Chapter 10

Writing major files

■ Overview

In order for Academic Audit to determine if a student has met the requirements for graduation in his major, it first must be told what those graduation requirements are. This is accomplished through the use of major files. Major files contain information describing the graduation requirements for a particular major. Writing—or coding, as it's often referred to—these files is the single largest task a school must undertake in the implementation of Academic Audit. It need not be difficult, however, if approached methodically.

The following measures, if taken, will make the major-writing process go more smoothly:

1. **Determine who will be responsible for writing and maintaining the files.**
   One key to success in the major-writing process is selecting the right individual(s) for the job. Although the process is similar to writing a computer program, it is not nearly as technical or difficult. The only computer skills required are the ability to use a PC-based word processor or text editing program. This includes the ability to store files on the computer's hard disk and retrieve them when desired. If the person does not have this ability, they should work in close proximity to someone who does and who can assist them.

2. **Schedule regular time each day for coding.**
   Coding major files is a skill best learned by actual experience. For the beginner, the skill will be developed more quickly and retained longer if it is practiced on a daily basis. For example, working on major files 1½ hours per day, 5 days per week, will produce better results than devoting one full day per week to the process.

3. **Establish a file-naming convention in advance.**
   Rules for naming major files are discussed in detail later in this chapter. This section should be reviewed prior to beginning the coding process, and a standard set of rules for the naming of major files should be established. Since in most cases, the names of major files should match the codes used to indicate a student's major (or minor or concentration) in the student's data record, it may be necessary to consult with your school's computer center to determine what these codes are.
CHAPTER 10: WRITING MAJOR FILES

Storing Major Files

4. Read this chapter from beginning to end before starting.
The information in this chapter is presented in a specific order, and to understand each succeeding bit of information, it may be necessary to have already learned information presented earlier. It is therefore advisable to read this chapter from beginning to end before attempting to create major files. Once a general understanding of the process has been gained, this chapter may be used as a reference.

5. Write the core requirements major file first.
“Core requirements” refers to those requirements common to all degree programs offered at a school. These should be coded separately in their own major file. In most cases core requirements will only need to be coded once.

Storing Major Files

Academic Audit major files may stored as normal text files on a network drive, within the SQL database, or both. Where the files are stored is indistinguishable to the end user. In general usage, it is good practice to set Academic Audit to read files from only one location to insure the proper ones are being used.

Regardless of where the files are stored, it is good to think of them as being stored within a directory structure, or “tree.” There is a top-level major files directory, which may contain major files and/or subdirectories. In turn, those subdirectories may contain major files and/or subdirectories. No more than two levels of subdirectories should be used.

CREATING THE DIRECTORY TREE
The structure of the directory tree should be decided upon prior to creating the major files. The purpose of using a directory tree, rather than placing all of the major files in the same directory (as with previous versions of Academic Audit) is so that files can be grouped according to some common property they share. For instance, all of the majors for the College of Engineering may be kept within a single directory.

CONFIGURING THE MAJORS DIRECTORY
The first step is to tell Academic Audit the location of the main directory under which all of the major files will be stored. This directory is referred to as the “majors directory,” and is specified in the IRONSOFT.INI file. (See Majors, page 7-2) Once this directory has been selected, decide how the major files will be grouped into subdirectories. One possible example is to categorize them according to catalog year. In this event, there will be a subdirectory created for each catalog year, as shown below:

```
Top-Level
Majors Directory → MAJORS
     → 1995
     → 1996
     → 1997
     → 1998
```

Subdirectories
CHAPTER 10: WRITING MAJOR FILES

Storing Major Files

Perhaps, within the directory for each catalog year, we would like to further categorize the major files according to their respective colleges within our university, as shown below:

```
MAJORS
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB_ARTS</td>
<td>LIB_ARTS</td>
<td>LIB_ARTS</td>
<td>LIB_ARTS</td>
</tr>
<tr>
<td>ENGNRG</td>
<td>ENGNRG</td>
<td>ENGNRG</td>
<td>ENGNRG</td>
</tr>
<tr>
<td>NURSING</td>
<td>NURSING</td>
<td>NURSING</td>
<td>NURSING</td>
</tr>
<tr>
<td>BUSINESS</td>
<td>BUSINESS</td>
<td>BUSINESS</td>
<td>BUSINESS</td>
</tr>
</tbody>
</table>
```

By using a directory structure like the one above, a user, when selecting a major to audit a student's transcript against, can first select the catalog year and then the appropriate college, and only the major files within that subdirectory will be displayed in the selection box.

MOVING MAJOR FILES BETWEEN THE SQL SERVER AND A PC

During normal operation of Academic Audit / SQL, the major files are stored in the SQL database. During the initial setup of Academic Audit however, when major files are being written and tested, it may be more convenient to keep them in a directory structure on a PC hard drive. (See -developmajors, page 7-13.) Academic Audit provides an easy way to copy an entire directory structure from a local PC to the SQL database, and from the database to the PC.

Uploading Major Files

To copy all or part of the majors directory structure from a local hard drive to the SQL server, open the Setup menu and select Upload requirements files to server... This will open a dialog box like the one at right. Type the path, using colons (:) instead of backslashes (\), of the desired subdirectory, and a file pattern matching the files to be uploaded. The path must be relative to the majors directory specified in IRONSOFT.INI. (If no majors directory is specified, the upload function will not be available.) Select the appropriate file overwriting option, and click on OK. To upload the entire majors directory structure, simply put a single asterisk (*) in the text box and click OK.

Downloading Major Files

To copy all or part of the majors directory structure from the SQL server to a local hard drive, open the Setup menu and select Download requirements files to server... This will open a dialog box similar to the one used for uploading. Type the path, using colons (:) instead of backslashes (\), of the desired subdirectory, and a file pattern matching the files to be downloaded. Select the appropriate file overwriting option, and click on OK. To download the entire majors directory structure, put a single asterisk (*) in the text box and click OK.
CHAPTER 10: WRITING MAJOR FILES

Anatomy of a Major file

CONFIGURING THE STUDENT WINDOW
If a directory tree, rather than a single majors directory is used, the student window should be properly configured to allow the user to select subdirectories. This is done within the IRONSOFT.INI file.

If there is only one level of subdirectories below the majors directory, set the following parameters within the [Policies] section of the IRONSOFT.INI file:

    MajorComponent1Fields=1
    MajorComponent2Fields=0

This causes a single drop-down listbox to be displayed on the student window from which the subdirectory can be selected.

If there are two levels of subdirectories below the majors directory, set the parameters like this:

    MajorComponent1Fields=1
    MajorComponent2Fields=1

This causes two drop-down listboxes to be displayed on the student window. The top one lists the subdirectories in the majors directory, and the second one lists the subdirectories in the subdirectory which is selected in the top drop-down listbox.

LABELING THE DROP-DOWN LISTBOXES
The labels that appear on the student window above each drop-down list box are user-configurable in the IRONSOFT.INI file. To label the top listbox "Year," set the following parameter in IRONSOFT.INI:

    MajorComponent1Name=Year:

Likewise, to label the second listbox "College," set the following parameter in IRONSOFT.INI:

    MajorComponent2Name=College:

■ Anatomy of a Major file

A major file is a simple ASCII file, or text file. It consists of a series of statements. There are four types of statements which make up a major file—comment statements, requirement statements, control statements and preprocessor statements—which will be explained in detail in this chapter. These four types of statements make up the Academic Audit requirements language.

Although Academic Audit contains a built-in text editor for creating and modifying major files, any commercial text editor (such as Windows Notepad or DOS Edit) or word processing program (such as Microsoft Word or Wordperfect) can be used.

■ Comment Statements

Comment statements, referred to simply as comments, are bits of text written into a major file to explain or
clarify the meaning of other statements. A comment can say anything, as long as it begins with a semi-colon. Here is an example of a comment statement:

    ; This is a comment statement.

When Academic Audit encounters a semi-colon in a major file, it knows that *everything between the semi-colon and the end of that line is a comment*, and will ignore it during processing. A comment does not have to begin at the beginning of a line, as the example below illustrates:

    #RESIDENTCHECK "residency.chk" ; Check residency requirement

In the above code sample, everything to the right of the semi-colon will be ignored during processing.

