Horner Electric
“C” Programming Tool Kit
for the HE693CPU100
90-30 Slot One CPU

HE693TKT100
Version 1.03

PRELIMINARY

MAN0157-01

Table of Contents

Software Installation

Step 1: Tasking (BSO) Compiler, Assembler and Utilities Installation

Follow the directions packaged with the kit to install the compiler, assembler, and utilities.

Step 2: Premia Codewright Editor and Development Environment Installation

Follow the directions to install the **16-bit Windows** version of the CodeWrite™ editor. The Tasking environment does not presently function with the 32-bit version of CodeWright™. You should **not** install the evaluation version of Codewright™ since the tool kit provides the full version.

Step 3: Tasking (BSO) Embedded Development Environment Installation

Follow the packaged directions to install the TASKING EDE.

Step 4: Horner Electric Slot One Tool Kit Installation

a) Place the tool kit installation disk in your floppy drive.

b) Using DOS , change to the drive containing the installation floppy (a: or b:)

c) Start the installation program by typing "`install [X:\abc]`" (no brackets)

Where `X:\abc` is an optional drive and directory location for installation of the tool kit.

If no location is specified, the tool kit is installed to `C:\CPU100`.

Manual installation can be accomplished by manually copying the directories from the installation floppy to their desired location.

Note: If this is an upgrade installation, copy any modified files from the tool kit to another location. The installation will overwrite all toolkit files. Next type `install X:\abc -U`

Where `X:\abc` is the required, current path of the tool kit.

The `-U` option installs as an upgrade, the `autoexec.bat` is not modified.

Step 5: Check Your Autoexec.bat

Check that the path in your `autoexec.bat` was modified correctly. The …`\C196\bin` directory should have been added. Some installations of Tasking's software have encountered problems in this step. The Horner tool kit should have added the `pctools` directory of the tool kit to the path and set the `CPU100` environment variable. For example:

```
set cpu100=C:\CPU100
path=%path%;C:\CPU100\pctools
```

Step 6: Reboot your PC

The Tasking compiler and Horner Electric's tool kit installation will modify the path and set some environment variables that are only recognized after a re-boot.

Step 7: Setup Your 90-30 Rack and Slot One CPU Hardware

Follow the GE/Fanuc documentation for connecting the power supply and additional modules to the rack. Consult the HE693CPU100 hardware documentation for additional information on CPU specific installation. Connect the serial cable and RS-232 to RS-485 from the PC RS-232 Com port 1 or 2 to the RS-485 port on the 90-30 power supply.

Step 8: Try Building a Sample Application

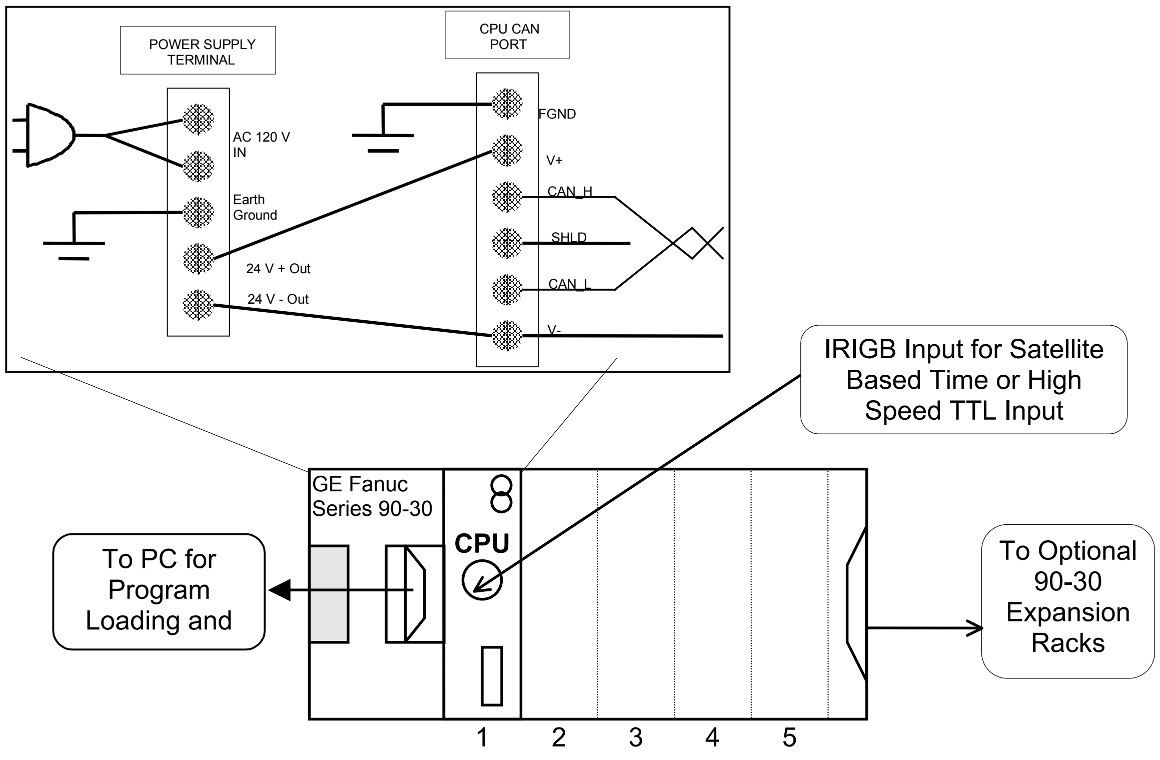
Using a DOS shell, change to the "…`\examples\leds`" directory in the CPU100 tool kit. Type "`mk196`". This will use Tasking's make utility to build the project based on the information in the "`makefile`" and the "`blinky.cmd`" linker command file. The compiler and linker should indicate there were no errors or warnings.

Step 9: Load and Run the Sample Application

The application must be loaded from the PC into the CPU100's EEPROM. The Chipview debugger can load and verify code, but we have included a small DOS utility to make this task faster and easier. From the "…`\examples\leds`" directory type "`rload.exe blinky.hex 1 38400`". This will load the Intel hex file into the CPU100 using PC com port 1 at 38400 bits per second. If you have the communication cable connected to COM 2 change the "1" to a "2".

The rload utility should show the number of bytes successfully written to the CPU100. If any errors occur, check the cable connections and repeat the process. Once the load has successfully completed, the application is started. You should see the LEDs on the CPU100 module blinking in a pattern.

HE693CPU100 Hardware Setup



Note: You will need a RS-485 to RS-232 adapter to connect the CPU100 to your personal computer. Horner Electric's HE693SNP232 is sufficient for program loading, but the HE693RSM232 is needed for debugging or loading with the Chipview® debugger.

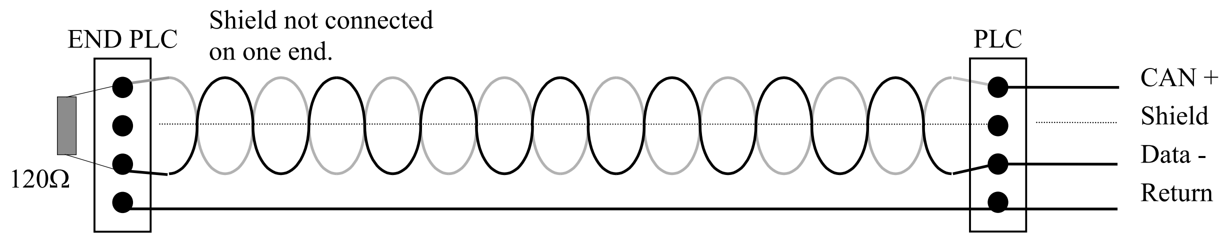
CAN Wiring Rules

1. A CAN network should be wired in a daisy-chained fashion, such that there are exactly two physical endpoints on the network.

2. The two nodes at the physical endpoints, should have 120Ω terminating resistors connected across CAN + and CAN -.

3. The data conductors (CAN+ and CAN-) should be a 24 AWG shielded twisted pair, with 120Ω characteristic impedance.

4. Notice that for a section of cable between two nodes, the cable shield is connected to the terminal at one end of the cable only.



HE693DRT900 Hardware

The HE693DRT900 is an optional piece of the hardware designed as a communication coprocessor for the HE693CPU100. The DRT900 adds one standard RS-232 serial port and a second serial port that can be configured as RS-232 or RS-422. The DRT900 handles all low-level serial communications by providing a 256 byte buffer for each incoming and outgoing stream. While the DRT900 can handle serial stream up to 57,600 baud, continuous incoming data at speeds greater than 38400 can cause lost characters due to 90-30 back plane overhead.

