Configurator: A Configuration Program
for Satellite Dish Hardware
by
Mark Paul Hurst
S.B., Massachusetts Institute of Technology (1994)
Submitted to the Department of Electrical Engineering and Computer Science in partial
fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1995
© Mark P. Hurst, MCMXCV. All rights reserved.
The author hereby grants to MIT permission to reproduce and to distribute copies of this
thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science
Jan 27, 1995

Certified by

Professor M. Frans Kaashoek
Thesis supervisor

Certified by

Mike J. Skeith
Assistant Vice President, Hughes Network Systems
Company supervisor

Certified by

Professor Frederic R. Morgenthaler
Committee on Graduate Students
Installation of satellite dishes requires certain procedures to be run on the communications hardware. The installer uses a configuration program from a PC to perform these procedures. The configuration program sends commands to the target hardware through a serial port to read and write locations in target memory. In addition to setting configuration parameters, some monitor and control functions may be run.

This thesis describes the Configurator, a configuration program which replaces the ones used previously by installers. The user interfaces of previous configuration programs were text-based and poorly designed. The Configurator allows installers to configure different targets with a consistent, well-designed graphical user interface. The Configurator offers installers additional features, such as specialized calculation screens.

The code of the Configurator is designed such that periodic updates (to account for changing targets) are easy. Also, new Configurator instantiations may be built for new targets with little difficulty. Each program in the family of Configurators shares a common core, which provides functionality like the user interface. Placing as much functionality as possible in the core ensures that consistency is maintained between Configurators. Functionality which is specific to a particular target is contained in target-specific code. Target-specific code takes the form of statically defined tables and specially written functions. Using tables allows lab technicians to maintain a Configurator without relying on software engineers.

Thesis Supervisor: Professor M. Frans Kaashoek, Massachusetts Institute of Technology
Company Supervisor: Mike J. Skeith, Assistant Vice President, Hughes Network Systems
Acknowledgements

¡Pasu mecha! What a ride this has been. I never would have made it without the support of a zillion people. First and foremost, I want to thank my family for being a constant support and faithful cheering section: M&D, K&J&C&G&C, G&G&G&G, Aunt Pat, and the best dog in the world, Captain. Thanks to Phil for being a great roommate, marathon partner, and friend. Thanks to Stephen, Brian, and Mike for all the great times of xconq, Nintendo, and hangin' out in general. And thanks to all my Peruvian friends (Cecilia, Walter, Yani, Carmen, Lisbeth, Pati) for letting me teach, and for teaching me.

Hughes has been a great place to work. Thanks to all the folks in the Software Technology Department, who at various times offered encouragement or wry comments. Thanks to the several managers who guided me through my internships: Carl Symborski, Trevor Eagling, Tom Maricle, and Mike Skeith. Thanks to Stephen Harpster for being my mentor within the department ever since I walked in the door in 1992. Thanks to Jeff for being a cool officemate. Thanks to the guys in Sustaining Engineering for all the testing and useful feedback. Thanks to Bridgette Atkinson and Mark Balzer for supporting me on the human resources side, and thanks to John Kenyon, who called me up three years ago and hired me. I still have no idea where he found my resumé. Overall, I want to thank Hughes Network Systems for having the initiative to join VI–A and bring me in as their very first master’s thesis intern. I wish the best of luck to the VI–A students who come after me.

Thanks to the VI–A program for providing me with such a great opportunity for work and research. Thanks also to Prof. Frans Kaashoek, who volunteered his time to advise me on the thesis; it was a pleasure working with him. Finally, one huge thank–you goes to John Lacey, who was a patient customer, constructive critic, walking encyclopedia, and in general the easiest person in the world to work with. We should all have the privilege to work with people like John.

This thesis is brought to you by: emacs, Phish, WIRED, Mari Mar, Hard Times Cafe, the two–and–a–half mile mark, a dislike of stupid interfaces and an evil empire, and the faith that I would someday return for a tall, frosty Guinness in the Thirsty Ear Pub. Coming up next: comics on the Web.
## Contents

### Chapter 1
**Introduction** ................................................. 11
- 1.1 Typical Installation ........................................ 11
- 1.2 Previous Configuration Programs ........................ 15
- 1.3 The Installer's PC ........................................ 21
- 1.4 Thesis Overview ........................................... 22

### Chapter 2
**Configuration Requirements** ............................... 23
- 2.1 User Requirements ........................................ 23
- 2.2 IDU Requirements ........................................ 25
- 2.3 Summary .................................................. 36

### Chapter 3
**Implementation** .............................................. 37
- 3.1 Configurator structure ...................................... 37
- 3.2 User interface ............................................. 37
- 3.3 Data types and the core/target-specific interface .... 39
- 3.4 Modules .................................................. 44
- 3.5 Target-specific implementations ........................ 49
- 3.6 Summary .................................................. 54

### Chapter 4
**Related Work and Discussion** ............................. 55
- 4.1 Related Work ............................................... 55
- 4.2 Suggestions for improvement ............................. 61
- 4.3 Summary .................................................. 67

### Chapter 5
**Evaluation** .................................................. 68
- 5.1 User reactions ............................................. 68
- 5.2 Design process ........................................... 68

### Chapter 6
**Conclusion** .................................................. 71

### References .................................................. 72
Figures

Figure 1. An Indoor Unit (IDU). ........................................ 13
Figure 2. An Outdoor Unit (ODU) and an antenna. .................. 13
Figure 3. The installation process. .................................... 15
Figure 4. HES can communicate with TES and PES. ................. 17
Figure 5. The trouble with previous configuration programs. ....... 17
Figure 6. The four main components of the HES IDU. ............... 18
Figure 7. The Term Protocol, a configuration program for the RFM. . 19
Figure 8. Dandy, a configuration program for the CU. ............... 20
Figure 9. A screen from the Configuration Editor. .................... 21
Figure 10. Term Protocol packet format. .............................. 26
Figure 11. Dandy protocol packet format. ............................. 30
Figure 12. CU packet formats. ........................................ 31
Figure 13. Configurators and the field service disk. ................. 37
Figure 14. A typical view of the RFM Configurator. .................. 38
Figure 15. Modules of the Configurator GUI. ......................... 44
Figure 16. Relationship between configuration parameters and NVRAM. 47
Figure 17. The RFM Configurator's KCM Rx Channel calculation. .... 51
Figure 18. The CU Configurator. .................................... 53
Figure 19. The IFM Configurator. .................................... 54
Tables

Table A. RFM Command Set .......................................................... 26
Table B. RFM Configuration Parameters ........................................... 28
Table C. CU Command Set ............................................................. 31
Table D. CU Configuration Parameters ............................................ 33
Table E. IFM Command Set ............................................................ 34
Table F. IFM Configuration Parameters ............................................ 35
Table G. Modules of the GUI .......................................................... 45
Table H. Modules called from the menu bar. ..................................... 49
Table I. RFM-specific modules. ....................................................... 51
Table J. CU-specific modules. ....................................................... 52
Table K. IFM-specific modules. ....................................................... 53
Installing satellite dishes can be expensive. Installers and equipment may have to be transported to a remote part of the globe. Once there, the installers must be paid for time spent doing the installation. Cement must be poured, the dish set in place, and the antenna pointed. Even then the installation is incomplete. There are still some remaining operations left to carry out on the hardware controlling the satellite dish. These operations, which could not be done during manufacturing and so must be done during installation, are collectively known as the configuration of the satellite dish.

This thesis will approach the subject of satellite dish configuration from the perspective of Hughes Network Systems (HNS), the company at which this research was carried out. HNS is an appropriate choice for the research: a leader in telecommunications systems, it spends hundreds of thousands of dollars yearly installing network systems around the globe. The two main network systems sold by HNS, Personal Earth Station (PES) and Telephony Earth Station (TES), currently have installed 60,000 and 10,000 dishes, respectively. As of December 1994, new PES’s are being installed at a rate of 500 a month, and TES’s at a rate of 100 a month. Each installation costs the company between US $1,000 and $10,000 just to pay for the manpower. With those numbers in mind, a project was proposed in June 1994 to create a computer program that could substantially cut the costs of installation. That program, called the Configurator, is the topic of the thesis.

The Configurator provides installers with an easy-to-use interface, which speeds up the installation process and reduces the number of errors made. The code structure makes it easy to modify the program when the target hardware changes. Overall, the Configurator has been a success. It has been used at installation sites in three Asian countries, with positive results each time. It is also in daily use in the hardware development labs at Hughes Network Systems. Incorporating elements of user interface design, configuration management, and satellite technology, the Configurator was an excellent topic for an internship thesis.

This chapter gives the background of the project. Section 1.1 reviews the procedures involved in a typical installation. Section 1.2 briefly describes previous configuration programs and identifies the need for the Configurator. Section 1.3 describes the installer’s PC, and Section 1.4 gives an overview of the thesis.

1.1 Typical Installation
This section describes the procedure involved in a typical satellite dish installation. By understanding the shortcomings of current installation tools, the advantages of the Configurator are more easily ap-
preciated. Before the details, however, a note about terminology: The unit being installed may be referred to as a “satellite dish” in a general sense, but technically the “dish” is just a piece of metal shaped in such a way to serve as an antenna. It is more common to refer to the unit by its product name, such as “PES” or “TES.” In this thesis, however, most discussion will focus on one of the two main components of the unit: the Indoor Unit (IDU) or the Outdoor Unit (ODU). An IDU is shown in Figure 1.
The ODU is the hardware which is physically attached to the antenna (see Figure 2). (The ODU and antenna together are known as a Very Small Aperture Terminal (VSAT), but the ODU is the important component.) The functionality of the ODU varies from system to system; in general, however, it controls the transmission and reception of messages to and from the satellite. The ODU is dedicated to the low-level processing of signals being exchanged with the satellite.

How the ODU is set in place depends on the physical environment. In some installations, installers pour cement for the ODU to sit in; in others, the ODU may be mounted on a pole. It is also necessary
to connect the power supply, point the dish at the satellite, and possibly run some diagnostic proce-
dures. Once it is plugged in and pointing at the satellite, however, there is no need to change anything.
The ODU, for the most part, is non-configurable; its settings and parameters are "hardwired" into
the hardware during manufacturing. In this way the ODU is different from the IDU.

Once the ODU is in place, the installer moves inside the customer's building to install the Indoor Unit
(IDU). Depending on the network system, the building in which the IDU is installed might be an
office building, a factory, a store, or a house (see Section 1.2.1). The IDU is a box containing hard-
ware which handles the satellite communication at a higher level than the ODU does. The IDU also
interfaces with the customer's hardware: telephones, cash registers, etc.

Contained on the IDU's circuit boards is Non-Volatile Random Access Memory (NVRAM). Like
conventional (volatile) RAM, NVRAM may be read and written. Unlike RAM, NVRAM is not
erased when the power is turned off. NVRAM is contained on the IDU so that the installer may write
data to it during installation, and so that those memory locations will not lose their data when power
is cycled. Specifically, NVRAM is used to hold data which could not be set during manufacturing.
Since these parameters vary between installations, they are known as site-specific parameters. For
example, the IDU uses its latitude and longitude to perform a certain calculation. Because each IDU
will be installed at a different geographic location, the latitude and longitude are site-specific param-
eters. "Burning" this information into each IDU's hardware during manufacturing would be as in-
flexible as it would be inefficient. Instead, the installer can write the latitude and longitude to the
IDU's NVRAM during installation. The installer's act of setting these site-specific parameters is
known as the configuration of the IDU (or, from a more general standpoint, the "satellite dish config-
uration," as the thesis is entitled). Likewise, the site-specific parameters are also commonly known
as configuration parameters.

In order to configure the IDU, the installer must have some tool which can communicate with the IDU
to read and write its NVRAM. This tool is a personal computer (PC) running a configuration program,
which is designed to "talk" to the IDU. To use the configuration program, the installer con-
nects the PC to the IDU by a serial line, which connects the two machines' serial ports together (see
Figure 3). The installer then runs the configuration program, which communicates with the IDU
through the serial port. Assuming it can communicate with the IDU, a configuration program may
only be judged on one other factor: its user interface. A good user interface can decrease both the
time it takes and the number of errors made during configuration. Before the Configurator became
available, there were no good configuration programs for any network system being installed for
HNS.
1.2 Previous Configuration Programs

Interestingly enough, this project came about not only because existing user interfaces were poorly designed, but because installers had to learn *too many* interfaces to do their job. Hybrid Earth Station (HES), a new product introduced in spring 1994, required installers to learn three unique interfaces to run the configuration software. Some aspects of the user interface required so much knowledge of the IDU hardware that, instead of contracting the work out to one of the usual installation companies, HNS had to send its own engineers to the remote sites to install the HES’s. Even then the job of using the configuration programs was time-consuming and difficult.

1.2.1 Summary of Network Systems

It is important to understand why there were three configuration programs for the HES. As implied by its name, HES is a hybrid of two existing products: PES and TES. PES and TES are the two most popular network systems sold by HNS, and some customers wanted the functionality of both products inside one box.

1.2.1.1 Personal Earth Station (PES)

PES is a satellite communications system with a star topology. Within the network there is one central *hub*, which manages the network, and many *remote units*, which are the smaller dishes requiring installation as described above. Remote units communicate with each other by sending messages through the hub. A remote-to-remote message undergoes a “double hop” — one trip through the satellite to get to the hub, and one to get to the receiving unit. PES remotes may also communicate with the hub, in which case messages only hop once. PES is mainly used by geographically distrib-
uted companies who need data communication from individual stores or offices to a central head-
quarters, where the hub is located. Holiday Inn is one such customer of PES.

