C166 Bootstrap Code Developer's Toolkit

DRAFT
Version 1.00
October, 1996

For Limited Distribution Only

RIGEL CORPORATION
PO Box 90040
Gainesville, Florida 32607
(352) 373-4629
FAX (352) 373-1786
www.rigelcorp.com

(C) 1996 by Rigel Corporation.

Legal Notice:
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Rigel Corporation.

The abbreviation PC used throughout this guide refers to the IBM Personal Computer or its compatibles. IBM PC is a trademark of International Business Machines, Inc. MS Windows is a trademark of Microsoft, Inc.

Information in this document is provided solely to enable use of Rigel products. Rigel assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Rigel products except as provide in Rigel's Customer Agreement for such products.

Rigel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Rigel retains the right to make changes to these specifications at any time without notice.

Contact Rigel Corporation or your Distributor to obtain the latest specifications before placing your order.

Our Policy:
We attempt to up date all software, software manuals, and hardware manuals every 3-6 months. If there is a problem with the software or documentation it is corrected immediately. The newest version is then put on our home page (http://www.rigelcorp.com) with the date noted. Documentation is coded with version number and a date. The version number is the version of the board (V1.0), and the date (September 1996) refers to the last date the document was written.

We welcome any and all comments about our products.

WARRANTY
RIGEL CORPORATION- CUSTOMER AGREEMENT

1. Return Policy. This return policy applies only if you purchased the products directly from Rigel Corporation. If you are not satisfied with the items purchased, prior to usage, you may return them to Rigel Corporation within thirty (30) days of your receipt of same and receive a full refund from Rigel Corporation. You will be responsible for shipping costs. Please call (352) 373-4629 prior to shipping. A refund will not be given if the READS package has been opened.

2. Limited Warranty. Rigel Corporation warrants, for a period of sixty (60) days from your receipt, that software disk(s), hardware assembled boards and hardware unassembled components shall be free of substantial errors or defects in material and workmanship which will materially interfere with the proper operation of the items purchased. If you believe such an error or defect exists, please call Rigel Corporation at (352) 373-4629 to see whether such error or defect may be corrected, prior to returning items to Rigel Corporation. Rigel Corporation will repair or replace, at its sole discretion, any defective items, at no cost to you, and the foregoing shall constitute your sole and exclusive remedy in the event of any defects in material or workmanship.

THE LIMITED WARRANTIES SET FORTH HEREIN ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

YOU ASSUME ALL RISKS AND LIABILITY FROM OPERATION OF ITEMS PURCHASED AND RIGEL CORPORATION SHALL IN NO EVENT BE LIABLE FOR DAMAGES CAUSED BY USE OR PERFORMANCE, FOR LOSS PROFITS, PERSONAL INJURY OR FOR ANY OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES. RIGEL CORPORATION'S LIABILITY SHALL NOT EXCEED THE COST OF REPAIR OR REPLACEMENT OF DEFECTIVE ITEMS.

IF THE FOREGOING LIMITATIONS ON LIABILITY ARE UNACCEPTABLE TO YOU, YOU SHOULD RETURN ALL ITEMS PURCHASED TO YOUR SUPPLIER.

3. READS 166 (referred to as simply READS) License. The READS being purchased is hereby licensed to you on a non-exclusive basis for use in only one computer system and shall remain the property of Rigel Corporation for purposes of utilization and resale. You acknowledge you may not duplicate the READS for use in additional computers, nor may you modify, disassemble, translate, sub-license, rent or transfer electronically READS from one computer to another, or make it available through a timesharing service or network of computers. Rigel Corporation maintains all proprietary rights in and to READS for purposes of sale and resale or license and re-license.

BY BREAKING THE SEAL AND OTHERWISE OPENING THE READS PACKAGE, YOU INDICATE YOUR ACCEPTANCE OF THIS LICENSE AGREEMENT, AS WELL AS ALL OTHER PROVISIONS CONTAINED HEREIN.