Port A Wiring

Pin	Signal Name	Direction
1	[DCD] Always High	Output
2	[TXD] Transmit Data	Output
3	[RXD] Receive Data	Input
4	No Connection	N/A
5	[GND] Signal Ground	N/A
6	[DSR] Always High	Output
7	[CTS] Clear To Send	Input
8	[RTS] Request To Send	Output
9	[RI] Always High	Output

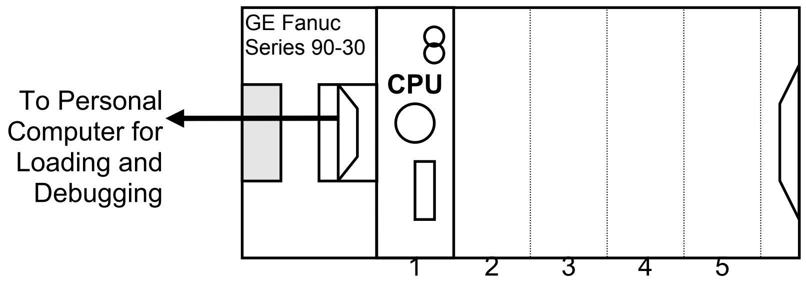
Port B Wiring

Pin	Signal Name	Direction
1	[CTS] Clear To Send (RS-232)	Input
2	[TXD] Transmit Data (RS-232)	Output
3	[RXD] Receive Data (RS-232)	Input
4	[RTS] Request To Send (RS-232)	Output
5	[PWR] 5 VDC Power	N/A
6	[RTS-] Request To Send (RS-485)	Output
7	[GND] Signal and Power Ground	N/A
8	[CTS+] Clear To Send (RS-485)	Input
9	[TERM] Termination (RS-485)	Input
10	[RXD-] Receive Data (RS-485)	Input
11	[RXD+] Receive Data (RS-485)	Input
12	[TXD-] Transmit Data (RS-485)	Output
13	[TXD+] Transmit Data (RS-485)	Output
14	[RTS+] Request To Send (RS-485)	Output
15	[CTS-] Clear To Send (RS-485)	Input

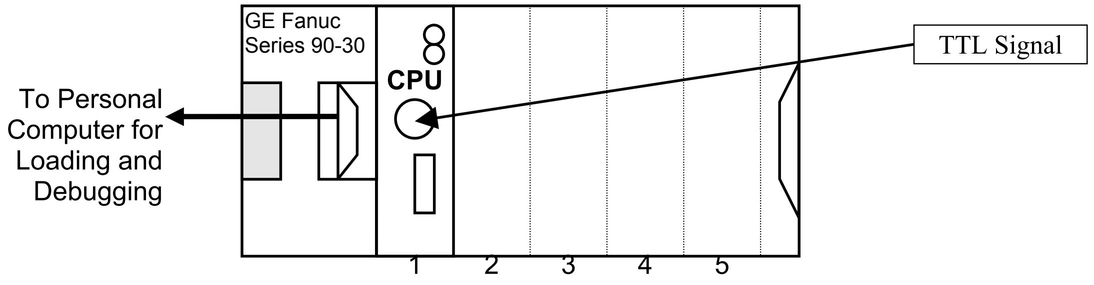
Hardware Setup for Example Programs

Note: The examples that use an ANSI terminal are setup for PORT A, 9600 baud, 8 bits, no parity. These examples show a five slot rack, a ten slot rack may be substituted.

