Example Embedded Application for an Industrial Control System

Real Time O/S

- An application has a *hard real-time* requirement if it must always meet a time constraint, otherwise the system will fail.
- For example, some systems must be able to respond to an interrupt on time, every time. If it ever fails to meet its deadline, then the system won't work.
- LynxOS was designed for hard real-time problems

Designing a real-time system

- A real-time system is composed of multiple threads running concurrently and doing different tasks. (I/O, Processing, Analysis, etc)
- Threads can be *user threads* or *kernel threads*.
- In the process of the design of your application and device drivers it is important to identify:
  - Which threads, functions or system calls will affect the response time of your hard real-time requirement.
    - Example: The update of the database can be put in a low priority thread.
  - What are the data that need to be transferred between threads.
    - Example: Update of the screen can be another thread that will read its value from shared memory or a message queue.
  - What type of communication do you need between threads. (Synchronization, Data transfer, etc)
    - Example: Use a signal when the thread should update the screen.
  - Is there critical code or resources that will be shared between threads.
    - Example: changing a shared data structure concurrently by two different threads or any other critical code region protected by mutexes.

Processes model

- In UNIX, a program is executed by a *process*.
- Each LynxOS process has at least one flow-of-control, called the *main thread* or *initial thread.*
- The process also includes resources such as file descriptors, signal masks, and many other items.
- In LynxOS, each process has its own *virtual address space.* This means that programs running in different processes are protected from each other. No process can see the memory used by another process, so processes cannot inadvertently damage each other.
- IPC, Inter-Process Communication means under LynxOS a communication between two different *main threads*

Threads model

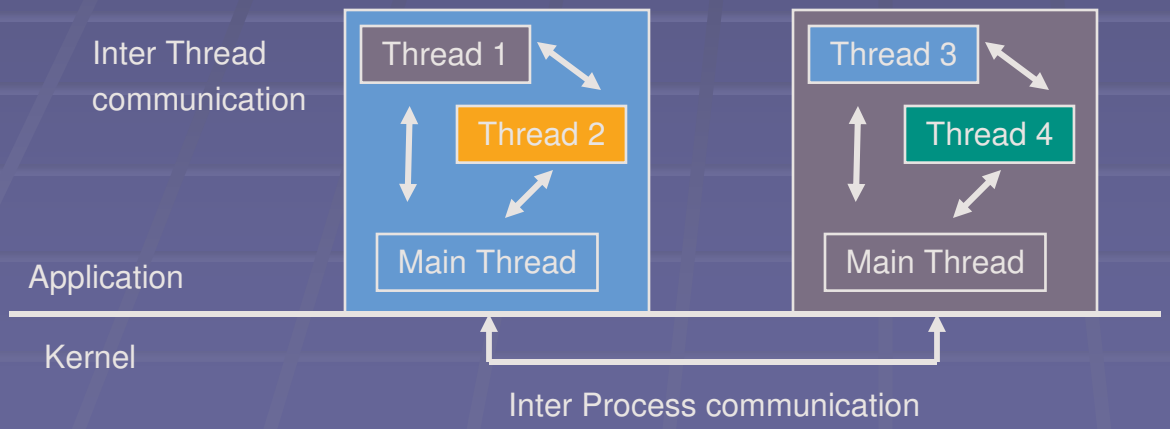
- Within a process, there may be more than one thread. These threads share the virtual address space of the process. This means that threads within a process can see the same memory.
- LynxOS also has *kernel threads.* They are part of the operating system, and are often used in device drivers.
- The threads are the entities that get scheduled. There is only one scheduler in LynxOS, so user threads and kernel threads are all scheduled the same way.
- This means that a user thread can preempt a kernel thread.
- ITC, Inter-Thread Communication means under LynxOS a communication between two different *threads* running in the same process context or not.

Kernel Threads model

- Kernel threads are schedulable and prioritized, just like user threads.
  - You can see the *kernel thread* with *ps -tT* command.
- Application programmers do not create kernel threads.
- LynxOS can use threads within the kernel to perform much of the I/O processing that is typically done by an ISR in other OS.
  - The goal is to have a short ISR. The code that services the interrupt is moved to a kernel thread that will be scheduled according to its priority.

Threads

- A thread is a flow-of-control that runs within a process context.
- Threads in the same process share the same virtual address space.
- Threads are the running entities of LynxOS.
- Threads don't have a parent-child hierarchy.
- Any thread created within a process can terminate itself or can read the exit status of any other thread
- Exception: If the main thread exits, the process terminates. All other threads in the process are terminated.



- **Each process has its own protected address space. Communications between processes require kernel services.**
- **Threads in a single process exist in the same address space. They can communicate and share data using globals**

Multi-threaded program

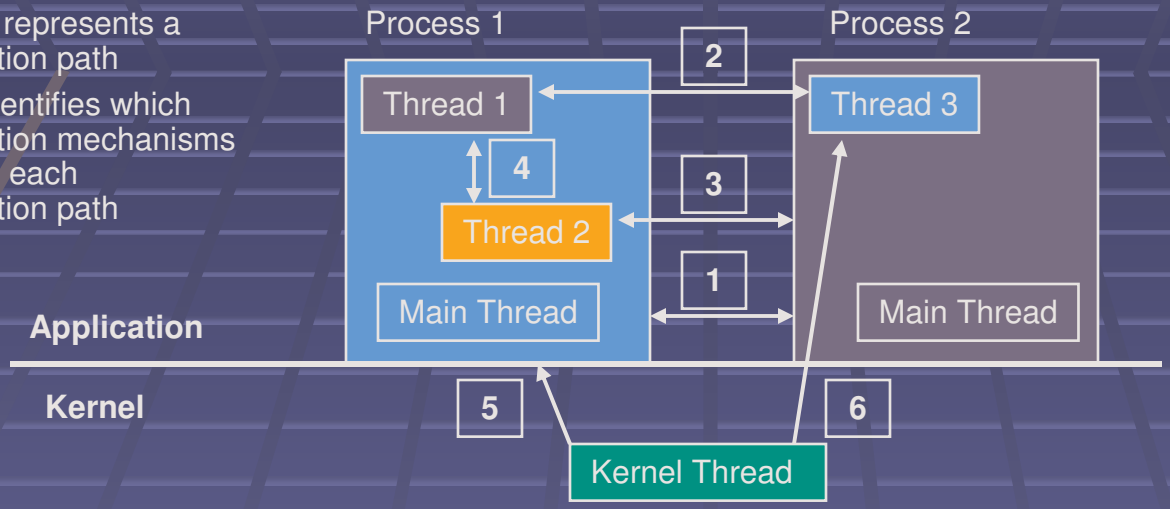
- If two threads are running concurrently and are calling the same function we cannot predict:
  - at what time you get preempted in the function call
  - the order of execution of the same function.
- So functions use in a multi-threaded program should be thread safe or you should provide your own synchronization.
- Use stack variables, not static or global variables.
  - Every thread that executes the function gets its own copy of the stack variable.
- A thread safe function is a function that may be executed by two or more threads at the same time and still behaves the same. Known also as a reentrant function.

Design Issue

- Is a multithreaded application better than a multi-process?
- Multithreaded:
  - Advantage
    - Run in the same address space, switch time, communication, and synchronization between threads are fast.
  - Disadvantage
    - All threads terminate if the main thread terminates.
- Multi-processes:
  - Advantage
    - Each main thread runs in a separate protected memory address space.
  - Disadvantage
    - Switch time, communication, and synchronization between main threads are slower.

Inter-Thread Communications

- Each arrow represents a communication path
- The table identifies which communication mechanisms are valid for each communication path



- Main thread is the running entity of the process

- Yes* : If declared in shared memory
- Yes**: If using BUILPID() macro
- Yes***: If using *sigwait()*
- Yes**** If processes are related, such as parent & child.

	Unnamed Pipes	Signals	RTS Signals	Named pipes, Message Queues, Named Semaphores, Shared Memory	Unnamed Semaphore, Mutex, Condition Variables, Barriers, RW Locks
1	Yes****	Yes	Yes	Yes	Yes*
2	Yes****	Yes**	Yes**	Yes	Yes*
3	Yes****	Yes**	Yes**	Yes	Yes*
4	Yes	Yes	Yes	Yes	Yes
5	No	Yes	No	No	No
6	No	Yes***	No	No	No

Introduction to Semaphores

- High performance inter-threads synchronization mechanism
- Used for synchronizing access to shared resources
  - Global variables, Shared memory segments
- Semaphores can also be used:
  - to help threads co-operate: One thread blocks on a semaphore, waiting for another thread to signal it.
  - for managing a pool of resources: A semaphore is used to count free resources in a pool, so a task can block when no resources are available.
- There are 2 types of POSIX semaphores:
  - Named Semaphores
  - Unnamed semaphores
- LynxOS has also its own semaphore implementation that offers priority inheritance but are not POSIX and must be linked with *liblynx.a* library.

Locking and Unlocking a Semaphore

- To test and block on a semaphore, you use the functions *sem_wait()* or *sem_trywait()* for the non-blocking version.
  - `int sem_wait(sem_t *sem);`
    - If semaphore value > 0, value is decremented and thread continues
    - If semaphore value <= 0, value is decremented and thread is blocked, put on semaphore queue (priority ordered)
  - `int sem_trywait(sem_t *sem);`
    - If semaphore value > 0, value is decremented and thread continues
    - If semaphore value <= 0, returns immediately with -1, errno == EAGAIN
- To post a semaphore, you use the function *sem_post()*
  - `int sem_post(sem_t *sem);`
    - If semaphore value >= 0, value is incremented
    - If semaphore value < 0, value is incremented and highest priority thread is made ready to run.

