Innovasic Semiconductor
fido1100
Application Note 120

Developing Multi-Context Applications on the fido1100 Microcontroller

Version 1.0
December 2006

Table of Contents

Index of Tables

Introduction

Many embedded applications do not require a large number of threads of execution, sufficing with 2 or 3 independent threads, plus some number of interrupt handlers. In such instances, the developer is faced with either paying for an operating system, or writing and maintaining an executive. This paper provides an example of building such an application on the fido1100 using only the inherent hardware threading capabilities of the processor.

Approach

The example application described in this application note performs the following functions:

1. Read I/O from a set of position sensors, compute a solution, and drive commands to actuators. This function requires very strict determinism to provide good control.
2. Process time/safety critical interrupts associated with limit switches, light curtains, etc..
3. Manage data-flow to/from various other modules over 2 channels of UART.
4. Handle data/control information of an Ethernet channel.
5. Perform Continuous Built-In Test and various house-keeping functions.

Step 1: Context Task Allocation

The fido1100 supports up to 5 hardware contexts. Each context includes user registers, program counter, etc., as well as context-specific timers, MPU controls, etc. Contexts are numbered 0 through 4, where context 0 is also known as the master context. The master context (context 0) differs from the others in the following ways:

- Context 0 always has the highest priority – if it is ready to run, it will run.
- Context 0 has write access to a number of configuration registers and fields of registers that can not be written by any other context. Registers of this type, discussed in this example, are: The context fields on interrupt control registers, system timer setup registers (control and prescale), the memory-mapped copies of context-specific registers (e.g. Context 2 program counter), and Context Control registers.
- Context 0 handles certain classes of critical events (double bus fault, double MPU fault, etc.).
- Context 0 does not have a claim register (hardware semaphore).
- The fido1100 TRAPX instruction traps to context 0.

All other contexts are identical to each other except in terms of priority – if two contexts have the same priority and are ready to run, the lower-numbered context will execute. E.g. if contexts 1 and 4 are both programmed to a priority of 3, and both are ready to run, the context 1 will get the processor.

Because context 0 always has the highest priority, and it is the arbiter of last resort (when something goes very wrong in another context, the master context handles it), you should carefully select those elements of your application that are most critical in terms of timing and/or functionality to operate in that context. For the given example task will be divided into contexts using the following scheme:

Time-critical I/O and the safety-related position switch handlers are placed in context 0.
Context 1 will serve as the general interrupt handler for the rest of the I/O (UARTS, Ethernet, etc).
Context 2 will be used to compute outputs for the time-critical control function.
Context 3 will service the Ethernet I/O (contains the stacks/etc).
Context 4 will perform the UART I/O, CBIT and housekeeping.

Step 2: Determine Relative Timing

The next step in developing a multi-context application is to determine the relative timing and priorities of the contexts. Priority is represented by a number 0-7, with 7 being highest.

Function	*Priority*	*Operating Frequency*
Real-time/Safety/Master	Max	120 Hz
I/O interrupt handling	6	N/A
Control Computation	4	120 Hz
Ethernet Stacks	2	50+ Hz
UART I/O, CBIT, etc.	0	20+ Hz.

Table 1: Relative Timing of Tasks

Step 3: System Task Timing

Generally, we want the control processing to operate something like the following:

1. Read Inputs
2. Process inputs and compute outputs
3. Write Outputs

The timing is critical for tasks 1 and 3. The I/O processing must be performed with a minimum of jitter. The only timing requirement for task 2 is that it start after the completion of task 1 and finish before task 3. One approach to accommodating this is to operate the I/O routines in an interrupt handler that operates at twice the I/O frequency. On even occurrences it will process input data, on odd occurrences it will process outputs.

