

**MIL-STD-1553 Remote Terminal Design Example
Using HI-6110, ARM7 & ANSI C Application Note**

INTRODUCTION

The HI-6110 is a versatile message processor for MIL-STD-1553 applications. It operates from a 3.3VDC power supply. The device includes an integral dual bus transceiver, and provides independent decoders for each bus, and a 1553 encoder for transmission. The HI-6110 can be configured as a Bus Controller, Remote Terminal or Bus Monitor, with or without assigned RT address.

This document presents an example design for a 1553 Remote Terminal based on the HI-6110 and an ARM7 16-bit microcontroller programmed in ANSI C language. The design was tested to and complies with "RT Validation Test Plan" requirements. Evaluation board has these features:

- Two 1553 bus interfaces, transformer- or direct-coupled.
- Signal headers for connecting a logic analyzer
- Status LEDs, buttons for board reset and test sequencing
- DIP switch for selecting RT address and parity
- Serial port for an optional computer interface
- Microcontroller JTAG header for ARM7 software debug
- Self-contained 3.3V power supply

Figure 1 is a block diagram of the evaluation board. Some features (like LEDs and pushbuttons) apply to the demonstration circuit because they support development, but are not typically found in a Remote Terminal application.

REMOTE TERMINAL OPERATION

For RT mode, a DIP switch sets the 5-bit Remote Terminal address and parity. At reset, the HI-6110 checks its address inputs for valid parity. If parity is wrong, an error is signaled to the host controller, and the device will not function until address and parity are corrected and reset is applied again.

Each bus has a dedicated 1553 decoder. Overlapping valid commands on both buses are decoded properly. All 1553 words received are checked for "word validity" – proper sync, 16 data bits plus correct parity, and proper Manchester II encoding.

"Command validity" is also checked. Commands addressed to other remote terminals are ignored. When a received command word address matches the preset HI-6110 RT address (or indicates broadcast command) the command is validated. The HI-6110 asserts a RCVA or RCVB output to signal which bus received the valid command. When RCV is asserted, the processor reads the HI-6110 Message Register and valid Command Word to determine appropriate action.

When a valid non-broadcast command is received, the HI-6110 automatically begins Status Word response after the command is received in full. The device Status Word Register is combined with the preset RT address inputs to fully specify the status word response. If the Status Word Register contains 0000 hex, a "clear status" (CS) response is transmitted to the MIL-STD-1553 Bus Controller.

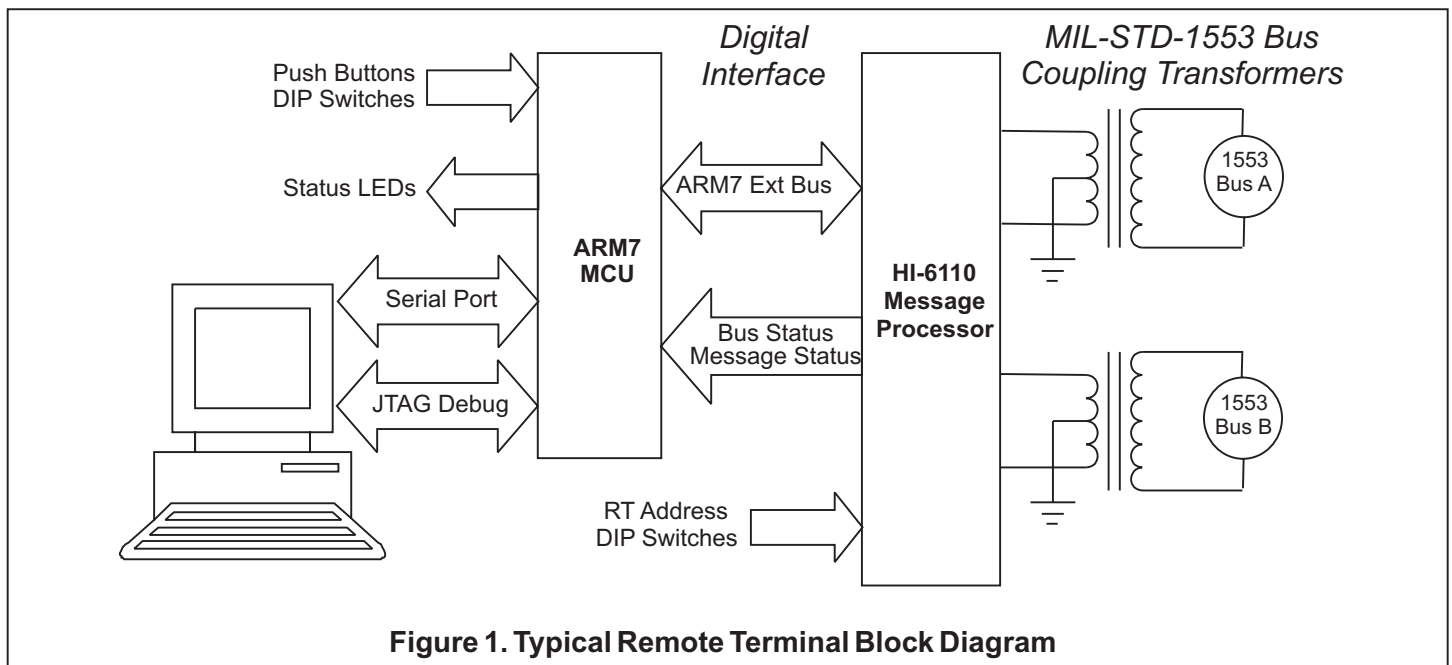


Figure 1. Typical Remote Terminal Block Diagram

For valid transmit commands, status word transmission begins automatically, about 6 μ S after the Command Word is completed. During the 20 μ S Status Word transmit interval, the host begins loading the Transmit Data FIFO or Transmit Mode Data Register.

Status word transmission is automatically handled for valid, non-broadcast receive commands. The HI-6110 stores received Data Words in its 32-word receive data FIFO, and resets its FFEMPTY (receive FIFO empty) status flag when the first data word is received. When low, the FFEMPTY flag indicates at least one unread data word is available in the FIFO. Successive FIFO fetches will read data words in the order received. When the last word is read, FFEMPTY goes high and stays high.

