



---

# Running Foundation from a Makefile

---

# Introduction

---

In this document, I will describe how I use a makefile to compile VHDL code into bitstreams for XILINX FPGAs and CPLDs. The makefile sets up the options and runs the individual software tools in Foundation. This gives me the following benefits:

**Automation:** I can start a makefile script running and leave. I no longer have to guide Foundation through the phases of compilation using the GUI.

**Fewer Errors:** It's always possible to make an error or forget to set an option when I point and click with the Foundation GUI. The makefile encapsulates the options for my design and they are consistently applied whenever I recompile.

**Simpler Archiving:** I can use a revision control system to archive my design just by checking-in the makefile, VHDL files, constraint files, and a few files containing the design options. I don't have to store any strange binary files which are incompatible between different versions of the software tools, and I don't need to store the entire project directory hierarchy.

That said, there are disadvantages to using a makefile when I am developing the *first version* of a design. The non-interactive nature of makefiles makes error reporting and correction of my VHDL files more difficult. And it is difficult to remember the correct settings for all the compilation options. So I like to use the Foundation GUI when I develop a design. Once the design is working, I move all the settings for the options into the appropriate places in my scripts.

## Caveats

---

I am assuming you have the following UNIX-like utilities on your PC: `make`, `rm`, `cat`, `echo`, and `mv`. These are necessary to run the makefile. You can get these utilities from several sites on the Web.

You will also need the `fe_shell` utility that lets you run the FPGA Express synthesizer from a command script. This utility is not included in the Student Edition of the XILINX Foundation tools.

# Makefile Design Flow

---

Here is a high-level view of how I use makefiles to compile a bitstream:

1. I edit the files which store the parameters which control some aspects of the synthesis and implementation process. These files are `fe.fst`, `hitop.ctl`, and `bitgen.ut`.
2. I edit the makefile which controls the overall sequence of compilation steps. This typically involves specifying the names of the VHDL source files and libraries and selecting the type of device which will be targeted.
3. I invoke the makefile. The makefile starts FPGA Express to synthesize the VHDL into an XNF file (XILINX netlist format file). Then the makefile runs the appropriate design implementation tools to generate either an FPGA .BIT file or a CPLD .SVF file.

## The Makefile

A makefile for compiling a simple VHDL design for an XC4000 FPGA and an XC9500 CPLD is shown below. It looks complicated. XILINX built their Foundation GUI to hide all this from you. I will explain what each line does below.

---

```
1  FEXP = C:/fndtn/synth/bin-wi~1/fe_shell.exe
2  FEXP_FST_TEMPLATE = fe.fst
3  CTL_TEMPLATE = hitop.ctl
4  TOP = leddcd
5
6  all: xs40-005x1 xs95-108
7
8  clean:
9      rm -f *.log *.bld *.mrp *.ngd *.pro *.pad \
10     *.pcf *.xbt *.jed *.ngm *.tsp *.ngo *.gyd \
11     *.lst *.vm6 *.cmd *.ncd *.dly *.bgn *.drc \
12     *.ll *.rpt *.par
13
14  extra_clean: clean
15     rm -rf $(TOP)
```

```

16         rm -rf DPM_NET
17         rm -f *.bit
18         rm -f *.svf
19
20     xs40-005x1: FEXP_TARGET = XC4000XL
21     xs40-005x1: FEXP_DEVICE = 4005XLPC84
22     xs40-005x1: DEVICE = xc4005x1-3-pc84
23     xs40-005x1: BIT_FILE = leddcd05x.bit
24     xs40-005x1: FST_FILE = fe05x1.fst
25     xs40-005x1: VHDL = "{leddcd.vhd}"
26     xs40-005x1: LIB = "xsboard"
27     xs40-005x1: LIB_VHDL = "{xsboard.vhd}"
28     xs40-005x1: FPGA_UCF = leddcd40.ucf
29     xs40-005x1:
30         $(mk_fpga_fst_file)
31         $(mk_fpga)
32         rm -f $(FST_FILE)
33
34     xs95-108: FEXP_TARGET = XC9500
35     xs95-108: FEXP_DEVICE = 95108PC84
36     xs95-108: CTL_DEVICE = XC95108-15-PC84
37     xs95-108: DEVICE = XC95108
38     xs95-108: SVF_FILE = leddcd108.svf
39     xs95-108: FST_FILE = fe108.fst
40     xs95-108: VHDL = "{leddcd.vhd}"
41     xs95-108: LIB = "xsboard"
42     xs95-108: LIB_VHDL = "{xsboard.vhd}"
43     xs95-108: CPLD_UCF = leddcd95.ucf
44     xs95-108:
45         $(mk_cpld_fst_file)
46         $(mk_cpld)
47         rm -f $(FST_FILE)
48
49
50     define mk_fpga_fst_file
51     echo "set proj $(TOP)" > $(FST_FILE)
52     echo "set vhdl $(VHDL)" >> $(FST_FILE)
53     echo "set top $(TOP)" >> $(FST_FILE)
54     echo "set chip $(TOP)" >> $(FST_FILE)
55     echo "set target $(FEXP_TARGET)" >> $(FST_FILE)
56     echo "set device $(FEXP_DEVICE)" >> $(FST_FILE)
57     echo "set lib $(LIB)" >> $(FST_FILE)