The fido1100 provides a System Timer that provides an interrupt interface to the various contexts. The programmer sets up a base frequency and the timer provides interrupt outputs at the base frequency to system timer interrupt 0, one-half the frequency to system timer interrupt 1, one-fourth the frequency to system timer interrupt 2, one eighth the frequency to system timer interrupt 3, and one-sixteenth the frequency to system timer interrupt 4. Thus, if we set the base frequency to 240 Hz for context 0 we get the following:

System Timer Interrupt	*Frequency*
0	240 Hz
1	120 Hz
2	60 Hz
3	30 Hz
4	15 Hz

Table 2: System Timer Interrupt Frequencies

Now we can extend our table:

Context	*System Timer Interrupt*	*Function*	*Priority*	*Operating Frequency*	*Max Period*
0	0	Real-time/Safety/Master	Max	240 Hz	4.17 msec
1	None	I/O interrupt handling	6	N/A	N/A
2	1	Control Computation	4	120 Hz	8.33 msec
3	2	Ethernet Stacks	2	60 Hz	16.7 msec
4	3	UART I/O, CBIT, etc.	0	30 Hz.	33.3 msec

Table 3: Context and Priority Timing

NOTE: *While the example provided here uses the system timer to generate interrupts for all periodic contexts, any combination of system timer, interrupts, Timer Counter Unit, software interrupts, etc., can be used to time contexts.*

Step 4: Determine Context Modes

Another issue with multiple contexts is to determine which context type should be used for each. Contexts can operate in one of three modes:

- Standard mode: A context in standard mode provides an operating environment like a standard 68K. Programs can be written in user and/or supervisor mode. The context can service interrupts locally (standard operation with stacking of PC and status register, return from interrupt unstacks, etc.). Interrupts can be nested.

- Fast Vectored mode: A context in fast vectored mode serves as an interrupt handler. On receiving an interrupt a vector is fetched and execution begins at the vector. There is no stacking of processor state prior to beginning execution. Also, this mode does not support nesting of interrupts. However, it provides a simpler and faster environment for handling many interrupts.

- Fast Single-Thread mode: A context in fast single-threaded mode can be thought of as a single interrupt handler. On receipt of an interrupt event, the processor simply begins execution at the current program counter. This is typically used for very fast/simple interrupt handlers that must operate at exceptionally high frequencies.

In this example the context modes are define as follows:

Context 0 – Fast Single-Thread Mode

This context will handle two highly deterministic I/O interrupts and some safety-sensitive handlers. In addition, the Master Context must have some degree of handlers for fatal faults in the other contexts (these are faults that would typically lock up a standard microprocessor (double bus fault, etc.). The fido1100 allows the program to detect these conditions in a context other than zero and take some action to correct it).

Because there are multiple events to respond to, context 0 can not operate in Fast Single-Threaded mode. The choice between Fast Vectored and Standard modes depends on the latency requirements for the safety-critical interrupts. Because interrupts can not be nested in a fast mode, it is possible that the processor will have just begun to handle the periodic interrupt when it receives a safety-related interrupt. The safety interrupt will not be serviced until the I/O handler has finished. In standard mode, all interrupts will take a bit longer to execute, but the safety-related interrupts can be given higher priority and nest on top of the I/O interrupts.

For the purposes of this exercise we'll assume that the latency imposed by the I/O routines is not an issue for the safety-related interrupts.

Context 1- Fast Vectored Mode

This context is performing various I/O and interrupt handling (e.g. Ethernet/DMA/UART/...). It is only operating interrupt handlers. It could be implemented as either a Fast Vectored or standard mode context. The advantage to Fast vectored is that it is faster and doesn't need as much stack (because you don't have to worry about nesting of interrupts).

Context 2 – Fast Single-Thread Mode
This context will be performing some processing on the I/O gathered by Context 0. It is the only processing performed in this context. Any of the modes could be applied to this problem. Because it has only one purpose, and there is no particular need to perform the processing in an interrupt context, we'll use Fast Single Thread mode for this context.

Context 3 – Standard
Context 3 is executing Ethernet stacks and related processing. While it could be treated as a single process, the flexibility of a standard context may pay off over the course of development, allowing various interrupts/inter-context communications to be used.

Context 4 - Standard
Context 4 has effectively two 'tasks': Handle UART I/O periodically, and perform various housekeeping chores. One way to approach this is to treat the housekeeping functions as a preemptable background task, with the UART I/O processing in the foreground of the context. A simple way to accomplish this is to use the context in standard mode, with the background processing running in a typical environment, then execute the entire UART I/O process in an interrupt context. This allows the user to have a set of preemptable tasks without the trouble/overhead of a scheduling kernel.