1.2.1.2 Telephony Earth Station (TES)
TES, a newer product than PES, implements a mesh topology. There is no hub; therefore, every TES
unit in the network is a remote unit. Every message is a remote-to-remote message and only under-
goes a single bounce en route to the receiving unit. The advantages of TES are that without a hub,
the network is less cumbersome than PES, and messages take roughly half the time to travel. Because
of the reduced latency, TES is well-designed for voice communications (hence the word "Telepho-
ny" in the name). A typical customer of TES is a locally distributed company in a place (a city or
small country, say) in which there is no other reliable means of communication. For example, one
customer bought a TES unit for each of its offices in Singapore. The customer had essentially bought
its own telephone network.

1.2.1.3 Hybrid Earth Station (HES)
Recently some customers who wanted the advantages of both systems bought both PES and TES
equipment. This created an awkward situation. Whenever the customer placed a PES and a TES in
the same place (on the same office building, for example), there was some redundancy in the two sets
of equipment communicating with the same satellite. This was an expensive redundancy, and it was
keeping some customers from buying both systems.

Thus, Hybrid Earth Station (HES) was created. HES connects the two IDU's of PES and TES to one
ODU, through which the IDU's may communicate with the satellite (see Figure 4). The HES IDU
is simply a box containing the PES and TES IDU's. With few exceptions, the original PES and TES
IDU hardware remained unchanged in the transition to HES. (At any rate, it is beyond the scope of
this document to discuss the changes, so the thesis will consider the PES and TES IDU's to be the
same hardware in the IDU.)

Each of the IDU's is comprised of two main components. A TES IDU is made up of a Radio Frequen-
cy Module (RFM) and several Channel Units (CU), and a PES IDU is comprised of an Intermediate
Frequency Module (IFM) and several Port Cards. Collectively, these four components make up the
HES IDU.

1.2.2 Three Configuration Programs
Three of the four HES IDU components require configuration by the installer. The RFM, CU, and
IFM are configured by the installer with programs called the Term Protocol, Dandy, and PES Config-
uration Editor (CE), respectively. The Port Cards are configured by the hub (via the satellite), so the
installer does not have to work with them during installation. Figure 6 shows this information graphically.

To install an HES, an installer would have to work with the three different interfaces of the Term Protocol, Dandy, and the PES CE. Making matters worse is that none of the configuration programs are designed very well (see Figure 5). The following briefly describes the shortcomings of each program.
1.2.2.1 The Term Protocol (RFM)

The Term Protocol, used for configuring the RFM, is the worst of the three configuration programs, mainly because it is not much of a program at all. Running the program basically turns the PC into a dumb terminal through which the user can type commands directly to the RFM. All the program does is encapsulate the commands in a certain format and send them through the serial port. (This encapsulation is the “Term Protocol” from which the program gets its name.) Therefore, the interface is command-driven, and the user's command set is simply the commands accepted by the RFM. The interface is extremely rigid, and in some cases even a small mistake by the installer can cause serious problems. Here are some examples of Term Protocol commands: “DF020”, “Q0F3”, “AF32 0F 2B C3”. Because the interface is so low-level, the Term Protocol user must have a working knowledge of the RFM’s structure. More importantly, the user must be comfortable with hexadecimal arithmetic. This alone was a deciding factor in many of the cases in which HNS engineers had to be sent to TES or HES installations. Figure 7 shows the user interface of the Term Protocol.

One may wonder how such a difficult program was allowed to assume the significant role of configuration program. The answer is the same for all three HES configuration programs: no one found it worth the time and money to make a better program. The engineers on the project, after all, were
experienced in hardware (or at best, embedded software); who had the time to design a high-level software program? The result was that whatever was readily available when the product first shipped was adopted as the configuration program.

In the case of Term Protocol, the protocol was already designed into the RFM, so it took little effort to build a simple terminal program around it.

1.2.2.2 Dandy (CU)

Dandy is another example of a configuration program adopted out of necessity. Originally written by an HNS engineer several years ago, Dandy was used as a debugger during the development of the CU’s embedded software. Engineers could connect a PC running Dandy to the CU through a serial line to accomplish the debugging. When it came time to ship the CU, Dandy was the only available program which could communicate with the CU, so it was chosen to be the configuration program. Like the Term Protocol, Dandy has a command-driven interface, although some effort has been made to organize the screens into menus. These menus make it easier to use than the Term Protocol, but the user is still required to type in most parts of commands, resulting in the same problems of rigidity and hexadecimal limitations that the Term Protocol has. Figure 8 shows the Dandy interface.
1.2.2.3 Configuration Editor (IFM)

Of the three HES configuration programs, the Configuration Editor (CE) is the most advanced. Written in BASIC, the CE implements a mostly menu-driven interface. Although the user interface is better than the other two programs, the program leaves much to be desired. Here are the CE’s shortcomings:

- The interface is text-based; no graphics whatsoever are used to show the menus.

- Several screens in the interface give the user no indication how to cancel the operation, and give no feedback when the program executes a command.

- The different modules of the program do not communicate well. In one case, operation B takes as input the output of operation A. The user must run operation A, physically write down the output, run operation B, and then type in the data that was written down.

- The speed of the program is unbearably slow, since it is written in interpreted BASIC. Also, serial communications are slow. For example, one operation reads sixteen NVRAM locations by sending sixteen commands to the RFM, when all of the locations could be read with a single command.
The readability of the code is non-existent. In nine hundred lines of code, there are only three comments.

The CE has been used by installers throughout the nine years that PES has been in existence. Surprisingly, PES engineers have been able to keep the CE up to date with the various changes that the IFM has experienced throughout the years. One probable reason for this feat is that the original author of the CE still works at HNS. Figure 9 shows one of several screens in the CE interface.

Figure 9. A screen from the Configuration Editor, a configuration program for the IFM. Notice the misspelling.

1.2.3 Need for the Configurator

With the three configuration programs in mind, consider the job of the HES installers. They must use the cryptic Term Protocol to configure the RFM. One mistake on the command line could cause them to have to start the installation from the beginning. They then switch to the Dandy interface for the CU, in which it is just as easy to make a mistake. Finally, they bring up the third unique interface, this time to configure the IFM with the CE. If all goes well, an installer can finish a configuration in under an hour.

Installations under such conditions rarely go smoothly, mainly because mistakes are so easy to make. Managers in charge of HES realized this, and in July 1994, HES asked HNS's Software Technology Department to build a configuration program which could replace at least the Term Protocol, if not two or all three of the existing HES programs. Thus the Configurator was born.

1.3 The Installer's PC

There is one more piece of history to explain before the Configurator is described. Even if the configuration program provided a quick, friendly, and forgiving interface, the installer would still have li-
mitations. More specifically, every HES installer has a flaw which affects the design of any configuration program: the PC.

To the installer, the PC is just another tool. Like the hammers, wrenches, and drills which ride next to it in the back of the truck, the PC has a specific job to do, beyond which it is not used. As such, installation companies generally provide their installers with the least expensive (and therefore, least powerful) PC that can do the job of configuration. Currently the least powerful PC used for HNS configurations is the Toshiba T1000. Some installers carry higher-powered PC’s, but the “lowest common denominator” is the important platform to know about. HES required that there only be one version of the Configurator released to the installers; therefore, it had to be able to run on the Toshiba T1000. (Luckily for the Configurator project, HNS recently dropped support for the TRS–80.)

1.3.1 Toshiba 1000

The Toshiba T1000 Portable Personal Computer was a new product in the early 1980’s. Small, lightweight, and running DOS, it competed favorably with the other portables on the market. Today, however, the T1000 is hopelessly out of date. It does not contain a hard disk or support a mouse, the microprocessor is an Intel 8086, and the internal RAM size is a tiny 256 kilobytes (K). To make matters worse, DOS must be loaded from a floppy diskette, which takes up roughly 30K of RAM. That leaves roughly 220K in which to fit any application which will run on the machine.

These limitations never hindered the existing HES configuration programs, because their sizes did not come close to the 256K limit. With the number of features planned for the Configurator, however, the size limit was a problem from the start. Even with more internal memory, the lack of mouse support still would have prevented the Configurator from running well under Microsoft Windows. The limitations of the T1000 provided quite a challenge to the development of the Configurator.

1.4 Thesis Overview

This thesis describes the development of a powerful, user–friendly configuration program for satellite dish hardware. The so–called “Configurator” is intended to cut costs in HNS satellite dish installations by reducing the time spent on hardware configuration. The code is designed for easy maintenance and customization. Chapter 2 explains the requirements for a configuration program presented by the HES project at the beginning of the thesis internship. Chapter 3 details the implementation of Configurator. Chapter 4 reconsiders the design of the Configurator, suggests possible future design changes, and relates the Configurator to comparable products in industry. Chapter 5 evaluates the Configurator by considering the feedback from use in the field. Chapter 5 also evaluates the development process of the Configurator. Chapter 6 summarizes and concludes the thesis.
Chapter 2
Configuration Requirements

This chapter describes the various requirements for the Configurator as produced by HES engineers at the beginning of the thesis internship. Note that since only one version of the Configurator may be released, all requirements must be met by the same package.

2.1 User Requirements

There are three types of Configurator users: installers, lab technicians, and code maintainers. The design of the program is geared primarily towards the installer, but since there can only be one release of the Configurator, all types of users must be accounted for in the design.

2.1.1 Installer

The installer imposes only a few requirements the Configurator. Briefly, the Configurator must run on the installer’s PC, and it must be easy enough to use.

2.1.1.1 PC

The Configurator must run on a PC with the following configuration (it would also be able to run on PC’s with more advanced configurations):

- Intel 8086 microprocessor
- 256 K RAM
- no hard drive
- low density floppy disk drive
- no mouse
- monochrome VGA monitor
- one serial port
- one parallel port

2.1.1.2 User Interface

In order for the Configurator to be usable by the “average installer,” it must require no knowledge of the following:

- mapping of configuration parameters to NVRAM values
- hexadecimal arithmetic
- layout of NVRAM locations
- serial line protocols

(The previous HES configuration programs required the knowledge of all of these items.)

Using the previous list as a guide, the following list comprises the requirements for the Configurator user interface. The Configurator user interface must do the following:

- display NVRAM values in English whenever possible, or in “real” units such as MHz, seconds, or dB;
- save (to the PC) and retrieve files of configuration parameters;
- when hex numbers must be displayed, do any necessary calculations or conversions internally, or provide controls of these calculations in English;
- hide the physical layout of NVRAM locations from the user;
- prevent the user from setting any protected or otherwise “important” NVRAM locations; and
- hide the serial line protocol from the user.

2.1.2 Lab Technician

The lab technician is an HNS employee who uses the Configurator for the following purposes:

- development and testing of new HES hardware
- troubleshooting faulty HES hardware
- preparing “installation packages” to be used by installers in the field
- providing customer support to installers using the Configurator in the field

The Configurator must provide the lab technician with much more access to the HES hardware than what the installer gets. Specifically, the installer must be able to do the following:

- set the value of any NVRAM location;
- save (to the PC) and retrieve files of configuration parameters;
- view in hex format the values of all NVRAM locations; and
- download files of IDU commands to the target.
2.1.3 Code Maintainer

The code maintainer is the HNS employee who makes periodic changes to the Configurator when the interface with the IDU changes. For example, the maintainer must modify the code when a new NVRAM location is added to the IDU. Thus, the Configurator’s code must be designed in such a way to allow maintenance to occur easily and quickly.

2.2 IDU Requirements

This section describes the requirements of the IDU by explaining its interface with configuration programs and showing how that interface relates to its internal structure. The IDU is comprised of four main components, three of which – the RFM, CU, and IFM – require configuration by the Configurator. The RFM, CU, and IFM are known as the target boards, or more commonly, the targets.

Chapter 1 was not completely accurate in implying that setting NVRAM is the only operation required of the configuration program. In general, the configuration program performs two types of actions on the target: NVRAM operations (see Section 1.1) and so-called “monitor and control” operations. Monitor and control (commonly known to installers as “M&C”) is the set of functions which handle configuration parameters which must be monitored or controlled in real time. An M&C function may involve setting NVRAM, but it always has a dynamic, real time element; this is what distinguishes the M&C functions and their associated configuration parameters from “ordinary” static parameters. For example, the RFM’s 10 MHz Ref parameter controls, in real time, a frequency output on the board. By rapidly incrementing or decrementing the value of 10 MHz Ref, the installer acquires real time control of the frequency output.

2.2.1 RFM interface

This section describes the RFM’s interface with the configuration program.