## Requirement Statements

**OVERVIEW**

The bulk of a typical major file is comprised of requirement statements. Requirement statements are used to translate the requirements for a major into a language which Academic Audit can understand. There are two basic parts to a requirement statement: the *requirement name* and the *requirement description*. Following is an example of a simple requirement statement.

```
(English Comp I) ENG-101
```

- The requirement name is a user-defined descriptive name for the requirement. It should be limited to 20 characters in length because only the first 20 characters will print on the audit.
- The requirement name is always contained in curly braces. {}
- The requirement description is a list of *course patterns* and/or non-course criteria connected by valid operators in a way that tells Academic Audit how the requirement may be satisfied.

## Rules for Course Patterns

One of the primary components of a requirement description is the *course pattern*. A course pattern is a string of characters which either describe a single course or a pattern which a number of courses adhere to. In most cases, a course pattern will consist of two parts, a *department code* and a *course number*, connected by a dash. In the code sample below,

```
(English Comp I) ENG-101 ; Must take English 101
```

ENG-101 is a course pattern, "ENG" is the department code (signifying "English"), and "101" is the course number. Every course pattern *must* consist of at least a department code and a course number, however, other optional codes may be included to narrow the criteria a course must satisfy in order to fit the pattern. A *section*
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

code may be appended after the course number, and a campus code may be added before the department code, as in the following example:

    (English Comp I) MC-ENG-101-1 ; English 101 must be taken as
    ; a day class at Main Campus.

Keep these rules in mind when coding course patterns:

- **Course patterns MUST BE CODED IN ALL CAPITAL LETTERS.**
- **Department codes** may have a maximum of 6 **characters**.
- **Course numbers** may have a maximum of 4 **characters**.
- **Section codes** may have a maximum of 2 **characters**.
- **Campus codes** may have a maximum of 2 **characters**. (Note: Campus codes are now obsolete.)
- If Academic Audit finds only one dash (-) in a course pattern, it assumes that the pattern refers to a department and course number. If two dashes are found within the description, it is assumed that the department, course and section are being used. Three dashes indicate that the campus location, department, course and section are all being used. If you want to include a campus code but not a section code, use an asterisk (*) where the section code would normally go, as in this example:

    (English Comp I) MC-ENG-101-* ; English 101 must be taken at
    ; Main Campus.

**USING WILDCARDS IN COURSE PATTERNS**

"Wildcards" are very powerful additions to the Academic Audit requirements language. Wildcards are special symbols which may be used within certain codes of a course pattern to "broaden" the criteria a course must satisfy in order to fit that pattern. The three wildcard symbols that may be used in course patterns are the asterisk (*), the question mark (?), and the greater-than (with an implied equal-to) symbol (>).

* The asterisk is a wildcard symbol which means "any." It can be used in the campus, department, course or section code.

? The question mark can be used as a wildcard for only one position (character) of a campus, department, course or section code. If two positions are wildcarded, they should be coded as "??.”

> The greater-than (with an implied equal-to) symbol can be used only with the course number code.
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

The table below illustrates several examples of the use of wildcards:

<table>
<thead>
<tr>
<th>Course Pattern</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG-*</td>
<td>Any ENG course.</td>
</tr>
<tr>
<td>*-100</td>
<td>Any 100 course.</td>
</tr>
<tr>
<td><em>-</em></td>
<td>Any course at all.</td>
</tr>
<tr>
<td>ENG-2??</td>
<td>Any 200-level course.</td>
</tr>
<tr>
<td>ENG-25?</td>
<td>Any ENG course beginning with 25.</td>
</tr>
<tr>
<td>ENG-&gt;200</td>
<td>Any ENG course greater than or equal to ENG-200.</td>
</tr>
</tbody>
</table>

Using Non-Course Criteria

Many degree programs require a student to meet certain “non-course criteria” in order to graduate. An example of this non-course criteria may be a certain minimum score on a standardized test. It may be a certain number of hours of community service. Perhaps a student may meet a particular non-course criterion in lieu of taking certain courses. In cases like these, the Academic Audit requirements language provides a way to code non-course criteria into requirement statements.

The presence of a non-course criterion in a major file tells Academic Audit to check the contents of a custom field in the student’s database record. (See The Custom Fields Table, page 6-1.) When custom fields are coded in a major file, they are enclosed in double square brackets as in the following formats:

```
[[Fieldname = value]]
[[Fieldname > value]]  ">" means “greater than or equal to"
[[Fieldname < value]]  "<" means “less than or equal to"
```

The character strings which indicate non-course work may be used in a requirement statement anywhere a course pattern may be used. Operators (which are explained below) may be used to combine them with other custom fields or course patterns, as the following example shows:

```
{English Pre-Req}  |(ENG-099 [[SATVERB > 500]])
; Students with a SAT verbal score less than 500
    ; must take remedial English.
```

- Note: When requirements are satisfied with non-course criteria, the name of the user-defined field and the value in it are printed on the audit report in place of the course.
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

Using Logical Operators

OVERVIEW
Logical operators are special symbols used to logically connect course patterns and/or non-course criteria. They are used to code requirements in which a student must take all of the courses in a list of courses, or choose one course from a list, or take a specific number of courses chosen from a list.

Each operator, with the exception of the rarely-used CHECK operator, is followed by a pair of parentheses which contain the course patterns on which the operation is performed. Table 2 below lists the operators included in Academic Audit's requirements language:

Table 2:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; ( )</td>
<td>AND</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( )</td>
<td>OR</td>
<td>CHECK:</td>
</tr>
<tr>
<td>- ( )</td>
<td>NOT</td>
<td>!! ( )</td>
<td>DOUBLE GROUP</td>
</tr>
<tr>
<td>! ( )</td>
<td>GROUP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AND Operator

The AND operator tells Academic Audit that the student must take all of the courses listed inside the parentheses.

Format:

(Requirement Name)&(list of courses, course patterns or non-course criteria)

Examples:

{Science Track} & (CHEM-101 CHEM-102)
; Must take CHEM-101 and CHEM-102

{Humanities} & (HUM-101 HUM-102 ENG-154 PHI-2??)
; Must take HUM-101, HUM-102, ENG-154 and any 200-level PHI course.

OR Operator

The OR operator is denoted by the pipe symbol (|), which is found on the keyboard on the same key as the backslash (\). The OR operator tells Academic Audit that the student must take one of the courses, or satisfy one of the course patterns listed in the parentheses.
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

Format:

(Requirement Name) | (list of courses, course patterns or non-course criteria)

Examples:

[Computer Literacy] | [CIS-100 CIS-120] ; Take CIS-100 or CIS-120

[English Comp I] | [ENG-101 [SATVERB > 700]]
; Satisfied by taking ENG-101 or having a SAT Verbal score
; of 700 or better.

NOT Operator

The NOT operator is denoted by the tilde character (~), which is found on the key to the left of the “1” key. The NOT operator tells Academic Audit that a requirement can be satisfied by any course that fits the first course pattern listed but does NOT fit any listed course patterns which follow the first.

Format:

(Requirement Name) ~(Course-Pattern1 Course-Pattern2
Course-Pattern3... etc.)

Examples:

[History Elective] ~(HIS-* HIS-440 HIS-490)
; Take any History except 440 or 490

[Upper Div. Elective]~(*-4?? ENG-* HIS-*)
; Take any 400-level course except English or History courses.

Course Group Operator

The COURSE GROUP operator has the following format, with N and X representing integers:

\[ \text{(Requirement Name)} \text{ !}(N \text{ X course-group1 course-group2... etc.)} \]

Translated literally, this statement tells Academic Audit, “student must take N courses from the following groups of courses, with no more than X courses from any one group.

Examples:

{4 Fine Arts} !{4 1 ART-* MUS-* THE-* ENG-* LIT-* SPE-* COM-*}
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

```
PHI-* PRE-* GER-* RUS-* SPN-* HON-281 HON-295)
; Take 4 courses from the above groups, no more
; than 1 course from any group.
```

In the above example each course pattern is considered a group of courses. If ENG-* and LIT-* are to be considered one group together, they may be grouped within parentheses as in the example below:

```
{4 Fine Arts} !(4 1 ART-* MUS-* THE-* (ENG-* LIT-*)) SPE-* COM-*
PHI-* FRE-* GER-* RUS-* SPN-* HON-281 HON-295)
```

If the distribution of the courses among groups is not important, the second number should be equal to the first. In the following example, all 4 courses could be taken from any of the groups listed:

```
{4 Fine Arts} !(4 4 ART-* MUS-* THE-* ENG-* LIT-* SPE-* COM-*
PHI-* FRE-* GER-* RUS-* SPN-* HON-281 HON-295)
; Take 4 courses from the above groups.
```

- Note: The two numbers used in a course group operator must be integers. This tells Academic Audit that the operator is referring to a required number of courses rather than a required number of credits.
- Although operators can generally be used in combination with each other, GROUP operators may not have AND's, OR's, or other GROUP operators under them, i.e. inside the parentheses for the GROUP.