4. Board Kit. If you are purchasing a board kit, you are assumed to have the skill and knowledge necessary to properly assemble same. Please inspect all components and review accompanying instructions. If instructions are unclear, please return the kit unassembled for a full refund or, if you prefer, Rigel Corporation will assemble the kit for a fee of $30.00. You shall be responsible for shipping costs. The foregoing shall apply only where the kit is unassembled. In the event the kit is partially assembled, a refund will not be available, however, Rigel Corporation can, upon request, complete assembly for a fee based on an hourly rate of $50.00. Although Rigel Corporation will replace any defective parts, it shall not be responsible for malfunctions due to errors in assembly. If you encounter problems with assembly, please call Rigel Corporation at (325) 373-4629 for advice and instruction. In the event a problem cannot be resolved by telephone, Rigel Corporation will perform repair work, upon request, at the foregoing rate of $50.00 per hour.

5. **Governing Law**. This agreement and all rights of the respective parties shall be governed by the laws of the State of Florida.

1. Overview

The SAC C166 and ST10F166 family of microcontrollers allow the user code to be downloaded without the need of any system ROM. Such downloaded code can then be placed in nonvolatile memory, such as FLASH EEPROMs or battery-backed RAMs, and run the system. This feature is often used to download monitor programs or test programs to evaluation boards. Once a product is developed, this feature provides a convenient means to upgrade the system code in the field.

This document and accompanying utility programs allow the code developer to create custom bootstrap loaders. The document is written from the programmer's perspective. Refer to the data books, or contact Rigel Corporation regarding the hardware design elements needed to place the microcontrollers into the bootstrap mode.

2. Bootstrap Basics

The microcontroller is placed in a bootstrap mode only if certain conditions are met at a hardware interrupt. In C166 processors, the level of the ALE line is interrogated. In the C167 core, the port pin P0.4 is sampled. If the logic levels of the lines meet the conditions, the microcontroller enters the bootstrap mode. It is noteworthy that software resets disregard these conditions and thus a software reset never causes the microcontroller to enter the bootstrap mode. In fact, a software reset is almost always used to leave the bootstrap mode and run the system program downloaded by bootstrapping.

Although we think of the microcontroller being bootstrapped to be a ROM-less device, it actually has a minimal program which runs the bootstrapper logic. Bootstrapping uses serial port 0. It expects asynchronous serial communications with 8 data bits, 1 stop bit, and no parity bit. The Baud rate is determined by the bootstrap logic. From the viewpoint of the microcontroller, the bootstrapper accomplishes the following tasks.

1. Wait for a (binary) zero byte. That is a start bit, 8 zero bits, and a stop bit. The time it takes to transmit the 8 zero bits determines the Baud rate.

2. Transmit an identification byte, also referred to as an acknowledgment byte. The value of the identification byte depends on the microcontroller. The C166 microcontroller, for example, returns 055h, while the C167CR microcontrollers return a 0C5h.

3. Wait for 32 more bytes and place these bytes at internal RAM starting at 0FA40. The microcontroller remains in the bootstrap mode until all 32 bytes are received.

4. Once the 32 bytes are received, the microcontroller branches to address FA40 and starts executing.

It should be noted that while the microcontroller is in the bootstrap mode, all registers contain values which are determined by the hardware reset. Also, when the

microcontroller starts executing the 32 bytes just downloaded, it is still in the bootstrap mode. Another reset is needed to terminate the bootstrap mode. A software reset is usually preferred since it ignores the physical conditions which force the microcontroller to enter the bootstrap mode. Thus, without removing a jumper on P0.4, for example, a C167 may leave the bootstrap mode simply by executing the SRST instruction. If a hardware reset is used, the physical conditions which causes the microcontroller to enter the bootstrap mode must be removed.

Although the basic process is straightforward, there are a few issues which need attention in using the bootstrap loader mechanism.

1. Clearly, the 32-byte block is not large enough to contain all the necessary user code. This block must be stuffed with the precious few instructions which allow downloading the rest of the user program.

