resetting the GCB11 causes all state information about the user program being run to be lost, and restarts GBUG. Although GBUG attempts to salvage breakpoint information, the MC68HC11F1 does not guarantee that any internal data is preserved. Since resets can occur asynchronously, data in RAM may also be corrupted.

Interrupts

At reset, all interrupts not being used by GBUG, GNET, or GROS are vectored to GBUG. An error is printed if one of these interrupts is used.

5.3.5.3 Run on Reset

The Run on Reset functionality allows GBUG to start executing a program at a memory location specified by the **`GBUGRR`** GCB register immediately after reset. All GBUG initialization is preformed, but no GBUG sign-on or prompt is issued. Run on Reset is disabled by setting the **`GBUGRR`** register to 0 (note that this can be accomplished by reseting the GCB registers as described in **Section 3.11.3**).

5.3.6 EEPROM

While displaying EEPROM memory locations is accomplished as with any other memory location, modifying EEPROM must be done using special commands designed for that purpose. Before using EEPROM, the user should read Section 4.3 of the M68HC11 Reference Manual.

The following commands should be used to modify EEPROM:

- Copy to EEPROM (**CE**) to copy blocks of memory from RAM or EPROM to EEPROM
- EEPROM Erase commands (**EE**, **EB**, and **ER**) to erase bytes, rows, or all of EEPROM
- Modify EEPROM (**ME**) to modify individual bytes of EEPROM

Because EEPROM has a limited number of write-erase cycles before going bad, it is suggested that any bulk data which is to be placed in EEPROM (such as program code or data) be set up and tested in RAM or EPROM and then moved to EEPROM using the **CE** command.

Care should be taken when modifying EEPROM so that the GCB Registers, which are stored at the bottom of EEPROM, are not corrupted. See **Section 3.6** for the locations of the GCB Registers.

5.3.7 Interrupts

Interrupts are not disabled when GBUG is executing. This means that any interrupt service routines that are installed will continue to service their interrupts, and any side effects of these routines will continue to occur. This also means that breakpoints should never be placed in an interrupt service routine. Unfortunately, without an In-Circuit Emulator, this can make it extremely difficult to debug programs that use interrupts.

NOTE: interrupts must be enabled for GBUG character I/O to function.

5.4 Error Codes

1 -	"bad command"	GBUG did not recognize the command given.
2 -	"bad args"	GBUG could not parse the arguments to the command.
3 -	"too many values"	Too many values were given as arguments to a command.
4 -	"bad expression"	GBUG could not parse an expression.
5 -	"not an srecord"	A badly formed s-record was received. This could mean that a character was dropped or that the transfer was attempted into a read-only memory area (e.g. EPROM, EEPROM).
6-	"bad record type"	A bad s-record type was received.
7 -	"checksum error"	The checksum on a s-record did not match.
8 -	"bad register"	GBUG could not parse a register mnemonic.
9 -	"bad format"	GBUG could not pars a display format.
10 -	"can't open stream"	The stream driver requested is not available or an error occurred while attempting to open it.

5.5 Command Summary

?	- Help
BC [*addr...*]	- Breakpoint Clear
BD *driver* [U]	- Set GBUG stream
BS [*addr...*]	- Breakpoint Set
CE *src dest len*	- Copy to EEPROM
CM *src dest len*	- Copy Memory
DA *addr len*	- Display Assembly
DG [*gcbreg*]	- Display GCB Register
DI [*ioreg*]	- Display I/O Register
DM *addr len*	- Display Memory
DR	- Display CPU Registers
EB	- Erase EEPROM Register
ER *addr*	- Erase EEPROM Row
GC [U]	- GCB Register Checksum
GE	- GCB Register Copy to EEPROM
GR	- GCB Register Copy to RAM
MA *addr*	- Modify Assembly
ME *addr val...* [R *rep*]	- Modify EEPROM
MG *gcbreg val* [U]	- Modify GCB Register
MI *ioreg val*	- Modify I/O Register
MM *addr val...* [R *rep*]	- Modify Memory
MR *reg val*	- Modify CPU Register
NE	- Network Exit
NI	- Network Initialization
R [N] [*addr*]	- Run
RU [N] [*addr*] *stop*	- Run Until
SO [N] [*addr*]	- Step Over
SS [*addr*]	- Single Step
ST	- SCI Test
TD [*offset*]	- Transfer Down

5.6 GBUG Commands

? Help ?

Synopsis: **?**

Description: Print a summary of GBUG commands.

See Also:

Examples:

BC Breakpoint Clear BC

Synopsis: **BC** [*addr...*]

addr - expression for address of breakpoint to clear

Description: Remove breakpoints at *addr* from RAM.

If *addr* is missing, all breakpoints are removed.

See Also: **BS** - Breakpoint Set, **R** - Run, **RU** - Run Until, **SS** - Single Step, **SO** - Step Over

Examples:

BD GBUG Driver BD

Synopsis: **BD** *driver* [U]

driver	- Stream driver to use as GBUG Stream
U	- update the EEPROM copy of the **GBUGSTRM** GCB Register.

Description: Attach the indicated Stream driver to the GBUG Stream. If the U flag is present, the **GBUGSTRM** GCB Register is updated in EEPROM. The possible values of *driver* are:

C	- SCI driver
N	- GNET driver
P	- SPI driver

See **Section 5.3.1, I/O Streams**, for more details.

See Also: **DG** - Display GCB Register, **MG** - Modify GCB Register

Examples:

BS Breakpoint Set BS

Synopsis: **BS** [*addr...*]

addr - expression for address of breakpoint to set

Description: Insert a breakpoint into RAM at *addr*.

If *addr* is missing, all breakpoints are displayed.

NOTE: breakpoints should never be placed in an interrupt service routine.

See Also: **BC** - Breakpoint Clear, **R** - Run, **RU** - Run Until, **SS** - Single Step, **SO** - Step Over

Examples:

CE Copy to EEPROM CE

Synopsis: **CE** *src dest len*

src	- source address for move
dest	- destination address for move
len	- number of words to move

Description: Moves *len* words of memory from *src* to *dest*. *Src* must be in RAM or EPROM. *Dest* must be in EEPROM.

See the M68HC11 Reference Manual for details on EEPROM operation.

See Also: **CM** - Copy Memory

Examples:

CM Copy Memory CM

Synopsis: **CM** *src dest len*

src	- source address for move
dest	- destination address for move
len	- number of words to move

Description: Moves *len* words of memory from *src* to *dest*. *Src* must be in RAM, EPROM, or EEPROM. *Des*t must be in RAM. Overlap is handled.

See Also: **CE** - Copy to EEPROM

Examples:

DA Display Assembly DA

Synopsis: **DA** *addr len*

addr	- expression for the address of first byte to disassemble
len	- expression for the number of lines of assembly to display

Description: Disassembles the memory bytes indicated. If a line of assembly starts with an illegal opcode, the mnemonic is listed as `?????`.

See Also: **MA** - Modify Assembly

Examples:

DG Display GCB Registers DG

Synopsis: **DG** [*gcbreg*]

gcbreg - mnemonic of GCB Register to display

Description: Displays the contents of the GCB Register indicated in hexadecimal format. The GCB Registers and their mnemonics are described in **Section 3.6**.

If *gcbreg* is missing, all GCB Registers are displayed.

See Also: **MG** - Modify GCB Register

Examples:

DI Display I/O Registers DI

Synopsis: **DI** [*ioreg*]

ioreg - mnemonic of I/O Register to display

Description: Displays the contents of the I/O Register indicated in hexadecimal format. The mnemonics are those used by Motorola in the MC68HC11F1 Technical Data book.

If *ioreg* is missing, all I/O Registers are displayed.

See Also: **MI** - Modify I/O Register

Examples:

DM Display Memory DM

Synopsis:	**DM** *addr len*
	addr - expression for address of first byte to display
	len - expression for number of words to display
Description:	Displays the contents of the RAM, EPROM, or EEPROM locations indicated.
See Also:	**MM** - Modify Memory
Examples:	

DR Display CPU Registers DR

Synopsis: **DR**

Description: Displays the CPU registers and the next instruction to be executed.

See Also: **MR** - Modify Register

Examples:

EB EEPROM Block Erase EB

Synopsis: **EB**

Description: Erase all of EEPROM.

See Also: **ER** - EEPROM Row Erase

Examples:

ER EEPROM Row Erase ER

Synopsis: **ER** *addr*

addr - address of row to erase

Description: Erase the 16 byte row of EEPROM containing *addr.*

See Also: **EB** - EEPROM Block Erase

Examples:

GC GCB Register Checksum GC

Synopsis: **GC** [U]

U - update the EEPROM GCB Register checksum.

Description: Checksums the EEPROM GCB Registers. If the U flag is present, it writes a new, correct checksum. Prints “checksum ok” or “bad checksum”.

See Also:

Examples:

GE GCB Register Copy to EEPROM GE

Synopsis: **GE**

Description: Copy GCB Registers from EPROM to EEPROM and updates the checksum.

See Also: **GR** - GCB Register Copy to RAM

Examples:

GR — GCB Register Copy to RAM — GR

Synopsis: **GR**

Description: Copy GCB Registers from EEPROM to RAM.

See Also: **GE** - GCB Register Copy to EEPROM

Examples:

MA Modify Assembly MA

Synopsis: **MA** *addr*

addr - expression for starting address of assembly

Description: Assemble code into RAM. See **Section 5.3.3.3** for assembly language syntax. The user is prompted for each line with the address in memory where the assembled code will be placed and the current instruction and operands. After entering the assembly line, pressing <return> causes the line to be assembled and the user is prompted for the next line. Pressing <return> at the prompt retains the previous values and prompts the user with the next line. Typing a period '.' followed by <return> at the address prompt stops the operation.

See Also: **DA** - Display Assembly

Examples:

ME Modify EEPROM ME

Syntax: **ME** *addr val*... [R *rep*]

addr	- expression for address of first word to modify
val	- expression for new value of memory word
R	- indicates that *val*... should be repeated *rep* times
rep	- expression for repeat count

Description: Set the contents of EEPROM starting at *addr* to the value(s) of the expression *val*.... If the R flag is used, *val*... is placed in memory *rep* times starting at *addr.*

If *val* is missing, an interactive mode is invoked. The user is prompted by the memory address *addr* and current value of the word in that location. Entering a expression and pressing <return> modifies that location and prompts the user with the next word in memory. Pressing <return> at the prompt retains the previous value and prompts the user with the next word. Typing a period '.' followed by <return> stops the operation.

See Also: **CE** - Copy to EEPROM

Examples:

MG Modify GCB Register MG

Synopsis: **MG** *gcbreg val* [U]

gcbreg	- mnemonic of GCB Register to modify
val	- expression for new value of GCB Register
U	- update the EEPROM copies of the GCB register.

Description: Modify the contents of the GCB Register indicated. The GCB Registers and their mnemonics are described in **Section 3.6**. If the U option is specified the corresponding EEPROM copy of the GCB Register is updated along with the EEPROM checksum.

See Also: **DG** - Display GCB Register

Examples:

MI Modify I/O Register MI

Synopsis: **MI** *ioreg val*

ioreg	- mnemonic of I/O Register to modify
val	- expression for new value of I/O Register

Description: Modify the contents of the I/O Register indicated. The mnemonics are those used by Motorola in the MC68HC11F1 Programming Reference Guide.

See Also: **DI** - Display I/O Register

Examples:

MM Modify Memory MM

Synopsis: **MM** *addr val*... [R *rep*]

addr	- expression for address of first word to modify
val	- expression for new value of memory word
R	- indicates that *val*... should be repeated *rep* times
rep	- expression for repeat count

Description: Set the contents of RAM starting at *addr* to the value(s) of the expression *val*.... If the R flag is used, *val*... is placed in memory *rep* times starting at *addr.*

If *val* is missing, an interactive mode is invoked. The user is prompted by the memory address *addr* and current value of the word in that location. Entering a expression and pressing <return> modifies that location and prompts the user with the next word in memory. Pressing <return> at the prompt retains the previous value and prompts the user with the next word. Typing a period '.' followed by <return> stops the operation.

See Also: **DM** - Display Memory

Examples:

MR Modify CPU Register MR

Synopsis: **MR** *reg val*

reg	- mnemonic of CPU register or CCR flag to modify
val	- expression for new value of CPU register

Description: Modify the contents of a CPU register or condition code register flag. The register mnemonics available are:

a	- Accumulator A
b	- Accumulator B
d	- Double Accumulator D
sp	- Stack Pointer
ix	- Index Register X
iy	- Index Register Y
pc	- Program Counter
ccr	- Condition Code Register

See Also: **DR** - Display CPU Registers

Examples:

NE GNET Exit NE

Synopsis: **NE**

Description: Exit GNET by calling `gn_exit_comms`.

See Also: **NI** - GNET Initialization

Examples:

NI GNET Initialization NI

Synopsis: **NI**

Description: Initialize GNET by calling `gn_init_comms`.

See Also: **NE** - GNET Exit

Examples:

R Run R

Synopsis: **R** [N] [*addr*]

N	- don’t install breakpoints
addr	- address to start execution

Description: Start execution at *addr*. Breakpoints are installed unless the N flag is present. If a breakpoint is reached, execution returns to the monitor.

If *addr* is missing, the current value of the pc register is used.

See Also: **BC** - Breakpoint Clear, **BS** - Breakpoint Set, **RU** - Run Until, **SS** - Single Step, **SO**- Step Over

Examples:

RU Run Until RU

Synopsis: **RU** [N] [*start*] *stop*

N	- don’t install breakpoints
start	- address to start execution
stop	- address to stop execution

Description: Start execution at *start* and run until *stop*. Breakpoints are installed unless the N flag is present. If a breakpoint or *stop* is reached, execution returns to the monitor.

If *start* is missing, the current value of the pc register is used.

See Also: **BC** - Breakpoint Clear, **BS** - Breakpoint Set, **R** - Run, **SS** - Single Step, **SO**- Step Over

Examples:

SO Step Over SO

Synopsis: **SO** [N] [*addr*]

N	- don’t install breakpoints
addr	- address of instruction to execute

Description: Start execution at *addr* and run until the pc register points to the next instruction in memory. Breakpoints are installed unless the N flag is present. This has the effect of stepping over subroutine calls and branch instructions. When the next instruction in memory is reached, execution returns to the monitor.

If *addr* is missing, the current value of the pc register is used.

