Developing Firmware in the PC87591x OBD Environment

National Semiconductor
Application Note AN-1195
Itay Frommer
March 2001
Revision 1.0

1.0 Scope

This Application Note describes how to develop firmware in the On Board Development (OBD) environment of the National Semiconductor® PC87591x family of LPC Mobile Embedded Controllers.

Firmware development in OBD environment requires:

- Integrating the program under development ("program") with the Target Monitor library, Tmonlib, to create one executable file ("executable"); see Sections 2.0 and 3.0.
- Loading the executable into the on-chip flash memory, using the Flash Loader; see Section 4.0.
- Debugging the program, using the CompactRISC™ (CR) Debugger; see Section 5.0.

An example directory contains examples that can be used as base code for programs (see Section 6.0). The flow of code in the OBD environment is shown in Section 7.0.

2.0 PC87591x Operating Environments

This section describes PC87591x main operating environments in general and the OBD environment in particular. For further information about operating environments, refer to the *PC87591E and PC87591S LPC Mobile Embedded Controller Datasheet*.

On power-up reset, the PC87591x selects one of the following operating environments:

- Internal ROM Enabled (IRE)
- Development (DEV)
- On Board Development (OBD)

The IRE environment is used while the PC87591x operates in the production system and executes the application. The on-chip flash is the default source of code for the device.

The DEV and OBD environments are used for code debugging. In both DEV and OBD, the interface to a debugger running on the host is enabled using a JTAG-based debugger interface. However, there are some important differences between these environments.

DEV Environment

The DEV environment is used in Application Development Boards (ADB) or In System Emulators (ISE). In this environment, the on-chip flash is replaced with off-chip SRAM memory to allow flexible and fast development of application code. Some pins are allocated for development system use, and the GPIO functions associated with them are replicated using off-chip logic as part of the ADB system.

In DEV environment, an on-board Target Monitor (TMON) is used. This TMON is burned into ROM devices on the ADB. After reset, TMON loads itself into the ADB SRAM and sends a reset message via the JTAG communication channel to communicate with the debugger. It is then ready to load the program to the SRAM (see Figure 1).

In DEV environment, the CR Debugger supplies full debugging functionality.

OBD Environment

The OBD environment is used for debugging PC87591x firmware while it is mounted on its final production board. All application pins have their IRE functionality, and the interface to a debugger running on the host is enabled using the JTAG interface. In this environment, the on-chip flash is the main source of code for the device.

In OBD environment, there is no need for any external memory (ROM or SRAM). A special TMON, Tmonlib, is linked with the program; then both are loaded into the on chip flash using the Flash Loader utility. After reset, the program under development starts being executed from the flash memory; the start code of this program must call Tmonlib for debugging functionality (see Figure 1).

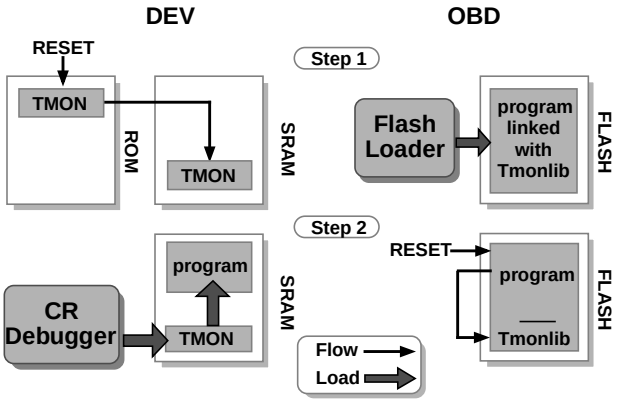

Figure 1. Code Load and Flow in DEV and OBD Environments

The CR Debugger is currently tuned for debugging code loaded into SRAM. Therefore, in OBD environment, special steps must be taken:

- The CR Debugger is not used to load the program into the internal flash; the Flash Loader is used instead.
- The CR Debugger can not place software breakpoints in code in flash memory; instead, hardware breakpoints and fixed breakpoints are used.