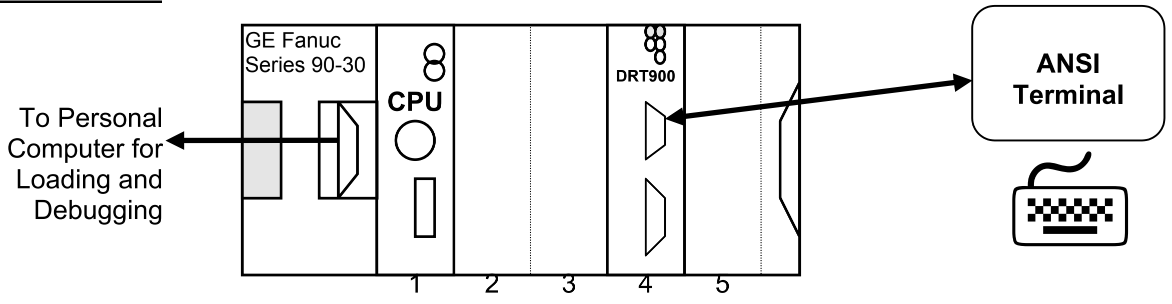
LEDS



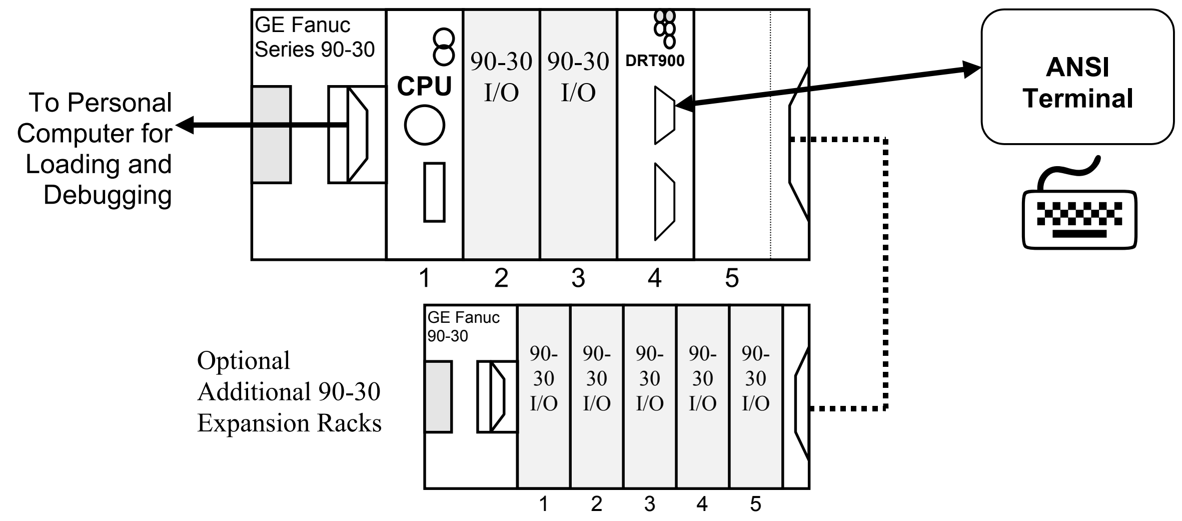
TTL_IO



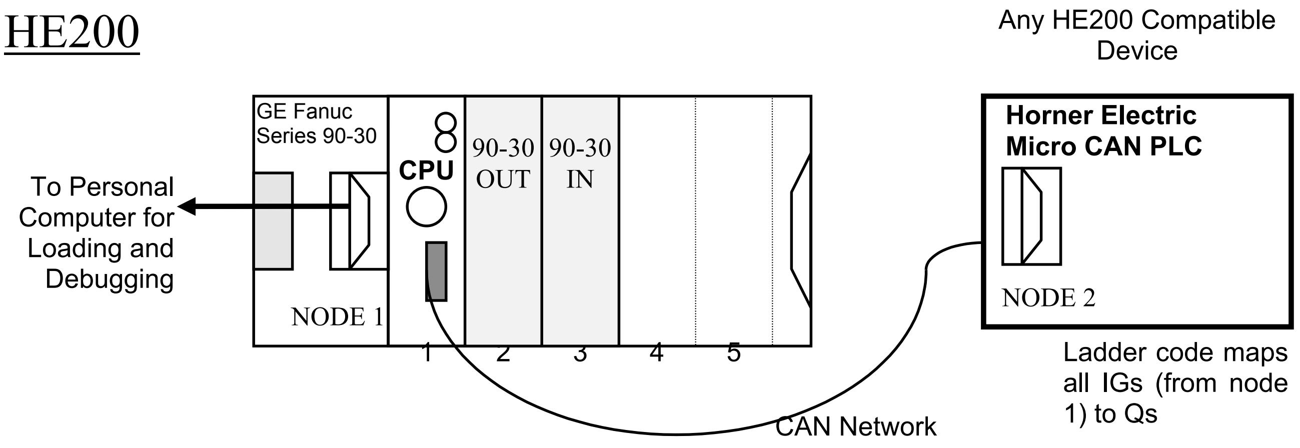
NVRAM



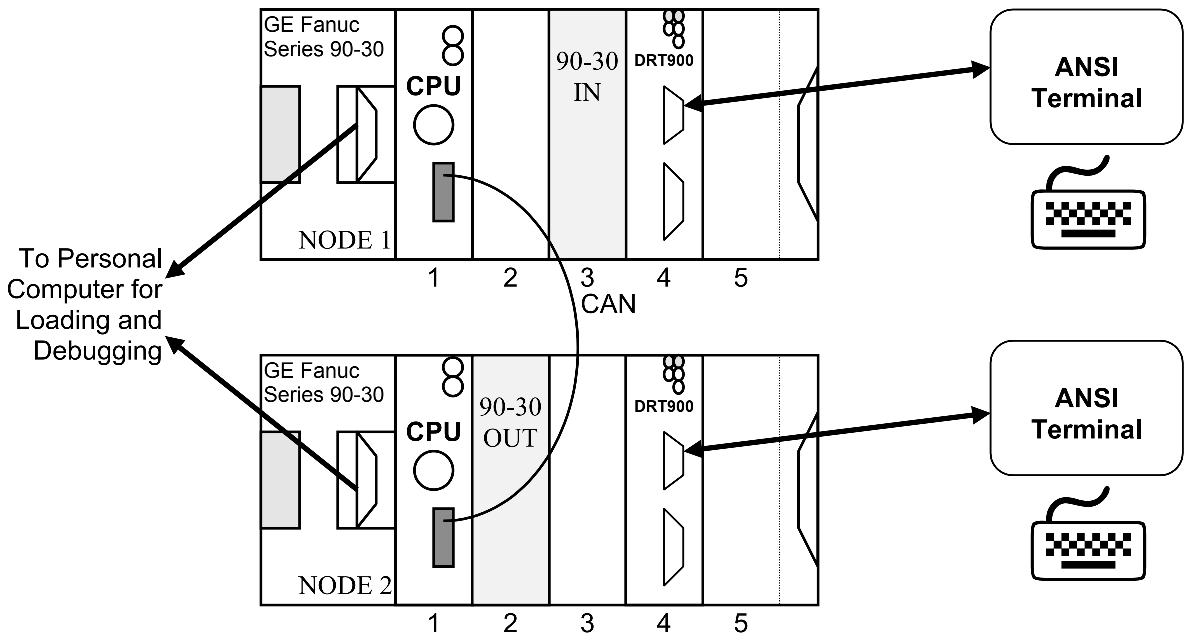
PIF196



HE200



HAL300



REALTIME

(Same as NVRAM Hardware)

SSERIAL

(Same as NVRAM Hardware)

TIMERS

(Same as LEDS Hardware)

Tool Kit Functions

Function Reentrancy

As the BSO compiler documentation states, some of the library functions are not reentrant. For example, the function `memcpy()` is not reentrant in the BSO library. If `memcpy()` were executing in the programs main loop, and a timer callback function starting executing a second copy of `memcpy()`, data would be corrupted. Functions that access hardware usually have the same problem. For example, the real time clock access functions, the PIF 90-30 module access functions, and the serial module access functions should only be called 30from one location: the main loop, timer callback functions, or the HAL network callback functions.

90-30 Module Access Functions

`pif196.h` (See PIF Additional Documentation)

The PIF module access functions were originally written for a PC based “rack controller”. None of these racks contained a CPU and rack number zero did not exist. With the slot one CPU, the CPU resides in rack zero. The module slot labeled “2” is actually accessed as slot 0 in the access functions. Since the CPU uses slot one, the available slots are 0 to 8 or 0 to 3 for a ten slot or five slot rack respectively. Expansion and remote racks are still accessed as slots 0 to 9 or 0 to 4 since these racks do not contain a CPU.

Function	Description
init_pif300	Initializes an PIF196 module and interface hardware
auto_cfg	Scans for attached I/O racks and modules
get_cfg_file*	Gets 12-byte config. file from a smart module
put_cfg_file*	Sends 12-byte config. file to a smart module
put_cfg_par*	Sends 1 config. parameter to a smart module
get_init_file*	Gets n-byte init. file from a smart module
put_init_file*	Sends n-byte init. file to a smart module
read_module*	Reads I/O data from an input module
write_module*	Writes I/O data to an output module

* These functions allow access to smart module configuration and initialization parameters, and are for smart I/O modules only.

In addition to library functions, the tool kit library provides a data base of constant tables. These tables contain information about specific types of I/O racks, I/O modules and module configuration parameters as follows:

Table	Description
rack_info	Contains information about each I/O rack type
module_info	Contains information about each I/O module type
par_info	Contains information about each parameter type

Appendix B for more information on the data base.

The source code file should `#include` the `PIF196.H` header file, to gain access to the library functions and tables.

Several of the library functions require the caller to pass a special formal parameter called `cfg_info`.

This parameter is actually an array of structures which stores I/O configuration information. The information stored in `cfg_info` includes the following:

1. The type(s) of I/O racks connected to the IC693PIF300,
2. The type(s) of I/O modules in each slot of each I/O rack,
3. A 12-byte configuration file for each (smart) module.

The user's application program should declare the `cfg_info` array of structures as follows:

```
struct
      {
      unsigned short  rack_type;          /* Rack type             */
      struct
            {
            unsigned short  board_id;     /* Module type           */
            unsigned short  param[12];    /* Configuration file    */
            } module[10];
      } cfg_info[5];                      /* Rack 0 to 4           */
```

Note that `PIF196.H` defines a special `typedef`, called `CFG_TYPE`, so that `cfg_info` may be declared more easily, as follows:

```
CFG_TYPE cfg_info[5]; /* Rack 0 to 4 */
```

The following C constructs show how information within `cfg_info` is accessed:

```
cfg_info[rack].rack_type
```

Stores a value from 0 to (num_rack_types-1) corresponding to the type of connected I/O rack whose DIP switches are set for the specified rack. Refer to the `PIF196.H` for a list of rack types.

```
cfg_info[rack].module[slot].board_id
```

Stores a value from 0 to (num_ids-1) corresponding to the type of I/O module plugged into the specified slot of the specified rack. Refer to `PIF196.H` for a complete list of board IDs for supported I/O modules. A `board_id` value of 32 or higher indicates that the module is a smart I/O module.

```
cfg_info[rack].module[slot].param[par]
```

Stores one of up to 12 configuration parameters associated with the module plugged into the specified slot of the specified rack. As stated before, only smart modules make use of configuration parameters.

Function Descriptions

init_pif300()

```
Function Call:     status = init_pif300 (port_addr);
Input(s):          port_addr
                     base address must be 0xFFF8
Output(s):         status
                     0 if PIF196 initialization went OK;
                     1 if PIF196 not found;
                     2 if function argument is bad
Description:       This routine initializes the PIF196 software module and the
90-30 backplane interface hardware.
```

auto_cfg()

```
Function Call:  status = auto_cfg (cfg_info);
Input(s):       none
Output(s):      status
                    0 if auto configuration went OK;
                    1 if PIF196 not initialized;
                    3 if parity error occurred

                cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration
```

Description: This routine scans all racks/slots on the 90-30 rack and expansion bus and loads the cfg_info array of structures with rack type, module type and configuration parameter information for all connected hardware.

***** C A U T I O N *****
The auto_cfg() function should only be called when the RUN output is OFF. Otherwise, it may write random data to output modules.

get_cfg_file()

```
Call:           status = get_cfg_file (cfg_info, rack, slot);
Input(s):       cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4 for rack whose configuration will be read

                slot
                    0 to 9 for slot whose configuration will be read
Output(s):      LSB of status
                    0 if configuration file read went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4

                cfg_info[rack].module[slot].param
                    12-word array loaded with 11-byte configura-
                      tion file  and  1-byte  initialization  file
                      size received from the module
```

Description: This routine reads the 11-byte configuration file and the 1-byte initialization file size from a smart I/O module. A smart I/O module is one whose board ID is 32 or higher.

The 11-byte configuration file is loaded into the 1st 11 words of the module's param array. The 1-byte initialization file size is loaded into the 12th word of the module's param array.

put_cfg_file()

```
Call:           status = put_cfg_file (cfg_info, rack, slot);
Input(s):       cfg_info
```

```
                    Array of structures containing rack and slot
                      information for the present configuration
                rack
                    0 to 4 for rack whose configuration will be written
                slot
                    0 to 9 for slot to be configured
                cfg_info[rack].module[slot].param
                    12-word array whose first 11 words are to be
                      sent to the module's 11-byte configuration file
```
Output(s):
```
                LSB of status
                    0 if configuration file write went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred;

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4
```
Description: This routine sends a 11-byte configuration file to a smart I/O module. A smart I/O module is one whose board ID is 32 or higher.

The put_cfg_file() function can NOT be used to configure a smart I/O module which stores its parameters in its own NVRAM, such as the ASCII-Basic Module (HE693ASCxxx). Use put_cfg_par() to configure this type of module.

put_cfg_par()

Call: status = put_cfg_par (cfg_info, rack, slot, par);

Input(s):
```
                cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4 for rack whose configuration parameter
                      will be updated

                slot
                    0 to 9 for slot to write parameter to

                par
                    0 to 10 for parameter to be configured

                cfg_info[rack].module[slot].param[par]
                    Contains configuration parameter to be sent to
                      the module
```
Output(s):
```
                LSB of status
                    0 if configuration parameter write went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred;

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4
```
Description: This routine sends a single configuration parameter to a smart I/O module. A smart I/O module is one whose board ID is 32 or higher.

Note that calling the put_cfg_par() function, is the ONLY way to modify a configuration parameter, for a smart I/O module which stores its parameters in its own NVRAM.