See Also: **BC** - Breakpoint Clear, **BS** - Breakpoint Set, **R** - Run, **RU** - Run Until, **SO**- Step Over

Examples:

SS Single Step SS

Synopsis: **SS** [*addr*]

addr - address of instruction to execute

Description: Execute the next instruction and return execution to the monitor.

If *addr* is missing, the current value of the pc register is used.

See Also: **BC** - Breakpoint Clear, **BS** - Breakpoint Set, **R** - Run, **RU** - Run Until, **SO**- Step Over

Examples:

ST SCI Test ST

Synopsis: **ST**

Description: Test network cable connections. See **Section 6.8, Jumper and Cable Configuration Testing**, for more details.

See Also:

Examples:

TD Transfer Down S-Records TD

Synopsis: **TD** [*offset*]

offset - expression for the amount to offset the S-record data

Description: After this command is issued, all subsequent data is treated as Motorola S-records until the end of an S9 record. The addresses specified in the S-records are modified by *offset*. Execution returns to the monitor when a S9 record is entered or an error occurs.

S-records are character strings terminated with a newline (carriage return, line feed, or carriage return-line feed pair). Each S-record contains 5 fields: type, length, memory address, data, and checksum. No characters are allowed between fields. Each field is described below:

Field	Size	Contents
Type	2	S-record type
Length	2	Number of bytes represented by record excluding type and length. Note that the actual number of characters is twice the number of bytes represented.
Address	4	Address at which the data is placed in memory
Data	0-2n	Data which is represented by a 2-digit 0 padded hexadecimal number. For example, a byte of value 1 would be represented by two character: 01, a byte of value 255 would be FF.
Checksum	2	2-digit 0 padded hexadecimal number representing the least significant byte of the one's compliment of the sum of the values of the length, address, and data fields.

GBUG supports three record types: S0, S1, and S9. Each type is described below:

Type	Description
S0	Optional header record. Address and data field is ignored.
S1	Data records.
S9	Termination record. The address field is loaded into the user program's pc register. The data field is ignored.

If *offset* is missing, an offset of 0 is used.

See Also:

Examples:

CHAPTER 6

GNET

6.1 Introduction

There are three different methods which can be used to communicate with the GCB11, as shown in Figure 6.1. Each of these are supported by the hardware and software supplied with the GCB11. Note that the GNET networking software is an option that must be purchased separately.

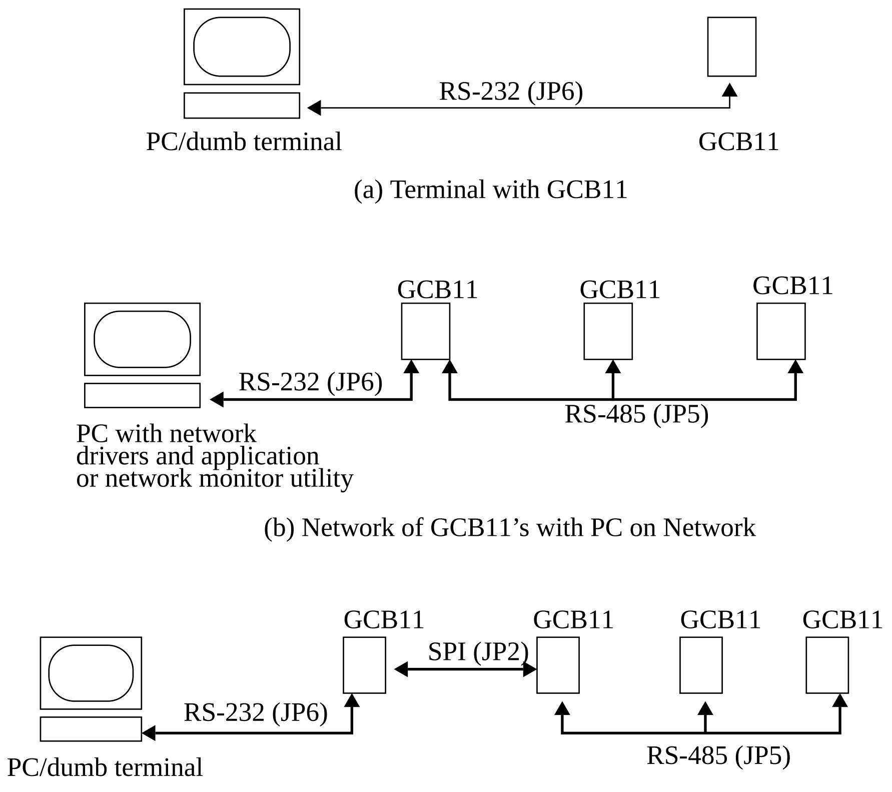


FIGURE 6.1 Methods of Communication with the GCB11

In the method shown in Figure 6.1(a) the SCI of the GCB11 is used to communicate with a dumb terminal through an ordinary RS-232 connection. This same configuration can also be used for point to point communication between GCB11s.

The method shown in Figure 6.1(b) depicts the GCB11 being used in a communications network. By using the appropriate communications software on the PC, it is possible to use the PC as a network console. In addition, it is possible to write applications for the PC such that they can communicate to any other node on the network. When used in this manner the PC is just another node on the network.

Finally, as shown in Figure 6.1(c), it is possible to use an additional GCB11 as a terminal for a GCB11 by using the SPI capabilities of the M68HC11. By doing this you can then use any dumb terminal and physically connect directly to a GCB11 when the SCI on that GCB11 is being used for the network connection.

These three communication methods correspond to the concept of Streams discussed in **CHAPTER 4, GIOS**. Briefly, GIOS supports the following three console I/O Streams:

- SCI used for normal asynchronous communications (Figure 6.1(a)).
- SCI used for network communications (Figure 6.1(b)).
- SPI connected to another GCB11 (Figure 6.1(c)).

Once a Stream is initialized and connected to the application, all console I/O such as `printf` and `gets` will go through that Stream in a manner which is transparent to the application. This feature gives the user a tremendous amount of flexibility to do things such as connect a network console to a GCB11 for the application I/O while connecting another terminal to the SPI for GCB11 ROM monitor I/O. It is possible to have a debug monitor separate from the applications console. This also allows the user to develop portable code using the normal console I/O functions and to simply change the Stream at run time.

For Streams (a) and (c) all that is needed is a dumb terminal connected to the appropriate connector, while Stream (b) requires that the console use network protocol to communicate with the nodes.

Table 6.1 shows all the resources used by communications.

Mode	Ports	68HC11 Registers	Interrupt Vectors
SCI Dumb Terminal	PORTD-PD0 PORTD-PD1 PORTD-PD2 or PORTG-PG2	SCDR (all) SCCR1 (all) SCCR2 (all) SCSR (all) BAUD (all)	intvsci
SCI Network	PORTD-PD0 PORTD-PD1 PORTD-PD2 or PORTG-PG2	SCDR (all) SCCR1 (all) SCCR2 (all) SCSR (all) BAUD (all) TFLG2 - RTIF TMSK2 - RTII TMSK2 - PR1 TMSK2 - PR0	intvsci intvrealtime
SPI Terminal	PORTD - PD2 PORTD - PD3 PORTD - PD4 PORTD - PD5	SPCR (all) DDRD (2-5) SPCR (all)	intvspi

TABLE 6.1 M68HC11 Resources Used by Communications

The remainder of this chapter describes each of these communications channels with special emphasis given to network communications. Please refer to **CHAPTER 4, GIOS** on how to use each of these channels for console I/O through the use of Streams.

6.2 Point to Point using the SCI

The SCI of the M68HC11 is a general purpose asynchronous UART. A wide variety of communications parameters and baud rates are supported. For more specific information on the SCI please consult the M68HC11 Reference Manual.

As shown in Figure 6.1(a), this mode of communication is normally used to connect a dumb terminal to the GCB11. Although the SCI is typically used as a console for screen and keyboard I/O, it is possible to have applications on a PC or another GCB11 communicate through this channel.

Connector JP5 is used for RS-485 communication while JP6 is used for RS-232. Both are driven by the SCI and it is possible to use either one. RS-485 is normally used for network communications while RS-232 is normally used for point to point communications.

As shown in **CHAPTER 2, Hardware** the reason that either connector can be used is that the RS-232 and RS-485 drivers on the GCB11 are tied together so that all receive and transmit signals go through both drivers. Because of this, care needs to be taken to ensure that the RS-485 drivers are enabled. Pin 1 of JP6 is used to control the active high enable line of the RS-485 driver. There is a pull up resistor on the RS-485 enable line so that if desired the pin can be left open, and the RS-485 driver will be enabled and not require any additional control. Another alternative is to connect the DTR or CTS line of the dumb terminal to pin 1 of JP6.

The receive and transmit lines of the SCI are routed through the RS-232 and RS-485 drivers of the GCB11 via jumpers J1 - J8. These jumpers are mainly used for configuring the line polarity of GCB11s on the RS-485 network. Consult **CHAPTER 2, Hardware** for more information on these jumpers.

6.3 Point to Point using the SPI

The SPI of the M68HC11 is a special high speed synchronous serial connection. When used to connect two GCB11s together over a short distance, no additional line driver hardware is necessary. For more specific information on the SPI please consult the M68HC11 Reference Manual.

The SPI is a flexible communications sub-system and can be used for a variety of applications. When configured as shown in Figure 6.1(c), an additional GCB11 and a dumb terminal become the console to another GCB11. By doing this it is possible to communicate to a particular GCB11 through the SPI without impacting or loading down the SCI or the RS-485 network. The companion disk contains an example program which provides this functionality. Although the preferred method for attaching a console to a node in the network is through the network, using an additional GCB11 and the SPI can prove quite useful when network bandwidth is critical or when the network may be compromised or unreliable.

In order to connect two GCB11s together, it is necessary to create a special cable as shown in Figure 6.2. The pins shown are accessible via JP2, a full pin-out of which is given in Figure 2.9.

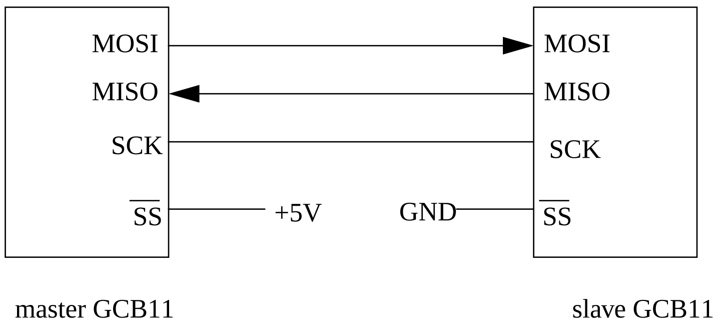


FIGURE 6.2 SPI Master/Slave Cable

In this configuration the master GCB11 acts as the I/O console for the slave GCB11 which is running the application. All console output is routed from the slave GCB11 through the SPI to the master GCB11, and then out the SCI of the master GCB11 to the dumb terminal. Likewise, console input goes from the dumb terminal to the master GCB11 through the SCI, and then to the slave GCB11 through the SPI.

6.4 GCB11 Network Communications

The GCB11 is designed to be used in a multi-drop RS-485 network as shown in Figure 6.1(b) and Figure 6.1(c). The network uses a master/slave configuration and the 9-bit communications mode of the SCI to reduce the overhead incurred on each of the slave nodes. Each node on the network is assigned an unique address between 1 and 63. There is one and only one master node in the network, and in accordance with specifications for RS-485 there can be up to 31 slaves (32 total physical devices).

All communications on the network are controlled by the master node. Each slave on the network is polled by the master to determine if they have any messages they want to send to any other node on the network. If so, the messages are routed by the master from the source node to the destination node. There is a polling table in the master which determines the order in which the slave nodes are polled.

As discussed in **CHAPTER 2, Hardware**, the GCB11 uses additional RS-232 to RS-485 conversion hardware which allows any RS-232 device to be connected to the network via JP6 on the GCB11. It must be noted that when the SCI is used for network communications it is no longer possible to connect a dumb terminal to JP6 and communicate with the GCB11. Therefore, supplied with the GCB11 is additional communications software for the PC which allows it to become either a master or slave node on the network via JP6. Any PC which has its own RS-485 serial communications board can be connected directly to the network cable without the need for a GCB11. When using the RS-232 to RS-485 conversion hardware on the GCB11, there can be as many PCs on the network as there are GCB11s.

In addition to communications drivers, there is software which allows any PC on the network to become a console for any other node on the network. This console facilitates screen and keyboard I/O for the GCB11 to which it is logically connected if the Stream for that GCB11 is configured to use the network. The PC network console tool also possesses additional functionality, such as program download and debug via the GCB11 ROM monitor. This tool is discussed in more detail in **Section 6.6, PC Monitor Program**.

Normally, a node on the network has an address in the range of 1 - 63. The exceptions to this are node addresses 65 - 127, which are reserved for network console monitors. Each address in the range of 65 - 127 is paired with a corresponding address in the range of 1 - 63 which has the same lower six bits in the address byte. For example, node 1 is paired with 65 while node 10 is paired with 74 (each pair is 64 apart).

A network console monitor is used to perform all console I/O for the GCB11 that it is paired with. This is used in conjunction with the network Stream drivers, and thus the two paired nodes in the network are called *Stream partners*. Any console I/O directed through the GNET Stream driver will be sent via GNET to the node address which corresponds to its Stream partner. Likewise GNET directs any messages received from its Stream partner to the GNET Stream driver. It is important to note that the ability to direct Stream I/O through the network only exists in the level 1 network drivers, which are discussed in more detail in **Section 6.7, Writing Applications to Use the Network (GNET)**. The level 0 network routines cannot be used in conjunction with network Stream I/O.

Because Stream partners are fixed, any network monitor can connect to any other node in the network by simply changing its own node address to be the corresponding Steam partner address for the node in question. This is a very powerful feature, and is used by the PC tool to allow it to become a console for any GCB11 on the network.

6.4.1 Connecting the GCB11 to the Network

Connection of the GCB11 to the network is done through JP5. This is a standard 4 pin RJ-11 connector and is intended to be used with twisted pair cables as shown in Figure 6.3. Connector JP5 is the same as that found on all standard telephone handsets.

RS-485 uses differential drivers and receivers. Normally, multi-drop networks which use RS-485 use a single pair of wires in a half duplex party line configuration. The GCB11 uses two pairs of wires in a full duplex configuration. This was done to simplify connecting a dumb terminal to the GCB11 when the network is not being used. Note that it is important that each end of the network bus be terminated by 50 ohm resistors as shown in the diagram.