The following sections explain how to prepare, load and debug the program in OBD environment.

3.0 Program/Tmonlib Integration

Tmonlib, a CR16B library file (`tmonlib.a`), contains a TMON implementation that is fully compatible with the standard TMON. After its initialization, Tmonlib provides debugging support for the program.

Tmonlib has three operation modes; this application note refers only to one of them, Load-and-Wait. For further information on Tmonlib and its operation modes, refer to the *PC87591 Tmonlib Version 3.1.2.3 Release Letter*.

To debug a program in OBD environment, it must first be integrated with Tmonlib; the resulting linked executable is loaded into the internal flash, using the Flash Loader. Integrating the program with Tmonlib is performed by:

1) Adding Tmonlib calls to the program's start-up routine (Section 3.1).

2) Linking the program and Tmonlib into one executable (Section 3.2).

Start-Up Routine

For program/Tmonlib integration, you must customize the program's start-up routine.

The start-up routine is special code that is executed before the `main` routine; it performs initializations essential for running the program (such as stack-pointer register and data initialization). Its entry point, which is also the entry point to the program, is labeled `start`. The CR Toolset includes the start-up library (`libstart.a`), which contains a default start-up routine. For more details about the start-up routine refer to *The Start-Up Routine* in *CompactRISC™ Introduction*.

The start-up routine source code (`start.s`) is provided with the CR Toolset (`<CR-Tools path>\src`) to enable modifying the start-up routine for specific needs. When using both a modified `start.s` file and `libstart.a` to build the program, there are two `start` labels; since the CR Linker considers only the first label, `start.s` must be specified before `libstart.a` (the flag `-lstart`) in the linkage command.

The start-up routine must be placed in address 0 of the internal flash because after reset, the CR16B core of the PC87591x starts executing instructions from that address.

3.1 ADDING TMONLIB CALLS TO THE START-UP ROUTINE

Two Tmonlib calls are used to integrate Tmonlib with the program: TmonLibStart and Special Supervisor Call (SVC) 108.

3.1.1 TmonLibStart

TmonLibStart is the Tmonlib initialization routine. It initializes:

- Tmonlib variables
- The communication channel
- The interrupt dispatch table

Interrupt Dispatch Table

Every CR program includes an interrupt dispatch table containing the addresses of all exception handlers. The exception list includes traps (e.g., breakpoint trap, trace trap) which are handled by TMON or Tmonlib, and also program-specific interrupts (e.g., timer, WATCHDOG), which are handled by the program.

The CR programming model includes a register, "intbase", which contains the address of the interrupt dispatch table. When an exception occurs, the CR processor uses the intbase register to determine the location of the interrupt dispatch table. For more details, refer to *The Interrupt Dispatch Table* in *CompactRISC™ Introduction.*

Tmonlib and the program share the same dispatch table, as follows:

- The interrupt dispatch table definition is part of the program. It is the program's responsibility to place pointers to the program-specific interrupt handlers (if any) in the interrupt dispatch table.
- The program, in its start-up routine, calls TmonLibStart with one parameter: a pointer to the interrupt dispatch table. In the TmonLibStart routine, Tmonlib places pointers[1] to the trap handlers in the interrupt dispatch table.
- When an exception occurs during program debugging, the appropriate handler is called, as follows:
  - — On a program-specific interrupt, it is an interrupt handler implemented by the program.
  - — On a trap, it is a trap handler implemented by Tmonlib.

Note: SVC 101, used to initialize the interrupt dispatch table in the default `start.s`, must not be used with Tmonlib.

1. Sometimes, the interrupt dispatch table is defined as "const" to save RAM space. In this case, Tmonlib can not place pointers because "const" variables are located in the code section (i.e., flash memory), which Tmonlib can not write to; therefore, the program must initialize the pointers to all handlers in the declaration of the interrupt dispatch table (see Step 2 in Section 3.1.3).