Example

```
#include <sys/types.h>
#include <stdlib.h>
#include <semaphore.h>
sem_t *sem_read, *sem_write;
void read_access() {
     sem_wait(sem_read);
     read_shared_memory();
     sem_post(sem_write);
}

void write_access() {
     sem_wait(sem_write);
     write_shared_memory();
     sem_post(sem_read);
}

main()
{
     int i;
     init_shared_memory();
     /* Create and Open /sem_read (lock) */
     if ((sem_read =
     sem_open("/sem_read",O_CREAT|O_EXCL,
     S_IRUSR|S_IWUSR, 0))==(sem_t*)-1) {
             perror("sem_open"); exit(EXIT_FAILURE); }
```

```
     /* Create and Open the /sem_write (unlock) */
     if ((sem_write =
     sem_open("/sem_write",O_CREAT|O_EXCL,
     S_IRWXU, 1))==(sem_t*)-1) {
             perror("sem_open"); exit(EXIT_FAILURE); }

     /* Remove /sem_xxx when they will be closed */
     if (sem_unlink("/sem_read")==-1) {
             perror("sem_unlink"); exit(EXIT_FAILURE); }
     if (sem_unlink("/sem_write")==-1) {
             perror("sem_unlink"); exit(EXIT_FAILURE); }

     switch(fork()) {
             case -1:
             perror("fork"); exit(EXIT_FAILURE);
             case 0:
                     for (i = 0; i < 10000; i++)
                             write_access();
                     break;
             default:
                     for (i = 0; i < 10000; i++)
                             read_access();
     }
     close(sem_read); close(sem_write);
     close_shared_memory();
}
```

Design Issue

- Semaphores are an efficient way to synchronize a single reader/writer or to protect a counting variable or any countable resources.
- POSIX Semaphores should not be use to protect critical code region, you can end up with a priority inversion situation. Use LynxOS semaphores or pthread mutexes.
- Disadvantage
  - Priority inversion problem with POSIX semaphores.
- Advantage
  - System call overhead only if the thread blocks

Mutexes

- A mutex is a mechanism to be use to protect critical code region.
- A mutex has only two states
  - locked and unlocked
  - Threads waiting on a locked mutex are queued in a priority-based queue
- Unlike a semaphore
  - mutexes have no concept of a count
  - A mutex has the property of ownership: only the thread currently possessing a mutex can release the mutex.

Using a Mutex

- A thread may lock a mutex to ensure exclusive access to a resource by issuing either *pthread_mutex_lock()* or *pthread_mutex_trylock()*
  - `pthread_mutex_lock(mutex_t *mutex)`
  - `pthread_mutex_trylock(mutex_t *mutex)`
    - non-blocking version
    - Returns 0 or EBUSY if mutex is already locked
- A thread executing *pthread_mutex_lock* on an owned mutex will block until the owner of the mutex releases the mutex
- A thread can release a mutex using *pthread_mutex_unlock()*
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- The mutex becomes not owned only if no threads are queued on the mutex
- If threads are waiting, ownership of the mutex is given to the thread of highest priority that has been queued for the longest period

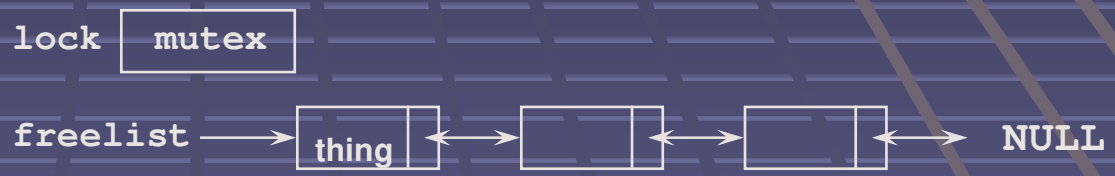
Example

```
struct resource {
        pthread_mutex_t lock;
        thing *freelist;
}

thing *thing1, *thing2;

list *allocate ()
{
        pthread_mutex_lock (&resource->lock);
        ptr = freelist;
        freelist = ptr->next;
        pthread_mutex_unlock (&resource->lock);
        return (ptr);
}


void deallocate (list *ptr)
{
        pthread_mutex_lock (&resource->lock);
        ptr->next = freelist;
        freelist = ptr;
        pthread_mutex_unlock (&resource->lock);
}
```

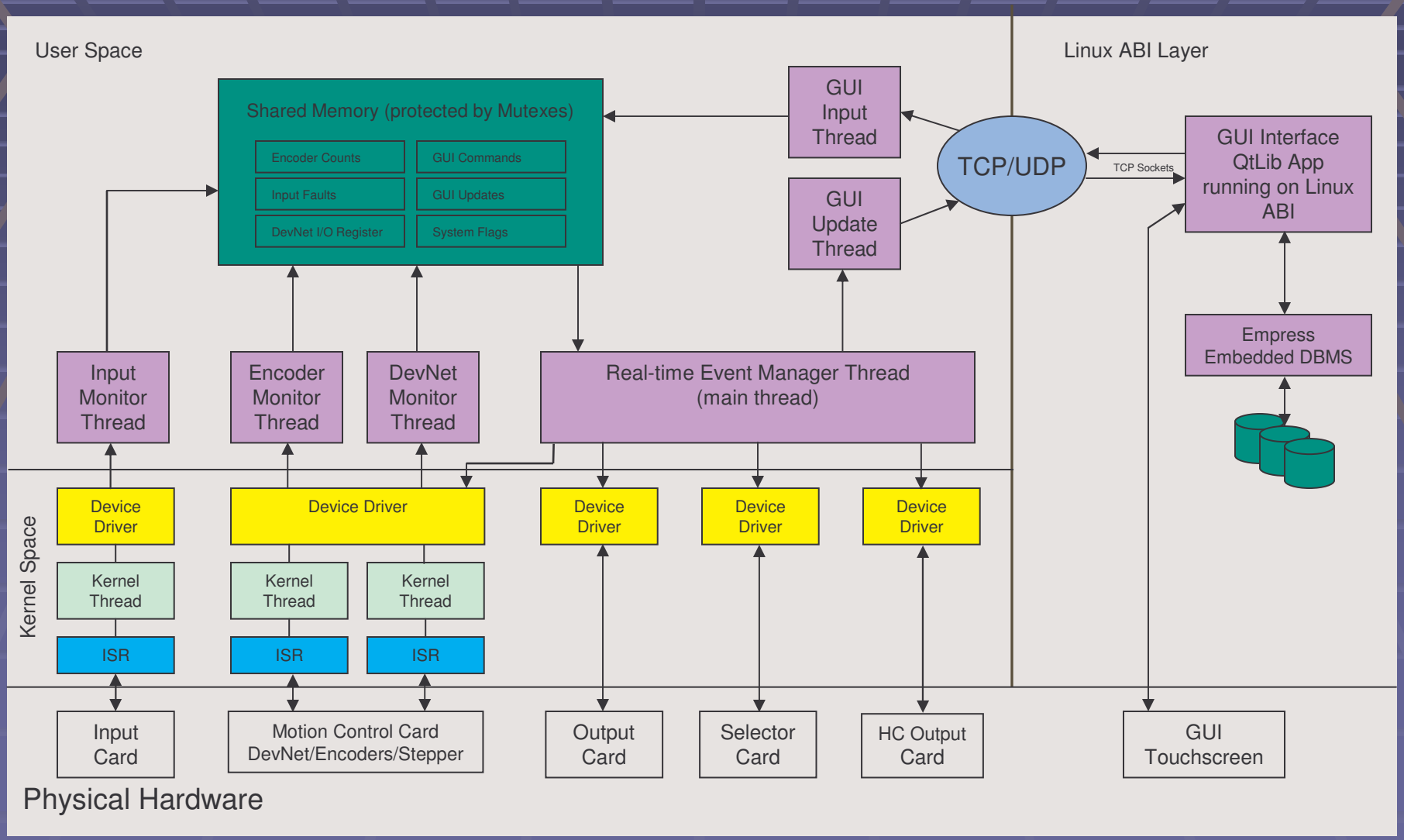


Knitting Machine

System Requirements

- Motion Control
- Cylinder Position
- Stitch CAM
- Selector Actuation
- Motion Faults (Stop Motion Inputs)
- Outputs
- User Interface
- Network Interface
- User Program and Configuration (Database)

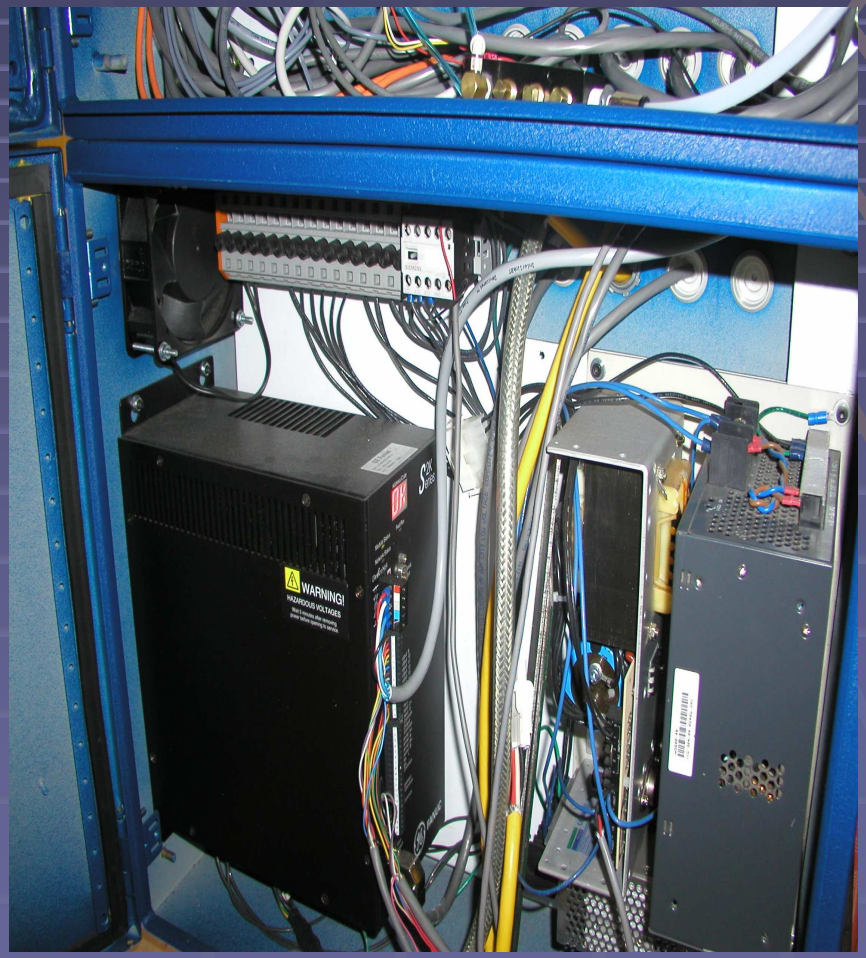
System Architecture



Motion Control

- Main Axis - Cylinder
  - Servo Control
  - Dynamic Motion Profiles
  - Bi-directional
  - Variable Speed
  - 50 to 300 RPM
- Stitch CAM
  - Stepper motor
  - Encoder feedback
  - Bi-directional
  - Incremental Positioning