If a valid receive command is followed by data word error, the HI-6110 asserts its ERROR output. The host then reads the device Error Register to determine the error type. When required, the host processor can write to the HI-6110 Status Word Register to set the "message error" (ME) bit. Changes to the Status Word Register must be completed before mid-parity of status word transmission.

By handling low-level 1553 protocol requirements, the HI-6110 reduces real-time demands for the embedded Remote Terminal processor. The processor can then focus on higher level tasks.

HARDWARE CONSIDERATIONS

This example uses an Atmel ARM7 microprocessor with external bus interface ("EBI") that interfaces to the HI-6110 device. All HI-6110 register operations are memory-mapped.

Two Atmel ARM7 processors are directly compatible with the example program: AT91R40008 and AT91FR40162S. The AT91R40008 uses external flash memory for program storage using the basic system architecture of the Atmel EB40A evaluation board ARM7 evaluation board. Atmel provides a range of sample programs that can expand the utility of this design by using ARM7 processor peripherals not used here.

The AT91R40008 processor is available in a 100-pin TQFP package. The flash memory used has a 48-pin TSOP package. The AT91FR40162S combines a T91R40008 processor die and 2 MByte flash memory die in a single 121-pin BGA package.

The ARM7 processor has 32 bits of PIO ("parallel I/O"), and most PIO pins are multiplexed with a range of on-chip peripherals. The built-in peripherals are not used in this design. For flexibility in reassigning processor I/O, the ARM7 processor connects to external circuitry through solder jumpers on the bottom of the circuit board. With software changes, these jumpers permit processor I/O reassignment to free up integral processor peripherals for application use.

The schematic diagram is provided in Appendix C. Page 1 shows a serial port that connects to a processor USART if jumpers JP9 and JP10 are soldered closed. Serial port routines are not implemented in the example, but are easily added by the user from Atmel programming examples for their EB40A board. Several LEDs and pushbuttons are available to provide diagnostic feedback and manual input during software development. Board features for development and debug are provided that would probably be eliminated once an application is finalized. This includes the various signal headers and logic analyzer headers. The logic analyzer connection includes a circuit for bus signal "scaling" (clipping) so signal timing relative to 1553 words on the bus can be observed. With 10K ohm input impedance, the clipper circuit should not affect bus performance, but input solder jumpers JP29 and JP30 should be opened for electrical characterization tests during validation.

The ARM7 processor can run in high performance "ARM mode" using 32-bit instructions, or can use lower performance "Thumb mode" using a ROM-efficient 16-bit instruction set. Execution in ARM mode is faster for some tasks but requires two 16-bit fetches per instruction, compared to one fetch per instruction for Thumb mode. After reset, the ARM7 processor initializes in ARM mode. Afterward, source program compiler directives tell the processor when to switch between ARM and Thumb modes. Low level initialization and interrupt entry and exit handlers always use ARM mode, while the rest of the example program uses Thumb mode.

The ARM7 processor is clocked at 64 MHz. This processor is rated for faster clock speeds, but 64 MHz is the limit when using memory-mapped interface to the HI-6110. Each external bus chip select has programmable characteristics. Using the maximum number of 8 wait states, the HI-6110 is not compatible with higher processor clock frequency.

Processor clock frequency also affects basic memory configuration and power up sequencing. Using flash memory for nonvolatile program storage presents a dilemma. Contemporary flash memory is not fast enough for zero wait state instruction fetches when the processor is clocked at speeds beyond 30 MHz or so. The Atmel AT91 ARM7 family uses "romcopy" then "remapping" to overcome the flash speed limitation. At power-up, chip select for the program flash memory is initialized to use 6 or 7 wait states per instruction fetch, greatly reducing effective execution speed. After reset, execution begins from flash at reduced speed. The start-up sequence copies the entire executable program from flash memory into processor SRAM in a process called "romcopy". Once copy is completed, "remap" occurs to begin full speed (zero wait state) program execution from processor SRAM.

JTAG port J9 provides a standardized interface for connecting a debugger compatible with the chosen development tool set. Green Hills MULTI uses their own "Probe" or "Slingshot". Other compilers will use different JTAG interface cables. Flash programming is performed

through the JTAG debugger interface. Depending on the software tool set used, the Debugger may or may not be able to debug “romcopy” project builds described earlier. Even if the Debugger allows “romcopy” debugging, flash programming after each change is slow and inconvenient. For this reason, most development is conducted using “ramrun” project builds in which the Debugger loads software directly into processor SRAM for execution. Naturally the loaded “ramrun” program is volatile and lasts only as long as power is applied.

Linker directives specify whether the project is built for “romcopy” or “ramrun”. Using “ramrun” gives fastest debugging results because it is not necessary to program the flash to test each program modification. Once software is developed and verified with “ramrun,” the user compiles a “romcopy” build and programs the flash memory for nonvolatile “plug and play” operation.

The schematic diagram includes a reset circuit providing five separate means for processor reset: (a) power-up reset, (b) manual push button reset, (c) watchdog timer overflow reset, (d) reset initiated by an external JTAG debugger, and (e) processor-initiated hard reset performed when a mode code 8 (MC8) “reset remote terminal” mode code command is received.

Processor-initiated reset after MC8 can take no longer than 5mS to meet the requirements of MIL-STD-1553 Notice 2. The example program meets the 5 mS requirement while performing full romcopy with remapping after MC8 is received. Since it employs hardware reset, the processor-initiated MC8 reset cannot be evaluated using a debugger because resetting disconnects the debugger.

Faster MC8 reset may be achieved by causing execution to go to the remap address without executing ROM copy. This method might be compatible with JTAG debugger operation. Details are not provided for this modification, and depend on the compiler used for software development.

Two 40-pin logic analyzer signal headers provide connections for all HI-6110 signals. This includes isolation networks for direct connection of Agilent logic analyzer 40-pin pod connectors without using the “flying lead” cables. The Agilent E9340A PC-hosted logic analyzer is a suitable example. Other logic analyzer types can be connected using conventional “flying leads”.