Context	System Timer Interrupt	Function	Priority	Operating Frequency	Max Period	Mode
0	0	Real-time/Safety/Master	Max	240 Hz	4.17 msec	Fast Vectored
1	None	I/O interrupt handling	6	N/A	N/A	Fast Vectored
2	1	Control Computation	4	120 Hz	8.33 msec	Fast Single-Threaded
3	2	Ethernet Stacks	2	60 Hz	16.7 msec	Standard
4	3	UART I/O, CBIT, etc.	0	30 Hz.	33.3 msec	Standard

Table 4: Complete Context Configuration Matrix

Step 5: System Timer Setup

The System Timer Prescale Register defines the base rate for the system timer.

System Timer Prescale Register																															
Reserved																Prescale Value															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

To calculate the value to put in this register:

Prescale = (desired period)*(system clock rate)/(2 * (bit rate from SystemTimerControlRegister)

E.g. for a 66 Mhz clock with a period of 1/240Hz (to interrupt 0):

*Prescale = (1/240 * 66,000,000)/(2 * 4) = 34375 = 0x8647*

Where the bit rate was selected to give a whole-number value for the prescale register.

The System Timer Control Register enables the timer and selects the bit rate used above.

System Timer Control Register															
Reserved											Bit Rate Select				En
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- En : Bit 0
  - 1 System Timer Enabled
  - 0 Not Enabled
- Bit Rate: Bits 4:1
  - 0000 - Bit 0 (Divide by 2)
  - 0001 - Bit 1 (Divide by 4)
  - 0010 - Bit 2 (Divide by 8)
  - 0011 - Bit 3 (Divide by 16)

- 0100 - Bit 4 (Divide by 32)
- 0101 - Bit 5 (Divide by 64)
- 0110 - Bit 6 (Divide by 128)
- 0111 - Bit 7 (Divide by 256)
- 1000 - Bit 8 (Divide by 512)
- 1001 - Bit 9 (Divide by 1024)
- 1010 - Undefined (Divide by 1024)
- 1011 - Undefined (Divide by 1024)
- 1100 - Undefined (Divide by 1024)
- 1101 - Undefined (Divide by 1024)
- 1110 - Undefined (Divide by 1024)
- 1111 - Undefined (Divide by 1024)

To enable the System Timer with a bit rate of Divide by 4 => 0x00000003

System Timer Interrupt Control Register															
Reserved				Clear CT	Mode	Flag	En	Reserved		Context			Priority		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Clear CT
  - Setting to a 1 indicates that the 'interrupt' event generated by the timer will also clear all of the context timers (but not the idle timer).
  - Setting to a 0 indicates that the context timers will not be cleared.
- Mode
  - Setting to a '0' indicates that the 'interrupt' event generated by the timer will work as a standard interrupt request to the assigned context.
  - Setting to a '1' indicates that the 'interrupt' event generated by the timer will simply move the related context from not-ready to ready. If the context is halted or already ready then the timer event has no visible effect. NOTE: Aside from setting the context in motion - there is no need for the handler to 'ack' the cause of context startup.
- Flag -- A 1 indicates that the interrupt occured, cleared on read.
- Enable - A 1 enables generation of this interrupt, 0 disables.
- Context[2:0] - Identifies the context this interrupt is associated with.
- Priority[2:0] - Defines the priority of interrupt.

There are five copies of this register. For our example, we will use four of them,one for each context except the one that is only handling interrupts (non-timer interrupts). A register-by-register description is included below.

System Timer Register Name	Value	Description
Control Register	0x0003	Enable the system timer, use divide-by-4.
Prescale Register	0x8647	Given setting in Control Register, drive highest-frequency interrupt at 240 Hz
Interrupt Control Register 0	0x0104	Enable interrupt for Context 0 at priority 4.
Interrupt Control Register 1	0x0514	Enable interrupt for Context 2 at priority 4. Note that the Mode field is set to 1. This is typical for Fast Single-threaded contexts – they needn't acknowledge the interrupt.
Interrupt Control Register 2	0x011C	Enable interrupt for Context 3 at priority 4.
Interrupt Control Register 3	0x0123	Enable interrupt for Context 4 at priority 3.
Interrupt Control Register 4	0x0000	Not used (not enabled)

Table 5: System Timer Register Map

There is nothing special about the interrupt priorities selected in the table above, they can be adjusted as necessary to accommodate any other interrupts to the context. (Interrupt priority is context-specific, interrupts to different contexts have nothing to do with each other in these terms).