```

```

58 echo "set lib_vhdl $(LIB_VHDL)" >> $(FST_FILE)
59 echo "proj_fsm_coding_style = \"onehot\"" >> $(FST_FILE)
60 cat $(FEXP_FST_TEMPLATES) >> $(FST_FILE)
61 endif
62
63 define mk_fpga
64 $(FEXP) -f $(FST_FILE)
65 ngdbuild -p $(DEVICE) -uc $(FPGA_UCF) -dd . -nt on \
66     DPM_NET/$(TOP).xnf $(TOP).ngd
67 map -p $(DEVICE) -o map.ncd $(TOP).ngd $(TOP).pcf
68 par -x -w -ol 2 -d 0 map.ncd $(TOP).ncd $(TOP).pcf
69 bitgen $(TOP).ncd -l -w -f bitgen.ut
70 mv $(TOP).bit $(BIT_FILE)
71 endif
72
73 define mk_cpld_fst_file
74 echo "set proj $(TOP)" > $(FST_FILE)
75 echo "set vhdl $(VHDL)" >> $(FST_FILE)
76 echo "set top $(TOP)" >> $(FST_FILE)
77 echo "set chip $(TOP)" >> $(FST_FILE)
78 echo "set target $(FEXP_TARGET)" >> $(FST_FILE)
79 echo "set device $(FEXP_DEVICE)" >> $(FST_FILE)
80 echo "set lib $(LIB)" >> $(FST_FILE)
81 echo "set lib_vhdl $(LIB_VHDL)" >> $(FST_FILE)
82 echo "proj_fsm_coding_style = \"binary\"" >> $(FST_FILE)
83 cat $(FEXP_FST_TEMPLATES) >> $(FST_FILE)
84 endif
85
86 define mk_cpld
87 $(FEXP) -f $(FST_FILE)
88 ngdbuild -p $(DEVICE) -uc $(CPLD_UCF) -dd . -nt on \
89     DPM_NET/$(TOP).xnf $(TOP).ngd
90 echo DEVICE_OPTIONS: $(CTL_DEVICE) > $(TOP).ctl
91 cat $(CTL_TEMPLATES) >> $(TOP).ctl
92 hitop -f $(TOP).ngd -s -o $(TOP)
93 hplusas6 -i $(TOP) -s -a -l $(TOP).log -o $(TOP)
94 hprep6 -i $(TOP) -r jed -a
95 echo part $(DEVICE):$(TOP) > $(TOP).cmd
96 echo program $(TOP) -j $(TOP) >> $(TOP).cmd
97 echo quit >> $(TOP).cmd
98 rm -f $(TOP).SVF
99 jtagprog -svf -batch $(TOP).cmd