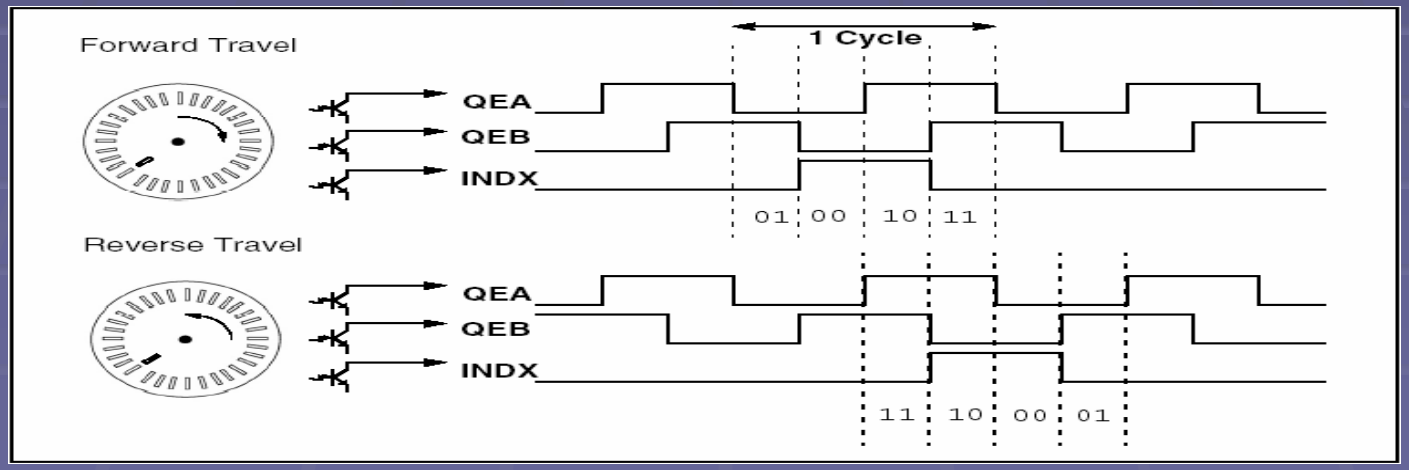
Main Servo System



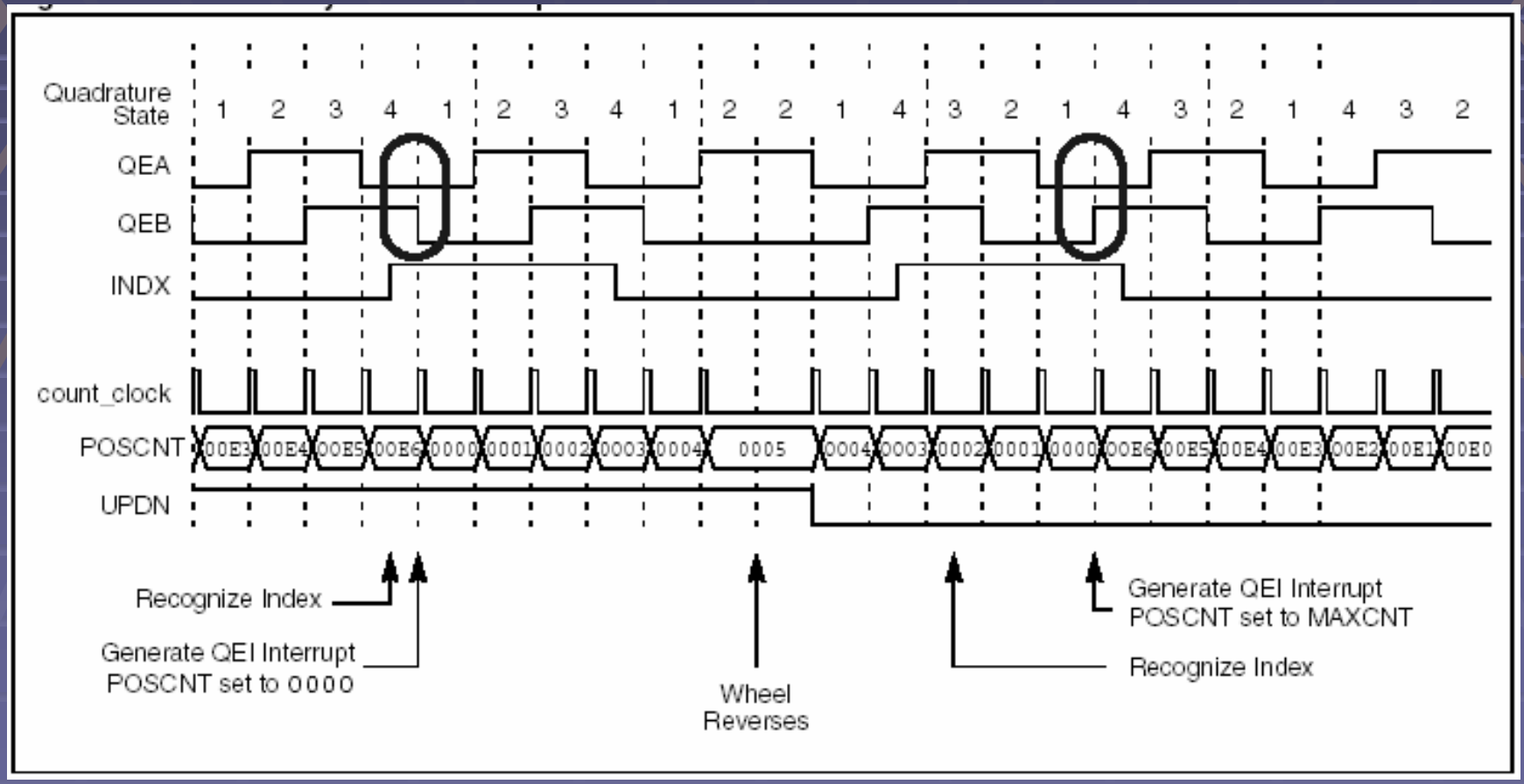
Main Cylinder – 84 Needles

Cylinder Position

- 84 or 108 Needles
- Encoder Interface
  - 2048 Pulses/Revolution
  - ABZ Inputs with complementary inputs
  - Gated Index line



Quadrature Encoder Signals



Encoder Timing Requirements

- Maximum RPM = 300 RPM
- Maximum Needle count = 108 Needles
- Encoder generates 2048 pulses per revolution

- At max speed:

$$\frac{300\ \text{Revs}}{1\ \text{Minute}} \times \frac{1\ \text{Minute}}{60\ \text{Seconds}} \times \frac{2048\ \text{Pulses}}{1\ \text{Rev}} = \frac{10{,}240\ \text{Pulses}}{\text{Second}}$$

- This equates to an encoder interrupt every 97.65µs!

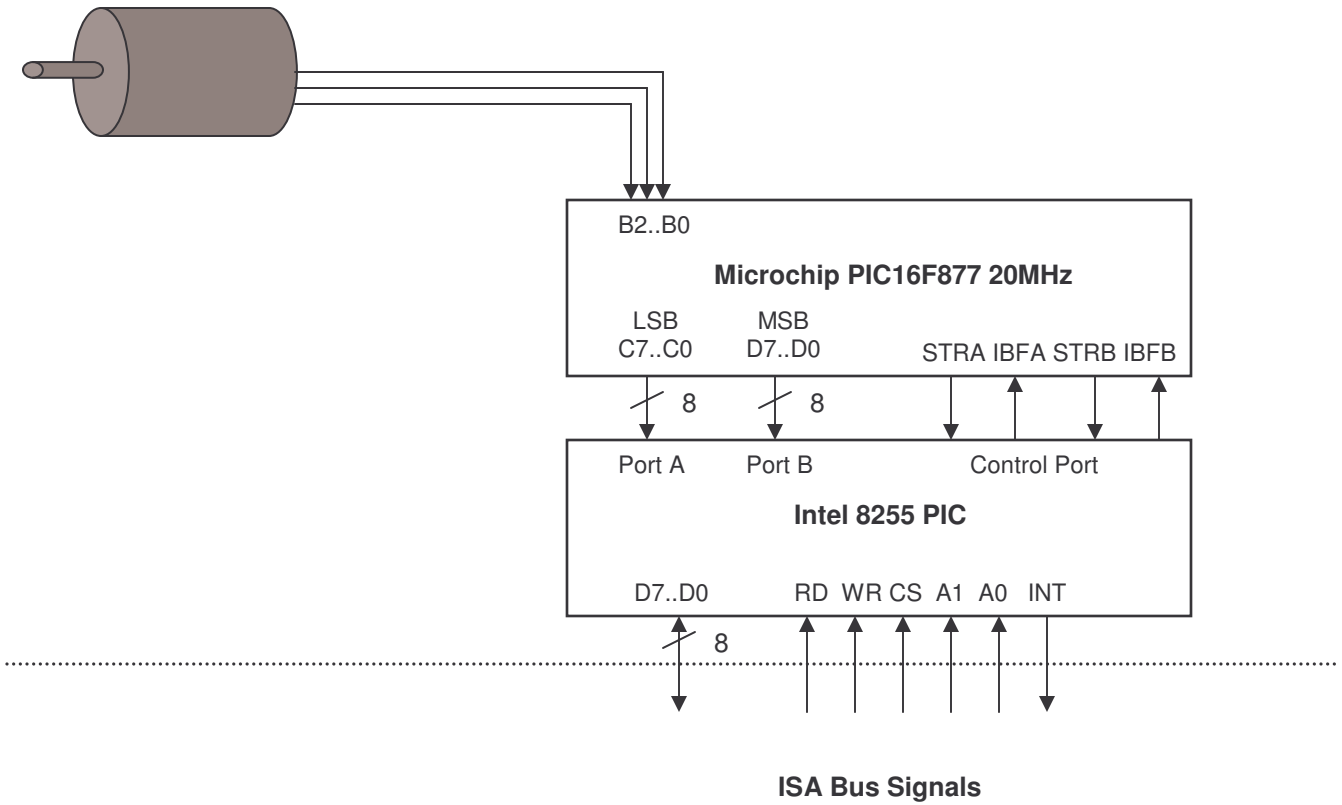
Hardware partitioning

- Alternative is to interrupt per needle change
- At max speed:

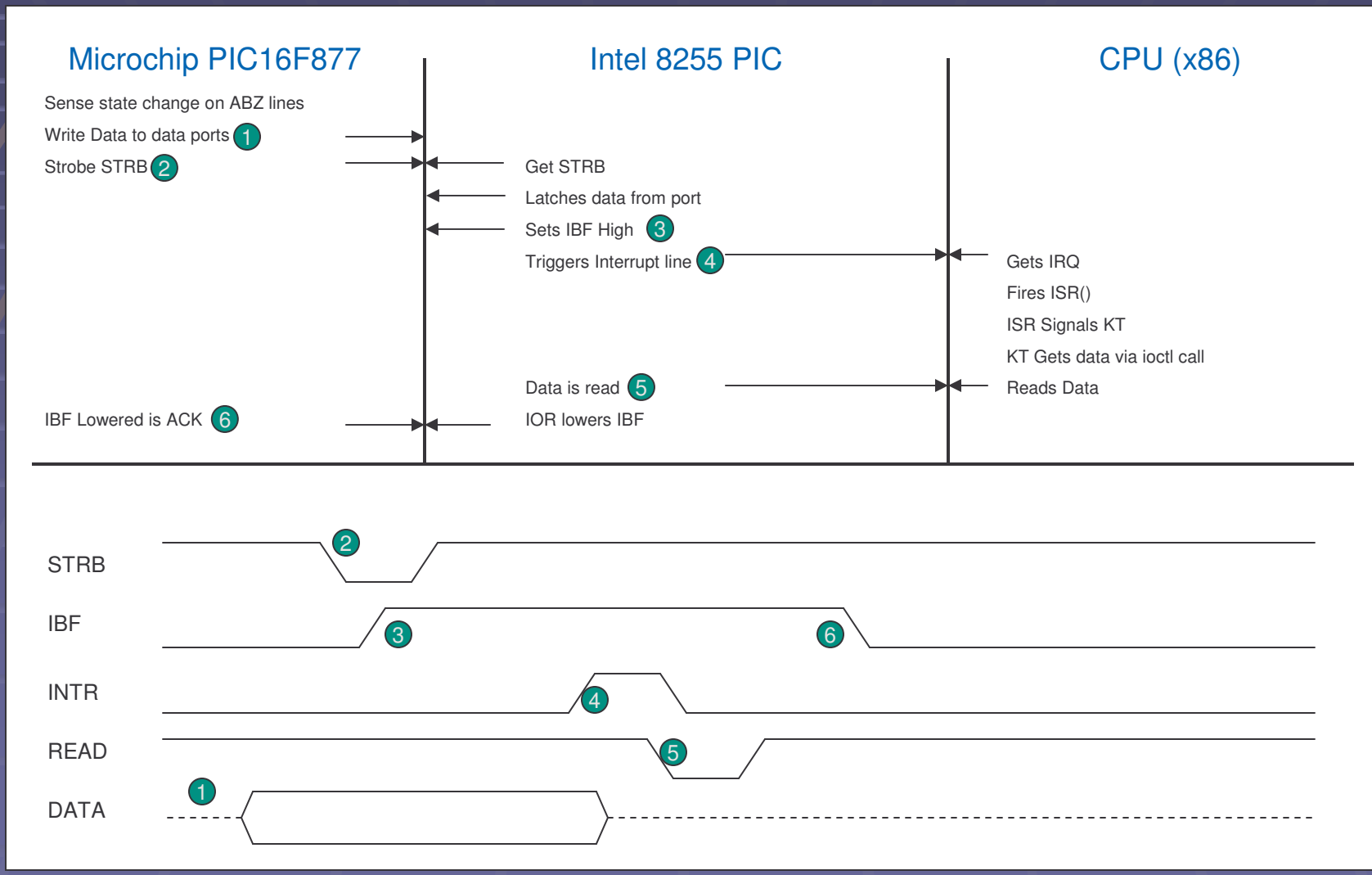
$$\frac{300\ \text{Revs}}{1\ \text{Minute}} \times \frac{1\ \text{Minute}}{60\ \text{Seconds}} \times \frac{108\ \text{Needles}}{1\ \text{Rev}} = \frac{540\ \text{Needles}}{\text{Second}}$$

- This equates to a needle interrupt every 1.85ms
- To achieve this, encoder processing must be moved off of the main CPU to a dedicated microcontroller