Software for the serial port connected to the host microcontroller is not implemented. The microcontroller has an integral UART connected to the RS-232 level shifter and DB-9 connector. Program changes would permit use of this communication port.

The schematic diagram for the board is provided at the end of this application note. Assembled evaluation boards for the design are available from Holt Integrated Circuits.

SOFTWARE CONSIDERATIONS

The example program is written in ANSI C and was developed using Green Hills MULTI software development environment. Details of the software package, evaluation licenses etc can be found at the Green Hills internet web site, <http://www.ghs.com> . Other compilers may be used, but program files will require adjustments.

Typeface conventions used in this application note:

Hardware Signal Names and Component References

Normal-Arial font, all caps

Examples: VALMESS and ERROR are signal names
JP1 and LED2 are component references

Program File Names

Bold-Arial font

Examples: **interrupt_init.c** and **vectors.arm** are files

Program Function Names

Bold-Courier font, all lower case, ending with “()” ellipsis

Examples: **standby()** and **main()** are C functions

Program Variable Names

Bold-Courier font, all lower case

Example: **saved_sp** and **last_cw** are variables

Program Labels, Constants and Macro Names

Bold-Courier font, upper case

Examples: **IRQ_ENTRY** is a macro and **PI** is a constant

For register read/write operations, the HI-6110 needs just four EBI address lines, plus READ/WRITE signal and /STROBE. The ARM7 EBI has separate NRE read enable and NWE write enable; these signals are not directly compatible with the HI-6110. The example design logically-ORs the NRE and NWE signals to provide a combined strobe, and a fifth EBI address bit serves as the read/write signal, mapping HI-6110 reads and writes to separate address spaces.

Special precautions apply when writing the HI-6110 Control register. Clocking the processor at 64 MHz results in tight timing. When the EBI interface is set-up for writing the Control register, even transitory bus data can affect 6110 operation while /STROBE is low. For example, if control bits TXA or TXB accidentally go low even momentarily, the corresponding decoder for that bus will be reset. The example program prevents this by first writing the desired data to another register address (e.g. Clear Transmit FIFO) immediately before addressing and writing the Control register, as the bus data floats between the two operations. Similar precautions are needed if a Write-Control is followed immediately by a write operation to a different register.

Three signals from the HI-6110 provide IRQ interrupts for the ARM7 processor: RCVA, RCVB and ERROR. These three signals are assigned the highest interrupt priority level. To

prevent late responses, no other interrupt should be assigned the same priority level. The ARM7 processor FIQ interrupt is not used, so is available for application expansion.

The Atmel ARM7 processor is programmed for two additional PIO interrupt inputs, RCMDB and RCMDA from the HI-6110. A PIO interrupt input can originate from any combination of the 32 parallel I/O signals found in the Atmel ARM7 device. Unfortunately these PIO interrupts are inflexible. All inputs programmed as PIO interrupt sources generate the PIO interrupt for both rising and falling edges. The specific signal causing the interrupt must be determined by polling PIO inputs within the interrupt routine.

In our design, we are only concerned with rising edge interrupts, so falling edge interrupts are a nuisance. The nuisance falling edges for RCMDB or RCMDA are coincident with important IRQ interrupts for RCVA and RCVB respectively. Because we use just two PIO interrupts, the nuisance factor is manageable in software by adjusting priority of PIO v. IRQ interrupts. If application requirements call for additional PIO interrupt inputs, Appendix A describes a method for eliminating nuisance falling edge PIO interrupts altogether.

PROJECT SOFTWARE DESCRIPTION

Page 6 lists the files in this Remote Terminal project build. Hardware feature definitions for the circuit board are found in header file **Holt_6110.h**. This file defines processor signal assignments and symbols used throughout the program. The file also declares the C structure (struct) for the HI-6110 interface. The program relies heavily on Atmel's inline library functions for the ARM7 processor. Some degree of translation will be needed if porting the program to a different processor.

Function **main()** is called by the low level ARM code following reset. Its only purpose is hardware initialization. Loading ARM7 control registers configures many processor options. Hardware external to the processor is initialized, including the HI-6110 and its registers; the Control register is loaded for RT mode operation.

Once **main()** completes initialization, it calls the function **standby()**. This is considered a "root-level routine". Once called, execution never returns to **main()**. The top of **standby()** performs additional set-up the first time it is called by **main()**. It then performs repetitive housekeeping and system tasks while awaiting interrupts. RCVA or RCVB interrupts signal arrival of a new MIL-STD-1553 command. An ERROR interrupt will occur upon failure of an in-process command previously recognized by a RCV interrupt.

The ERROR interrupt also occurs when a valid command word is immediately followed by something unexpected. Here, "valid" is used in the MIL-STD-1553 context – a

command is "valid" if addressed to the remote terminal (or to RT31 if broadcast commands are allowed). Examples of unexpected outcomes include (a) a receive command without a contiguous data word, (b) a transmit command with contiguous data word, or (c) a receive command in which an encoding error or bus distortion occurs during a data word following the valid command word.

If an error is detected immediately after the valid command word, ERROR may be asserted without first asserting a RCV interrupt. If an error occurs later in the message, it will follow the RCV interrupt for the valid command word.

The HI-6110 generates a RCVA or RCVB interrupt when a valid command arrives. Interrupt handlers for RCVA and RCVB are purposely brief. No command processing is done in the interrupt handlers, only bus switching, if needed. Variable **command_pending** is updated to a non-zero value. Using a method explained below, execution returns to function **standby()**. Detecting a non-zero value for variable **command_pending**, function **standby()** calls the function **process_received_command()**. This is explained in further detail on the following page.

Every effort was made to minimize processor-dependence in this design. However, one aspect of MIL-STD-1553 causes problems with C language programming when the target uses interrupts: the MIL-STD-1553 Remote Terminal must always respond to the last valid command received. If a new command arrives while an earlier command is unfinished on the other bus, the earlier command response must be reset. This means special handling is required for management of the RCVA, RCVB and ERROR interrupts.