3.1.2 SVC 108

SVC 108 is the entry point for the Tmonlib `_main` loop. It is implemented using the SVC trap with the value 0x108 in register r0. When SVC 108 is called, the Tmonlib `_main` loop is entered and a reset message is sent from Tmonlib to the CR Debugger; therefore, SVC 108 can be called only once. Program execution is halted until the debugger sends a Go command.

Note: SVC 108 is optional; in one of the operation modes of Tmonlib, Load-and-Go, it is not used. For information on Tmonlib operation modes, refer to the *PC87591 Tmonlib Version 3.1.2.3 Release Letter*.

3.1.3 Procedure

Follow these steps to add Tmonlib calls to the start-up routine:

Step 1

1) From the CR-Tools source directory (`<CR-Tools path>\src\libstart`), copy the `start16.s` file to the working directory (where the program under development is located). Note: It is assumed that the CR16B core in the PC87591x is running in Small mode; that is, the code is in the first 128 Kbytes of memory, the dispatch table is in the first 64 Kbytes of memory and each entry in the dispatch table is two bytes long.

2) Rename `start16.s` to `start.s`.

3) In `start.s`, replace all SVC 101 call with Tmonlib calls, as follows:

Change:

```
     movw   $__dispatch_table,r1
     movw   $0x101, r0
     excp   svc
     lpr    r1,intbase

to:
     movw   $__dispatch_table,r2
     lpr    r2,intbase
     bal    ra,_TmonLibStart

     movw   $0x108, r0
     excp   svc

     program_start::
```

The Label: program_start

After reset, the start-up routine is executed up to SVC 108 (see Section 4.2). Usually, the command just after SVC 108, `bal ra,_main`, should be the first command to be executed from the debugger. The label `program_start::`, placed just after SVC 108, is used as the entry point of the program, i.e., the debugger starts executing the program from this label. In some cases, however, `_main` can be used as the entry point (see Step 2 in Section 3.2).

Step 2 (Optional - performed when the internal dispatch table is in flash memory)
To save RAM, the dispatch table can be declared as a const array. The TmonLibStart call can not initialize it properly because it is located in the internal flash (ROM) and Tmonlib does not support flash write operations. In this case, the program must initialize the dispatch table in the declaration, as follows:

Note: Some of the trap's entries of the dispatch table are reserved; a void null handler (`null_handler`) is defined and inserted where there is a reserved entry.

```
#pragma interrupt (null_handler)              //void null_handler declaration
void null_handler(void) {}

extern void NmiHandler(void);                 //extern trap handlers (of Tmonlib)
extern void SvcHandler(void);
extern void DvzHandler(void);
extern void FlgHandler(void);
extern void BptHandler(void);
extern void TrcHandler(void);
extern void UndHandler(void);
extern void DbgHandler(void);
extern void IseHandler(void);

typedef void (*handler_type) (void);          //dispatch table declaration
const handler_type _dispatch_table[] =
{
       null_handler,
       NmiHandler,
       null_handler,
       null_handler,
       null_handler,
       SvcHandler,
       DvzHandler,
       FlgHandler,
       BptHandler,
       TrcHandler,
       UndHandler,
       null_handler,
       null_handler,
       null_handler,
       DbgHandler,
       IseHandler,
       HANDLER_FOR_INTERRUPT_0,
       HANDLER_FOR_INTERRUPT_1,
       ... etc.
}
```

3.2 LINKING THE PROGRAM AND TMONLIB

Figure 2 shows the block diagram for the integration process.