Step 6: External Interrupt Setup

In this example the safety switches come in on external interrupt lines. The fido1100 supports up to eight external interrupts. Similar to the System Timer interrupt control registers, the goal here is to assign the appropriate interrupts to the correct context (in terms of this example, context 0).

External Interrupt Control Register															
Reserved		Enable	Status	Polarity	Level	Reserved		Priority			Reserved		Context		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- Context -- This field defines associated context for that interrupt channel. At reset all interrupts are assigned to MC0.
- Priority -- This field allows the priority of the interrupt channel to be assigned, 0 is the lowest, 7 the highest. At reset all interrupts are assigned the lowest priority.
- Level -- Defines the trigger characteristics for the interrupt:
  - 0 -> Level Sensitive, when we return from interrupt if the signal is still active we will interrupt again.
  - 1 -> Edge Sensitive, only a new signal transition will cause a new interrupt.
- Polarity -- Defines the signal polarity:
  - 0 -> Low level or Falling edge triggers interrupt
  - 1 -> High level or Rising edge triggers interrupt

- Status -- Read this bit to determine the interrupt channel status, 1 = interrupt pending. **Cleared on read**.
- Enable -- Set to 1 to enable interrupt channel. If int channel is disabled the interrupt logic is held in a reset state.

For our example, the appropriate interrupts would be set accordingly:

Register	***Setting***	***Description***
Interrupt 0 Control	0x2CA0	Enable, rising edge trigger, priority 5, context 0
Interrupt 1 Control	0x24B0	Enable, falling edge trigger, priority 6, context 0
Interrupt 2 Control	0x20B0	Enable, low level trigger, priority 6, context 0

Table 6: Interrupt Assignments

Step 7: I/O Interrupt Setup

This will vary depending on the types of I/O involved (external memory-mapped, timer/counter units, or UICs). In any case, each I/O path has associated interrupts (in the case of the UICs, the interrupts are routed through corresponding PMU channels), and the control registers for these interrupts include Context fields. These fields should be set to point to the desired context (in our example, all of this I/O would go to Context 1).

Step 8: Implementing Priority and State With Context Control Registers

Each context has an associated context control register.

Context Control Register															
	Mode		Reserved		Priority			Reserved						State	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

- State -- This is the execution state of the associated context:
  - 00 - HALTED - The context will not execute when in the halted state. Only a write to this register can move a context out of this state.
  - 01 - NOT READY - The context can be made ready by any interrupt/exception targeted to it.
  - 10 - READY - The context is ready to run, and will be placed into execution when it is the highest priority ready context.
- Priority -- This is the priority of the context with respect to other contexts as well as on-board peripherals.
  - 111-000 -- 111 = highest priority, 000 is lowest priority.
- Mode -- defines the type of response the context makes to a pending interrupt.
  - 00 -- Standard mode. Operates like a standard microprocessor, if interrupt is of higher than or equal priority than the current interrupt mask in context status register then the current state of the context is stacked and execution begun at the interrupt vector.
  - 01 -- Fast mode. All interrupts are masked while the context is in the Ready state, when a context is in the NOT READY state an interrupt of priority higher than or equal to the mask in the status register for that context will cause a vector fetch and execution begins, nothing is

stacked.

- 11 -- Fast single-threaded mode. All interrupts are masked while the context is in the Ready state, when a context is in the NOT READY state an interrupt of priority higher than or equal to the mask in the status register for that context will cause execution to begin at the current Program Counter location for that interrupt. Nothing is stacked, no vector fetch is performed.

Register	***Value***	***Description***
Context 0 Control	0x2702	Fast Vectored mode, highest priority, ready to run
Context 1 Control	0x2601	Fast Vectored mode, priority 6, not ready
Context 2 Control	0x6402	Fast Single-Threaded mode, priority 4, ready
Context 3 Control	0x0201	Standard mode, priority 2, not ready
Context 4 Control	0x0001	Standard mode, priority 0, not ready

Table 7: Context Control Register Map

Notice that the state (READY, NOT READY, etc.) is different for various contexts:

Context 0 should be set to ready until initialization of the application is complete, it will then issue a 'sleep' instruction and move itself to the NOT READY state (if the context control register is written with a NOT READY setting, context 0 will immediately stop processing, before initialization is complete).