```

```
100 mv $(TOP).SVF $(SVF_FILE)
101 rm -f $(TOP).cmd
102 rm -f $(TOP).ctl
103 endif
```

---

Lines 1-2: The `FEXP` variable is defined that points to the location of my `fe_shell` executable. (Yours may be in a different place or you might not even have this program if you are using the Student Edition of Foundation.) `fe_shell` is a program that lets you control the FPGA Express synthesis tool using text commands. It will also execute a sequence of commands stored in a script file. The `FEXP_FST_TEMPLATE` variable points to the basic synthesis script template that I use. I keep a separate synthesis script template in each of my project directories in case I need different synthesis options for a particular project.

Line 3: The `CTL_TEMPLATE` variable points to the file which stores the options that control the process of fitting the design into a CPLD. I keep a separate CPLD control template in each of my project directories in case I need different fitting options for a particular project.

Line 4: The `TOP` variable stores the name of the top-level module in my VHDL design. This name is used as the base name for a lot of the files generated during the compilation process.

Line 6: I list my targets here. In this example I am targeting an XS40-005XL Board (which has an XC4005XL FPGA on it) and an XS95-108 Board (which has an XC95108 CPLD on it). These are just my personal choices - you can use any names for targets. To compile the bitfiles for both the XS40-005XL Board and the XS95-108 Board, I just issue the command `make all`.

Lines 8-18: These are two other targets I use to make it easy to remove all the files generated during the compilation process. To remove all the extra files generated in my top-level directory, I type `make clean`. To start from a clean

slate I type `make extra_clean` which removes all the extra files, subdirectories, and previously-compiled bitstream files.

Lines 20-28: These lines set the variables for compiling a bitstream for the XS40 Board. Each line starts with the name of one of the targets from line 6 (`xs40-005x1` in this case). The `FEXP_TARGET` variable lists the device family that the FPGA Express synthesizer will be targeting, while the `FEXP_DEVICE` variable indicates the particular device within that family. The `FST_FILE` variable stores the name of the file that holds the command script for the synthesizer. The `DEVICE` variable tells the Foundation implementation tools the which device, speed grade, and package type they will be targeting. In this example, the target is a -3 speed XC4005XL FPGA in an 84-pin PLCC. (The `DEVICE` variable seems redundant given that we already have this information in the `FEXP_TARGET` and `FEXP_DEVICE` variables, but the variables are intended for two separate tools and I don't have a simple way to generate one from the other.) The `BIT_FILE` variable indicates the file where I want the final bitstream stored. The `VHDL` variable lists the names of the VHDL source files in my design separated by spaces. (There is only the `leddcd.vhd` VHDL file in this design.) The `LIB` variable lists the name of the master library for my design. The `LIB_VHDL` variable holds the list of VHDL source files that are to be included in the master library. (If you don't use a library in your design, just set `LIB` to the string `\"`.) The `FPGA_UCF` variable points to the file that stores the pin assignments and timing constraints for my design.

Lines 29-32: The actual synthesis and implementation for the FPGA occur on these lines. `$(mk_fpga_fst_file)` executes a procedure defined in the makefile (lines 50-61) that generates the script file for controlling the FPGA Express synthesizer. Then `$(mk_fpga)` executes another procedure defined in the makefile (lines 63-71) that runs the synthesizer and the implementation tools to generate a bitstream. The last line removes the script that controls the synthesizer.

Lines 34-43: These lines set the variables for compiling a bitstream for the XS95 Board. Each line starts with the name of one of the targets from line 6 (`xs95-108` in this case). The `FEXP_TARGET` variable lists the device fam-