2. From a hardware point the software reset is the more convenient way to exit the bootstrap mode. Once the software reset is invoked, the microcontroller starts executing from memory address 0. Address 0 will almost always be in external memory. (If code exists in internal ROM, we need not bootstrap the microcontroller.) Thus the microcontroller registers, namely SYSCON and BUSCON0, must be configured to access external memory. Some start up code must also be placed at address 0.

Systematic approaches to these issues are developed in the remainder of this document.

3. The First 32 Bytes

All bootstrapping starts with the first 32 bytes written to internal memory at address 0FA40h. These 32 bytes may hold up to 16 instructions, clearly not enough to setup all registers and run an application. Instead, the first 32 bytes are almost always a loop to download more instruction bytes. Consider the following secondary loop.

```
      mov     R0, #0FA60h              ; R0 is the address to the next byte
 W0:
      jnb     S0RIR, W0                ; wait until a byte is received
      movb    [R0], S0RBUF             ; place the received byte in memory
      bclr    S0RIR                    ; clear serial receive flag
      movb    S0TBUF, [R0]             ; transmit (echo) the byte
      cmpi1   R0, #(0FA60h+NUMBYTES-1) ; see if all bytes are received
     jmpr    cc_NE, W0                 ; if not repeat
     nop
      nop
      nop
      nop
```

This loop takes exactly 32 bytes to implement. Note that NOP (no operations) instructions are used to pad the loop to make it exactly 32 bytes. Once the 32-byte loop is received, the microcontroller branches to 0FA40h and starts executing this 32-byte loop. This loop expects more characters to be received and places the received characters in internal memory. So the host which sends the 32-byte loop may continue

to send more bytes, now to be placed in memory by the loop rather than by the microcontroller's bootstrap logic.

The loop uses the register R0 as a pointer to memory. The received bytes are placed in internal memory starting at address 0FA60. This is not an arbitrary value. It is 0FA40h + 20h, or the address of the memory immediately after the loop. This way, when the loop finishes downloading the bytes, execution simply continues, and after the four NOP instructions, the instruction bytes downloaded by the loop are executed.

As mentioned, this scheme is convenient from the viewpoint of the host computer which sends the bootstrap code. From this point of view, we send the 32-byte loop followed by more instruction bytes as a single block. We refer to this block as the **bootstrap script**. The bootstrap script, which has a variable length determined by the constant NUMBYTES, is placed in internal memory and executed. Although such execution is made up of two phases, from the programmer's perspective, it is a single logical step: download a variable-length bootstrap script and it will be executed from internal RAM.

4. The Bootstrap Script

The loop to download more instruction bytes implemented by the 32-byte code is referred to as the **primary loop**. The block of additional instruction bytes downloaded by the primary loop typically has two tasks: initialize the external bus control logic (for example, SYSCON and BUSCON0) and download a block of code to external memory starting from address 0. The block of additional instruction bytes downloaded by the primary loop is referred to as the **secondary loop**. When the secondary loop is executed, the system is ready to leave the bootstrap mode. That is, the external bus is configured and a program is downloaded to external memory starting at address 0. The secondary loop terminates by a software reset (SRST) instruction, and the code loaded by the secondary loop is executed. The bootstrap script thus consists of the primary loop and the secondary loop.

4.1 The Primary Loop

The simple primary loop described in the previous section is not the only possible loop. For example, it is not necessary to start placing the additional instruction bytes immediately after the loop. Say we place the bytes at 0FA80h. The four NOP instructions may be replaced by a jump to address 0FA80h. This effectively leaves a gap of 32 more bytes between the loop and the additional instruction bytes. This gap may perhaps be used by the additional instruction bytes as a scratchpad area for storing variables. It should be noted that there is a limit on internal RAM, and thus any such gap should be kept small.