Encoder Architecture



Encoder Signals

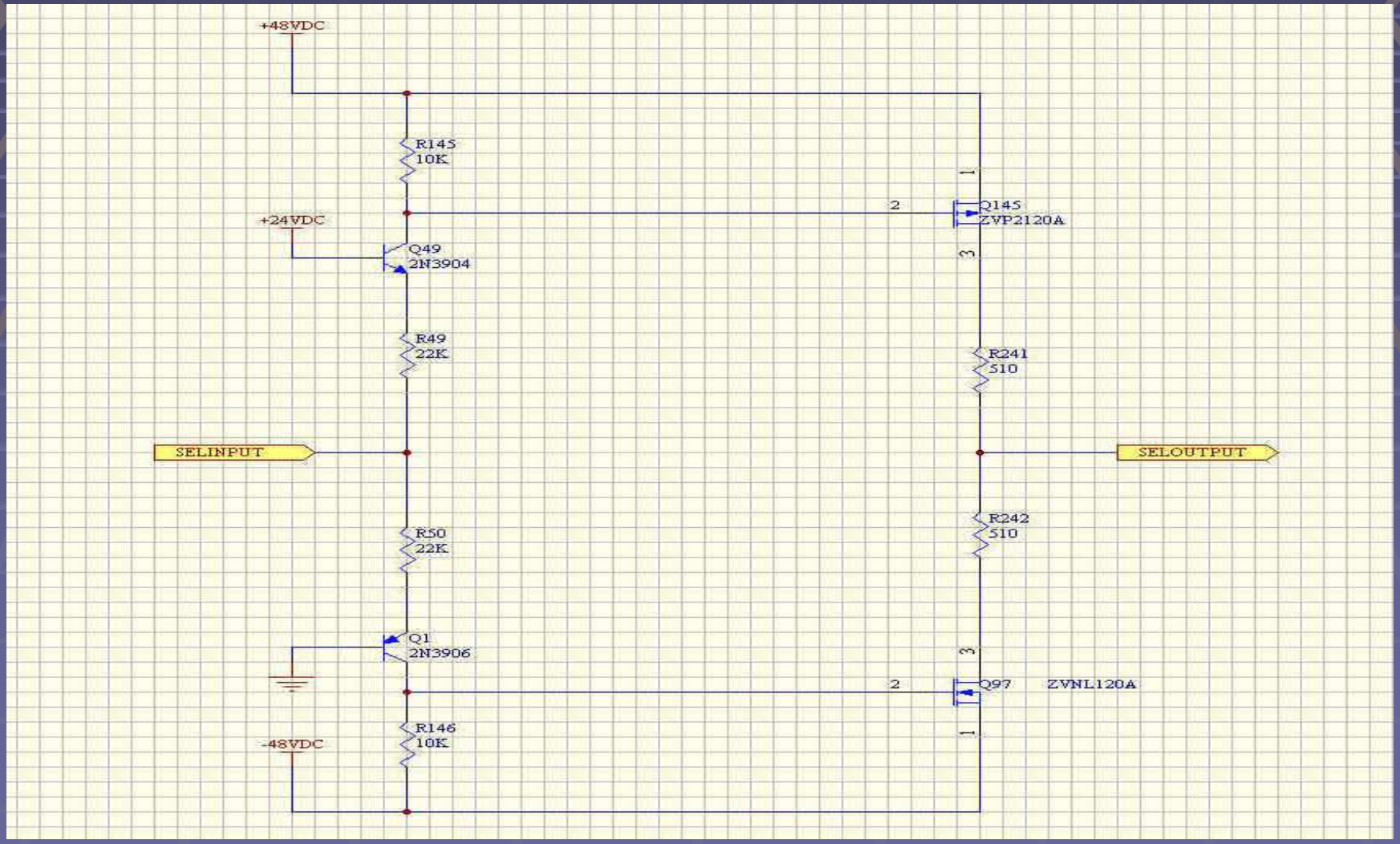


Selector Actuation

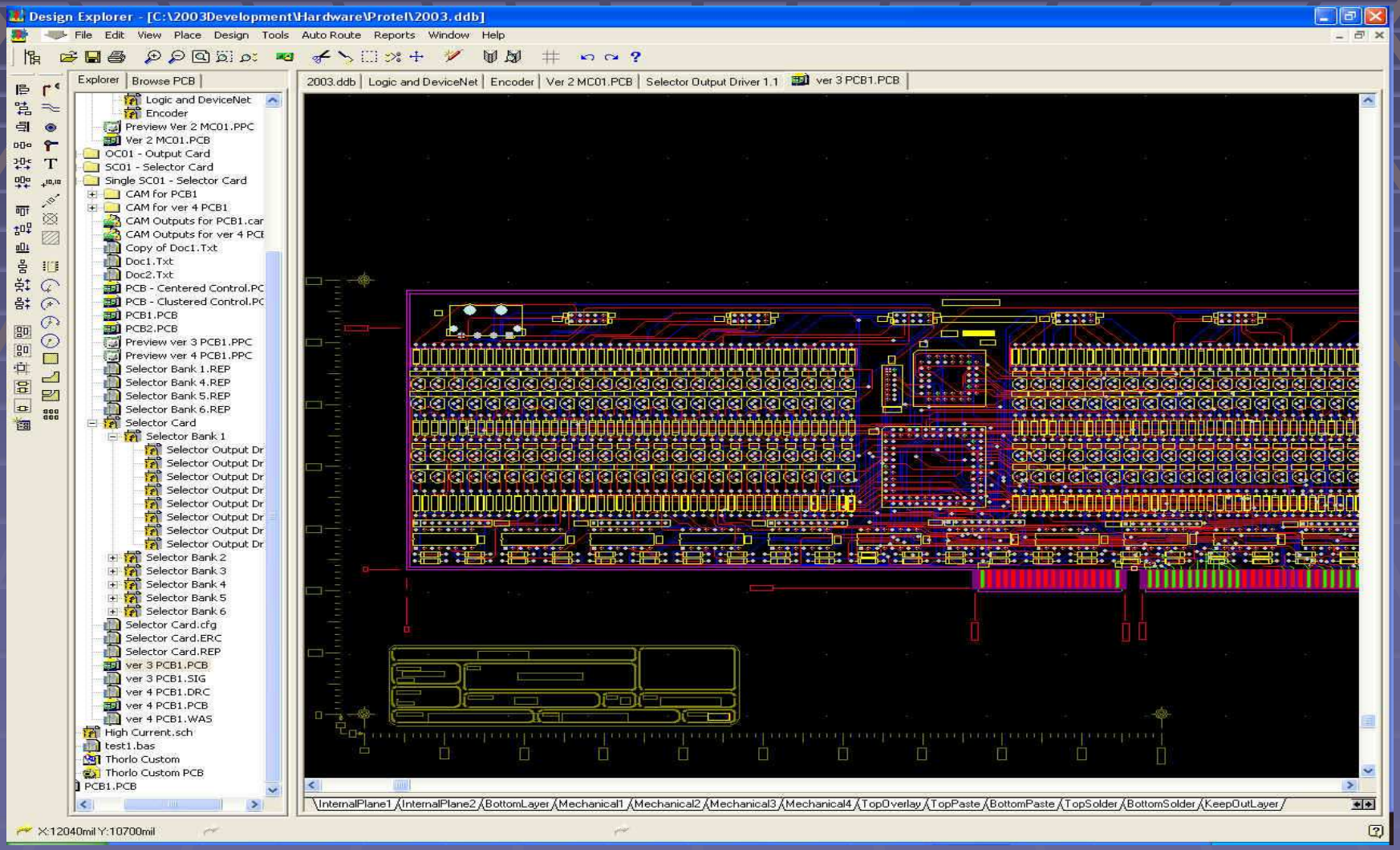
- Selector Blocks utilize bi-polar morphing actuators
- Control of actuator is via bi-polar +/-48VDC
- Needed a custom solution (+/-48VDC outputs not COTS)
- Each selector output has to be activated and deactivate within 100µs.
- Drive signals need to be standard 5VDC TTL levels
- Result solution was a design incorporating bi-polar drivers
  - Inputs -> 5VDC TTL
  - Outputs -> +/-48VDC MOSFET Drivers