ily that the FPGA Express synthesizer will be targeting, while the `FEXP_DEVICE` variable indicates the particular device within that family. The `FST_FILE` variable stores the name of the file that holds the command script for the synthesizer. The `CTL_DEVICE` variable tells the Foundation implementation tools the which device, speed grade, and package type they will be targeting. In this example, the target is a -15 ns XC95108 CPLD in an 84-pin PLCC. The `DEVICE` variable names the generic family of devices that will be targeted by the implementation tools. (The `CTL_DEVICE` and `DEVICE` variables seem redundant given that we already have this information in the `FEXP_TARGET` and `FEXP_DEVICE` variables, but the variables are intended for two separate tools and I don't have a simple way to generate one from the other.) The `SVF_FILE` variable indicates the file where I want the final bitstream stored. The `VHDL` variable lists the names of the VHDL source files in my design separated by spaces. (There is only the `leddcd.vhd` VHDL file in this design.) The `LIB` variable lists the name of the master library for my design. The `LIB_VHDL` variable holds the list of VHDL source files that are to be included in the master library. (If you don't use a library in your design, just set `LIB` to the string `\"`.) The `CPLD_UCF` variable points to the file that stores the pin assignments and timing constraints for my design.

Lines 44-47: The actual synthesis and implementation for the CPLD occur on these lines. `$(mk_cpld_fst_file)` executes a procedure defined in the makefile (lines 73-84) that generates the script file for controlling the FPGA Express synthesizer. Then `$(mk_cpld)` executes another procedure defined in the makefile (lines 86-103) that runs the synthesizer and the implementation tools to generate a bitstream. The last line removes the script that controls the synthesizer.

Lines 50-61: The `mk_fpga_fst_file` procedure is defined on these lines. This procedure generates the script file that controls the operations of the FPGA Express synthesizer. The first eight lines of this procedure just set up the variables in the synthesizer script file pointed to by the `FST_FILE` variable. These variables define the project name, VHDL files, top-level module name, chip name, targeted FPGA family, FPGA family device, VHDL master library, and master library VHDL source files. Then the coding style

for finite-state machines is set to `onehot` (which is usually the best setting for FPGAs). Finally, the commands in the `FEXP_FST_TEMPLATE` file are appended to the file pointed to by `FST_FILE`. At this point, a complete synthesis script for an FPGA exists in the file pointed to by `FST_FILE`.

Lines 63-71: These lines define the `mk_fpga` procedure. The first line of the procedure passes the synthesis script to the FPGA Express synthesizer. After the synthesizer completes its operations, the next line in the procedure activates the `ngdbuild` implementation tool. `ngdbuild` converts the XILINX netlist file created in `DPM_NET/$(TOP).xnf` by the synthesizer into the NGD format that is understood by the rest of the implementation tools. Then the `map` tool is activated that maps the circuitry described in the `$(TOP).ngd` file to the architecture of the FPGA device. The mapping is passed to the `par` tool on the next line which performs a place-and-route of the circuitry. Finally, the bitstream for the routed circuit is generated by the `bitgen` tool under the influence of the options stored in the `bitgen.ut` file. The bitstream file is moved into the file pointed to by the `BIT_FILE` variable. (I usually get these command lines from the `fe.log` file in the `xproj/veri/revj` subdirectory after I have finished developing the design using the Foundation GUI.)

Lines 73-84: The `mk_cp1d_fst_file` procedure is defined on these lines. This procedure generates the script file that controls the operations of the FPGA Express synthesizer. The first eight lines of this procedure just set up the variables in the synthesizer script file pointed to by the `FST_FILE` variable. These variables define the project name, VHDL files, top-level module name, chip name, targeted CPLD family, CPLD family device, VHDL master library, and master library VHDL source files. Then the coding style for finite-state machines is set to `binary` (which is usually the best setting for CPLDs). Finally, the commands in the `FEXP_FST_TEMPLATE` file are appended to the file pointed to by `FST_FILE`. At this point, a complete synthesis script for an CPLD exists in the file pointed to by `FST_FILE`.