Of the I/O modules supported in the current software release, only the following modules fall into this category:

HE693ASCxxx	ASCII-Basic Module
HE693NET485	PID network module
HE693PIDNET	PID network module

get_init_file()

```
Call:           status = get_init_file (cfg_info, rack, slot, buf);
Input(s):       cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4 for rack whose initialization file will
                      be read
                slot
                    0 to 9 for slot whose initialization file will
                      be read
Output(s):      LSB of status
                    0 if initialization file read went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4

                Array pointed to by buf
                    n-byte array  loaded with  initialization file
                      received from the module
```

Description: This routine reads the n-byte initialization file from a GE smart I/O module. A GE smart I/O module is one whose part number begins with IC693.

```
put_init_file()
Call:           status = put_init_file (cfg_info, rack, slot, buf);
Input(s):       cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4 for rack whose initialization file will be written

                slot
                    0 to 9 for slot whose initialization file will be written

                Array pointed to by buf
                    n-byte array containing initialization file to
                      be sent to the module
Output(s):      LSB of status
                    0 if initialization file write went OK;
                    1 if IC693PIF300 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred;

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4
Description:    This routine sends an n-byte initialization file to a GE smart
I/O module.   A GE smart I/O module is one whose board ID is 32 or higher and
whose model number begins with IC693.
```

```
read_module()
Function Call:  status =
                    read_module (cfg_info, rack, slot, buf);
Input(s):       cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4 for rack whose configuration will be read

                slot
                    0 to 9 for slot to be read
Output(s):      LSB of status
                    0 if read went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred

                MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4

                Array pointed to by buf
                    Loaded with 1 to N words of data depending on
                      the the module type
Description:    This routine reads all data from an input module.
```

The number of %I and %AI data words read from the module depends on the module's board ID, and can be calculated as follows:

```
i_ai_words_read = ((module_info[board_id].num_i + 8) / 16) +
                    module_info[board_id].num_ai;
```

If module_info[board_id].regio is TRUE, the module supports register I/O and the module's 1st configuration parameter contains the number of register (%R) input bytes. The total number of data words read from the module is then as follows:

```
total_words_read = i_ai_words_read +
                   cfg_info[rack].module[slot].params[0] / 2;
```

If a module has more than one type of input (I, AI and/or RI), they will be returned in the buffer in the following order: I words, AI words, RI words.

write_module()

```
Call:           status =
                    write_module (cfg_info, rack, slot, buf);
Input(s):       cfg_info
                    Array of structures containing rack and slot
                      information for the present configuration

                rack
                    0 to 4  for rack  whose configuration  will be read

                slot
                    0 to 9 for slot to be written

                Array pointed to by buf
                    Contains 1 to N words of data depending on the
                      the module type
Output(s):      LSB of status
                    0 if write went OK;
                    1 if PIF196 not initialized;
                    2 if function argument is bad;
                    3 if parity error occurred;
                    4 if board ID error occurred;
                    5 if RUN output is disabled

                 MSB of status
                    0 if status LSB isn't 4;
                    Module's current board ID if status LSB is 4
```

Description: This routine writes all data to an output module.

The number of %Q and %AQ data words written to the module depends on the module's board ID and can be calculated as follows:

```
q_aq_words_written = ((module_info[board_id].num_q + 8) / 16) +
                       module_info[board_id].num_aq;
```

If module_info[board_id].regio is TRUE, the module supports register I/O and the module's 2nd configuration parameter contains the number of register(%R) output bytes. The total number of data words written to the module is then as follows:

```
total_words_written = q_aq_words_written +
                      cfg_info[rack].module[slot].params[1] / 2;
```

If a module has more than one type of output (Q, AQ and/or RQ), they should be placed in the buffer in the following order: Q words, AQ words, RQ words.

PIF196 Table Descriptions

```
Table Name:     rack_info
Table Access:   rack_info[rack_type].str
                    14-character rack descriptor string

                rack_info[rack_type].num_slots
                    Number of I/O module slots in the rack

                rack_info[rack_type].speed
                    LO_SPEED (0) if rack communicates at 250 KHz;
                    HI_SPEED (1) if rack communicates at 1 MHz
```
Where: rack_type
    Rack type from 0 to (num_rack_types - 1)

Description: The rack_info constant array of structures provides information about each type of I/O rack.

Table Name: **module_info**

```
Table Access:   module_info[board_id].str
                    14-character null-terminated module descriptor
                      string

                module_info[board_id].num_i
                    Number of digital input bits

                module_info[board_id].num_ai
                    Number of analog input words

                module_info[board_id].num_q
                    Number of digital output bits

                module_info[board_id].num_aq
                    Number of analog output words

                module_info[board_id].num_par
```

```
                    Number of configuration parameters

                module_info[board_id].regio
                    TRUE if module supports register (%R) I/O

                module_info[board_id].nvram
                    TRUE if module keeps parameters in its own NVRAM

                module_info[board_id].par_type
                    Index of module's 1st parameter in par_info
```

Where: board_id
    Module type from 0 to (num_ids - 1)

Description: The module_info constant array of structures provides information for each supported module type.

Note that a board ID of 32 or higher indicates a smart I/O module. Only a smart module can have configuration parameters.

For smart modules with configuration parameters, module_info tells how many parameters there are (num_par), whether or not they support register I/O (regio), where they are saved (nvram), and where to find more informatation about the parameters in par_info (par_type).

If module_info[board_id].regio is TRUE, the first two configuration parameters contain the number of %R register input and output bytes respectively.

Table Name: **par_info**

Table Access:

```
                par_info[par_type].str
                    14-character   null-terminated  parameter  de-
                      scriptor string

                par_info[par_type].deflt
                    Parameter's default numeric value

                par_info[par_type].lolim
                    Parameter's minimum numeric value

                par_info[par_type].hilim
                    Parameter's maximum numeric value

                par_info[par_type].str_index
                    0xffff if parameter is not enumerated;
                    Otherwise, indexes 1st string of enumerated
                      parameter's value string list in str_info
```

Where: par_type
    Parameter type from 0 to (num_par_types - 1) obtained from module_info[board_id].par_type

Description: The par_info constant array of structures provides information for each configuration parameter.

Note that some parameters are enumerated, and some are not.

All parameters have a numeric value from lolim to hilim inclusive.

An enumerated parameter always has a lolim of 0, and also has a unique character string, which corresponds to each possible numeric value.

Note that par_info[par_type].str_index indexes the string in str_info, which corresponds to the parameter's numeric value of zero.

LED Control Functions

`pif196.h`

These functions allow access to the various LEDs on the 90-30 power supply and the CPU module. Please note that when `run_off()` is called the outputs can not be changed until `run_on()` is called. If the HAL CAN protocol is used, the `hal()` function sets the Network Status LED to flash green/red when receiving/sending data, turn red when no power is applied to the CAN port, and flash red when there are excessive network errors.

`WORD run_on(void)`

This function turns on the green RUN LED on the 90-30 power supply and sends a signal to the modules that the CPU is in RUN mode. This function should return RUNNING unless there is a hardware fault.

`WORD run_off(void)`

This function turns off the green RUN LED on the 90-30 power supply and removes the signal to the modules that the CPU is running. Output modules will not be updated when run is turned off. The DRT900 serial module will continue to function when run is off. This function should return RUNNING unless there is a hardware fault.

`WORD run_check(void)`

This function checks the running status (RUN indicator to the modules) and returns the defined value of `RUNNING` or `NOT_RUNNING`.

`void ok_led(int stat)`

This function sets or clears the green OK LED on 90-30 power supply. The LED is turned on if `stat` is set to TRUE (non-zero), and is turned off if `stat` is set to FALSE (zero).

`void batt_led(int stat)`

This function sets or clears the red BATT LED on 90-30 power supply. The LED is turned on if `stat` is set to TRUE (non-zero), and is turned off if `stat` is set to FALSE (zero). The battery in the 90-30 power supply is not used for this CPU, and the state of the real time clock battery is not determinably by the CPU. Therefore, this LED can be used as needed by the user.

`leds.h`

`void LedInit(void)`

This function sets up a timer for LED blinking and performs a LED test. This LED test conforms to the initialization standards setup for DeviceNet devices. Note: this function should be called after `TimerInit()` since is uses one timer for blinking the LEDs as needed.

`void SetLed(WORD led_command)`

This function allows the NS (Network Status) and MS (Module Status) LEDs on the CPU module to be changed. Each LED has a red and green element that can be illuminated. The possible values are defined in the `leds.h` header file.

Virtual Timer Functions

`timers.h`

These function provide access to virtual timers that are based on one of the CPU's hardware timers (Timer 2 and EPA channel 0). These functions provide a very flexible means for timing events that are often very important in control applications.