2.2.1.1 Term Protocol

The Term Protocol is the name of the existing RFM configuration program, but it also refers to the serial line protocol used in configuring the RFM; this section describes the protocol. The Term Protocol is a very simple protocol. It is asynchronous, so there are no timing requirements. The only requirement is that all packets (commands, echos, etc.) be encapsulated with a line feed (ASCII 10) prefix and carriage return (ASCII 13) postfix (see Figure 10). The IFM also uses the Term Protocol to communicate with the Configurator.
2.2.1.2 RFM Command Set

The RFM responds to every command with an echo packet, which is identical to the command packet. If appropriate to the command, the RFM also sends a packet of output data. Table A lists the set of RFM commands used by the configuration program. Listed with each command is the output packet, if one exists. For the sake of brevity, examples do not include echo packets, which always precede output packets. Finally, note that numerical data is always transmitted in hex format.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;addr&gt; (with args)</td>
<td>This is the only command which requires arguments. Writes consecutive NVRAM values starting at address addr. The number of bytes on the input line dictates how many NVRAM values will be written; the maximum is 16. Each byte argument must be separated by one space character.</td>
</tr>
<tr>
<td>Output</td>
<td>A OK</td>
</tr>
<tr>
<td>Example</td>
<td>Configurator: AF002 D2 44 3B This writes to locations 0xF002, 0xF003, and 0xF004. No other NVRAM locations are affected.</td>
</tr>
<tr>
<td>D&lt;addr&gt;</td>
<td>Reads sixteen consecutive NVRAM values starting at addr.</td>
</tr>
<tr>
<td>Output</td>
<td>Byte values of the sixteen locations beginning with addr. Each value is separated by one space character.</td>
</tr>
<tr>
<td>Example</td>
<td>Configurator: DF020 RFM output: 00 FF 21 45 13 A0 54 34 45 E0 BA 03 08 1A D3 00 In this example, 0x21 is the value at location 0xFO22.</td>
</tr>
<tr>
<td>L</td>
<td>Computes the checksum of the configuration parameters and stores the result in a pre-determined location in NVRAM.</td>
</tr>
<tr>
<td>Output</td>
<td>L OK</td>
</tr>
</tbody>
</table>
2.2.1.3 RFM Configuration Parameters

The following table shows the name of each configuration parameter and its location in NVRAM. If the parameter maps to a discrete set of NVRAM values, the mapping is shown in the Description field. Appropriate ranges and default values are showed where applicable. Since the configuration program is only responsible for knowing how to map parameters to NVRAM, further knowledge about how each parameter is used internally within the RFM is not necessary. The Offset field shows the offset of each parameter from the 0xF000 base address of configuration parameters in NVRAM.

<table>
<thead>
<tr>
<th>Table A. RFM Command Set (continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;mode&gt;</strong></td>
</tr>
<tr>
<td>These are several commands which set the RFM’s mode (see Section 2.2.1.4.2):</td>
</tr>
<tr>
<td>MN – Normal mode</td>
</tr>
<tr>
<td>MA – Pointing mode</td>
</tr>
<tr>
<td>MD – Diagnostic mode</td>
</tr>
<tr>
<td>MT – Tx Comm mode (transmit communications)</td>
</tr>
<tr>
<td>MR – Rx Comm mode (receive communications)</td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>M OK</td>
</tr>
<tr>
<td><strong>OT</strong></td>
</tr>
<tr>
<td>Gets ODU info. Internally, the RFM recognizes the ‘O’ as an ODU command prefix and sends a ‘T’ command to the ODU. The RFM returns the ODU’s output without any modifications.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>&lt;partnum&gt;&lt;majrel&gt;&lt;minrel&gt;</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>Configurator: OT</td>
</tr>
<tr>
<td>RFM output: 1011836-00050201</td>
</tr>
<tr>
<td>The output is interpreted as follows: 1011836-0005 is the ODU’s part number, 02 is the major firmware release number, and 01 is the minor release number.</td>
</tr>
<tr>
<td><strong>Q&lt;val&gt;</strong></td>
</tr>
<tr>
<td>Sets the 10 MHz Ref parameter (see Section 2.2.1.4.1). The val argument must be exactly three characters long.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>Q OK</td>
</tr>
<tr>
<td><strong>Example</strong></td>
</tr>
<tr>
<td>Configurator: Q006</td>
</tr>
<tr>
<td>This sets the 10 MHz Ref parameter to 0x06.</td>
</tr>
<tr>
<td><strong>Z</strong></td>
</tr>
<tr>
<td>Resets the RFM.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>none</td>
</tr>
</tbody>
</table>

2.2.1.3 RFM Configuration Parameters

The following table shows the name of each configuration parameter and its location in NVRAM. If the parameter maps to a discrete set of NVRAM values, the mapping is shown in the Description field. Appropriate ranges and default values are showed where applicable. Since the configuration program is only responsible for knowing how to map parameters to NVRAM, further knowledge about how each parameter is used internally within the RFM is not necessary. The Offset field shows the offset of each parameter from the 0xF000 base address of configuration parameters in NVRAM.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Offset (hex)</th>
<th>Length (bytes)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>00</td>
<td>1</td>
<td>00</td>
<td>Mode of operation.&lt;br&gt;00 – normal&lt;br&gt;01 – pointing&lt;br&gt;02 – diagnostic&lt;br&gt;03 – tx comm&lt;br&gt;04 – rx comm</td>
</tr>
<tr>
<td>Transponder Window</td>
<td>01</td>
<td>1</td>
<td>none</td>
<td>The frequency window in which the RFM tunes the ODU. Window numbering scheme depends on the type of ODU connected to the IDU. Regardless of the ODU, window numbers are contiguous integer values.</td>
</tr>
<tr>
<td>10 MHz Ref</td>
<td>02</td>
<td>2</td>
<td>none</td>
<td>Factory set; should not be altered except in maintenance procedures. May be set with the Q command (see Table A). (Range 0000..0FFF)</td>
</tr>
<tr>
<td>RFM CFG</td>
<td>04</td>
<td>1</td>
<td>00</td>
<td>Not used.</td>
</tr>
<tr>
<td>RFM U/C Gain</td>
<td>05</td>
<td>2</td>
<td>none</td>
<td>Factory set; do not alter. (Range 0000..0FFF)</td>
</tr>
<tr>
<td>ODU Gain</td>
<td>07</td>
<td>1</td>
<td>00</td>
<td>Not used.</td>
</tr>
<tr>
<td>RFT Gain</td>
<td>08</td>
<td>2</td>
<td>0000</td>
<td>Not used.</td>
</tr>
<tr>
<td>Rx Gain</td>
<td>0A</td>
<td>2</td>
<td>0000</td>
<td>Not used.</td>
</tr>
<tr>
<td>ODU Class</td>
<td>0C</td>
<td>1</td>
<td>00</td>
<td>Not used.</td>
</tr>
<tr>
<td>ODU Latency</td>
<td>0D</td>
<td>2</td>
<td>0000</td>
<td>Not used.</td>
</tr>
<tr>
<td>Startup ALC</td>
<td>0F</td>
<td>1</td>
<td>none</td>
<td>Function of four site-specific (non-NVRAM) parameters and the ODU type.</td>
</tr>
<tr>
<td>Keep Alive Time</td>
<td>10</td>
<td>2</td>
<td>0258</td>
<td>Units are seconds. Default 0x0258 = 600 decimal, which is 10 minutes.</td>
</tr>
<tr>
<td>ODU Power Offset</td>
<td>12</td>
<td>2</td>
<td>0000</td>
<td>Range 0000..01FF</td>
</tr>
<tr>
<td>ODU Alarm Flag</td>
<td>14</td>
<td>1</td>
<td>00</td>
<td>00 – ignore alarms&lt;br&gt;01 – respond to alarms</td>
</tr>
<tr>
<td>CU Power Mode</td>
<td>15</td>
<td>1</td>
<td>00</td>
<td>00 – varying&lt;br&gt;01 – constant</td>
</tr>
<tr>
<td>ODU Power Configuration</td>
<td>16</td>
<td>1</td>
<td>01</td>
<td>00 – constant power&lt;br&gt;01 – constant gain</td>
</tr>
<tr>
<td>Power Control Mode</td>
<td>17</td>
<td>1</td>
<td>02</td>
<td>00 – no power control&lt;br&gt;01 – satellite power control&lt;br&gt;02 – EIRP power control&lt;br&gt;03 – satellite and EIRP power control</td>
</tr>
<tr>
<td>Commissioned Tx Gain</td>
<td>18</td>
<td>2</td>
<td>none</td>
<td>Function of three site-specific parameters, similar to Startup ODU ALC Level.</td>
</tr>
</tbody>
</table>
2.2.1.4 RFM Monitor and Control

There are three monitor and control functions required in RFM configuration. Two of them write to locations in NVRAM, but the important characteristic about these functions is that some aspect of their execution is affected by a time requirement.

2.2.1.4.1 10 MHz Ref

This parameter is part of a feedback loop by which the installer can “tune” the equipment. Setting the 10 MHz Ref parameter affects a particular frequency output on the RFM. The installer reads this output by connecting it to a frequency counter. By “tweaking” the 10 MHz Ref value up or down in small increments, the installer can gradually bring the output on the frequency counter to the right value.

2.2.1.4.2 Mode

The Mode parameter does not require polling like the 10 MHz Ref operation; thus it is less a “monitor” function than a “control” function. The Mode commands (see Table A) only set the mode, not

---

### Table B. RFM Configuration Parameters (continued)

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Offset (hex)</th>
<th>Length (bytes)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Startup ODU Tx Gain</td>
<td>1A</td>
<td>2</td>
<td>0000</td>
<td>Should always be set to 0000.</td>
</tr>
<tr>
<td>KCM Present</td>
<td>1C</td>
<td>1</td>
<td>none</td>
<td>Whether a certain module is installed in the RFM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 - not present</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 - present</td>
</tr>
<tr>
<td>PES Present</td>
<td>1D</td>
<td>1</td>
<td>01</td>
<td>Whether a PES IDU is connected to the RFM.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 - not present</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 - present</td>
</tr>
<tr>
<td>KCM Receive Channel</td>
<td>1E</td>
<td>2</td>
<td>none</td>
<td>Function of Transponder Window and satellite type.</td>
</tr>
<tr>
<td>ODU Minimum Measurable Power</td>
<td>20</td>
<td>2</td>
<td>none</td>
<td>Two-byte hex number.</td>
</tr>
<tr>
<td>ODU Maximum Measurable Power</td>
<td>22</td>
<td>2</td>
<td>none</td>
<td>Two-byte hex number.</td>
</tr>
<tr>
<td>Ku–band Power Detector Range</td>
<td>24</td>
<td>1</td>
<td>none</td>
<td>One-byte hex number.</td>
</tr>
<tr>
<td>HPA Present</td>
<td>25</td>
<td>1</td>
<td>none</td>
<td>Whether a certain module is installed in the ODU.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>00 - not present</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>01 - present</td>
</tr>
</tbody>
</table>
read it. When a Mode command is sent, the RFM does not accept commands for several seconds while it sets itself to the new mode.

2.2.1.4.3 Reset RFM
This forces a “reboot” of the RFM. It is often done after new parameters have been written to NVRAM, so that they may take effect on the system. Similar to the Mode commands, this falls in the “control” category of M&C. The configuration program must wait several seconds after a reset before issuing more commands.

2.2.2 CU interface
This section details the CU’s interface with the configuration program.

2.2.2.1 Dandy protocol
Like the RFM, the name of the CU’s existing configuration program is also the name of the serial line protocol. The Dandy protocol, like the Term Protocol, is asynchronous and very simple. The packet format, however, is slightly different: the only required encapsulation is a carriage return at the end of the data. Figure 11 shows the format of a Dandy protocol packet.

![Figure 11. Dandy protocol packet format.](data CR)

2.2.2.2 CU Command Set
Unlike the RFM, the CU does not echo commands. Every command, however, elicits an output packet. CU command and output packets have a different format than RFM packets. CU packets never separate data (bytes, arguments, etc.) by spaces. Figure 12 shows the format of CU command and output packets; elements shown in brackets (<> ) may or may not be in the packet, depending on the type of command. Table C lists the set of CU commands used by the configuration program. For the sake of brevity, the ‘S’ character that precedes every command and output is not shown; each command is listed by its ID. Listed with each command is the output packet. Since NVRAM is byte addressed, unless otherwise noted an “NVRAM location” refers to one byte of memory.