Credit Group Operator

The CREDIT GROUP operator has a format similar to the COURSE GROUP operator, except that the numbers used must have at least one decimal place:

```
[Requirement Name] !(N.0 X.0 course-group1 course-group2... etc.)
```

Translated literally, this statement tells Academic Audit, "student must take N.0 credits from the following groups of courses, with no more than X.0 credits from any one group.

Examples:

```
{12cr Fine Arts} !(12.0 3.0 ART-* MUS-* THE-* ENG-* LIT-* SPE-* COM-*
PHI-* FRE-* GER-* RUS-* SPN-* HON-281 HON-295)
; Take 12 credits from the above groups, no more
; than 3 credits from any group.
```

In the above example each course pattern is considered a group of courses. If ENG-* and LIT-* are to be considered one group together, they may be grouped within parentheses as in the example below:

```
{12cr Fine Arts} !(12.0 3.0 ART-* MUS-* THE-* (ENG-* LIT-*)) SPE-*
COM-* PHI-* FRE-* GER-* RUS-* SPN-* HON-281 HON-295)
```
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

If the distribution of the credits among groups is not important, the second number should be equal to the first. In the following example, all 12 credits could be taken from any of the groups listed:

{12cr Fine Arts} !!(12.0 3.0 ART-* MUS-* THE-* (ENG-* LIT-*) SPE-*
COM-* PHI-* FRE-* GER-* RUS-* SPN-* HON-281 HON-295)
; Take 12 credits from the above groups.

- Note: The two numbers used in a course group operator must be floating-point numbers with at least one decimal place. This tells Academic Audit that the operator is referring to a required number of credits rather than a required number of courses.
- Although operators can generally be used in combination with each other, GROUP operators may not have AND's, OR's, or other GROUP operators under them, i.e. inside the parentheses for the GROUP.

Double Course Group Operator

The DOUBLE COURSE GROUP operator has the following format, with N, J, K and L all representing integers:

{Requirement Name} !!(N (J course-group1)
(K course-group2)
(L course-group3)
... etc.
)

Translated literally, this statement tells Academic Audit, "student must take N courses from the following groups of courses, with no more than J courses from course-group1, K courses from course-group2, L courses from course-group3, etc." It is very similar to a normal course group operator, except it provides greater flexibility in how the courses will be distributed among the groups.

Examples:

- {8 Fine Arts} !!(8 (3 ART-*) (3 MUS-*) (1 THE-*))
(2 ENG-*)(2 LIT-*)(2 SPE-*)
; Choose 8 courses from the above groups, no more than 3 Art,
3 Music, 1 Theatre, 2 English, 2 Lit, or 2 Speech.

In the above example each course pattern is considered a group of courses. If ENG-* and LIT-* are to be considered one group together, they may be grouped within parentheses as in the example below:

{8 Fine Arts} !!(8 (3 ART-*) (3 MUS-*) (1 THE-*))
(2 ENG-* LIT-*)(2 SPE-*)
; Choose 8 courses from the above groups, no more than 3 Art,
3 Music, 1 Theatre, 2 English or Lit, or 2 Speech.
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

- Note: Numbers used in a double course group operator must be integers. This tells Academic Audit that the operator is referring to a required number of courses rather than a required number of credits.
- Although operators can generally be used in combination with each other, DOUBLE GROUP operators may not have AND's, OR's, or other GROUP operators under them, i.e. inside the parentheses for the GROUP.

Double Credit Group Operator

The DOUBLE Credit GROUP operator has the following format, with N,N, J,J, K,K and L.L all representing floating-point numbers:

\[
\text{(Requirement Name) !! (N.N (J,J course-group1) (K,K course-group2) (L,L course-group3) ... etc. )}
\]

Translated literally, this statement tells Academic Audit, “student must take N,N credits from the following groups of courses, with no more than J,J credits from course-group1, K,K credits from course-group2, L,L credits from course-group3, etc.” It is very similar to a normal credits group operator, except it provide greater flexibility in how the credits will be distributed among the groups.

Examples:

\[
(8.0 \text{ Fine Arts}) !! (8 (3.0 \text{ ART}^*) (3.0 \text{ MUS}^*) (1.0 \text{ THE}^*) (2.0 \text{ ENG}^*) (2.0 \text{ LIT}^*) (2.0 \text{ SPE}^*) )
\]

; Choose 8 courses from the above groups, no more than 3 Art,
; 3 Music, 1 Theatre, 2 English, 2 Lit, or 2 Speech.

In the above example each course pattern is considered a group of courses. If ENG-* and LIT-* are to be considered one group together, they may be grouped within parentheses as in the example below:

\[
(8cr \text{ Fine Arts}) !! (8.0 (3.0 \text{ ART}^*) (3.0 \text{ MUS}^*) (1.0 \text{ THE}^*) (2.0 \text{ ENG}^* \text{ LIT}^*)(2.0 \text{ SPE}^*) )
\]

; Choose 8 credits from the above groups, no more than 3 Art,
; 3 Music, 1 Theatre, 2 English or Lit, or 2 Speech.

- Note: Numbers used in a double credit group operator must be floating-point numbers. This tells Academic Audit that the operator is referring to a required number of credits rather than a required number of courses.
- Although operators can generally be used in combination with each other, DOUBLE GROUP operators may not have AND's, OR's, or other GROUP operators under them, i.e. inside the parentheses for the GROUP.
CHAPTER 10: WRITING MAJOR FILES

Requirement Statements

CHECK FOR CONDITION Operator

The CHECK operator is used to limit the choices for a requirement on a per-student basis. Starting with the CHECK and ending with the next logical parenthetical closing, the requirement choices are only enabled if the CHECK statement was true for that student. Choices may be limited based either on a value in one of the user-defined fields or on the existence of a course on the student's transcript.

This operator may be used to set a track of courses that should be followed or to enhance the performance (speed) of the matching algorithm.

Format:

CHECK course description :
CHECK [[ uservar-operator-value]]:

Example 1:

{Science Track} |
CHECK BIO-101: & (BIO-101 BIO-102)
CHECK BIO-201: & (BIO-201 BIO-202)
CHECK CHM-101: & (CHM-101 CHM-102)
CHECK PHY-101: & (PHY-101 PHY-102)
CHECK *-*: ! (2 Z EAS-*))

In example 1, the match will be faster because only those tracks where the beginning course has been taken will be explored, eliminating unnecessary options and thus increasing response time.

EXAMPLE 2:

{Freshman Math} |
CHECK [[MATHSCORE=1]]: & (MAT-098 MAT-099 MAT-100)
CHECK [[MATHSCORE=2]]: & (MAT-099 MAT-100)
CHECK [[MATHSCORE >2]]: | (MAT-100 [[MATHSCORE >3]]))

In example 2, the requirement for remedial courses was determined by the placement test scores. [[MATHSCORE]] is a user-defined (custom) field.

- **Note:** When using CHECK statements, make sure that at least one of the CHECK's in a requirement is true for every student. Otherwise, the match may not know how many unfilled slots to show on the printed audit report.

PARALLEL OR Operator || ( )

This operator is similar to the simple OR since it requires that only one of the choices under it be completed in order to satisfy the requirement. It is different because it is used when the expressions under it are complex. It
CHAPTER 10: WRITING MAJOR FILES

Using Modifiers

is a signal to the matching algorithm to solve the expressions “in parallel” because even though the choices are complex, they are all equal in terms of courses or credits.

Example 1:

{Science Core} ||{| (3 3 BIO-*), (3 3 CHM-*), (3 3 PHY-*), (3 3 PHS-* EAS-*), (3 3 GEO-* MFT-*))

Example 2 (This example shows course-level modifiers which are described in the next section.):

{2 SCIENCES-1 SC 1 LAB} ||{| CHECK AST-11???: & (AST-11???, AST-11???, P, 1.0) CHECK PHY-1???: & (PHY-1???, PHY-1???, P, 1.0) CHECK PHY-25???: & (PHY-25???, PHY-25???, P, 1.0) CHECK CHM-1???: & (CHM-1???, CHM-11???, P, 1.0) CHECK BIO-1???: & (BIO-1???, BIO-1???, P, 1.0) CHECK BIO-2???: & (BIO-2???, BIO-2???, P, 1.0) CHECK BIO-3???: & (BIO-3???, BIO-3???, 2.0) CHECK HON-2???: & (HON-2?01, HON-2?02, P, 1.0)

• Note: The PARALLEL OR Operator can only be used if every choice under it is satisfied with the same number of courses or the same number of credits.