Another common alteration to the 32-byte loop is suppressing the byte echo. Although echoing each byte back to the host may be used as a way of verifying correct reception, it may place unnecessary stress on the host computer. Our experience with MS Windows indicates that waiting for each byte echo before sending the next byte may considerably slow down the bootstrap and download operations, especially with slower

386 machines. If correct reception is important, a checksum may be computed after the bootstrap and download process and reported as a single byte or word.

4.2 The Secondary Loop

The secondary loop has two tasks: configure and enable the external bus and download instruction bytes to be placed in external memory starting from address 0. The secondary loop terminates by a software reset instruction, which also terminates the bootstrap mode.

Consider the following secondary loop for the C167CR processor..

```
; ********************************************************************
; --- PART 1: System initialization ---
; ********************************************************************

; --- initialize bus configuration ---------------------------------
      mov     SYSCON, #F480h
      nop
      mov     SYSCON, #F480h
      nop

; --- initialize CPS -----------------------------------------------
      jmps    far next          ; far intersegment jump to update CPS
next:
; --- initialize DPPx ----------------------------------------------
      mov     DPP0, #0
      mov     DPP1, #1
      mov     DPP2, #2
      mov     DPP3, #3

; --- initialize ADDRSEL1 ------------------------------------------
      mov     ADDRSEL1, #1008h

; --- set up WR# as an output --------------------------------------
      bset    P3.13             ; set WR#
      bset    DP3.13            ; make WR# an output

; --- set up serial port for 40MHz oscillator frequency ------------
      bset    DP3.10
      mov     S0CON, #8011h
      mov     S0BG,  #3Fh
      mov     S0TIC, #0
      mov     S0RIC, #0
      mov     S0EIC, #0

; --- disable watchdog timer ---------------------------------------
      DISWDT

; --- issue end-of-initialization command --------------------------
      EINIT

; --- system initialization done -----------------------------------


; ********************************************************************
; --- PART 2: Notify host ---
; ********************************************************************
      mov     R0, #'#'    ; send a special character, say '#'
      mov     S0TBUF, R0  ; transmit special character
W1:
```

```
      jnb     S0TIR, W1   ; wait until character is transmitted
      bclr    S0TIR       ; clear transmit completion indicator flag

; ****************************************************************


; --- PART 3: Load code starting from 0 ---
; ****************************************************************
; ----------------------------------------------------------------
      mov     R0, #0            ; R0 holds the address
W2:
      jnb     S0RIR, W2         ; wait until a byte is received
      movb    [R0], S0RBUF      ; save byte
      bclr    S0RIR             ; clear receive flag
      movb    S0TBUF, [R0]      ; echo byte back to host
      cmpi1   R0, #(NUMBYTES-1) ; repeat NUMBYTES times
      jmpr    cc_NE, W2
; ****************************************************************
; --- PART 4: Terminate secondary loop and execute code from 0 ---
; ****************************************************************
      SRST
```

The secondary loop is divided into 4 parts for clarity. Part 1 initializes the important system registers, sets up the serial port, and disables the watchdog timer. The end-of-initialization command (EINIT) is issued as the last order of business in Part 1.

Part 2 is an optional step. A notification character is sent to the host. This may be a more meaningful character or string. Similarly, a checksum may be sent back to verify reception integrity.

Part 3 is the actual loop that loads instruction bytes to external memory starting from address 0. This loop is similar to the primary loop. Again, bytes are echoed back to the host. This may be suppressed if it causes transmission congestion. The number of bytes such downloaded is determined by the constant NUMBYTES. Note that there is typically much more external memory than internal memory. The constant NUMBYTES may be a large number. The bootstrap script that is given with Rigel's evaluation boards uses this loop to download the minimal monitor RMinMon6x.

The last part terminates the secondary loop and causes the microcontroller to execute the instruction bytes downloaded to external memory, starting at address 0.