Contexts 1, 3, and 4 are all initialized to NOT READY. They will be moved to the ready state when they receive a system timer interrupt.

Context 2 is initialized to the ready state. This is to allow it to perform some one-time initialization before periodic processing begins (this approach can be used for a context in any mode, but it is most common for single-threaded contexts).

Example Source Code

Initialization

To initialize the fido1100 for our application we include the CHRmemmap.h file, declare our VectorBase and enter the main routine (see below).

Main.C Sample Code

```
#include "CHRmemmap.h"
...
extern int VectorBase;  // typically defined in assembly (e.g. crt0)
...
int main(void){
...
  // Set up each context (assumes that stack/etc. are already set up in context 0
  // Notice that all contexts are started with interrupts enabled (int mask in status register == 0),
  // also notice that some contexts are started in supervisor mode, others in user mode.

  // Context 1 is Fast vectored, initial program counter value is not important
  *CHR_CTX1_PC      = (unsigned long)Dummy;         // Program Counter
  *CHR_CTX1_VBR     = (unsigned long)&VectorBase;  // Vector Base Register
  *CHR_CTX1_A7S     = CONTEXT_1_STACK_TOP;         // System Stack Pointer
  // Because there is no user code in context 1, no need to allocate user stack
  *CHR_CTX1_SR      = 0x2000;                      // Status Register
  *CHR_CTX1_CONTROL = 0x0601;                      // Context Control Register

  *CHR_CTX2_PC      = (unsigned long)Computation;  // Program Counter
  *CHR_CTX2_VBR     = (unsigned long)&VectorBase;  // Vector Base Register
  *CHR_CTX2_A7S     = CONTEXT_2_STACK_TOP;         // System Stack Pointer
  // Because there is no user code in context 2, no need to allocate user stack
  *CHR_CTX2_SR      = 0x2000;                      // Status Register
  *CHR_CTX2_CONTROL = 0x6402;                      // Context Control Register

  *CHR_CTX3_PC      = (unsigned long)EthernetMain; // Program Counter
  *CHR_CTX3_VBR     = (unsigned long)&VectorBase;  // Vector Base Register
  *CHR_CTX3_A7S     = CONTEXT_3_SYSTEM_STACK_TOP;  // System Stack Pointer
  *CHR_CTX3_A7      = CONTEXT_3_USER_STACK_TOP;    // User Stack Pointer
  // Start context 3 in user mode
  *CHR_CTX3_SR      = 0x0000;                      // Status Register
  *CHR_CTX3_CONTROL = 0x0201;                      // Context Control Register

  *CHR_CTX4_PC      = (unsigned long)BackGround;   // Program Counter
  *CHR_CTX4_VBR     = (unsigned long)&VectorBase;  // Vector Base Register
  *CHR_CTX4_A7S     = CONTEXT_4_SYSTEM_STACK_TOP;  // System Stack Pointer
  *CHR_CTX4_A7      = CONTEXT_4_USER_STACK_TOP;    // User Stack Pointer
  // Start context 3 in user mode
  *CHR_CTX4_SR      = 0x0000;                      // Status Register
  *CHR_CTX4_CONTROL = 0x0001;                      // Context Control Register

  // Now set up Context 0 control register
  // NOTE: never modifying context 0 stack pointer, status register, or PC via the mem-mapped interface
  *CHR_CTX0_CONTROL = 0x2702;

  // Initialize UICs/PMU, set up interrupts

  // Initialize external interrupts
  *CHR_INTCONTROLCH0 = 0x2702;
  *CHR_INTCONTROLCH1 = 0x2601;
  *CHR_INTCONTROLCH2 = 0x6402;
  *CHR_INTCONTROLCH3 = 0x0201;
  *CHR_INTCONTROLCH4 = 0x0001;

  // Initialize the System Timer
  *CHR_SYSTIMER_PRESCALE = 0x8647;
  *CHR_SYSTIMER_INT00CONTROL = 0x0104;
  *CHR_SYSTIMER_INT01CONTROL = 0x0514;
  *CHR_SYSTIMER_INT02CONTROL = 0x011C;
  *CHR_SYSTIMER_INT03CONTROL = 0x0123;

  // When I do this write, the timer begins to count
  *CHR_SYSTIMER_CONTROL = 0x0003;

  // Initialization complete! ontext zero drops interrupt priority and waits for timer interrupt
  WriteIntMask(0);
  __asm__("sleep");
```