TimerInit()

```
Call:             int TimerInit(int timebase)
Input(s):         timebase    - The desired timer resolution (in uS
                  assuming 16 MHz crystal, must be >= 100);
Return value:     the timebase (may have been set previously)
Description:      This routine sets the timer2 prescaler and
                  epa0 reload value for the desired timebase.
                  If the timebase was set previously, the original
                  timebase is unaffected, but it is returned.
                  The main program must call enable() to globally
                  enable interrupts after calling TimerInit().
```

get_timebase()

```
Call:             int get_timebase(void)
Input(s):         none
Return value:     current timer timebase in microseconds
Description:      This function allows a module to determine the current timebase at
                  any time.
```

AllocateTimer()

```
Call:             int AllocateTimer(int config, TIMER_TYPE compare,
                  (void *) func)
Input(s):         config - a bit-mapped value defined as follows:
                  Bit 0 = ignored         (set to 0)
                  Bit 1 = run             (0=alloc only, 1=alloc and start)
                  Bit 2 = dir             (0=down, 1=up)
                  Bit 3 = auto restart    (restart timer @ compare)
                  Bit 4 = func trigger    (call func @ compare)
                  Bits 5-15 = reserved    (set to 0)
                  compare -   If configured for auto restart or func trigger,
                  this is the timer compare value for up counting timers, or
                  it is the reload value for down counting timers.
                  func -      A pointer to a user function to be called at
                  timer expiration (if func trigger is enabled).
Output(s):        0 to (NUM_TIMERS-1) - Allocated timer FAILURE (-1) if no
                  timers available, or bad parameter.
Description:      This function scans the TimerTab to find the first available
unallocated timer, allocates it and returns the timer number.  If no timers are
available, FAILURE (-1) is returned.
```

ReleaseTimer()

Call: int ReleaseTimer(int timer_num)
Input(s): The timer number to return to the timer pool
Output(s): 0 for success FAILURE (-1) if specified timer was not allocated
Description: This function returns the specified timer_num to the timer pool.

StartTimer()

Call: TIMER_TYPE StartTimer(int timer_num)
Input(s): 0 to (NUM_TIMERS-1), the desired timer to start (this is the value returned by GetTimer()).
Output(s): The specified timer's current value. FAILURE (-1) any of the following conditions are TRUE:
The specified timer is invalid (>= NUM_TIMERS)
The specified timer is not allocated
Description: This function starts a halted timer. If the timer is invalid or unallocated, FAILURE (-1) is returned. Otherwise, the timer's current value is returned.

StopTimer()

Call: TIMER_TYPE StopTimer(int timer_num)
Input(s): 0 to (NUM_TIMERS-1), the desired timer to stop (this is the value returned by GetTimer()).
Output(s): The specified timer's current value. FAILURE (-1) any of the following conditions are TRUE:
The specified timer is invalid (>= NUM_TIMERS)
The specified timer is not allocated
Description: This function stops a running timer. If the timer is invalid or unallocated, FAILURE (-1) is returned. Otherwise, the timer's current value is returned.

SetTimer()

Call: int SetTimer(int timer_num, TIMER_TYPE timer_val)
Input(s): timer_num - the timer to be manipulated timer_val- the value to set timer_num to
Output(s): 0 = success FAILURE (-1) if the specified timer_num is unallocated
Description: This function simply writes the timer_val to the specified timer_num. Timer operation is unaffected.

GetTimer()

Call: TIMER_TYPE GetTimer(int timer_num)
Input(s): timer_num - the timer to be read
Output(s): 0 = success FAILURE (-1) if the specified timer_num is unallocated
Description: This function simply reads the specified timer and returns it's current value.

Serial Module Access Functions

`sserial.h`

These functions allow accessing the DRT900 serial module to send or receive serial data. When putch() or sputchar() is called the character to be transmitted is sent across the 90-30 backplane to the DRT900 module where it is placed in a 256 byte FIFO (First In First Out) buffer. The processor on the DRT900 then transmits the character out the desired serial port. When a character is received by the DRT900, it places it in another 256 byte FIFO buffer. When getch() or sgetchar() is called it request the byte across the 90-30 backplane and the function is able to return the character.

Note: After calling `init_pif300()` and **before** calling `autocfg()` the variable `sserial_slot` needs to be set with the logical slot number where the DRT900 serial module is placed. The logical slot is the number as labeled minus 2. For example, if the DRT900 module were placed in the slot labeled “9”, the following would be used to initialize it:

```
        init_pif300(0xfff8);
        sserial_slot = 7;       /* slot 9 - 2 = 7 */
```

`void com(int port)`

This function selects the serial communication port to direct all initializations, character input and output. Setting `port` to 2 directs `stdout`, `stdin` and `stderr` to the DRT900 serial module’s port A. While setting the `port` to 3 directs `stdout`, `stdin`, and `stderr` to port B.

`int set_com_led(int value)`

This functional allows the RUN LED on the DRT900 serial module to be turned on or off. When value is zero (`FALSE`) the LED is turned off. When value is non-zero (`TRUE`) the LED is turned on. This function is not affected by the `com()` function.

`int initcom(int slot, int baud, int param)`

This function initializes the serial communication port that was last selected by the `com()` function. The slot number should be the rack's label for the slot where the DRT900 serial module is placed. The baud rates are defined in sserial.h in a range from `BAUD_600` to `BAUD_57600`. The `param` variable should be set using one or more of the parameters ORed together found in `sserial.h`. Com 3 or port B can be setup as a RS-485 port by passing the `RS485` parameter. Without this parameter this port functions as an additional RS-232 port.

<u>Baud Rate Parameters Define in `sserial.h`</u>

BAUD_600
BAUD_1200
BAUD_2400
BAUD_4800
BAUD_9600
BAUD_19200
BAUD_38400
BAUD_57600

<u>Communication Parameters Define in `sserial.h`</u>

ONE_STOP	Sets the serial communication port to use 1 stop bit.
TWO_STOP	Sets the serial communication port to use 2 stop bit.
DATA_7	Sets the serial communication port to use 7 data bits.
DATA_8	Sets the serial communication port to use 8 data bits.
NONE	Sets the serial communication port to use no parity bits.
ODD	Sets the serial communication port to use odd parity bits.
EVEN	Sets the serial communication port to use even parity bits.
HS_NONE	Sets the serial communication port to use no handshaking.
HS_SW	Sets the serial communication port to use software handshaking.
HS_HW	Sets the serial communication port to use hardware handshaking.
HS_MD	Sets the serial communication port to use multidrop handshaking. (For RS-485 port B only)
TX_OFF	Turns the serial transmitter OFF.
TX_ON	Turns the serial transmitter ON.
TX_CTS	Turns the serial transmitter ON based on the CTS signal.
TX_RTS	Turns the serial transmitter ON based on the RTS signal.
RS485	Sets the port for RS485. (Port B only)

`int scheckchar(void)`

This function returns the number of incoming bytes waiting in the receive buffer for the port set by the `com()` function.

`int sgetchar(void) - getch()`

This function returns one character from the receive buffer selected by the `com()` function. Calling `getch()` or any ANSI calls that reads `stdin` will call this function. Note: getch() waits for a character by calling `scheckchar()` until a character is received then calls `sgetchar()`.

`int sputchar(void) - putch()`

This function places one character in the transmit buffer. This character will be sent to the port selected by the `com()` function. This function returns zero if no error occurred. Calling `putch()` or any other ANSI calls that write to `stdout` or `stderr` will call this function.

Real Time Clock Access Functions

`realtime.h`

The Slot One CPU contains a battery backed real time clock. In addition to providing time of day and date, the real time clock provides the number of seconds since power was applied to the CPU. This data is requested with the `uptime()` function. Time is requested using the ANSI standard `time()` function and is set using the `stime()` function.

`time_t time(time_t *timer)`

This function is the harware specific version of the ANSI `time()` function. The current time in seconds since 00:00:00 January 1, 1970 is returned and is stored in the location pointed to by `timer`. This real time clock is set using the `stime()` function. Note: This function is defined in the ANSI header `time.h`.

`time_t uptime(void)`

This function returns the number of seconds since power was applied to the CPU module. Brief interruptions in power may or may not reset this clock.

`int stime(time_t new_time)`

This function sets the real time clock. The value passed in the `new_time` variable should contain the number of seconds since 00:00:00 January 1, 1970. This function returns zero on success. This real time clock is accessed using the `time()` function.

HAL300 CAN Application Layer Functions

`hal300.h`

This CAN application layer is provided to allow communication and data sharing between many CPU modules. This network protocol can be used for time and data synchronization, data and resource sharing, or information collection. This protocol allows two types of communications: fast, efficient broadcasts of small amounts of data, or a complex (and less efficient) command and response message packets.

The fast message type allows 8 different types (0, 1, 2, 3, 4, 5, 6, 7) of 8 byte packets to be broadcast, received, or requested. The message types are defined by the user's application. This message type is not only simple but has no overhead except that required of all CAN messages.

The "command" message type allows 256 different commands to be transmitted with theoretically 65,536 bytes of data. Each command can have a response. This command response arrangement is only suggested and the function numbers and data are defined by the user's application. A command can be sent and not receive a response, and a response can be sent without receiving a command first. This aspect of the protocol is up to the application designer.