4.3 The Bootstrap Script Syntax

The bootstrap script is entered into the Rigel utility tools in a script format as shown below. As in assembly language, a semicolon marks the beginning of a comment. The individual bytes are represented as hexadecimal numbers in the C language syntax. For example, 0xE6 represents 0E6h or the decimal 230.

```
;================================================================ +
;  Primary Loop                                                     |
;================================================================ +
0xE6 0xF0 0x60 0xFA  ;      mov     R0, #0fa60h
                     ; W0:
0x9A 0xB7 0xFE 0x70  ;      jnb     S0RIR, W0
```

```
0xA4 0x00 0xB2 0xFE  ;      movb    [R0], S0RBUF
0x7E 0xB7            ;      bclr    S0RIR

0x86 0xF0 0xD3 0xFA  ;      cmpi1   R0, #0FAD3h   ; read 116 bytes (0x74)
0x3D 0xF8            ;      jmpr    cc_NE, W0

0xE6 0xF0 0x21 0x00  ;      mov     R0, #'!'
0xF6 0xF0 0xB0 0xFE  ;      mov     S0TBUF, R0

0xCC 0x00            ;      nop
0xCC 0x00            ;      nop

;============================================================== +
;  Secondary Loop                                                |
;============================================================== +
; (0xFA60)
; --- initialize bus configuration --------------------------------

0xE6 0x89 0x80 0xF4  ;      mov     SYSCON, #F480h
0xCC 0x00            ;      nop
0xE6 0x89 0x80 0xF4  ;      mov     SYSCON, #F480h
0xCC 0x00            ;      nop

; --- initialize CPS ------------------------------------------------
                     ;      jmps    far next      ; update CPS
0xFA 0x00 0x70 0xFA
                     ;next:
; (0xFA70)
; --- initialize DPPx -----------------------------------------------
0xE6 0x00 0x00 0x00  ;      mov     DPP0, #0
0xE6 0x01 0x01 0x00  ;      mov     DPP1, #1
0xE6 0x02 0x02 0x00  ;      mov     DPP2, #2
0xE6 0x03 0x03 0x00  ;      mov     DPP3, #3

; (0xFA80)
; --- initialize ADDRSEL1 -------------------------------------------
0xE6 0x0C 0x08 0x10  ;      mov     ADDRSEL1, #1008h

; --- set up WR# as an output ---------------------------------------
0xDF 0xE2            ;      bset    P3.13             ; set WR#
0xDF 0xE3            ;      bset    DP3.13            ; make WR# an output

; --- set up serial port --------------------------------------------
0xAF 0xE3            ;      bset    DP3.10
0xE6 0xD8 0x11 0x80  ;      mov     S0CON, #8011h
0xE6 0x5A 0x3F 0x00  ;      mov     S0BG,  #3Fh
0xE6 0xB6 0x00 0x00  ;      mov     S0TIC, #0
0xE6 0xB7 0x00 0x00  ;      mov     S0RIC, #0
0xE6 0xB8 0x00 0x00  ;      mov     S0EIC, #0
; --- serial port initialized for 40MHz oscillator frequency ---


; (0xFA9E)
; --- system initialization done ------------------------------------
0xa5 0x5a 0xa5 0xa5  ;      DISWDT
0xB5 0x4A 0xB5 0xB5  ;      EINIT


; (0xFAA6)
                     ; W1:
0x9A 0xB6 0xFE 0x70  ;      jnb     S0TIR, W1
0xE6 0xF0 0x23 0x00  ;      mov     R0, #'#'
0xF6 0xF0 0xB0 0xFE  ;      mov     S0TBUF, R0
                     ; W2:
0x9A 0xB6 0xFE 0x70  ;      jnb     S0TIR, W2
```

```
0x7E 0xB6            ;      bclr    S0TIR

; (0xFAB8)
; ------------------------------------------------------------------
; --- load code starting from 0 ---
0xE6 0xF0 0x00 0x00  ;      mov     R0, #0
;                ; W0:
0x9A 0xB7 0xFE 0x70  ;      jnb     S0RIR, W0
0xA4 0x00 0xB2 0xFE  ;      movb    [R0], S0RBUF
0x7E 0xB7            ;      bclr    S0RIR

;0xB4 0x00 0xB0 0xFE ;      movb    S0TBUF, [R0]
0xCC 0x00            ;      nop
0xCC 0x00            ;      nop

; --- the count here must match the size of minmon ---
0x86 0xF0 0x4F 0x01  ;      cmpi1   R0, #014Fh       ; read 336 bytes (0x0150)

0x3D 0xF6            ;      jmpr    cc_NE, W0

0xB7 0x48 0xB7 0xB7  ;      SRST


; (0xFAD4)
```