There are two states in which the CU operates. In normal operation, the CU is in RUN state, in which it runs its internal programs and does not respond to any commands except the break. The break commands put the CU in BREAK state, in which the CU halts its program run and will respond to any of the commands shown in Table C.
Figure 12. CU packet formats.

```
Si <seg> <off> <arg> CR
```

CU command packet

```
Si <loc> <loc> ... <loc> CR
```

CU output packet

where

- \( id \) is a one-byte hex number designating the type of command or output;
- \( seg \) is a two-byte hex number designating the segment of the address;
- \( off \) is a two-byte hex number designating the offset of the address;
- \( loc \) is a one-byte hex value of an NVRAM location.
- \( arg \) is a one-byte hex number of argument data; and

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D</td>
<td>Writes one NVRAM location at address ( seg:off ) with a value of ( arg ).</td>
</tr>
</tbody>
</table>

**Output**

```
S0D
```

**Example**

Configurator: **S0D7000000310**
This sets NVRAM location 0x7000:0x0003 to the value Ox 0.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0A</td>
<td>Reads ( arg ) number of consecutive NVRAM values, starting with ( seg:off ). The maximum value of ( arg ) is 0x10, or sixteen locations. The output contains the values of the ( arg ) locations (not separated by any space characters).</td>
</tr>
</tbody>
</table>

**Output**

```
S03<loc><loc><loc>...<loc>
```

**Example**

Configurator: **S0A700000003**
Output: **S0321A450**
This reads three NVRAM locations starting at 0x7000:0x0003. The respective values are 0x21, 0xA4, and 0x50.
Table C. CU Command Set (continued)

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Similar to the 0D command, except that it reads ( \text{arg} ) number of two-byte words. The maximum value of ( \text{arg} ) is 08, so a maximum of 16 locations may be read. Output is similar to 0D output.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>S03&lt;loc&gt;&lt;loc&gt;&lt;loc&gt;...&lt;loc&gt;</td>
</tr>
</tbody>
</table>
| **Example** | Configurator: S177000000002  
Output: S0321A450BA  
This reads two two-byte words from NVRAM starting at 0x7000:0x0003. The respective values are 0x21, 0xA4, 0x50, and 0xBA. |
| 0B 14 | Issued consecutively, these two commands put the CU in BREAK state. The CU must be in BREAK state before any other commands may be sent. Neither command needs any arguments. The ellipsis on the end of the second output refers to 19 hex digits following the “S09.” These contain no information meaningful to the command. |
| **Output** | S03  
S09... |
| 15 | This command “unbreaks” the CU and puts it in RUN state. In RUN state, the CU ignores all commands except the break. |
| **Output** | S12 |
| 0F | This command resets the CU. Note that there is no output to this command. |
| **Output** | none |

### 2.2.2.3 CU Configuration Parameters

The following table shows the name of each CU configuration parameter and its location in NVRAM. If the parameter maps to a discrete set of NVRAM values, the mapping is shown in the Description field. Appropriate ranges and default values are showed where applicable. Since the configuration program is only responsible for knowing how to map parameters to NVRAM, further knowledge about how each parameter is used internally within the CU is not necessary. The Offset field shows the offset of each parameter from the base address of the configuration parameters in NVRAM, which is contained in address 0x7000:0x0001. Note that the NVRAM locations of the parameters are not contiguous.
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Offset (hex)</th>
<th>Length (bytes)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksum</td>
<td>00</td>
<td>2</td>
<td>none</td>
<td>The checksum, although technically not a configuration parameter, holds the checksum of all the other parameters. The configuration program is responsible for keeping the checksum accurate.</td>
</tr>
</tbody>
</table>
| Mode after Reset  | 03           | 1              | 00      | 00 – Operational  
01 – Debug/Config |
| OCC Access        | 04           | 1              | 00      | 00 – IF/RF Link  
01 – Serial port (CCU) |
| OCC Data Rate     | 05           | 1              | *       | If OCC Access is IF/RF link:  
01 – 4800 bps  
02 – 9600 bps  
03 – 19200 bps | *Default for IF/RF link  
04 – 16000 bps  
05 – 32000 bps  
06 – 56000 bps  
07 – 64000 bps  
If OCC Access is Serial port (CCU):  
44 – 300 bps  
66 – 1200 bps  
88 – 2400 bps  
99 – 4800 bps  
0BB – 9600 bps  
0CC – 19200 bps | *Default for Serial port (CCU) |
| OCC IF Freq       | 06           | 2              | none    | This parameter is mapped from its hex value to a decimal MHz value by a simple formula. |
| OCC FEC Rate      | 08           | 1              | 01      | 00 – rate 1  
01 – rate 1/2  
02 – rate 3/4 |
| OCC Modulation Type | 09          | 1              | 00      | 00 – QPSK  
01 – BPSK |
| OCC Sweep Range   | 0B           | 1              | 05      | 05 – 50 kHz  
06 – 60 kHz |
| ALC               | 0C           | 1              | 01      | 00 – disable  
01 – enable |
| Initial Tx Level  | 0D           | 1              | 46      | A one-byte hex number. (Range 00..FF) |

### 2.2.2.4 CU Monitor and Control

The user must be able to put the CU in BREAK or RUN mode, and reset it.
2.2.3 IFM interface

This section details the IFM's interface with the configuration program.

2.2.3.1 Term Protocol

As mentioned in Section 2.2.1.1, the IFM uses the Term Protocol for serial port communications. This merits some explanation. As a board in a “standalone” PES IDU, the IFM uses the Term Protocol. As a target inside the HES IDU, however, it uses the Term Protocol out of necessity: commands to the IFM are received by the RFM and routed internally to the IFM. Thus the configuration program must use the Term Protocol when sending IFM commands, because it is actually sending them to the RFM. Therefore, IFM commands are encapsulated in the same format as RFM commands (as shown in Figure 10).

2.2.3.2 IFM Command Set

The IFM's command set is almost identical to the RFM's. The “D” and “A” commands (for reading and writing NVRAM locations) are the same, as is the “Z” reset command. The IFM does not support the RFM's M&C commands “L” or “Q", and it does not recognize the “OT" ODU info command. The IFM does have a Mode command, but the format is different from the RFM's Mode command. Table E describes the IFM Mode command.

Since in the HES IDU IFM commands are received by the RFM and routed to the IFM, the overlap of command sets causes a problem. The RFM must distinguish, for example, a “D” command intended for the RFM from one intended for the IFM. To do this, an additional requirement is imposed on the IFM command set: all IFM commands must be prepended by a dash (ASCII 45). For example, the reset command in for HES IFM's is “-Z". If the IFM is not part of a HES but is instead inside a standalone PES, the configuration program sends its commands directly to the IFM, and the dash is not needed.

<table>
<thead>
<tr>
<th>A, D, Z</th>
<th>Identical* to those described in the RFM Command Set (Table A).</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;</td>
<td>These are several commands* which set the IFM's mode (see Section ):</td>
</tr>
<tr>
<td></td>
<td>WN – Normal mode</td>
</tr>
<tr>
<td></td>
<td>WI – Pointing mode</td>
</tr>
<tr>
<td></td>
<td>WD – Diagnostic mode</td>
</tr>
</tbody>
</table>

*In HES IDU’s, all IFM commands must be prepended by a dash (ASCII 45).

2.2.3.3 IFM Configuration Parameters

One difference between IFM NVRAM and other targets is that the IFM configuration parameters are stored in two “parameter pages” in NVRAM, starting at absolute addresses 0x2000 and 0x2100, re-
spectively. Each page contains values for all the configuration parameters, but only those contained in the “active page” are used. Thus, at a given time exactly one page is designated the active page. (The only exception to this is before the IFM is installed, when neither page is active.) The configuration program reads and writes to the active page only. If a major error occurs (which wipes out all the memory in the active page, for example), the inactive page can be made active, and normal operation can resume. Thus the inactive page may be thought of as maintaining a backup copy of the configuration parameters.

The following table shows the name of each IFM configuration parameter and its location in NVRAM. If the parameter maps to a discrete set of NVRAM values, the mapping is shown in the Description field. Appropriate ranges and default values are showed where applicable. The Offset field shows the hexadecimal offset of each parameter from the beginning of the parameter page. In two–byte parameters, the least significant byte occupies the lower offset.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Offset (hex)</th>
<th>Length (bytes)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Page Flag</td>
<td>00</td>
<td>1</td>
<td>none</td>
<td>01 – This parameter page is the active page. 00 – This is the inactive page.</td>
</tr>
<tr>
<td>Network ID</td>
<td>01</td>
<td>2</td>
<td>none</td>
<td>An arbitrary two–byte number.</td>
</tr>
<tr>
<td>Primary Out-route Frequency</td>
<td>03</td>
<td>3</td>
<td>none</td>
<td>Installers are most comfortable with hex representations of the Outroute Frequencies. Therefore, it is not a requirement to display them in units of MHz (as it is for other frequency parameters). There is a particular mapping formula which maps the frequencies from hex to MHz. Also, the byte significance is exceptional: the most significant byte occupies the lowest offset, the least significant byte occupies the “middle” offset, and the “middle” significant byte occupies the highest offset.</td>
</tr>
<tr>
<td>Secondary Out-route Frequency</td>
<td>06</td>
<td>3</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Inroute Frequencies</td>
<td>09–47</td>
<td>2</td>
<td>none</td>
<td>There are thirty–two Inroute Frequencies, each two bytes long. The first Inroute Frequency begins at offset 09, and the last begins at offset 47 (hex). All Inroute Frequencies are mapped from hex to MHz by the same formula. Usually an IFM will only use five or six of the thirty–two frequencies, but the configuration program must be able to read and write all of them.</td>
</tr>
<tr>
<td>Remote Base Address</td>
<td>4A</td>
<td>1</td>
<td>none</td>
<td>Arbitrary hex number, range (01..FF).</td>
</tr>
</tbody>
</table>
2.2.3.4 IFM Monitor and Control

There are several M&C functions for the IFM, only two of were made a requirements for this thesis: the Reset and Mode commands. The Reset command is identical to the RFM’s (see Section 2.2.1.4.3), except that the command must be preceded by a dash. Similar to the RFM’s Mode command (see Section 2.2.1.4.2), the IFM’s Mode command sets the IFM’s mode of operation. Table E details this command.

### Summary

This chapter has described the requirements which must be met by the Configurator. The installers and lab technicians need an easy-to-use interface, and the code maintainer needs the ability to modify the program easily. Each of the three targets — RFM, CU, and IFM — have different requirements. The protocol, command set, configuration parameters, and M&C requirements of each target were listed.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Offset (hex)</th>
<th>Length (bytes)</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port Card Base Addresses</td>
<td>4B–5A</td>
<td>1</td>
<td>none</td>
<td>There are sixteen Port Cards in the PES IDU, each of which has a unique “base address.” Each base address falls in the range (00..7C).</td>
</tr>
<tr>
<td>Port Card Configuration Value</td>
<td>5B–6A</td>
<td>1</td>
<td>none</td>
<td>Arbitrary hex number for each of the sixteen Port Cards. May differ from Port Card to Port Card.</td>
</tr>
<tr>
<td>Initial Transmit Power Level</td>
<td>6E</td>
<td>1</td>
<td>none</td>
<td>Like the RFM’s Startup ALC, a hex number mapped by a multi-argument formula.</td>
</tr>
<tr>
<td>Initial Timing Offset</td>
<td>6F</td>
<td>2</td>
<td>none</td>
<td>Arbitrary hex number.</td>
</tr>
<tr>
<td>Flags</td>
<td>71</td>
<td>1</td>
<td>none</td>
<td>The three least significant bits of this byte may be set. All other bits are zero.</td>
</tr>
<tr>
<td>Outroute Rate</td>
<td>72</td>
<td>1</td>
<td>none</td>
<td>00 – 128 kbit/s 01 – 512 kbit/s 02 – 1664 kbit/s</td>
</tr>
<tr>
<td>Inroute Rate</td>
<td>73</td>
<td>1</td>
<td>none</td>
<td>00 – 64 kbit/s 01 – 128 kbit/s 02 – 256 kbit/s</td>
</tr>
</tbody>
</table>
Chapter 3
Implementation

This chapter describes how the Configurator meets the requirements listed in the last chapter. Section 3.1 presents the overall structure of the Configurator. Section 3.2 summarizes the user interface. Section 3.3 goes into greater detail by describing the most important data types. Section 3.4 describes the modules in the core; its discussion is divided between GUI modules and modules called from the menu bar. Finally, Section 3.5 summarizes how the Configurator is tailored to meet the particular requirements of each target, and Section 3.6 summarizes the chapter.

3.1 Configurator structure

Currently, there is a Configurator for each of the three targets. The structure of each Configurator separates the code into two parts: a “core” of target-generic code, which is common to all Configurators, and certain target-specific code, which implements the target’s specific requirements. The core implements the user interface and any functionality required by all three targets. In practice, all three Configurators are contained on one “field service disk,” which the installer carries to the installation site (see Figure 13).

![Figure 13. Configurators and the field service disk.]

3.2 User interface

The C-scape graphics library, produced by Liant Corporation, is used to implement the user interface. C-scape was chosen because it specializes in interfaces without mouse support. Also, fairly complex interfaces may be implemented with C-scape in a code size of around 150K. The Configurator’s graphical user interface will hereafter be referred to as the GUI, in order to distinguish it from interfaces between the modules.
Figure 14 shows a typical “screen shot” of the GUI. Most of the screen is taken up by the parameter buffer, where the user may edit the values of the configuration parameters. The parameter buffer shows the title of each parameter and its value. Above the parameter buffer is the menu bar, which contains all the functions accessible to the user. A typical use of the Configurator consists of two basic steps:

1. Editing the parameter values.
2. Choosing the option on the menu bar which writes the values to target NVRAM.

The only other major aspect of the GUI is the mode switch. As stated in Section 2.1.2, the lab technician has different requirements from the installer. While the installer should be protected from making mistakes such as writing the wrong NVRAM values, the technician must be provided access to all the NVRAM values. The Configurator meets both sets of requirements by implementing dual-mode operation. Installer mode displays only those configuration parameters which are safe to modify. Technician mode displays the target’s entire set of configuration parameters. A special keystroke is used to toggle modes. Documentation about the mode switch is given to lab technicians only.

**GUI paradigm**

The Configurator’s GUI was designed with a paradigm of simplicity. Here are the principles of the paradigm:
• Consistency. No matter if the cursor is in the parameter buffer, menu bar, or any other window, these keyboard bindings always work the same:
  – The arrow keys always control the cursor. In the parameter buffer and any other windows, UP and DOWN move the cursor between fields. In windows with a row of buttons, LEFT and RIGHT can be used to highlight the desired button.
  – ENTER is always used to select the current field. In the parameter buffer, ENTER moves to the next field. On the menu bar, ENTER pulls down a menu, and it chooses the selected menu choice. In windows with buttons, it is the key the user types to “push” the selected button.