As an example: When `HalPutCommand()` is called, a CAN message is generated and is broadcast onto the network. The node that was intended to receive the message decodes the message and calls the user written function `HalGotCommand()`. This function is passed the sending node's ID number, the function number, the number of data bytes, and a pointer to the data. The user must decide how to handle this incoming data in the `HalGotCommand()`.

`Note:` The node ID zero (0) is reserved for global request and transmissions. Fast data request and commands can be sent to all nodes on a network by setting the `node_id` to zero.

`void  HalInit (BYTE node_id, BYTE baud_rate)`

This function initializes the CAN hardware and the application layer software (hal300). The `node_id` should be a value from 1 to 30 and should be unique to the network. Values for the baud rate are defined in `Hal.h` and are as follows: CAN_BAUD_125K, CAN_BAUD_250K `or` CAN_BAUD_500K.

`void  Hal (void)`

This function monitors for loss of power and updates the status of the LEDs on the CPU module. This function should be called in the application's main loop.

`BOOL  HalPutCommand (BYTE node_id, BYTE function, WORD count, BYTE *data)`

This function uses the smart protocol to transmit user data to another node. A function number is also attached and can be assigned as needed in the application. The value of node_id is the node that is to receive the command.

`BOOL  HalPutResponse (BYTE node_id, BYTE function, WORD count, BYTE *data)`

This function uses the smart protocol to send a response to a command. The function number was designed to equal that of the original command. The function does not have to be called after a command is received. The command / response organization is suggested, but is not required for the application layer to work.

`BOOL  HalPutDataReq (BYTE node_id, BYTE type)`

This functions uses the fast protocol to request data from the node specified by `node_id`. This function can request input or output data based on the value passed to `type`. After the request the other nodes will broadcast up to 8 bytes to ALL other nodes on the network.

`BOOL  HalPutData (BYTE type, BYTE count, BYTE *data)`

This function uses the fast protocol to broadcast up to 8 bytes of data to the other nodes on the network. The sate of inputs or outputs can broadcast based on the value of `type`. This function does not have to be called in response to a request. Data can be sent on a time or change of state basis as needed by the application.

Functions Contained in the User Application Program (User written function called by HAL300.c)

`Note:` These user written function are called from within the CAN receive interrupt. These functions should be as fast as possible to prevent missing new incoming CAN messages of the same type. Because these function do occur asynchronously to other processes, care should be taken when accessing hardware from within these function.

For example, if you main loop is reading a 90-30 module, `HalGotCommand()` should not print a character to the serial module. If your main loop were in the middle of reading the 90-30 module when a CAN message was received, you would attempt to transmit data to the serial module, possibly corrupting the data that was being read in the main loop.

As a general rule do not read or write 90-30 modules, print or request serial data, send CAN data, or set or read the real time clock from these CAN function or timer callback functions.

`void HalGotCommand (BYTE node_id, BYTE function, WORD count, BYTE *data)`

This function is call when a command is received from another node on the network. The sender's node is passed in the `node_id` variable. The function number and data are defined by the application.

`void HalGotResponse (BYTE node_id, BYTE function, WORD count, BYTE *data)`

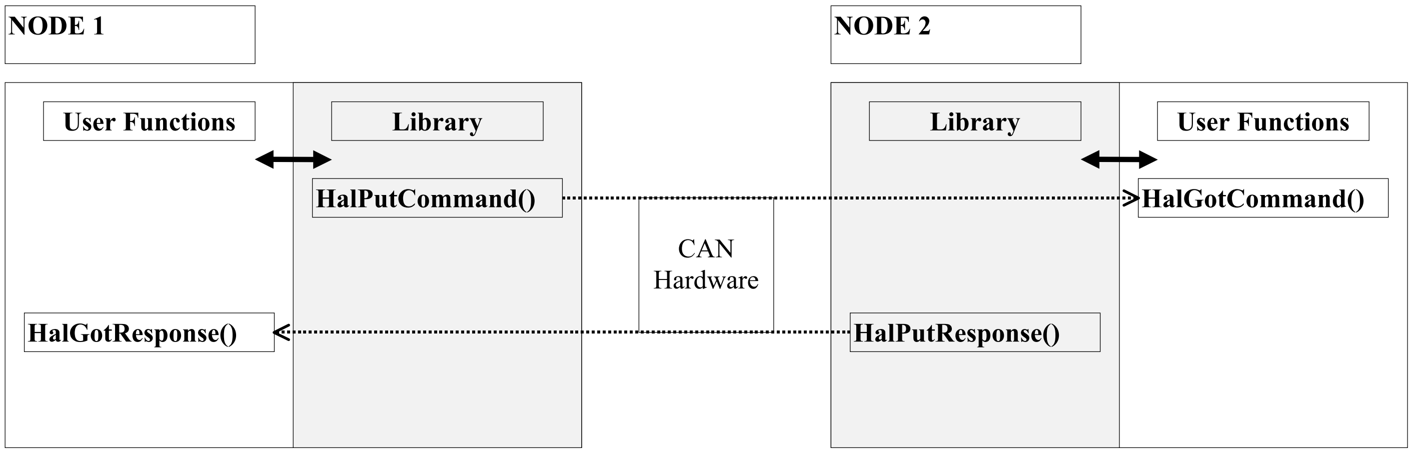
This function is called when a response is received from another node. The sender's node is passed in the `node_id` variable. The function number and data are defined by the application.

`void HalGotDataReq (BYTE type)`

This function is called when fast data is requested by another node on the network. The data requested can be input or output depending on the `type` variable defined in `hal.h.` This request was designed to be followed by a `HalPutData()` function call to broadcast the requested data to any node on the network that requires the data.

`void HalGotData (BYTE node_id, BYTE type, BYTE count, BYTE *data)`

This function is called when another node on the network broadcasts it's data using the `HalPutData()` function. The sender's node is passed in the `node_id` variable. The type is defined in `hal.h` as an input or output type, and a low or high section of data.



HE200 CAN Application Layer Functions

he200.h

The HE200 CAN protocol is an efficient, simple protocol used in Horner Electric CAN devices such as the MICRO CAN PLC. Currently HE200 devices only operate at 125K baud to provide the most noise immunity and the greatest network distance. Currently, this protocol is best for sharing small amounts of digital data between a large number of CAN devices (up to 253).

He200Init()

Call: `void He200Init(BYTE node_id, BYTE baud_rate);`
Input(s): node_id - unique node number assigned to the unit
baud_rate - CAN baud rate to use for communication defined in can196.h as CAN_BAUD_125K, CAN_BAUD_250K, or CAN_BAUD_500K
Output(s): None
Description: This function initializes the CAN hardware and the application layer software (HE200). The node_id should be a value from 1 to 253 and should be unique to the network.

network_send()

Call: `void network_send(BYTE id, WORD, data);`
Input(s): id - network id of the sender - this should be the unit's ID, but can be any unused ID on the network
data - 16 bits of data to broadcast to other nodes.
Output(s): None
Description: This function broadcast 16 global data bits to all other nodes on the network using the supplied network ID.

network_request()

Call: `void network_request(BYTE id);`
Input(s): id - network ID of the node to request data from
Output(s): None
Description: This function sends a request for global data to a node on the network.

Global Variables Used in the Application Layer

Variable: WORD network_data[254]

This is an array of global data from all nodes on the network. When global data is received from another node it is immediately placed in this array. The array index is equal to the node number. For example, the global data for node 9 is store in network_data[9].

Variable: BYTE network_requested

This byte is set TRUE when another node request a unit's global data. When this byte is set TRUE, the user's application should send the data using network_send() and then should set network_requested FALSE.

Variable: BYTE can_off_detected

This byte is set TRUE when a CAN bus off condition occurs.

Variable: BYTE connect_id
Variable: BYTE remote_mode

These bytes are used to control the host data communication state.
Note: These are only used by advanced host tools (see below).

Communication State	connect_id	remote_mode
Local	unit’s ID	FALSE
Pass-Thru	not the unit’s ID	FALSE
Remote	not the unit’s ID	TRUE

In local mode, the unit is receiving host commands from a source other than the CAN port (serial port, dual port ram …).
In pass-thru mode, the unit is receiving commands from a source other than the CAN port, but the commands are actually for another unit on the network

Host Tool Command Support Functions

(These functions are for more advanced supervisory tools)

chknet()

Call: BYTE chknet(void);
Input(s): None
Output(s): TRUE if the buffer contains data from another node.
FALSE if the buffer is empty.
Description: This function checks the host data buffer for a empty or non-empty state.

getnet()

Call: BYTE getnet(void);
Input(s): None
Output(s): The next byte in the host data buffer
Description: This function removes and returns one byte from the host data buffer.

putnet()

Call: void putnet(BYTE ch);
Input(s): ch - a single character to send as host data
Output(s): None
Description: This function sends a single byte as a HOST to NODE or NODE to HOST data packet depending on the current state.

putnet_buf()

Call: void putnet(BYTE *buff, BYTE num);
Input(s): buff - pointer to the data to send
num - number of data bytes to send (1 to 7)
Output(s): None
Description: This function sends up to seven bytes as a HOST to NODE or NODE to HOST data packet depending on the current state.

flushnet()

Call: void flushnet(void);
Input(s): None
Output(s): None
Description: This function sets the host data buffer as empty.