Using Modifiers

OVERVIEW

Modifiers, as their name implies, are used to modify requirements. Generally, they are used to add additional criteria that must be met in order to satisfy a requirement. There are two basic types of modifiers; requirement-level modifiers and course-level modifiers.

Requirement-Level Modifiers

Requirement-level modifiers are used to modify an entire requirement. In most cases, a requirement-level modifier is used to define additional criteria a course must pass before it can be used to satisfy that particular requirement. Requirement-level modifiers include checks for course residency, passing (but not letter grade) status, minimum grade checks, minimum credit checks, and the indicator that courses matched to a certain requirement belong to one of the three custom QPA groups.

A requirement-level modifier is coded immediately following the {Requirement Name} and must be enclosed in square brackets [ ]. Requirement-level modifiers refer to ALL the course options described in the requirement. These modifiers may be combined by coding multiple modifiers within one set of square brackets [ ]
and separating them with commas.

Format: \( [A \text{ Requirement}][\text{Modifier1, Modifier2, etc.}] \)

### Resident Course Modifiers

**Syntax:** \( [\text{R}] \)

**Indication:** Any course that is used to satisfy this requirement must be a resident course.

**Example:**

\( [\text{Urban Geography}][\text{R}] \quad | \quad (\text{GEO-210 GEO-214}) \)

### Passing Grade Modifiers

**Syntax:** \( [\text{P}] \)

**Indication:** Allows courses taken as pass/satisfactory/credit which do not normally add into the QPA or match requirements in this specific case. A “passing” grade code status is defined in the grades table dialog box (see *The Grades Table*, page 6-3.) The modifier “P” is used regardless of the particular grade code (P,S,CR) you use. Courses with valid letter grades will also match.

**Example:**

\( [\text{Free Electives}][\text{P}] \quad | \quad (18.0 \quad 18.0 \quad *-* \quad ) \)

### Minimum Grade Modifiers

**Syntax:** \( [\text{MG=X}] \)

**Indication:** Any course used to satisfy this requirement must have at least a grade of X. Grade hierarchy is determined in the system grade table (Sec *The Grades Table*, page 6-3.) Minus grades must be coded in quotes. \([\text{MG=\text{"C-"}}]\)

**Example:**

\( [\text{German Electives}][\text{MG=C}] \quad \& \quad (\text{GER-1111} \quad | \quad (\text{GER-1112 GER-1713})) \)
CHAPTER 10: WRITING MAJOR FILES

Using Modifiers

Minimum Credit Modifiers

Syntax: \[MC=\{Y.Y\}\]

Indication: Any course used to satisfy this requirement must have at least Y.Y credits. (Note: In the IRONSOFT.INI file, it is possible to specify globally (for all requirements and all majors) the minimum number of credits a course must have in order for it to be matched to a requirement. This modifier, when present, overrides that value for the requirement it is modifying.)

Example:

\{(Free Electives)[MC=3.0] \| (18.0 \ 18.0 \ -(** MAT-\*)\)\}
; Any 3 credit or better course, except math courses, may be used
; as free electives.

Custom QPA Modifiers

Syntax: \[QPA_\{n\}\] where \(n = 1, 2 \text{ or } 3\).

Indication: Any course used to satisfy this requirement will also be used in the calculation of QPA\(n\).

Example:

\{(German Electives)[QPA2] \& (GER-1111 | (GER-1112 GER-1713))\}

Course-Level Modifiers

Course-level modifiers are used to define additional criteria a course must pass before it can be used to satisfy a certain requirement in the audit. Available modifiers include checks for course residency, passing grade (but not letter grade) status, minimum grade checks, minimum credit checks, and year and semester taken.

Course-level modifiers are coded immediately after a course pattern, and apply to that course pattern only. They are separated from the course description by a forward slash (/) for the Year and Semester modifier, and by a comma (,) for others. When used, the "YRSM" modifier must be the first modifier after the course description. Multiple modifiers may be placed on a single course description by separating them with commas, as in the following example:
CHAPTER 10: WRITING MAJOR FILES

Using Modifiers

{German Electives} & (GER-1111 | (GER-1112 GER-1713/95SP,R))
; If GER-1713 is used for this requirement, it must be taken in the
; Spring 95 semester or later, and it must have been taken
; in residence.

### Year-Semester Modifiers

**Syntax:**

/YRSM or /<YRSM or /YRSM

**Indication:**

Specifies a particular year (YR) and semester (SM) which the course must be taken before, during or after in order for it to match the requirement.

**Examples:**

/YRSM (Course must be taken during YRSM.)

{German Electives} & (GER-1111 | (GER-1112 GER-1713/95SP))
; If GER-1713 is used for this requirement, it must be taken in the
; Spring 95 semester.

/>YRSM

{German Electives} & (GER-1111 | (GER-1112 GER-1713/95SP))
; If GER-1713 is used for this requirement, it must be taken in the
; Spring 95 semester or later.

/<YRSM

{German Electives} & (GER-1111 | (GER-1112 GER-1713/<95SP))
; If GER-1713 is used for this requirement, it must be taken in the
; Spring 95 semester or before.

### Minimum Credits Modifier

**Syntax:**

,f.f

**Indication:**

When this course-level modifier is used, the course it refers to must have at least f.f credits to satisfy the requirement. (Note: In the IRONSOFT.INI file, it is possible to specify globally (for all requirements and all majors) the minimum number of credits a course must have in order for it to be matched to a requirement. This modifier, when present, overrides that value for the course it is modifying. Therefore, it can be used to allow a lesser-credit course to be matched, or increase the minimum number of credits required for a course to be matched.)
CHAPTER 10: WRITING MAJOR FILES

Control Statements

Example:

{Computer Lab} CSC-110,1.0
; Normally a course must have 2 credits in order to be matched to a
; requirement, in this case a 1.0 credit course is allowed.

Passing Grade Modifiers ,P

Syntax: , P

Indication: Allows courses taken as pass/satisfactory/credit which do not normally add into the QPA or match requirements to match in this specific case. A "passing" grade code status is defined in the grades table (see The Grades Table, page 6-3.) The modifier "P" is used regardless of the particular grade code (P,S,CR) you use. Courses with valid letter grades will also match.

Example: (Humanities Elective) HUM-*,P

Minimum Grade Modifiers ,MG=X

Syntax: , MG = X

Indication: This course modifier requires that if the course it modifies is used to satisfy the requirement, it must have a minimum grade of x. Grades are determined to fall below or above minimum grade based on the hierarchy established in the grade table (see The Grades Table, page 6-3.) Minus grades must be coded in quotes. (MG="C-"

Example: (Computer Literacy) CSC-101,MG=C

Resident Course Modifiers ,R

Syntax: , R

Indication: This course modifier requires that the course it modifies be taken in residence.