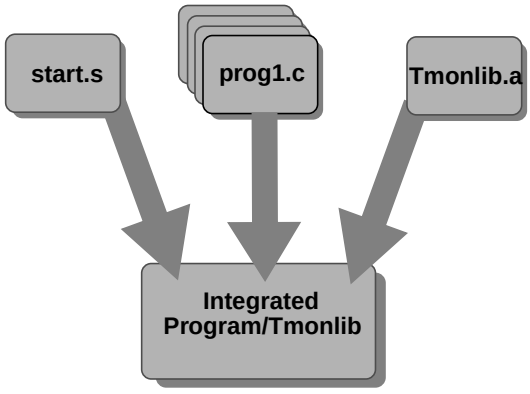

Figure 2. Integration Block Diagram

3.2.1 Procedure

Follow these steps to integrate the program with Tmonlib:

Step 1

Make sure the program's start-up routine is in address 0. If the code (`.text`) section in the linker directive file starts at address 0, then `start.o` must be the first object in the object list.

Note: Another way to ensure that the start-up routine starts at address 0 is to bind the start-up routine to that address. To do this, use a special linker directive in the linker directive file to control the location of the start-up code.

For example:

```
memory {
   ROM: origin=0 length=0xE000
...
}

sections {
   .text BIND(0): { start.o(.text) }
   .text into(ROM): { *(.text) }
...
}
```

Step 2

Use the modified start-up file and linker directive file to create the integrated executable. For example, when the program file (`prog.c`), the modified start-up file (`start.s`) and the appropriate linker directive file (`link.def`) are in the working directory, run:

```
crcc -g start.s prog.c tmonlib.a -Wl,-d,link.def -Wl,-e,program_start
```

Or (step-by-step):

```
crasm -g start.s
crcc -c -g prog.c
crlink start.o prog.o tmonlib.a -d link.def –lstart -e program_start
```

Notes:

- The `–l…` flags are used to specify to the linker which library to use. The default start-up routine contains function calls of functions that are implemented in the start library; therefore, when linking `start.o` step-by-step, the `–lstart` flag must be specified.
- The `-e program_start` flag specifies to the linker that the entry point of the program is `program_start`, i.e., the debugger starts executing the program from this label (see Step 1 in Section 3.1). In some cases, however, the `-e _main` flag can be used instead, as follows: If there are no commands between `excp svc` (i.e., SVC 108) and `bal ra,_main` (in `start.s`), and if `_main` is an endless routine, the `bal ra, _main` command is not necessary for debugging and the debugger can start executing the program from `_main`. In this case, after loading the program, the CR Debugger displays the code of the main file (in the above example, `prog.c`) instead of the start-up routine code.

4.0 Loading the Executable into Flash Memory

The Flash Loader is a stand-alone application that runs on a host platform. It uses the JTAG communication channel to communicate with the PC87591x and supports various operations that use the PC87591x on-chip flash, for example, erase, read and write.

The current version of the CR Debugger is tuned for debugging code loaded into SRAM and can not load the integrated program/Tmonlib executable to the internal flash. Instead, the Flash Loader must perform this function.

This section describes how to use the Flash Loader to load the executable into the PC87591x internal flash. For more details about the Flash Loader application, refer to its online help; in addition, refer to the *PC87591 Flash Loader Alpha Version 1.00 Release Letter*.

4.1 PREPARING THE FLASH LOADER

The Flash Loader supports two data formats: binary format and Intel Hex 32 format. The crprom utility, available under the CR environment, is used to convert a CR executable to an Intel Hex format file.

For example, to convert the executable file `cr.x` to an Intel Hex format file (`cr.hex`), run:

```
crprom -i -w1 cr.x -n -o cr.hex
```

To prepare for loading:

- Convert the executable to Intel Hex format (as explained above).
- Verify that the JTAG communication channel is properly connected between the host PC and the PC87591x.
- Verify that the JTAG communication channel is not being used by any other application, particularly the CR Debugger.
- Verify that the PC87591x is in OBD environment by setting the strap pins correctly and cycling the power, if required; on the PC87591x ADB, straps are set using jumpers JP2 and JP3 (see the *PC87591-ADB Reference Manual*).

4.2 LOADING THE EXECUTABLE AND ACTIVATING TMONLIB

After you have finished preparing the Flash Loader, use it to load the executable into the PC87591x internal flash and to activate Tmonlib, as described in Sections 4.2.1 and 4.2.2.