• Clarity. No unnecessary details are ever exposed to the installer. This implemented by the following:
  – Parameters are given names most easily understood by the installer.
  – Where applicable, parameter values are displayed in the appropriate units.
  – Parameters which the installer does not need to know about are not displayed in installer mode.
  – Instructions on how to get to technician mode are not displayed, and the keystroke used for the mode switch is difficult to type by accident. The technician mode interface displays instructions on how to return to installer mode, in case the mode switch keystroke is typed by accident.
  – Messages to the user are written in plain, clear English.

• Exit signs. Instructions on how to exit the current operation are always visible. Windows called from the menu bar either provide a CANCEL button or display instructions about the ESC key. If the user accidentally enters technician mode, the menu bar displays the way to return to installer mode.

In addition to these principles, the Configurator also provides certain features for users who are familiar with common PC interfaces. For example, typing the ESC key escapes from every popup screen, and the ALT key can be typed with a letter key to access the menu starting with the letter.

3.3 Data types and the core/target-specific interface
The separation between the core and target-specific code is central to the Configurator. Target-specific code interfaces with the core through the use of tables and functions. Collectively, these inter-
faces draw upon the three most important data structures in the code: the fdefs table, the menu bar
tables, and the state structure.

3.3.1 The fdefs table

fdefs is a statically defined table which defines the contents of the parameter buffer. It contains all the vital information about a target's configuration parameters: their names, valid values, how and where they map to NVRAM, and how and where they are displayed in the parameter buffer. Here is its type definition, from the target-generic header file types.h:

```c
typedef struct {
    /* GUI info */
    char *title;        /* parameter name */
    char *datal;       /* non/writeables, or choices */
    int row, col;      /* position in parameter buffer */
    int ftype;         /* field type */
    char *deflt;       /* default value string */
    char *range;       /* valid range string */

    /* map info */
    void *field_var;   /* field variable pointer */
    short standard_map; /* standard map or not */
    unsigned nvram_loc; /* nVrAm offset it maps to */
    short numbytes;    /* its number of bytes in nVrAm */
} Def;
```

Each Def object contains all the data necessary to define one configuration parameter. To hold the data for all the configuration parameters, fdefs has type Def[]. Here is an example initialization of fdefs:
In the above example, `fdefs[]` is initialized with three configuration parameters. Commissioned Tx Gain has field type `UDEC_FIELD`, which stands for “unsigned decimal field”. “##.# dB” defines the appearance of the parameter’s field. The (0, 0) position makes the parameter appear in the upper left corner in the parameter buffer. The `standard_map`, `nvram_loc`, and `numbytes` elements define how the parameter maps to target NVRAM. The `NULL` element after the first two parameters signals the beginning of technician mode parameters. The two consecutive `NULL` elements signal the end of the table.

### 3.3.2 Menu bar tables

There are two menu bar tables: one defining the installer mode menu bar, and one for the technician mode menu bar. The initialization of a menu bar table is even simpler than that of `fdefs[]`. For example:

```c
Menubar my_menu = {
   { "File ", NULL, 0 },
   { "Open File", open_file, 0 },
   { "Save File", save_file, 0 },
   { FRAME_END },
   { "NVRAM ", NULL, 0 },
   { "Write to NVRAM", write_nvram, 0 },
   { FRAME_END },
   { FRAME_END }
};
```
In this example, the File menu has two options, and the NVRAM menu has one option. Beside each option name is a pointer to the function it calls when selected by the user. Single FRAME_END elements separate individual menus, and two FRAME_END elements signal the end of the table.

### 3.3.3 Using the tables

Each table described above is defined statically in a target-specific header file. The tables are given the same names in each header file; thus, it is simple for the core to reference any one of them. For example, fdefs may be declared by a core module with the following:

```c
extern Def fdefs[];
```

The advantage of tables is that they are simple to modify. Most of the code maintainer’s work, in fact, is limited to the header file which contains fdefs and the menu bar tables.

The problem with tables is that they are limited in capability. Mainly containing scalar data, tables are best suited to “filling in the blanks” in core-defined structures. For example, the parameter buffer is filled in with data from fdefs. Thus, the core defines the extent of a table’s usefulness. Whenever a target needs an extension to the core, however, tables are insufficient. This is the case in which a target-specific function is required.

### 3.3.4 state and target-specific functions

Several locations in the core call target-specific functions to allow the tailoring of a Configurator to the particular target. Target-specific functions must give the programmer as much freedom as possible in extending, or making exceptions to, the core. Therefore, the state object is passed as an argument to target-specific functions. state gives the functions access to virtually everything in the program. Here is its type definition:

```c
typedef struct {
    /* GUI state */
    int which_mode; /* which mode we’re in */
    sed_type barsed; /* menu bar */
    sed_type dentrysed; /* parameter buffer */

    /* I/O state */
    int ok_to_write_to_target; /* flag OK to write nvram */
    int comport; /* which COM port to use */
    unsigned long base_addr; /* base address of nvram */
    unsigned num_nvram_locs; /* number of nvram locs */
    int *offs; /* array of nvram offsets */
}
```

42
Three of the elements of state—\texttt{which\_mode}, \texttt{barsed}, and \texttt{dentrysed}—concern the GUI. \texttt{which\_mode} represents the current mode of the program (installer or technician). \texttt{barsed} and \texttt{dentrysed} are pointers to the menu bar and parameter buffer, respectively. The state members under the "\texttt{comm\ state}" comment are used in communicating with the target; these are generally used only by functions which read or write the values in the parameter buffer. Finally, \texttt{tspec} is provided to point at any target-specific state data.

Passing \texttt{state} to a target-specific function thus allows the programmer great freedom. \texttt{barsed} and \texttt{dentrysed} allow modifications to any part of the GUI, such as creating new screens with arbitrary interfaces. The next five elements in the definition allow the function to send arbitrary commands to the target. Finally, \texttt{tspec} can access a data type defined to suit the needs of the target.

Despite the advantages gained by passing \texttt{state} as an argument, target-specific functions are still limited. The purpose of target-specific code is to allow the programmer some freedom in tailoring the Configurator to the target. It would be nice if the line between target-specific and target-generic functionality was absolute; after linking in the core modules, obligations to the core would be forgotten and target-specific modules could contain arbitrary code. However, that is not the case. The flexibility of target-specific code is limited by requirements of the core. (Chapter 4 discusses how most of these problems could have been solved with an object-oriented approach.) Here are the main limitations:

- Only the core can define where a target-specific function may be called. Ideally, target-specific code would be accessible at arbitrary locations. Nonetheless, the present design has proven sufficient for the Configurators of three separate targets, each with its own modifications to the core.

- Target-specific functions may be required to do certain things. For example, \texttt{main} is required call target-generic functions to initialize \texttt{state} and begin the mode loop.

- Function prototypes are core-defined. However, as explained above, \texttt{state} is a sufficient argument to implement a wide range of functionality.

- Every target-specific function called directly from the core must be defined somewhere in the target-specific modules. Even if the function contains only a null instruction, it must exist.
Lest these limitations seem too restrictive, the merits of the design are worth mentioning. The best aspect about the Configurator's generic/specific separation is that files are atomic; that is, target-specific code never appears in the same file as target-generic code. This allows for a clean distinction between target-generic and target-specific modules.

3.4 Modules
This section summarizes the modules which comprise the Configurator. Note that `<targ>` in a file name refers to the name of the target; for example, `<targ>_init.c` in the RFM Configurator is actually `rfm_init.c`.

3.4.1 GUI modules
The following is a module-by-module description of the GUI. Refer to Figure 15 for a graphical representation of the following discussion, or to Table G for a list of modules and their descriptions.

```
Figure 15. Modules of the Configurator GUI.
```

`main()` is contained in the file `<targ>_main.c`. It is deliberately placed in a target-specific module so that target-specific code may be placed before or after the the core is entered. For example, the RFM implements a special system choice screen which is called before any of the target-generic GUI modules. `<targ>_main.c` also includes the header file which initializes `fdefs` and the menu bars. `main()` calls functions in `init.c`, which initialize `state`. `init.c` calls functions
in `<targ>_init.c` to initialize state->tspec, or anything else which requires target-specific initialization (such as the PC's COM port).

If the PC has two COM ports, the user may choose which COM port a particular Configurator will use. The `get_comport` function in `init.c` reads the initialization file `CNFRGRTR.INI`, which contains the COM port settings for each Configurator. Different targets may be set to different COM ports (for PC's that have two ports). For example, the CU may be connected to COM1 and the RFM to COM2.

With everything initialized appropriately, `main` calls the function `mode_loop` in `loop.c`. This begins the `mode loop` in which the menu bar and parameter buffer are repainted on every mode switch. `mode_loop` calls code in `windefs.c` to define the menu bar and parameter buffer from `fdefs` and the menu bar tables. `mode_loop` then calls `sed_Go` to pass control to the first field in the parameter buffer. The mode loop continues or exits depending on the return value of `sed_Go`.

`sed_Go` uses `field functions` of the various field types to handle the editing of parameter values and cursor movement within the parameter buffer. The modules `fnd.c`, `fnh.c`, and `fnl.c` define the field functions for all parameter buffer fields. `fngn.c` and `list.c` contain auxiliary field function code, and `fndentry.c` defines how the user can access the menu bar.

Control remains with the field functions (or functions called from the menu bar) until the user chooses the "Quit" option from the menu bar. At this point, `sed_Go` quits and the mode loop exits, returning control to the module `<targ>_main.c`.

Throughout the GUI there are calls to helping functions in the modules `conv.c` and `dialog.c`. `conv.c` contains hex conversion utilities, and `dialog.c` provides messages to the user.

<table>
<thead>
<tr>
<th>Table G. Modules of the GUI.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GUI startup</strong></td>
</tr>
<tr>
<td><code>&lt;targ&gt;_main.c</code></td>
</tr>
<tr>
<td><code>init.c</code></td>
</tr>
<tr>
<td><code>&lt;targ&gt;_init.c</code></td>
</tr>
<tr>
<td><strong>mode loop</strong></td>
</tr>
<tr>
<td><code>loop.c</code></td>
</tr>
</tbody>
</table>

45
**Table G. Modules of the GUI. (continued)**

<table>
<thead>
<tr>
<th>screen and menu defs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>windefs.c</code></td>
</tr>
<tr>
<td><code>&lt;targ&gt;_win.c</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>field functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fnd.c</code></td>
</tr>
<tr>
<td><code>fnh.c</code></td>
</tr>
<tr>
<td><code>fnl.c</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>field function aux</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fndentry.c</code></td>
</tr>
<tr>
<td><code>fngn.c</code></td>
</tr>
<tr>
<td><code>list.c</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>helping</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conv.c</code></td>
</tr>
<tr>
<td><code>dialog.c</code></td>
</tr>
</tbody>
</table>

### 3.4.2 Menu bar functions

Apart from those in the GUI, the only modules in the Configurator are the ones called from the menu bar. The two menus common to all three Configurators are the File and `<targ>` menus. The File menu reads and writes parameter values to files, and the `<targ>` menu reads and writes parameter values through the PC's serial port to the target. To do their jobs, both menus require the ability to convert parameter values to their “native” byte values.

### Configuration parameters

There are two perspectives in considering the task of configuration. On one side there is the target board, which “thinks” strictly in terms of NVRAM values. Configuration parameters don’t exist; only values in memory which get read and written and have some subsequent effect on the board’s operation. On the other side there is the user, who thinks in terms of configuration parameters. A parameter’s title, such as “PES Present,” has some meaning, as do the possible values of the parameter (“present” or “not present,” in this case). The user does not need to know anything about where or how the parameter is represented in the target’s NVRAM; meaning is achieved by abstracting away
the details. The Configurator achieves this abstraction by revealing only the information which has some meaning to the user. The Configurator provides the user an interface to the target by relating configuration parameters to NVRAM locations, and values of configuration parameters to NVRAM values. Figure 16 shows an example of this.

![Figure 16. Relationship between configuration parameters and NVRAM.](image)

NVRAM values may be related to parameter values in different ways. For example, suppose a byte of NVRAM contains the value 3. Depending on the interpretation of that value as a configuration parameter, the value may refer to 3 seconds, 3 MHz, choice #3 in a list, or it may be a number without any units. How the byte is interpreted is known as the mapping of the value from NVRAM to a configuration parameter. Conversely, there also exist mappings from configuration parameters to NVRAM.

More accurately, each configuration parameter has a mapping function, which takes as input the value of the parameter and outputs the appropriate NVRAM value. Each NVRAM value has a mapping function which maps it to a value for its corresponding configuration parameter. It is important to note that one configuration parameter may map to one or more NVRAM locations; therefore, configuration parameters may map to more than one NVRAM value, and a set of NVRAM values may collectively map to one configuration parameter.

Mapping functions are the common thread linking the three targets. Every mapping function maps an NVRAM value to a configuration parameter value in one of four ways:

1. A small number of discrete values map to a list of character strings. For example, in the RFM, the value 0x00 at offset 0x16 maps to "constant power," and the value 0x01 maps to "constant gain."
2. The identity function: the NVRAM value is the same as the configuration parameter value. For example, in the RFM, if NVRAM offset 0x20 has a value of 0x1ABC, the configuration parameter, ODU Minimum Measurable Power, has a value of 0x1ABC. Both have the same hexadecimal value. (Note that in this case, two bytes of NVRAM map to a single configuration parameter.)