Example: {Senior Seminar) HIS-499,R

Control Statements

Control statements, as their name implies, help Academic Audit control certain parts of the audit process. They are used to control how certain items print out on an audit report. They are also used to define certain non-course requirements, such as minimum QPA's.
CHAPTER 10: WRITING MAJOR FILES

Control Statements

#TITLE

The #TITLE statement is the first statement in a major file. It tells Academic Audit what to display in the Title line of a printed report. Here is an example of a #TITLE statement:

#TITLE "BA in English, 1995 Catalog Year"

Notice that the text which follows the #TITLE keyword must be contained in quotation marks. The above example would produce the following output on an audit report:

<table>
<thead>
<tr>
<th>SOFT University</th>
<th>Run Date:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progress Evaluation</td>
<td>06/28/96</td>
</tr>
<tr>
<td>For ENGLISH\JOURNALISM 1993 CATALOG YEAR</td>
<td></td>
</tr>
<tr>
<td>1. ENG COMP I : ENG-101 93FA A 3.00</td>
<td></td>
</tr>
<tr>
<td>2. ENG COMP II : ENG-102 93SP D 3.00</td>
<td></td>
</tr>
<tr>
<td>3. AMERICAN HISTORY : HIS-101 93SP A 3.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HIS-102 94SP B 3.00</td>
</tr>
<tr>
<td>4. NATURAL SCIENCE : * 2 unfilled courses *</td>
<td></td>
</tr>
<tr>
<td>5. 4 FINE ARTS : ART-497 93FA B 3.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENG-106 93FA B 3.00</td>
</tr>
<tr>
<td></td>
<td>HON-295 93SP C 3.00</td>
</tr>
<tr>
<td></td>
<td>2100 94SP C 3.00</td>
</tr>
</tbody>
</table>

#PRINT

The #PRINT statement functions similarly to the #TITLE statement. It is used to print lines of text within the body of the audit report to add clarity to the report. The location of the text line on the report is determined by the location of the #PRINT statement within the major file. Consider the following code sample which contains a #PRINT statement:

{(FINE ARTS) ! (4 1 ART-* MUS-* THE-* ENG-* LIT-* SPE-* COM-* PHI-* FRE-* GER-* RUS-* SPN-* 10-10 HON-281 HON-295 HON-388) #PRINT "Free Electives: 10.0 credits required" (ELECTIVES) ! (18.0 18.0 *-*)

Notice that the text which follows the #PRINT keyword must be contained in quotation marks. For this exam-


### CHAPTER 10: WRITING MAJOR FILES

**Control Statements**

Example, the printed audit might look something like this:

<table>
<thead>
<tr>
<th>4. NATURAL SCIENCE</th>
<th>HIS-102</th>
<th>94SP</th>
<th>B</th>
<th>3.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. FINE ARTS</td>
<td>ART-497</td>
<td>93FA</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>ENG-106</td>
<td>95FA</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>HON-295</td>
<td>95SP</td>
<td>C</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>FIN-100</td>
<td>94SP</td>
<td>C</td>
<td>3.00</td>
</tr>
<tr>
<td>Free Electives: 18.0 credits required</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. ELECTIVES</td>
<td>CSC-105</td>
<td>94FA</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>EDF-100</td>
<td>94FA</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>ENO-107</td>
<td>93FA</td>
<td>A</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>ESY-205</td>
<td>94SP</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>ESY-320</td>
<td>95SP</td>
<td>B</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>SOM-270</td>
<td>93FA</td>
<td>A</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>ENG-151</td>
<td>95FA</td>
<td>A</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Text from #PRINT statement appears above the same requirement the #PRINT statement is above in the major file.

Notice that in the code sample, the #PRINT statement is located directly above the "fine arts" requirement statement, and on the report, the text produced by the #PRINT statement appears directly above the audit results for the "fine arts" requirement.

### #REQUIREDCREDITS

Most schools set a minimum number of credits which a student must attain in order to graduate. Generally, this minimum number is specified globally (for all majors) in the IRONSOFT.INI file. (See SYSTEM FILES in Chapter 6, Configuration.) In cases where a particular major requires that a different number of credits than the global value be attained, a #REQUIREDCREDITS statement can be inserted into the major file to override the value specified in the IRONSOFT.INI file. The following is an example of a #REQUIREDCREDITS statement:

```
#REQUIREDCREDITS 128.0
```

Note: The value specified for a #REQUIREDCREDITS statement must be a floating-point number with at least one decimal place.

### #REQUIREDCOURSES

As with credits, most schools set a minimum number of courses which a student must take in order to graduate. Generally, this minimum number is specified globally (for all majors) in the IRONSOFT.INI file. (See SYSTEM FILES in Chapter 6, Configuration.) In cases where a particular major requires that a different number of courses than the global value be taken, a #REQUIREDCOURSES statement can be inserted into the major file to override the value specified in the IRONSOFT.INI file. The following is an example of a #REQUIREDCOURSES statement:

```
#REQUIREDCOURSES 40
```

Notice that the value specified for a #REQUIREDCOURSES statement must be an integer.
CHAPTER 10: WRITING MAJOR FILES

Control Statements

#REQUIREDOVERALLQPA

Most schools set a minimum QPA which must be attained in order for a student to graduate. Generally, this minimum number is specified globally (for all majors) in the IRONSOFT.INI file. (See SYSTEM FILES in Chapter 6, Configuration.) In cases where a particular major requires that a different minimum QPA be attained, a #REQUIREDOVERALLQPA statement can be inserted into the major file to override the value specified in the IRONSOFT.INI file. The following is an example of a #REQUIREDOVERALLQPA statement:

#REQUIREDOVERALLQPA 2.20

- Note: The value specified for a #REQUIREDOVERALLQPA statement must be a floating-point number with at least one decimal place.

#REQUIREDMATCHEDQPA

The #REQUIREDMATCHEDQPA statement is similar to the #REQUIREDOVERALLQPA, except that it sets a minimum matched QPA which must be attained in order to graduate. The matched QPA is calculated using only courses that are actually matched with requirements during the audit process.

An example of a #REQUIREDMATCHEDQPA statement:

#REQUIREDMATCHEDQPA 2.30

- Note: As with the #REQUIREDOVERALLQPA statement, the value specified must be a floating-point number with at least one decimal place.
- It should also be noted that there is no provision for globally setting the minimum matched QPA in the IRONSOFT.INI file. It can only be set with a #REQUIREDMATCHEDQPA statement.

#ALSOCHECK

As a general rule, once Academic Audit matches a particular course to a major requirement, that course may not be used again to satisfy any other requirements. In cases where courses should be allowed to be used again, an #ALSOCHECK statement is used. This statement is used to force a check of a secondary set of requirements after the current set is checked. During an #ALSOCHECK, all courses are free to be re-used.

For example, suppose that in order to graduate at a particular school, 42 of a student’s total credits must be from upper division courses (300 or 400 level.) In order so that courses already used to satisfy other requirements may also be used to satisfy this requirement, the following #ALSOCHECK statement is placed in the major file:

#ALSOCHECK "UPPERDIV.CHK"

Then, a separate file called UPPERDIV.CHK is created and stored in the same directory as the other major
CHAPTER 10: WRITING MAJOR FILES

Control Statements

files. This file would contain a requirement statement requiring a student to take 42 credits of 300 or 400 level courses:

```
#TITLE "Check for 42.0 upper division credits"
(Check) !(42.0 42.0 *-3?? *-4??)
```

Some things to remember when using #ALSOCHECK statements:

- On the printed audit report, the results of the #ALSOCHECK appear immediately below the primary requirements.
- There may be up to five #ALSOCHECK statements in a single file, but they may not be nested. (i.e., the file which an #ALSOCHECK statement refers to may not also contain #ALSOCHECK statements.)
- The file name used in an #ALSOCHECK statement must be in quotation marks, and the full file name (including the DOS extension) must be used.

#SILENTCHECK

The #SILENTCHECK statement uses the same syntax as, and works identically to an #ALSOCHECK statement, except detailed results of the check are not included on the report. Only the title of the checked file (from the file's #TITLE statement) and a single line indicating whether or not the check was satisfied are printed.

```
SMO-100  92PA A  3.00
PSY-205  94SP B  3.00
PSY-320  95SP B  3.00
SOM-270  93PA A  3.00
```

For SECONDARY CHECK FOR WRITING REQUIREMENT
*** Not Satisfied ***

Printed results of a #SILENTCHECK statement.

#INCLUDE

An #INCLUDE statement, when encountered in a major file, tells Academic Audit to include the contents of another major file in the file in which the #INCLUDE statement appears. For example, if you have a major file called MAJOR.MAJ which contains the statement:

```
#INCLUDE "CORE95.MAJ"
```

Academic Audit will act as if all of the statements contained in CORE95.MAJ are also contained in MAJOR.MAJ (in addition to the other statements in MAJOR.MAJ.)

The #INCLUDE statement is used primarily when there are blocks of requirements common to many or all majors (such as core requirements.) #INCLUDE statements should follow these rules:

- #INCLUDE statements can be placed anywhere in a major file, except before other control statements.
- There is no limit to the number of #INCLUDE statements that can be used in a major file, and they can be
CHAPTER 10: WRITING MAJOR FILES

Control Statements

nested up to five levels.

- The file name used in an #INCLUDE statement must be in quotation marks, and the full file name (including the DOS extension) must be used.
- It is good practice to give "included" files a different DOS extension than .MAJ so that they do not show up on the list of major files available to the user in the software.