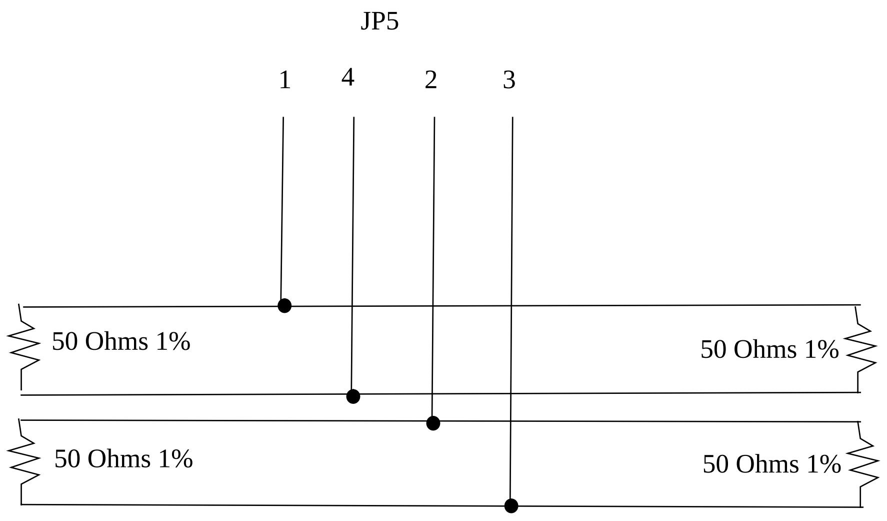


FIGURE 6.3 RS 485 Network Communications Cable

The following is a set of guidelines to be used when making cables for the GCB11 485 network:

- Use twisted pair whenever possible.
- Make sure that the length of cable from the GCB11 board to the 485 bus is kept to a minimum, i.e. a few inches. Making this cable too long causes reflections in the line.
- Make sure that the terminating resistors are 1%. They must be matched as closely as possible in value.
- Standard telephone connectors vary widely in quality. If using telephone equipment make sure it is high quality.
- The RJ-11 socket on JP5 may be removed and replaced by a standard 4 pin in-line header if this will ease cabling for the GCB11.

When making a network it is a good idea to test the cabling using the procedures described in section **6.8, Jumper and Cable Configuration Testing**.

6.4.2 Jumper Configurations for the Network Bus

Each node on the network transmits on a particular pair of wires, and receives on the other pair. Since the network uses a master/slave configuration, the master must transmit on the pair of wires that the slave nodes receive data on, and likewise receive data on the pair of wires that the slave nodes transmit on.

The jumpers used to configure the network for the GCB11 are J5 - J8. These jumpers control which of the differential send and receive lines are connected to the send and receive lines of the GCB11. There are only two basic settings, one for the master and one for the slave, as shown in Figure 6.4.

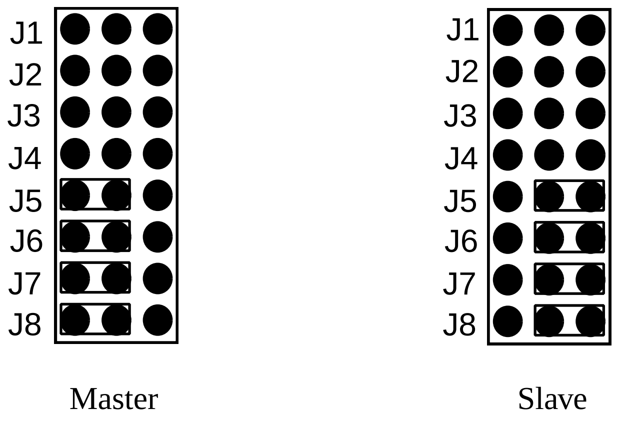


FIGURE 6.4 Jumper Setting for Master/Slave Configuration of GCB11

Since the network is multi-drop, the RS-485 drivers must be disabled when they are not actually transmitting anything on the line. Currently the user has the option of using either PG2 or PD2 to perform this function. One of these two are selected via jumper J13 as shown in Figure 6.5. If the jumper is open, the RS-485 driver will be enabled via a pull-up resistor. This is advisable when the network is not in use. If the network is used then the jumper MUST be in place on one of the two. It is important to note that when the network is in use, the user loses the use of the I/O port selected as the control line.

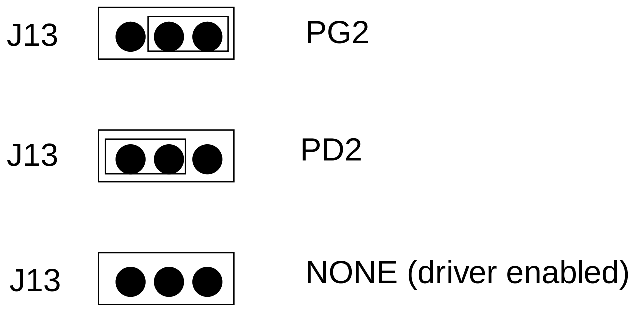


FIGURE 6.5 Selection of RS485 Enable.

These jumpers only affect the cabling and enable line for the drivers. In order to configure the network software properly to behave as a master or a slave and use the appropriate control line, it is necessary to set various parameters in the GCB Registers. This is discussed in more detail in the next section.

6.4.3 Communications and Network Configuration

Various parameters related to the network are stored in the GCB Registers. GCB Registers are described in **Section 3.6**. When the network is initialized, these parameters are used to configure the network.

- `GNETMODE` - Master or slave on the network.
- `GNETNODE` - Node address designation for network operation.

- **GNETPOLL** - Mode of polling for the master node.
- **GNETTABLE** - Polling table.
- **GNETCOUNT** - Number of nodes in the polling table.
- **SCIBAUD** - SCI communications baud rate.
- **SCIDRIVER** - Communications driver for the SCI.
- **SCIENABLE** - Control line used to enable the RS-485 drivers.
- **SCISTART** - Network on or off on start-up.

The first five Registers are described in detail in the following sections. The remaining four are detailed in **Section 3.11.4, Communications**.

6.4.3.1 GNETMODE

This Register designates whether the node is a master or a slave in the network. Each network contains a single master and one or more slaves. Each GCB11 can act as either a master or a slave depending upon how it is configured. It is important that only one of the nodes in the network (including the PCs) act as the master. Failure to follow this restriction will result in a total network failure. The setting of the node to either a master or a slave must be consistent with the setting of jumpers J5 - J8.

If **GNETMODE** is 0 then the node is a slave; otherwise it is a master.

6.4.3.2 GNETNODE

This Register designates the node address of the GCB11. No two nodes on the network may have the same address. The address for any particular node may take on any value in the range 1 - 63.

6.4.3.3 GNETPOLL

This Register specifies the automatic polling mode of the master node. If the value is 0, then the master does no automatic polling to determine if there are additional nodes plugged into the network. If this value is non-zero, it specifies the number of times the master goes through the polling table before doing a poll for a node which is not already in the polling table. This allows for the automatic addition of nodes into the network.

6.4.3.4 GNETABLE

This Register is actually an array of up to 64 bytes which contains the address of each slave node in the master’s polling table. Each node in this table is polled in succession by the master node.

6.4.3.5 GNETCOUNT

This Register gives the number of nodes in the polling table.

6.4.4 Special Bootstrap Mode

If for whatever reason the EEPROM locations where the communications parameters are stored are destroyed, or the GCB11 is configured in such a way that it is difficult to subsequently change the parameters, the GCB11 can be put into a default configuration by placing a jumper between pins 1 and 2 of JP2 and resetting the board. This will cause the GCB11 to come up with the following communications parameters:

- 9600 baud

- Network disabled

These parameters will be stored in EEPROM and essentially put the GCB11 in a known default state. See **Section 3.6, GCB Registers** for more details on this special bootstrap mode.

6.5 Configuring and Using a PC in the Network

6.5.1 Connecting a PC to the Network

Connecting a PC to the network is done using the RS-232 connection on JP6. In this respect, the hardware interface for the network is the same as for normal serial communications, and requires no additional hardware on the PC.

Like the RS-485 driver used by the GCB11, the RS-485 driver which is linked to the RS-232 driver must be configured so that the receive and transmit lines are connected to the appropriate pair of wires. This is depicted in Figure 6.6. As can be seen, the PC can be configured as either a master or a slave on the network.

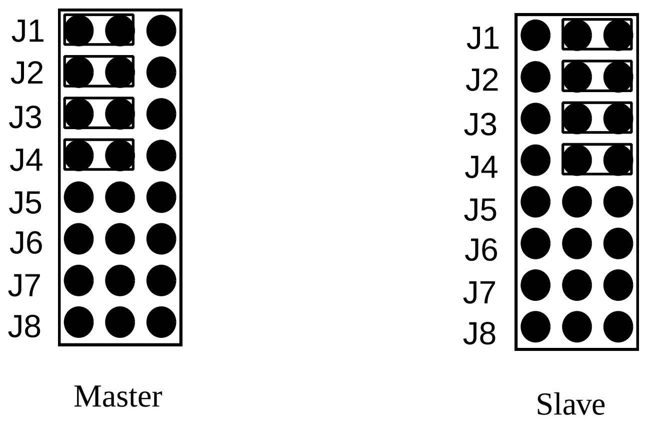


FIGURE 6.6 Jumper Setting for Master/Slave Configuration of PC.

The RS-485 driver used by the PC also requires an enable control line. As previously discussed in **Section 2.5, Asynchronous Serial Communications**, this is accomplished using pin 1 of JP6. The network driver software for the PC toggles both RTS and DTR to enable the RS-485 driver, and either one of these can be selected by the user for enable control. On a DB25 connector, RTS is pin 4, while DTR is pin 20.

6.5.2 PC Network Communications Drivers

In order to use a PC in the network, it is necessary to use the network communications library with your application running on the PC. These functions enable the PC to send and receive messages over the network in a variety of ways. The functions in this library are described in **Section 6.7, Writing Applications to Use the Network (GNET)**, and are the same functions provided in the network library on the GCB11. This enables the user to write portable application code which can run on either the PC or the GCB11.

The drivers on the PC support interrupt driven communications through any of the ports COM1 - COM2. Currently only one of the COM ports at a time may be used for network communications.

6.6 PC Monitor Program

Supplied with the GCB11 is an application which allows the PC to be either a dumb terminal or a network console. The primary purpose of this application is to provide console I/O for the GCB11 through any of the Streams previously discussed.

The major features supported by this program include the following:

- Ability to use the program as either a dumb terminal, a network console, or both simultaneously.
- Ability to split screens to allow up to four console windows, which may be connected to different GCB11s.
- Ability to individually label console windows so that they can be tailored to the individual scenario.
- User configured baud rate.
- Ability to use any of COM1 - COM2.
- Ability to dynamically change the node to which a network console is connected.
- Ability to upload and download S-record programs to the GCB11.
- Ability to log all screen I/O to a file.
- Ability to load and save parameters.
- Ability to control when new devices are polled for connection when using GNET.

6.6.1 Running the Monitor Program

The monitor program is called `gpc` and has the following syntax:

gpc

If the configuration file `gpc.dat` exists when `gpc` is executed, it is used to set up all the parameters used by `gpc`. This allows the user to create a set of default parameters which are used upon start-up. How this file gets created is discussed later in this section.

6.6.2 Monitor Program User Guide

The monitor program can support two separate views, each of which is connected to a different GCB11. Each view is represented by a window on the screen and both can be displayed simultaneously. Data arriving to the PC is displayed in the appropriate view automatically, but keyboard output only corresponds to the currently selected view. The currently selected view is discernible as the one containing the blinking cursor.

All options within `gpc` are available to the user through either a hot key or a menu selection. Options are divided into two main categories: those that apply to the program as a whole, and those that apply to a particular view. Each of the options available are listed below with a brief description.

Main menu F1 - Hitting F1 brings up the main menu. This menu contains the following options:

- **Help F1**
- **Resize view F2**
- **Open views F3**
- **Close views F4**
- **Change view F5**

- **Save parameters ALT-S**
- **Node Search ON ALT-O**
- **Node Search OFF ALT-F**
- **Display version ALT-V**
- **Quit SH-F10**
- **Log screen ON F6**
- **Log screen OFF SH-F6**
- **Program download F7**
- **Node Address F8**
- **View parameters F9**
- **View label ALT-L**

Resize view F2 - This option allows the user to resize and re-position the currently selected window by using the arrow keys.

Open views F3 - This option doubles the number of views currently displayed on the screen. Up to four can be displayed simultaneously.

Close views F4 - Reduces the number of views currently displayed on the screen.

Change view F5 - Allow the user to sequence through and select the currently displayed views.

Save parameters ALT-S - Saves the current parameter settings into the file `gpc.dat`. This file is used upon start-up by `gpc`.

Node Search ON ALT-O - This option is used when gpc is a network monitor. It enables the automatic searching for new nodes as they are plugged into the network. This option only works if gpc is the master on the network. Searching for new nodes incurs an overhead which slows down the overall throughput of the network. Node searching is on be default.

Node Search OFF ALT-F - This option disables the searching of new nodes that may be plugged into the network. This is the companion for the above command. This option should be used after all nodes have been plugged into the network in order to increase network bandwidth.

Display version - Displays the current version of the GPC software.

Quit SHIFT-F10 - Exits the program.

Log screen data ON F6 - This option turns the logging of screen data to a file on. It applies to the currently selected view. When turning on the user is prompted for a file name to which to write the data.

Log screen data OFF SH-F6 - This option turns the logging of screen data to a file off. It applies to the currently selected view.

Program download F7 - Allows the downloading of S-records to the GCB11 connected to the currently selected view. User is prompted for the name of the file to download.

Node Address - This sets the network node address GBUG will be connected to when it is acting as a network monitor. Note that the address entered here should be the address of the node to be monitored. If there are multiple windows on the screen each one can be set to monitor a different node on the network.

View label ALT-L - Each view has a label which signifies the name of the view as well as its configuration. This option allows the user to input a custom name. In this way the user can give more meaningful names to the console windows. For example, the user may display two views at once and want to give a functional name to each of the views depending upon the GCB11 it is connected to, e.g. water pump, proximity sensor.