Problems occur when normal return-from-interrupt is used after a superseding command is processed. Timing is sufficiently tight that command-processing routines cannot be interrupted by repetitive, periodic tests to detect new superseding commands. Such testing would slow execution, causing late responses and other problems. Further, the program must not resume execution of an unfinished command when the superseding command is completed. This project demonstrates a method that effectively stops processing for prior commands when a new command or error is detected.

Conventional interrupt management resumes execution from the point at which the interrupt occurred. Normal interrupt management is defined by macros **IRQ_ENTRY** and **IRQ_EXIT** provided by Green Hills Software in the file, **irq_ghs.mac**. The assembly language macro **IRQ_ENTRY** saves return address and working registers to the IRQ stack; **IRQ_EXIT** restores working registers from the stack then loads the return address directly from stack to program counter to resume interrupted execution. For reasons given earlier, the conventional handling causes problems with our RCVA, RCVB or ERROR interrupts.

A new IRQ macro called `IRQ_ENTRY_SPECIAL` was added to file `irq_ghs.mac`. It is used only for ERROR, RCVA and RCVB interrupts from the HI-6110. The low level ARM handlers for these three interrupts were changed, substituting this new macro for the conventional macro, `IRQ_ENTRY`. Instead of storing the usual interrupt return address to the IRQ stack, `IRQ_ENTRY_SPECIAL` stores the start address for function `standby()` to the IRQ stack. When the interrupt routine is complete, the unmodified macro `IRQ_EXIT` always jumps to the starting address of function `standby()` where command processing is handled.

When returning to `standby()` after a RCV or ERROR interrupt, all working register contents are reinitialized as needed. No old working register data or status flags are used. To prevent user stack pointer "creep," the function `standby()` reinitializes the user SP to the value in effect the first time `standby()` was reached after power-up reset. This method is not entirely trouble-free during software development. See Appendix B for important details. The appendix also describes a improved method for user SP initialization that was developed after RT Validation tests were finished. Because the improvement was not part of the validation software, it is not included in the standard project build, but the new version can be evaluated after substituting one project file.

Aside effect of this method for RCV and ERROR interrupts is that housekeeping tasks performed in `standby()` may be interrupted and never resumed. This disruption applies to function calls occurring in `standby()` or to nested function calls there. Critical housekeeping tasks should employ a method of "state awareness" to recognize when disruptions occur. One method would use state tracking variable(s) that are given a value when a critical task begins, and a different value when the critical task is completed. By polling variable(s) at `standby()` reentry, the program can detect when a process was interrupted before completion. The unfinished task could then be repeated. No critical `standby()` tasks are used in the example program.

Upon `standby()` reentry, the program tests whether a new command was received or not. If yes, the function `process_new_command()` is called. This function decodes the received command by retrieving data from one of two tables in file `tables.c`. One table is used for broadcast commands; the other table is used for non-broadcast commands. Each table is indexed by the lower eleven bits of the received command word, CW bits 10:0. The table returns an integer with a predefined unique value for each command type. Other values denote illegal commands. The user can edit file `tables.c` to illegalize any combination of command transmit-receive bit, subaddress and word count or mode code. See the text header at the top of file `tables.c` for complete details.

Function `process_new_command()` uses the fetched decode table value as argument in a switch statement. The switch code calls the appropriate C-handler for the received command type. Command handlers for all command types and illegal commands are contained in file `1553cmd.c`. When complete, the command handler returns to function `process_new_command()` where most messages await the command result (VALMESS or ERROR) from the HI-6110. If VALMESS occurs the message was successful and function `process_new_command()` returns to function `standby()`. If an ERROR interrupt occurs instead of VALMESS, the ERROR interrupt C-handler performs its programmed tasks then returns to the top of `standby()` by the method described earlier.

The example program is fully annotated. Program structure should be reasonably clear.

USER INTERFACE

Programmed behavior of LEDs, DIP switch and pushbuttons is described here.

Diode LED2 is turned on during reset initialization. When lit, LED2 indicates reset just occurred due to power disruption, mode code MC8 or manual RESET button press. LED2 is extinguished when the first valid MIL-STD-1553 command is received.

During reset initialization, the ARM7 processor initializes the HI-6110 Control register by writing a value to it. The processor then reads back the value written. If mismatch occurs (or no HI-6110 response) two red LEDs below the Bus A and Bus B jacks will quickly flash alternately.

The board's DIP switch sets the 5-bit RT Address and Parity bit. MIL-STD-1553 calls for odd parity. The HI-6110 checks for valid RT address-parity after its MR input goes low. If address-parity mismatch is detected the HI-6110 Error Register is written and ERROR goes high. When the ARM7 processor detects this condition. two red LEDs below the Bus A and Bus B jacks flash slowly. This state will persist until the DIP switch settings are changed to a valid address-parity setting and the board is then reset using power-up reset or the RESET pushbutton.

The two green bus LEDs quickly alternate while executing the "do nothing" loop within function `standby()`. This serves as a visual indicator of valid bus activity.

Only the RESET pushbutton is connected on the default evaluation board. Jumpers are provided to connect pushbuttons SW1 or SW2 as processor inputs. With appropriate software changes, these buttons provide a convenient way to set and reset Subsystem, Busy or Terminal Flag bits.

LIST OF PROJECT FILES

HEADER FILES

Holt_6110.h

the hardware feature definition file for the Holt HI-6110 evaluation board. This file defines all ARM7 processor I/O assignments and contains the C structure (struct) for the memory-mapped HI-6110 interface to the ARM7 external bus.

bootConfig.h

provided by Green Hills Software to work with their MULTI Integrated Development Environment, this file sets up debug for "romcopy" project builds. A variable defined in the file tells the compiler to insert a wait loop in the program's reset routine so the Debugger can attach to the program to perform debugging. This file is not used with "ramrun" builds or if using a different compiler.