Context 0 Operation and Sample Code

```
// Note the attribute “noreturn”, this should be used
// for any handler in a FAST Vectored or Fast single-threaded context.
// Also, the file containing such handlers should be compiled with the
// “-fomit-frame-pointer” switch to gcc.
void C0SafetyHandler0(void) __attribute__((noreturn));
...
void C0IOhandler(void) __attribute__((noreturn));

int input_pass = 1;

void C0SafetyHandler0(void)
{
  ...
  __asm__(“sleep”);
}

void C0IOhandler(void)
{
  if (input_pass) {
    ...
    input_pass = 0;
  }
  else {
    ..
    input_pass = 0;
  }
  __asm__(“sleep”);
}
```

```
/* assembly example of handler */
.text
.align 4
.globl C0SafetyHandler1
C0SafetyHandler1:
  /* You can go right to work, no need to save registers, etc. */
  ...
  sleep
```

Context 1 Operation and Sample Code

This just consists of a set of interrupt handlers. For the basic structure look at examples for context 0.

Context 2 Operation and Sample Code

Because this is a Fast Single-Threaded context, the 'handler' can be any patch of code. The typical approach is to write it as a loop. Note that there is first-pass initialization code prior to the loop. In this example, this would execute as soon as context 0 issues the sleep instruction after initialization and before the system timer starts any other processing.

```
void Computation(void)
{
  // Local Declarations
  ...
  // local initialization code

  while(1) {
       __asm__(“sleep”);
   // calculate next set of outputs
   ...
   // do anything else (model expected next inputs/etc.)
   ...
  }
}
```

The same approach can be taken for an assembly language body for a single-threaded context.

```
.text
.align 4
.globl Computation
Computation:
  // Load up pointers, constants, etc. that you'll be using in body, they only need to be loaded once
  move.l #ACONSTANT,%d0
  move.l #somepointer,%a0
 ...
LOOPTOP:
  sleep
  /* do the work */
  bra LOOPTOP
```

Context 3 Operation and Sample Code

```
void EthernetMain(void){
  // This program can do whatever it needs to do.
  // The only caveat is that at some point it needs
  // to issue a sleep instruction so that it won't
  // starve the lower priority context.

  while (1) {

    ProcessReceiveBuffers(...);
    ProcessTransmitMessages(...);

    __asm__(“sleep”);
  }
}
```

Context 4 Operation and Sample Code

```
void UARTioHandler(void) __attribute__((interrupt_handler));

void UARTioHandler(void)
{
   // Note that this handler uses the “interrupt_handler” attribute,
   // this causes the compiler to save any used registers (it's in a
   // standard context) and use an RTE instruction to return rather
   // than RTS.
   ...
}

void BackGround(void)
{
  while(1) {
    ...
  }
}
```

Conclusions

There are other features of the fido1100 that could be applied to this application.

- Context timers could be used to guarantee that no context starves lower-priority contexts.
- The Memory Protection Unit can be used to partition memory between portions of the application so that errors in one portion don't impact other parts.
- Various performance-critical pieces of the application can be loaded into the Rapid Execution Memory.

Also, the example above does not represent the only, or even the best, way to put together this kind of application. For instance, Context 0 is set up as a Fast Vectored context with I/O handlers and Safety-related handlers. As discussed above, the Safety handlers can be delayed by the time it takes to execute an instance of the I/O handler. Another way to manage this problem would be to set up Context 0 as a standard context with the I/O handler operating in the 'body' of the context as a continuous loop, the safety handlers would still be interrupt based. Now the safety handlers would preempt IO handling.

Thank You

Thank you for taking the time to review this application note. We hope you have found the information included in this application useful and east to understand. Please feel free to contact the Innovasic Support Team any time with questions or comments.

Innovasic Support Team
3737 Princeton NE
Suite 130
Albuquerque, NM, 87107

(505) 883-5263

support@innovasic.com
http://www.innovasic.com