Protection Word

The protection word is stored in address 0xFE in the PC87591x flash Information block (Block 2). The protection word contains information on access rights to the flash and the size of the core and host boot blocks. By default, the protection word is set to 0xFFFF, and the CR16B is kept in reset.

After loading the executable, you must set the protection word; in particular, configure the Core Boot Block bits (bits 0-3 of the protection word) to a value different from 0xF. For more details about the protection word refer to the *PC87591E and PC87591S LPC Mobile Embedded Controller Datasheet*.

Activating Tmonlib

When the protection word is set properly, disconnecting the Flash Loader (by clicking Disconnect) drives the CR16B core out of reset. The program's start-up routine is executed until the point where it enters Tmonlib's loop, i.e., SVC 108. At this stage, Tmonlib is ready to communicate with the CR Debugger (see Section 5.0).

4.2.1 Via the GUI

Step 1

Connect the Flash Loader to the PC87591x Chip: Click Connect.

Step 2

Erase the Entire Flash: In the Erase tab, select Entire Flash and click Start.

Step 3

Load the Executable:

1) In the Load tab, select the Intel Hex format file (use Browse, if required).

2) Under Input Data Format, select Intel Hex 32 and click Start.

Step 4

Activate Tmonlib:

1) In the Direct Write tab, configure the Data Location to Block = 2, Address = FE.

2) In the Word of Data to Write window, specify the protection word's value. For example, set the protection word to 0xFFFC so that the Core Boot Block bits are 1100b (i.e., the Core Boot Block size is 16 Kbytes); the other bits do not change.

Step 5
Disconnect the Flash Loader from the PC87591x Chip: Click Disconnect.

4.2.2 Via the Command Line

For intensive loading, using Flash Loader batch commands is less complicated than using the GUI and is therefore highly recommended. To use batch commands, perform the following:

Step 1
Copy the converted executable file, `out.ihx`, to the working directory.

Step 2
With a binary file editor, create a binary file, `set_pw.bin`, with the required value of the protection word.
For example, for a Core Boot Block size of 16 Kbytes, with all other bits the same as the default, create a file with the following binary contents:

FC FF

Note: In this example, the byte order is reversed (that is, it is not FFFC) because the CompactRISC architecture supports little-endian memory addressing; this means the byte order in the CR16B is from the least significant byte to the most significant byte.

Step 3
In the working directory, create an empty file, `load.txt`, and copy the following lines to it:

```
connect -c jtag
erase
write -b 1 -f out.ihx -intel
write -b 2 -f -a fe set_pw.bin -binary
disc
```

The `load.txt` file is now a Flash Loader batch file that represents steps 1-5 of the GUI load (see Section 4.2.1).

Step 4
Under MS-DOS (or a DOS window), go to <load_dir> and run:

```
<path>\vfl -f load.txt
```

where <path> is the path of the Flash Loader application.

5.0 Debugging Using the CR Debugger

The current version of the CR Debugger is tuned for debugging code loaded into SRAM; when the on-chip flash is the main source of code, there are some limitations. This section explains how to bypass these limitations. For more information about the CR Debugger, refer to its online help; in addition, refer to the *CompactRISC™ Debugger Reference Manual*.

Note that future versions of the CR Debugger may support flash debugging, making major parts of this section no longer relevant.

5.1 LOAD OPERATION

In OBD environment, the debugger does not load an executable file into the flash memory. However, symbolic information and other data about the executable must be loaded.

To bypass this conflict, follow these steps:

1) Load the executable to the internal flash, using the Flash Loader application (as described in Section 4.0).

2) "Load" the executable using the CR Debugger (as described in steps 1-4 below).

The second load is in quotation marks because the debugger does not really load the executable into the flash memory; only the symbolic information and other data about the executable are loaded.

Using the CR debugger in OBD environment also requires setting the Startup flag.