View parameters F9 - This option is used to bring up a configuration menu which applies to the currently selected view. The menu allows the configuration of the following parameters:

- Network or dumb terminal console.
- Baud rate.
- COM port to use.

By configuring each view separately, it is possible to have a dumb terminal out each of COM1-COM2 simultaneously. The same COM port cannot be used for different views. Another possibility is to connect a view to one of the COM ports as a dumb terminal and connect a different view to another COM port as a network console.

Note that when using gpc as a network monitor the pc must be the master node on the network.

6.6.3 Common Problems

When using gpc as a dumb terminal the following set of problems can occur:

- Receive output from the GCB11 but it does not accept input. This is most commonly caused by a cabling or COM port problem. Because of the nature of the GCB11 communications hardware it is important that the port being used on the PC support hardware handshaking. In addition it must be wired properly so that the appropriate handshake lines toggle the enable line of JP6. If necessary the enable line can be tied to ground.

When using **gpc** as a network monitor the following set of problems can occur:

- No output from the GCB11. The following items should be checked:
  1. Make sure cabling is correct.
  2. Make sure the streams on the GCB11 have been re-directed to the network.
  3. Make sure that **gpc** is connected to the GCB11 via the F8 key.
- The displaying of characters by **gpc** suddenly slows down. This occurs when gpc starts polling for a node which is no longer on the network. This can happen if a node was at one time on the network and then gets taken off or if there is excessive noise in the network and gpc inadvertently adds a node to the polling table which never really existed. The remedy is to exit gpc and restart it. The adding of nodes to the polling table can be controlled with the ALT-F and ALT-O keys as previously described.

6.7 Writing Applications to Use the Network (GNET)

The GROM module GNET provides routines for users who wish to use the network in their applications to send and receive messages. As discussed previously, the GNET drivers can also be used for network Stream I/O. As will be discussed, extra care needs to be taken if the application wishes to send and receive messages AND use the network for console Stream I/O.

There are two levels of routines provided for interfacing to the network: level 0 and level 1. Each level interfaces to the network in a different manner and cannot be mixed within an application. In addition to the two levels, there is a large set of common routines which may be used in conjunction with either level. Briefly, each level can be described in the following way:

- Level 0 - Direct calls to GNET. These are the most efficient means to use the network. Interfaces to the application via call-backs.
- Level 1 - Provide a more transparent and simpler to use interface to the network, but also have more overhead associated with them. Manage receive/send queues and the network Stream I/O.

In order for an application to use the network, the following simple steps must be followed:

- Configure the network parameters if default values in EEPROM are not appropriate.
- Initialize and start the network drivers.
- Connect the application to the network using either a level 0 or level 1 open call.
- Send and receive messages.
- Disconnect the application from the network by using either the level 0 or level 1 close call, depending upon how it was opened.
- Disable the network drivers.

The remainder of this section outlines the functions which are available within each level to perform the steps described above. Please refer to **CHAPTER 9, Programmer’s Reference** for more detailed information concerning each function call.

6.7.1 Common Functions

The following set of routines are common to both level 0 and level 1 and can be used within either context. These routines are used to configure various parameters, initialize the network drivers, and obtain information about them.

`gn_config_node` - Configures the various parameters for the network. These parameters remain in effect until the next call to **`gn_init_comms`** or **`gn_config_node`**. Parameters which can be configured include master/slave, baud rate, node address, and enable line for the RS-485 drivers.

`gn_get_config` - Fills out the structure **`config`** with all the current configuration values being used.

`gn_save_config` - This routine saves all the network configuration parameters in EEPROM. These parameters include:

- master/slave flag
- baud rate
- enable line used for drivers
- node address
- polling table
- auto configure flag

`gn_restore_config` - This routine is used to restore the communications parameters from values stored in the EEPROM. The parameters restored include the same as those given for **`gn_save_config`**.

`gn_init_comms` - Used at start-up to initialize the network drivers. Must be called before any of the level 0 or level 1 routines can be used. This routine sets up the default configuration of the node which was stored in EEPROM unless a call to **`gn_config_node`** has been made.

`gn_exit_comms` - Used to disable and clean up the network drivers. Should be called before exiting any application that is using the network.

`gn_get_stats` - Returns the network statistics such as number of retries and time-outs currently experienced.

`gn_reset_net` - Resets and initializes the node to its start-up state.

`gn_suspend_poll` - If the node is a master then polling of the slave nodes is suspended until the network is reset or the **`start_poll`** function is called.

`gn_start_poll` - If the node is a master then the slave polling sequence is started. Each node in the polling table is queried in succession.

`gn_set_poll` - If the node is a master then this routine sets up a polling table which is supplied by the application.

`gn_get_poll` - If the node is a master then this routine returns a pointer to the polling table used to poll the slaves.

`gn_add_poll` - This routine adds a node address to the end of the polling table.

`gn_auto_poll` - This function enables automatic searching for nodes which are not explicitly put into the polling table by the application. This allows nodes to be added to the polling table automatically when they are plugged into the network. The major drawback of using this feature is that it will impact network performance, because in cases where the master polls for nodes that do not exist, there will be an inordinate delay while the master times out waiting for a response. There are additional parameters which can be supplied to lessen the impact of this effect.

6.7.2 Level 0 Functions

Level 0 functions represent the closest connection to the network with the least overhead. They should be used only when level 1 functions are not appropriate. Typical reasons why level 1 functions may not be appropriate include:

- There exist actions which must be performed IMMEDIATELY upon receipt of a message.
- Handling of messages received should happen completely in the background as opposed to polling a receive message queue.
- Application needs tighter control over memory management.

Level 0 routines exist for sending messages, but the method for receiving messages involves the application installing a call-back routine when the connection to the level 0 drivers is first established. The call-backs are supplied by the application and will be called by the network drivers due to the following events:

- message received
- message sent
- incoming message

Care must be taken to ensure that these call-back routines are written in a brief and efficient manner, since they will be called in the middle of an interrupt service routine. Assembly language may be necessary for more involved routines.

Level 0 functions cannot be used if the network Stream I/O is already in use. Likewise, if level 0 functions are already in use by the application then network Stream I/O cannot be used.

The following level 0 functions are supported.

`gn_open_port_l0` - This function is used to connect the application to the level 0 network drivers for message handling. It installs a series of call-back routines which are supplied by the application. This routine must be called after **`gn_init_comms`**, but before any messages are to be processed. The call-back routines are executed whenever a message is about to be received, or when a message is finished transmitting or receiving. This provides maximum flexibility in the application memory management.

`gn_close_port_l0` - This function is called to disconnect an application from the network services while leaving the network running.

`gn_send_message_l0` - This routine is used to send a message to another node in the network. The application simply specifies the message to send, the destination node address, and whether to block while waiting for it to be sent.

6.7.3 Level 1 Functions

Level 1 functions build upon the level 0 functions to provide a more transparent interface to the network. The difference between the level 0 and level 1 interface is the way the port is opened and the way messages are sent and received.

The level 1 functions eliminate the need for the application to supply call-back routines to do the message queue management. Essentially all incoming and outgoing messages are put into message queues, and the memory they require is managed by the level 1 library routines. Messages are received by the application using a pair of routines which can check if any messages are in the message queue, and can get messages from the message queue. This simplifies reception of messages, but also means that the message queue must be polled by the application.

One of the most fundamental differences between level 0 and level 1 functions is the way that the memory used to receive and transmit messages is managed. In level 0 functions the application has complete control over the allocation of memory to be used for messaging. By contrast, level 1 functions malloc memory for messages and copy them around. This puts certain restrictions on the size of messages that can be sent and received, and may fragment the applications heap.

In order to compensate for the loss of control over memory management, the level 1 functions require the application to provide the routines which will be used to both malloc and free message memory. In the most trivial case, the application can simply specify the standard library functions **`malloc`** and **`free`**. In applications where memory management is an issue the application can provide its own malloc and free routines in order to maintain tighter control over memory usage.

As mentioned earlier, the network Stream drivers are integrated within the level 1 network drivers. This allows the application to connect to the Stream I/O drivers as well as the network Stream drivers. When level 1 drivers are used and the network Stream I/O is enabled, all messages received from the node's Stream partner are directed to the Stream I/O buffers.

The following level 1 routines are supported.

`gn_open_port_l1` - Used to initialize and open a level 1 port for communications. This should be called before any other level 1 routines, but after any node configuration which may be required. The parameters supplied by the application include a malloc and free routine to be used for memory management, as well as a receive queue size.

`gn_close_port_l1` - Should be called before exiting an application which uses the network. The network drivers and port will be closed and disabled.

`gn_send_message_l1` - Used to send messages to other nodes in the network.

`gn_get_message_l1` - Used to get the next available message from the receive message queue.

`gn_message_rdy_l1` - Used to determine the number of messages in the receive message queue.

6.8 Jumper and Cable Configuration Testing

There is a simple set of tools provided with the GCB11 for the PC which will quickly test the following elements of your network configuration:

- Cabling from the PC to the GCB11 (JP6).
- Cabling between the GCB11s in the network (JP5).
- Master/Slave jumper settings (J1 - J8).
- Enable line jumper setting (J13).
- Enable line configuration in **`SCIENABLE`**.

To run this test, it is necessary to use a GCB11 as an SPI terminal off of each of the GCB11s that you want to test. From a practical standpoint, it is only necessary to make a single SPI terminal and swap it among the GCB11s on the network. In order to set up a network and test the configuration, perform the following steps:

- Through the SCI terminal interface, configure the GCB11s on the network to use the SPI console Stream by running the GBUG **`BD`** command.
- Assemble the SPI cable and connect to an SPI terminal on the GCB11s that you want to test. If the GBUG prompt appears when the GCB11 is reset then the SPI terminal is properly configured and connected.
- Configure the jumpers on each of the networked GCB11s for master or slave (J1 - J8).
- Configure J13 to use the desired enable line.
- Set up the GCB Register **`SCIENABLE`** to correspond to the setting of J13. This can be done through the GBUG monitor.
- Make all network cables and connect the GCB11s via JP5.
- Make an RS-232 cable and connect the PC (if any exists) via JP6.
- On the GCB11, run the **`ST`** GBUG command.
- On the PC, run **`com_tst`**. Be sure to specify the COM port (0-3) which is being used on the PC.
- Simply type characters. Any characters which are typed on any of the slave consoles should appear on the master console. Likewise, all characters typed on the master console should appear on ALL of the slave consoles.

The following is a list of common problems and possible solutions.

- When typing on the PC, nothing appears on any console. Be sure to check the cable between the PC and the GCB11. Pay special attention to the enable signal.
- When typing on a GCB11, nothing appears on any console. Be sure that the jumper setting of J13 matches the value in the GCB register **`SCIENABLE`**. Also check the cabling coming out of JP5.
- When typing on a slave node, output appears on multiple consoles. More than one of the GCB11s is configured as a master. Also check the cable coming out of JP5. It is possible that there are wires crossed.

- When typing on any of the consoles, garbage appears on the other consoles. This is indicative of a cabling problem. Possible problems include reversed wires on the cable, or line termination problems. Consult section **6.4.1, Connecting the GCB11 to the Network** for more information on making cables.

CHAPTER 7 GAPP

7.1 Introduction

This chapter describes the GCB11 Application Routines, or GAPP, which supply miscellaneous support functions for applications. GAPP is supplied in source code form on the Companion Disk and should be compiled and linked with user programs. GAPP includes such items as PWM motor control and event counting. Each application is listed in the sections which follow.

7.2 Motor Control

The motor control library is designed to perform a wide variety of control functions on DC and stepper motors. These functions are written to be completely interrupt driven, and to run as efficiently as possible. This enables the application to run freely in the foreground, while all essential motor control functions are running transparently in the background. All that is needed from the application is the issuing of commands to make the motor perform the desired task.

The motor control library uses the output and input compare functions of PORT A on the M68HC11. Because of this, it is possible to simultaneously perform pulse width modulation (PWM) on up to four motors. In addition, three of these motors can receive feedback and perform position control via encoders attached to the input capture pins.

7.2.1 Motor Control Basics

This discussion will concentrate on DC motor control since it is the simplest and most widely applicable. Most of this discussion can be applied to stepper motors as well but the application is more involved and dependent upon the type of stepper motor used.

Motor control consists of the following components:

- Motor to be controlled.
- Motor driver to convert from logic level signals to power signals appropriate to drive the motor.
- Control logic to control the motor’s speed, direction, and position.
- Feedback used to control some variable of the motor, such as speed or position.

Feedback is not required and when omitted the control is referred to as *open loop*.

The most common method for performing speed control is the use of pulse width modulation (PWM). This is simply a stream of pulses sent to the motor at a fixed frequency, as shown in Figure 8.1. The amount of voltage or power available to the motor as a result of these pulses is the average area under the pulse over an entire pulse period. Therefore, the wider the pulse width, the greater the voltage delivered to the motor, which results in higher speeds and torque. There are a number of engineering reasons for using PWM, but perhaps the most compelling is the relative ease with which highly accurate pulse widths can be generated and controlled.

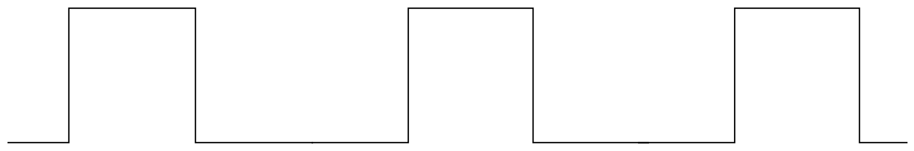


FIGURE 8.1 Pulse Width Modulation

The output of the motor driver is usually connected to the DC motor (permanent magnet) using two wires. These wires connect to the motor's rotor windings and thus form a closed circuit. The direction of current through the wires determines the motor's direction.

The input to the motor driver can take many forms. In addition to the PWM signals, there are usually additional control signals for things such as direction, enabling the motor, dynamic braking, etc. The motor control library supports up to four PWM lines and any number of control signals, assuming that all control lines for a single motor are on a single port.

7.2.2 PWM Port Assignments

In order to perform PWM, the motor control library relies on the timer functions which are tied to port A. Therefore, the programmer must avoid using the associated timer functions when also using the motor control library. Up to four PWM signals can be supported simultaneously, and each of these are hard wired to a particular logical motor number. Each of the particular timer functions and port pins used for each logical motor are shown in Table 8.1. Note that a logical motor only refers to a single set of PWM/control lines. It is possible to have two PWM signals tied to a single physical motor. This is often necessary when the two halves of a single H-bridge driver are isolated and require separate control lines. It is often easiest to simply connect a separate PWM signal to each side of the H-bridge, and control each side of the H-bridge separately through software.