Lines 86-103: These lines define the `mk_cp1d` procedure. The first line of the procedure passes the synthesis script to the FPGA Express synthesizer. After the synthesizer completes its operations, the next line in the procedure acti-

vates the `ngdbuild` implementation tool. `ngdbuild` converts the XILINX netlist file created in `DPM_NET/$(TOP).xnf` by the synthesizer into the NGD format that is understood by the rest of the implementation tools. Next, a control file that holds the various CPLD fitting parameters is created in the file with the name `$(BASE).ctl`. The first line of the control file specifies the type of CPLD that is being targeted. Then the options stored in the file pointed to by `CTL_TEMPLATE` are appended to the control file. The control file directs the operations of the `hitop`, `hplusas6`, and `hprep6` tools on the next three lines. (I usually get these three lines from the `fe.log` file in the `xproj/veri/revj` subdirectory after I have finished developing the design using the Foundation GUI.) The resulting JEDEC file output by `hprep6` must be converted to an SVF bitstream file. `jtagprog` is directed to generate the SVF file for given type of CPLD and then terminate by the three commands stored in the `$(TOP).cmd` file. The bitstream file is moved into the file pointed to by the `SVF_FILE` variable. Then the command and control files which were created previously are removed.

## The Synthesis Script Template (`fe.fst`)

The synthesis script template is stored in a file pointed to by the `FEXP_FST_TEMPLATE` variable in the makefile. The template contains synthesis commands which don't usually change from one design to another. An example script template that I often use is shown below.

---

```
1  set export_dir DPM_NET
2
3  file delete -force $proj
4
5  create_project -dir . $proj
6
7  open_project $proj
8
9  proj_export_timing_constraints = "no"
10 proj_fsm_when_others = "safest"
11 proj_xlx_ppr = "M1"
12 proj_default_clock_frequency = 50
13
```

```
14  if {$lib != ""} {
15      create_library $lib
16      foreach i $lib_vhdl {
17          add_file -library $lib -format VHDL $i
18      }
19  }
20
21  foreach i $vhdl {
22      add_file -format VHDL $i
23  }
24
25  analyze_file -progress
26
27  create_chip -progress -target $target -device $device -name $chip $top
28
29  current_chip $chip
30
31  set opt_chip $chip-Optimized
32
33  optimize_chip -progress -name $opt_chip
34
35  list_message
36
37  file delete -force $export_dir
38  file mkdir $export_dir
39
40  export_chip -progress -dir $export_dir
41
42  close_project
43
44  exit
```

---

This script is adapted to various designs when the makefile prepends the assignments for the proj, vhdl, top, chip, target, device, lib, and lib\_vhdl variables. The template then performs the following functions:

Line 1: This line specifies the subdirectory where FPGA Express will place its synthesis results. In this example, the subdirectory is hard-coded to DPM\_NET.

Lines 3-7: Any existing project with the same name as that stored in the `proj` variable will be deleted. This removes any old files that could affect the current synthesis process. Then a new project with the name stored in the `proj` variable is created and opened.

Line 9: FPGA Express is directed not to export any timing constraints from the synthesis process on to the implementation tools.

Line 10: When VHDL is synthesized, all state machines are coded to transition to a safe state if an unspecified state is ever entered.

Line 11: The place-and-route implementation tools is specified to be the one found in the XILINX M1 tools.

Line 12: The minimum operational frequency for the synthesized design is specified to be 50 MHz.

Lines 14-19: If the master library name in `lib` is not blank, then the VHDL source files listed in the `lib_vhdl` variable are each added to the master library.

Lines 21-23: The VHDL source files listed in the `vhdl` variable are added to the project.

Line 25: The VHDL files are analyzed for syntax errors. Any syntax errors will cause the script to terminate.

Line 27: The VHDL source is synthesized into a netlist for the given target family and device. The `top` variable tells the synthesizer which module is at the top of the design hierarchy. The synthesized netlist is associated with a chip having the name stored in the `chip` variable.

Line 29: The synthesized chip is specified as the current chip that will be optimized.

Line 31: The variable `opt_chip` is declared and set to the name of the current chip with a suffix of `"-Optimized"`.

Line 33: The synthesized netlist output by the operations on line 27 are optimized for the target device architecture.

Line 35: This line directs the FPGA Express to output various status messages which occur during the synthesis process.