Startup Flag

The Startup flag is required for debugging in the OBD environment because the CR Debugger is tuned for SRAM (and not OBD) debugging. The CR Debugger works as follows:

1) It loads the executable file and puts a software breakpoint at the label `_main`.

2) It runs the program, which halts at the breakpoint (i.e., `_main`). Debugging then proceeds from `_main` (the start-up routine of the program usually does not require debugging).

In OBD environment, however, the debugger can not write to the flash memory; although the debugger thinks it is writing the software breakpoint to the flash, the breakpoint is not written. The program, as a result, does not halt at `_main` and eventually returns an error. Therefore, in OBD environment, before loading the executable, set the Startup flag in the CR Debugger (Execute→Debugmode→Startup); the flag directs the debugger to start debugging from the start-up routine (i.e., not to place a breakpoint at `_main`).

To load the executable using the CR Debugger:

Step 1

1) Integrate the program with Tmonlib, as described in Section 3.0.

2) Use the crprom utility to convert it to Intel Hex format file (`cr.hex`), as described in Section 4.1.

Step 2

Load `cr.hex` to the PC87591x internal flash, as described in Section 4.2 or Section 4.2.2.

Step 3

1) Open the CR Debugger and select the JTAG channel (target board).

2) Under the main menu, select Execute→Debugmode→Startup to mark the startup debug-mode flag.

Step 4

In the CR Debugger open menu, select File→Load_Coff_File and select `cr.x`. Click OK to "load" the executable.

5.2 DEBUGGING THE PROGRAM

After loading the executable to the PC87591x internal flash, you can run it using the CR Debugger's Go button. You can abort the program and rerun it as well. However, in OBD environment, the main source of code is the flash memory, which the debugger can not write to. Thus the code can not be changed and software breakpoints can not be written. In addition, this limits the use of Step and Next commands because the debugger implements them with software breakpoints in most cases.

5.2.1 Using a Hardware Breakpoint and the Step Instruction Command

The easiest way to bypass the software breakpoint issue in OBD environment is to use hardware breakpoints. Unfortunately, the CR-core supports only one hardware breakpoint.

The easiest way to bypass the Step and Next command issue is to use the Step Instruction (stepi) command to execute one assembly command at a time (in the debugger, select Execute→Step instruction).

5.2.2 Using Software Breakpoints in a RAM Segment

A more effective solution is to concentrate debugging within a relatively small segment of code, for example, one function. Isolate the code segment in one file and direct the linker to place the file in the internal RAM. This gives the code segment full debugging capabilities, that is, the debugger can insert software breakpoints, execute all Step and Next commands, etc.

Note that the amount of RAM available for placing a code segment is limited to:

Size of internal RAM (e.g., 4 Kbytes) – Size of data – Size of stacks

To place a segment of code in the internal RAM, first put it in a separate file, for example `seg.c`; then direct the linker to bind the object file (`seg.o`) to the internal RAM used in the directive file. For example:

```
memory {
          ROM: origin=0 length=0xE000
          RAM: origin=0xE800 length=0x1000
       }

sections {
         .seg  ALIGN(2) into(RAM): { seg.o(.text) }
         .text into(ROM): { *(.text) }
       ...
       }
```

5.2.3 Using Fixed Breakpoints in a Flash Segment

To place more than one breakpoint in a flash segment, the above solutions are not sufficient; as a result, using fixed breakpoints might be considered. Unlike normal software breakpoints, which are placed in the code by the debugger, fixed breakpoints are software breakpoints that the programmer places manually in the code; for example, to stop before `j ++`

```
__asm__("excp bpt");
j ++;
```

However, the CR Debugger responds to a fixed breakpoint as though it were a normal software breakpoint[1]; as a result, the program can not proceed past the fixed breakpoint command. To enable the program to continue, you must instruct the CR debugger to increment the program counter. To use fixed breakpoints correctly, follow steps 1-4, below:

Step 1

Define a new global variable and macro in the program:

```
int debug_break = 0;
#define DebugBreak() {debug_break =1;__asm__("excp bpt");}
```