■ Working with Custom QPA's

Academic Audit's requirement language provides a special set of control statements for working with custom QPA's. Custom QPA's are QPA's defined by the user. One possible example would be the QPA for all of the core requirements. Another would be the QPA for the major requirements. Academic Audit allows for up to three user-defined custom QPA's, referred to in the requirements language as QPA1, QPA2 and QPA3. The instructions for defining these QPA's appear in the section on Custom QPA Modifiers on page 10-16. Here we'll discuss the control statements used to set the minimum value a student must achieve for these QPA's and control how they appear on the printed audit report.

#REQUIREDQPA1, #REQUIREDQPA2, #REQUIREDQPA3

The #REQUIREDQPA_n (where n = 1, 2 or 3) statement functions similarly to the #REQUIREDOVERALLQPA and #REQUIREDMATCHEDQPA statements. It is used to set a minimum QPA a student must achieve for QPA1, QPA2 or QPA3. For example, if QPA1 is defined as the QPA for core requirements, then the statement,

    #REQUIREDQPA1 2.40

tells Academic Audit that a student must achieve a QPA of at least 2.40 for the core requirements.

- Note: The value specified for a #REQUIREDQPA_n statement must be a floating-point number with at least one decimal place.
- Major QPA's may also be defined and printed using MAJORQPA control statements.

#QPA1LABEL, #QPA2LABEL, #QPA3LABEL

The #QPA_nLABEL (where n = 1, 2 or 3) statement tells Academic Audit what label to print next to QPA_n on the audit report. These labels can also be specified in the IRONSOFT.INI file. If a #QPA_nLABEL statement appears in a major file, it will override the label specified in IRONSOFT.INI. If this label is not defined in either the major file or the IRONSOFT.INI file, no value will be printed for QPA_n.

Following is an example of a #QPA_nLABEL statement:

    #QPA3LABEL "Concentration"

- Note: The label specified with a #QPA_nLABEL statement should be in quotation marks.
- Major QPA's may also be defined and printed using MAJORQPA control statements.
Using the MAJORQPA Feature

In addition to the control statements used to define and label custom QPA's, Academic Audit provides a set of control statements for defining and computing what is referred to as a MAJORQPA. Unlike with custom QPA statements, a MAJORQPA is computed from all the courses taken that match a list of departments and/or specific courses, regardless of whether or not they have been matched to requirements. This computation can also include all failed or repeated courses or can target only the “completed” courses.

Here are four types of control statements which can be used to describe how to compute, print and check minimums for a MAJORQPA:

1. \#MQBEGIN
   \#MAJORQPADEPARTMENTS
   \#MQEND

2. \#MAJORQPA COURSES

3. \#MAJORQPALABEL

4. \#MAJORQPA

If a MAJORQPA is computed, it can be queried in reporting, and it will be printed on the audit with the other QPA’s.

#MAJORQPADEPARTMENTS

The \#MAJORQPADEPARTMENTS statement is the main part of a set of three control statements used to tell Academic Audit what courses should be counted toward calculation of the MAJORQPA. The format is as follows:

**Format:**
\#MQBEGIN
\#MAJORQPADEPARTMENTS (list of courses, course patterns or departments)
\#MQEND

**Example:**
\#MQBEGIN
\#MAJORQPADEPARTMENTS (ACCT-3163 ACCT-3153 ACCT-3253 ACCT-3303 ACCT-4?? - (FINA-* FINA-1001) BUS-* )
\#MQEND

In the above example, courses listed inside the parentheses, or which fall within the course patterns listed, would be used in the calculation of the student’s MAJORQPA.

- **Note:** All of the syntax which is used in the requirements language to describe course patterns, including the NOT operator (~), can be used to describe this list of courses. The other (logical) operators are not functional within this control statement.
- The \#MQBEGIN and \#MQEND statements are required as shown. This set of control statements trigger the computation and printing of the MAJORQPA.
CHAPTER 10: WRITING MAJOR FILES

Control Statements

#MAJORQPA COURSES

Earlier it was mentioned that Academic Audit can be configured so that a MAJORQPA can be calculated by including or excluding a student's repeated, failed or attempted courses. This can be configured globally (for all majors) within IRONSOFT.INI, or for individual majors using the #MAJORQPA COURSES statement. To configure this parameter within a major file, use the #MAJORQPA COURSES statement in one of the following two ways:

Format 1:  #MAJORQPA COURSES ALL
This format tells Academic Audit to include repeated, failed or attempted courses in the calculation of the MAJORQPA.

Format 2:  #MAJORQPA COURSES COMPLETED
This format tells Academic Audit to exclude repeated, failed or attempted courses from the calculation of the MAJORQPA.

• Note: If a #MAJORQPA COURSES statement is not used in a major file which includes a #MAJORQPA DEPARTMENTS statement, ALL is the default value, which means that all of the courses on the transcript which match the specified patterns will be used in the computation.

#MAJORQPA LABEL

The #MAJORQPA LABEL statement is used to indicate the label which should print with the MAJORQPA on the audit. It overrides the value specified in IRONSOFT.INI. The default Label is “Major QPA.”

Format:  #MAJORQPA LABEL "XXXXXXXX"
Example:  #MAJORQPA LABEL "History Major"

#MAJORQPA

The #MAJORQPA statement sets the minimum value a student must attain for MAJORQPA in order to graduate. It overrides the minimum value, if any, specified in IRONSOFT.INI. The results of this comparison are printed in the computed fields window of the audit report, with the minimum value on this statement printed as the “required” value.

Format:  #MAJORQPA $ff
Example:  #MAJORQPA 2.75

• Note: The value specified for a #MAJORQPA statement must be a floating-point number with at least one decimal place.
CHAPTER 10: WRITING MAJOR FILES

Control Statements

■ Checking Residency

UNDERSTANDING THE RESIDENCY FEATURE
Academic Audit provides two basic methods for auditing residency requirements. Residency requirements can be set globally (for all students and all majors) in the I Ronsoft.INI file, or they can be set for individual majors in major files. In cases where both methods are used, the requirements set in major files will always override those set in I Ronsoft.INI. The section below explains the rules for setting residency requirements within major files.

Residency requirements are audited by checking a student's transcript for the following conditions:

1. A specific list of courses must be taken in residence.
2. The last xx of yy credits must be taken in residence.
3. The last xx or yy credits from matched courses (courses applied to a degree requirement) must be taken in residence.
4. Any combination of 1, 2 and 3.

If multiple residency control statements are used, all of the statements must be true for the student to pass residency. Any one failure turns the “passes residency check” indicator to “N.” This indicator prints (if printing has not been suppressed) in a box directly below the summary window and is displayed in the COMPUTED FIELDS window.

Residency checks affect the audit results and will be used to determine if a student passes or fails their complete audit.

#RESIDENTCHECK

The #RESIDENTCHECK statement tells Academic Audit to check another major file—referred to as a residency file—which has been coded to check for the specific list(s) of courses which must be taken in residence. This control statement overrides the residency file specified in I Ronsoft.INI. Below is an example of a #RESIDENTCHECK statement:

#RESIDENTCHECK *residency.chk"

For this example, the residency file RESIDENCY.CHK would have to be present in the directory where major files are stored. The contents of RESIDENCY.CHK might look something like this:

#TITLE "Residency check"
{Checking residency}[R] !(15.0 15.0 ENGL-1200 ENGL-1803 ENGL-2803 ECON-2103 ECON-2113 RELI-* HIST-1603 HIST-1613 SOWO-3863 THEA-* MUSI-* RTV-*)
CHAPTER 10: WRITING MAJOR FILES

Control Statements

• Note: The [R] modifier is required on all requirement statements within a residency file to ensure residency.

• #RESIDENTCHECK statements work exactly like #ALSOCHECK statements except that the requirement slots do not print anywhere on the audit. A #RESIDENTCHECK statement simply sets the "passes residency check" indicator to "N" if it fails.

#LASTCREDITS

The #LASTCREDITS control statement is used to scan all of the completed courses to determine if the correct number of "recent" credits have been taken in residence. The courses are first sorted by YEAR/SEMESTER (most recent to oldest) and TYPE (all resident courses first, then all transfer courses) and then processed as indicated below. Format 2 below allows for an "OR" condition on this statement.

Format 1:        #LASTCREDITS XX.X/YY.Y
Example:         #LASTCREDITS 30.0/30.0

In the above example, 30 of the last 30 credits must be taken in residence. This statement also requires that the final 30 credits taken before graduation be taken in residence. In the example below,

#LASTCREDITS 45.0/60.0

45 of the last 60 credits taken must be resident.