3. Conversion via a simple formula to a particular set of units. NVRAM values are represented in hex format, but some configuration parameters may be represented in decimal units such as seconds or decibels (dB). The map may only involve adding the correct units to the end of the value, or it may require division by a power of ten. For example, if RFM offset 0x10 contains 0x22, it maps to a Keep Alive Time of 34 seconds. If Keep Alive Time had units of tenths of seconds, an NVRAM value of 0x22 would map to 3.4 seconds.

4. Conversion via a complex formula to a particular set of units. For example, the RFM's Transponder Window is a function of an NVRAM value and the type of ODU connected to the IDU.

Since every mapping function falls into one of these four categories, the core maps most configuration parameters to NVRAM using only target-generic code. (Mapping functions of type 4 must be written with target-specific code.) Supplemented with some target-specific code, then, the core can hide from the installer any information about the target's NVRAM.

The module params.c contains the parameter mapping functions. fill_parms maps NVRAM values to parameter values, and fill_nvram_array maps parameter values to NVRAM values. Both functions contain entry points to target-specific code, which allows for parameters with non-standard maps (like type 4 above).

Different targets have different ways of formatting multi-byte parameters in NVRAM. Depending on the architecture, the most significant byte may occupy the lowest or the highest address. Whether a target uses "big endian" or "little endian" addressing is defined by a macro in a target-specific header file.

**File and <targ> menus**

As stated above, both File and <targ> menus require the mapping functions provided in params.c. When reading values from NVRAM or a saved file, fill_parms is called to fill the parameter buffer with mapped parameter values. When writing parameter values, fill_nvram_array is called to convert the parameters to the byte values which will be written to the file or NVRAM. Before writ-
ing parameter values, the check.c module is used to check the current parameter values against any default values defined in fdefs. The user is notified of any non-default values and is allowed to continue or cancel the write operation.

When writing values to target NVRAM, target-specific code in <targ>_rw.c is called to implement the target’s command set and serial line protocol. <targ>_reset.c contains code for resetting the target after a write to NVRAM, if necessary. The tell.c and io.c modules handle general serial line communications.

**Display menu**

This menu contains an option which creates a window displaying NVRAM values in their native hexadecimal format. The parms.c module is not called, since parameter values are not read or written.

Table H summarizes the modules called from the menu bar. Together, Tables G and H list every module in the core and every target-specific module called directly from the core.

<table>
<thead>
<tr>
<th>Table H. Modules called from the menu bar.</th>
</tr>
</thead>
<tbody>
<tr>
<td>check.c</td>
</tr>
<tr>
<td>display.c</td>
</tr>
<tr>
<td>file.c</td>
</tr>
<tr>
<td>fnch.c</td>
</tr>
<tr>
<td>io.c</td>
</tr>
<tr>
<td>ll.c</td>
</tr>
<tr>
<td>parms.c</td>
</tr>
<tr>
<td>targ_rw.c</td>
</tr>
<tr>
<td>&lt;targ&gt;parms.c</td>
</tr>
<tr>
<td>&lt;targ&gt;reset.c</td>
</tr>
<tr>
<td>&lt;targ&gt;_rw.c</td>
</tr>
<tr>
<td>tell.c</td>
</tr>
</tbody>
</table>

### 3.5 Target-specific implementations

This section details the target-specific modules contained in each of the three Configurators.

#### 3.5.1 RFM Configurator

The RFM Configurator had the highest priority for development. Because of this, development focused on implementing features required by the RFM. This resulted in the RFM Configurator having
the most target-specific features of the three Configurators. In addition to the standard File, <targ>, and Display menus, the RFM Configurator's menu bar contains Calculate and M&C menus, and an extra option on the Display menu.

**Calculate menu**
Some configuration parameters are mapped to NVRAM values by a function which takes inputs from the user. Previous configuration programs required the user to calculate the function by hand and manually input the result. The Calculate menu options allow the user to input the arguments to the mapping function using a well-designed interface. Once the value is calculated, the user may do any of three things:

- Calculate the value again with other inputs.
- Type ALT-W to write the answer value to the appropriate field in the parameter buffer and close the calculation window.
- Type ESC to close the window, without having affected the parameter buffer in any way.

Four of the RFM's configuration parameters may be calculated with calculation screens. Figure 17 shows the KCM Rx Channel calculation screen. In the figure, the user has chosen satellite type Aus-sat and window number 11 as the inputs to the calculation function; the resulting value is 695.

**M&C menu**
The M&C menu is provided to meet the RFM's M&C requirements. As discussed in Chapter 2, M&C includes any functions which monitor or control the target in real time. The menu options create screens to set the 10 MHz Ref and Mode parameters, and another menu option is provided to reset the RFM.

Table I lists every RFM-specific module.
Figure 17. The RFM Configurator's KCM Rx Channel calculation.

Table I. RFM-specific modules.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc_alc.c</td>
<td>Creates window of Startup ALC Level and Commissioned Tx Gain calculation.</td>
</tr>
<tr>
<td>calc_rx.s</td>
<td>Creates window for KCM Rx Channel calculation.</td>
</tr>
<tr>
<td>calc_txw.c</td>
<td>Creates window for Transponder Window calculation.</td>
</tr>
<tr>
<td>fnalc.c</td>
<td>Field functions for the Startup ALC Level calculation window.</td>
</tr>
<tr>
<td>fnrxc.c</td>
<td>Field functions for the KCM Rx Channel calculation window.</td>
</tr>
<tr>
<td>fntxw.c</td>
<td>Field functions for the Transponder Window calculation window.</td>
</tr>
<tr>
<td>mode.c</td>
<td>Handles the “Set IFM/RFM mode” menu option.</td>
</tr>
<tr>
<td>rfm_disp.c</td>
<td>Displays ODU information.</td>
</tr>
<tr>
<td>rfm_init.c</td>
<td>RFM-specific initializations of state and the COM port.</td>
</tr>
<tr>
<td>rfm_main.c</td>
<td>RFM-specific main(). Calls system_choice before beginning the mode loop.</td>
</tr>
<tr>
<td>rfm_rw.c</td>
<td>RFM communications. Implements RFM command set.</td>
</tr>
<tr>
<td>rfm_win.c</td>
<td>RFM-specific menu bar and parameter buffer painting.</td>
</tr>
<tr>
<td>rfmparms.c</td>
<td>RFM-specific parameter mapping. Calls functions in txw.c to handle the mapping of Transponder Window, the only RFM parameter with a nonstandard map.</td>
</tr>
<tr>
<td>rfmreset.c</td>
<td>RFM-specific reset code.</td>
</tr>
</tbody>
</table>
Table I. RFM–specific modules. (continued)

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sch.c</td>
<td>Prompts user for IDU and ODU info.</td>
</tr>
<tr>
<td>sq.c</td>
<td>Downloads files to the RFM.</td>
</tr>
<tr>
<td>table.c</td>
<td>Functions pertaining to the table data type.</td>
</tr>
<tr>
<td>tmr.c</td>
<td>Implements the M&amp;C function for the 10 MHz Ref parameter.</td>
</tr>
<tr>
<td>txw.c</td>
<td>Contains all the code related to the Transponder Window parameter.</td>
</tr>
</tbody>
</table>

3.5.2 CU Configurator

As discussed in Chapter 2, the CU may be in either BREAK state or RUN state. To accommodate this, the M&C menu allows the user to force the CU into either state. There is no Calculate menu. Table J lists the CU–specific modules in the CU Configurator. Figure 18 shows the CU Configurator; notice how similar it is to the RFM Configurator, shown in Figure 14. This consistency in the user interface makes the program easy to use for installers.

Table J. CU–specific modules.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cu_init.c</td>
<td>CU–specific initializations.</td>
</tr>
<tr>
<td>cu_main.c</td>
<td>CU–specific main().</td>
</tr>
<tr>
<td>cu_rw.c</td>
<td>CU communications.</td>
</tr>
<tr>
<td>cu_state.c</td>
<td>Sets the CU's mode.</td>
</tr>
<tr>
<td>cu_win.c</td>
<td>Stub functions.</td>
</tr>
<tr>
<td>cuparms.c</td>
<td>Non–standard parameter mappings.</td>
</tr>
<tr>
<td>cureset.c</td>
<td>Sends the reset command.</td>
</tr>
</tbody>
</table>
3.5.3 IFM Configurator

The IFM Configurator has the least number of target-specific features. It does not have a Calculate menu, and the M&C menu only contains a reset option. Table K lists the IFM-specific modules. Figure 19 shows the IFM Configurator.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifm_init.c</td>
<td>IFM-specific initializations.</td>
</tr>
<tr>
<td>ifm_main.c</td>
<td>IFM-specific main().</td>
</tr>
<tr>
<td>ifm_rw.c</td>
<td>IFM communications.</td>
</tr>
<tr>
<td>ifm_win.c</td>
<td>Stub functions.</td>
</tr>
<tr>
<td>ifmparms.c</td>
<td>Non-standard parameter mappings.</td>
</tr>
<tr>
<td>rfmreset.c</td>
<td>Sends the IFM reset command.</td>
</tr>
</tbody>
</table>
3.6 Summary

This Chapter has described the implementation of the Configurator. The code is divided into target-generic code, known as the core, and target-specific code. The user interface, contained in the core, is designed with a paradigm of simplicity and is consistent between the three Configurators. `defs`, the menu bar tables, and `state` are the three most important data structures. The interaction between the various modules was summarized, and some limitations of the design were discussed. Finally, the target-specific functions of each Configurator were described briefly.
Chapter 4
Related Work and Discussion

An extensive literature search was conducted to determine how the Configurator relates to products or research projects being pursued in places outside HNS. Source materials discuss configurable user interfaces, configuration management, and other “configurators” in industry. Section 4.1 examines the related work, and Section 4.2 suggests specific ideas for improving the Configurator’s design.

4.1 Related Work
This section examines the related work found in the literature search. Where appropriate, the Configurator is compared to the project being discussed.

4.1.1 Menu-driven Interfaces
The literature search discovered two articles about menu-driven interfaces which are relevant to the Configurator. Perhaps the product most similar to the Configurator discovered in the literature search was the home system interface described in [8]. In this system, users can control various electronic components in the home through the use of a menu-driven interface displayed on the TV. The system is designed to be used by users who are not familiar with computers, and it employs many of the principles that comprise the Configurator’s GUI paradigm (see next section). One notable difference, however, is its input device: users do not manipulate a cursor to navigate through the menu choices. Instead, they use a keypad with twelve buttons, each one of which corresponds to one menu option. This method of input is more easily learned and used by users who are less experienced with computers. Changing the Configurator’s interface to support this style of input would require fundamental changes to the design; they are discussed later on.

Greene et al. [11] details a study conducted on human–computer interaction with menu-driven interfaces. In the experiment, subjects were tested on their proficiency in quickly and accurately entering large amounts of data into different menu-driven interfaces. It was found that “for both experienced and inexperienced users, the most effective method was entry combined with autocompletion (i.e., string completion). Users took the least amount of time to carry out tasks with this method and also preferred it over both entry without autocompletion and selection from either a one-item menu or an N-item menu.” This would seem to suggest that the Configurator would benefit by replacing its N-item menus (list parameters, for example) with entry fields supplemented by autocompletion. The benefits of making such a design change, however, are doubtful. Data entry in the Configurator is not optimized for speed (as fields with autocompletion are), and autocompletion would only be help-
ful to installers who had memorized the choices of the list field. Nonetheless, this study could be relevant in the future if the Configurator ever implements a field type whose data entry could benefit from autocompletion.

4.1.2 User interface principles

Several sources discussed general interface design principles. Schneiderman [17] gives a rigorous treatment of all aspects of user interfaces on computers, while Norman [14] discusses design of ordinary objects from a psychological perspective. [8] (discussed above) and [1] discuss user interfaces designed, like the Configurator, mainly for computer novices.

The three sources concerning computer interfaces share one important design principles in common:

- **Consistency.** The operation of the GUI components should function the same throughout the interface. Actions, inter-field navigations, and keystrokes should all be handled in a consistent manner.

This is an extremely important principle in the Configurator’s GUI. The limitations placed on the program may force the interface to be less than optimal, but no matter how crude the interface, it must be consistent. The user must be able to predict what the cursor will do on a given keystroke. This principle is met by handling all numeric field keystrokes in one module, and all list field keystrokes in one module.

All four sources listed above shared these three principles:

- **Simplicity.** Words and phrases should be displayed in the simple, clear, language. Also, the user must be able to carry out an operation with the minimum number of input actions.

- **Feedback.** The user must be kept informed of what is going on in the program.

- **Error tolerance.** Whenever possible, the user must be prevented from making harmful errors.

Simplicity is attained in the Configurator by adding units to numeric fields and using English phrases for list fields, parameter titles, menu options, and dialog messages. The use of technician mode also serves to reduce the amount of superfluous information in the installer’s interface. Feedback is handled well in the Configurator by displaying a message at any time the user does not have cursor control (during target communications, for example).

Error tolerance is the area in which the Configurator could be improved. The user is warned before taking certain potentially harmful actions (like writing a nonstandard value), but these are not
enough. More errors could be prevented if the Configurator had greater knowledge about the target and the relationships between its configuration parameters. One example is that some parameters have nonstandard values which are functions of other parameter values. The default value check could be extended to take these relationships into account. More possibilities for improvement in this area are discussed below, in the discussion about expert systems.