FILES WRITTEN IN ANSIC

ICompiler should be set to optimize two files for execution speed: main.c and interrupt_ext.c

main.c contains these functions (and other misc functions):

main() performs post-reset initialization, then calls function standby()

standby () first reinitializes user stack pointer. If arrival at **standby ()** after a new RCVA or RCVB interrupt, **process_received_command ()** is called. Otherwise **standby ()** performs repetitive system tasks while awaiting RCV or ERROR interrupts from the HI-6110.

process_received_command () decodes newly received command word and calls the appropriate function from **1553_cmd.c** to perform command-specific tasks, then awaits message result and returns to function **standby ()**.

NOTE: File **main_new.c** has an improved **standby ()** function. See Appendix B for details.

interrupt_init.c contains function

interrupt_init() which is called by **main ()** during post-reset initialization

interrupt_timer.c

C language handlers for timer interrupts

interrupt_ext.c contains these functions:

C handler for irq0 interrupt, ERROR from HI-6110
C handler for irq1 interrupt, RCVA from HI-6110
C handler for irq2 interrupt, RCVB from HI-6110
C handler for PIO interrupts, RCMDA and RCMDB from the HI-6110

1553_cmd.c contains these functions:

C handlers for each legal 1553 command type
C handlers for illegal commands

Note: Some optional mode codes are not fully implemented.

tables.c

two command decoding tables, and C functions which index the tables. The tables define legal and illegal 1553 commands and return an argument used in 1553 command decoding.

aic_utils.c

provided by Green Hills Software, the function in this file initializes the ARM7 Advanced Interrupt Controller (AIC). A different compiler will perform equivalent tasks, but will not use this file.

LOW LEVEL FILES WRITTEN IN ARM ASSEMBLY

interrupts_init_lowlevel.arm

provided by Green Hills Software, this routine performs stack and IRQ management initialization. If using a different compiler, compilation will perform equivalent tasks, but will not use this file.

vectors.arm

provided by Green Hills Software, this file defines ARM exception/interrupt vector addresses. If using a different compiler, compilation will perform equivalent tasks, but not use this file.

irq_ghs.mac

provided by Green Hills Software, this file contains ASM macros used for **IRQ_ENTRY** and **IRQ_EXIT**. This file was edited by Holt to add new macro **IRQ_ENTRY_SPECIAL**.

irq_pio.arm

provided by Green Hills Software or Atmel, file contains low level ASM handler for PIO interrupts

irq_irq0.arm

provided by Green Hills Software or Atmel, this file contains the low level ASM handler for IRQ0 interrupt (HI-6110 ERROR signal). This file was edited by Holt to replace the normal GHS macro call **IRQ_ENTRY** with macro **IRQ_ENTRY_SPECIAL**.

irq_irq1.arm

provided by Green Hills Software or Atmel, this file contains the low level ASM handler for IRQ1 interrupt (HI-6110 RCVA signal). This file was edited by Holt to replace the normal GHS macro call **IRQ_ENTRY** with macro **IRQ_ENTRY_SPECIAL**.

irq_irq2.arm

provided by Green Hills Software or Atmel, this file contains the low level ASM handler for IRQ2 interrupt (HI-6110 RCVB signal). This file was edited by Holt to replace the normal GHS macro call **IRQ_ENTRY** with macro **IRQ_ENTRY_SPECIAL**.

APPENDIX A ELIMINATING FALLING EDGE PIO INTERRUPTS FOR ATMEL ARM7 PROCESSOR

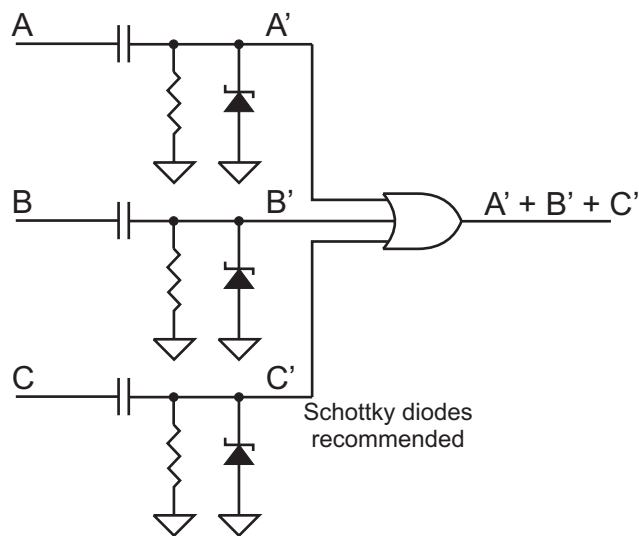
In our design, we are just concerned with rising edge interrupts for HI-6110 signals RCMDA and RCMDDB. But the Atmel ARM7 generates PIO interrupts for both rising and falling edges. In the example software, falling edge PIO interrupts are a manageable nuisance since we have just two PIO interrupt signals. If application requirements call for additional PIO interrupt inputs, here is a way to eliminate nuisance falling edge PIO interrupts.

This circuit uses an R-C differentiator on each PIO interrupt input to generate a brief positive pulse ($\sim 1\mu\text{s}$) when a rising edge occurs. Nothing is generated for falling edges. The brief pulses are ORed and the OR gate output signal is programmed in software as the only PIO input source.

For example, assume the application needs three PIO interrupts, A, B and C, depicted below. The ORed interrupt signal $A'+B'+C'$ connects to a PIO input programmed as the only PIO interrupt source. Two other inputs connect directly to A and B. The interrupt routine polls A and B to determine the active interrupt source. Signal C is identified active by exception. If polling finds A and B inputs both low, then signal C must be the active interrupt. For higher reliability, the application can poll an additional C input if a spare input pin is available.

Once a PIO interrupt is received by the Atmel ARM7, the next PIO interrupt cannot be triggered until the interrupt handler clears the existing PIO interrupt in software. This occurs at the first C statement in `pio_c_irq_handler()`.

Converting long duration interrupt signals (A, B or C below) to a brief pulse ($A'+B'+C'$) is only helpful if the brief pulse goes low before the interrupt handler clears the existing PIO interrupt. This scheme eliminates nuisance falling edge interrupts (when A falls for example) and emulates the behavior of processors besides Atmel ARM7 that can be configured to only generate IRQs for rising edge inputs.