EEPROM Write Function

`nvram.h`

`void nv_write(BYTE *dst, BYTE *src, WORD length)`

This function copies a variable number of bytes (determined by `length`) from the source pointer to the destination pointer (pointing to a location in EEPROM). This function only prevents over writing the “ROM” debugger and **will** overwrite user code if improperly used. Note: Writing each byte can take up to 10 milliseconds, and interrupts are disabled during this time. Therefor, writing to EEPROM should happen only during a non-time-critical periods.

TTL I/O Functions

`ttl_io.h`

The high-speed TTL I/O port on the CPU100 was first designed as an input for an IRIGB time signal. The following functions allow this port to be used as a TTL input **or** output port. While the TTL port is protected against noise and over-voltage, this port is not isolated and will be damaged if care is not taken. The port is very limited as an output. The circuitry used to protect the port as an input limits the output current to about 50 micro-Amps sinking and about 300 micro-Amps sourcing.

init_ttl_in()

```
Function Call:  status = init_ttl_in (void *func, int edge);
Input(s):       func
                    User function to call when an edge is detected.
                edge
                    Edge type define in ttl_io.h:
                    NO_CAPTURE = edges are ignored
                    NEG_EDGE =   low to high pulses cause the function to be
                                 called
                    POS_EDGE =   high to low pulses cause the function to be
                                 called
                    BOTH_EDGED = any edge at the input causes the function to
                                 be called.
Output(s):      status
                    0 if init went OK
                    non-zero if an error occurred
Description:    This function initializes the TTL input port to call a user
callback function when the inputs senses a rising and / or falling edge.
```

read_ttl_port()

```
Function Call:  status = read_ttl_port (void);
Input(s):       NONE
Output(s):      status
                    0 if the TTL input is LOW
                    non-zero if the TTL input is HIGH
Description:    This function reads the logic level of the TTL input.
```

init_ttl_out()

```
Function Call:  status = init_ttl_in (void);
Input(s):       NONE
Output(s):      status
                    0 if init went OK
Description:    This function initializes the TTL port as a push/pull output.
```

write_ttl_port()

```
Function Call:  write_ttl_port (BOOL state);
Input(s):       state
                  0 sets the output LOW
                  non-zero sets the output high
Output(s):      NONE
Description:    This function changes the state of the TTL output.
```

Using the Tasking Floating Point Library

To use floating point calculations:

• Note: `fpinit()` does not need to be called if the default (OMF(2)) object files are used.

• The floating point library `fpal96.lib` should be added to your link list (see **Creating Linker Command Files**).

Note: If any of the following functions are used, the `c96fp.lib` library should be added to your link list **before** the `c96.lib`, in addition to the `fpal96.lib`.

```
acos(), cos(), cosh(), asin(), sin(), sinh(), atan(), tan(), tanh(), atan2()
ceil(), floor(), fmod(), exp(), log(), log10(), ldexp(), pow(), sqrt(), fabs()
atof(), strtod(), frexp(), modf()
```

If any of the following print or scan routines are used with %f, %g, or %e floating point formats, the c96fp.lib library should be linked to provide floating point printing and scanning support.

```
printf(), sprintf(), fprintf(), vfprintf(), vprintf(), vsprintf(), scanf(),
sscanf(), fscanf()
```

Creating Custom Makefiles

#	=	marks a comment line
\	=	continues to the next line

file: makefile

```
ABS_dependencies =     \
 makefile              \
 zzzz.cmd              \
 xxx1.obj              \
 xxx2.obj              \
 ..\..\lib\hecpu100.lib
```

This defines the dependencies for the output file. If any of these files change, the project is re-linked, creating a new `zzzz.abs` and `zzzz.hex`.

```
ASM_switches = debug source symbols xref
```

Set the command line switches for the assembler.

```
C_switches   = debug code regconserve model(CA) pl(32767) pw(132)
```

Set the command line switches for the compiler.

```
# Create zzzz.hex from zzzz.abs absolute file.
zzzz.hex: zzzz.abs
  OH196 zzzz.abs to zzzz.hex
```

Creates an Intel hex format version of the linker output if `zzzz.abs` changes.

```
# Create zzzz.ABS from its constituent .OBJ modules.
zzzz.abs: $(ABS_dependencies)
  rl196 & < zzzz.cmd
```

Links the object files defined in the linker command file (`zzzz.cmd`). This link is dependent on the `ABS_dependencies` defined above.

```
# Compile xxx1.c into xxx1.obj using the
# compiler switches defined above
# Compiles if xxx1.c or xxx1.h changes.
xxx1.obj: xxx1.c xxx1.h
  c196 xxx1.c $(C_switches)

# Compile xxx2.c into xxx2.obj using the
# compiler switches defined above
# Compiles if xxx2.c, xxx2.h or xxx1.h changes.
xxx1.obj: xxx2.c xxx2.h xxx1.h
  c196 xxx2.c $(C_switches)
```

These section compile the C source code using the command line switches defined by `C_switches`. The compiler creates `.obj` files. The dependencies must be determined and added as shown.

In the line `xxx1.obj: xxx1.c xxx1.h`
xxx1.obj is the compiler output
xxx1.c and xxx1.h are the defined dependencies.

Creating Custom Linker Command Files

& = continue to next line

File: zzzz.cmd

`xxx1.OBJ,                    &`
`xxx2.OBJ,                    &`
	User object file from compiled or assembled code.

`..\..\lib\HECPU100.LIB(RISM), &`
`..\..\lib\HECPU100.LIB,       &`
	CPU100 library functions including a forced link on the "ROM debugger.

`CA_SFRS.OBJ,                 &`
`KR\C96.LIB                   &`
	Tasking (BSO) ANSI C library functions. If you use floating point match see the Tasking directions for the correct library to add for floating point support.

`TO zzzz.ABS                  &`	Name of the output file.
`MODEL(ca)                    &`	Processor model - DO NOT CHANGE.
`STACKSIZE (+20)              &`	Use the Compilers stack size suggestion, plus 20 extra bytes.

`RAM(30H-43H)                 &`
`RAM(100H-1FFH)               &`
`RAM(200H-3FFH(STACK))        &`
`RAM(400H-4FFH)               &`
`RAM(500H-1EFFH)              &`
`RAM(202FH-202FH)             &`
`RAM(9FF0H-9FFFH)             &`
`RAM(0A000H-0FFFFH)           &`
`ROM(2000H-202EH)             &`
`ROM(2030H-9FEFH)             &`
	RAM and ROM locations for this platform.
	Do not change this.

`PAGEWIDTH(132)               &`
`IXREF`
	Linker options:
	pagewidth(132) = ouput map file in 132 character wide format.
	Ixref = intermodule cross reference in map file.
	
	See Tasking documentation for these and other options.

DOS Based Tools

The PCTOOLS directory in the Horner tool kit contains some DOS based utilities that may be useful for application development. The installation should have added the PCTOOLS directory to your path so these commands should execute from any directory.

RLOAD.EXE

This is a serial hex loader for the HE693CPU100. This allows a user to load an Intel hex formatted file into the CPU100 using the serial port on the power supply. See the RLOAD.EXE documentation in the following pages for more details.

TERM.EXE

This is a simple terminal emulator for DOS. Many of the example programs require an ANSI terminal be connected to the DRT900 serial coprocessor module. This program will allow your PC to function as an ANSI terminal. `TERM.EXE` also has display modes for raw ASCII and hexadecimal. These can be very useful when debugging serial communications. See `term.txt` in the DOCS directory for more detailed information.

CC196.BAT

This is a small DOS batch file that compiles a C source file using a standard set of compiler options. It can be edited as needed for special applications. To use this batch file type:

```
cc196 xyz.c
```

The file should be compiled using Tasking's C compiler and should generate `xyz.obj`.

LINK196.BAT

This batch file and associated linker command file (`rl.cmd`) will link up to ten compiled files using a standard set of libraries and options. The linker command file `rl.cmd` found in the PCTOOLS directory is used to complete the linking. To use this batch file type:

```
link196 xxx.obj yyy.obj zzz.obj
```

This will use Tasking's linker to link the supplied files and the standard libraries (including Horner's C tool kit) and generates `xxx.abs` and `xxx.hex`. If additional libraries are required for linking, they can be passed as a command line option after the object files to link.

HEXCALC.EXE

This small utility **approximately** calculates the number of bytes in EEPROM an Intel HEX file will require. This can be very useful when trying to develop a large program that may be on the verge of not fitting in the supplied EEPROM. To use this utility type:

```
hexcalc xyz.hex
```

This will calculate the number bytes needed to write xyz.hex to EEPROM and print this value to the screen.

EMCPU.BAT

See the following section on Chipview.