Selector Banks

Selector Driver

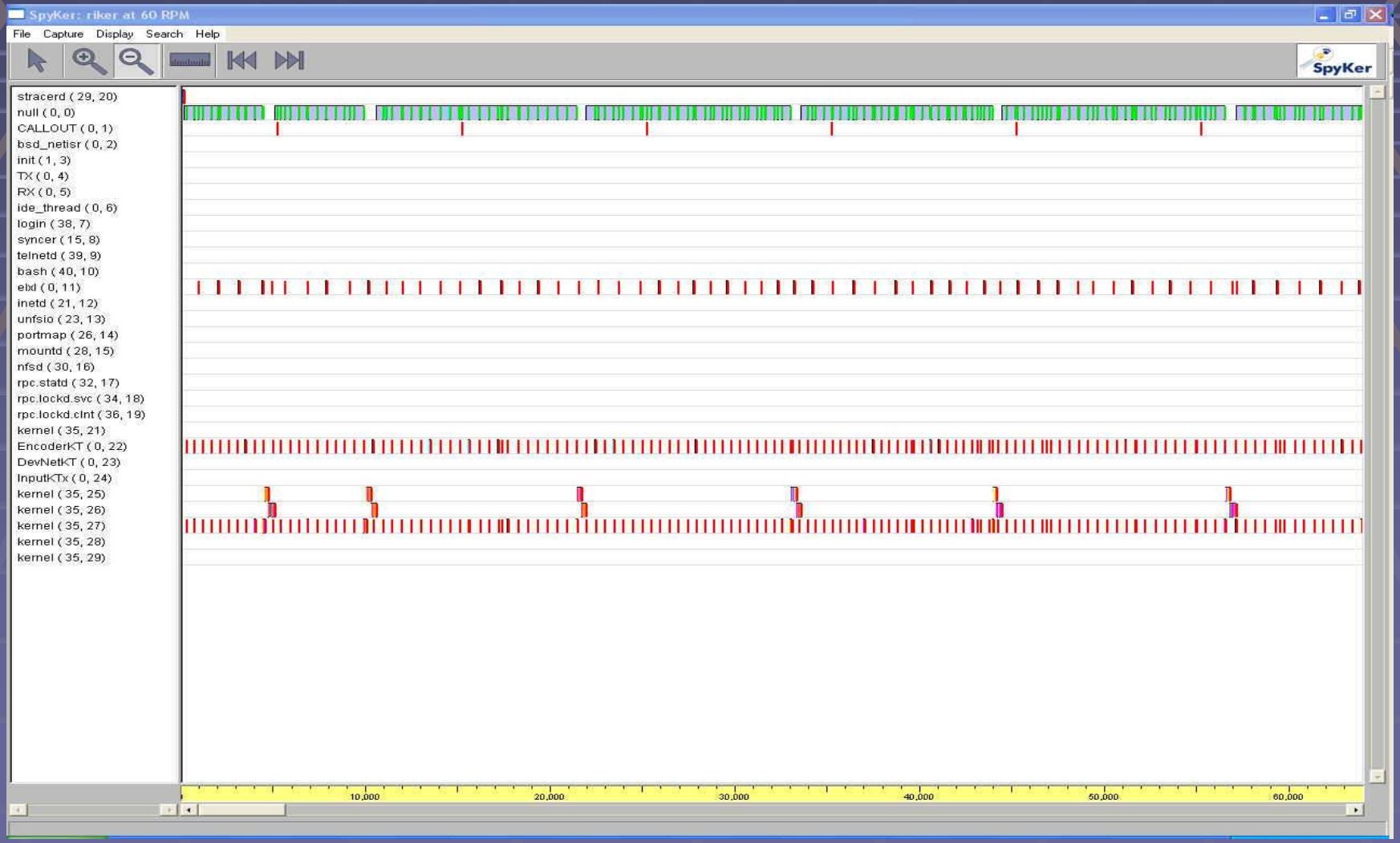


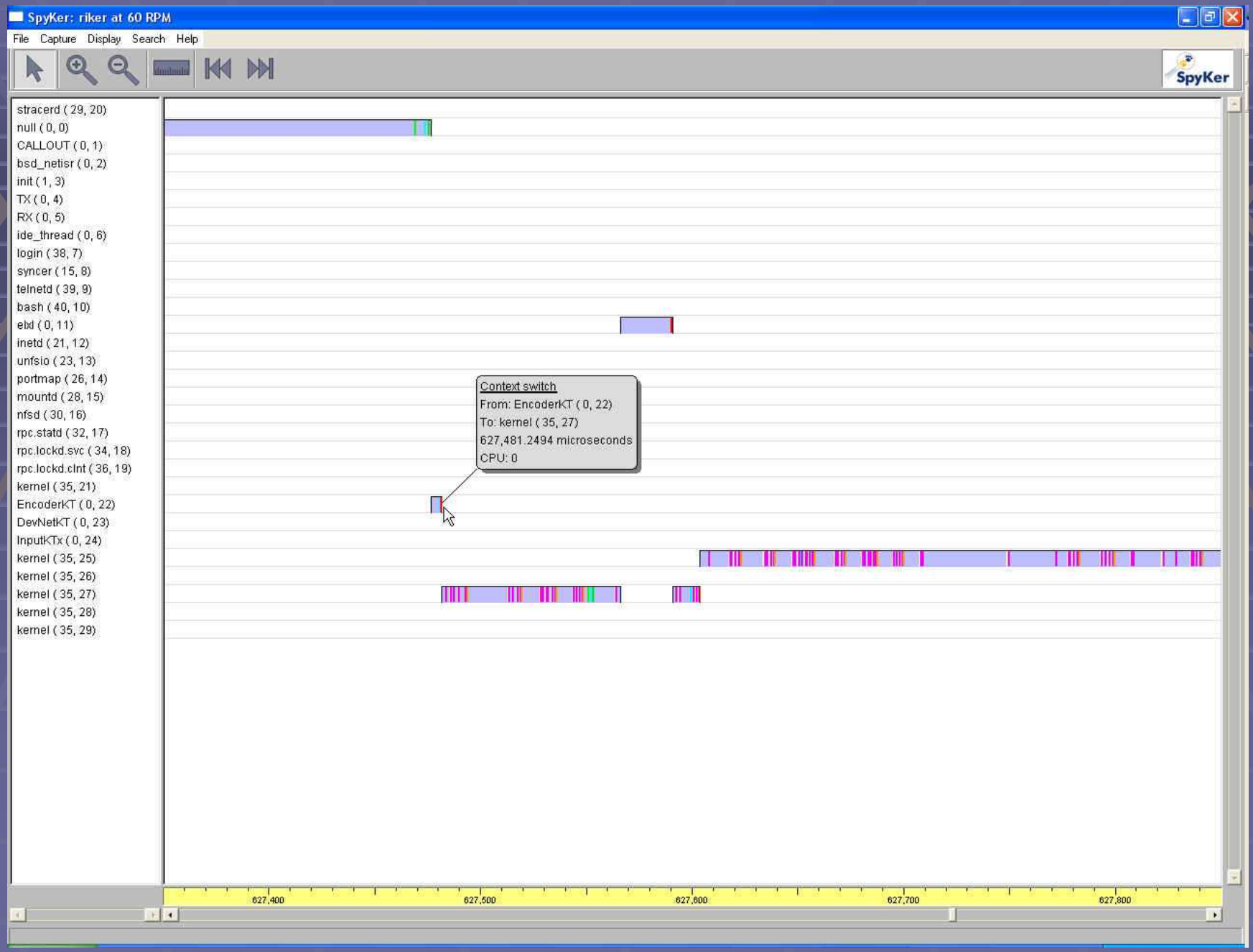
Selector Card Design

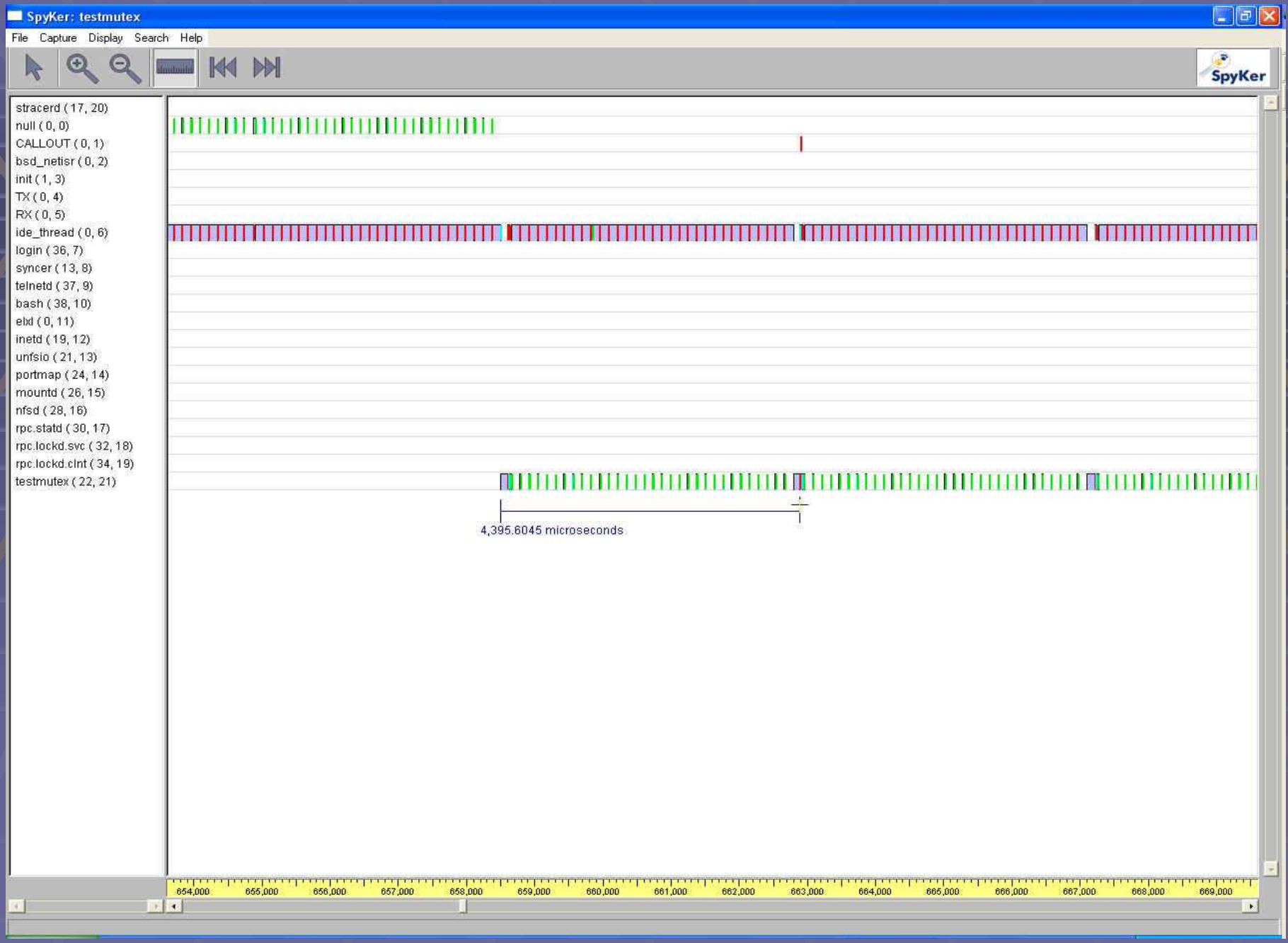


Pneumatic Outputs

Real-time Performance




SpyKer: riker at 60 RPM
File Capture Display Search Help
SpyKer
stracerd (29, 20)
null (0, 0)
CALLOUT (0, 1)
bsd_netisr (0, 2)
init (1, 3)
TX (0, 4)
RX (0, 5)
ide_thread (0, 6)
login (38, 7)
syncer (15, 8)
telnetd (39, 9)
bash (40, 10)
elxl (0, 11)
inetd (21, 12)
unfsio (23, 13)
portmap (26, 14)
mountd (28, 15)
nfsd (30, 16)
rpc.statd (32, 17)
rpc.lockd.svc (34, 18)
rpc.lockd.clnt (36, 19)
kernel (35, 21)
EncoderKT (0, 22)
DevNetKT (0, 23)
InputKTx (0, 24)
kernel (35, 25)
kernel (35, 26)
kernel (35, 27)
kernel (35, 28)
kernel (35, 29)
Context switch
From: EncoderKT (0, 22)
To: kernel (35, 27)
627,481.2494 microseconds
CPU: 0
627,400
627,500
627,600
627,700
627,800


SpyKer: testmutex
File Capture Display Search Help
SpyKer
stracerd (17, 20)
null (0, 0)
CALLOUT (0, 1)
bsd_netisr (0, 2)
init (1, 3)
TX (0, 4)
RX (0, 5)
ide_thread (0, 6)
login (36, 7)
syncer (13, 8)
telnetd (37, 9)
bash (38, 10)
elxl (0, 11)
inetd (19, 12)
unfsio (21, 13)
portmap (24, 14)
mountd (26, 15)
nfsd (28, 16)
rpc.statd (30, 17)
rpc.lockd.svc (32, 18)
rpc.lockd.clnt (34, 19)
testmutex (22, 21)
4,395.6045 microseconds
654,000
655,000
656,000
657,000
658,000
659,000
660,000
661,000
662,000
663,000
664,000
665,000
666,000
667,000
668,000
669,000