Logical Motor	PWM output	Registers Used	Interrupt Vector
0	PA6 (OC2)	TCTL1-OM2 TCTL1-OL2 TFLG1-OC1F OC1M6 OC1D6 TOC1	intvtmroc1
1	PA5 (OC3)	TCTL1-OM3 TCTL1-OL3 TFLG1-OC1F OC1M6 OC1D6 TOC1	intvtmroc1
2	PA4 (OC4)	TCTL1-OM4 TCTL1-OL4 TFLG1-OC1F OC1M6 OC1D6 TOC1	intvtmroc1
3	PA3 (OC5)	TCTL1-OM5 TCTL1-OL5 TFLG1-OC1F OC1M6 OC1D6 TOC1	intvtmroc1

TABLE 8.1 Timer and Port Assignments for PWM Output

The PWM is driven off of the OC1 interrupt. On each OC1 interrupt the corresponding pins for each PWM signal are either set, reset, or toggled, depending upon the state of the PWM for that motor. For the most part, this is handled automatically by the timer control register TCTL1. Motors 0 - 3 use outputs OC2 - OC5 (PA6 - PA3) of port A. Therefore, when using a particular motor it is important for the application to avoid using the corresponding control and flag bits for that output in all relevant registers. These registers include TCTL1, TCTL2, TFLG1, and TMSK1.

7.2.3 Motor Control Functions

The motor control library is built around the concept of a *logical* motor. To this logical motor is assigned a single PWM signal and numerous user defined control signals. As discussed in the previous section, the PWM signal and the logical motors are hard wired together; everything else is completely flexible. The following library routines are supported:

`ga_setup_motors` - Used to initialize and set up the motor drivers with the base frequency.

`ga_init_motor` - Used to initialize a particular logical motor. Parameters such as PWM polarity and additional direction control lines are supplied.

`ga_motor_speed` - Used to set the motor speed to a specific value.

`gs_motor_dir` - Used to set the motor direction via the additional direction control lines specified by the user.

`gs_motor_reverse` - Reverses the direction of the motor.

`gs_motor_faster` - Increases the speed of the motor by a specified amount.

`gs_motor_slower` - Decreases the speed of the motor by a specified amount.

`ga_motor_state` - Return the current state of a motor.

7.3 Digital Input

Digital input is defined as input to a single digital I/O pin on the GCB11. There are up to 20 digital I/O lines available for this purpose. The following functions can be performed using the digital input routines provided with the applications library:

- Count events such as pulses
- Read and debounce digital inputs

Each of these are described in more detail in the sections that follow.

7.3.1 Event Counting

There are four logical counters available for event counting, and each of these are hard wired to specific ports on the GCB11. The routines are flexible enough to be used to perform counting functions ranging from a simple event counter to an up-down counter with reset for quadrature encoders. The ports and registers used to perform event counting are shown in Table 8.2.

Logical Counter	Port used for Events	Registers Used	Interrupt Vector
0	PA2	DDRA-DDA3 TCTL2-EDG1A TCTL2-EDG1B TMSK1-IC1I TFLG1-IC1F	intvtmric1
1	PA1	DDRA-DDA1 TCTL2-EDG2A TCTL2-EDG2B TMSK1-IC2I TFLG1-IC2F	intvtmric2
2	PA0	DDRA-DDA0 TCTL2-EDG3A TCTL2-EDG3B TMSK1-IC3I TFLG1-IC3F	intvtmric3
3	PA3	DDRA-DDA3 TCTL2-EDG4A TCTL2-EDG4B TMSK1-I4O5I TFLG1-I4O5F PACTL-I4/O5	intvtmroc5

TABLE 8.2

The following routines are available for counting events:

`ga_start_counter` - Used to initialize and start the event counter.

`ga_stop_counter` - Used to stop and disable the event counter.

`ga_set_counter` - Used to set the counter values.

`ga_get_counter` - Used to retrieve the current counter values.

A logical counter is initialized with the following parameters:

- Positive or negative edge trigger of the events to be counted.
- Optional auxiliary I/O port which determines whether the counter will go up or down when an event is detected.
- Positive or negative reset pulse for zeroing out the count.

All counting and event monitoring occurs in an interrupt driven fashion without need for any intervention from the application. If an auxiliary port is specified it will be used to determine whether the counter will count up or down when an event is detected. When the event is detected the counter driver will read the auxiliary port. If it is high the driver will increment the count. If it is low the count is decremented. This is typically used for quadrature encoders which have two channels of pulse trains that are out of phase in a manner dependent upon the direction of revolution of the encoder. When used in this manner, one of the channels is connected to the event monitor and the other becomes an input to an auxiliary port which is defined by the user.

The reset pulse is useful when there is a reference pulse which can be used to set the counter to zero. Encoders often have an index pulse which can be used for this purpose.

CHAPTER 8 Building Applications

8.1 Introduction

This chapter serves as a general reference for users developing and running applications using GROM. In order to develop code for the GCB11 it is necessary to take into consideration the following issues, each of which is discussed in a section below:

- Examples
- Header Files
- Memory Map
- M68HC11 Resources
- Initialization
- Using GROM Routines
- Using Third-Party Libraries
- Running applications from GBUG
- Building Custom EPROMs

While this chapter does not refer to or depend upon any specific compiler or assembler being used for development, examples on the companion disk are provided for several free and commercial C compilers/development environments.

8.2 Examples

The primary guide for developing applications for the GCB11 is the **`examples`** directory on the companion disk. These examples contain source code, makefiles, and linkfiles for building complete applications to run both from GBUG and in custom stand-alone EPROMs. If you are using one of the compilers for which examples are given, the easiest path to completing an application may well be to modify one of the example programs. Even if there are no specific examples for the compiler you are using, it will probably be easier to port the example files than to start from scratch.

The simplest example program, and a good place to start, is **`hello`**. **`hello`** prints the text “hello world” to the STDIO stream. Each of the other examples demonstrates or tests one or more of the functionalities of GROM. Usually an application is initially developed in RAM and run from GBUG, even if it will eventually be put into EPROM and run stand-alone. The **`hello`** example makes two versions of the program, one that can be run from GBUG and another that can be used to make a stand-alone EPROM. One other technique that can be used is to store the code in EEPROM or EPROM and use the GBUGRR GCB Register to transfer execution to it on reset. See the **`spi_mon`** example for more details on this.

8.3 Header Files

The GCB11 comes with a set of C and assembly header files on the companion disk which should be used when writing applications. Each library function in GROM is declared in one of these headers. Each header declares a set of related functions, plus any necessary types and macros needed to facilitate their use. All C header files have the **`.h`** extension, while the corresponding assembly header files have the **`.inc`** extension. See the **`README`** file in the root directory for the locations of these files on the disk.

8.3.1 System Headers

These header files are used to define all the system constants and definitions, such as the location of M68HC11 registers. They must be included by all C and assembly routines.

- **`coactive.h, coactive.inc`**
- **`gcb11.h, gcb11.inc`**

8.3.2 GROM Headers

There is a header file for each major subsystem within GROM. Refer to the appropriate chapter for a discussion of a specific GROM module. The corresponding header file should be included by any source file which uses functionality from that subsystem.

- **`gapp.h, gapp.inc`** - Application library definitions.
- **`gios.h, gios.inc`** - GCB11 specific definitions used by the GIOS subsystem.
- **`gnet.h, gnet.inc`** - Definitions used by the network drivers.

8.3.3 Standard C Library Headers

As discussed in **CHAPTER 4, GIOS**, the GIOS module contains a subset of the ANSI Standard C Library functions. ANSI C compatible header files for these routines are located on the companion disk. The following Standard C Library header files are shipped with the GCB11:

- **`ctype.h`**
- **`limits.h`**
- **`stddef.h`**
- **`stdio.h`**
- **`stdlib.h`**
- **`string.h`**

For a complete description of these files, see any reference on the ANSI C Standard (e.g. *The C Programming Language* by Kernighan and Ritchie and *The Standard C Library* by Plauger)

Note that some files are missing from the full Standard C implementation, and some of the files included are not complete implementations. A common practice is to use the standard library supplied with the C compiler which is being used for development for any additional library functions which are needed. Since the names of these files may follow the same ANSI Standard, the headers from the two source, GROM and the C compiler, will need to be merged. Examples of this merging are supplied on the companion disk for several compilers.

8.4 Memory Map

GROM reserves blocks of RAM, EEPROM, and EPROM for its own use. Refer to the memory map in **Section 3.2** to ensure that your application does not clash with memory used by GROM.

When using RAM, applications must be careful of memory reserved by GROM data and any locations which are being used by the M68HC11F1. These include the I/O registers at the bottom of page zero, the blocks of code used by the I/O chip selects if these are being use, and the EEPROM located at 7E00 hex. Also note that the GCC 3.2.2 compiler uses locations 74 hex through 8C hex for pseudo registers.

EEPROM is free except for the block at the bottom reserved for the GCB Registers.

When making an EPROM version of an application that uses GROM, the application code must be placed in memory along with the GROM code. The companion disk contains S-record files for GROM in several configurations (see **Section 8.10** for more details). GROM code always starts at the bottom of EPROM (8000 hex) and uses a single block of memory. The size of each GROM code S-record file is listed in the `readme` file on the companion disk. In addition to code, GROM uses the top of EPROM for the call table, interrupt vectors, and any constant data it requires. These two blocks of reserved EPROM leave a block of free EPROM between the end of GROM code and the beginning of GROM Constant Data. This free space is where EPROM applications must reside.

C compilers often allow the use of multiple segments within the application. These segments are usually divided into code and data segments. The linker is then responsible for the placement of these segments within the memory map. The management of segments is left entirely up to the user, and is restricted only by the compiler and the GROM memory map. Examples exist on the companion disk for several compiles and linkers. Porting these to a new development environment should be straight forward in most cases.

The examples programs which run from GBUG (with the exception of `spi_mon`, which is placed in EEPROM) all start at 100 hex and grow upward. The code is located at the bottom and followed by immediately by data and stack space. EPROM example code starts immediately after the last byte of GROM code, and EPROM example data starts at 100 hex.

8.5 MC68HC11 Resources

The GROM routines use many of the registers, interrupt vectors, and ports available on the GCB11. It is important that the application not interfere with these resources if they intend to also use the corresponding GROM routines. For example, if the application reprograms the SCI interrupt, this would interfere with the normal Stream interrupt and cause console I/O to fail. The application is obviously free to modify any register on the GCB11 as long as it is ready to accept the consequences of losing the corresponding GROM functionality. The M68HC11F1 resources used by each GROM code module are listed in a table in chapter covering that module. The I/O registers listed in **Section 3.3** cannot be modified, or GROM will not function.

8.6 Initialization

Several hardware and software systems on the GCB11 need to be initialized before user applications using GROM can run. Initialization can be divided into five steps:

- Set up MC68HC11F1 I/O registers
- Set up MC68HC11F1 stack pointer CPU register
- Check MC68HC11F1 CONFIG register for corruption
- GROM initialization
- Compiler/assembler start-up

Each of these steps is detailed in a section below. Note that if the application is being run from GBUG, only the second and last steps need to be performed (the rest are handled by GBUG at reset).

The companion disk contains two example assembly language files for application initialization: **`startrom`** and **`startram`**. **`startrom`** contains all five steps and also includes the interrupt vectors (see **Section 8.10, Building a Customized EPROM**) and should be used when making stand-alone EPROMs. **`startram`** contains steps two and five only and should be used when making applications which run from GBUG.

8.6.1 I/O Registers

GROM requires certain MC68HC11F1 I/O registers to be configured before they can function. See **Section 3.3, M68HC11 I/O Registers** for details on which registers these are and what values they need to be set to. Note that some of these registers must be changed in the first 64 E cycles after the processor is reset. See the M68HC11 Reference Manual for more details.

8.6.2 Stack Pointer

Before an application can make function calls or use stack operations, the MC68HC11F1 Stack Pointer (SP) CPU register must be initialized. A block of memory for the stack should be reserved and the SP register should be set to point to the end of this block (the stack grows down in memory). The GROM stack requirements and initialization are discussed in **Section 3.10** and **Section 3.11** respectively.

8.6.3 CONFIG Register

The MC68HC11F1 CONFIG I/O register is a special case because it cannot be changed without special EEPROM programming and resetting of the CPU. The example start-up code on the companion disk demonstrates one method of dealing with this problem.

8.6.4 GROM Initialization

GROM initialization consists of the following steps:

- Initialize Interrupt Vector Table
- Check for corrupted GCB Registers, copying the Registers from EPROM to EEPROM if they are corrupted
- Copy GCB Registers from EEPROM into RAM
- Initialize Stream driver

- Initialize Stream and assign it to `STDIO`

All of these operations are described in **CHAPTER 3, GROM** and **CHAPTER 4, GIOS**.

8.6.5 Compiler/Assembler Start-up

Most GCB11 applications will need additional initialization which is required by the compiler or assembler that is being used for development. This is often done in a start-up routine supplied with the compiler or assembler. The following functions are normally performed within a C language start-up routine:

- Set up heap for the application.
- Copy static variables into RAM.

GROM places no restrictions upon what is performed within this module.

Note that GCC 3.2.2 does not initialize global data when the code is in EPROM.

8.7 Calling GROM Routines

The resources provided in GROM exist in EPROM and therefore do not need to be linked directly with the application. This enables the application to be written in either C or assembly language, although the routines are designed to allow easy interfacing to C. Refer to **Section 3.9, Call Table**, for the calling conventions and memory locations of all the GROM routines. **Section 3.9** also describes the use of wrapper functions in assembly, which convert the arguments from a C compiler to the GROM argument format.

Note that a more efficient method of calling GROM functions than using wrappers when no argument translation is needed is to use C macros or assembly `EQU` statements to directly call the GROM call table. This technique eliminates a redundant `JMP` instruction.

The companion disk contains libraries of wrapper functions for several compilers. These libraries are divided by code module and can be linked into the application as needed.