Another principle relevant to the Configurator is the avoidance of “creeping featurism,” described in [14] as “the tendency to add to the number of features that a device can do, often extending the number beyond all reason.” As the Configurator project has developed, HNS lab technicians have proposed a number of features that, had they been implemented, could have been seen as creeping featurism. Efforts were made to implement only those features which were most important to the users, so as to reduce code size and the complexity of the menu bar. As the project continues to grow, it will become increasingly important to maintain the simplicity of the program.

Of the other relevant design issues listed in the above sources, the most important is software configurability, which is addressed next.

4.1.3 Configuration Management

The last section discussed the sources relevant to the way in which the Configurator interfaces with installers and lab technicians. This section describes the sources which relate to the job of the code maintainer. [1] lists three design principles which apply to the code structure of a program with a GUI. Here they are, quoted from the article:

- Common specifications — There should be a single interface specification for all applications.
- Reuse — Entire software modules should be reused across applications whenever possible.
- Customizability — Build in the ability to customize the interface for the specific needs and tasks of a particular user.

Common specifications and module reuse are attained by basing every Configurator on a common core. The Configurator also provides customizability, if one views the “particular user” as a user (installer, etc.) of a particular target board. RFM–specific code, for example, customizes the Configurator for users of the RFM. Customizability based on personal qualities (like the user’s computer expertise) is discussed above.

It is encouraging to see that the Configurator satisfies the above principles; however, they are only the most basic guidelines in designing the code. Viewing the issue of code maintenance from another
perspective—that of configuration management—permits a deeper investigation and yields several suggestions for improvement.

In [10], Gentleman gives this definition of configuration management:

> Configuration management is a term used to describe two activities in building complex systems. One is managing the development of the components from which a complex system is built, and integrating those components to form that system. Alternatively, the term is used to describe the selection of appropriate components to produce a particular instance from a family of related complex systems.

[9] and [10] examine both activities, especially with respect to multi-installation programs. Like the Configurator, these programs must tailor a common functionality to the specific requirements of each target environment, thereby requiring a sound configuration management strategy.

**Target-specific maintenance**

[9] describes two ways of achieving portability, both of which are employed by the Configurator. First, “the most widely used technique for making programs configurable is the static configuration table. Data structures are provided for things that might differ in different installations of the program, and the installer is responsible for providing appropriately initialized instances for a specific installation.” The fdefs table is a prime example of this method. The second method is the use of “indirect naming conventions. A common portability technique has been to use a generic name for a function, e.g. serial_put_byte(), that can be used at all points of call, then to provide alternate implementations corresponding to the [separate targets].” This, of course, is exactly how the core provides entry points to target-specific code.

The article goes on to describe another technique “where the names of the alternate functions are unique, e.g., i8274_put_byte() and s2681_put_byte(), but there is a data structure whose fields contain names of functions to be called to perform required actions, e.g., p->PUT_BYTE(), so that an instance of this data structure can be set up for the specific device to be used at this point.” This idea suggests an improvement to the Configurator (see below).

Although static configuration tables may be the most common solution, they have some important shortcomings. The problems, Gentleman writes, are that “because they are purely declarative with no language-defined semantics, constraint verification and consistency checking can be difficult, let alone error checking.” One way of getting around these faults is to add a layer between the code maintainer and the tables. Gentleman writes:

> Sometimes configuration records are not directly prepared as initialization records in the programming language of the system, but rather are produced as database entries or expressed
as sentences in a grammar, with some tool provided to generate from these the programming language records the system will actually use. This can be particularly useful when several programs need to be implemented for the same configurations.

Section 4.2.1 examines the various possibilities for such a code generation tool.

**File maintenance**

Gentleman's second definition of configuration management, discussed in [10], concerns the process of developing software for multiple targets. Instead of maintaining the target-specific code, the strategies in [10] maintain the files which comprise a Configurator. These strategies relate to the Configurator's design on the module level.

The prime objectives of the project (named Database and Selectors Cel, or DaSC) are to "deal separately with the two principal problems: managing the many versions of a program that exist at any one time and managing the evolution of those versions over time." This is accomplished by creating "a database of source code for a family of programs, each version defined by a (possibly ordered) set of selectors that identifies in the database the particular granules of code that make up that version." The granules which comprise a program are contained in an "inclusion file" (roughly equal to a "selector," mentioned above), which contains the names of all source and header files used by the project in `#include` lines. `#include` commands are only contained in inclusion files, not in source or header files.

The database of selectors is implemented in the file system of the development platform. Core (or "master") code is stored in one branch of the file tree, target-specific code in a different branch, and selector sets defining the composition of the targets in another branch. This clearly demarcates the modular divisions between target-generic and target-specific code. Finally, a set of specially written utilities manages the database.

The DaSC project is mentioned because as the Configurator project grows, some improvement will have to be made to the maintenance of its files. Currently, target-specific files are contained in the same directory as core files; in most cases there is no immediate way of knowing which category a file falls into, let alone what version of what Configurator it belongs to. As more files (and, potentially, more targets) are added to the project, keeping the files organized will become increasingly difficult.

**4.1.4 Other configurators**

Surprisingly, the literature search turned up several sources describing configuration programs used in industry. [3], [20], [2], and [18] describe configurators which are remarkably similar in function. Rather than being a small program for setting parameters in a piece of hardware, these are mammoth,
knowledge-based expert systems built for configuring a set of system components (sold commercially by the company) to fit a given specification. For example, the IPCS configurator described in [3] has the problem of first responding quickly and accurately to enquiries [sic] for such equipment and, following the taking of such an order, ensuring that the delivered equipment is complete, mutually compatible and meets the customer's requirements. A configuring system translates a high level description of a functional requirement into a low level listing of all the hardware and software that must be provided to meet that requirement.

Although at first glance this type of configurator may seem not to have much in common with the HNS Configurator, there are some relevant similarities. Specifically, the knowledge-based configurators contain adaptive software which is configurable with acquired knowledge. The operation of the program is defined by the knowledge base, a set of rules and constraints. Like target-specific code, knowledge of different products may be used to "teach" the program how to configure different products. Possible applications to the Configurator are discussed in Section 4.2.2.

4.1.5 M&C in industry

As the Configurator project grows, more functionality for each target will be included in each Configurator. The most likely area of expansion is that of M&C. Sources were acquired which describe the state of the art of M&C in the manufacturing industry. M&C in manufacturing brings together real-time processing of data and the design of user interfaces for a class of users similar to the installers who use the Configurator. Even if the products themselves are not applicable to the Configurator, the design of their interfaces is applicable.

M&C on the shop floor is typically controlled by Programmable Logic Controllers (PLC's). A PLC is a low-power computer (an 8-bit CPU running at 1 MHz) supplemented by several I/O boards and dedicated to processing relatively large amounts of real-time data [19]. Manufacturing processes are thus monitored and controlled by sets of PLC's. A PLC, in turn, may be controlled by an operator using an operator interface (OI). An operator interface panel (OIP) is a "terminal" which takes input from the operator and relays it to the PLC. Data is usually input to an OIP through the use of buttons, a touch-sensitive screen, or both.

The software which runs on an OIP is written for its particular (physical) user interface and the computing environment present in the shop; such software would never be found, for instance, on a PC. Nonetheless, the design principles of the OI remain the same: the two "graphic design guidelines" listed in [4] are consistency and simplicity. The properties of an OI, in fact, may be more relevant to the Configurator interface than those described in the more intensive GUI studies cited above. OI's are modelled for users with the same computer background as the Configurator installers.
While interfaces running on OIP’s are relevant to the Configurator, it is even more applicable to examine operator interfaces which run on PC’s. As PC’s become more versatile and less expensive, they are increasingly replacing the OIP’s as the OI provider on the shop floor [12]. [5] notes that “the technology gaining the most ground in PLC factory–floor territory is the personal computer. There’s no shortage of software that lets a PC do a PLC’s job.” It is this software which is most applicable to the Configurator.

4.2 Suggestions for improvement
This section presents specific suggestions for improving the Configurator’s design. Most are based on the related work discussed in the last section. Some suggestions could be implemented under the present circumstances; others would require installers to abandon the current PC baseline.

4.2.1 Improvements using current hardware
The installer’s PC makes many improvements to the Configurator impossible. Specifically, any modification which significantly increases the code size is unacceptable. Minor additions to the code, however, are possible. Despite the code size requirement, there are essentially no limitations on how complex the development environment can be. Any improvement which would make the job of the code maintainer easier is viable. This section discusses suggestions for design improvements, given the current limitations of the installer’s PC.

User interface design
As shown above, the Configurator satisfies most of the major GUI design principles listed in [1], [8], [14], and [17]. One other interface principle listed by [1] and [17], however, is worth mentioning: flexibility for users with varying amounts of expertise. Separating the GUI into two modes partially addresses this issue, but more could be done, especially for experienced users. More key bindings could be defined to handle the most complex operations. The GUI could be personalized by individual users through the use of macros.

Code maintenance
A simple improvement comes from [9], which suggests including function pointers in configuration tables. The only tables in the Configurator which contain such function pointers are the MenuBar objects, which contain the (arbitrary) names of menu bar functions. One way of improving the Configurator would be to include function pointers in fdefs for parameters with nonstandard maps. Each mapping function could access any necessary input data by taking state as its one argument. Using this method would eliminate the need for the explicit hooks to target–specific maps (fill_parms_nonstandard and fill_nvram_nonstandard) and instead contain the information implicitly in the fdefs table.
A more important suggestion from [9] is a tool for automatic code generation, which would create a layer between the tables and the code maintainer. A scripting language could be used to implement such a tool, which would mainly be used in the maintenance of tables, not target-specific functions. A maintenance "shell" could be written in the scripting language to interpret input on the command line. Commands would probably be included to view, add, and delete members from a table. For example, interaction with the shell might look like the following ("maint>" is the command prompt):

```
maint> view sat
Table sat: Supported satellite types.
-----------------------------
  name     gain
-----------------------------
Eutelsat  1500
JCSat     1748
Aussat    1750
C-band    2225
-----------------------------
maint> add sat "US Dom Ku" 2300
Satellite type "US Dom Ku" has been added to the sat table.
maint> build RFM
Building RFM Configurator...
Done. You may now run C:\RFM.EXE.
maint>
```

The problem with maintaining tables this way is that it may not be easier than modifying the code itself. For example, consider the sat table:

```
Table sat[] = {
  { "Eutelsat", 1500.0 },
  { "JCSat", 1748.0 },
  { "Aussat", 1750.0 },
  { "C-band", 2225.0 },
  { "US Dom Ku", 2300.0 },
  { NULL }
};
```

The syntax of Configurator tables is not very complex. Without the maintenance shell, code maintainers can modify tables directly, without having to learn or bother with shell commands. Error correction is essentially handled by the compiler, which flags syntax errors.
Nonetheless, there are several reasons why a script might be a good idea. The most important reason is that the shell would be understandable to users regardless of computer skill; lab technicians themselves could perform most of the duties of code maintainer, most notably those which involve statically defined tables. The job of code maintainer could thus be reduced to a peripheral duty of a software engineer on another project.

Other reasons for using a script focus on possible future changes to the Configurator. A script would be extendible to accommodate any changes which might dissuade maintainers from directly editing configuration tables. For example, if the table data type was ever changed, or if other statically defined types were added, the script could hide the details from the maintainer.

Another advantage of a script would be its enhanced error correction. The compiler can only check a table’s contents syntactically, while a script could use an arbitrary set of constraints. For example, the script could define a range in which antenna gains must fall. The shell would output an error message if the user entered an out-of-range value.

Finally, the script could be implemented elegantly. If written in an object-oriented scripting language such as Python (see [16]), shell commands would map to member functions of the class being accessed (table, for example). Python also offers other advantages, such as diverse string-handling utilities, which would ease the implementation of the shell.

There is one last suggestion for target-specific maintenance which bears mentioning. Maintaining the fdefs table by using a script or directly modifying the definition is inherently problematic. fdefs, which contains the definition of the parameter buffer, is a textual representation of a graphical entity. Modifying it with a script or text editor maps textual actions to graphical results. Instead, it would be ideal to modify fdefs through direct manipulation (see [17]), by which fdefs would be edited graphically. A graphical fdefs editor would resemble the parameter buffer, but would provide different commands — perhaps on the menu bar — to allow the user to edit the appearance of the fields. Everything in fdefs would be modifiable through actions on the screen: parameter titles, the appearance of data entry fields, etc.

There are two ways in which direct manipulation of fdefs could be implemented. One is by creating a new program using the GUI core of the existing Configurator. This fdefs editor (essentially a “meta-Configurator”) would have a menu bar specially written to edit the parameter buffer and create files containing the new fdefs definition. This approach would take some time to implement, but it would not affect the existing core of the Configurator, since the format of fdefs would
remain the same. Also, code maintainers could still modify the fdefs table definition with a text editor if they so desired.

The other way to create an fdefs editor would be to use the C-scape utility designed for editing by direct manipulation, Look and Feel. Look and Feel works much the same way as the implementation proposed in the preceding paragraph: a menu bar contains functions for editing the buffer below. The main difference is that the format of the output files (called “screen files”) is defined by C-scape. The Configurator core would have to be modified to be able to read screen files. fdefs would be stored implicitly in the screen file, so it would not be able to be edited outside of Look and Feel. The advantage of such an implementation would be that the editing utility would not have to be written; time would only have to be spent modifying the core to accommodate the changes.