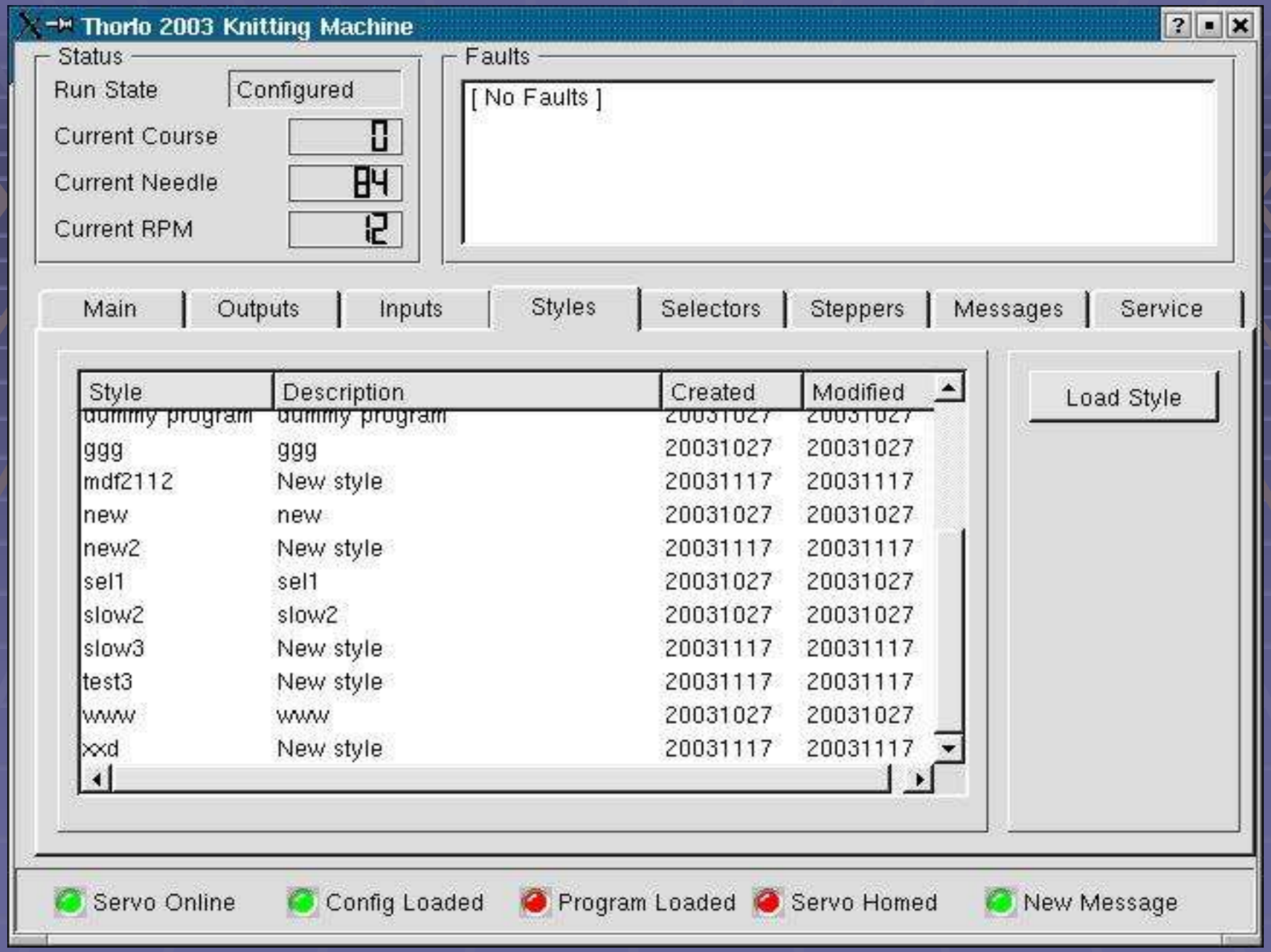
Final System

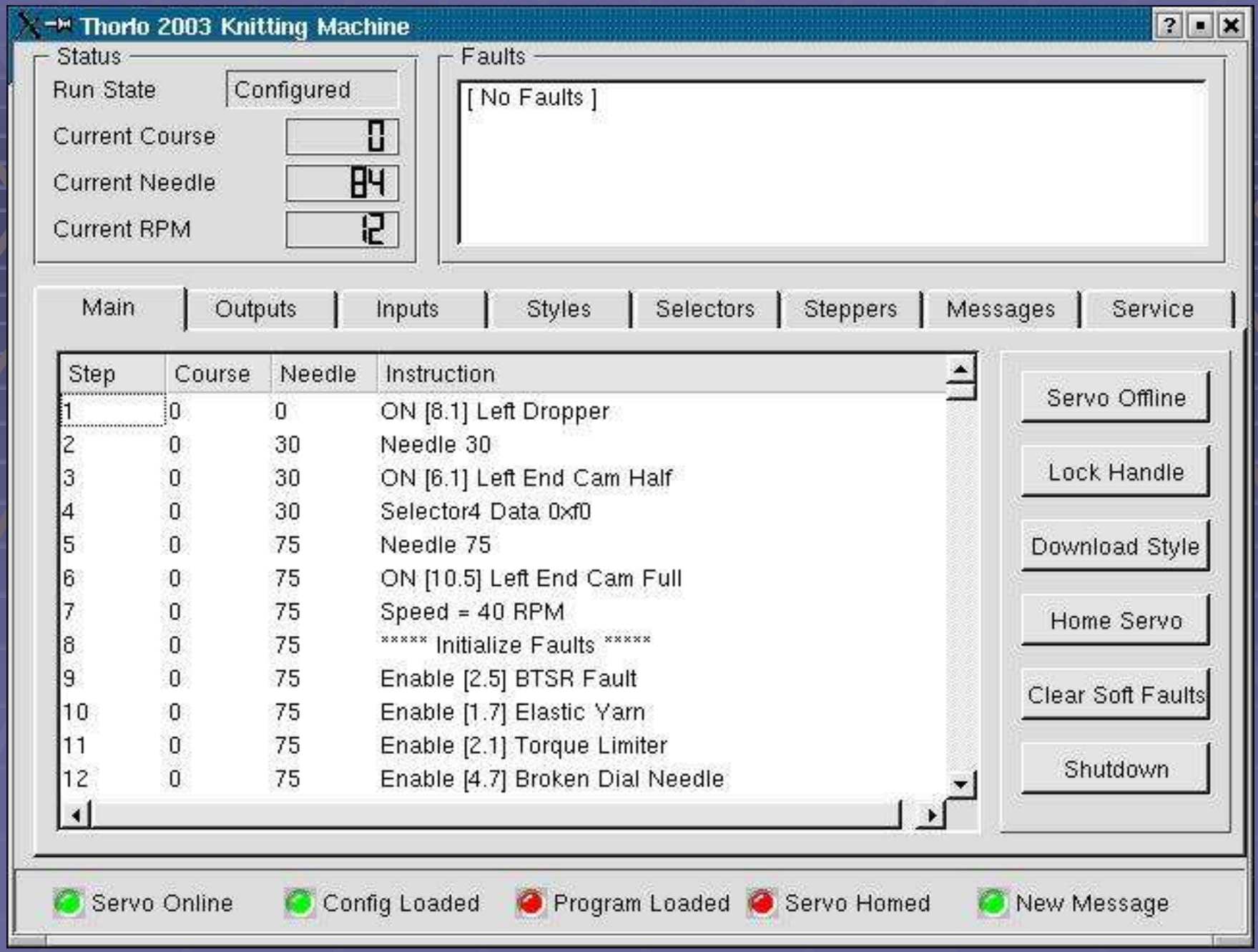
Software Packages Used

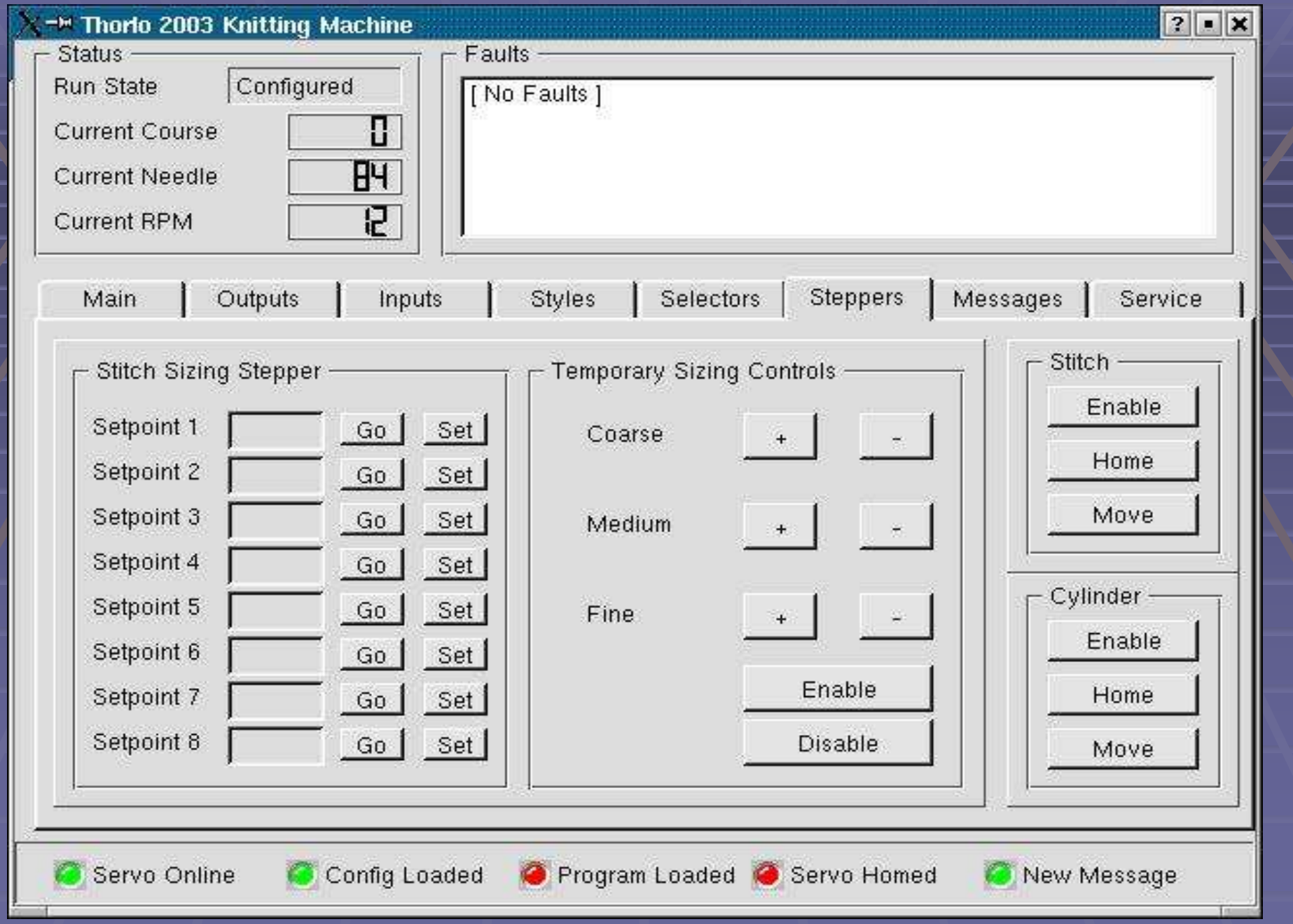
- Hardware Design
  - Protel – Schematic capture and PCB Design
  - Xilinx – VHDL Design
  - ModelSim – VHDL Simulation
- Software Design
  - CIMplicity MotionFX – Motion Control Design
  - C/C++ - Gnu toolchain
  - Microchip
    - MLAB – IDE for development, simulation, and debugging
  - LynxOS – RTOS from Lynuxworks
  - Linux – RedHat 7.2 for GUI development
  - KDeveloper – KDE/QtLib IDE for Linux native development
  - Empress – Embedded SQL database
  - SpyKer – Kernel Timing Capture