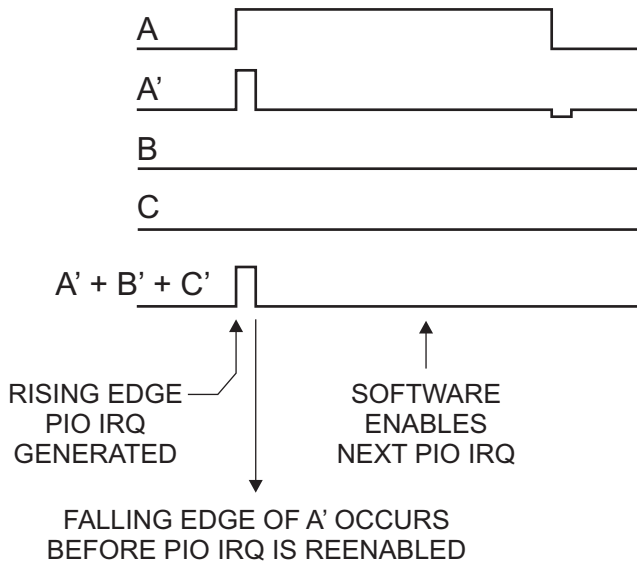


Page 2 of the evaluation board schematic diagram shows a "PIO Interrupt Positive Edge Detector and Logical OR" circuit using the described method. By default, this circuit is not used: PIO interrupts occur for RCMDA and RCMDDB falling edges. This configuration applied in validation testing.

To enable logical ORing for RCMDA and RCMDDB on the evaluation board, hardware board changes are needed: Close jumper JP7. Open jumper JP21. Program changes are also required. These changes simplify execution and enhance performance when PIO interrupts are added:

1. In function `pio_c_irq_handler()` remove the code that disables PIO interrupts. This code temporarily disabled PIO interrupts in the interval when falling edges occur for the HI-6110 signals RCMDA and RCMDDB.
2. In function `standby()`, the statement that reenables the PIO interrupts can be removed. This was the complementary action to the program task removed in the step above.
3. In the function `pio_c_irq_handler()`, the determination of active PIO interrupt source must be modified to poll only RCMDA. The input previously used for RCMDDB is now PIO interrupt input for the OR signal.
4. File `interrupt_init.c` configures both RCMDA and RCMDDB as PIO interrupt inputs. Edit this file so the only PIO interrupt source is the logically-ORed signal. After jumper changes described above, the ORed signal is present at the RCMDDB input pin as defined in header file `Holt_6110.h`.

The first C statement in `pio_c_irq_handler()` enables the next PIO IRQ. **Make sure this statement always executes after falling edge of OR gate signal!**



APPENDIX B

DEBUGGING INTERRUPT MANAGEMENT FOR RCVA, RCVB & ERROR IRQS

As described earlier, these three interrupts do not use conventional return from interrupt. Instead, the C-handlers branch to the starting address of the function, `standby()`. At each reentry, the top of function `standby()` reinitializes the user stack pointer to the same value that applied the first time `standby()` was called from `main()` after reset.

User SP reinitialization is comprised of two steps: (1) loading user SP with an immediate value determined from the GHS compiler / linker, then (2) adjusting the SP by subtracting a fixed value.

The adjustment value may change if program edits alter execution occurring before first arrival at function `standby()` after reset.

If malfunction occurs, here are the steps for testing and correcting the step (2) adjustment value. The new value should remain valid as long as start-up programming remains unchanged.

Should malfunction occur, follow these steps:

1. Set up the program Debugger with a breakpoint at the top of function `standby()`.
2. Perform reset using the Debugger interface.
3. When execution stops at the breakpoint, use the Debugger register window. Record the contents of user stack pointer, register R13. Because the Debugger uses ARM Supervisor (svc) mode, the Debugger may identify the user SP register as SVC_R13.
4. Single step execution through the first ASM instruction. This loads R13 with the value initialized by the compiler/linker.
5. Single step execution through the second ASM instruction that subtracts a fixed offset from R13. If the offset value is correct, R13 will return to the value recorded at step 3.
6. If step 5 results in a mismatch with the value recorded at step 3, adjust the offset value subtracted at step 5 and recompile/rebuild the project. Repeat steps 1 – 5 to verify that steps 3 and 5 give the same R13 value.

IMPORTANT: Improved Method Found Later !

The ARM7 software preprogrammed on the HI-6110 evaluation board is exactly what was used for Remote Terminal validation in December 2006. After completion of validation testing, a vastly improved method for user stack pointer reinitialization in `standby()` was found. This method is self-adjusting and no maintenance is required.

The improvement can be evaluated by replacing project file `main.c` with the file `main_new.c` found in the same directory. The project should then be rebuilt.