The reader may notice that this section has focused on maintenance of tables and has only mentioned target-specific functions in passing. This is because it is nearly impossible to add a layer between the code maintainer and the target-specific function without decreasing the level of flexibility the target-specific code needs. The issue at stake is the degree to which target-specific information may be parameterized in the core. For example, it is easy to parameterize the names of the configuration parameters in the fdefs table; but how could fdefs be changed to include the information needed, for example, to create a new Calculation menu option? Currently, scalar parameterization and simple function parameterization (with function pointers) seems to be the realistic limit of tables. [6] explores the nature and limits of parameterization, especially as it applies to configurable user interfaces.

The DaSC model, discussed in [10], suggests improvements to file maintenance. Currently, standard makefiles are used in code development. The development platform, Microsoft Visual C/C++, organizes files into projects, each of which has a dedicated makefile. Using the DaSC model would be better, however, because file organization could be independent of the compiler used for development, and the evolution of a Configurator would be maintained by saving all version information (which the Microsoft tool does not do). Perhaps most importantly, DaSC is implemented by using the existing functionality of the resident file system. An expensive configuration management tool is unnecessary; only the utilities which handle the selector files are needed. Configurator file management does not have to take the exact form described in the DaSC paper, but would be a good idea use the DaSC ideas to some degree, since it is designed especially to handle the separation from core and target-specific code present in the Configurator.
Object-oriented design
The suggestion which may have the greatest potential for the Configurator is to convert it to an object-oriented design. The logical choice for a new language would be C++, since it would be compatible with the C-scape libraries, which are written in C.

Making the Configurator object-oriented would solve many of the problems associated with the core/target-specific interface (see Section 3.3.4). In particular, target-specific code would not be forced to conform to the interfaces defined by the core. Instead, the core would be implemented as a collection of classes whose objects and methods could be drawn upon as required by the target-specific code.

The information contained privately in core objects would roughly correspond to the code in the core now. The member functions of core classes would define a clean interface with target-specific code. For example, the core could contain a MenuBar class, which might define functions for adding menu options and running the menu bar. The appearance and keystroke handling of the menu bar would be internal to the class, so that menu bars would remain consistent among the various Configurators. Where or how a target used a menu bar, however, would depend on how the target-specific code made use of the member functions.

Converting the Configurator to C++ would do more than correct the flaws of the current design; it would extend the design to solve a wider range of problems. For example, a Configurator could be built for virtually any target board. No matter how complex a target's configuration parameters or mapping functions, a Configurator could be written which would offer the same, consistent user interface.

4.2.2 Improvements using any hardware
The design improvements listed up to this point in the chapter may all be implemented under the current requirements imposed by the installer's PC. The following paragraphs list suggestions for improving the Configurator, given an arbitrarily powerful computer for the installer. These suggestions could only be implemented with cooperation of the installers, who would have to upgrade to better computers, and HNS, which would have to allocate enough money to support the development of the solutions.

Expert systems
There are direct applications from the knowledge-based configurator to the HNS Configurator. In particular, the Configurator could be extended to implement knowledge-based operation. The technical details normally hidden from installers could be used to build a knowledge base about the targets. Some of the knowledge would be constructed from relationships between parameters (for ex-
ample, appropriate ranges of parameters which are functions of other parameters). It would take quite a bit of effort to encode that implicit information into the present design of the Configurator; with the knowledge-based structure, such encodings would be much easier.

With the target knowledge, the Configurator could monitor the installer’s actions and issue warnings or suggestions when appropriate. This, after all, is the job of the knowledge-based expert system, whose “primary goal is to consistently duplicate the results of human experts rather than understand the basic techniques used by experts to arrive at a given solution” [15]. In this case, the Configurator would mimic the guidance that would be given to the installer by an engineer of the target board. A criticism of this suggestion might be that the configuration process is too trivial a problem to bother with an expert system; [15] notes that tasks in which expert systems are employed must have “a logical complexity such that a single human expert could not perform it without help from computers or manuals.” Whether installers would benefit from having an “expert” present is debatable, but it is at least a suggestion worth noting. HNS has funded research in expert systems for network management in the past, so the consideration of such a project is viable.

**M&C improvements**

Several improvements to the Configurator draw upon the use of M&C in the manufacturing industry. With an appropriately upgraded PC, HES installers could run OI software to perform M&C on the target boards. The interface would be specially designed for operators, and relatively little coding would be required to customize the software for the targets’ needs. Writing comparable software within HNS would require the use of graphics libraries much more extensive than C-scape. The disadvantage of PC M&C solutions is that they are expensive and may require expensive I/O boards in the PC. On the other hand, they may be worth investigating further, since several provide graphical interface editors (similar, but superior to, Look and Feel) that could ease maintenance of the configuration program [13]. Furthermore, some M&C programs implement a GUI using multimedia to mimic an expert system: one program could be configured to “record a shutdown procedure [on video], store it away on disk, and provide it for users to call up when they’re not familiar with a procedure. Thus, combined with the Windows help system, it provides services similar to an expert system” [13].

It is unclear whether the Configurator project really needs the functionality provided by PC M&C packages, since there are not many M&C functions required by the three targets. If the Configurator expands to support more targets, or if more M&C functions are added (both of which are possibilities), the M&C packages will become more attractive solutions.
**PDA's**

There is one somewhat radical suggestion concerning the possible upgrade of installer PC's: instead of buying PC's, have the installers buy Personal Digital Assistants (PDA's). PDA's are more lightweight than PC's and inherently use GUI's (because there is no keyboard to support a command-line interface). The interface, in fact, might resemble that of an OIP, since both mainly accept touch-screen input.

Using a PDA would require a total rethinking of the Configurator's user interface. If PDA's were used, interface design might draw on principles from the touch screens used in the manufacturing industry. The design of other simple interfaces, such as the home-based system described in [8], might also provide some guidance.

**4.3 Summary**

This chapter has reconsidered the design of the Configurator. An extensive literature search produced descriptions of related work in the areas of user interface design, configuration management, and expert systems. Many of the ideas contained in these sources could be implemented as improvements to the design of the Configurator. Ultimately, the range of design improvements will be dictated by the hardware in the installer's PC.
Chapter 5
Evaluation

This chapter evaluates the Configurator. Section 5.1 evaluates the response of the users, and Section 5.2 reviews the development process.

5.1 User reactions

The Configurator is already a success at HNS. Customers in Japan, China, and Thailand have been shown demos (and given unofficial copies) of the RFM Configurator and have reacted positively each time. In fact, many installers are already using the RFM Configurator on a regular basis, despite the fact that HNS has not yet made an official release of the program. Technicians also like the program, citing in one case the “warm fuzzy” provided by some of the features. When a demo of the CU Configurator was recently given to customers visiting HNS, they asked when the program would be made available to them. When the IFM Configurator is fully implemented, the reaction will likely be the same.

The enthusiasm generated by the Configurator is clearly caused not by anything wonderful about its interface, but instead by the lack of quality in the programs it replaces. If the installer PC’s supported a mouse, or a hard drive, or even a code size bigger than 256 K, then perhaps a wonderful program could have been produced. Under the circumstances, however, the Configurator turned out to be a good, solid program whose design is guided more by intuition and common sense than any groundbreaking research.

If there is anything especially notable about the Configurator, it is its flexibility. Technicians and software engineers alike can make modifications to the program at the level they are most comfortable with. Technicians can edit configuration tables, and software engineers can add new windows and features in target-specific functions. Perhaps more impressive is the ease with which new Configurators may be built for additional targets. The basic construction of the last of the three programs — the IFM Configurator — took only ninety minutes, and most of that time was taken by typing in the information for the lengthy list of IFM configuration parameters. (Additional time was needed to write the IFM-specific I/O code.) Basic GUI and file capability could be provided for an arbitrary target in a similar amount of time.

5.2 Design process

Configurability of different targets was not the original goal of the Configurator. The project was initially created to build an RFM Configurator, to replace the Term Protocol, and possibly a CU Con-
figurator if time permitted. At that time it was believed that installers would not want a replacement of the IFM's Configuration Editor, having become accustomed to it over nine years of use. When the first prototypes of the RFM Configurator were shown in September, however, it became clear that installers would be willing to endure a learning curve in exchange for such an improvement in the interface. Thus the IFM Configurator was added as an “time permitting” item. From that point on, the design of the Configurator laid the foundation for the core/target-specific separation that would eventually come, even if it came after the internship was over.

The project hit a crucial point in late autumn. Lab technicians were requesting new features for the RFM Configurator to the detriment of development of the other Configurators. Technicians wanted depth-first development, so that the RFM would get the attention it deserved, the project itself having been created for an RFM Configurator. Breadth-first development — refining the core and implementing the three targets in parallel — was clearly the best policy for the thesis’ sake. A compromise was reached. All three Configurators would be implemented, but with varying degrees of functionality. The RFM Configurator would offer enough functionality to merit an official release to HES installers, the CU Configurator would implement some CU-specific functionality, and the IFM Configurator would mainly serve as a starting point for future development.

Those goals were not only met but surpassed by the fact that the CU Configurator, as it currently stands, may be released to installers. Both the RFM and CU Configurators implement enough functionality to essentially replace their respective predecessors. The goal for the IFM Configurator was also met; much work remains to be done before it replaces the CE, but the crucial first step has been taken.

How well the RAM size requirement was met is debatable. The RFM Configurator is a few dozen kilobytes larger than the limit, the CU Configurator is around the limit, and the IFM Configurator is well below it. (It was never determined exactly how much space in an executable file actually gets loaded into RAM on the Toshiba 1000, or how much RAM is taken up by the installers’ version of DOS; sizes of executables were compared to the RAM requirement with educated guesses.) Although this would seem to imply that the RFM and CU Configurators could not be released for installer use, this is not the case. Most HES installations currently occur in Asia, where installers generally have better PC’s than the Toshiba 1000. The RFM and CU Configurators will be released, therefore, regardless if they are a few kilobytes larger than the baseline requirement. Installers may upgrade their PC’s soon, or if some installers insist on using the large Configurators on Toshiba 1000’s, superfluous modules (like the RFM’s Calculation menu) may be deleted or moved to a separate executable. It is most likely, however, that the random nature of target development (see next paragraph) will pre-
vent these code sizes from ever being a problem. (The requirements for a small program were quite stringent at the beginning of the project, and technically still exist today, so the requirement remains in the thesis.)

One challenging aspect in the development of the Configurator was the somewhat haphazard way in which requirements and requests for features were made. The commonalities between the targets were always in flux, causing uncertainty in the domain of the core. Changes to the core — the fundamental structure of the program — were being made even in the final months of the project. The haphazardness, however, was not caused by any fault of the engineers making the requirements; instead, it was caused by the breakneck pace at which target hardware develops under a deadline. When time is tight, “ship it” is a more common phrase than “spend some time getting it just right.” This seems to be the case in most companies, though, and it was useful in adding that much more corporate reality to the thesis experience.

It should be noted that the “haphazardness” of development was mitigated substantially by the intervention of one particular engineer, John Lacey. Lacey acted as the filter of feature requests flowing in from the other engineers, only letting the most important features become “real” requests. Lacey was, in a sense, an intelligent agent, filtering the signal from the noise better than any General Magic product. Furthermore, he was the main source of documentation on the targets and in general an indispensible partner. Although the entirety of the Configurator code was designed and written by the author, Lacey made it a two-man project.

The future of the Configurator project is uncertain. A programmer has been hired temporarily to fill in some of the gaps in the target–specific code, but it is unclear whether he will work on just the CU, or the RFM, or all three of the Configurators. It is unknown whether the IFM Configurator will ever be extended enough to replace the CE, and it is unknown whether any more targets will be added for Configurator support. Probably the most important question, though, is whether the installers will upgrade their PC’s. If they do, how much will they increase their RAM? What processor will they move to? Would they trust the Configurator developers enough to move to PDA’s? The answers to each of these questions will dictate which of the suggestions in the last chapter will ever be implemented.
Chapter 6
Conclusion

The Configurator was proposed as a replacement to three poorly designed programs which help installers configure satellite dish hardware during installation. The Configurator was implemented successfully, providing a graphical user interface optimized for the PC on which it runs, and dual-mode operation for two classes of user. The code is organized into two sections: a core, which controls functionality like the interface shared by all Configurators, and target-specific code, which tailors a Configurator to a particular target. This design makes it easy to perform periodic updates to the code.

There are several ways in which the project could be improved in the future. If the installers do not upgrade their PC's, most of the improvements will be limited to code maintenance and development. A shell script or graphical editor could be used for maintaining the target-specific code. A more ambitious improvement would be to convert the entire project to C++, thereby reaping the benefits of object-oriented design. If installers do upgrade their PC's, the project will be presented with a much wider range of possibilities. The Configurator could be enhanced to include expert system capability, or the interface could be improved with better graphics.

The Configurator was an excellent topic for an internship thesis. Development of the program included drew upon such diverse areas as satellite technology, user interface design, and configuration management. Perhaps more importantly, developing the Configurator was a satisfying effort, because it has surpassed all the initial expectations of the project. It has been used in real-life installations, it has saved the company time and money, and the users are excited about it. Subsequent improvements to the Configurator will extend this success into the future.
References


DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

Some pages in the original document contain color pictures or graphics that will not scan or reproduce well.