8.8 Third Party Libraries

Most applications will use library functions from third party libraries, such as the libraries shipped with a compiler, in addition to the functions in GROM. This presents no problems unless there is a name conflict with the GROM function. This can occur in two situations:

- Both functions provide the same functionality. In this case, listing the libraries of GROM wrapper functions before third party libraries will insure that the GROM functions will be used in place of the third party library functions when naming conflicts occur. Reversing the order of the libraries on the link line will reverse this situation: the third party library functions will be used in place of the GROM functions. To selectively use functions from third party libraries in place of GROM functions, the wrapper functions for the specific GROM functions not needed can be removed from the appropriate library of GROM wrapper functions.
- The functions provide different functionality and just happen to be named the same. In this case, the easiest solution is to rename the GROM function in the library of GROM wrapper functions and use this new name to reference the GROM function in the application.

8.9 Running from GBUG

When the distribution GROM EPROM is socketed and the GCB11 is reset, control is passed directly to GBUG. Normally, the GBUG sign-on and prompt will appear unless the GBUGRR GCB Register has been used to automatically run a program at a specified memory address. If the application resides in RAM, it must be downloaded before it can be executed. This is accomplished by using the GBUG **TD** command. GBUG accepts S-records for download purposes. An S-record file can be the result of either an assembly or a link. Once the code is downloaded, control is passed to it by simply issuing the GBUG **R** command. See **CHAPTER 5, GBUG** for more details.

Each example program has a **`readme`** file associated with it. The **`readme`** explains what the example does and how to compile, assemble, download, and run the code. This process can be generalized as follows:

- Clean the old objects and S-record files from the directory. This is handled by the “clean” target of the makefile.
- Make the application. This is handled by the default target of the makefile.
- Download the program to the GCB11 using GPC. From the GCB11, type “TD” and press return. Transfer the program from the PC using a terminal emulation program (from GPC, press <F7>, enter the file name, and press return).
- Run the program. All examples on the companion disk start at 100 hex. From GBUG, type “R 100” and press return.

8.10 Building a Customized EPROM

There are two types of EPROM applications: those that run from GBUG and those that run stand-alone. A stand-alone program will begin executing each time the GCB11 is reset.

8.10.1 ROMable Code

For an application to run in EPROM, it must be *ROMable*, which means it must:

- Never write into code space
- Access constant data in EPROM and volatile data in RAM
- Copy initialized data from EPROM into RAM upon start-up

Most compilers have a flag which will generate code that is ROMable. It is up to the user to ensure that any assembly language code is ROMable.

8.10.2 Using GROM Resources

From the point of view of the GCB11 and the GROM, developing code to run in an EPROM is no different than developing code to run in RAM. The only difference is the location of the application in the memory map. GROM routines are still called in exactly the same manner.

8.10.3 Interrupt Vectors

If GBUG is not included in the EPROM, an interrupt vector table will need to be placed at the top of EPROM. For GROM to operate, all interrupt vectors except reset should point to the GROM Interrupt Jump Table. The reset vector should point to the code in EPROM which is the entry point for the application. The file **`startrom`** on the GCB11 companion disk contains a sample vector table.

8.10.4 GROM S-Records

The companion disk provides S-record files of GROM to allow custom EPROMs to be built. When building a custom EPROM, two GROM S-record files will need to be downloaded to the programmer in addition to the application S-record file(s):

- GROM code S-record file. In order to give the user maximum flexibility in garnering EPROM space to hold the application, there are a number of GROM versions in S-record files delivered with the GCB11. These files allow the user to use only the GROM code modules needed for an application, in order to leave the maximum amount of free space in the EPROM. Each S-record file is named according to which GROM code modules it includes.
- Constant data S-record file. In order to make GROM code S-record files crystal speed independent, GROM Constant Data is supplied in separate S-record files, one for each common crystal speed. If an application needs to run on a GCB11 with a crystal speed for which there is no supplied S-record file, one can easily be built by using the **`const`** example as a guide. See **Section 3.8** for more details on GROM Constant Data.

The GROM S-record files are located in **`gen/gen/bin`** on the companion disk. See the **`readme`** file for details on what each file contains.

Note that an offset of 8000 hex should be used when downloading the S-records to an EPROM programmer, since the EPROM is located at 8000 hex in the memory map.

CHAPTER 9 Programmer's Reference

9.1 Introduction

This chapter gives a detailed description of every function available in the GROM except the GIOS Standard C Library Functions. The Standard C Library include files on the GCB11 companion disk contain ANSI C prototypes for each of the functions. For a complete description of these functions, see any reference on the ANSI C Standard, including *The C Programming Language* by Kernighan and Ritchie and *The Standard C Library* by Plauger.

All function descriptions are given in C syntax. See **Section 3.9** for a description of GROM function calling conventions from assembly language.

Functions in each section are listed alphabetically. Note that some descriptions contain more than one function.

9.2 Reference

Each GROM function which is not part of the standard C library is preceded by a two-letter prefix which identifies which GROM module the routine belongs to. The following prefixes are used:

- ga -GAPP module.
- gi - GIOS module.
- gn - GNET module.

ga_get_counter ga_get_counter

Name: **ga_get_counter** - get the current event counter value.

ga_set_counter - set the current event counter value.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

ga_get_counter(int *counter***, COUNT_REC ****count_data***)**

ga_set_counter(int counter, unsigned int *count_value***, unsigned char** *zero_hit***)**

counter	Logical counter. Must be in the range of 0 - 3.
count_data	Pointer to structure containing the current count value and the current *zero_hit* value. See `gapp.h`.
count_value	New value to put into the counter.
zero_hit	New value to put into the zero hit flag.

Description: These routines are used to get and set the current counter values. The flag zero_hit is set whenever the zero index pulse is detected. This is useful on start-up and initialization of a device.

Return Value: SUCCESS or FAILURE

Examples:

ga_init_motor ga_init_motor

Name: **ga_init_motor** - initialize a particular logical motor.

ga_setup_motor - initialize the PWM frequency.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

ga_init_motor(unsigned char *motor_num***,**
unsigned char ****dir_port*,**
unsigned char *dir_mask***,**
unsigned char *dir_right***,**
unsigned char *dir_left***,**
unsigned char *polarity***);**

ga_setup_motors(unsigned int *max_speed***)**

motor_num	logical motor number.
**dir_port*	port to use for direction control.
dir_mask	bit of direction port to use.
dir_right	bit pattern for right rotation.
dir_left	bit pattern for left rotation
polarity	polarity of PWM.
max_speed	maximum number of cycles per pulse.

Description: These functions are used to setup and initialize the PWM signals that are used to perform motor control. Function **ga_setup_motors** must be the first motor control routine called. The parameter *max_speed* is used to generate the PWM frequency. It represents the number of timer ticks (E cycles) per PWM cycle. Once set the same PWM frequency is applied to all logical motors.

The function **ga_init_motor** is used to configure a single logical motor (PWM signal). As well as specifying which motor to initialize an optional direction port can also be specified. This is used when there are additional direction control signals associated with the PWM signal. Any number of bits from a single port can be used for direction control. The mask *dir_mask* specifies which bits are to be used while *dir_right* and *dir_left* specify what the particular bit patterns should be for the desired directions. If no direction lines are required then *dir_port* should be set to zero.

The polarity of the PWM signals can be set using *polarity*. If *polarity* is zero then the PWM signal is active low meaning that a speed specification of *max_speed* will be a constant output of zero on the PWM output line. Likewise if *polarity* is non-zero then a speed specification of *max_speed* will be a constant output of high on the PWM output line.

Return Value: SUCCESS or FAILURE.

See Also: **ga_motor_dir(), ga_motor_speed(), ga_motor_state().**

ga_motor_dir ga_motor_dir

Name: **ga_motor_dir** - set the direction of the motor.

ga_motor_reverse - reverse the current direction of the motor.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

int motor_dir(int motor_num, int dir)

int motor_reverse(int motor_num)

motor_num	logical motor number.
dir	direction of the motor.

Description: The function **motor_dir** sets the direction of the motor to either **MOTOR_LEFT** or **MOTOR_RIGHT**. These correspond to the direction bits specified by **ga_init_motor**.

Function **ga_motor_reverse** reverses the direction of the motor. **CAUTION:** Extreme care should be taken when using this function since reversal of a motor going full speed will often cause serious damage to the motor driver circuitry.

Return Value: none

See Also: **ga_init_motor(), ga_motor_speed(), ga_motor_state().**

ga_motor_speed ga_motor_speed

Name:

ga_motor_speed - set speed of the motor.

ga_motor_faster - make motor go faster.

ga_motor_slower - make motor go slower.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

int ga_motor_speed(int *motor_num*, **int** *speed***)**

int ga_motor_faster(int *motor_num*, **int** *speed_inc***)**

int ga_motor_slower(int *motor_num*, **int** *speed_inc***)**

motor_num	logical motor number.
speed	speed of the motor.
speed_inc	amount to increase/decrease speed.

Description:

These functions change the speed of the motors. The function **gn_motor_speed** sets the absolute speed of the motor. The value of *speed* must be between 0 and the value *max_speed* that was specified when the function **ga_setup_motor** was called. The resulting pulse width after **gn_motor_speed** is called is equal to *speed / max_speed.*

The functions **ga_motor_faster** and **ga_motor_slower** increase and decrease the speed of the motors respectively. The speed of the motor is changed by the amount specified in *speed_inc.*

Return Value: SUCCESS or FAILURE.

See Also: **ga_init_motor(), ga_motor_dir(), ga_motor_state().**

ga_motor_state ga_motor_state

Name: **ga_motor_state** - get current state of a motor.

ga_motor_record - return pointer to internal motor control record.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

int ga_motor_state(int *motor_num*, **int** **speed,* **int** **direction***)**

int ga_motor_record(int *motor_num*, **MOTOR_REC** ***motor_recs***)**

motor_num	logical motor number.
speed	current speed of the motor.
direction	current direction of the motor.
motor_rec	pointer to internal motor control record.

Description: **ga_motor_state** returns the current *speed* and *direction* of the motor specified. The units are the same as those used when the motor speed is set using **ga_motor_speed**.

ga_motor_record returns a pointer to the internal motor control record. This can be used if there are type of control or information necessary that is not provided by the other motor control interface routines. Great care should be taken in modifying this record.

Return Value: SUCCESS or FAILURE.

See Also: **ga_init_motor(), ga_motor_dir(), ga_motor_speed().**

ga_start_counter ga_start_counter

Name: **ga_start_counter** - start event counter.

ga_stop_counter - stop the event counter.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gapp.h>

int ga_start_counter(int *counter*, **char ****dir_port*, **unsigned char** *dir_mask*,
int *edge*, **int** *zero*, **int** *zero_edge*, **unsigned int** *max***)**

int ga_stop_counter(int *counter***)**

counter	Counter id. Must have values 0 - 3.
dir_port	Port containing auxiliary direction bits.
dir_mask	Bits of *dir_port* to be used for the count direction.
edge	Determines which edge triggers the event. If zero then falling edge, else rising edge.
zero	Logical zero number for zeroing index pulses. Must be different from counter, but also have values from 0 - 3.
max	Maximum counter value.

Description: These routines are used to start and stop an event counter. An event is defined to be either a rising or falling pulse edge. There are up to four event counters supported and each one is hardwired to the corresponding input capture pin on the 68HC11. Counter 0 corresponds to input capture 1 while counter 3 corresponds to input capture 4.

The auxiliary port is used to set the direction of the count on each event. If the auxiliary input is high the count is incremented while if it is low it is decremented.

The *zero* parameter is used to reset the count when another event occurs. This event also tied to one of the input capture pins and thus must be different than *count* or any other event counter which may be active.

There is a flag associated with each counter called *zero_flag* that is set every time the zero pulse is detected. The application may reset this flag with **ga_set_counter** and then poll waiting for it to be set. This can be useful during initialization.

max is used to limit the counting. When the count reaches *max* it is reset back to zero.

Return Value: SUCCESS or FAILURE

gi_drvr_close gi_drvr_close

Name: **gi_drvr_close** - close a Stream driver

Synopsis: **#include <gios.h>**

int gi_drvr_close(Driver ***driver)**

driver	pointer to driver structure

Description: **gi_drvr_close** calls the **close** function in the **DriverTable** structure contained in *driver*. The **close** function is implemented by the Stream driver code and should deactivate the input and output channels and free all the necessary system resources.

driver must be initialized with **gi_drvr_init** before being open or closed.

Return Value: Passes the return code of **close** to the caller.

See Also: **gi_drvr_init(), gi_drvr_ioctl(), gi_drvr_open()**

Examples:

gi_drvr_init

Name: **gi_drvr_init** - initialize a Stream driver

Synopsis: **#include <gios.h>**

int gi_drvr_init(Driver ***driver)**

driver	pointer to driver to initialize

Description: **gi_drvr_init** calls the **init** function in the **DriverTable** structure contained in the *driver* with a single argument containing *driver*. The **init** function is implemented by the Stream driver code and should perform any initialization needed for **gi_drvr_open**, **gi_drvr_close**, and **gi_drvr_ioctl** to function.

Return Value: Passes the return code of **init** to the caller.

See Also: **gi_drvr_close(), gi_drvr_ioctl(), gi_drvr_open()**

Examples:

gi_drvr_ioctl gi_drvr_ioctl

Name: **gi_drvr_ioctl** - configure a Stream driver

Synopsis: **#include <gios.h>**

int gi_drvr_ioctl(Driver ***_driver_,**...**)**

driver	pointer to driver to configure
...	arguments to configuration function

Description: **gi_drvr_ioctl** calls the **ioctl** function in the **DriverTable** structure contained in the *driver.* The **ioctl** function is implemented by the Stream driver code and all additional arguments are passed on to this function. **ioctl** can be used to do custom configuration or to change default driver parameters**.**

driver must be initialized with **gi_drvr_init** before being configured.

Return Value: Passes the return code of **ioctl** to the caller.

See Also: **gi_drvr_close(), gi_drvr_init(), gi_drvr_open()**

Examples:

gi_drvr_open gi_drvr_open

Name: **gi_drvr_open** - open a Stream driver

Synopsis: **#include <gios.h>**

int gi_drvr_open(Driver ***driver*)**

driver	pointer to driver to open

Description: **gi_drvr_open** calls the **open** function in the **DriverTable** structure contained in *driver*. The **open** function is implemented by the Stream driver code and should activate the input and output channels and allocate all the necessary system resources to allow a Stream to use the driver

driver must be initialized with **gi_drvr_init** before being open or closed.