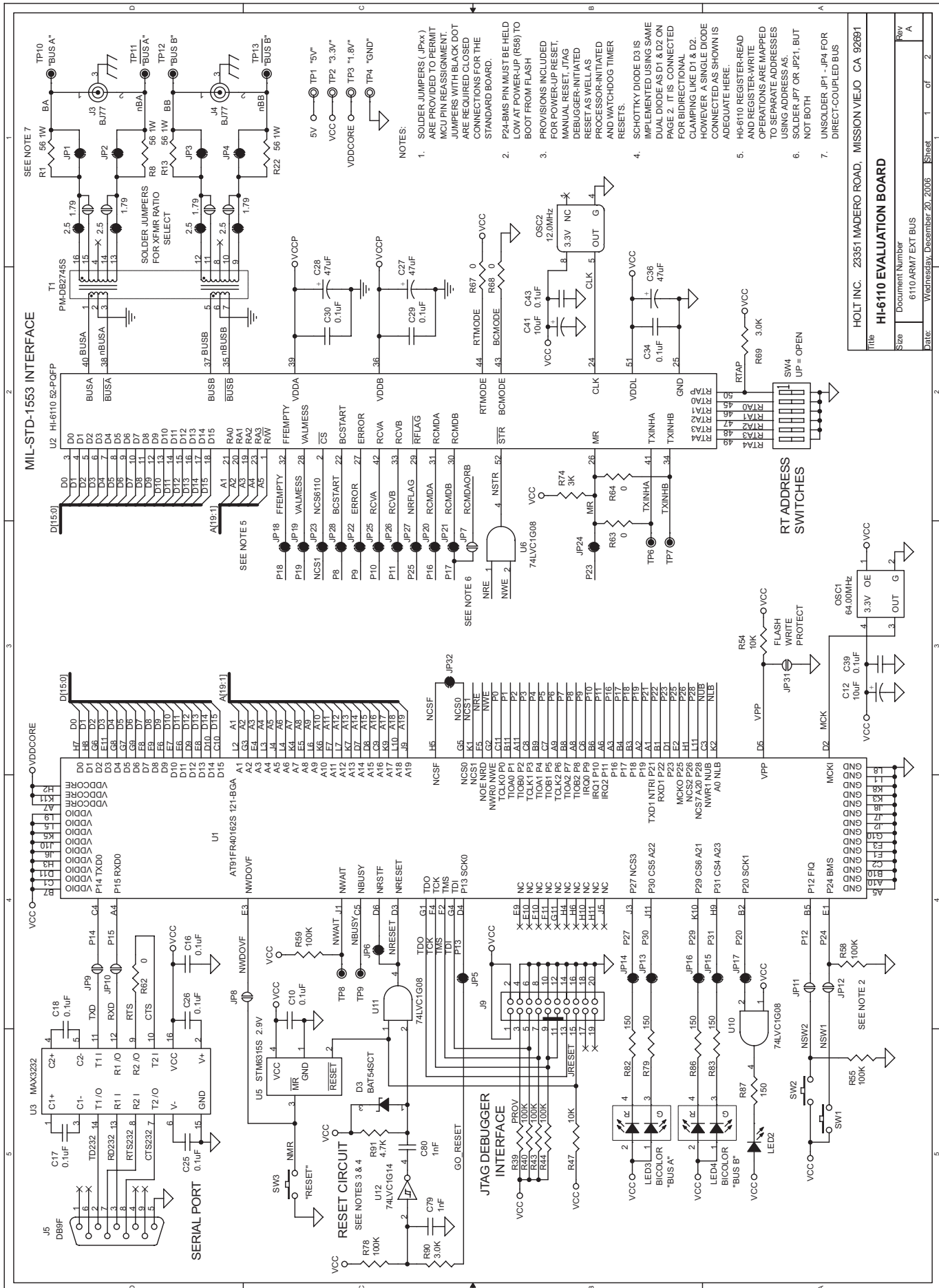
The new file is purposely excluded in the standard project build because it differs from the design that passed validation testing. No functional differences should arise from this file substitution. Added benefits include (a) predictable behavior during software development, (b) consistent behavior between “ramrun” and “romcopy” project builds and (c) no adjustments needed if initialization software is modified.

The improved version works as follows: In function `standby()`, a new static variable `saved_sp` is declared. Upon first entry to function `standby()` following reset, `saved_sp` is initialized with the user stack pointer value read using an “asm macro”. Upon subsequent returns to `standby()`, a different “asm macro” copies the `saved_sp` value to the user stack pointer.

For information on “asm macros” see Green Hills Software Help documentation entitled, “*Enhanced ASM Macros for C/C+*”.

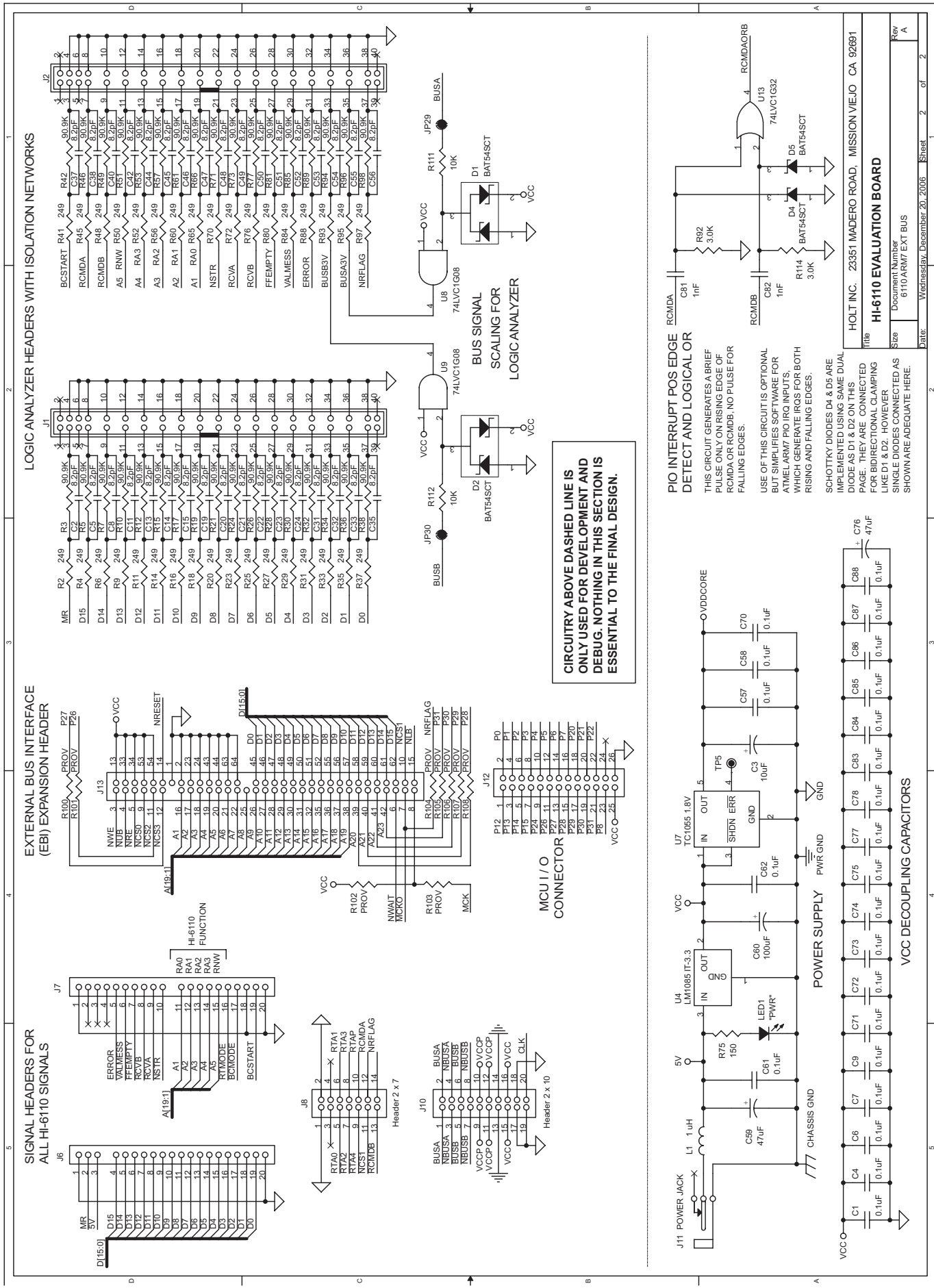
**APPENDIX C
SCHEMATIC DIAGRAM
EXAMPLE HI-6110 ARM7 REMOTE TERMINAL**