Lines 37-38: The directory where the optimized netlist will be exported is deleted (to remove any old netlists) and then recreated.

Line 40: The netlist from the optimization step is exported as an XNF file to the directory named in `export_dir`.

Line 42: The current project is closed.

Line 44: The script terminates.

### **The Bitstream Generator Control File (`bitgen.ut`)**

This file specifies the options that control the FPGA bitstream generator. `bitgen.ut` is not used when I generate SVF files for CPLD devices. An example of a `bitgen.ut` file is shown below:

---

```
1  -g ConfigRate:SLOW
2  -g TdoPin:PULLNONE
3  -g M1Pin:PULLNONE
4  -g DonePin:PULLUP
5  -g CRC:enable
6  -g StartUpClk:CCLK
7  -g SyncToDone:no
8  -g DoneActive:C1
9  -g OutputsActive:C3
10 -g GSRInactive:C4
11 -g ReadClk:CCLK
12 -g ReadCapture:enable
13 -g ReadAbort:disable
14 -g M0Pin:PULLNONE
15 -g M2Pin:PULLNONE
```

---

In general, I usually copy the `bitgen.ut` file generated by the Foundation GUI after I finally have the design development completed. The file is usually found in the `xproj/veri/revj` subdirectory where *i* and *j* are the version and revision number of my final design.

### The CPLD Fitter Control Template File (`hitop.ct1`)

This file specifies the options which control the CPLD fitting process. It is not used if I am only targeting FPGA devices. An example of a CPLD fitter control file is shown below:

---

```
1 DT_SYNTHESIS:TRUE
2 MC9500_INPUT_LIMIT:36
3 GSR_OPT:TRUE
4 FASTCLOCK_OPT:TRUE
5 FOE_OPT:TRUE
6 MC9500_PTERM_LIMIT:20
7 TIMING_OPT:TRUE
8 LOWPWR:STD
9 SLEW:FAST
10 DRIVE_UNUSED_IO:FALSE
11 IGNORE_PIN_ASSIGNMENT:FALSE
12 FM_PARTITION:TRUE
13 CREATE_LOCAL_FEEDBACK:TRUE
14 CREATE_PIN_FEEDBACK:TRUE
15 IGNORE_TSPEC:FALSE
16 MULTI_LEVEL_LOGIC_OPTIMIZATION:TRUE
```

---

In general, I usually copy the `hitop.ct1` file generated by the Foundation GUI after I finally have the design development completed. The file is usually found in the `xproj/veri/revj` subdirectory where *i* and *j* are the version and revision number of my final design.

## LED Decoder Design Example

---

The example design included with this document consists of the following files:

`ledccd.vhd`: This is the top-level module of a circuit which accepts a four-bit input and displays the corresponding hexadecimal digit on a 7-segment LED.

`xsboard.vhd`: This is a library file that contains several modules, one of which is an LED decoder used by the top-level `ledccd` module.

`ledccd40.ucf`: This file contains the pin assignments for the `ledccd` design when it is targeted to an XS40 Board.

`ledccd95.ucf`: This file contains the pin assignments for the `ledccd` design when it is targeted to an XS95 Board.

`fe.fst`: This is the FPGA Express script template that was discussed earlier.

`hitop.ct1`: This is the CPLD fitter control file that was discussed earlier.

`bitgen.ut`: This is the bitstream generator control file that was discussed earlier.

`makefile`: This is the makefile that compiles the `ledccd` design for both the XS40 and XS95 Boards.

Just place these files into a directory. The LED decoder is compiled for both the XS40 and XS95 Boards by issuing the following command in a DOS window:

```
C:> make all
```

You can compile the design for either the XS40 or XS95 Board with the commands:

```
C:> make xs40-005x1
```

```
C:> make xs95-108
```

After the makefile is done, you will find the FPGA bitstream in `ledcd05x.bit` and the CPLD bitstream in `ledcd108.svf`.