Step 2

Instruct the debugger to increase the program counter register and clear the `debug_break` flag when `debug_break` is set: In the CR Debugger menu, select Break→**Cmds_on_Break, and add these commands:**

```
mo %pc,((debug_break == 1)?(%pc+2):(%pc))
mo debug_break,0
```

Note: It is possible to save the following commands in an INI file, as follows:

```
autocommand -a mo %pc,((debug_break == 1) ? (%pc+2) : (%pc))
autocommand -a mo debug_break,0
```

Then, in the CR Debugger menu, select File→**Command_File**, choose this INI file and click Open.

Step 3

Now you can add fixed-breakpoints to the in-flash code of the program. For example, to stop before `j ++`, type:

```
DebugBreak();
j ++;
```

Step 4

Program Debugging:

1) Compile and run the new code; the program stops at every fixed breakpoint.

2) Click Go; the program proceeds from the next command.

---

1. When the debugger places a software breakpoint in code, it stores the original command, which was replaced by the breakpoint, in its memory. When Go is executed, the debugger halts at the breakpoint, restores the original command and executes it. Since a fixed breakpoint is inserted manually, no “original” command is restored when Go is executed; the debugger halts at the fixed breakpoint command, tries to execute it and is unable to continue.

6.0 Example Directory

The Tmonlib package contains an example directory for this application note. The example files in this directory can be used as a base code to develop your program.

This section lists the contents of the example directory and explains how to use it. For more examples, refer to the *PC87591x Tmonlib Version 3.1.2.3 Release Letter*.

6.1 CONTENTS

The example directory contains three types of files: program files, batch files and scripts.

6.1.1 Program Files

The executable is built of these files.

- `start.s` - start-up routine file.
- `prog.c` - Virtual I/O example program in C.
- `link.def` - linker directive file.
- (`tmonlib.a` - also used in the example but is located in the root directory of the Tmonlib package.)

6.1.2 Batch Files

- `do.bat` - to build the executable file.
- `load.bat` - to load the executable via the Flash Loader.

6.1.3 Scripts

- `load.txt`, `set_pw.bin` - to load an executable via the command line (as described in section 4.2.2).
- `command.ini` - to instruct the CR Debugger to increase the program counter register and clear the `debug_break` flag when `debug_break` is set (as described in section 5.3).

6.2 HOW TO USE

Under CR Environment, follow these steps:

Step 1
Program/Tmonlib integration:
Run `do.bat` to integrate Program/Tmonlib to one executable file, `out.x` (as described in section 3.3), and to convert it to Intel-hex format file, `out.ihx` (as described in section 4.1).

Step 2
Load the executable:
Run `load.bat` to load the Intel-hex format executable, `out.ihx`, into the PC87591x flash memory and to change the protection word of the PC87591x (as described in section 4.2.2).

Step 3
Debug the program**,** as described in Section 5.0.

Step 4
(Optional step) - Place a fixed breakpoint at `_main`:
1) Uncomment the `DebugBreak()` in file: `prog.c`.

2) Rerun `do.bat` (to create a new executable).

3) Rerun `load.bat` (to load the new executable).

4) Follow steps 3-4 of section 5.1 to “load” the program in the CR Debugger.

5) In the CR Debugger, select File→**Command_File**, choose `command.ini` file from the example directory and click Open. This instructs the CR Debugger to increase the program counter register and clear debug_break flag when debug_break is set (as described in section 5.3).

6) In the CR Debugger, click the Go button to run the program to `_main`; the program stops on the `DebugBreak()` after `_main`. Click the Go button again to proceed running the program.

Step 5
Create your own program. Use the files in this example directory as a base code for your own program.

7.0 Code Flow in the OBD Environment

Figure 3 is a graphic representation of the flow of code of the integrated executable:

1) After reset, the code located in memory address 0 is executed.

2) The Start-Up routine is entered. This routine calls TmonLibStart and enters the Tmonlib kernel.