MIL-STD-1553 Remote Terminal Design Using the HI-6110

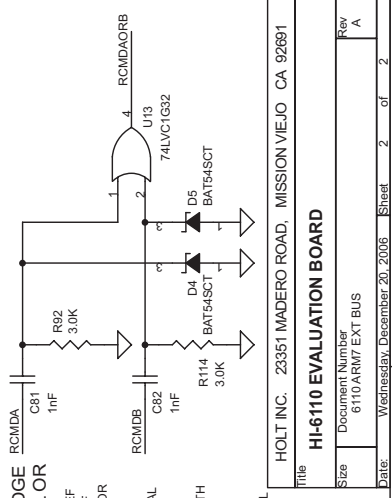


- NOTES:**
- SOLDER JUMPERS (JPxx) ARE PROVIDED TO PERMIT MCU PIN REASSIGNMENT. JUMPERS WITH BLACK DOT ARE REQUIRED CLOSED CONNECTIONS FOR THE STANDARD BOARD.
 - P24-BMS PIN MUST BE HELD LOW AT POWERUP (R59) TO BOOT FROM FLASH.
 - PROVISIONS INCLUDED FOR POWER-UP RESET, MANUAL RESET, JTAG DEBUGGER-INITIATED RESET AS WELL AS PROCESSOR-INITIATED AND WATCHDOG TIMER RESETS.
 - SCHOTTKY DIODE D3 IS IMPLEMENTED USING SAME DUAL DIODE AS D1 & D2 ON PAGE 2. IT IS CONNECTED FOR BIDIRECTIONAL CLAMPING LIKE D1 & D2 HOWEVER A SINGLE DIODE CONNECTED AS SHOWN IS ADEQUATE HERE.
 - HI-6110 REGISTER-READ AND REGISTER-WRITE OPERATIONS ARE MAPPED TO SEPARATE ADDRESSES USING ADDRESS A5.
 - SOLDER JP7 OR JP21, BUT NOT BOTH
 - UNWELDER JP1 - JPA FOR DIRECT-COUPLED BUS

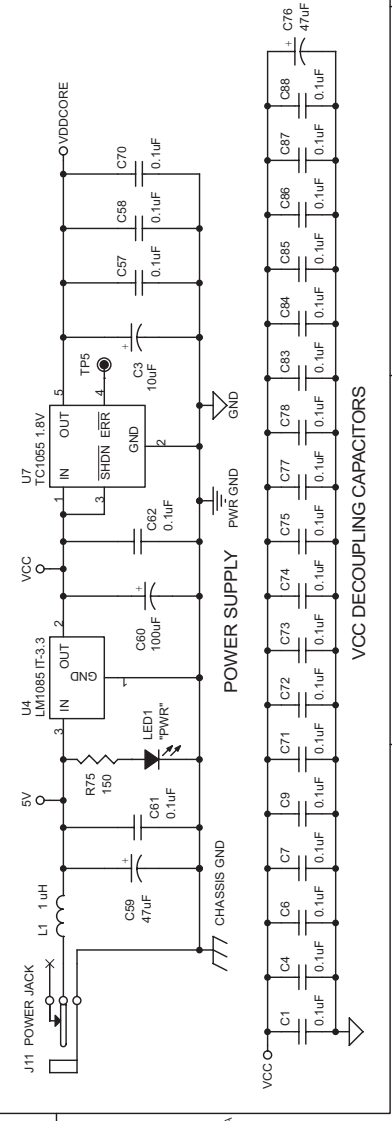
Title	HOLT INC. 23351 MADERO ROAD, MISSION VIEJO CA 92681
Doc Number	HI-6110 EVALUATION BOARD
Size	6110ARW7 EXT BUS
Date	Wednesday, December 20, 2006
Sheet	1 of 2
Rev	A



CIRCUITRY ABOVE DASHED LINE IS ONLY USED FOR DEVELOPMENT AND DEBUG. NOTHING IN THIS SECTION IS ESSENTIAL TO THE FINAL DESIGN.



USE OF THIS CIRCUIT IS OPTIONAL BUT SIMPLIFIES SOFTWARE FOR ATWEL ARM7 PIO I/O INPUTS, WHICH GENERATE IRQS FOR BOTH RISING AND FALLING EDGES. SCHOTTKY DIODES D4 & D5 ARE IMPLEMENTED USING SAME DUAL DIODE AS D1 & D2 ON THIS PAGE. THEY ARE CONNECTED FOR BIDIRECTIONAL CLAMPING LIKE D1 & D2. HOWEVER, SINGLE DIODES CONNECTED AS SHOWN ARE ADEQUATE HERE.



HOLT INC. 23351 MADERO ROAD, MISSION VIEJO CA 92691
HI-6110 EVALUATION BOARD
 Document Number 6110 ARM7 EXT BUS
 Date: Wednesday, December 20, 2006 Sheet 2 of 2