Using the Chipview Debugger

Chipview is an interface for a DOS based personal computer to the built-in “ROM” debugger (called RISM for Reduced Instruction Set Monitor) included with the CPU tool kit. This product is produced by Chiptools and is available from Horner Electric for use with the HE693CPU100 slot one CPU. Included in the PCTOOLS directory is a small batch file named `EMCPU.BAT` used to start Chipview for DOS using the correct parameters. For this batch file to work, chipview must be included in your path or the batch file must be edited to included the full path of the Chipview debugger.
`EMCPU.BAT` contains the following:

```
cvm196 -ac196CA -l -zb57600 -zd250 -zi- -zmh+ -zmr+ -d1 -zri-
```

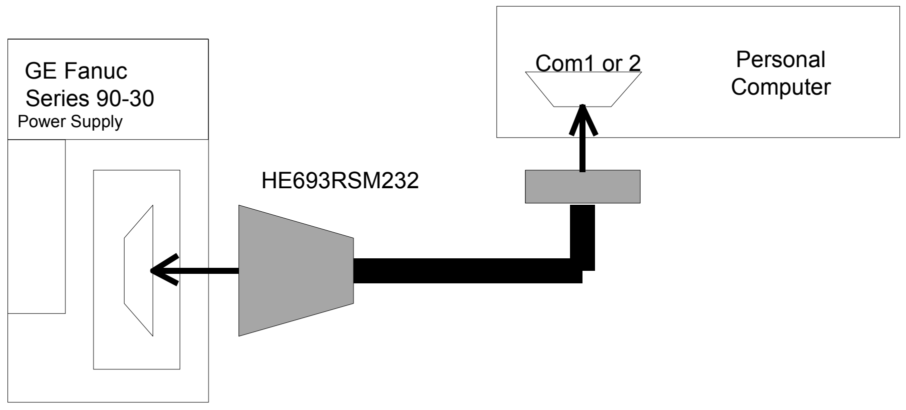
Chipview executable

Serial Communication Speed Default is 57600, reduce if problems occur

When Chipview or another utility sends RISM a “reset monitor” command the monitor clears an EEPROM byte at location 202Fh. When the HE693CPU100 powers-up if byte 202Fh is clear it will wait for debugger commands and will NOT run the user application. This is indicated by the module status and network status LEDs turning orange. Chipview can be used to set byte 202Fh to FFh, this will allow the application to automatically run at the next power-up. The easiest way to set this byte is to select VIEW → DUMP, go to byte 202Fh and type “0FFh”. The RLOAD.EXE utility will automatically set this byte after attempting a download.

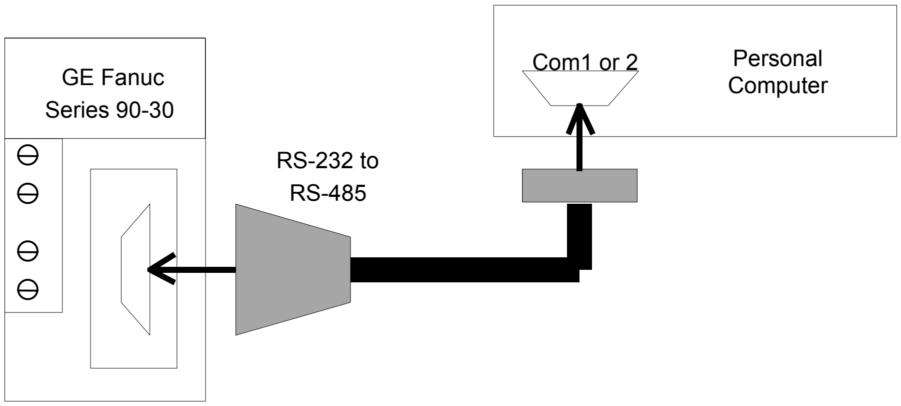
When using Chipview, the HE693RSM232 RS-485 to RS-232 adapter must be used to connect the PCs serial port to the RS-485 port on the power supply of the slot one CPU.

Connect the PC to the HE693CPU100 serial port on the power supply as follows:



RLOAD.EXE Hex File Loader

1. Connect the device that is to be updated to the PC's serial communication port. "Com1" or "Com2" may be used.



2. Type: rload xxxxx.HEX # bbbbb

Where:

xxxxx.HEX is the name of the HEX file containing the firmware update to be loaded. If the file is not in the same directory as RLOAD.EXE the full path should be included. For example:

```
RLOAD C:\UPDATES\HE12345.HEX
```

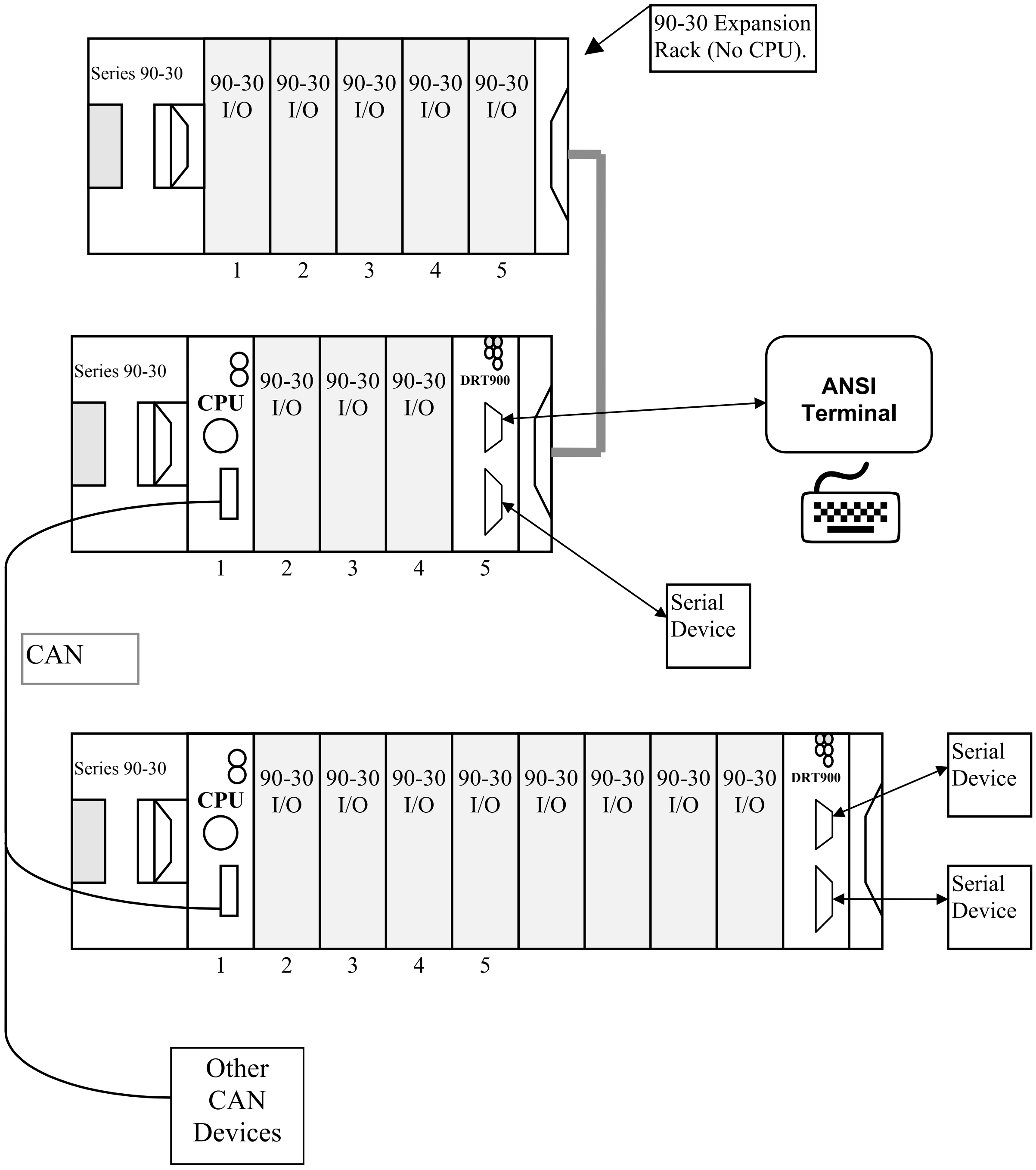
# is the com port that the device is connected. (1 or 2 currently supported)

bbbbb is the baud rate to communicate with the device. The CAN CPU will work up to 38400 baud. This value can be left out and 9600 will be used.

3. The update file will now be read and sent to the attached device. Each byte is verified as it is written. If an error occurs, RLOAD will report where the error occurred and exit. If this occurs, try the load process again from step 2. If RLOAD reports it could not get the device to respond, check the cables, connections, and verify the correct com port is being used and repeat the process from step 2.

Once the loading process starts the number of bytes successfully sent to the device will be displayed on the screen. When the processes has successfully completed, "Done." should appear. The new application will now begin to run.

Sample 196 Slot One CPU Setup



Function Index