After compiling the design, you can clean the directory with the command:

```
C:> make clean
```

To clean the directory and remove the subdirectories and the bitstreams, use the command:

```
C:> make extra_clean
```

## Modifying the Files for Your Own Designs

---

Here is a summary of the steps I go through to develop a makefile and script files for a design:

1. Develop the initial version of the design using the Foundation GUI.
2. Create an empty directory for the project scripts and copy the `makefile` and `fe.fst` files from the script directory of a previous project. (You could use the `makefile` and `fe.fst` files from the LED decoder example as your starting point.) Then copy the `bitgen.ut` and `hitop.ctl` files from the `xproj/veri/revj` directory created in step 1.
3. Set the following variables for each target architecture in the makefile:

`TOP`: Set it to the name of the top-level module of the design.

`FEXP_TARGET`: Set it to one of the family of devices recognized by FPGA Express.

`FEXP_DEVICE`: Set it to one of the members of the device family.

`DEVICE`: Set it to the detailed name of the FPGA device (e.g. `DEVICE = xc4010e-3-pc84`) or the shortened name of the CPLD device (e.g. `DEVICE = XC95108`).

`CTL_DEVICE`: For CPLD designs, set it to the detailed name of the target device (e.g. `CTL_DEVICE = XC95108-15-PC84`). This variable is not used in FPGA designs.

`BIT_FILE/SVF_FILE`: Set these to the name of the final BIT or SVF file.

`VHDL`: Set it to a space-separated list of VHDL files for the design (e.g. `VHDL = "{file1.vhd file2.vhd}"`).

**LIB:** Set it to the name of the master library used in the design or make it empty if no library is needed (i.e. LIB = "").

**LIB\_VHDL:** Set it to the list of files to be included in the library using the same space-separated format as with the VHDL variable.

**FPGA\_UCF/CPLD\_UCF:** Set these to the names of the pin assignment files for the FPGA and CPLD designs.

4. Edit fe.fst to change or add FPGA Express settings.
5. Edit hitop.ctl to adjust fitter options for the CPLD.
6. Edit bitgen.ut to adjust bitmap generation options for the FPGA.

## “It Doesn’t Work!”

---

You may encounter problems when you try to compile the LED decoder example or designs of your own. You can usually solve VHDL syntax errors or problems with option settings by first compiling the design using the Foundation GUI. Then you can extract the correct VHDL and the control options from the debugged project and include them in your makefile and template files.

If you just can’t stand to use the Foundation GUI (i.e. you started with the UNIX command line interface and you intend to die that way), then you will have to run the makefile and watch for error messages as they appear and then go into the offending file and fix it. It’s not impossible to do, but it’s not the most efficient use of your time. Any way, it’s your call so do what you want.

You may encounter problems you don’t immediately understand or whose solution is not immediately obvious. First I will tell you what *not* to do: *do not send email to XESS Corp. requesting help!* I have received a few too many messages over the years which read like this:

“I do as you say but when I run makefile it no working. What is wrong? Please help me!”

Please! If I could fix problems based on a description like that, then I would make my living as a psychic.

There are better ways to approach the problem. The first is to say:

“Well, I didn’t know how to use a makefile to run the XILINX tools before, so if this script doesn’t work I’m no worse off than I was. I’ll just chuck it and go back to doing it the way I used to.”

Surprisingly, this is usually a pretty good response. But for those who won’t give up, you can employ the following attitude:

“Well, the makefile isn’t running but I’ll bet it’s 95% of the way there. I’m probably smarter than the guy that wrote it so I should be able to figure out the last 5% and make it work for me.”

Then if you solve the problem, feel free to send me a good description of how the problem occurred and how you solved it. If I ever revise this document, I will include your insights so that others can benefit from them.

## Future Enhancements

---

The makefile and scripts I have described are simple but they get the job done for me. Their most noticeable deficiency is that they don't use the file dependency features of `make`. So all the files are recompiled from scratch whenever you run `make`. It would be more efficient if the makefile recognized the existence of intermediate files and could resume the compilation process from that point. Maybe someone else will make that enhancement.