5. The User Code

The secondary loop downloads the variable-length user code starting at address 0. This may be the application code which is executed upon the software reset. Note, however, that the user code downloaded by the secondary loop must be a consecutive block of code. The application code is not always single contiguous block of code. Moreover, often the application code is placed at a higher address while a jump instruction is placed at address 0 to branch to the application. There are two approaches to such concerns:

1. We may modify the secondary loop to accept not only a contiguous block of code, but code along with its address. For example, the secondary loop may accept records in the Intel HEX format.

2. A temporary monitor program may be downloaded by the secondary loop. The monitor program may then communicate with the host and download hex files to various ranges in memory.

The latter approach is implemented by the Rigel's monitor programs. There are two reasons for this preference. First, any extended download program implemented in the secondary loop must be short enough to fit in internal memory. Second, code downloaded to external memory may be written with high-level languages, simplifying the implementation of more sophisticated download schemes.

As mentioned, Rigel's monitor programs load the minimum monitor via the secondary loop. The minimum monitor understands only two commands, to download an Intel HEX file into memory, and to branch to a program in memory. With these two commands, a more elaborate monitor, RMON16x is loaded and executed. The minimum monitor is discarded after the larger monitor is loaded.

6. Binary versus ASCII Communications

The development of bootstrap code involves writing code for the host, writing the bootstrap script, and writing the user code. At least some of the host-to-board communications must be in binary. For example, the initial zero byte and the identification byte returned in binary. Similarly, the first 32-bytes are transmitted in binary.

Files downloaded to, say, the secondary loop may be in ASCII (for example send the two characters C5 as two bytes rather than a single byte of value 0C5h). ASCII transmissions are preferable in the sense that all such communication may be monitored by an ASCII terminal. On the other hand, binary communications require half the number of bytes transmitted, and thus result in faster bootstrap cycles. Since at least the initial part of the bootstrap process requires binary transmission, it is perhaps best to download the user code via the secondary loop also in binary.

7. Bootstrap Code Development Steps and Utility Tools

There are two utility tools internally used by Rigel Corporation in developing bootstrap code: HEX2BTL and BTL2BIN. The first is a program that converts Intel HEX files to the bootstrap script format described in Section 3.3. The second converts files in the bootstrap script format to binary files.

Below is a list of typical steps needed to develop bootstrap code.

1. Write the bootstrap script including the primary and secondary loops. It is often sufficient to use the template given above and simply supply the constants (NUMBYTES) which determine size of the secondary loop and of the user program or minimum monitor.

2. Run BTL2BIN to obtain a binary file of the bootstrap script.

3 Write the minimum monitor (or any other user program) using an assembler. The demo version of Rigel's assembler Ra66 is sufficient to assemble the minimum monitor.

4. Convert the HEX code generated by the assembler to the bootstrap script format using the utility program HEX2BTL. Then use BTL2BIN to obtain a binary file of the minimum monitor (or user program).

5. You are now ready to bootstrap the microcontroller. Simply send the two binary files, first the bootstrap script, and then the minimum monitor to the board.

The host program must be able to send the zero byte, wait for the identification byte, open the two binary files and transmit its contents to the board. The C program XHost.C written in Borland C++ for DOS gives a minimal implementation of such host code.

Perhaps the best way to start developing bootstrapping code is to experiment with existing ones. You may use the existing components as a baseline. It is a good idea to test intermediate versions as you modify the various aspects of the bootstrap code. The bootstrap script X67CR.BTL and the minimum monitor XMM67CR.ASM are written for the RMB167-CRI board.