Format 2:        #LASTCREDITS XX.X/YY.Y OR QQ.Q/ZZ.Z
Example:         #LASTCREDITS 45.0/90.0 OR 60.0/125.0

In the above example, either 45 of the last 90 or 60 of the last 125 credits must be taken in residence.

• Note: Floating-point numbers must be used with a #LASTCREDITS statement.

#LASTCREDITSUSED

The #LASTCREDITSUSED statement works identically to the #LASTCREDITS statement, except that it deals only with credits that were "used in the match," i.e., applied toward a degree requirement. It causes Academic Audit to scan all of the courses which were completed and used in the match to determine if the correct number of "recent" credits have been taken in residence. The courses are first sorted by YEAR/SEMESTER (most recent to oldest) and TYPE (all resident courses first, then all transfer courses) and then processed as indicated below. Format 2 below allows for an "OR" condition on this statement.

Format 1:        #LASTCREDITS XX.X/YY.Y
Example:         #LASTCREDITS 30.0/30.0

In the above example, 30 of the last 30 credits used in the match must be taken in residence. This statement also requires that the final 30 credits taken before graduation that were applied to degree requirements be taken in residence. In the example below,
CHAPTER 10: WRITING MAJOR FILES

Preprocessor Statements

#LASTCREDITS 45.0/60.0

45 of the last 60 credits used in the match must be resident.

Format 2: #LASTCREDITS XX.X/YY.Y OR QQ.Q/ZZ.Z
Example: #LASTCREDITS 45.0/90.0 OR 60.0/125.0

In the above example, either 45 of the last 90 or 60 of the last 125 matched credits must be taken in residence.

• Note: Floating-point numbers must be used with a #LASTCREDITSUSED statement.

#PRINTRESIDENT

The #PRINTRESIDENT statement is used to suppress the printing of the Residency Indicator Box on the audit. Since residency usually does not become an issue until later in the student’s career, it is possible to suppress printing for students who have not attained a minimum number of completed credits.

Format: #PRINTRESIDENT XX.X
Example: #PRINTRESIDENT 90.0

In the above example, residency indicators would be printed for any students who had completed at least 90.0 credits.

Preprocessor Statements

Preprocessor statements are a very powerful addition to the Academic Audit requirements language. They are called preprocessor statements because the actions they invoke take place before the major file is processed, i.e., before the audit is run.

Preprocessor statements are similar to control statements in syntax. Like control statements, a preprocessor statement begins with the # symbol. Users familiar with the C programming language will recognize that Academic Audit preprocessor statements are similar in appearance and function to those used in C.

#DEFINE

A #DEFINE statement is used to define a macro. Generally speaking, a macro is a string of alpha characters used to represent a longer string of characters. Consider the following example:

#DEFINE WritingCourses ENG-121 ENG-127 ENG-144 LIT-141

When Academic Audit encounters the above statement in a major file, it will scan the rest of the file for the string “WritingCourses.” Wherever it encounters the string “WritingCourses,” it will replace it with the string “ENG-121 ENG-127 ENG-144 ENG-203 LIT-141.” (Note: The replacement occurs in memory only, not in the actual disk file.)
CHAPTER 10: WRITING MAJOR FILES

Preprocessor Statements

If the above macro has been defined, it might then be used in a requirement statement as in this example:

; Two writing courses required
(Writing Requirement) !(2 2 WritingCourses)

Academic Audit will process this statement as if it actually reads:

; Two writing courses required
(Writing Requirement) !(2 2 ENG-121 ENG-127 ENG-144 LIT-141)

The process of replacing a macro with its literal value is called macro expansion. In other words, when Academic Audit encounters the macro “WritingCourses,” it expands it into “ENG-121 ENG-127 ENG-144 LIT-141.”

Multi-line Macros
When Academic Audit parses a define statement, the first string of characters after #DEFINE becomes the macro. Everything following the macro, to the end of the line, is what the macro is expanded to. This may not always be practical if the literal value of a macro is exceptionally long. To break the literal value across two or more lines, a backslash (\) is used as a continuation character, as in this example:

#DEFINE PaintingCourses ART-145 ART-146 ART-147 ART-165 ART-202\ ART-203 ART-313 ART-344

Care should be taken that the "\" character is the last character on the line. There should be no trailing spaces after it.

Null-value Macros
In some cases, which will be explained later, it is desirable to define a macro as null, i.e., an empty character string. This is done simply by not including a literal value in a define statement, as shown here:

#DEFINE GEN_ED_REQUIREMENTS

Recursive Macro Expansion
Macros can be defined in terms of other macros. For instance, consider these two #DEFINE statements:

#DEFINE WritingCourses ENG-121 ENG-127 ENG-144 LIT-141
#DEFINE EnglishElectives ~(ENG-* ENG-267 WritingCourses)

Notice that the literal value of the EnglishElectives macro contains the WritingCourses macro. When Academic Audit encounters “EnglishElectives” in a major file after it has been defined as a macro, it will expand it to:

~(ENG-* ENG-267 ENG-121 ENG-127 ENG-144 LIT-141)

This is called recursive macro expansion. What it means is that when Academic Audit expands a macro to its literal value, it then scans the literal value for the presence of other macros, which it in turn expands. It conin-
CHAPTER 10: WRITING MAJOR FILES

processor Statements

; to scan each expanded macro until it finds no more macros.

rules for Macros

The following rules should be kept in mind when using macros:

Macros may consist only of alphabetic characters and the underscore ( _ ) character. The literal values they’re defined as, however, may contain any printable characters as well as spaces.
Macros are case-sensitive. WRITINGCOURSES is not the same as WritingCourses or writingcourses.

\F / #ELSE / #ENDIF

An #IF statement is always used in combination with an #ENDIF statement, and sometimes in combination with an #ELSE statement.

Together, an #IF statement, an #ENDIF statement, and all of the statements in between them form what is called an #IF block.

The #IF Statement Format

An #IF statement has the format:

\#IF <expression>

Here <expression> is a coded expression that is either true or false. For example, in this #IF statement:

\#IF SATMATH < 400

The expression “SATMATH < 400” is true if the contents of the student’s SATMATH database field (assuming custom field called SATMATH has been defined) is less than or equal to 400. (Remember, in Academic Audit, both the > and < operators imply “equal to” as well as “less than” or “greater than.”) If the student’s SATMATH score is greater than 400, the expression is false.

The expression used in an #IF statement evaluates as true, then all of the statements following that if statement, until its corresponding #ENDIF statement, will be included in the major file prior to processing. If the expression evaluates false, then those statements are not included. For example:

\#IF SATMATH < 400
\#PRINT "Remedial Math Required"
{Remedial Math} MATH-098
\#ENDIF

Before processing a major file containing the above block of code, Academic Audit will check the value of the student’s SATMATH field. If it is less than or equal to 400, then the two lines between the #IF and the #ENDIF statements are included in the file during processing. Otherwise, the lines will not be included.
CHAPTER 10: WRITING MAJOR FILES

Preprocessor Statements

Nesting #IF Blocks

#IF blocks can be nested, that is, one #IF block can be contained within another. That is why it is important to realize that the next #ENDIF statement that appears after an #IF statement is not necessarily the one that corresponds with that #IF statement. (Note: #IF blocks may be nested a maximum of 24 levels deep.)

Testing Data Values with #IF Statements

One common usage for #IF statements is to check the value of a particular field in a student's database record. This is illustrated above, where a student's SATMATH score is used in a logical expression with a less-than operator (<). Database fields which contain character-string data may also be used in logical expressions within an if statement. The only difference is that the literal value used for comparison must be contained within quotation marks. For example:

```c
#define major "COMPSCI"
```

The comparison operators that can be used within an #IF statement are = (equals), > (greater than or equal to), < (less than or equal to) and != (not equal to).

Using Functions With #IF Statements

In addition to checking the value of database fields, an #IF statement may be used in conjunction with one of the following four functions: HASCOURSE, VALIDCOURSE, DEFINED and RULESET. They are described below.

HASCOURSE — The HASCOURSE function has the following format:

```c
HASCOURSE("CoursePattern1", "CoursePattern2", ... etc.)
```

Where "CoursePatternN" is a course pattern in quotation marks, for example, "MAT-240". Academic Audit checks a student's transcript for courses that match one or more of the course patterns in the HASCOURSE function. If any matching course is found, the function is considered a true expression. Consider this statement:

```c
#define HASCOURSE("MAT-240", "MAT-255", "MAT->300")
```

If, for example, the student's transcript contains MAT-255, but not MAT-240 or any 300-level or better Math course, the HASCOURSE function will still evaluate as true.