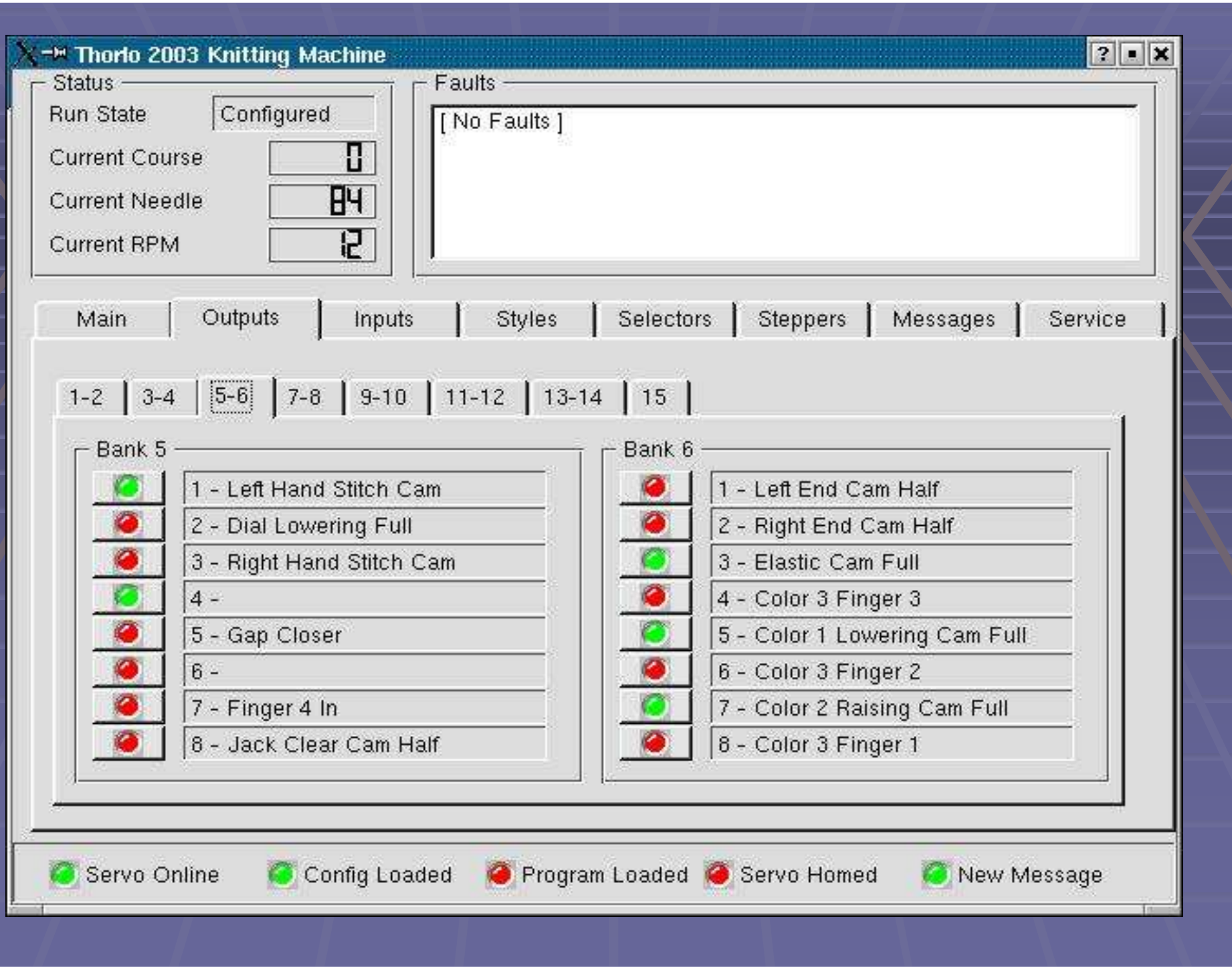
Technologies Employed

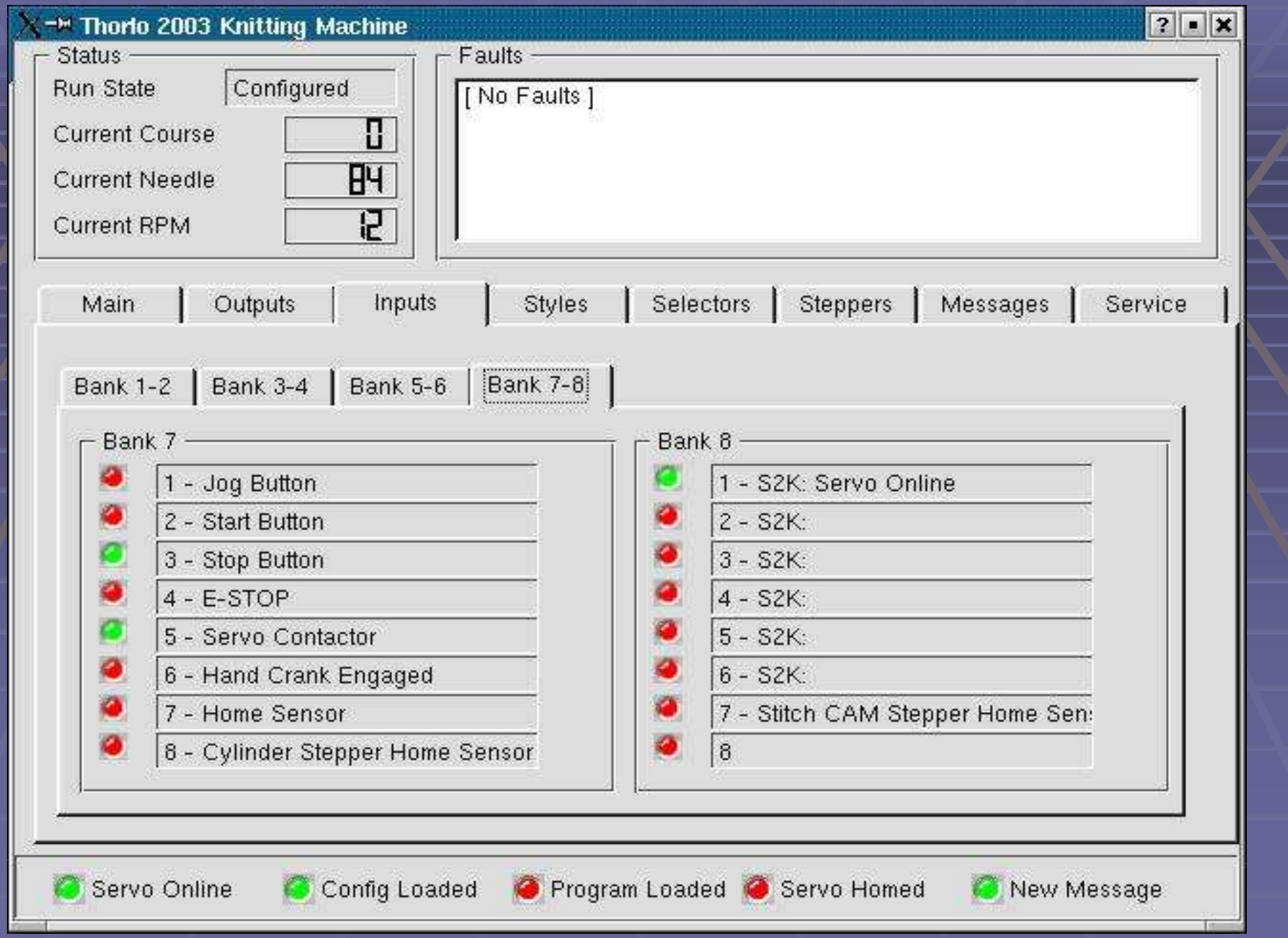
- C/C++
- Assembler
- VHDL
- Device Driver development
- TCP/IP Sockets
- KDE/QtLib
- XWindows
- SQL
- Schematic Capture
- PCB Layout/Design
- Motion Control (Servo/Steppers)
- RTOS
- Multi-threaded application design
- Concurrency / Interprocess communication issues


Thorlo 2003 Knitting Machine
Status
Run State
Configured
Current Course
0
Current Needle
84
Current RPM
12
Faults
[No Faults]
Main
Outputs
Inputs
Styles
Selectors
Steppers
Messages
Service
Style
Description
Created
Modified
dummy program
dummy program
20031027
20031027
ggg
ggg
20031027
20031027
mdf2112
New style
20031117
20031117
new
new
20031027
20031027
new2
New style
20031117
20031117
sel1
sel1
20031027
20031027
slow2
slow2
20031027
20031027
slow3
New style
20031117
20031117
test3
New style
20031117
20031117
www
www
20031027
20031027
xxd
New style
20031117
20031117
Load Style
Servo Online
Config Loaded
Program Loaded
Servo Homed
New Message


Thorlo 2003 Knitting Machine
Status
Run State
Configured
Current Course
0
Current Needle
84
Current RPM
12
Faults
[No Faults]
Main
Outputs
Inputs
Styles
Selectors
Steppers
Messages
Service
Step
Course
Needle
Instruction
1 0 0 ON [8.1] Left Dropper
2 0 30 Needle 30
3 0 30 ON [6.1] Left End Cam Half
4 0 30 Selector4 Data 0xf0
5 0 75 Needle 75
6 0 75 ON [10.5] Left End Cam Full
7 0 75 Speed = 40 RPM
8 0 75 ***** Initialize Faults *****
9 0 75 Enable [2.5] BTSR Fault
10 0 75 Enable [1.7] Elastic Yarn
11 0 75 Enable [2.1] Torque Limiter
12 0 75 Enable [4.7] Broken Dial Needle
Servo Offline
Lock Handle
Download Style
Home Servo
Clear Soft Faults
Shutdown
Servo Online
Config Loaded
Program Loaded
Servo Homed
New Message


Thorlo 2003 Knitting Machine
Status
Run State
Configured
Current Course
0
Current Needle
84
Current RPM
12
Faults
[No Faults]
Main
Outputs
Inputs
Styles
Selectors
Steppers
Messages
Service
Stitch Sizing Stepper
Setpoint 1
Setpoint 2
Setpoint 3
Setpoint 4
Setpoint 5
Setpoint 6
Setpoint 7
Setpoint 8
Go
Set
Temporary Sizing Controls
Coarse
Medium
Fine
+
-
Enable
Disable
Stitch
Enable
Home
Move
Cylinder
Enable
Home
Move
Servo Online
Config Loaded
Program Loaded
Servo Homed
New Message


Thorlo 2003 Knitting Machine
Status
Run State
Configured
Current Course
0
Current Needle
84
Current RPM
12
Faults
[No Faults]
Main
Outputs
Inputs
Styles
Selectors
Steppers
Messages
Service
1-2
3-4
5-6
7-8
9-10
11-12
13-14
15
Bank 5
1 - Left Hand Stitch Cam
2 - Dial Lowering Full
3 - Right Hand Stitch Cam
4 -
5 - Gap Closer
6 -
7 - Finger 4 In
8 - Jack Clear Cam Half
Bank 6
1 - Left End Cam Half
2 - Right End Cam Half
3 - Elastic Cam Full
4 - Color 3 Finger 3
5 - Color 1 Lowering Cam Full
6 - Color 3 Finger 2
7 - Color 2 Raising Cam Full
8 - Color 3 Finger 1
Servo Online
Config Loaded
Program Loaded
Servo Homed
New Message


Thorlo 2003 Knitting Machine
Status
Run State
Configured
Current Course
0
Current Needle
84
Current RPM
12
Faults
[No Faults]
Main
Outputs
Inputs
Styles
Selectors
Steppers
Messages
Service
Bank 1-2
Bank 3-4
Bank 5-6
Bank 7-8
Bank 7
1 - Jog Button
2 - Start Button
3 - Stop Button
4 - E-STOP
5 - Servo Contactor
6 - Hand Crank Engaged
7 - Home Sensor
8 - Cylinder Stepper Home Sensor
Bank 8
1 - S2K: Servo Online
2 - S2K:
3 - S2K:
4 - S2K:
5 - S2K:
6 - S2K:
7 - Stitch CAM Stepper Home Sen
8
Servo Online
Config Loaded
Program Loaded
Servo Homed
New Message