Return Value: Passes the return code of **open** to the caller.

See Also: **gi_drvr_close(), gi_drvr_init(), gi_drvr_ioctl()**

Examples:

gi_gcb_checksum gi_gcb_checksum

Name: **gi_gcb_checksum** - validate the EEPROM GCB Registers.

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_gcb_checksum(int *mode***)**

mode	- check or update mode

Description: **gi_gcb_checksum** computes the checksum of the GCB Registers in EEPROM. If mode is **GCB_CHECK**, the new checksum is compared to the value stored in the byte following the last GCB Register and **FAILURE** is returned if they are not equal. If *mode* is **GCB_UPDATE**, the checksum is written in the byte following the last GCB Register.

Return Value: **SUCCESS** or **FAILURE** if checksums are not equal.

See Also: **gi_gcb_eetor(), gi_gcb_etoee(), gi_gcb_loopback()**

Examples:

gi_gcb_eetor gi_gcb_eetor

Name: **gi_gcb_eetor** - copy GCB Registers from EEPROM to RAM.

Synopsis: **#include <gios.h>**

void gi_gcb_eetor(void)

Description: **gi_gcb_eetor** copies all the GCB Registers from EEPROM to RAM.

Return Value: none

See Also: **gi_gcb_checksum(), gi_gcb_etoee(), gi_gcb_loopback()**

Examples:

gi_gcb_etoee gi_gcb_etoee

Name: **gi_gcb_etoee** - copy GCB Registers from EPROM to EEPROM.

Synopsis: **#include <gios.h>**

void gi_gcb_etoee(void)

Description: **gi_gcb_etoee** copies all the default GCB Registers from EPROM to EEPROM. A checksum of the all the Registers is written immediately following the last Register. Only values that are different are actually written.

Return Value: none

See Also: **gi_gcb_checksum(), gi_gcb_eetor(), gi_gcb_loopback()**

Examples:

gi_gcb_loopback

gi_gcb_loopback

Name: **gi_gcb_validate** - preform loop-back test on port D.

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_gcb_loopback(void)

Description: **gi_gcb_loopback** performs a loopback test between pins PD0 and PD1 on JP2. If the pins are shorted together, it returns **FAILURE**.

Return Value: **SUCCESS** or **FAILURE** if pins PD0 and PD1 are shorted.

See Also: **gi_gcb_checksum(), gi_gcb_eetor(), gi_gcb_etoee()**

Examples:

gi_eeprom_erase gi_eeprom_erase

Name: **gi_eeprom_erase** - erase a byte in EEPROM.

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_eeprom_erase(void ****address*)**

address	address of byte to erase

Description: **gi_eeprom_erase** erases the EEPROM byte at *address*. A check is made to ensure that *address* is in EEPROM.

Return Value: **SUCCESS** or **FAILURE** if *address* is not in EEPROM

See Also: **gi_eeprom_write(), gi_eeprom_erase_bulk(), gi_eeprom_erase_row()**

Examples:

gi_eeprom_erase_bulk gi_eeprom_erase_bulk

Name: **gi_eeprom_erase_bulk** - erase all of EEPROM.

Synopsis: **#include <gios.h>**

int gi_eeprom_erase_bulk(void)

Description: **gi_eeprom_erase_bulk** erases the all EEPROM locations.

Return Value: none

See Also: **gi_eeprom_erase(), gi_eeprom_erase_row(), gi_eeprom_write()**

Examples:

gi_eeprom_erase_row gi_eeprom_erase_row

Name: **gi_eeprom_erase_row** - erase a16 byte row of EEPROM.

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_eeprom_erase_row(void ****address*)**

address	address of byte in row to erase

Description: **gi_eeprom_erase_row** erases a row EEPROM bytes at *address*. A check is made to ensure that *address* is in EEPROM. A row is 16 bytes starting on an even 16 -byte boundary (7E00, 7E10, 7E20, etc...)

Return Value: **SUCCESS** or **FAILURE** if *address* is not in EEPROM

See Also: **gi_eeprom_write(), gi_eeprom_erase(), gi_eeprom_erase_bulk()**

Examples:

gi_eeprom_write

Name: **gi_eeprom_write** - write a byte in EEPROM.

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_eeprom_write(void ****address*, **int *value***)**

address	address of byte to write
value	value of byte to write

Description: **gi_eeprom_write** writes the EEPROM byte at *address* to *value*. A check is made to ensure that *address* is in EEPROM. The byte should be erased first.

Return Value: **SUCCESS** or **FAILURE** if *address* is not in EEPROM

See Also: **gi_eeprom_erase(), gi_eeprom_erase_bulk(), gi_eeprom_erase_row()**

Examples:

gi_intr_disable gi_intr_disable

Name:	**gi_intr_disable** - disable MC68HC11 interrupts
Synopsis:	**#include <gios.h>**
	void gi_intr_disable(void)
Description:	**gi_intr_disable** executes a SEI instruction**.**
Return Value:	none
See Also:	**gi_intr_dummy(), gi_intr_enable(), gi_intr_rti(), gi_intr_swi()**
Examples:	

gi_intr_dummy gi_intr_dummy

Name: **gi_intr_dummy** -dummy interrupt service routine

Synopsis: **#include <gios.h>**

void gi_intr_dummy(void)

Description: **gi_intr_dummy** consists of a single RTI instruction. This routine can be used to fill in unused interrupt vectors.

Note that is this routine does nothing to clear the interrupt. If it does not clear on its own, the interrupt will continue to occur and execution will never return to user code.

Return Value: none

See Also: **gi_intr_disable(), gi_intr_enable(), gi_intr_rti(), gi_intr_swi()**

Examples:

gi_intr_enable gi_intr_enable

Name: **gi_intr_enable** - enable MC68HC11 interrupts

Synopsis: **#include <gios.h>**

void gi_intr_enable(void)

Description: **gi_intr_enable** executes a CLI instruction**.**

Return Value: none

See Also: **gi_intr_disable(), gi_intr_dummy(), gi_intr_rti(), gi_intr_swi()**

Examples:

gi_intr_rti

Name: **gi_intr_rti** - execute a RTI instruction

Synopsis: **#include <gios.h>**

void gi_intr_rti(void)

Description: **gi_intr_rti** pops its return value off the stack and executes a single RTI instruction. This routine can be used to return from a C interrupt service routine. Note that this routine works only if the SP register points to the interrupt stack frame (i.e.the SP is the same value as it was when entering the ISR). Without special care, this will not always be true. Some compilers always mess with the stack upon entering a C function (e.g. GCC 2.3.3). Also, declaring automatic variables in the ISR will usually change the SP.

Return Value: none

See Also: **gi_intr_dummy(), gi_intr_disable(), gi_intr_enable(), gi_intr_swi()**

Examples:

gi_intr_swi gi_intr_swi

Name: **gi_intr_swi** - execute a SWI instruction

Synopsis: **#include <gios.h>**

void gi_intr_swi(void)

Description: **gi_intr_swi** pops its return value off the stack and executes a single SWI instruction. This routine can be used to return GBUG from a C routine.

Return Value: none

See Also: **gi_intr_dummy(), gi_intr_disable(), gi_intr_enable(), gi_intr_rti()**

Examples:

gi_sci_free gi_sci_free

Name: **gi_sci_free** - free the SCI.

Synopsis: **#include <coactive.h>**

#include <gios.h>

void gi_sci_free(int **sci_driver*, **Stream** ***stream***)**

sci_driver	value of **SCIDRIVER** GROM RAM variable before call
stream	GBUG Stream pointer before call

Description: **gi_sci_free** frees the SCI by closing the SCI or GNET Stream driver and detaching any Stream which is connected to it. This is useful when running a program from GBUG which needs to use the SCI without using the Stream driver which GBUG is using (e.g. starting up the network after using GBUG though the SCI Stream driver). *sci_driver* and *stream* are used to return the values of the **SCIDRIVER** GROM RAM variable and the GBUG Stream which were in use before calling **gi_sci_free**. **gi_sci_reinit** can use these to restore the state of the SCI to what it was before **gi_sci_free** was called.

Return Value: none.

See Also: **gi_sci_reinit()**

Examples:

gi_sci_reinit gi_sci_reinit

Name:	**gi_sci_reinit** - reinitialize the SCI Stream driver	
Synopsis:	**#include <coactive.h>**	
	#include <gios.h>	
	void gi_sci_reinit(int *sci_driver*, **Stream** **stream*)	
	sci_driver	`SCIDRIVER` GROM RAM variable retuned by **gi_sci_free**
	stream	Stream returned by **gi_sci_free**

Description: **gi_sci_reinit** reinitializes the SCI Stream driver if it was previously freed with **gi_sci_free**. *sci_drvier* and *stream* should contain the values returned by **gi_sci_free**.

Return Value: none.

See Also: **gi_sci_freet()**

Examples:

gi_strm_attach gi_strm_attach

Name:	**gi_strm_attach** - assign a Stream driver to a Stream
Synopsis:	**#include <coactive.h>**
	#include <gios.h>
	int gi_strm_attach(Stream ***stream*****, Driver** ***driver*****)**
	stream — Stream to attach
	driver — Driver to attach
Description:	**gi_strm_attach** assigns Stream driver *driver* to *stream*. Detaches any existing driver first. Any data in the Stream output buffer is sent to *driver.*
	stream must be initialized using **gi_strm_init** before it can be attached and detached.
Return Value:	**SUCCESS** or **FAILURE** if the stream is not initialized.
See Also:	**gi_strm_detach(), gi_strm_init()**
Examples:	

gi_strm_detach gi_strm_detach

Name: **gi_strm_detach** - detaches a Stream driver from a Stream

Synopsis: **#include <coactive.h>**

#include <gios.h>

int gi_strm_detach(Stream ****stream*)**

stream	Stream to detach

Description: **gi_strm_detach** detaches Stream *stream* from its driver. The Stream's input and output buffers are preserved and reads and writes on the Stream can still be performed.

stream must be initialized using **gi_strm_init** before it can be attached and detached.

Return Value: **SUCCESS** or **FAILURE** if the stream is not initialized.

See Also: **gi_strm_attach(), gi_strm_init()**

Examples:

gi_strm_flush gi_strm_flush

Name: **gi_strm_flush** - wait for a Stream output buffer to empty

Synopsis: **#include <gios.h>**

void gi_strm_flush(Stream ****stream*)**

stream	Stream to flush

Description: **gi_strm_flush** waits for the output circular buffer of *stream* to empty.

Return Value: none

See Also: **gi_strm_attach(), gi_strm_init()**

Examples:

gi_strm_init

Name: **gi_strm_init** - initializes a Stream

Synopsis: **#include <coactive.h>**

#include <gios.h>

void gi_strm_init(Stream ***stream*****, char** ***in_buf*****, int** *in_size***,**
int *in_min***, int** *in_max***, char** ***out_buf*****,**
int *out_size***, int** *flow***)**

stream	Stream to initialize
in_buf	allocated input buffer
in_size	size of input buffer
in_min	software flow control XON trigger
in_max	software flow control XOFF trigger
out_buf	allocated output buffer
out_size	size of output buffer
flow	flow control enable flag

Description: **gi_strm_init** initializes a Stream. The input and output buffers must be allocated before making this call. *in_min* and *in_max* specify the number of characters in the input buffer which will trigger software flow control XON and XOFF. *flow* indicates if software flow control in enabled. Possible values for flow are:

YES	- enable flow control
NO	- disable flow control

gi_strm_init must be called before *stream* can be attached and detached.

Return Value: none

See Also: **gi_strm_attach(), gi_strm_detach()**

Examples:

gi_strm_input gi_strm_input

Name: **gi_strm_input** - read a character from a Stream

Synopsis: **#include <gios.h>**

int gi_strm_input(Stream ****stream*)**

stream	Stream for input

Description: **gi_strm_input** returns the first character in *stream*'s input buffer. If there are no characters in the input buffer, **gi_strm_input** will loop until one is available.

Return Value: character read

See Also: **gi_strm_output()**

Examples:

gi_strm_output

Name: **gi_strm_output** -write a character to a Stream

Synopsis: **#include <gios.h>**

void gi_strm_output(Stream ****stream*, unsigned char** *c***)**

stream	Stream for output
c	character to write

Description: **gi_strm_output** writes *c* to the end of *stream*'s output buffer. If there is no space available, **gi_strm_output** will loop until space is available.

Return Value: none

See Also: **gi_strm_input()**

Examples:

gi_strm_pull gi_strm_pull

Name: **gi_strm_pull** - pull a character from a Stream's output buffer

Synopsis: **#include <gios.h>**

unsigned int gi_strm_pull(Driver ***driver*****)**

driver	pointer to Stream driver

Description: **gi_strm_pull** returns the first character in the Stream's output buffer which is attached to *driver*. The unsigned int value returned contains the character to send in the least significant byte and the number of characters in the buffer before the function is called in the most significant byte.

This function is designed for use in a Stream driver. Since global data structures are manipulated, it is not reentrant and care should be taken if this function is used while interrupts are enabled.

Return Value: MSB - number of characters in buffer before the call, 0 if *driver* in not attached

LSB - first character in output buffer

See Also: **gi_strm_push()**

Examples:

gi_strm_push gi_strm_push

Name: **gi_strm_push** - push a character onto a Stream's input buffer

Synopsis: **#include <coactive.h>**

#include <gios.h>

unsigned int gi_strm_push(Driver ****driver*, unsigned char** *c***)**

driver	pointer to Stream driver
c	character to push

Description: **gi_strm_push** places *c* at the end of the Stream's input buffer which is attached to *driver*. If the buffer is full, **FAILURE** is returned.

This function is designed for use in a Stream driver. Since global data structures are manipulated, it is not reentrant and care should be taken if this function is used while interrupts are enabled.

Return Value: **SUCCESS** or **FAILURE** if buffer is full

See Also: **gi_strm_pull**

Examples:

gi_strm_size gi_strm_size

Name:	**gi_strm_size** - return the number of characters in Stream's input or output buffer
Synopsis:	**#include <gios.h>**
	unsigned int gi_strm_size(Stream ****driver*, char** *mode***)**
	driver driver to size
	mode size mode
Description:	**gi_strm_size** returns the number of characters in the input or output circular buffer of *stream*. Mode indicates whether the input or output buffer is being checked. The possible values for mode are:
	STRM_INPUT - input buffer
	STRM_OUTPUT - output buffer
Return Value:	size of buffer
See Also:	
Examples:	

gi_strm_stdio gi_strm_stdio

Name: **gi_strm_stdio** - assign a stream to the STDIO GROM variable

Synopsis: **#include <gios.h>**

void gi_strm_stdio(Stream ****stream*)**

stream	stream to assign

Description: **gi_strm_stdio** assigns *stream* to the GROM variable `STDIO`. `STDIO` is used by all Standard C Library I/O calls as the default GIOS Stream. This call must be made before using any of these functions.