3) On a Go command (from the debugger), Tmonlib runs the program from its entry point (`program_start`). The Start-Up code calls `_main`.

4) On non-endless programs, when `_main` ends, it returns to the `__eop` label, implemented in `start.s`.

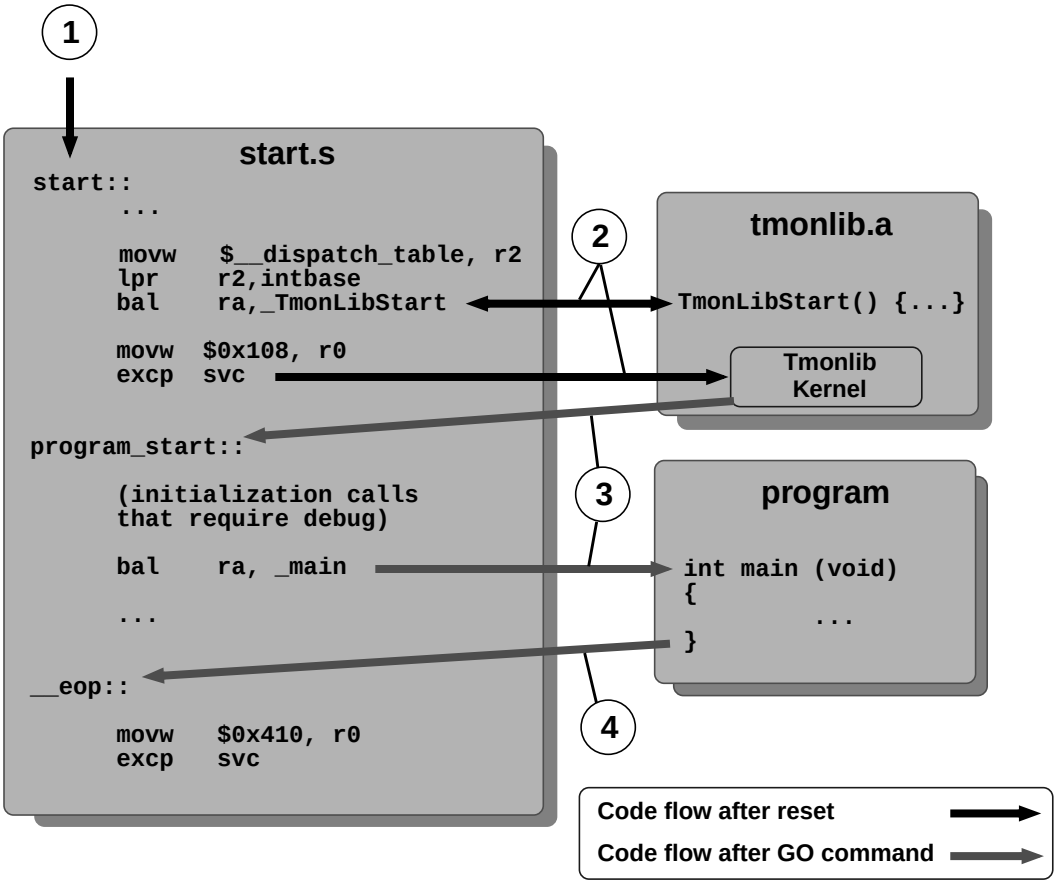

Figure 3. Code Flow in the Integrated Executable Program/Tmonlib

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
Americas
Email: new.feedback@nsc.com

National Semiconductor
Europe
Fax: +49 (0) 180-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 69 9508 6208
English Tel: +44 (0) 870 24 0 2171
Français Tel: +33 (0) 1 41 91 87 90

National Semiconductor
Asia Pacific Customer
Response Group
Tel: 65-2544466
Fax: 65-2504466
Email: ap.support@nsc.com

National Semiconductor
Japan Ltd.
Tel: 81-3-5639-7560
Fax: 81-3-5639-7507
Email: nsj.crc@jksmtp.nsc.com

www.national.com