VALIDCOURSE — The VALIDCOURSE function works the same as HASCOURSE, except a student must have a valid course that matches one of the listed course patterns. A valid course is any course that can be matched to a requirement, e.g. not failed, not withdrawn, etc.

Note to TransferPro users: HasCourse operates on the raw (pre-articulated) transcript courses, while ValidCourse operates on the articulated courses. One common technique in TransferPro .ART files is to use the preprocessor to conditionally include equivalencies into MASTER.ART:

```c
#define HasCourse("3365:***")
#define INCLUDE "3365.ART"
#endif
```

DEFINED — The DEFINED function determines whether one or more macros in a list have been defined. It
CHAPTER 10: WRITING MAJOR FILES

reprocessor Statements

as the following format:

```
DEFINED(Macro1, Macro2, Macro3 ... etc.)
```

If any of the listed macros have been defined, the expression evaluates as true. Considering the following block of code:

```
#DEFINE LABCHECK ;LABCHECK is defined as an empty string
#IF DEFINED (LABCHECK)
#PRINT "LABCHECK is a defined macro"
#ENDIF
```

The above code block, as written, will cause “LABCHECK is a defined macro” to be printed on the report. Since LABCHECK is indeed a defined macro, the #IF statement is true. If the #DEFINE LABCHECK statement is taken out, the #IF statement will be false, and the #PRINT statement will not be included for processing.

RULESET — The RULESET function has the following format:

```
RULESET("Ruleset1", "Ruleset2", ... etc.)
```

The RULESET function compares the name of the rule set being used for the audit against the list of rule set names supplied to the function. The rule set name is the name of the major file (minus the .MAJ extension) which appears in the listbox in the student window, which is selected for auditing. If this rule set name matches any of the ones listed in the RULESET parameter list, the expression is true. The * and ? wildcards may be used in the listed rule set names. (See Understanding Wildcards, page 4-8, Using Wildcards in Course Patterns, page 10-6.)

Negating Functions with “!”
The “!” character, which is used as the group operator in the Academic Audit requirements language, also acts as a negation operator in #IF statements. When placed before a function, it causes it to evaluate as true if and only if it would normally evaluate as false. For example,

```
#IF !HASCOURSE("MAT-240")
```

The above #IF statement will evaluate as true if and only if MAT-240 is not present on the student’s transcript. Likewise, the following statement:

```
#IF !HASCOURSE("MAT-240", "MAT-244", "MAT-262", "MAT-263", )
```

will evaluate as true only if none of the listed courses appear on the student’s transcript.

For another example, the following code block would print a message only if none of a series of courses were taken:

```
#IF !VALIDCOURSE("ENG-355", "ENG-376", "ENG-381")
#PRINT "You have not started a concentration."
#ENDIF
```
CHAPTER 10: WRITING MAJOR FILES

Practical Preprocessor Applications

Using #ELSE statements
#ELSE statements are used in conjunctions with #IF statements to tell the preprocessor what to do if an #IF expression evaluates as false. An #IF block using an #ELSE statement has the following structure:

```
#IF <expression>
    Action to take if expression is true...
#ELSE
    Action to take if expression is false...
#ENDIF
```

A typical #IF/#ELSE block might read:

```
#IF SATMATH < 399
    #PRINT "Remedial Math Required"
    (MATH-098)                        MATH-098
    {Freshman Math}                  MATH-101
#ELSE
    {Freshman Math}                  MATH-101
#ENDIF
```

The above #IF/#ELSE block would cause the additional requirement for remedial math to be included in the major file, and the explanatory note to appear on the audit report, if a student's SAT Math score was less than or equal to 399. Otherwise, only the Freshman Math course requirement would be included.

### Practical Preprocessor Applications

Many potential applications of preprocessor statements are obvious from the examples given above. Other important potential applications, however, are not so obvious. Some of these applications are described below.

**Avoiding Multiple File Inclusion**
When using Academic Audit's combination major function, (See Combining Majors, page 3-2.) a common pitfall is multiple file inclusion. This occurs when the same file is included (using an #INCLUDE statement) in the two or more files that are combined. To illustrate, consider two major files, COMPSCL.MAJ, and APPMATH.MAJ, which are going to be used in combination with each other for a degree audit. Prior to running the audit, Academic Audit joins the two files together into one.

Suppose both files contain the line:

```
#INCLUDE "GENRD.MJT"
```
CHAPTER 10: WRITING MAJOR FILES

Practical Preprocessor Applications

This would cause the contents of GENED.MJI to be included twice in the combination file, which means the student would have to satisfy each requirement in GENED.MJI twice in order to pass the audit. Obviously, this behavior is undesirable.

To prevent a file called GENED_MJI from being included more than once in a combination major file, place these lines at the beginning of the file:

```c
#define !DEFINED(GENED_MJI) ; remember, can't use "." in macros
#define GENED_MJI
```

and put the corresponding #ENDIF statement at the end of the file, after all of the other contents:

```c
#undef GENED_MJI
```

This way, the entire contents of GENED.MJI are contained within an #IF block, and are therefore only included if the macro GENED_MJI has not been defined (as an empty string or otherwise.) The first time Academic Audit encounters #INCLUDE "GENED.MJI", it includes all of it in the combination major file, because the macro GENED_MJI has not yet been defined. However, in so doing, the #DEFINE GENED_MJI statement is included, which means that from that point onward in the file, GENED_MJI will be a defined macro. The second time Academic Audit encounters #INCLUDE "GENED.MJI", it also encounters #IF !DEFINED(GENED_MJI), which means "if" GENED_MJI is not a defined macro, include this #IF block for processing. Since GENED_MJI has now been defined, none of the #IF block (i.e. the entire GENED.MJI file) will be included for processing.

Changing Requirements Depending on Major

In most major files, an #INCLUDE statement is used to include sets of requirements which are common to a number of different majors. The most common example is placing all of a school's general education requirements, which must be satisfied regardless of what a student's major is, into a single file. This file, which might be called GENED.MJI, is then included in all of the major files with an #INCLUDE statement.

This process may not always work as smoothly as expected. For example, consider a set of general requirements which require a student to take the Computer Science course CS-100, which is an introductory look at personal computers and software applications. This requirement might be coded like this:

```
{Intro to Computers}       CS-100
```

A problem arises if, for certain majors, this requirement need not be satisfied. Perhaps Computer Science majors do not have to take this course because their introductory course is at a higher level. One solution would be to maintain two sets of general requirements in two files, GENED_1.MJI and GENED_2.MJI, and include the appropriate one in each major file. This can be unwieldy, however, and increases the risk that important changes made to one might not be made to the other.

There are several easier ways to do this with with preprocessor statements. One is through the use of a macro and an #IF block. Let's say the major file for Computer Science is called COMPSCI.MAJ. On the first line of this file, we put the line

```
#define COMP_SCI
```
which creates a macro COMP_SCI, which is defined as an empty string. Below that, we put:

```c
#include "GENED.MJI"
```

which replaces that statement with the contents of GENED.MJI in the final processing file.

Within GENED.MJI, we code the Intro to Computers requirement like this:

```c
#if !defined(comp_sci)
   {Intro to Computers} CS-100
#endif
```

If GENED.MJI has been included within COMPSCI.MAJ, the #if statement detects that COMP_SCI is a defined macro, and thus the Intro to Computers requirement is not included for processing. If GENED.MJI is included in any other major file, the Intro to Computers requirement is included, because COMP_SCI has not been defined as a macro.

Another way to achieve the same effect is to directly test the contents of the major field in the student's database record, as described earlier and illustrated here:

```c
#if major = "COMPSCI"

#else
   {Intro to Computers} CS-100
#endif
```

The above #if/#else block tells Academic Audit, "if the student's major is COMPSCI, do nothing, otherwise include the Intro to Computers requirement. This method might, however, produce unintended results if a student's declared major is not COMPSCI, but his course history is being audited against the COMPSCI major for advising purposes, perhaps in anticipation of changing majors. In this case, the Intro to Computers requirement would be erroneously left in for processing. Therefore, caution should be used when testing the value of a field in the student's database record. Is that really the factor which determines whether an #if block's statements should be included for processing? In many cases, such as this one, the use of #define and #if defined() statements are the only way to insure correct results."