Return Value: none

See Also:

Examples:

gn_close_port_l0 gn_close_port_l0

Name: **gn_close_port_l0** - close level 0 network connection to application.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

void gn_close_port_l0()

Description: This function disconnects an application from the network services and releases it for other applications to connect to. This program must be called before **gn_exit_comms**. After this routine is called the network continues running in the background but will not send or receive messages. This is necessary since if the node is the master on the network it must continue to service messages between the slaves on the network.

Return Value: SUCCESS

Examples:

gn_close_port_l1 gn_close_port_l1

Name: **gn_close_port_l1** - close level 1 network connection to an application.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_close_port_l1()

Description: Should be called before exiting an application which uses level 1 network resources. As with **gn_close_port_l0** the connection to the application will be closed, but the network will continue to run in the background.

Return Value: SUCCESS.

Examples:

gn_config_node gn_config_node

Name: **gn_config_node** - configure network node.

gn_get_config - get the current node configuration.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_config_node (int *master*, **int *address*, int *baud*, int *enable*)**

int gn_get_config (CONFIG_REC ****config*)**

master	set to 0 if this node is a slave or non-zero if it is a master.
address	address of the node.
baud	baud rate to use for the communications.
enable	on the GCB11 this is used to signify which of two possible enable line are used to enable the line drivers. It is set to 0 if PG2 is used for the enable line, else set to non-zero if PD2 is used as the enable line. On the PC it is used to signify which COM port is used for the network communications. Either COM1 or COM2 may be specified.
config	pointer to structure with configuration parameters as defined below:

```
typedef struct  config_rec
{
  char        open;            /* flag set to non-zero if port open */
  char        address;         /* node address */
  char        baud;            /* baud rate */
  char        master;          /* flag to determine if master node */
  char        enable;          /* the enable line used on the GCB11 */
  int        (*recv_start)();  /* function to call at start of message */
  int        (*recv_end)();    /* function to call at end of message */
  int        (*xmit_end)();    /* function to call at end of transmitting */
}            CONFIG_REC;
```

Description: Configures and get the various parameters for the network. When setting the parameters they remain in effect until the next call to **gn_config_node** or the GCB11 is reset. the function **gn_config node** should be called before **gn_init_comms** if the default values stored in the EEPROM are not to be used. The call to gn_config_node changes only those values stored in the GCB corresponding GCB registers. It does not immediately take effect if the network is already running.

gn_get_config returns the current parameters being used by the network. Note that this may be different from the values stored in the GCB registers from a previous call to **gn_config_node**. This is because a call to **gn_config_node** does not take effect until the network is reset.

These functions can be called when using either level 1 or level 0 network communications.

Return Value: SUCCESS

Examples:

gn_get_message_l1 gn_get_message_l1

Name: **gn_get_message_l1** - Return a level 1 message from the message queue.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int get_message_l1(MESS_REC * ***mess*****)**

mess	pointer to structure containing all the information concerning a message. See gnet.h for a definition of this structure.

Description: This function returns the next message from the level 1 message queue. This function should only be called if level 1 network functionality has been enabled.

Return Value: SUCCESS if message returned, else FAILURE of no messages available.

Examples:

gn_get_stats gn_get_stats

Name: **gn_get_stats** - get network statistics.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_get_stats(STATS_REC *_stats_),

stats	pointer to structure for the network statistics with the following parameters:

```
typedef struct  stats_rec
{
  unsigned short        timeouts;       /* number of character timeouts */
  unsigned short        retries;         /*numberoftransmissionerrors*/
}               STATS_REC;
```

Description: Returns the network error statistics as defined above.

Return Value: SUCCESS

Examples:

gn_init_comms gn_init_comms

Name: **gn_init_comms** - Network initialization.

gn_exit_comms - Disable network.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_init_comms()

int gn_exit_comms()

Description: **gn_init_comms** is used at start-up to initialize the communications drivers. This routine will set up the network to whatever was stored in EEPROM unless a call to gn_config_node is made before this routine. Must be called before any of the level 0 or level 1 network calls.

gn_exit_comms is the counterpart to **gn_init_comms** in that it is called in order to disable the communications drivers after they have been enabled.

Return Value: Returns SUCCESS or FAILURE.

Examples:

gn_message_rdy_l1 gn_message_rdy_l1

Name: **gn_message_rdy_l1** - check for message ready in level 1 queue.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int message_rdy_l1(unsigned int ****length*)**

length	Number of bytes in the next message to be obtained from the message queue.

Description: Returns the number of messages in the level 1 receive queue. This function is mainly used to poll the receive message queue.

Return Value: Number of messages in the level 1 receive queue.

Examples:

gn_open_port_l0 gn_open_port_l0

Name: **gn_open_port_l0** - Initialize level 0 network communications.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_open_port_l0 (int (**recv_start***)(MESS_REC ***message**),**

int (**recv_end***)(int flag, MESS_REC *message),**

int (**xmit_end***)(int flag, char *buffer))**

recv_start	Call back routine supplied by the application. This routine is called when an incoming message requires that the application provide space to store the message. The message size is stored in message->length and the buffer supplied by the application should be stored in message->buffer. The call back must return SUCCESS or FAILURE depending upon wether the buffer was successfully allocated.
recv_end	Call back routine supplied by the application. This routine is called when an incoming message has arrived and is ready to be processed by the application. The *flag* variable is used to signify if the message was received successfully. The message and all details about it are stored in the structure *message*.
xmit_end	Call back routine supplied by the application. This routine is called at the completion of a message being sent when the application sends a message, but does not block to wait for it. In this way the application will know wether the message in question was sent successfully and that it may use the buffer allocated for the message. The *flag* variable is used to signify wether the message was sent successfully and the *buffer* is the where the transmitted message was stored.

Description: This function is used to ready the network drivers for level 0 message handling. It installs a series of call back routines which are supplied by the user. This routine must be called after **gn_init_comms**, but before any messages are to be processed.

Return Value: SUCCESS or FAILURE.

Examples:

gn_open_port_l1 gn_open_port_l1

Name: **gn_open_port_l1** - open connection to level 1 network services.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_open_port_l1(void *(**user_malloc***)(size_t** *size***), void *(***user_free***)(), int** *queue_size***)**

user_malloc	application supplied memory allocation routine used by the level 1 drivers when necessary to malloc memory for received messages. In the most trivial case the user can simply supply the standard C malloc routine, but if desired the user can write his own in order to have more control over the memory management.
user_free	application supplied routine to free memory allocated by *user_malloc.*
queue_size	size of the receive queue. An array of queue_size by MESSAGE_REC is **`malloc`**ed for use as the receive queue. This parameter specifies how many messages may be queued up at any one point.

Description: This function establishes a level 1 network connection to the application. This routine can not be called in conjunction with any level 0 network routines, but must be called before any other level 1 access routines.

Return Value: SUCCESS or FAILURE.

Examples:

gn_reset_net gn_reset_net

Name: **gn_reset_net** - reset network.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

void name(Param ****param*)**,

param	pointer to driver to close

Description: Resets and initializes the network. All transmit and receive message queues are cleared. This function can be useful if **gn_config_node** has been called since the network was initialized and you want the network drivers to assume the new configuration parameters.

Return Value: SUCCESS

Examples:

gn_restore_config gn_restore_config

Name: **gn_restore_config** - restores default configuration for the network.

gn_save_config - saves default network communications configuration.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_restore_config(),

int gn_save_config()

Description: These functions restore and save the following GCB registers to and from EEPROM:

- SCIENABLE
- SCIBAUD
- GNETNODE
- GNETMODE
- GNETTABLE
- GNETCOUNT
- GNETPOLL

These functions can be called when using either level 1 or level 0 net work communications.

Return Value: SUCCESS

Examples:

gn_send_message_l0 gn_send_message_l0

Name: **gn_send_message_l0** - send a level 0 message.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_send_message_l0 (int node, char ****buffer*, unsigned int** *length***, int** *block***)**

node	node address of the recipient of message.
buffer	pointer to buffer containing data to be sent.
length	number of bytes in buffer to send.
block	flag to signify wether to wait for message to be sent.

Description: This routine is used to send information to a node in the network. The flag *block* should be set to non-zero if the routine is to wait for the message to be sent and return with the appropriate completion status. If block is zero then send_message_l0 returns immediately and the call back routine that was installed by **gn_open_port_l0** will be called when the message is finished being sent.

This routine is for level 0 message passing only and can not be called in conjunction with level 1 network routines.

Return Value: SUCCESS or FAILURE.

Examples:

gn_send_message_l1 gn_send_message_l1

Name: **gn_send_message_l1** - send level 1 network message.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_send_message_l1(int *node*, **char ****buffer*, **unsigned int** *length*, **int** *block***)**

node	node address of the recipient of the message.
buffer	pointer to buffer containing the data to send.
length	number of bytes in the buffer to send.
block	flag to signify wether to wait for message to be sent.

Description: This function send a level 1 message to the node specified by *node.* If block is non-zero this routine will wait until the message is transmitted and return either SUCCESS or FAILURE depending upon whether the message was sent successfully. If block is zero then this routine returns SUCCESS immediately after queueing the message to be sent.

This function can not be called in conjunction with any level 0 network routines.

Return Value: SUCCESS or FAILURE.

Examples:

gn_set_poll gn_set_poll

Name:

gn_set_poll - set up the slave node polling table.

gn_get_poll - get the current polling table.

gn_add_poll - add node to the polling table.

gn_auto_poll - enable automatic node addition to polling table.

Synopsis:

#include <gios.h>

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_set_poll (unsigned char ****table*, int** ***table_size*****)**

int gn_get_poll (unsigned char *****table*, int ****table_size*****)**

int gn_add_poll(unsigned char ***node_address*****)**

int gn_auto_poll(int ***loop_cnt*****)**

table	pointer to polling table.
table_size	number of nodes in the polling table.
node_address	address of the node to add to the polling table.
loop_cnt	Number of times master should go through the polling table before testing for a node not currently in the polling table.

Description:

These routines are used to manage the slave polling table. They are only valid for the master node.

gn_set_poll copies the table pointed to by *table* into the polling table.

gn_get_poll returns a pointer to the polling table used to poll the slaves. *table_size* contains the number of nodes in the table. Note that application should not alter the contents of this table directly. In order to change the contents the table should first be copied to some other areas, modified and then used as an argument to **gn_set_poll**.

gn_add_poll adds the node address specified in *node_address* to the end of the polling table if it does not already exist. Returns FAILURE if the node was not added to the table.

gn_auto_poll enables the automatic searching for nodes which are not explicitly put into the polling table by the application. This allows nodes to be added to the polling table automatically when they are plugged into the network. The major drawback of using this feature is that it will impact network performance because in cases where the master polls for nodes that do not exist there will be an inordinate delay while the master times out waiting for a response. A method for reducing this impact is given by the parameter *loop_cnt* which specifies the number of times the master will go through the polling table before trying to poll for a node not already in the table. This parameter may take on any value between 0 - 255. A value of 0 disables automatic polling.

Return Value: Returns SUCCESS if master else returns FAILURE.

Examples:

gn_suspend_poll gn_suspend_poll

Name: **gn_suspend_poll** - suspend polling of slave nodes.

gn_start_poll - start polling of slave nodes.

Synopsis: **#include <gios.h>**

#include <gcb11.h>

#include <coactive.h>

#include <gnet.h>

int gn_suspend_poll()

int gn_start_poll()

Description: Suspends and starts the polling of slave nodes by the master node. This function has no effect if the node is not the master. This will essentially suspend all network activity. This is a good way to start and stop the network without having to reset node in the network.

These functions can be when using either level 1 or level 0 network communications.

Return Value: SUCCESS if master node else FAILURE.

Examples:

CHAPTER 10

Support

10.1 Introduction

At Coactive Aesthetics we realize that it takes much more than a good piece of hardware to develop an embedded application. Because of the nature of embedded systems, there are far more off-the-shelf hardware components than there are software packages. We therefore have attempted to provide as many tools as possible to aid in the overall development of your embedded system, *especially* in the area of software. Everything we develop, from the GCB11 hardware to the motor control applications, is developed initially for in-house use. We have developed a wide variety of embedded applications, ranging from simple instrument control to complicated robotic devices. We therefore try to create re-usable components for fast prototyping and development. This also means that we have the expertise *and* experience to help you integrate the GCB11 into your next application. Since we are our own biggest customer, we are constantly making enhancements and upgrades to the software as our application base grows.

Using or adapting third party software for embedded applications is not always trivial. Sometimes the documentation is not as clear to the user as it was to the writer, while other times there may even be a bug. Whatever the problem, we would like to try to make ourselves as available to your needs as possible. Below you will find details on how to obtain support. We are always open to suggestions for ways to make our support to you better.

Please complete and return the registration card included with this documentation as soon as possible so that we can service you better.

10.2 Email

The most preferred method of support is through electronic mail on the Internet. Simply mail your questions to **gcb11@coactive.com**. You can expect a fast reply through this medium. If you are having problems with some piece of code, please include a sample of the code.

10.3 FAX

After electronic mail, the best medium is through the FAX. Our FAX number is **(415) 626-6320**. Please include a way for us to respond to your question, either through voice or FAX. Again, please include as much supporting documentation as possible, e.g. pictures, sample code, etc.

10.4 Voice

For those occasions when you must have an answer immediately, or you feel more comfortable talking to a human voice, you can call us at **(415) 626-5152**.

10.5 Software Upgrades

It is our intention to make minor software enhancements and bug fixes available to our customers on a regular basis. These will be free if they can be delivered electronically via email or ftp. For those customers that require shipping, there will be a nominal shipping and handling charge.To receive GROM updates, check the appropriate boxes on the registration card and mail it to Coactive Aesthetics. Please detail in the supplied space how you would like to receive the upgrade. When the upgrade is available, we will either send it to you electronically or mail you a letter with details about the nature of the upgrade and how much it will cost to have it shipped to you.